Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Ingegneria elettronica, telecomunicazioni e tecnologie dell'informazione

Ciclo 31

Settore Concorsuale: 09/E3 - Elettronica

Settore Scientifico Disciplinare: ING-INF/01 - Elettronica

A Dataflow Framework For Developing Flexible Embedded Accelerators

A Computer Vision Case Study

Presentata da: Arthur Stoutchinin

Coordinatore Dottorato

Alessandra Costanzo

Supervisore

Luca Benini

Esame finale anno 2019

Arthur Stoutchinin: *A Dataflow Framework for Developing Flexible Embedded Accelerators* PhD Thesis © october 2018.

E-MAIL:
arthur.stoutchinin@gmail.com

Dedicated to my mother, my source of support, and to my kids,
Girgory and Anastasia, my source of energy and motivation .....

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABSTRACT

The focus of this dissertation is the design and the implementation of a computing platform which can accelerate data processing in the embedded computation domain. We focus on a heterogeneous computing platform, whose hardware implementation can approach the power and area efficiency of specialized designs, while remaining flexible across the application domain. In a heterogeneous platform, programmable cores are combined with the application-specific hardware elements. The programmable cores guarantee flexibility — that is, the ability to support different applications; the application-specific hardware elements provide the computational capability and efficiency needed to meet the requirements of modern applications.

In the embedded systems domain, limited area and energy budgets are key design constraints which make the design of such platforms challenging. These limitations call for innovative and effective solutions - since the last few years we have been witnessing a growth in popularity of multi-core heterogeneous shared memory clusters. An important advantage that they provide is a flexible communication infrastructure leveraging on an efficient shared memory. Using shared memory it is possible to build different communication topology for a wide variety of communication patterns. Above all, shared memory clusters are programmable and are able to meet the fast changing needs of the applications.

The multi-core architectures require parallel programming, which is widely-regarded as more challenging than sequential programming. Although shared memory parallel programs may be fairly easy to write (using OpenMP, for example), they are quite hard to optimize; providing embedded application developers with optimizing tools and programming frameworks is a challenge. The heterogeneous specialized elements make the problem even more difficult. Dataflow is a parallel computation model that relies exclusively on message passing, and that has some advantages over parallel programming tools in wide use today: simplicity, graphical representation, and determinism. Dataflow model is also a good match to streaming applications, such as audio, video and image processing, which operate on large sequences of data and are characterized by abundant parallelism and regular memory access patterns. Dataflow model of computation has gained acceptance in simulation and signal-processing communities. This thesis evaluates the applicability of the dataflow model for implementing domain-specific embedded accelerators for streaming applications. In particular, it investigates 1) how a shared memory cluster platform can support the

dataflow model of computation in a simple and efficient way; 2) how sequential reference algorithms can be transformed into a dataflow implementation; 3) how an efficient dataflow run-time can be implemented; 4) what are performance characteristics of a dataflow implementation.

This thesis describes StreamDrive, a dynamic dataflow framework that includes a dataflow cluster architecture template, a dataflow programming API, and an efficient runtime system for executing dynamic dataflow applications. Based on StreamDrive architecture template, a full Computer Vision Engine (CVE) have been implemented, targeting streaming image processing applications including Convolutional Neural Networks (CNN). It is experimentally demonstrated that sequential reference algorithms can be systematically transformed into a dataflow implementation through an incremental process, called successive refinement transformation. Using state-of-the-art image processing applications implemented in StreamDrive, the performance overhead, memory requirements, performance scaling, and tolerance to reduced off-cluster memory bandwidth of the dynamic dataflow model have been evaluated. It is shown that it is possible to support efficient dataflow processing on heterogeneous multi-core shared memory clusters with simple hardware extensions and a high-performance dynamic dataflow runtime.

Finally, a specialized tightly-coupled convolution hardware block (HWC) has been developed, optimized to efficiently execute the CNN convolutional layers. The computation of convolutional layers in neural networks must be scheduled such that their working set fits with limited local memory, while the number of data transferred from the next level in memory hierarchy is minimized. This thesis proposes a new methodology for scheduling the convolutional CNN layer computations that yields better schedules than previously published methods. The algorithm developed while designing the HWC is significant not only in the context of the CNN processing, but also in the more general context of scheduling nested loop computations using application-managed memory buffers.

This thesis is based on work done by the author within the project conducted by the ST Microelectronics and by the University of Bologna. It is a summary of the individual research done by the author while working within the team framework. Therefore, the author refers to work being done individually as "we" instead of "I" because without the dedication and contribution of the entire team, the individual research presented within this thesis would not have been possible.

My grandfather once told me that there are two kinds of people, those who do the work and those who take the credit. He told me to try to be in the first group; there was less competition there.

— Indira Gandhi

ACKNOWLEDGEMENTS

This dissertation is a result of an unusual academic path. I have started my PhD studies in 1996 in the University of Delaware, USA. I went over and completed the required course-work, passed the PhD qualifying examination, made my Doctoral proposal in 1999, and ... have left the school. This decision was motivated by several personal circumstances and I have not returned to the idea of completing my PhD degree until 15 years later, in 2015, while working at ST Microelectronics in Grenoble. However, many exceptional people that I have met during that period have greately influenced my following life and were an inspiration for my decision to give it a second try. My first thougt goes to them.

First of all I want to express my greatest gratitude to professor Guang Gao from the University of Delaware to whom I owe my entire engineering carrier. Dr. Gao has been my advisor for many years, first during my Master's thesis work in Montreal, and then in Newark, in Delaware. Thanks to professor Gao I have discovered the world of computer science, electrical enginnering, and scientific research. One thing that encouraged me to work hard is seeing professor Gao's dedication to his work and research. It was an honor to be his student - an honor that I am proudly cherishing through my entire carrier and life. Most of all, our relationship went far beyond professional. I also dedicate this dissertation to Prof. Gao's wife, Peggy, and I wish to thank Gao family for supporting me and for making their house also mine during the many years in Montreal and in Newark.

I also wish to thank my good school friends Andres Marquez, Shadi Abughazaleh, Vasco Jerenic, Jose Nelson Amaral for having been great friends during all this time. I always remember the good times we had together. I wish to thank Steve and Lilly Weissinger for their help and support during my summers at Silicon Graphics, California, while I was working on my initial research proposal. I thank those brilliant people whom I met at Silicon Graphics at that time, John Ruttenberg, Woody Lichtenstein, Suneel Jain, Jim Dehnert, and others, and who had greately influenced my formation during my first PhD period.

Going back to school after 15 years, and while keeping a day job, is not easy. It would never have happened if I had not encounter Prof.

Luca Benini, from the University of Bologna, who I sincerelly thank for having accepted to became my PhD adviser. Luca is one of the most brilliant people that I have encountered in my carrier and I am infinitely greatefull to him for his guidance and support.

I am specially greatfull to Philippe Galliard, my manager at ST Microelectronics, for supporting me in my endeavor. Without Philippe's support, this dissertation would not have happened. Many thanks go to my collegues from Grenoble and Bologna who have helped me during my work on this dissertation. I thank Mario Toma, Didier Fuin, Min Xue, Francesco Conti, and others, for many discussions and ideas that were inspiring for this work.

Finally, I wish to thank Prof. Kevin Martin from the University of Bretagne Sud and Prof. Andrea Calimera from Politecnico di Torino for serving as my external reviewers and for critically evaluating my dissertation. I have specifically appreciated Prof. Martin's comments and suggestions that improved the overall presentation of the dissertation.

Grenoble, october 2018 Arthur

INTRODUCTION

Science can amuse and fascinate us all, but it is engineering that changes the world.

— Isaac Asimov

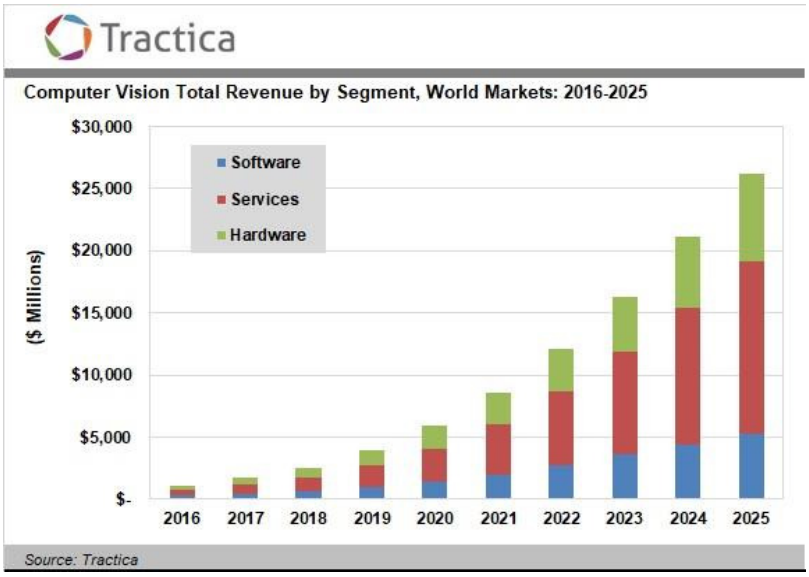
Application-specific hardware acceleration has surfaced as the prevailing approach across many application domains in embedded systems because of its performance and energy benefits compared to general-purpose solutions. Depending on the domain, accelerators often bring greater than a 10× advantage in performance, or cost, or power over a general-purpose processor. Accelerators can have macro architectures that span from fixed-function, special-purpose chips (early generations of graphics chips were of this variety) to highly programmable engines tuned to the needs of a particular domain.

When hardware accelerators of various forms are designed, there is always a trade-off between how general versus function-specific to make the architecture. While fixed-function hardware accelerators are effective, they pose many challenges. As algorithms change rapidly, hardware must be re-designed and re-verified, which is costly in terms of development cost and time-to-market. As a corollary, innovation in algorithms becomes more difficult without access to flexible hardware. Furthermore, fixed accelerators cannot be shared across applications, making them more costly in terms of silicon. Finally, from the academic viewpoint, it is difficult to formalize and apply improvements from narrowly specialized hardware to the broader field of computer architecture – limiting the intellectual impact of such work.

Thus, the flexibility of a programmable accelerator, where modifications and enhancements can be done by software changes rather then by re-spins of the hardware presents important advantages. Flexible accelerators are applicable across a variety of applications, allow bug fixes and support new product requirements over the lifetime of the accelerator without a chip re-spin. Ideally, what is required is hardware that is capable of executing compute-intensive algorithms at high performance with much lower power than programmable architectures, while remaining broadly applicable and adaptable. This thesis focuses on development of such programmable accelerator.

STREAMING ACCELERATORS

Streaming applications represent an important class of high performance computations. Defined by their regular processing of sequences of data, streaming applications appear in the context of audio, video, and image processing, digital signal processing, networking, encryption, and other areas. Streaming application properties lend themselves to very efficient hardware implementations through exploitation of parallelism. From a more technical perspective, streaming applications are amenable to an accelerator-based solution, for which a combination of parallelism, pipelining, and regularity of computation are necessary. Accelerators use parallelism to gain their performance advantage over the general-purpose processors, while the streaming computation exhibit substantial parallelism to take advantage of the accelerator. Accelerators benefit from simple (or none) address generation logic, while typical streaming applications demonstrate predictable, well-behaved streaming memory access patterns.



There is strong demand for streaming applications, such as computer vision solutions, for example. The computer vision market size is estimated to grow to many billions of dollars over the coming years [1]. Tractica forecasts that global revenue from computer vision software, hardware, and services will grow from $1.1 billion in 2016 to $26.2 billion by 2025. This prompts new research and commercial solutions to appear, specifically targeting the mobile sector.

Recently, heterogeneous multi-core platforms have emerged as the general and structured approach for designing flexible domain-specific accelerators [2]. To achieve high performance, lower cost, energy-efficiency, and required flexibility, these platforms combine

programmable processor cores with a number of application-specific hardware elements.

The application-specific hardware elements can exploit the parallelism available within target applications by executing coarse-grained functions. For a large set of streaming applications, coarse-grained processing elements (i.e., processing elements implementing coarse-grained functions) result in the best balance between efficiency and programmability [3]. A number of architectures also advocate the approach of coupling the processing cores and specialized processing elements by sharing a level of memory hierarchy [4, 5, 6, 7, 8, 9, 10, 11]. The GreenDroid architecture [12], features clusters with a single general-purpose core, which is coupled smaller conservation cores (or c-cores) [11] that communicate with the rest of the system through a first level coherent cache. Fajardo [8] proposed coupling of programmable processors and application-specific hardware units via a *buffer-integrated-cached* substrate. The EXOCHI architecture [13] features hardware accelerators modeled as coarse-grain MIMD functional units, collaborating with IA32 CPU cores by sharing of the same virtual memory space. Cong *et al.* [14] developed a heterogeneous multi-core architecture with shared-memory accelerators, which communicate by means of shared level two caches, accessible through NoC nodes. Dehyadegari [7], Conti [6], and Burgio [4] proposed architectures for interfacing RISC32 programmable cores and application-specific hardware blocks via shared scratchpad memory.

The work in this dissertation is an evolution of the fabric of tightly-coupled homogeneous clusters, called Platform 2012 built by the STMicroelectronics [15]. In the tightly-coupled cluster paradigm, each cluster is composed by a relatively small number of simple RISC cores that communicate through a fast low-latency interconnection to a shared data memory. Multiple clusters can be connected through a high-bandwidth scalable medium such as a network-on-chip. HeP2012 [16] extended the P2012 with hardware elements (HWPEs) and proposed a methodology for the semi-automatic definition and instantiation of shared-memory HWPEs from a C source. This thesis builds upon this earlier work.

PROGRAMMING STREAMING ACCELERATORS

To take advantage of the potential performance offered by a heterogeneous multi-core platform, applications need to manage multiple programmable cores, application-specific hardware elements, limited on-chip memory, and explicitly managed memory hierarchy. Among existing programming models, OpenMP has been particularly appealing for programming shared memory clusters, mainly for two rea-

sons: (1) it is based on familiar standard C language, extended with intuitive pragmas, and (2) a number of OpenMP ports for embedded systems exist, which provides guidelines and code to get started [5].

In practice, although initial parallelization of the reference code with OpenMP is simplified, tuning them for performance is hard and may require considerable development effort. There is a gap between the application specification in a standard sequential imperative language such as C and its parallel implementation that must be "filled" by the application developer.

The OpenMP does not provide sufficient abstraction for helping to achieve the right trade-off between various optimization parameters. Getting good performance on a multi-core heterogeneous platform often requires sophisticated code and data structure transformations to expose the right variety and amount of parallelism. One typically needs to perform simultaneous optimization of the algorithm, selection of data structures, task granularity, data-tiling dimensions, data prefetch distance, and loop unrolling. These parameters are often not orthogonal to each other: changing one may require changing another due to limited resources. The performance effect of varying these parameters is often non-intuitive and requires actual code development and execution time measurements to quantify. One more source of complexity arises from the fact that application-specific hardware elements often have software-managed memories, which require additional software complexity for management or tuning for the higher performance. For example, software-managed Static Random Access Memory (SRAM) requires additional work on part of the developer to manage the SRAM and to remap data structures to the address space of the local memories. The OpenMP does not provide a suitable abstraction for these details so as to insulate the developer from managing their own on-chip memory.

The OpenMP does not match well with streaming applications, and suffers from the inability to tolerate long memory latencies and waits due to synchronization events [17]. Memory barriers (such as used by the OpenMP) are one of the key sources of performance degradation in communication intensive (e.g. streaming) applications. What makes optimization even more difficult is that data transfer between the CPU and the hardware elements is often a major performance overhead, and so needs to be tuned and optimized. To reduce this overhead, hardware elements granularity of computation needs to be of relatively large size, and data sent to the hardware element in a decoupled, pipelined fashion, without stalling on the return of the results. Therefore, the application developer must overlap the computation with data transfer in a pipelined fashion, which often requires major changes to the program structure surrounding those components to be offloaded.

In OpenMP, hardware units are working under the control of the CPU, which is running the bulk of the application code, offloading the computations that can be accelerated onto the hardware elements. The nature of hardware elements as separate architectural entities working in concert with the general-purpose CPU, makes the application development intrinsically more complex. An application developer must identify the portions of the computation to offload, isolate the data structures required by these portions, manage the transfer of these data structures to the hardware element memory (if the hardware elements have a separate address space from the processing cores), synchronize between the processing cores and the hardware elements, transfer the result data back to the main memory, and integrate the results into the original data structures. These offloading tasks are in addition to the original development effort required for parallelizing the application.

The lack of sufficiently high-level abstraction for pipelining, asynchronous task execution, and data communications between tasks, increases the complexity of parallel programming using the OpenMP. This is less of a constraint in general-purpose computing. In a typical embedded application scenario, however, engineering cost can quickly erode the performance benefit. Finally, practical OpenMP implementation requires sophisticated runtime system support, which typically implies important space and time overheads. The applicability of the approach is thus often limited to applications exhibiting units of work which are coarse-grained enough to amortize these overheads. While this is often the case for general-purpose systems and associated workloads, things are different when considering embedded streaming applications.

DATAFLOW COMPUTATION MODEL

The dataflow model of computation describes an algorithm as a network of communicating computational kernels, also called actors. Actors are connected by directed, loss-less, FIFO channels. This makes the flow of data explicit between actors, which are not permitted to share data in any other way than by sending each other messages, called tokens. The use of a dataflow computation model creates several opportunities for efficient implementation in embedded context:

1. The dataflow exposes the maximum degree of parallelism in a program since the data-flow model only enforces true data-dependencies.

2. The dataflow allows asynchronous data-driven execution of finer-grained tasks; fine-grain tasks have a greater potential to efficiently use the underlying hardware [18, 19]. Furthermore,

finer-grain tasks have smaller memory footprint reducing the size of required memory.

3. The dataflow can tolerate memory and synchronization latencies efficiently [17].

4. The dataflow does not require power hungry modules like out-of-order execution (since there are only true data-dependencies) and can utilize non-coherent memory hierarchies.

5. The dataflow model matches well application-specific streaming hardware elements.

Among numerous dataflow models, the statically decidable models offer predictability, strong formal properties, and are amenable to automated optimization techniques. However, for many streaming applications, it is not possible to represent all of the functionality in terms of decidable dataflow models. This is due to the increasing levels of application dynamics that must be supported, such as the need to support multiple standards, variable data rate processing, or complex forms of data dependent application behaviors. On the other hand, the dynamic dataflow models do not provide compile-time guarantees such as deadlock-free execution and must be scheduled dynamically in general, because their expressive power does not allow them to be statically analyzed. A run-time system is required to schedule the execution of actors and to manage their communications. The important dynamic dataflow model is the Dataflow Process Network (DPN) model in which the token production and consumption rates of actors can vary in ways that are not entirely predictable at compile time.

This thesis is addressing the issues related to supporting DPN execution in a heterogeneous clustered shared memory platform, such as mentioned earlier. It also addresses the DPN implementation efficiency issues related to executing streaming applications in resource- and energy-constraint embedded context. Finally, this thesis addresses the issue of transforming and optimizing existing sequential reference applications into the DPN form.

CONTRIBUTIONS OF THIS THESIS

Our ultimate goal is to provide an effective dynamic dataflow framework and to investigate its applicability in a realistic industrial environment in the embedded domain. To do it, this thesis elaborates the dataflow shared memory cluster concept to yield an approach that combines hardware and software elements into an experimental dynamic dataflow framework, called StreamDrive. The applicability of the dynamic dataflow model of computation to parallelizing

streaming applications on this platform is then experimentally investigated. In particular, the thesis addresses the following questions:

1. What architecture support is necessary for efficient execution of dataflow applications in tightly-coupled clusters with streaming hardware elements ?

2. How do we transform a sequential reference code into an optimized dynamic dataflow implementation and what is the required effort of doing so ?

3. How can we make an efficient run-time system for executing dataflow applications on top of a memory limited embedded platform ?

4. What are the performance characteristics of applications implemented within our dataflow framework?

This work is based on the Platform2012 shared memory cluster [15] extended with tightly-coupled application-specific streaming hardware elements. Since none of the existing dynamic dataflow implementations (see chapter 2) where readily available on the P2012 shared-memory cluster, the StreamDrive Application Programming Interface (API) and the runtime environment have been implemented from scratch. We have estimated that a clean-start implementation would not take considerably longer time than porting and implementing the missing functionality for an existing implementation. A clean-start implementation also removes the risks of encountering bugs in a large and unfamiliar code-base.

To answer the first question, the requirements for dataflow synchronization and communication in a shared memory platform have been thoroughly analyzed. A lock-free counter based synchronization have been proposed by Bhattacharyya *et al.* in [20] for implementing the dataflow applications in a shared memory multiprocessors. The synchronization counters can significantly reduce synchronization overhead in a dataflow execution. However, the implementation also requires polling of the shared counter locations in order to detect changes in synchronization counter value. Polling has performance penalty and, most importantly, will saturate the shared memory interconnect in the context of a tightly-coupled cluster. One possibility would be to use the interrupts for signaling when a counter value changes, but interrupts incur unacceptably high performance overhead. StreamDrive addresses the issue by implementing a lightweight *event* mechanism. Events signal the change in synchronization counter value similar to interrupts, however unlike the interrupts, they do not deviate the execution of the program and do not need a context-switch. Our processor instruction set architecture is extended with a few instructions dedicated to generating and handling events. The Direct Memory Access Unit (DMA) is also extended

with event generation functionality, so that DMA memory transfers can be handled as usual dataflow actors. In the StreamDrive cluster, the application-specific hardware elements are connected to cluster shared memory similar to programmable cores. A special Hardware Block Bridge (HBB) is inserted between the hardware elements and the shared memory, which is responsible for synchronizing the hardware elements execution via synchronization counters and events. Thus, in StreamDrive architecture all elements are able to send and receive dataflow synchronization events. In order to efficiently communicate these events to all actors, a dedicated event network connects all cluster programmable cores, the DMA, and hardware elements inside the cluster.

In StreamDrive architecture, dataflow actors communicate via the shared memory. Software implemented actors use a dataflow FIFO abstraction via the StreamDrive API to send and receive dataflow tokens. The API implementation is based on standard load/store accesses to shared memory. The application-specific hardware elements, on the other hand, are streaming and cannot generate load/store addresses to access the shared memory. Instead, these hardware elements generate streaming requests without address, while the HBB translates these requests into a sequence of load/store accesses with addresses. Altogether, the HBB and the synchronization event network enable efficient dataflow execution in a shared memory cluster.

To answer the second question, an observation has been made that turning the sequential code into a Kahn Process Network (KPN) is relatively straightforward. Moreover, the transformation can be performed incrementally, one KPN process (actor) at a time, while verifying modified application functionality and properties at each step. Further, converting the KPN network into a DPN network consists in dividing continuous KPN processes into a sequence of firings, and in associating the *firing rules* to them. This turns out to be relatively straightforward as well and also can be done incrementally, one process and one firing rule at a time. For example, dividing a KPN process into firings can be as simple as executing its *while* loop one iteration per firing. Altogether, transformation of a sequential reference algorithm into an optimized DPN implementation can proceed as a structured process consisting in a sequence of well-defined *successive refinement* steps. In order to allow such incremental approach, the StreamDrive scheduler allows simultaneous execution of both, the KPN processes and the DPN actors, inside the same application. In particular, this allows only a subset of an application to be transformed into the DPN form at any given step of the transformation process. Our successive refinement methodology significantly reduces the effort required for parallelizing the streaming applications. One additional benefit of simultaneously supporting the two execution models is that the software implemented actors and the hardware functions

can be executed as efficiently as possible. The software actor execution is most efficient under the DPN scheduler control, while the hardware implemented functions are best executed as KPN processes without the scheduler intervention.

To answer the third question, investigation of several implementation techniques for optimizing dataflow execution under a very limited shared memory has been conducted. Although zero-copy communication has been implemented earlier via sharing the dataflow buffers between the producing and the consuming actors of a dataflow channel [21, 22, 23, 24], this thesis went further by allowing sharing a buffer by multiple actors. This enables much more efficient and less memory hungry implementation of broadcasting data between several actors, and efficient implementation of data-parallel actors. Usually, these functionalities are cumbersome and costly in dataflow implementations.

One particular problem that arises when one intends to support KPN process execution is the excessive runtime stack size. Because during the KPN process suspension and resuming, the stack contents need to be saved along with other current process state, each KPN process needs to be allocated a dedicated runtime stack in memory. When multiple KPN processes exist, this uses too much memory. Alternatively, in the DPN implementation actors can share one runtime stack per processing core, making the memory requirements independent of the number of actors in the network. In StreamDrive successive refinement approach, an actor starts out as a KPN process and evolves via a sequence of transformations into a DPN actor with firing rules. The two models seem incompatible and the worst case implementation requires one runtime stack per actor for supporting the KPN execution. StreamDrive introduces a novel *stack spilling* technique that allows runtime stack sharing during the DPN execution, while supporting a dedicated stack per KPN process.

Our case-studies are focused on computer-vision applications. To answer the fourth question, a complete tightly-coupled shared memory cluster aimed at computer vision applications, the Computer Vision Engine (CVE), has been implemented. We have then used CVE to implement several real-world problems, the Oriented FAST and Rotated Brief (ORB) [25], the Face Detection (FD) [26], and several Convolutional Neural Networks (CNN). The experiments show that StreamDrive based implementation achieves high performance, small memory footprint, scales well from 1 to 16 processing cores, and is tolerant to the external memory latency.

During this research, one additional topic has also been investigated, not originally addressed by our questions leading to development of the application-specific hardware unit, the Hardware Convolution Block (HWC), optimized for neural network convolution processing. The main bottleneck for implementing efficient and cost ef-

fective embedded accelerators for state-of-the-art CNN is the memory system. The working set of a typical CNN convolution layer does not fit entirely with limited internal HWC storage, and the CNN computation volume needs to be tiled into smaller pieces. Different tilings lead to different computation schedules and memory access patterns having a dramatic effect on performance, energy and the cost of the accelerator. Inefficient memory access can potentially void accelerator advantages.

The problem of CNN computation scheduling and tiling has been addresses before but the memory usage model was not accurate for the case of application managed memory buffers, which is the most common memory architecture template for CNN accelerators [27, 28, 29, 30, 31, 32, 33, 34, 35, 36]. Existing work on scheduling CNN computations [37, 38, 39] is based on memory model originally developed for cache-based memory hierarchies [40]. Previously published models essentially search the tiling (or blocking) space of the CNN convolution loop-nest with the objective to identify the innermost loops set such that the working set of these innermost loops fits the available internal storage while the data transfers between the internal and external memories are minimized. In case of application managed buffers, published models overestimate internal storage requirements of the CNN computation and result in sub-optimal computation schedules. This thesis proposes a new analytical memory performance model to evaluate computation schedules in terms of their required footprint and memory access bandwidth. The proposed analytical model is more accurate than existing models in case of application managed buffers. This model has been validated by applying it in the context of our CVE cluster.

The CVE cluster described in this thesis enables more flexibility in comparison to competing solutions. The common shared memory enables efficient exchange of data between the HWCs and programmable cores, achieving high degree of flexibility by computing non-convolutional functions, such as pooling, normalization, etc., in software. The CVE can also efficiently support traditional computer vision algorithms, ORB, HOG, etc. because many image processing algorithms are essentially based on convolutional operations.

ORGANIZATION OF THIS THESIS

FIRST CHAPTER gives background information on the dataflow model of computation. It describes two dataflow models, the Kahn Process Networks (KPN) and the Dataflow Process Networks (DPN), discusses their implementation issues and goes over a number of decidable dataflow models derived from the DPN. The Chapter then reviewes a landscape of state-of-art im-

plementations focusing on dynamic dataflow models. It shows that existing implementations failed to fully address three important issues: (1) efficiently support the dataflow processing with built-in hardware mechanism while remaining compatible with standard and familiar development tools, (2) deliver the highly optimized implementation of dataflow applications on resource-constrained embedded platforms, and (3) propose a systematic sequential applications transformation flow that would simplify DPN applications development.

SECOND CHAPTER describes StreamDrive implementation in depth, focusing on our zero-copy and lock-free communication protocol, the dataflow shared memory cluster architecture template, the StreamDrive API, the successive refinement application transformation flow, and the efficient runtime system implementation. StreamDrive extends the standard dataflow communication protocol with the possibility for dataflow actors to share communication channels in order to reduce the application memory footprint and improve performance. StreamDrive implements hardware built-in support for the dataflow synchronization to the shared memory cluster. This Chapter also describes a dataflow hardware bridge for connecting streaming application-specific hardware elements to the shared memory cluster. Based on these hardware mechanisms and the StreamDrive runtime implementation, the Chapter describes a systematic incremental successive refinement flow for transforming a sequential reference algorithm into a highly optimized DPN parallel implementation. The future development consists in automating different steps of this process. Finally, the Second chapter also describes StreamDrive's API and several advanced runtime system implementation techniques required for a small footprint, low overhead implementation.

THIRD CHAPTER is dedicated to design of an image processing accelerator particularly focusing on Convolutional Neural Networks (CNN) acceleration. First, the Chapter reviews existing CNN hardware accelerators and expalains their inefficiencies. It then describes the implementation of our Computer Vision Engine (CVE). The main element of the CVE is a tightly-coupled convolution hardware element, the HWC block. The Chapter explains the design of the HWC based on a novel analytical memory performance model for evaluating the trade-off between the internal accelerator buffering capacity versus the off-accelerator memory bandwidth. This analytical model is tailored for implementations using application-managed buffers rather than cache based memory hierarchies and is more accurate than previously published models for such implementations.

FORTH CHAPTER is dedicated to performance evaluations. Two case studies are conducted. The first case study visits two traditional image processing applications, the Oriented FAST and Rotated BRIEF (ORB), and the Face Detection (FD). Experiments show that StreamDrive implementation of these applications achieves high performance even with very limited cluster memory, that it is scalable with increased number of processing elements, and that it is stable under varying off-cluster memory access latency. The second case-study investigates the shared memory bandwidth requirements of our HWC block. It compares this bandwidth with requirements of a state-of-the-art tightly-coupled convolution hardware block from [29]. Simulations show that the HWC achieves several times reduction in shared memory traffic. Finally, these results are put together by proposing three CVE configurations, from a low-cost implementation with lower performance to a high-performance but also higher cost one.

CONCLUDING CHAPTER summarizes the most important insights gained through this thesis work and offers future research directions.

1 DATAFLOW COMPUTATION MODEL REVIEW

All models are wrong, but some are useful.

— George Box

Unlike general-purpose systems, where the ease of programming and code portability have priority, in embedded platforms, efficiency (in terms of latency and throughput) is of primary concern, as well as power consumption, memory requirements, and code size. As we have explained in the previous chapter, the tightly-coupled shared memory clusters with multiple programmable cores and application-specific hardware units are becoming widely used in modern embedded platforms. Currently, the OpenMP [41] is the most widely used method for developing applications for this attractive architecture.

OpenMP is based on the *fork-join* model of parallel execution: an application starts with a main thread which forks off a team of threads when it encounters a parallel region of code. All threads are synchronized at the barrier of the parallel region. OpenMP primarily targeted the data parallelism that works by processing in parallel independent data elements of a single computation, and was not specifically designed to support *streaming applications*. A streaming computation works by applying different *tasks* to each element of a data stream in the program. Although starting with 3.0 version the OpenMP has been extended with the *task* construct to manage the execution of a set of tasks, there are still limitations in this approach: OpenMP fork-join model is inherently limited for expressing pipelining of tasks; and the OpenMP barrier based synchronization is inefficient in streaming applications.

Our goal is to investigate the use of the *dataflow model* for developing applications for the tightly-coupled cluster platform. The use of the dataflow model of computation is motivated by the following observations:

- The dataflow computation model matches well streaming applications; it provides a natural and intuitive representation of streams, therefore having a positive effect on readability, robustness, and programmer productivity.
- The dataflow computation model matches well specialized streaming hardware elements; they usually communicate with the system via data queues and synchronize on the availability of data in the queues.

- The dataflow computation model simplifies parallelization of sequential reference algorithms; there is no need to think in parallel, no critical sections, mutexes, race conditions, etc.
- The dataflow communication buffers reduce the negative impact of limited shared memory in two complementary ways: (1) they naturally hide memory latency; and (2) they favor shared memory communications, bypassing off-cluster memory.

The advantages of the dataflow model for application developers are related to the ability of expressing the natural parallelism of an algorithm without complex synchronization mechanisms. This is made possible by representing the computation as a network of processing blocks that only communicate through communication channels. This removes the potential concurrency issues that could arise when the application must explicitly manage the synchronization between parallel computations [42, 43].

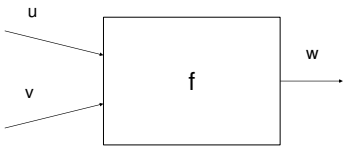
In this chapter, we will review two basic dataflow models, the *Kahn Process Networks* (KPN) and the *Dataflow Process Networks* (DPN), followed by a brief overview of a plethora of more restricted dataflow models derived from these two. We will then discuss the ways in which these models have been used in the past, specifically in the embedded context, and explain the associated issues.

1.1 KAHN PROCESS NETWORKS (KPN)

The KPN model of computation [44] was defined by Gilles Kahn as a network of autonomous, concurrently executing processes that communicate point-to-point via unbounded FIFO channels. It is a natural model for describing streaming applications where infinite streams of data are incrementally transformed by processes executing in sequence or in parallel. Formally,

Definition 1. A KPN is a directed graph $G = (N, C)$, where:

- N is a set of nodes, representing computational processes. Each node is connected to n_{IN} input channels and n_{out} output channels.
- C is a set of unidirectional channels representing infinite data FIFO buffers. The processes communicate with each other via these FIFO buffers by sending and receiving data elements, called tokens. Each FIFO is connected to a source process (which provides tokens to the FIFO) and a destination process (which consumes tokens from the FIFO). A FIFO can possess some initial number of data tokens stored in it at the beginning of the application.




```
process f (in int u, in int v, out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```

(a) A Simple Kahn Process **(b)** Kahn Pseudo code

Figure 1.1: A Kahn Process Network Example

Figure 1.1 taken from original Kahn paper [44] shows an example of a simple KPN process. The process in the Figure 1.1(a) has two input channels u and v, and one output channel w. The process alternately reads integer values from the two input channels, prints the value, and writes it to the output channel. Kahn proposed a simple language for specifying the KPN. The Figure 1.1(b) shows the Kahn pseudo-code for describing the process. In particular, the process interface includes declaration of the FIFO channels and their data type, while the process function contains the $wait()$ method returning a data token from an input channel (blocking on an empty channel), and the non-blocking $send()$ method for writing a data token to the output channel.

Each KPN process runs asynchronously and independently of other processes, and its state is inaccessible to other processes. Communication between processes is possible only by sending and receiving data tokens over channels. A process may have multiple input and output channels, but each channel is connected to exactly two processes – the sender on the one end of the channel and the receiver on the other end. In theory, it is always possible to send a token because channels have infinite capacities. A process may at any time attempt to receive a token from any of its input channels, but if the channel is empty, the process becomes blocked until the token arrives.

The restrictions which KPN places on processes may be summarized as follows:

- A process may not access the state (code, data, program counter, etc.) of another process.
- The receive operation always blocks on an empty channel, and a blocked receive cannot be interrupted.
- A process is not allowed to test for the presence of data tokens on a channel.

These restrictions have one very important consequence - the KPN behavior is *deterministic*: the history of tokens in channels is indepen-

dent of the order in which processes are executed. In other words, given the same input, a KPN will produce the same sequence of tokens on all channels during every run. This determinism is very important for the parallel application development. More generally, the KPN model benefits application development in several ways:

- *Sequential coding of individual components*. KPN processes are written in the usual sequential manner where synchronization is implicit in communication primitives (data send and receive). Developers can thus reuse their existing development environment, tools, and expertise, and need not worry about orchestrating the concurrent execution.
- *Composability*. Determinism guarantees that connecting the outputs of processes $f(x)$ and $g(x)$ to the inputs of a process $h(x, y)$ will result in $h(f(x), g(x))$. Thus, processes can be developed and tested individually, and later assembled into more complex programs.
- *Transparent parallelization*. The actual mechanisms for achieving parallelization are hidden from developers. Components are written in a sequential way and can run on multi-core platforms under a run-time system control.
- *Reliable reproduction of program faults*. An otherwise notoriously difficult problem with concurrent systems, is resolved because the KPN has deterministic behavior.

These benefits make KPN computation model very attractive for its simplicity and relatively low effort required for developing a KPN application.

1.2 DATAFLOW PROCESS NETWORKS (DPN)

The Dataflow Process Networks (DPN) are a special case of KPNs, where process execution (processes are called *actors* in DPN), instead of being continuous, is composed of a sequence of "atomic" firings [45]. A firing is "atomic" because an actor cannot be interrupted during a firing. Activation of actor firings is controlled by a set of *firing rules*. In each firing, the actor will read a specific number of data tokens from the input channels, perform computations and write a specific number of tokens to the outgoing channels. The firing rules specify the number of tokens that have to be available on the input ports to fire the actor. An actor can only fire if all the input tokens are available on the channels.

Definition 2. A DPN is a directed graph $G = (N, C)$, where:

- N is a set of nodes, called actors, representing computational processes. Each actor is connected to n_{IN} input channels and n_{out} output channels. Each actor is a tuple (A, F, R) with:
  - A is the actor function.
  - F is a set of firing rules. A firing rule is a condition which, when satisfied, enables the execution (firing) of this actor function.
  - R is a set of rates. A rate is the number of tokens consumed at an input channel or produced at an output channel corresponding to a specific firing rule. Each firing rule has $n_{in} + n_{out}$ corresponding rates: one for each communication channel.
- C is a set of channels representing infinite data FIFO buffers. These FIFOs are the only channels allowed to perform data communications between actors. Each FIFO is connected to a source actor (which provides tokens to the FIFO) and a destination actor (which consumes tokens from the FIFO). A FIFO can possess some initial number of data tokens stored in it at the beginning of the application.

DPNs are more difficult to develop than KPNs; it involves dividing actors into firings and specifying the firing rules. Practical DPN implementations can create the *artificial deadlock* because in practice the communication buffers are not infinite. The DPN dataflow model is dynamic and cannot be statically analyzed, i.e. it is impossible to determine whether the algorithm will deadlock and how much buffering is necessary for the algorithm to execute without a deadlock. The DPN model may also be costly to implement because the DPN scheduling is dependent on the runtime data of the actors.

Numerous dataflow models placing restrictions on the DPN have evolved over the past three decades aiming to balance conflicting concerns of expressiveness, analyzability, and implementability [46]. These models are called static or decidable because they have predictable behavior making them analyzable and easier to implement efficiently. The statically analyzable dataflow models can also be scheduled at compile time. However, they offer a reduced expressiveness and lower flexibility for application developers, when compared with Turing-complete DPN model. Below we quickly overview a few of the most important decidable dataflow models that have been used in domain-specific embedded computing; for a complete review of static dataflow models the reader is referred to [47].

1.2.1 The Synchronous Dataflow (SDF) Model

The most widely used decidable dataflow model is the Synchronous Dataflow (SDF) model introduced by Lee and Meserschmitt [48]. An SDF model imposes restrictions on dataflow actors firing rules and token rates:

- There is only one firing rule per actor.
- Rates for all channels are fixed and constant for the entire execution.

A major advantage of SDF is that it is predictable and analyzable at compile-time. If it exists, a bounded schedule can be found statically. Such a schedule ensures that each actor is eventually fired (*liveness*) and that the dataflow graph returns to its initial state after a certain sequence of firings (*boundedness* of the FIFOs). Such sequence is called an *iteration* and can be repeatedly executed. Useful features of SDF include formal validation of deadlock-free operation and bounded memory requirements; support for efficient static scheduling; and memory size optimization [49]. SDFs can be very efficiently implemented and have been widely accepted as computation model in the domain of digital signal processing. The important drawback of the model, which prevents its wider use, is its lack of expressive power; the SDF cannot express dynamic algorithms where the processing conditionally depends on input data or the results of intermediate computations.

Over the years, many extensions to SDF model have been proposed to extend its expressive power, while maintaining its compile-time predictability as much as possible.

The Cyclo Static Dataflow (CSDF) model [50] extends SDF actors allowing the number of tokens produced and consumed in each firing to vary cyclically. This variation is modeled with a state in the actor, which returns to its initial value after a defined number of firing. However, the CSDF model only improves SDF compactness. An application that can be described in CSDF can also be described in SDF; generally, the CSDF description requires fewer actors than in SDF. A transformation has even been proposed in [51] to transform a CSDF graph into an SDF graph with same properties. All techniques used to analyze a SDF graph can be applied to CSDF after transformation. Similarly, the Affine Dataflow (ADF) model [52] is another extension of SDF model with the same objective: improving model compactness. While CSDF has an infinite sequence of rates, the ADF replaces the unique rate of SDF with an initial sequence of firing rates followed by an infinitely repeated sequence of firing rates. These two concatenated sequences are called the ultimately periodic sequence. A transformation also exists to convert an ADF graph into an equivalent SDF graph.

The Heterochronous Dataflow (HDF) model and the Scenario-Aware Dataflow (SADF) model are limited to a range of applications that follow a sequence of fairly static scenarios. The idea of the HDF model [53] is to allow dynamic change in consumption and production rates of the actors and represent it via the *Finite-State-Machine* (FSM). All FSMs in the HDF can change state only once the dataflow graph has executed a full iteration. After the iteration is finished the HDF actors are free to change their state and their token consumption and production rates. For the duration of next iteration the rates stay unmodified and the HDF executes an SDF schedule. The SADF model [54] views applications as collections of different SDF graphs. SADF is able to perform some worst-case and stochastic analyses, and to provide implementation with limited run-time overhead, while relaxing some of the limitations of the SDF.

Buck's Boolean Dataflow (BDF) model [55] extends the SDF model with token production/consumption rates that depend on an input control channel that itself consumes one token per firing. Basically, the number of tokens consumed or produced by a given data channel can be controlled by its associated control channel, which can vary the token rate for the data channel. The fundamental dynamic actors of the BDF model are the *Switch* and *Select* that simply choose one of its two inputs or outputs according to the control token. Scheduling techniques for BDF graphs attempt to build *quasi-static schedules* derived at compile-time that reduces the complexity of run-time scheduling involved. The Integer Controlled Dataflow (IDF) model [56] used integer control variables instead of boolean values. The BDF and IDF models has been proven Turing-complete but they imply a very restrictive coding style that is not useful for a practical application.

1.2.2 Parameterized Dataflow Models

Parameterized dataflow models are *meta-models*: they are applied to any underlying synchronous dataflow model that has a well-defined notion of dataflow graph iteration. Parameterized models extend the targeted model semantics by adding dynamically reconfigurable hierarchical actors and achieve increased expressiveness due to parameters modifiable at runtime. A reconfiguration occurs when values are dynamically assigned to the parameters of a reconfigurable actor, causing changes in the actor computation and in its production and consumption token rates. As presented in [57], reconfigurations can only occur at certain points, namely *quiescent points*, during the execution of a graph in order to ensure the runtime integrity of the application. The Parameterized Synchronous Dataflow (PSDF) model introduced by Bhattacharya and Bhattacharyya in [58] can be applied to SDF [58] or to CSDF [59]. The Schedulable Parametric Dataflow (SPDF) model [60] is designed to avoid some of the PSDF

model restrictions. The Parameterized and Interfaced Synchronous Dataflow (PiSDF) model [61] is another extension of the SDF model. In addition to the SDF semantics, the PiSDF model contains a set of parameters and parameter dependencies that can be used to change the production and consumption token rates of actors. The PiSDF improves parameterization compared to PSDF by introducing an explicit parameter dependency tree and by enhancing graph composition. This allows to overcome some restrictions on dataflow scheduling and analysis imposed by the PSDF structure. The Boolean Parametric Dataflow (BPDF) model [62] combines integer parameters to express dynamic rates, and boolean parameters to express the activation and deactivation of communication channels. Application dynamism is provided by integer parameters which can change at each dataflow graph iteration and boolean parameters which can change within the iteration. Parameterized models rely on *quasi-static schedules* for their execution where part of the scheduling decisions are defined statically but also contain parameterized parts that are resolved at runtime. Nevertheless, in practice parameterized models' coding style is quite restrictive, while implementation efficiency was not clearly demonstrated for modern streaming applications.

1.2.3 Structured Dynamic Dataflow Models

Alternatively, some degree of analyzability and a more efficient implementation of the dynamic dataflow model can be achieved by structuring dataflow actor semantics.

Plishker *et al.* [63] proposed the Enable-Invoke Dataflow (EIDF). In the EIDF, an actor has a set of valid modes in which it can execute. The actor specification is divided into separate *enable* and *invoke* functions. The *enable* is designed to be used as a "hook" for the dynamic scheduler to rapidly query actors at run-time, and check whether or not they are executable. The *invoke* function implements actor functionality and can generally change the mode of the actor for the next invocation. The EIDF subset, the Core Functional Dataflow (CFDF) restricts actor execution to proceed deterministically to a single "next mode" of execution. The EIDF and CFDF were used for prototyping and simulation of dataflow applications. Plishker *et al.* [64] have presented an analysis method that can exploit the CFDF to improve the scheduler.

Similarly, Kienhuis and Deprettre proposed the Stream Based Function (SBF) model for streaming applications [65]. An SBF is composed of a set of functions, called function repertoire, a transition and selection function, called controller, and a combined function and data state, called private memory. The controller selects a function from the function repertoire that is associated with a current function state, and makes a transition to the next. The SBF model is equivalent to

a deterministic DPN model but differs from the DPN model in that the functions themselves check for the availability of data whereas in the DPN model, data is available when a function is enabled. As a consequence, an SBF could be closer to a possible hardware implementation.

The structured dynamic dataflow models focus on defining a specific execution semantics which enables some degree of compile time analysis. They do not address the questions of simplifying dynamic dataflow applications development, and of efficient implementation in a resource constrained embedded environment.

1.3 LANDSCAPE OF DATAFLOW IMPLEMENTATIONS

Due to its elegance and simplicity, the dataflow model of computation has been the subject of many research efforts. Since the early 1970s, a number of computer prototypes have been built and evaluated based on the dataflow model of computation. The representative dataflow architectures were the Manchester Dataflow machine [66], the MIT Tagged-token Dataflow architecture [67], the SIGMA-1 [68], the Monsoon dataflow processor [69], and others. Several dataflow embedded systems have also been designed such as the Hughes Dataflow Multiprocessor (HDFM) [70], or the AT&T Enhanced Modular Signal Processor [71]. These systems implemented elaborate hardware to execute dynamic scheduler and employed expensive communication networks to route data tokens. These early dataflow computers have failed to deliver the promised performance mainly due to following limitations: (1) too fine-grained (instruction level) synchronization, (2) difficulty in exploiting memory hierarchies, and (3) the inefficient use of the pipeline.

With the proliferation of multiprocessor computing platforms, the renewed interest has emerged to the dataflow computation model, particularly in the embedded systems context. Very broadly, the existing dataflow frameworks fall into 3 categories: (1) domain-specific platforms with specialized languages and tools; (2) model based frameworks for hardware/software codesign; and (3) API based frameworks provided in the form of runtime libraries.

The domain-specific platforms and the model based frameworks rely on automated analysis and generation tools. However, usually efficient implementations are only possible for decidable dataflow models, when the automated analysis is possible. On the other hand, these frameworks require a complete rewrite of the original reference application with a new language. Embedded system developers have been familiar with sequential programming like C for a long time. In fact, around 85% of embedded system developers still use

C/C++ [72]. Therefore, apart from a very specialized signal processing domain, no new parallel programming models/languages have been widely adopted in embedded platforms so far.

The API approaches are mostly developed for existing fixed platform architectures, some including specialized application-specific hardware units. The API based approaches do not require the developer to heavily modify the original source code. This brings more efficiency in terms of parallelization effort. These approaches usually support expressive models of computation, such as KPN or DPN but have not yet been able to close the gap between specification and implementation so as to achieve the computational performance and the energy efficiency of handcrafted solutions.

Overall, apart from a few specialized application domains, the dataflow model has not been widely adopted by industry. There are three main reasons for this:

- Motivated by necessity to amortize development cost over a large number of units, and by intensified time-to-market constraints, the IC and system companies are pushing toward platform-based designs, where new applications can be developed much more efficiently. The existing platforms lack necessary built-in hardware support for efficient dataflow execution.

- The main premise of existing dataflow tools, that a dataflow application can be specified at a high abstraction level and automatically transformed into an efficient implementation, has not been fulfilled. While statically decidable dataflow models do not allow to represent all of the required functionality for many streaming applications, improved expressive power of dynamic dataflow models results in problems with unbounded buffers and runtime efficiency. Thus, the main challenge that dataflow computation model has to face is the demonstration of efficient implementations that can achieve functionality and performance constraints imposed by modern applications.

- Dataflow computing is associated with reliance on development frameworks which are hard or inefficient to use for many practical applications. Reference implementations are typically developed by research teams and specified as sequential programs using imperative programming languages such as C/C++ or Matlab. Transforming a sequential reference algorithm into a dataflow representation is a complex, manual, time-consuming process. In particular, the adoption of the dynamic dataflow has been hampered by the need to start from scratch with all (software and hardware) components of computing. A paper by Denning and Dennis [73] brings forth many of the issues related to the canonical parallel processing model and dataflow computing.

Framework	Model	Target	Programming
RAW Machine	SDF	Streaming Applications	StreamIT
Imagine	Streaming	Streaming Applications	KernelC+StreamC
Ptholemy	SDF,CSDF,HDF, PSDF,KPN,BDF, DPN,SDF	Simulation and design	Specialized Language
Peace	SDF	HW/SW Codesign	Specialized Language
DIF	BDF,PSDF,EIDF CSDF	DSP	DIF Language
TDIF	CFDF	GPU	DIF Language
Daedalus	PPN	HW/SW Codesign	C+coordination
Compsoc	SDF,CSDF,KPN	HW/SW Codesign	C+coordination
Koski	KPN	HW/SW Codesign	UML
PREESM	PiSDF	TI Keystone	C+coordination
Shim	Rendez-vous KPN	Multi-core	C Extension
DOL	KPN	Embedded	C API
RVC	SDF,CSDF,DPN	Video Coding	CAL
OpenDF	SDF,CSDF,DPN	General	CAL

Table 1.1: Selected related work summary

Below, we examine a few selected state-of-the-art dataflow implementations from three perspectives:

- Expressiveness of the dataflow model versus the implementation efficiency.
- Ease of parallelizing a sequential C reference implementation so that the parallel efficiency of the target hardware platform can be exploited.
- Integration of specialized and application-specific hardware units with the hardware platform.

Table 1.1 summarizes the dataflow implementations reviewed in this chapter. In the table, the first column lists the dataflow framework; the second column in the table contains the dataflow model used by the framework; the third column specifies the target architecture platform; and the last column lists the dataflow description language when possible. Three types of dataflow specification exist: (1) a specialized language (eg. Ptolemy, StreamIT), (2) a standard language (C, C++, etc.) extended with a dataflow API, and (3) a combination of a standard language (C, C++, etc.) for actor description along with a *coordination language* for specifying the dataflow network.

1.3.1 Domain-specific Architectures

The MIT Reconfigurable Architecture Workstation (Raw) [74], and Stanford Imagine [75] were the two early foundational works in the area of stream processing. The stream model is derived from the SDF dataflow model of computation. In addition to SDF expressiveness problems, non-linear communication patterns are difficult to implement efficiently with streaming architectures because of linear stream abstraction. In order to overcome this limitation, the Raw programming language, StreamIt [76], introduced the notion of *teleport messages* [77]. Teleport messages allow one actor to sporadically send a message to another; that is, rather than sending a message on every firing, only some firings send messages. As another example, Imagine implemented *conditional streams*, accessed conditionally based on the condition codes (CC) [78]. Conditional streams enable implementation in presence of a data-dependent conditions.

More examples of streaming architectures include the Reconfigurable Streaming Vector Processor (RSVP) [79], which exposes streams in a core's ISA to communicate to reconfigurable hardware; the Triggered instructions [80], featuring some streaming memory capability to feed its dataflow fabric; or the Stream-Dataflow [81], a reconfigurable dataflow architecture that uses streams as underlying communication abstraction.

Several other domain-specific accelerators use the dataflow computation model, such as Eyeriss [82, 32], a domain-specific accelerator for convolutional neural networks, or the Kalray MPPA multiprocessor platform programmed using the dataflow ΣC language [83] that implements the CSDF model.

Overall, such domain-specific platforms failed to attract the embedded community because (1) they suffer from reduced expressiveness and flexibility, and (2) they rely on specialized languages and development tools unfamiliar to the vast majority of developers in the field.

1.3.2 Model Driven Frameworks

The model-driven dataflow frameworks are designed to support efficient design space exploration - a systematic methodology for selecting an embedded system implementation from a set of alternatives. In these tools, system designers are able to develop complete functional applications formally specified as a dataflow model and perform automated performance analysis, simulation, synthesis and verification of the implementation. Design space exploration is performed by iteratively analyzing and optimizing the application along with the underlying hardware and software architecture. In embedded domain, particularly popular are decidable dataflow models be-

cause the reduced runtime overhead and analyzability of compile time scheduling is considered a big advantage.

The Ptolemy (and its successor Ptolemy II) environment [84, 85, 86] is developed at the University of California at Berkeley. Ptolemy is targeted towards hardware/software codesign and in particular towards the system synthesis and verification. Ptolemy supports a wide collection of computation models including the majority of dataflow models. Ptolemy evolved to Ptolemy II, which proposes a *modal* approach where finite state machines (FSM) are combined with a dataflow model in a hierarchical fashion. The modal approach overcomes certain limitations of decidable dataflow models in expressive power, while it can be refined to final implementation since both FSM and decidable dataflow models provide methods of system synthesis. In Ptolemy, not all models can be used for system implementation. In particular, synthesis from decidable dataflow models has been extensively researched, while other models serve for simulation purpose only.

PeaCE (Ptolemy extension as a Codesign Environment) [87] is an extension to Ptolemy II that provides a hardware software co-design framework. PeaCE uses extended SDF and finite-state machines to model data flow and control flow of multimedia applications. The platform architecture consists of a number of processors and synthesizable IP cores, which are connected through a communication infrastructure. The two step design space exploration is used: (1) selection of processing elements and mapping of application tasks on these processing elements, and (2) exploration of the communication architecture such as bus and memory allocation. However, the framework is still limited in applicability by its SDF semantics.

The Dataflow Interchange Format (DIF) [88], developed at the University of Maryland, is a textual language for specifying dataflow models for DSP systems. DIF captures essential modeling information that is required in dataflow based analyses and optimization, such as algorithms for consistency analysis, scheduling, memory management, etc. DIF provides an extensive repository of models, analyses, and transformations, for a number of dataflow models including dynamic models such as BDF, the PSDF, the EIDF, and the CFDF. DIF itself does not generate implementation of dataflow descriptions but can be used by different DSP tools. For example, the DIF-to-C tool [89] allows generation of C code for the target DSP platform from a SDF dataflow specification. For the final implementation DIF-to-C relies on C compiler and optimized libraries provided by the DSP processor vendor. Shen *et al.* proposed the Targeted Dataflow Interchange Format (TDIF) [90] extending the DIF with the CFDF software synthesis. This implementation targeted CUDA code generation for the NVIDIA GPUs, and has not demonstrated its efficiency in a more constrained embedded platforms context.

Nikolov *et al.* [91, 92] presented Daedalus framework for architectural exploration, synthesis, and prototyping multicore platforms. The Daedalus combines KPNgen [93], Seasame [94], and ESPAM [95, 91] tools. Applications are specified as KPN networks, which are either derived manually or automatically using the KPNgen. However, automatic generation of KPNgen networks from application's sequential C code is only possible if the application is specified as *Static Affine Nested Loop Program* (SANLP). A SANLP is a nested loop program in which loop bounds, conditions and variable index expressions are affine expressions in the iterators of enclosing loops and static parameters [93, 96]. Because many applications are not static, i.e. include nested loops which can contain `if-then-else` constructs with no restrictions on the condition, loops with no condition on the bounds, `while` statements other than `while(1)`, dynamic parameters, etc., the KPNgen usability remains limited. For the resulting process networks it is possible to compute static schedule. The input KPN network is fed to Sesame modeling and simulation tool [94, 97] to perform architectural design space exploration for mapping and scheduling the KPN processes. Daedalus uses a heterogeneous platform (created from a library of components) where the processing elements communicate via distributed memories. A set of KPNs and platform configurations from Seasame is passed to ESPAM for prototyping on FPGA. ESPAM generates C code for KPN software processes and synthesizable VHDL for platform hardware components from RTL component library. As explained later in this chapter, although it is relatively straightforward to manually derive a KPN network from a sequential code, KPN execution in software suffers from high performance overhead, making KPN model less suitable for the embedded implementation than the DPN model.

The *Composable and Predictable Multi-Processor System on Chip* (CompSOC) [98] is another design flow that supports a range of execution models, including the KPN model. The environment includes a complete multicore architecture, platform support libraries, libraries for synchronization and communication, and tools for formal verification. The main focus of CompSOC is to provide a design flow that supports simultaneous execution of multiple independent applications. In CompSOC, each application is given its own reconfigurable virtual platform. The CompSOC employs a two-level scheduling along with a resource sharing model in order to eliminate interference between different applications. At a single application level, CompSOC relies on SDF3 toolset [99] for mapping and scheduling a dataflow application on the hardware platform. However, while the decidable CSDF applications can be automatically mapped, verified, and executed on the CompSOC platform, mapping and analysis of KPN applications is not automated. The DPN model is not supported in CompSOC environment.

Similarly, Koski [100] framework provides environment for modeling, automated design-space exploration, synthesis, and FPGA prototyping of selected design. The input specification is given as KPN modeled in UML. The target architecture consists of synthesizable communication and processing resources, application software, and platform-dependent and platform-independent software.

The *Parallel and Real-time Embedded Executives Scheduling Method* (PREESM) [101] is a framework used to prototype and generate code for applications specified in PiSDF dataflow model, and targets heterogeneous multi-core embedded platforms. PREESM works with three inputs: a PiSDF dataflow graph defining the application; a *System-Level Architecture Model* (S-LAM) describing the target architecture; and a scenario including a set of parameters and constraints to link both of them. S-LAM supports the description of parallel architectures as a set of heterogeneous processing elements transmitting data through a set of communication nodes and data links. PREESM automatically schedules, maps and simulates the execution of the application and generates a compilable C/C++ code for the target architecture. PREESM supports and has been used to generate code for the x86 multiprocessors, the Texas Instruments Keystone DSPs, the Kalray MPPA many-core, Xilinx Zynq SoC, and the ARM Big.LITTLE & Multi-core ARM. The runtime responsible for managing runtime reconfigurations of the PiSDF dataflow graph is called SPIDER (*Synchronous Parameterized and Interfaced Dataflow Embedded Runtime*) [102]. SPIDER exploits the trade-off between dynamicity and predictability of the PiSDF model to verify application properties or to perform optimization at runtime. As with other parametric dataflow models, in the PREESM/SPIDER framework the token production and consumption rates cannot change arbitrarily, the existence of a dataflow graph iteration must be guaranteed. In practice, this leads to a restrictive coding style which may be difficult to use, while the implementation efficiency has not yet been clearly demonstrated.

Pursuing compile time analyzability and schedulability, the model based approaches often resort to dataflow models with restricted expressiveness and flexibility. Moreover, they rely on unfamiliar languages and specialized development tools.

1.3.3 C Based Frameworks

An alternative to frameworks based on decidable dataflow models with limited expressiveness is to integrate dynamic dataflow programming structures into familiar languages, using a lightweight API with an associated runtime environment. Such approach reduces the software development impact by allowing the tools for creating and debugging dataflow applications be basically the same as those for standard software: compilers, assemblers, debuggers, and cross-

compilers. The vast majority of such APIs implement the KPN model because of the low effort required to transform a sequential reference algorithm to the KPN form.

Many such implementations target large computing systems and rely on off-the-shelf OSes. For example, the QUeing And Runtime for Kernels (QUARK) [103], TIDeFlow [104], and OpenStream [105], have been developed in the context of the High-Performance Computing (HPC) applications. YAPI [106] and Nornir [107] support the KPN execution model on workstation computers. XKaapi [108, 109] is a runtime system for scheduling dataflow programs on multi-processors and clusters of multi-processors. Work in [110] proposed a design flow allowing implementation of dataflow applications on a multi-GPU computer cluster. The Intel Concurrent Collections [111] has been used for developing applications on large scale heterogeneous platforms that include general-purpose CPUs, GPUs, custom processors, and FPGAs. These implementations come with heavy performance and memory footprint overheads. This is an acceptable choice for running applications in big-size computers. In the embedded domain we need a lightweight approach: the small memory and the high performance requirements preclude using the full OS, a kernel-level scheduler, and dynamic data structures.

The *Software/Hardware Integration Medium* (SHIM) [112] was initially developed as a design space exploration dataflow model for specifying, validating, and synthesizing heterogeneous embedded systems. It has later been turned into a language development effort centered around scheduling and static analysis for programming shared memory multiprocessors [113, 114]. Shim relies on restricting the KPN semantics to help both programming and automated program analysis. SHIM implements a KPN restricted to support synchronous (*rendezvous*) communication. This choice eases scheduling, and guarantees that KPN programs are always executable in finite space because synchronous communication does not need buffering. The Tiny-Shim language is based on C (but is not a C subset) augmented with few constructs for concurrency, communication, and exceptions. SHIM imposes many syntactic restrictions on the input language which makes porting existing reference applications difficult. While it has been able to devise effective mechanisms for static scheduling and analysis (e.g., deadlock detection), the implementation relies on costly standard runtime support such as POSIX Pthreads library.

The *Distributed Operation Layer* (DOL) [115, 116, 23] is also a design flow framework based on the KPN model of computation and targeted at real-time multimedia and signal processing applications. The DOL design flow follows the Y-chart approach [117] in which the application specification is platform-independent and needs to be explicitly mapped on a target architecture. DOL supports the Cell Broadband Engine [118], the tile-based MpSoC Atmel Diopsis

940 [119], the MPARM platform [120], and the Intel SCC many-core architecture [121]. In DOL, KPN processes are described in C/C++ based on a simple API, while the network is described using the XML. Similar to model based frameworks, an application specified in DOL cannot be directly executed by simply compiling the provided source code of KPN processes. A synthesis step is required that generates the "glue code" implementing the processes and channels, the bootstrapping and the scheduling of the application. Specifically, synthesis is done first for a standard PC/workstation to support the functional verification and debugging of the application, and second for the target platform. DOL is different from other design flow environments in that it embeds an analytic worst-/best-case performance analysis method that targets real-time signal processing applications.

Striving to reduce the runtime overhead associated with KPN execution, DOL implements the KPN model using cooperative protothreads [122]. While cooperative scheduling eliminates context-switching overhead and simplifies the runtime stack handling, the protothreads impose a number of important language restrictions leading to additional performance penalty and to additional difficulties parallelizing the sequential reference code.

One example of a dynamic dataflow API is the *lightweight dataflow* (LWDF) [123] that implements the CFDF model. The CFDF corresponds to a deterministic subset of the DPN model with well-structured actor description that simplifies some implementation issues. Similar to most dataflow frameworks, the LWDF focuses on analyzability and compile time techniques rather than on ease and efficiency of the dataflow applications implementation.

1.3.4 CAL Dataflow Language

An important family of frameworks which exist for dataflow based programming is based on CAL programming language [124, 125] developed as part of the Ptolemy project at the University of California at Berkeley. While previously mentioned dataflow frameworks emphasize classes of applications that exhibit sufficiently regular behavior to permit compile-time analysis and scheduling, CAL is designed to also support the DPN model. In CAL based frameworks, the actors are described in CAL language with connections of the actors described in a network description language.

There are two independent implementation frameworks of CAL language, namely Open RVC-CAL [126], and OpenDF [127]. Both of them provide a compiler to parse CAL programs to their own Intermediate Representation (IR), and a simulator to simulate the generated IR.

The *Reconfigurable Video Coding* (RVC) framework, introduced by MPEG, is dedicated to the development of video coding tools, and

uses a subset of the CAL named RVC-CAL. The language is defined in MPEG-B (ISO/IEC 23001-4 [128]). The RVC framework is supported by an open-source programming toolset including the Orcc compiler [129] as well as an integrated development environment. Currently, Orcc is able to translate a RVC-CAL dataflow program into C, C++, Java, LLVM, VHDL and XLIM. Only source code is generated, and programs must be compiled with the usual tools. Using the Orcc front-end, a few back-ends have been developed. Cal2HDL [130] is able to generate VHDL implementation from CAL. Cal2C [131] generates the software implementation including the scheduler, using the SystemC runtime support. Roquier *et al.* [132] presented a design flow for the hardware and software synthesis of heterogeneous systems allowing to automatically generate hardware and software components as well as appropriate interfaces, from the RVC-CAL language. Several authors used RVC-CAL language as a starting point for description of SW and HW components in a heterogeneous platform [133, 134, 135, 136].

On top of the OpenDF framework, Ericsson has developed its own C code generator, D2C, targeting ARM processors as part of the European project ACTORS. It first translates the CAL intermediate representation into a C program, and then compiles the generated code into an executable. To support multiple cores, each actor instance is put into one POSIX thread, and scheduling is achieved by the Linux OS. The reader is referred to [137] for further details.

The Cal2Many compilation framework [138] has been developed on top of the CAL language for the Epiphany architecture [139]. The Cal2Many contains two intermediate representations: *Actor Machines* (AM) [140, 141] and *Action Execution IR* (AEIR) [138]. Each actor is first translated to an AM, which describes how to schedule execution of CAL actions. To execute AM, its constructs are transformed to the AEIR and integrated with the application-specific runtime dataflow scheduler. Finally, Epiphany back-end generates C code using a custom communications library and generates channels and mapping of actor instances on processing elements. This approach has been used to generating efficient software implementations for a few programs that are characterized by data-dependent behavior [142, 143]. More recently, motivated by the shortcomings of actor machines when generating highly parallel implementations (such as in hardware), and when composing machines, Janneck proposed *Dataflow Machines* [144] as a model for stream programs.

The RVC-CAL framework has also been extensively used for investigation of efficient implementation techniques for the DPN applications. For example, the Transport Triggered Architectures (TTA) Orcc back-end [24] targets implementation of dynamic dataflow programs on TTA-based multi-core platforms. Leveraging on shared memory architecture, the TTA code generator builds optimized communica-

tion and scheduling infrastructure for the DPN applications. Dynamic dataflow scheduling in the context of multi-core systems has also been studied by Michalska at al [145, 146].

In practice, several performance issues are associated with CAL code generation, such as redundant re-evaluation of actor conditions, large memory footprint required for explicit enumeration of actor states, etc. However, the main issue with the CAL is that it requires a complete re-write/re-implementation of the original reference application, including the actor functions implementation. This turned out to have an unacceptably high development cost for the industry.

1.4 SUMMARY

In this chapter, we have surveyed several important dataflow models of computation and their existing implementations.

In the first part of this chapter, we have explained why the dataflow computation model is well suited in the context of shared memory clusters with application-specific hardware elements. We have introduced the Kahn Process Networks (KPN) and the Dataflow Process Networks (DPN), and overviewed a number of decidable dataflow models derived from the DPN.

In the second part of this chapter, we have examined a number of selected state-of-the-art dataflow frameworks. They can be broadly classified into the domain-specific platforms, the model based frameworks, and the dataflow API based approaches. On the domain-specific side, streaming platforms have been popular with the multimedia applications. They turned out to be insufficiently flexible and required important changes to the familiar development flow to be widely adopted by embedded industry. The model based frameworks allow development and synthesis of complete custom specified systems including hardware and software components. They rely on automated compile-time analysis and code generation tools which are only efficient for restricted decidable dataflow models. The disruptive changes in software development flow and being able to only deliver efficient implementation for a restricted set of the dataflow models hinders adoption of the model based approach by the embedded industry. The API based frameworks extend a standard imperative language, such as C, with some dataflow constructs, and implement the runtime libraries for supporting the dataflow execution. The API approaches focus on the KPN dataflow model because it is straightforward to extend existing sequential implementation into a KPN network by inserting a few KPN constructs at appropriate places. The DPN model, however, is not well supported. Frameworks based on the CAL programming language stand aside among existing dataflow approaches. The idea of RVC-CAL is to provide a dataflow reference im-

plementation instead of a sequential imperative code. CAL approach gained limited traction - besides MPEG video codec no other RVC-CAL reference implementation has been provided. Finally, none of the described frameworks provides dedicated support for integrating tightly-coupled application-specific hardware blocks and for optimizing the dataflow execution.

In the next chapter, we present a new dataflow framework, called StreamDrive that addresses the above issues by (1) extending the tightly-coupled shared memory cluster platform with hardware support for dataflow communication and synchronization, (2) implementing efficient runtime dataflow communication and scheduling leveraging on cluster shared memory, and (3) supporting a *successive refinement* application development flow which reduces the effort required for parallelization of streaming applications.

2 DESIGNING STREAMING ACCELERATORS WITH STREAMDRIVE

Engineering problems are under-defined, there are many solutions, good, bad and indifferent. The art is to arrive at a good solution. This is a creative activity, involving imagination, intuition and deliberate choice.

— Ove Arup

StreamDrive is our configurable dynamic dataflow framework for small-scale clustered shared memory platforms. Its design is greatly influenced by our requirements: support for dataflow with application-specific hardware elements, limited available memory, easy experimentation with implementation techniques, and collection of run-time statistics. The main components of StreamDrive are (i) the hardware architecture optimized for supporting dataflow execution, (ii) the copy-free communication protocol, (iii) the dataflow API, and (iv) the runtime implementation including dataflow communication and scheduling. StreamDrive supports a successive refinement transformation process from a sequential reference algorithm to an optimized DPN implementation.

In StreamDrive, multiple cores are grouped together as clusters which share some local resources such as the internal memory, the DMA engine, etc. The processing cores inside a cluster are light-weight processors (with limited capabilities in terms of pipeline stages, cache mechanism, or virtual addressing). Cluster local memory is multi-bank in order to prevent memory conflicts via the usage of different banks. Within the cluster, the processing cores communicate via the shared memory. The memory may range in size from a few kilobytes up to 512KB and has internal processor pipeline access latency. The communication outside the cluster is managed with help of a DMA engine. Communication and synchronization inside a cluster is significantly faster than between clusters.

The Hardware Platform

The distinguishing feature of the StreamDrive architecture is inclusion of application-specific hardware blocks connected directly to the cluster shared memory. These hardware blocks are small and power efficient, and are essential for achieving the required performance while keeping the cost and the power consumption low. A tight cou-

pling of hardware blocks within a shared memory cluster has been studied in [147] outside of the dataflow context.

The application-specific hardware blocks are connected to the shared memory via a special Hardware Block Bridge (HBB), which supports the dataflow communication and synchronization protocol. The StreamDrive DMA engine also implements the dataflow synchronization protocol. In order to ensure low-overhead synchronization, the processing cores together with the HBB, and the DMA implement a lightweight *synchronization event* mechanism. Synchronization events are communicated via a dedicated event networks inside and outside the cluster.

The Development Flow

It is relatively easy to transform sequential reference code into a KPN form without using specialized languages and tools. A KPN process can be viewed as a sequential function extended with blocking send and receive calls, while the KPN scheduler takes care of saving persistent data across process interruptions and ensures the correct and fair execution. This makes transforming a sequential reference code into a KPN straightforward: it often requires minimal modifications to the sequential code, consisting mostly of inserting send and receive communication statements at appropriate places. The functions and data structures can be converted to KPN incrementally, one function at a time, further simplifying the process.

As explained in previous chapter, the KPN execution is inefficient in a software system. We would like to build a DPN application instead. However, DPN applications are notoriously difficult to develop. In DPN, the send and receive calls are non-blocking and the scheduler does not perform a context switch, thus actors are responsible for saving any data that persist across firings. The actors also need to communicate their firing rules to the scheduler at appropriate places. It is impossible to transform a sequential reference code into a DPN incrementally, one actor at a time, - all functions and data structures need to be converted to DPN form with firings for an application to be executable.

We argue that an evolutionary approach based on two simple ideas can bring high efficiency in terms of parallelization effort. First, the flow and tools for creating and debugging DPN applications must be basically the same as those for standard software: based on familiar and established imperative programming language, such as C. Second, given a sequential reference specification of the algorithm, the application development process should progress toward optimized dataflow implementation through well-defined stages, a process called *successive refinement*. The essential idea of successive refinement is to manipulate the algorithm description by introducing

additional details while both, preserving the original functionality and meeting the constraints that can be evaluated at each new refinement step. The smaller the steps are, the easier it is to verify that functionality is preserved and constraints are met. During successive refinement, it is often convenient to optimize a small part of the algorithm so that functional verification is facilitated.

The StreamDrive supports such incremental successive refinement development flow by allowing the KPN processes and the DPN actors to co-exist within a single dataflow application. During the succesive refinement, a sequential reference algorithm is first transformed into a KPN network, then into a DPN network by introducing the firing rules. The process is performed incrementally, one actor and one firing rule at a time, while verifying the functional correctness and implementation constraints at each code modification.

The Implementation

A runtime environment for dataflow applications can usually be implemented easily because not many services need to be provided. The applications can also be easily partitioned into actors running in hardware and actors running in software. This is due to the parallel specification of the dataflow application on the one hand, and due to the simple interaction of processes/actors over FIFO channels on the other hand. Nevertheless, efficient implementation of dynamic dataflow models of computation has been an important issue hampering the practical use of such models. The difficulties in dataflow implementation originate from the necessity to minimize the overhead for FIFO communication, synchronization, and the runtime scheduling.

The runtime scheduling overhead greatly affects application performance and scalability. Depending on whether the KPN or the DPN model is used, the runtime scheduler implementation faces different trade-offs. The main difficulty in implementing low-overhead scheduler for the KPN computation model is the necessity to implement the process context switch. The context-switch is a mechanism that enables time-multiplexing of several KPN processes over one processing core, and is one of the most performance-critical parts of the scheduler. The context switch involves manipulation of low-level processor state, registers contents, program counter and stack pointer, which may be costly in terms of overhead. Scheduling a KPN network also requires that each software KPN process has its own individual runtime stack. The context switch overhead and the necessity to allocate an individual stack space to each software KPN process, make software implementation of the KPN computation model inherently inefficient. On the other hand, KPN model is a natural fit for streaming application-specific hardware blocks. Because the hardware blocks

do not require scheduling or a software runtime stack, the above mentioned inefficiency does not apply to them.

The DPN scheduling does not require a context switch and is much more efficient. For executing DPN applications, a natural approach is the use of cooperative user-mode scheduling [148]. The scheduler sequentially tests the firing rules from several actors, and fires an actor if a firing rule is valid. A DPN execution in a software system is much more efficient than a KPN execution:

- Knowing beforehand which actors should produce/consume data and which actors should not produce/consume data, the blocking reads and writes are avoided.
- Since actors execute atomically, i.e., they cannot be interrupted in the middle of their execution, there is no need to save state between two firings of an actor, avoiding the costs of context switch.
- At each step during an execution of the network, all actors in the network that have the capability to produce or consume data can be enabled to perform firings. Such execution does not require a separate thread for each actor with storage reserved to hold the actor's stack. The scheduling of the actors can be done by one thread that functions like a scheduler and the runtime stack can be shared by all actors.

However, the DPN mode of execution is less efficient with application-specific hardware blocks. The DPN execution with firings requires a runtime scheduler control over when an actor can be executed. Being controlled by a scheduler is an unnecessary performance overhead for the hardware blocks.

The StreamDrive runtime scheduler provides execution environment in which DPN software actors can co-exist with KPN (software or hardware) processes while the application is executed. Such co-existence of the two models allows low-overhead scheduling where the application-specific functions are executed in hardware as KPN processes without firings, while the software actors follow the DPN semantics with firings.

Our presentation of the StreamDrive implementation proceeds on two parallel levels. On the higher level, we describe data structures and algorithms that we have used in our implementation. On the lower-level, our descriptions are also meant to expose important implementation details and to be a guide to the StreamDrive implementation and the source code.

2.1 STREAMDRIVE COMMUNICATION PROTOCOL

The core of the StreamDrive framework is its *Communication Protocol* specifically designed to support copy-free data communication and lock-free synchronization.

In StreamDrive, actors [1] are connected to communication channels via *input and output ports*. The actors carry the actual computation while exchanging application-specific units of data, called *tokens*, over the communication channels. Tokens are distinguished by their size and the data type. Tokens are *written* to and *read* from the communication channels in FIFO order. Reading from an input port blocks the actor until all required tokens are available in the channel, and writing to an output port blocks the actor until enough empty room is available in the channel for writing.

In order to gain higher efficiency, StreamDrive relies on a fixed-buffer implementation, i.e. token sizes and buffer sizes need to be specified at graph construction time and cannot change during graph execution. The drawback of this is that deadlocks cannot be resolved at runtime. However, the experience is that practical applications exhibit a regular communication behavior that allows software developer to quantify the capacity of the FIFO buffers such that deadlock will not occur. Nevertheless, the StreamDrive provides the runtime timeout service that allows detecting the deadlock condition. Upon detecting a deadlock, the StreamDrive gives debug information about the state of the dataflow graph which helps the developer to eliminate the deadlock.

A standard dataflow FIFO implementation where data must be copied from a source actor to the communication buffer and then from the communication buffer to the destination actor, causes a significant execution overhead. Instead, the StreamDrive communication protocol leverages the cluster shared memory and gives actors direct access to shared communication buffers avoiding memory copy operations.

The StreamDrive protocol defines four basic communication functions listed in table 2.1.

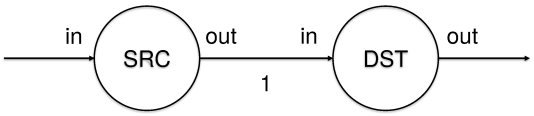
This is different from the standard dataflow `send` and `receive` protocol. The StreamDrive protocol splits each of the `send` function and the `receive` function into two. This allows copy-free implementation leveraging on cluster shared memory available in hardware. The source and the destination actors do not need to make local copies of the data, like the standard dataflow model, but instead can access data directly inside the shared communication buffer. Before writing into an output channel, a source actor must acquire a pointer to an available empty buffer entry via the `RESERVE` call. `RESERVE` is blocking if no room is available inside the given output buffer. When all data

1 In this thesis, we also use term actor for the KPN processes for the sake of simplicity.

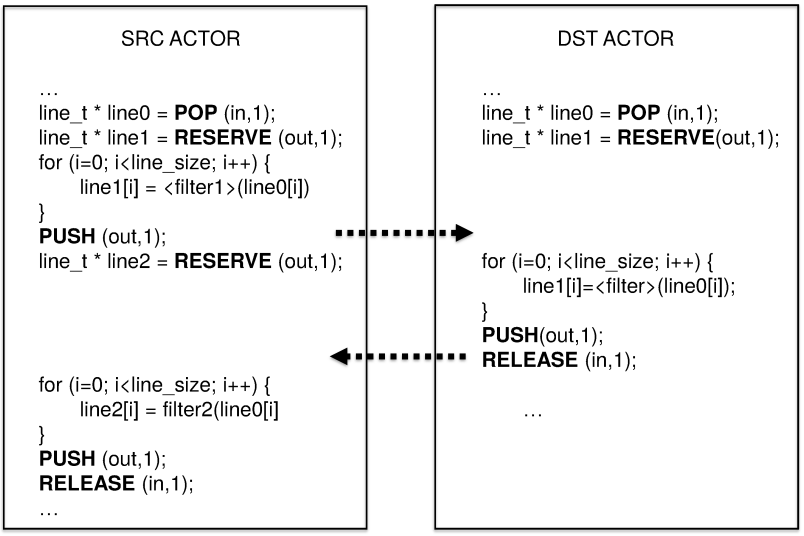
Function	Port	Description
RESERVE(p,n)	OUT	Ensure that at least 'n' tokens are free in output channel associated with port 'p'. Blocks the actor if not.
PUSH(p,n)	OUT	Signal that 'n' tokens have been written to output channel associated with port 'p' Not blocking.
POP(p,n)	IN	Ensure that at least 'n' tokens are ready in input channel associated with port 'p'. Blocks the actor if not.
RELEASE(p,n)	IN	Signal that 'n' tokens can be reused from input channel associated with port 'p' Not blocking.

Table 2.1: Basic functions of StreamDrive Communication Protocol

have been written to the output token, the source actor signals the availability of new tokens via the `PUSH` call. On the destination actor side, an actor must acquire a pointer to an input token via the `POP` call before reading the data. `POP` is blocking if there is not enough available tokens in the FIFO. When the destination actor no longer needs the data, it must signal the source actor that the buffer can be reused via the `RELEASE` function.



(a) Two chained actors in a dataflow graph



(b) The two actors pseudo-code

Figure 2.1: The Illustration of StreamDrive Communication Protocol

Figure 2.1 illustrates the StreamDrive communication protocol. The two actors in a figure are chained as shown in Figure 2.1(a) and perform some sort of a image filtering. Each actor has one input and one output port. The actors process the image line-by-line, therefore the token type, called `line_t` in this example, corresponds to one line of

the filtered image. The SRC actor reads one image line and produces two lines of the new image on each execution. The DST actor reads and writes one line of the image on each execution. The two actors are connected with a channel able to buffer one single token. From the Figure 2.1(b), the SRC actor first reserves one output token for writing the $line1$ data, then it tries to reserve the second token for writing the $line2$ data. Since the FIFO between the two actors can hold only a single token (image line), the SRC actors' execution is blocked on the second `RESERVE` call until the DST actor has `RELEASE`d the buffer location. In the Figure 2.1(b), the DST actor's execution is blocked on `POP` until the SRC actor signals the availability of the next image line via the `PUSH` call.

The four basic StreamDrive protocol functions implement efficient lock-free *bounded buffer synchronization* (BBS) based on shared synchronization counters [20]. In the BBS protocol, a source actor of a communication channel e increments a write pointer `wr(e)`, and a destination actor increments a read pointer `rd(e)`. Both pointers are shared and visible to other actors in the system. On a `send`, the difference $capacity(e) - (wr(e) - rd(e))$ gives the number of free entries in the output FIFO buffer; on a `receive`, the difference $wr(e) - rd(e)$ gives the number of available tokens in the input FIFO buffer. We have extended the original BBS to support the *reserve-push-pop-release* instead of the standard *send-receive* protocol. In our implementation, the source actor also increments a private (not visible to other actors) `reserved(e)`, while the destination actor increments a private `poped(e)` counters. Such private counters can be implemented in shared memory or as internal registers inside the application-specific hardware elements, for example. Duting the `RESERVE`, the difference $capacity(e) - (reserved(e) - rd(e))$ gives the number of free entries in the output FIFO buffer; the `reserved(e)` is incremented. On a `PUSH`, the `wr(e)` counter is incremented. At the destination actor, on a `POP`, the difference $wr(e) - poped(e)$ gives the number of available tokens in the input FIFO buffer; and the `poped(e)` is incremented. On a `RELEASE`, the destination actor increments the `rd(e)` counter.

Notice that in the new *reserve-push-pop-release* protocol, *reserving* a token by the source actor gives it access to the token location in the FIFO buffer but does not make it available for the destination actor. The token only becomes available when *pushed*. Similarly, *poping* of a token by a destination actor is decoupled from when the token can be reused by the source actor. The token can only be reused after the destination actor has *released* it. The decoupling of the *acquisition* of a token, on the one hand, and the *pushing* or *releasing* it on the other, allows actors to directly access token locations in shared memory instead of making local copies of tokens as in classical dataflow implementations.

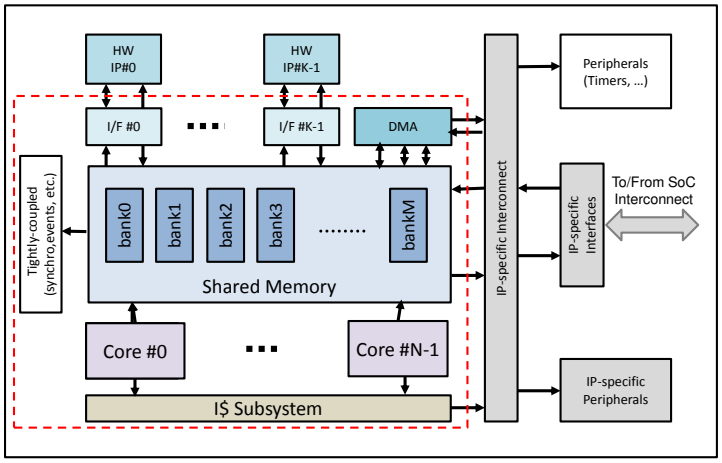


Figure 2.2: The StreamDrive Cluster Block Diagram

2.2 THE STREAMDRIVE ARCHITECTURE

The StreamDrive architecture evolved from the ST Microelectronics Platform 2012 project [15]. Figure 2.2 shows the block diagram of the StreamDrive cluster. StreamDrive is a heterogeneous tightly-coupled cluster composed of a number of programmable Processing Elements (PE), application-specific Hardware Processing Elements (HWPE), and a DMA, all connected together to a shared Tightly-Coupled Data Memory (TCDM). Using shared memory instead of a cache is a power-saving feature because caches consume more power than the scratch-pad memories.

The TCDM contains an application working set used by both, PEs and HWPEs. In this way, the TCDM storage replaces (fully or partially) the hardware elements' dedicated storage, with advantages both in terms of area (no buffer duplication) and performance (no need to copy buffers between different memories). The size of the TCDM has important impact on area-efficiency (GOPS/mm2) of the system: the larger the TCDM, the lower the area-efficiency. A relatively small TCDM memory (up to 512KB in current implementation) cannot hold the entire application working set and requires frequent data movement between the TCDM and the external memory. The DMA is used for transferring data between the TCDM and external memory. The DMA also ensures additional function of supporting stream synchronization of data transfers.

We use simple RISC in-house processing cores running at relatively low frequency (typically 500 MHz). Lowering the frequency of programmable cores improves power-efficiency of the cluster. The cores are extended with the *synchronization events* handling extension. The programmable cores are connected directly to the TCDM memory with internal pipeline access latency.

While the application working set is loaded into the TCDM, the application instruction code is kept in off-cluster external memory. The instruction cache sub-system ensures efficient fetching of application code from external memory to StreamDrive cores. We have chosen to use instruction cache rather than dedicated program memory for two reasons: (1) even though instruction cache energy consumption is somewhat higher, it generally requires smaller size than a program memory for the same application; (2) instruction cache can gracefully handle programs exceeding its hardware size. StreamDrive instruction cache is shared by all processing cores, which significantly reduces number of external memory accesses and conflicts on external memory bus. Because the StreamDrive is used in SoC systems with potentially high memory latency, we need to carefully dimension the instruction cache size. In our experience the biggest application is about 100KB, and a 64KB instruction cache is sufficient to eliminate virtually all replacement misses.

The application-specific HWPEs are essential for achieving the required performance while keeping the cost and the power consumption low. In order to even further optimize power-efficiency of the system, the HWPEs can run each in their own different dedicated clock domain, thus allowing for the adjustment of their frequency in accordance with application requirements. The connection between the HWPEs and the shared memory is ensured by the HBB (I/F 0, .. I/F K-1 in the figure) that serves as a bridge for streaming hardware blocks. The PEs, the HBB, and the DMA, all support the StreamDrive communication protocol based on shared memory - this creates a common infrastructure for the core-to-core, the core-to-hardware-block, or the hardware-block-to-hardware-block communication.

The StreamDrive cluster also includes a small number of tightly-coupled peripherals aiming at accelerating the synchronization, event handling, etc.

The key element of the StreamDrive cluster is its *logarithmic* interconnect [149] that allows multiple concurrent accesses to the multi-bank TCDM memory. In order to minimize the number of stalls due to conflicting simultaneous accesses to the same bank, the banking factor (i.e. the ratio between the number of TCDM memory banks and the number of access ports), needs to be correctly dimensioned. Such shared memory organization, although it has a limited scalability, corresponds well to the small-scale cluster architecture that we target. Our experience, confirmed by other studies on similar architectures [7], shows that this type of interconnect can support up to 32 access ports, each with a throughput close to 32-bits/cycle with latency compatible with the RISC core internal pipeline, under the embedded IP target frequencies. As a result, the logarithmic interconnect technology constraints limit the scale of a StreamDrive cluster to around 32 processing elements. When a single StreamDrive cluster cannot

deliver necessary performance, multiple clusters can be put together, thus allowing massive upscaling in performance while maintaining the initial power- and area- efficiency.

Following StreamDrive architecture elements ensure efficient implementation of the dataflow execution model: (1) the *Synchronization Event Network* together with a processing element's *Event Handling Extension* (EVTx) ISA extension, (2) the HBB for connecting the application-specific hardware elements to shared memory, and (3) the *Dataflow-Aware DMA*.

2.2.1 The Event Synchronization Network

An essential extension to the processing element's ISA that ensures efficient implementation of the dataflow synchronization is the EVTx. The EVTx is built around the concept of hardware *events*. An event is similar to the processor interrupt in that both, the interrupts and the events, are signals delivered to the processor asynchronously with respect to the normal execution flow. However, there is one important difference, which makes events much more efficient than interrupts for implementing multiprocessor synchronization primitives. When an interrupt occurs, the interrupt handler executes code that is not part of the normal execution flow. An interrupt is handled by the processor as soon as it arrives (eventually depending on the interrupt priority level) - normal execution is then interrupted, which implies a penalizing context switch while processing the interrupt. On the contrary, a hardware event handling may be delayed as long as the normal execution flow does not request that the event be handled. Thus, the event handler is a part of normal application execution. Event handling does not require a context switch and allows extremely efficient (few processor cycles) implementation of parallel synchronization primitives.

In StreamDrive, the hardware events are used to avoid active polling of shared memory locations while waiting for dataflow tokens to become available. It has been noticed previously that polling for dataflow firing rules may incur significant overhead in terms of performance and energy consumption [150]. One interesting solution for reducing this overhead has been proposed by Martin *et al.* [150]. The authors developed a concept of *Notifying Memories*, where special interconnect components can trigger/receive notifications according to some events. The particular events of interest are changes in the dataflow communication channels state. In StreamDrive, instead of a special interconnect components, it is up to the PEs, the HWPEs, and the DMA, to generate a hardware event every time that the dataflow graph state (the number of tokens in communication channels) changes. The hardware event approach is more lightweight, more flexible, and more scalable compared to *Notifying*

Memories. On the other hand, a PE or a HWPE can enter an energy saving idle state while waiting for a hardware event. For the event generation, the PEs use the EVTx, a tiny extension to the processor instruction set that implements instructions for generating hardware events and for inquiring event status; the HWPEs rely on the HBB for generating these events (see the Hardware Block Bridge description below); the DMA also integrates event generation functionality.

The StreamDrive *Event Synchronization Network* connects the hardware events from all platform elements. It allows selectively deliver hardware events from a set of sources to a set of destinations, depending on StreamDrive dataflow graph connections. Functionally, the Event Network essentially ORs the events from a set of sources and sends the result to a set of destination elements.

2.2.2 The Hardware Block Bridge

The HBB provides the HWPEs an interface that abstracts the system memory addresses into a simpler token based representation, which can then be implemented using the streaming type of communication. Such token representation may go from very simple, ex. a linear streaming with standard FIFO read and write operations, to more complex access patterns, such as a sliding convolution window, etc. The HBB performs the following tasks:

1. It transforms streaming HWPE read and write requests without the address, into a sequence of naturally aligned LOAD and STORE requests with full system address to pre-allocated buffers in shared memory.

2. It pipelines the generated memory LOAD and STORE transactions.

3. It manages the HWPE working set as rotating buffers of tokens by implementing dataflow synchronization compliant with the StreamDrive communication protocol.

4. It multiplexes multiple streaming requests from HWPEs into a limited number of shared memory ports.

5. It ensures the clock domain frequency crossing between the HWPEs and the StreamDrive cluster.

Among others, the token abstraction allows implementation of a *rotating buffer storage* model. This model is very useful in image processing. Each HWPE input and output channel has an associated rotating buffer inside the TCDM memory. For example, a HWPE applying a filter on an image, could require access to several lines of the input image at a time as a temporal window. When the entire image does not fit in the relatively small TCDM memory, the rotating buffer would

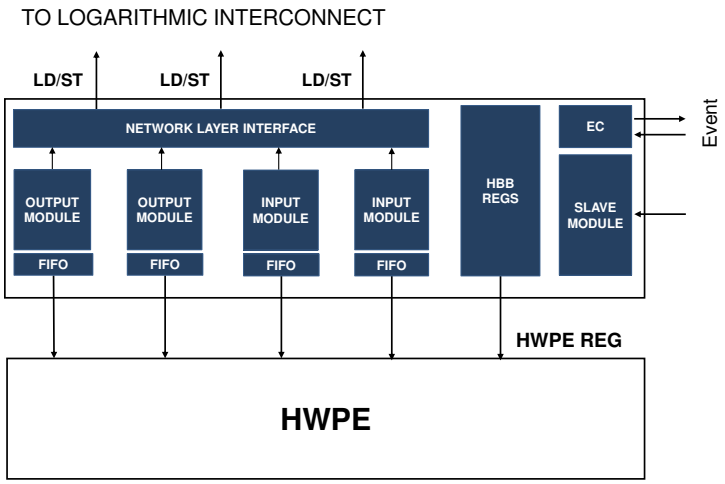


Figure 2.3: The HBB Block Diagram

hold enough image lines for an uninterrupted pipelined operation of the HWPE algorithm. Every time that the algorithm slides vertically by one line in the input image, the window is rotated, i.e. the oldest line(s) are replaced with the not yet seen line(s) of the input image. The rotating buffer model allows every data to be brought to the TCDM memory only once. Such rotating buffer storage model is different from a storage model used by standard interfaces, such as the OpenMP, for example. In a standard interface, the data are expected to be contiguously stored. Thus, in the above example, all image lines in a window need to be stored in the TCDM contiguously. The inefficiency of contiguously storing the temporal windows appear when lines in the overlapped region between consecutive windows need to be brought to the TCDM once for each window. For example, a 3x3 convolution, operating on a window of 3 lines at a time, would require every line to be brought in at least 3 times, as it is used in 3 different convolution windows.

Figure 2.3 shows the HBB block diagram.The HBB includes following components:

- The HBB Registers.
- The HBB Slave Module (SM) which provides the StreamDrive system with access to the HBB and HWPE registers.
- The HBB Input Module (IM) which provides the read TCDM interface to the HWPEs.
- The HBB Output Module (OM) which provides the write interface to the HWPEs.
- The HBB Event Controller (EC) which sends the in-coming events to the input and the output modules, and sends the outgoing events when the input or output module(s) generate one.

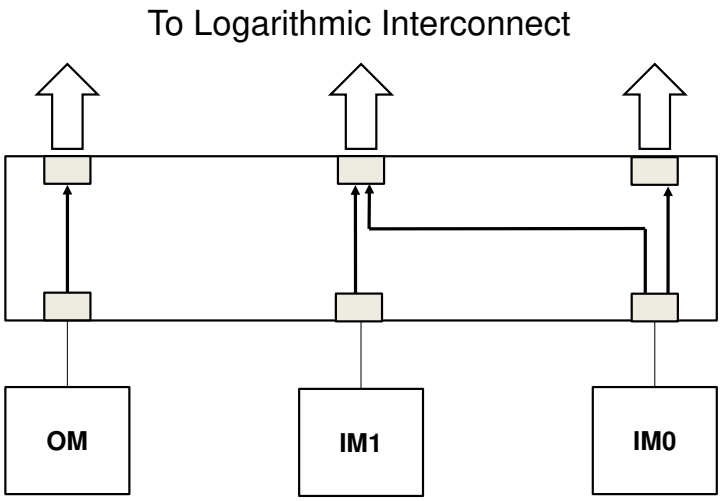


Figure 2.4: The HBB Network Interface Example

- The HBB Network Interface (NI) which connects the input and the output modules to the TCDM interface via a customized application-specific topology.

The HBB registers accessed via the HBB SM allow configuration of the HBB and its modules. In particular, the dataflow FIFO buffers need to be setup, eg. their location in TCDM, token size, FIFO depth, etc. The HBB registers also allow configuring the synchronization event generation.

The HBB Input and Output Modules implement the StreamDrive communication protocol, including the private synchronization counters. The Input Module transforms the stream of reads emitted from a HWPE into a sequence of LOADs to TCDM memory. On the TCDM side, each Input Module contains a small transport FIFO which is used to handle misaligned LOADs and to pipeline multiple TCDM transactions. The Input Module can also prefetch token data ahead of HWPE request. The Output Module transforms the stream of writes emitted from a HWPE into a sequence of STOREs to TCDM memory. On the TCDM side, each Output module contains a small transport FIFO which is used to handle misaligned STOREs, and to aggregate and pipeline multiple TCDM transactions. Each Input and Output module includes a small address generator which is used to compute pointers into TCDM FIFO buffers with eventual wrap around.

The HBB Network Interface connects the HBB to M logarithmic interconnect ports. The M and the port bit-width are design-time parameters allowing to customize the TCDM access bandwidth capacity in line with the application requirements. The Network Interface then combines and redirects requests from N HWPE Input and Output Modules into M logarithmic interconnect TCDM access ports.

A further design-time customization is possible via application-specific network topology: associating the HBB Input and Output

Modules to the logarithmic interconnect TCDM access ports. For example, Figure 2.4 shows a Network Interface with two Input Modules, IM0 and IM1, and with one Output Module, OM, connected to three TCDM access ports. The OM module, is connected directly to one of the TCDM access ports and is the only master on that port. The IM1 module is also connected to a single TCDM access port. However, the IM1 is sharing this port with the IM0 module. The IM0, in turn, is connected to two TCDM access ports. This topology may be useful, for example, if the IM0 requires high bandwidth, while the IM1 has very low bandwidth requirements.

The HBB is always a slave to the associated HWPE: it provides the HWPE with the implementation of the four basic StreamDrive communication protocol functions, and two additional `READ` and `WRITE` functions, shown in Table 2.2:

Function	Port	Description
READ(t,o,n)	IN	Read 'n' bytes from the token 't' at offset 'o'. Associated with an input port. Data is expected to arrive in packets of the input port width.
WRITE(t,o,n)	OUT	Write 'n' bytes to the token 't' at offset 'o'. Associated with an output port. Data is expected to be sent in packets of the output port width.

Table 2.2: Additional functions provided by the HBB to HWPEs.

2.2.3 The Dataflow-Aware DMA

In a StreamDrive cluster, data transfers between the off-cluster memory and the TCDM are handled by a dedicated DMA. In general, we want the source and the destination actors of a DMA transfer not to be same. If we restricted these to be the same actor, it would limit the dynamicity of program execution: the DMA transfer requests would be executed in some predefined order with respect to computations using the transferred data. By decoupling data transfer requests from computations, the data transfer requests are only constraint by the availability of free memory space for a new transfer to start. The difficulty in using the DMA within such a context is that it needs to be synchronized with the source and the destination dataflow actors.

First of all, both actors need to share the knowledge of the particular DMA transfer request: the source actor in order to launch the transfer, and the destination actor in order to synchronize on the transfer completion. Sharing such additional information in a dataflow environment is cumbersome leading to additional communication channels and unnecessary performance and memory overheads. The StreamDrive communication protocol enables implementation of the DMA functionality such that no additional information exchange between actors involved in a DMA transfer is necessary.

Thus, the source actor of the data launches a DMA transfer via a standard DMA API with data transferred as a token to a normal output dataflow channel; this is equivalent to *reserving* the token. Upon data transfer completion, the DMA signals that the token is ready; this is equivalent to *pushing* the token. The destination actor then finds the transferred data as a dataflow token from the corresponding input channel; this is equivalent to *poping* the token. When data is no longer used by the destination actor, it *releases* the token so that the source actor can reuse the FIFO buffer location for a new DMA transfer. No additional synchronization between the two actors about the DMA transfer is necessary. As a mater of fact, the destination actor does not even know that the data come from a DMA transfer.

The DMA synchronization mechanism needs to be very efficient because we want the DMA to be efficient not only with large DMA transfers (which is usually the case) but also with relatively fine-grain transfers. Traditionally, the DMA synchronization can be accomplished by busy-waiting on a DMA status register, or by the DMA generating an interrupt on transfer completion. This mechanism would be too penalizing for the finer-grain data transfers. Instead, the StreamDrive DMA leverages on the synchronization event mechanism described earlier. Thus, the StreamDrive DMA is extended with following functionality upon a DMA transfer completion:

- The DMA updates the value of associated shared synchronization counter.
- The DMA generates associated synchronization event towards the Event Synchronization Network.

As a result, the StreamDrive DMA API is extended with (1) the address of the synchronization counter associated with the given DMA transfer; (2) the new value for this synchronization counter (the DMA only performs writing of the new value, not the increment); and (3) the synchronization event mask associated with this transfer. These very simple extension to the standard DMA functionality results in a seamless integration of the DMA with the StreamDrive dataflow framework.

2.3 THE STREAMDRIVE API

The StreamDrive API is based on the C programming language and provides methods for defining the dataflow actors, for constructing the dataflow graph, and for controlling the runtime scheduler.

The StreamDrive application structure is shown in Figure 2.5. Arrows in the Figure show objects dependencies. For example, the code in `main.c` depends on declaration of actors from `<actor>.h` of all actors.

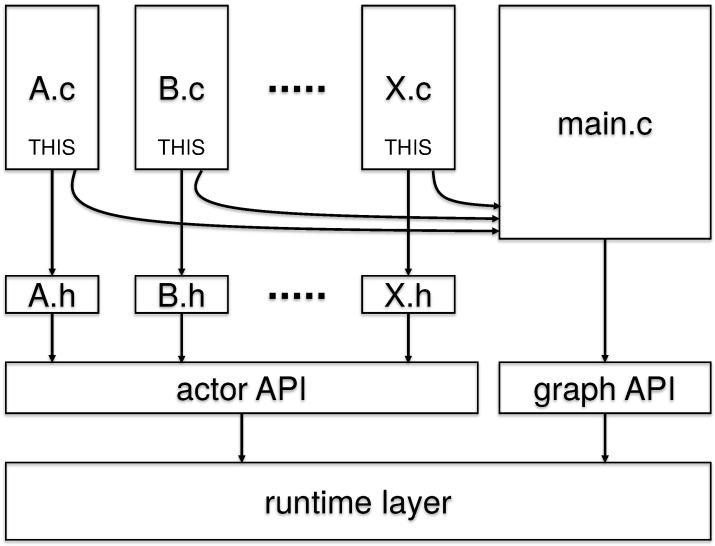


Figure 2.5: A StreamDrive Application Organization

The `main.c` objects depend on StreamDrive graph API, while the `<actor>.h` declarations depend on StreamDrive actor API, both declared in `streamdrive.h` header file. The StreamDrive API is implemented by the StreamDrive runtime system, included as a statically linked library, `libstream.a`

A StreamDrive application includes (1) a main part, written in C and defining the `main` function and any other code required by the `main` function, and (2) any number of actors conveniently specified as one declaration part, the `<actor>.h`, and one actor definition part, the `<actor>.c`. The main part builds, configures, and launches execution of StreamDrive dataflow graphs, as well as it terminates the processing. The dataflow graph management functions are defined by StreamDrive *graph* API. The graph API provides methods for creating actors and their ports, and for connecting actors via communication buffers. The StreamDrive graph description can be parameterized in number of actor instances and their connections. The API supports the configuration of dataflow graphs between executions by disabling actors, actor connections and by changing actor parameters. It is important to note that the application graph does not need to change depending on whether actors are implemented as software functions or as hardware blocks. The StreamDrive *actor* API defines a number of functions for managing dataflow actors. A single StreamDrive actor may be instantiated multiple times. All actors' data associated with a particular actor instantiation are accessible from actors' code via a special `THIS` pointer set by the runtime scheduler.

2.3.1 Actor API

A StreamDrive actor needs to define its symbolic name, its data type and declare its input and output ports. Listing 2.1 shows the header declarations for the FAST actor from the ORB application [25].

```
1  typedef struct {
2    ...
3    uint32_t        _cFastThreshold;
4  } fast_t;
5  STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);
6
7  #define FAST_IN_ESIZE             (sizeof(Image_t))
8  #define FAST_OUT_ESIZE            (sizeof(Keyp_t))
9  #define FAST_PORT_IN              0
10 #define FAST_PORT_OUT             1
11 #define FAST_PORT_COUNT           2
```

Listing 2.1: The FAST actor header declaration

The STREAM_DECLARE_ACTOR_TYPE macro associates the actors' symbolic name with its data type, it translates into: `typedef fast_t FAST_actor_t;`. The actor data type is any application specific type. The FAST actor in the Listing also defines symbolic names for the id numbers for the two communication ports, one input and one output, and for the token sizes for these two ports.

Each StreamDrive actor definition requires four basic functions: CONSTRUCTOR, DESTRUCTOR, INIT, and WORK. The CONSTRUCTOR and the DESTRUCTOR perform all actions required at actor creation and release time, in particular the actor ports are created inside the actor constructor function. These functions are called once at dataflow graph construction and termination time, respectively. The dataflow graph can be executed multiple times: this feature allows reconfiguring the dataflow actors and their connections for each new execution. The INIT function configures the actor for a new execution by initializing actor's internal state, therefore the INIT is called every time a dataflow graph is executed. Finally, the WORK function implements the actor functionality. Listing 2.2 shows the definition of the basic functions for the FAST actor.

Inside the CONSTRUCTOR function any actor specific initiations are performed. It accepts any application-specific arguments that may influence how the actor is instantiated. In particular, actor ports need to be instantiated inside the actor CONSTRUCTOR. The macro STREAM_ACTOR_MAKE_PORT_IN creates an input port `"in_p"` with `id=FAST_PORT_IN` , and with the token size FAST_IN_ESIZE. The port `id` and the token size have been defined in the earlier Listing. Similarly, the STREAM_ACTOR_MAKE_PORT_OUT macro creates an output port `"out_p"` with a different `id` and token size. These ports can be addressed from the actors' INIT, WORK, and DESTRUCTOR functions via a special THIS pointer.

```
1  STREAM_CONSTRUCTOR (void * arg) {
2    STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p", FAST_IN_ESIZE);
3    STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p", FAST_OUT_ESIZE);
4  }
5  STREAM_DESTRUCTOR (void) {
6    STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);
7    STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);
8  }
9  STREAM_INIT(void * arg) {}
10 STREAM_WORK(void) {
11   int16_t fastThreshold = THIS->_cFastThreshold;
12   uint8_t n_levels = cfg->n_levels;
13   Image_t * img_pyramid = cfg->img_pyramid;
14   uint32_t level;
15
16   for (level = 0; level < n_levels; ++level) {
17     int cornerCount;
18     fast9_detect(level, &img_pyramid[level], ..., &cornerCount);
19   }
20 }
```

Listing 2.2: The FAST actor definition

The INIT function in the Listing is empty because this actor does not require reconfiguration at execution time. Generally, any application-specific data can be passed to the INIT function as a `void *` argument, which can be used to specify the configuration parameters.

The WORK function implements the actor functionality. It is typically derived from the original reference sequential algorithm with StreamDrive actor API calls inserted at appropriate places in the code. For example, following functions implement the StreamDrive communication protocol:

```
1  STREAM_IN_PORT_POP (PORT_ID, NUM_TOKENS)
2    Implements the StreamDrive communication protocol pop function. Ensures that
3    NUM_TOKENS are available in channel connected to PORT_ID. If blocked then
4    invokes a KPN context switch.
5
6  STREAM_IN_PORT_RELEASE (PORT_ID, NUM_TOKENS)
7    Implements the StreamDrive communication protocol release function. Signals
8    that NUM_TOKENS can be reused in channel connected to PORT_ID.
9
10 STREAM_OUT_PORT_RESERVE (PORT_ID, NUM_TOKENS)
11   Implements the StreamDrive communication protocol reserve function. Ensures
12   that NUM_TOKENS can be written to channel connected to PORT_ID. If blocked
13   then invokes a KPN context switch.
14
15 STREAM_OUT_PORT_PUSH (PORT_ID, NUM_TOKENS)
16   Implements the StreamDrive communication protocol push function. Signals that
17   NUM_TOKENS are ready in channel connected to PORT_ID.
```

Of particular importance are functions for specifying the dataflow firing rules:

A firing rule is specified by requiring certain number of free slots (output ports) or available tokens (input ports) to be available in a

```
STREAM_PORT_SET_QUOTA (PORT_ID, NUM_TOKENS)
  Implements the dataflow firing rules. Requires that NUM_TOKENS are available
  (or can be written) in channel connected to PORT_ID.

STREAM_YIELD ()
  Implements dataflow actor end of firing by yielding control to the
  StreamDrive runtime scheduler. Does not invoke the context-switch.
```

given communication port before the next firing of the actor can take place. The firing rules can change for each new actor firing, supporting fully dynamic dataflow model. In the absence of firing rules, an actor behaves as a KPN process, possibly blocking during execution.

The StreamDrive model requires explicit management of the memory hierarchy, in particular of transferring data between the external and the local shared memory. Our experience is that streaming applications have regular memory access patterns and the advantages of explicit memory hierarchy management outweigh its inconveniences. The memory transfer management is supported by the StreamDrive DMA API.

2.3.2 Graph API

The StreamDrive graph API provides functions for instantiating actors, connecting the actors, enabling and disabling them, launching dataflow graph execution, etc. The two functions that declare dataflow actors are:

```
STREAM_DECL_SW_ACTOR (ACTOR, NUM_PORTS, STACK_SZ)
  Declares a software dataflow actor, ACTOR, and initializes associated actor
  structure: ports number, required stack size, pointers to this actor's
  constructor, destructor, init, and work functions.

STREAM_DECL_HW_BLOCK (ACTOR, NUM_PORTS)
  Declares a dataflow actor, ACTOR, implemented as an application-specific
  hardware element. Initializes associated actor structure: ports number,
  pointers to this actor's constructor, destructor, and init functions. The
  difference with a software actors is that no work function is specified, and
  no runtime stack needs to be allocated.
```

The StreamDrive actors must be declared outside of any function - in the global scope of the C language, for example. Once declared, the software actors or hardware blocks can be instantiated dynamically inside the main part of a StreamDrive application. The API function allowing to instantiate an actor is:

```
STREAM_ACTOR_MAKE (ACTOR, NAME_OR_ID, ARGS)
  Instantiates a software actor or an application-specific hardware element.
  Returns a pointer at ACTOR_actor_t of this instance. The pointer points either
  at the actor instance's data structure in shared memory, or at HWPE internal
  registers base. The NAME_OR_ID sets this instance's name string for a software
  actor or an integer HWPE id for the hardware blocks. Allocates actor data
```

```
7     structures within StreamDrive runtime scheduler, and a dedicated runtime stack
8     in off-cluster memory if necessary.
```

The actors need to be connected together to form a dataflow graph. To connect two StreamDrive actors, the communication channels need to be built and a binding API function needs to be called. For example, the binding function below allows connecting an output port of one actor to the input port of another actor:

```
1   STREAM_MAKE_BUFFER (SIZE, POOL)
2     Allocates a FIFO buffer of SIZE bytes in memory hierarchy level specified by
3     POOL. POOL can specify the cluster shared memory or the off-cluster memory.
4
5   STREAM_BIND_OUT_TO_IN (SRC, OUT, DST, IN, BUFFER)
6     Connects a port with id=OUT of an instance SRC of an actor to a port with
7     id=IN of an instance DST of an actor. The SRC and the DST are actor instance
8     pointers returned by the STREAM_MAKE_ACTOR macro. The ports are connected via
9     the FIFO channel BUFFER. The token sizes of the two connected ports do not
10    need to be same, but care must be taken that the size of one port token is
11    an integer multiple of the other token.
```

The dataflow model of computation defines a single source and a single destination communication FIFO buffers. This is an essential requirement for ensuring the dataflow execution properties and correctness. On the other hand, this also creates a significant execution overhead: when a source actor is connected to multiple destination actors, a special *broadcast* actor needs to be inserted between them in order to *copy-forward* the data from the source to each destination. As a result, several copies of the same data must be made and several copy operations executed, one for each destination actor. An important improvement of the StreamDrive framework over previously existing dataflow frameworks is that it allows different actors to efficiently share communication FIFO buffers.

Several solutions to broadcast data tokens can be found in the literature. Fischaber *et al.* [151] used non-destructive reads, also called FIFO *peeking*, as a way to read data tokens without popping them from FIFOs, hence avoiding the need for broadcast actors. Unfortunately, this technique cannot be applied without considerably modifying the dataflow model. Indeed, the use of FIFO peeking means that an actor does not have the same behavior for all firings. Otherwise, tokens of peeked FIFOs would never be consumed and would accumulate indefinitely. Desnos *et al.* [152] proposed a dataflow buffer merging technique to solve the broadcast issue for the SDF graphs. In SDF model, the broadcast buffers can simply be shared by all destination actors without requiring additional run-time synchronization. With dynamic dataflow models, special single-writer, multiple-readers FIFOs are needed that discard data tokens only when all readers have consumed them. Mamidala *et al.* [153] implemented such FIFOs targeting a shared memory architecture. This implementation relies on existence of atomic (locking) Fetch and Increment primitive, which

is very expensive in terms of performance cost. This cost is carried every time that an actor reads data from a broadcast FIFO.

In StreamDrive framework, a special *broadcast* connection allows one source actor and multiple destination actors to share a single FIFO buffer. It implements a lock-free, counter based, lightweight synchronization between the source and the destination actors of a *broadcast* connection. Each *broadcast* destination actor has a dedicated *release* counter – the *global release* counter is incremented only when all destination actors have incremented their dedicated *release* counters. The *global release* evaluation is *lazy*, i.e. it is carried over only when *broadcast* source actor is trying to *reserve* an entry in the FIFO and that the FIFO seems full. The `broadcast` connection is specified using following API functions:

```
1   STREAM_MAKE_BROADCAST (FANOUT, SIZE, POOL)
2     Returns a handle for a broadcast data structure with FANOUT output connections
3     and with buffer capacity of SIZE bytes allocated in the memory hierarchy level
4     POOL.
5
6   STREAM_BIND_OUT_TO_BROADCAST (SRC, OUT, BCST)
7     Connects a port with id=OUT of an instance SRC of an actor to a broadcast. The
8     SRC is an actor instance pointer returned by the STREAM_MAKE_ACTOR macro. The
9     BCST is a handle returned by the STREAM_MAKE_BROADCAST macro.
10
11  STREAM_BIND_BROADCAST_TO_IN (BCST, DST, IN, IDX)
12    Connects a broadcast to a port with id=IN of an instance DST of an actor. The
13    BCST is a handle returned by the STREAM_MAKE_BROADCAST macro. The DST is an
          actor
14    instance pointer returned by the STREAM_MAKE_ACTOR macro. The input port is
15    connected to the IDX output of the broadcast.
```

The baseline dataflow model does not provide efficient support for data-parallelism. Typically, some sort of *split* and *join* actors need to be inserted around a data-parallel actor to *copy-forward* tokens in a round-robin order to multiple data-parallel actor instances. This leads to significant overhead: the memory overhead for holding multiple copies of the same token; the performance overhead for performing multiple copy operations and for scheduling the *split* and the *join* actors.

In StreamDrive, we avoid having these additional *split* and *join* actors by leveraging on the above *broadcast* connection and its symmetric *collect* connection. The *collect* allows multiple source actors to be connected to a single destination actor and share a communication buffer. While the usefulness of the *broadcast* connection has been acknowledged previously, the *collect* has been overlooked. Intuitively, where with the *broadcast* one FIFO buffer is implementing multiple communication channels, with the *collect*, multiple FIFO buffers implement a single communication channel. Such *collect* connection leads to a reduction in memory footprint and a performance improvement by avoiding to physically copy the collected tokens into a single common FIFO. In StreamDrive implementation, each *collect* source actor has a dedicated *push* counter – the *global push* counter is incre-

mented when all *collect* actors have incremented their dedicated *push* counters. The *global push* evaluation is *lazy*, i.e. it is carried over when the destination actor is trying to *pop* a token from the FIFO and that the FIFO seems empty. The `collect` connection is specified using following API functions:

```
1  STREAM_MAKE_COLLECT (FANIN, SIZE, POOL)
2    Returns a handle for a collect data structure with FANIN input connections
3    and with a buffer of SIZE bytes allocated at memory hierarchy level POOL.
4
5  STREAM_BIND_COLLECT_TO_IN (DST, IN, CLCT)
6    Connects a collect to a port with id=IN of an instance DST of an actor. The
7    CLCT is a handle returned by the STREAM_MAKE_COLLECT macro.The DST is an actor
8    instance pointer returned by the STREAM_MAKE_ACTOR macro.
9
10 STREAM_BIND_OUT_TO_COLLECT (SRC, OUT, CLCT, IDX)
11   Connects a port with id=OUT of an instance SRC of an actor to a collect. The
12   SRC is an actor instance pointer returned by the STREAM_MAKE_ACTOR macro.The
13   CLCT is a handle returned by the STREAM_MAKE_COLLECT macro. The output port is
14   connected to the IDX input of the collect.
```

A data-parallel actor, then, can be constructed by connecting multiple parallel actor instances via the *broadcast* connection to source actors and via the *collect* connections to destination actors.

It is important to understand that, in order to guarantee that all tokens are always processed in the FIFO order, the *broadcast* and *collect* conceptually "forward" all the same tokens to all involved actors: each *broadcast* destination actor "consumes" all tokens in-order, and each *collect* source actor "generates" all tokens in-order. These actors do not need to physically read or write the tokens, they need only *reserve*, *push*, *pop*, and *release* them. Such implementation of the *broadcast* and *collect* connections gives a choice for a data-parallel implementation: data-parallel actors may choose to process a sub-part of every token, or to process a different token each, whichever results in lower parallelization overhead. There is no run-time overhead associated with either implementation because each StreamDrive communication protocol function handles multiple tokens.

The StreamDrive also provides the API for configuring a dataflow graph via enabling actors - it is possible to only enable a subset of all actors for any particular execution.

```
1  STREAM_ACTOR_ENABLE (ACTOR, ARGS)
2    Includes the actor ACTOR instance for execution with the dataflow graph.
3    The ACTOR is actor instance pointer returned by the STREAM_MAKE_ACTOR
4    macro. This calls the actor's INIT function and passes ARGS along to it.
```

Enabling or disabling actors allows the runtime scheduler to only deal with a subset of the dataflow graph. Disabling dataflow connections can be achieved by setting corresponding dataflow firing rules to 0 and in actor's code.

Finally, the StreamDrive API includes a few functions to control the runtime scheduler. For example, actors can be given scheduling priority or be assigned to a particular processing element. The complete API is specified in Appendix A.

2.4 THE SUCCESSIVE REFINEMENT FLOW

One important objective of the StreamDrive is to support the successive refinement transformation of a sequential reference code into an optimized dataflow form. In order to facilitate such transformation, the process is divided into a number of conceptually simple steps, each consecutive step is an incremental improvement over the previous one:

1. Identification of the dataflow part of the sequential application.
2. Identification of the dataflow actors and building the initial Kahn Process Network, KPN.
3. Refinement of the initial KPN by reducing actors granularity.
4. Identification and implementation of data parallel actors.
5. Conversion of the KPN into the Dataflow Process Network, DPN, by introducing the dataflow *firing rules*.
6. Optimization of the performance vs memory footprint trade-off.

The initial transformation of a sequential reference code into KPN form is facilitated by the fact that streaming applications are typically structured into a sequence of processing kernels that roughly correspond to Kahn processes. The biggest effort goes into achieving good performance vs memory footprint trade-off beyond the initial *basic* level. In this respect, the StreamDrive is not different from other parallelization approaches - usually a good understanding of the model is required in order to achieve high performance levels.

Importantly, all transformation steps can be performed incrementally, one actor at a time, allowing at each stage to debug the implementation and verify that it remains functionally correct with respect to the initial reference results. In order to gain a more precise idea of the StreamDrive incremental parallelization flow, we illustrate the process using a real-life example, the ORB application, mentioned earlier. The ORB algorithm identifies a set of objects inside an image and matches their descriptors to the descriptors of objects in a trained database. The objects are identified by detecting *keypoints* of interest via the *FAST algorithm*. The "false" keypoints are then removed via the *nonmax suppression* and the remaining keypoints are sorted using *Harris response* measure to retain only the "best" keypoints. For these keypoints, the algorithm computes object *orientation*, and the BRIEF *descriptor* of the object associated with each keypoint. The descriptor computation requires the *Gaussian filtered* image. In order to be independent from the distance-to-object, processing is repeated over a series of images representing scaled down original image, the *pyramid*. A detailed description of the algorithm can be found in section 4.1.

As a preparation step, the ORB application has been transformed from the floating-point version to the fixed-point suitable for an embedded implementation.

```
1   static unsigned int   n_levels = 8;   // RO
2   static size_t         n_features = 500;
3   ...
4   int                 * n_features_per_level;         // to be transformed
5   ...
6   main (int argc, char **argv) {
7     char * scene_obj = argv[1];
8     char * scene_db = argv[2];
9     Image_t img;
10    Descr_t descr_db;
11    Match_t match_db;
12    Point_t * keypoints = (Point_t*)malloc(n_levels*sizeof(Point_t));
13    orb_init(&img, scene_obj, &descr_db, scene_db, &match_db);
14    orb_run(&img, keypoints, &descr_db);
15    match (&descr_db, &match_db);
16    ... show results ...
17    orb_deinit(&img, keypoints, &descr_db, &match_db);
18  }
```

Listing 2.3: Extract from the reference ORB application

2.4.1 Identification of the dataflow graph

Listing 2.3 shows the reference code of the ORB main function. The ORB `main` function (line 6 in the listing) receives the names of the image to process and of the objects database as arguments. Inside the `main` function, the `orb_init` loads the input image from a file; loads the trained objects database initializing the `match_db` for matching image objects vs the database objects; and initializes some global parameters. The `orb_run` computes the keypoints and the object descriptors. The `match` function compares the `descr_db` vs the `match_db` classifying the objects found inside the input image. Finally, the `orb_deinit` releases resources allocated during the processing.

The very first step for transforming this code is to identify the part of the code which will become a dataflow graph. We will focus on the `orb_run` function, shown in Listing 2.4 where the compute intensive processing is required. The `match` function, accounting for about half of the processing requirements of the application, is another good candidate but is more communication than compute bound. In our actual implementation, the `orb_run` and the `match` have been implemented as two separate dataflow graphs. In order to simplify the illustration, we do not include building the scaled image pyramid (lines 2-3 inside the `orb_run` function) as part of the graph. Thus, we assume that the pyramid has been built during a pre-processing

```
1   void orb_run (Image_t img, Point_t * keypoints, Descr_t * descriptors) {
2     Image_t * img_pyramid = (Image_t *) malloc(n_levels*sizeof(Image_t));
3     ... compute re-scaled image pyramid from the img ...
4     computeKeyPoints(img_pyramid, keypoints);
5     for (level = 0; level < n_levels; ++level) {
6       Point_t * keyp = &keypoints[level];
7       computeOrientation(level, &img_pyramid[level], keyp);
8       Descr_t * descr = &descriptors[level];
9       Image_t * blur_img = (Image_t *) malloc(sizeof(Image_t));
10      computeGaussianFilter(level, &img_pyramid[level], blur_img, ...);
11      computeDescriptors(level, blur_img, keyp, desc);
12      free (blur_img);
13    }
14    free (img_pyramid);
15  }
```

Listing 2.4: Extract from the reference ORB application

step and the results have been stored in external memory [2]. We divide the ORB application into two parts: (1) the *main part* which takes care of the input and output, user interactions, allocating and freeing resources, etc., and (2) the *dataflow part* which corresponds to the compute intensive part of the application. This dataflow part is executed as a dataflow graph under the control of the StreamDrive dataflow scheduler.

Figure 2.5 shows the transformed `orb_run` function from the main part of the application. The ORB computation loop has been replaced with calls to new `Build_Graph`, `Exec_Graph`, and `Term_Graph` functions for building, executing and releasing the dataflow graph, respectively. The `Build_Graph` takes an application-specific structure as a parameter used to pass construction time arguments to the graph such as the pointer at the image pyramid, number of pyramid levels, image dimensions, etc. A dataflow graph, once built, can be executed multiple times, for example looping over several input images. In this case, the `Exec_Graph` parameter can be used to pass different execution parameters for each graph execution.

In StreamDrive successive refinement approach, we first build the initial KPN graph. The initial KPN graph in Listing 2.5 contains a single actor, `ORB`. The `STREAM_DECL_SW_ACTOR` macro declares a software actor, with the last parameter specifying how much runtime stack room this actor needs for execution. The ORB computation code from `orb_run` has been simply moved to `ORB` actor's `WORK` function. The `ORB` actor is instantiated via the `STREAM_ACTOR_MAKE` macro inside the `Build_Graph` function. The `Exec_Graph` function configures the dataflow graph by enabling its single actor. It also sets actor's priority and the timeout for the graph execution. Setting the timeout launches the deadlock detection mechanism, which interrupts the execution in the case the specified time

2 In actual implementation, we have implemented two variants of the ORB: (1) with a rescaler tightly-coupled HW block and where the pyramid construction is part of the dataflow graph, and (2) with the pyramid construction as a pre-processing step.

```
1  STREAM_DECL_SW_ACTOR(ORB,ORB_ACTOR_PORT_COUNT,2048);
2
3  static ORB_t * orbActor;
4  GlobalParam_t cfg;
5
6  int32_t Build_Graph (GraphBuild_t * arg) {
7    cfg = <initialize from arg>
8    orbActor = STREAM_ACTOR_MAKE(ORB, "orb", NULL);
9  }
10
11 int32_t Exec_Graph (GraphExec_t * arg) {
12   uint32_t timeout = arg->timeout;
13
14   STREAM_ACTOR_ENABLE (orbActor);
15   STREAM_ACTOR_SET_PRIORITY(orbActor, 0);
16
17   STREAM_GRAPH_SET_TIMEOUT (timeout);
18 }
19
20 int32_t Graph_Term () {
21   STREAM_ACTOR_TERM(orbActor);
22 }
23
24 void orb_run (Image_t img, Point_t * keypoints, Descr_t * descriptors) {
25   Image_t * img_pyramid = (Image_t *) malloc(n_levels*sizeof(Image_t));
26   ... compute re-scaled image pyramid from the img ...
27   GraphBuild_t build_parm;
28   GraphExec_t exec_parm;
29   build_parm.img_pyramid = img_pyramid;
30   build_parm.n_levels = n_levels;
31   ...
32   Build_Graph (build_parm);
33   Exec_Graph (exec_parm);
34   Term_Graph ();
35   free (img_pyramid);
36 }
```

Listing 2.5: The `orb_run` function modified to execute under the StreamDrive runtime

has been exceeded. This mechanism dumps the complete information about the dataflow actors states and allows developers easily identify actors involved in the deadlock cycle.

The ORB actor declaration and definition are shown in Listings 2.6 and 2.7.

```
1  typedef struct {
2    int32_t dummy;
3  } orb_t;
4  STREAM_DECLARE_ACTOR_TYPE(ORB,orb_t);
5
6  #define ORB_ACTOR_PORT_COUNT          0
```

Listing 2.6: The ORB actor definition

At this step, identification of the dataflow part of application, there is no dataflow graph *per se* yet. The one actor executes the same sequential code of the original reference algorithm. The single ORB

```
1   STREAM_CONSTRUCTOR (void * arg) {}
2   STREAM_DESTRUCTOR (void) {}
3   STREAM_INIT (void * arg) {}
4   STREAM_WORK () {
5     uint32_t n_levels = cfg->n_levels;
6     Image_t * img_pyramid = cfg->img_pyramid;
7     Point_t * keypoints = cfg->keypoints;
8     Descr_t * descriptors = cfg->descriptors;
9     computeKeyPoints(img_pyramid, keypoints);
10    for (level = 0; level < n_levels; ++level) {
11      Point_t * keyp = &keypoints[level];
12      computeOrientation(level, &img_pyramid[level], keyp);
13      Descr_t * descr = &descriptors[level];
14      Image_t * blur_img = (Image_t *) malloc(sizeof(Image_t));
15      computeGaussianFilter(level, &img_pyramid[level], blur_img, ...);
16      computeDescriptors(level, blur_img, keyp, descr);
17      free (blur_img);
18    }
19    return 0;
20  }
```

Listing 2.7: The ORB actor definition

actor is defined via two files, the `orbActor.h` shown in Listing 2.6, and the `orbActor.c`, Listing 2.7. The `.h` file declares actor's private data structure and actor ports. This actor has neither input, no output ports. The `.c` file defines the actor implementation. Notice that for the initial ORB actor, the CONSTRUCTOR, the DESTRUCTOR, and the INIT functions remain empty, while the work function is a copy-paste of the code from sequential reference. The only slight change from the reference code is that arguments, such as pointers to the `img_pyramid`, to the `keypoints`, etc. are read from the global `cfg` structure. This structure is setup during the dataflow graph construction from `Build_Graph` arguments (line 7 in Listing 2.5).

Notice two important points about our transformed application: (1) apart from little "syntactic sugar", the code changes to the initial sequential version are minimal; (2) the dataflow part already runs under the control of our StreamDrive scheduler and executed in the KPN mode (there are no firing rules yet). It is clear that the process of identification of the main part and the dataflow part of the application is application-specific. The global variables that are used in the dataflow part need to be identified and declared with the `cfg` structure.

2.4.2 Building the initial dataflow graph

As next transformation step, the initial dataflow graph is built: we need to (1) identify sections of code (kernels) which can be transformed into dataflow actors, and (2) introduce the communication channels connecting the actors. This step is facilitated by the fact that streaming applications are typically structured into a sequence

```
1   Keyp_t * fast9_nonmax (int level, Image_t * img, ..., int * n_corners) {
2     int nCorners;
3     Keyp_t * corners = fast9_detect (img, ..., &nCorners);
4     int * scores = fast9_score (corners, nCorners, ...);
5     Keyp_t * nonmax = nonmax_suppress (corners, scores, nCorners, ..., n_corners);
6     return nonmax;
7   }
8
9   void computeKeyPoints (Image_t * img_pyramid, Point_t * keypoints) {
10    ...
11    for (level = 0; level < n_levels; ++level) {
12      Point_t * keyp = &keypoints[level];
13      int cornerCount;
14      Keyp_t * results = fast9_nonmax(level, &img_pyramid[level], ..., &
            cornerCount);
15      keyp->data = (Keyp_t *)malloc(cornerCount * sizeof(Keyp_t));
16      copy (keyp->data, results, cornerCount);
17      computeHarrisResponse(level, img_pyramid[level], keyp, ...);
18      cullKeypoints(level, keyp, n_features_per_level[level], ...);
19      free (results);
20    }
21  }
```

Listing 2.8: The `Compute_Keypoints` function from the `ORB` application

of processing kernels that are a natural choice for dataflow actors. For example, from the Listing 2.7, the `ComputeOrientation`, `ComputeGaussianFilter`, and `ComputeDescriptors` seem to be good candidates for dataflow actors. The `ComputeKeyPoints`, shown in Listing 2.8 is in turn composed of suitable computation kernels, the `fast9_detect`, `fast9_score`, `nonmax_suppress`, `ComputeHarrisResponse`, and `CullKeypoints`. These will be our initial choice of the dataflow actors.

The channel introduction requires identifying, for the *input channels*, of the data that are read by this actor but written outside of it, and for the *output channels*, the data that are written by the actor and read elsewhere. This remains a manual task, although tools, such as Sprint [154] may be considered in future work. Once again, due to the kernel structure of the streaming applications, the channel introduction turns often to be relatively straightforward. From the ORB example, the `fast9_detect` kernel takes one image from the image pyramid in its input channel, and produces the array of `cornerCount FAST` keypoints and the `fast9_score` computes their scores. The `nonmax_suppress` takes the keypoints generated by the `fast9_detect` with scores as input and generates the set of corner keypoints by removing "uninteresting" keypoints from the set. The `ComputeHarrisResponse` reads these corners and computes the *Harris response* for each of them, which is a measure of "relevance" of each keypoint. The `ComputeHarrisResponse` output is the set of keypoints with their associated Harris response. The `CullKeypoints` performs the sorting of the keypoints with respect to their Harris response and reduces the keypoint set further by retaining at most the `n_features_per_level` best keypoints. For these remaining keypoints, the `ComputeOrientation` computes each keypoint's orien-

tation. The `ComputeOrientation` has two input channels, the keypoints sorted by the `CullKeypoints` and the scaled input image from corresponding image pyramid level. The output of the `ComputeOrientation` is a set of keypoints with their associated orientation measure. The `ComputeDescriptors` takes two input channels as well, the output keypoints from the `ComputeOrientation` and the Gauss filtered input image. The `ComputeDescriptors` output is the final set of keypoints and their descriptors.

```
1   typedef struct {
2     uint32_t        _cFastThreshold;
3     Image_t       * _img_p;
4   } fast_t;
5   STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);
6
7   #define FAST_IN_ESIZE             (sizeof(Image_t))
8   #define FAST_OUT_ESIZE            (sizeof(Keyp_t))
9   #define FAST_PORT_IN              0
10  #define FAST_PORT_OUT             1
11  #define FAST_PORT_COUNT           2
```

Listing 2.9: Initial `FAST` actor definition

```
1  void fast9_detect (Image_t * img, ..., int * num_corners) {
2      int xsize = img->width;
3      int ysize = img->height;
4      //int rsize=512;
5      Keyp_t * header = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
6      *num_corners = 0;
7      //Keyp_t * corners = (Keyp_t*)malloc(sizeof(Keyp_t)*rsize);
8      for (y = edge_threshold; y < ysize - edge_threshold; y++) {
9        for (x = edge_threshold; x < xsize - edge_threshold; x++) {
10         ... compute keypoint or not ...
11         if (corner) {
12           Keyp_t * token = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
13           //if (*num_corners == rsize) {
14    //  rsize*=2;
15    //  corners = (Keyp_t*)realloc(corners, sizeof(Keyp_t)*rsize);
16    //}
17           //corners[*num_corners] = *corner;
18           *token = *corner;
19    *num_corners++;
20         }
21       }
22     }
23     header->... = *num_corners;
24     STREAM_OUT_PUSH(FAST_PORT_OUT, *num_corners+1);
25     return;
26   }
27
28   STREAM_CONSTRUCTOR (void * arg) {
29     STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p", FAST_IN_ESIZE);
30     STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p", FAST_OUT_ESIZE);
31   }
32   STREAM_DESTRUCTOR (...) {
33     STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);
34     STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);
35   }
36   STREAM_INIT() {}
37   STREAM_WORK() {
38     int16_t fastThreshold = THIS->_cFastThreshold;
39     uint8_t n_levels = cfg->n_levels;
40
```

```
41      for (uint32_t level = 0; level < n_levels; ++level) {
42        int cornerCount;
43        THIS->_img_p = (Image_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
44        fast9_detect(level, &img_pyramid[level], ..., &cornerCount);
45        STREAM_IN_RELEASE (FAST_PORT_IN, 1);
46      }
47    }
```

Listing 2.10: Initial FAST actor definition

In order to introduce new actors, each processing kernel needs to be *wrapped* into the StreamDrive syntactic structure similar to the earlier ORB actor. Listings 2.9 and 2.10 show, as example, the FAST actor corresponding to the `fast9_detect` kernel. Inside the actor `.h` file, actor ports are described (their ids and token sizes). The FAST input tokens are of type `Image_t` and output tokens are keypoints of type `Keyp_t`. Inside the `.c` file, the actor constructor and destructor functions create and destroy, respectively, the actor ports. The change to the original `fast9_detect` function is again minimal and consists in inserting the StreamDrive communication primitives. Thus, the `STREAM_IN_POP` and the `STREAM_IN_PRELEASE` are called once for each image in the image pyramid. The `STREAM_OUT_RESERVE` is called for every new keypoint and all keypoints are *pushed* to the output channel via the `STREAM_OUT_PUSH`. In order to communicate the number of keypoints to the downstream actor, the FAST *reserves* one *header* token at the beginning of the processing. When the number of keypoints is known at the end of the outermost loop, the *header* is pushed to the output channel together with the keypoints [3]. It is interesting to notice that using the fixed size dataflow buffers allows us to get rid of costly dynamic memory allocation, commented lines in the listing.

We build the initial dataflow graph incrementally, adding one actor at a time, verifying the transformation correctness at each new actor. The process proceeds in the topological actor order, for example starting with first actor, FAST shown above. Until all actors have been added, the initial ORB actor keeps playing the placeholder role for the remaining of the graph. At the end of the process, the initial ORB actor is no longer needed and is removed. Listing 2.11 shows the initial StreamDrive graph `Build_Graph` function with eight actors. At this step we have not addressed the memory size and actor granularity issues, therefore all FIFO buffers have been allocated in large off-cluster memory using an appropriate `POOL` level.

```
1   STREAM_DECL_SW_ACTOR(SRC,SRC_ACTOR_PORT_COUNT,1024);
2   STREAM_DECL_SW_ACTOR(GAUSS,GAUSS_ACTOR_PORT_COUNT,1024);
3   STREAM_DECL_SW_ACTOR(FAST,FAST_ACTOR_PORT_COUNT,1024);
4   STREAM_DECL_SW_ACTOR(NONMAX,NONMAX_ACTOR_PORT_COUNT,1024);
5   STREAM_DECL_SW_ACTOR(HARRIS,HARRIS_ACTOR_PORT_COUNT,1024);
6   STREAM_DECL_SW_ACTOR(CULL,CULL_ACTOR_PORT_COUNT,1024);
7   STREAM_DECL_SW_ACTOR(ANGLE,ANGLE_ACTOR_PORT_COUNT,1024);
8   STREAM_DECL_SW_ACTOR(BRIEF,BRIEF_ACTOR_PORT_COUNT,1024);
9
```

3 any field of the `Keyp_t` structure can be used to communicate the number of corners.

```
10  static SRC_t    * srcActor;
11  static GAUSS_t  * gaussActor;
12  static FAST_t   * fastActor;
13  static NONMAX_t * nonmaxActor;
14  static HARRIS_t * harrisActor;
15  static CULL_t   * cullActor;
16  static ANGLE_t  * angleActor;
17  static BRIEF_t  * briefActor;
18
19  static stream_bind_t * broadcast;
20  static stream_bind_t * b0;
21  static stream_bind_t * b1;
22  static stream_bind_t * b2;
23  static stream_bind_t * b3;
24  static stream_bind_t * b4;
25  static stream_bind_t * b5;
26
27  GlobalParam_t cfg;
28
29  int32_t Build_Graph (GraphBuild_t * arg) {
30    cfg = <initialize from arg>
31
32    srcActor = STREAM_ACTOR_MAKE(SRC, ''src'', NULL);
33    gaussActor = STREAM_ACTOR_MAKE(GAUSS, "gauss", NULL);
34    fastActor = STREAM_ACTOR_MAKE(FAST, "fast", NULL);
35    nonmaxActor = STREAM_ACTOR_MAKE(NONMAX, "nonmax", NULL);
36    harrisActor = STREAM_ACTOR_MAKE(HARRIS, "harris", NULL);
37    cullActor = STREAM_ACTOR_MAKE(CULL, "cull", NULL);
38    angleActor = STREAM_ACTOR_MAKE(ANGLE, "angle", NULL);
39    briefActor = STREAM_ACTOR_MAKE(BRIEF, "brief", NULL);
40
41    broadcast = STREAM_MAKE_BROADCAST(4, SRC_OUT_DEPTH*sizeof(Image_t), MEM_EXT);
42    STREAM_BIND_OUT_TO_BROADCAST (srcActor, SRC_PORT_OUT, broadcast);
43    STREAM_BIND_BROADCAST_TO_IN (broadcast, fastActor, FAST_IN_PORT, 0);
44    STREAM_BIND_BROADCAST_TO_IN (broadcast, harrisActor, HARRIS_IN_PORT, 1);
45    STREAM_BIND_BROADCAST_TO_IN (broadcast, angleActor, ANGLE_IN_PORT, 0);
46    STREAM_BIND_BROADCAST_TO_IN (broadcast, gaussActor, GAUSS_IN_PORT, 0);
47
48    b0 = STREAM_MAKE_BUFFER (FAST_OUT_DEPTH*FAST_OUT_ESIZE, MEM_EXT);
49    STREAM_BIND_OUT_TO_IN (fastActor, FAST_PORT_OUT, nonmaxActor, NONMAX_PORT_IN,
          b0);
50
51    b1 = STREAM_MAKE_BUFFER (NONMAX_OUT_DEPTH*NONMAX_OUT_ESIZE, MEM_EXT);
52    STREAM_BIND_OUT_TO_IN (nonmaxActor, NONMAX_PORT_OUT, harrisActor,
          HARRIS_PORT_IN, b1);
53
54    b2 = STREAM_MAKE_BUFFER (HARRIS_OUT_DEPTH*HARRIS_OUT_ESIZE, MEM_EXT);
55    STREAM_BIND_OUT_TO_IN (harrisActor, HARRIS_PORT_OUT, cullActor, CULL_PORT_IN,
          b2);
56
57    b3 = STREAM_MAKE_BUFFER (CULL_OUT_DEPTH*CULL_OUT_ESIZE, MEM_EXT);
58    STREAM_BIND_OUT_TO_IN (cullActor, CULL_PORT_OUT, angleActor, ANGLE_PORT_IN, b3
          );
59
60    b4 = STREAM_MAKE_BUFFER (ANGLE_OUT_DEPTH*ANGLE_OUT_ESIZE, MEM_EXT);
61    STREAM_BIND_OUT_TO_IN (angleActor, ANGLE_PORT_OUT, briefActor, BRIEF_PORT_IN,
          b4);
62
63    b5 = STREAM_MAKE_BUFFER (GAUSS_OUT_DEPTH*GAUSS_OUT_ESIZE, MEM_EXT);
64    STREAM_BIND_OUT_TO_IN (gaussActor, GAUSS_PORT_OUT, briefActor, BRIEF_PORT_BLUR
          , b5);
65  }
```

Listing 2.11: The `Build_Graph` function that constructs the initial ORB graph.

Figure 2.6 draws the resulting ORB graph graphically. Notice that input image is read from off-cluster memory via a special `SRC` actor and is broadcast to several input ports: the `FAST`, the `GAUSS`, the `HARRIS`, and `ANGLE` input ports. This image initiates inside the external memory and needs to be copied from this external memory to the cluster shared memory for processing. The SRC actor uses the StreamDrive DMA API to launch requests for transferring input image data to dataflow buffer connected to the input data channels. The `SRC` actor does not have input ports and has one output port which data from the DMA transfer are sent to. The corresponding FIFO buffer is shared between several destination actors via StreamDrive broadcast connection. The `BRIEF` output port that writes final descriptors out to memory, does not have a matching port to connect to. Result data from the `BRIEF` actor are transferred to the off-cluster memory via the DMA API calls inside the `BRIEF` actor: there is no need to add a special DMA actor for this.

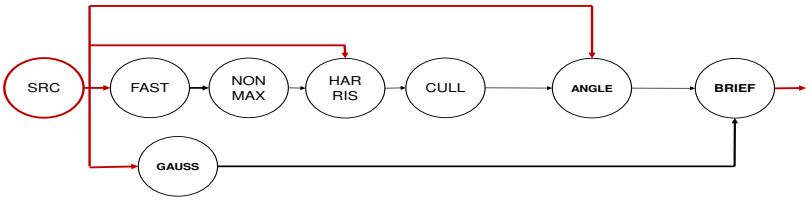


Figure 2.6: Initial ORB dataflow graph: the actors correspond to the original kernels; the input image data are read via the DMA by the `SRC` actor and are broadcast to the `FAST`, `GAUSS`, `HARRIS`, and `ANGLE` actors; the `BRIEF` writes result descriptors directly to the external memory.

At the end of this step, the initial dataflow graph is built with several dataflow actors identified. Following important points facilitate this transformation step: (1) the actor *granularity of execution* of the original application has been preserved; (2) we have avoided dealing with limited memory constraints by allocating all communication buffers in sufficiently large external memory; (3) the actor execution order corresponds to that of the original application because we have preserved the sequential code *granularity* and dependencies.

2.4.3 The dataflow graph refinement

The dataflow actor *granularity* refers to the amount of data that the actor needs for executing without being blocked, and is directly related to the size of actor input and output *tokens*. In previous transformation step, actors keep processing at the original reference algorithm *granularity*, i.e. the entire input image, full set of keypoint, etc. These data are to large to fit in cluster shared memory and are allocated externally. The next transformation step, the dataflow graph

refinement, reduces actor *granularity* so that dataflow communication buffers fit with limited cluster TCDM memory.

# Actor	Port	Initial token size	Refined token size
FAST	IN	One image	One image line
	OUT	All FAST keypoints	One keypoint
NONMAX	IN	All FAST keypoints	One keypoint
	OUT	All nonmax keypoints	One keypoint
HARRIS	IN	All nonmax keypoints	One keypoint
	REF	One image	One image line
	OUT	All nonmax keypoints	One keypoint
CULL	IN	All nonmax keypoints	One keypoint
	OUT	All sorted keypoints	One keypoint
ANGLE	IN	All sorted keypoints	One keypoint
	REF	One image	One image patch
	OUT	All sorted keypoints	One keypoint
GAUSS	IN	One image	One image line
	OUT	One image	One image line
BRIEF	IN	All sorted keypoints	One keypoint
	BLUR	One image	One image patch
	OUT	All descriptors	One descriptor

Table 2.3: Granularity of actors in the ORB application.

Table 2.3 shows refined token sizes for the ORB graph actors. Notice that we have chosen to fetch the `ANGLE` and the `BRIEF` image data one *patch* at a time: a patch is a small area around each keypoint. Because patches for different keypoints may overlap, we end up fetching same image pixels multiple times. However, the alternative of keeping the keypoints in raster scan order and fetching reference image line by line led to poorer performance.

Choosing actor granularity represents an important trade-off: finer granularity reduces the actor memory footprint while increasing the synchronization overhead [4]; coarser granularity suffers little synchronization overhead but may require too much memory. Although granularity vs. performance is an application-specific trade-off, the parallelization should preserve application's *natural* granularity. In this context *natural* means as close to the intrinsic algorithm structure as possible. In an image processing application, choosing one image line as a dataflow token is natural because it corresponds to the second level in the image processing nested loop: (1) frame, (2) line, (3) pixel. As an alternative, sets of lines, tiles, or similar, are less *natural* in a sense that they are algorithm-specific, require some non-intuitive changes to the initial application code, and result is often radically different from the sequential algorithm.

```
1  typedef struct {
2    uint32_t   cFastThreshold;
3    uint8_t *  line_p[3];
4  } fast_t;
5  STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);
6
7  #define FAST_IN_ESIZE             (sizeof(Line_t))
```

4 The synchronization overhead includes actions required to verify the token availability, and the associated scheduler actions

```
8   #define FAST_OUT_ESIZE            (sizeof(Keyp_t))
9   #define FAST_PORT_IN              0
10  #define FAST_PORT_OUT             1
11  #define FAST_PORT_COUNT           2
```

Listing 2.12: The FAST actor KPN definition.

```
1   STREAM_CONSTRUCTOR (void * arg) {
2     STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p", FAST_IN_ESIZE);
3     STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p", FAST_OUT_ESIZE);
4   }
5   STREAM_DESTRUCTOR (...) {
6     STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);
7     STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);
8   }
9   STREAM_INIT(void * arg) {}
10
11  void fast9_detect (int xsize, Line_t *line[3], ..., int * num_corners) {
12    *num_corners = 0;
13    for (x = edge_threshold; x < xsize - edge_threshold; x++) {
14      ... compute keypoint or not ...
15      if (corner) {
16        Keyp_t * token = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
17        *token = *corner;
18  *num_corners++;
19      }
20    }
21    return;
22  }
23
24  STREAM_WORK() {
25    int16_t fastThreshold = THIS->_cFastThreshold;
26    uint8_t n_levels = cfg->n_levels;
27
28    Keyp_t * header = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
29
30    for (uint32_t level = 0; level < n_levels; ++level) {
31      int xsize = cfg->img_width[level];
32      int ysize = cfg->img_height[level];
33      int cornerCount = 0;
34
35      // Build FAST window
36      for (i = 0; i < 3; i++) {
37        THIS->line_p[i] = (Line_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
38      }
39
40      for (y = edge_threshold; y < ysize - edge_threshold; y++) {
41        int count;
42        fast9_detect (level, THIS->line_p, ..., &count);
43        cornerCount += count;
44        // Rotate FAST window
45        STREAM_IN_RELEASE (FAST_PORT_IN, 1);
46        for (i = 0; i < 2; i++) {
47          THIS->line_p[i] = THIS->line_p[i+1];
48        }
49        THIS->line_p[2] = (Line_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
50      }
51
52      STREAM_IN_RELEASE (FAST_PORT_IN, 2);
53
54      header->... = cornerCount;
55      STREAM_OUT_PUSH(FAST_PORT_OUT, cornerCount+1);
56    }
57  }
```

Listing 2.13: The FAST actor KPN definition.

Refining actors' granularity requires changing its WORK function. Listings 2.12 and 2.13 show the FAST actor with refined input granularity: the input token corresponds to one image line instead of the entire image. Since the fast9_detect works on three lines at a time, we pass it a window of three lines, THIS->line_p, which is rotated on every iteration of the WORK function. The STREAM_IN_POP and the STREAM_IN_RELEASE are applied to one image line instead of one full image.

Once the granularity of the actors has been reduced, the communication channels can be moved to the cluster shared memory. However, some channels may need to buffer too many tokens to fit with this memory. For example, the SRC actor transfers the input image from the external to cluster memory one line at a time. While the FAST and the HARRIS actors consume these lines also one at a time, the ANGLE actor cannot start consuming the input image lines until the entire image has been seen and processed by the CULL actor. Therefore, the communication channel needs to buffer the entire image and is, thus, too big to fit the cluster shared memory. Similarly, the BRIEF actor can start consuming the blurred image produced by the GAUSS actor only after the CULL actor has generated the sorted list of interesting keypoints. In such cases, the communication channel buffer is allocated in the external off-cluster memory. Such excessive buffering requirements do not result from the actor granularity but from the necessity to accumulate large number of tokens in a communication channel before the first token can be consumed. The ANGLE actor, simply needs re-reading the input image from external memory after the the CULL actor has finished processing. This is done by the second SRC actor and a communication FIFO holding input image patches to feed the ANGLE actor. The situation is more subtle with the BRIEF actor. The GAUSS actor would write the blurred image to the buffer in off-cluster memory, while the BLUR DMA actor would read it in patches when the BRIEF needs it. The GAUSS needs to be able to tell the BLUR actor when the blurred image has been completely written out to the external buffer. We acomplish this by using special *sync* input and output ports. The *sync* ports are similar to the normal StreamDrive ports but do not have any FIFO buffers associated with them (the *sync* API is described in Appendix A). The refined ORB dataflow graph is shown in Figure 2.7.

The refinement transformation step enables parallel execution for the first time: actors can execute their work-functions in parallel, synchronizing at *reserve* and at *pop* points.

2.4.4 Adding application-specific hardware blocks

Before further optimization and introduction of the firing rules, it is convenient to perform software-hardware partitioning at this point.

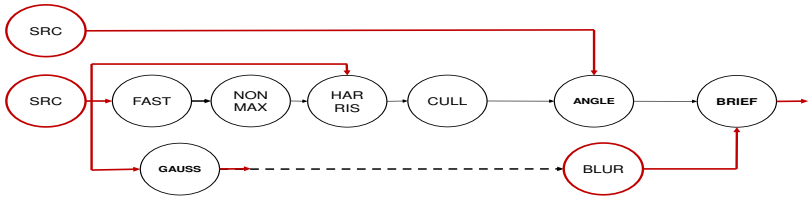


Figure 2.7: Refined ORB dataflow graph: most communication buffers have been moved to cluster shared memory. The `SRC` to `ANGLE`, and the `GAUSS` to `BRIEF` require bufferring capacity that exceeds the cluster memory, these data need to be stored externally. Two additional DMA actors, the second `SRC` and the `BLUR`, are used to handle this situation.

For example, Table 2.4 shows the breakdown of ORB kernel's execution time from the original application (first image pyramid):

Kernel	Execution Time, Mcycles
fast9_detect	3,77
fast9_score	0,39
nonmax_suppress	0,32
ComputeHarrisResponse	0,81
CullKeypoints	0,34
ComputeOrientation	0,55
ComputeGaussFiltering	7,66
ComputeDescriptors	2,09
Total	15,93

Table 2.4: The ORB execution time breakdown.

The Gaussian filtering kernel largely dominates the application execution time and, considering that filtering is a quite common function in image processing, is a good candidate for being implemented as an application-specific hardware block. Such application-specific hardware blocks are designed independently from the application and need to implement a streaming data interface (see Section 2.2 for a description of the hardware block integration within the shared memory cluster). In Section 3.3, we describe an application-specific hardware block for performing convolution that we designed using the high-level synthesis techniques.

Integrating application-specific hardware blocks with a StreamDrive application does not require changing the dataflow graph. Instead, it is sufficient to instantiate actor as a hardware block instead of as a software actor. The StreamDrive runtime will transparently handle the hardware block actor during the execution.

2.4.5 Data Parallelism

The above transformation steps build a dataflow graph by identifying and exposing the *functional parallelism*, where actors form execution pipeline over the input stream of data. The *functional parallelism* is inherent with the dataflow model of execution and can be exploited

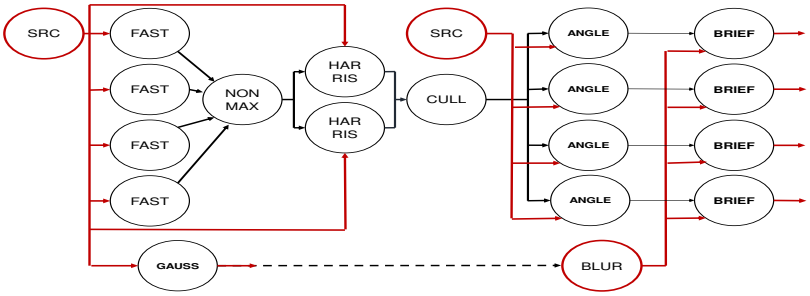


Figure 2.8: The ORB dataflow graph with data-parallel actors: the `FAST`, the `ANGLE` and the `BRIEF` actors are replicated 4 times, and the `HARRIS` actor is replicated 2 times.

by splitting and merging dataflow actors and by varying actor's granularity. Another important type of parallelism is the *data parallelism*. In data parallelism, multiple instances of the same actor are simultaneously created. The data parallelism leads to efficient execution when the computations are not data dependent and regular: (1) it is easy to identify and to expose, (2) it has lower parallelization overhead compared to the functional parallelism.

In the ORB application, the `fast9_detect`, the `ComputeHarrisResponse`, the `ComputeOrientation`, and the `ComputeDescriptors` kernels are regular and are easy to data parallelize. Parallelizing these kernels into a number of data-parallel instances has several advantages: (1) it balances the workloads of graph actors, and (2) it creates more actors for the scheduler to choose from. For example, from the Table 2.4, the workload of the `fast9_detect`, is few times that of the `nonmax_suppress` or of the `CullKeypoints` kernels, and dividing its workload among several data parallel instances helps balancing the workload of all these actors.

The StreamDrive *broadcast* and *collect* connections help efficiently support the data parallel actors. The *broadcast* enables sharing of the input tokens by the data-parallel actors, while the *collect* allows sharing the output tokens. Using these connections, it is very easy to build data-parallel actors. Two data sharing strategies can be considered:

1. The *single token* data parallel actors work all on the same input or output token, but each on different part of it, for example a different part of an image line.

2. The *multiple token* data parallel actors work each on one of `N` tokens in parallel and synchronize on all `N` tokens simultaneously.

The data sharing strategies apply to individual input or output ports, and therefore it is perfectly possible to mix different data parallel strategies within the same actor, at the same time having channels which do not implement any data parallel sharing.

The Figure 2.8 shows the ORB dataflow graph with data-parallel actors. The *broadcast* connections are used to share input tokens of these actors. The *collect* connections are mostly optimized away, only the

HARRIS actor data-parallel instances use the *collect* connection to share its output tokens. The FAST actor does not use the *collect* connection for its output because the downstream NONMAX actor needs the FAST keypoints to arrive in the raster scan order of the image. However, since the number of the keypoints in each image line is not known in advance, it is impossible for them to share the communication channel. As a solution, the NONMAX actor has N input ports, one for each upstream FAST actor, and reads them in a round-robin order ensuring that the FAST keypoints arrive in the raster scan order of the input image. Similarly, instead of using the *collect* connection for the output of the ANGLE actor, and then re-*broadcast* it to the BRIEF actors, we connect each ANGLE actor directly to the corresponding BRIEF actor, thus gaining efficiency.

The dataflow graph in Figure 2.8 shows the version with 4 FAST, 4 ANGLE, and 4 BRIEF data-parallel instances, as well as 2 HARRIS instances. Our implementation is parameterized in terms of the number of actors, their granularities, and the communication buffer sizes: it can be configured for 1 PE with one instance of each actor up to 8 PEs with 8 instances of the FAST, ANGLE, and BRIEF actors.

To illustrate modifications made to actors for supporting the data-parallelism, Listings 2.14 and 2.15 show the data parallel FAST actor. The new THIS->idx private field corresponds to the index of this data parallel instance among the N data parallel instances. This index is initialized via the actor constructor. The FAST actor implements the *multiple token* data parallel sharing in its input port. The changes to the actor's WORK function are minimal: the actors handle a shared rotating input window of $3 + (N - 1)$ lines instead of 3 lines, while each actor processes only those lines that correspond to this actors' index. The actor output is not changed since every FAST data parallel instance handles its own (not shared) output channel. The fast9_detect function remains unchanged.

```
1   typedef struct {
2     uint8_t    idx;
3     uint32_t   cFastThreshold;
4     uint8_t *  line_p[6];
5   } fast_t;
6   STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);
7
8   #define FAST_IN_ESIZE             (sizeof(Line_t))
9   #define FAST_OUT_ESIZE            (sizeof(Keyp_t))
10  #define FAST_PORT_IN              0
11  #define FAST_PORT_OUT             1
12  #define FAST_PORT_COUNT           2
```

Listing 2.14: Data-parallel version of the FAST actor.

```
1   STREAM_CONSTRUCTOR (void *arg) {
2     uint32_t idx = (uint32_t)arg;
3     STREAM_ACTOR_MAKE_PORT_IN(FAST_PORT_IN, "in_p", FAST_IN_ESIZE);
4     STREAM_ACTOR_MAKE_PORT_OUT(FAST_PORT_OUT, "out_p", FAST_OUT_ESIZE);
5     THIS->idx = idx;
6   }
```

```
7   STREAM_DESTRUCTOR (...) {
8     STREAM_ACTOR_TERM_PORT_IN(FAST_PORT_IN);
9     STREAM_ACTOR_TERM_PORT_OUT(FAST_PORT_OUT);
10  }
11  STREAM_INIT(void *arg) {}
12  STREAM_WORK() {
13    int16_t fastThreshold = THIS->_cFastThreshold;
14    uint8_t n_levels = cfg->n_levels;
15
16    for (uint32_t level = 0; level < n_levels; ++level) {
17      int xsize = cfg->img_width[level];
18      int ysize = cfg->img_height[level];
19
20      Keyp_t * header = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
21
22      int cornerCount = 0;
23
24      // Build FAST window
25      for (i = 0; i < 4+2; i++) {
26        THIS->line_p[i] = (Line_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
27      }
28
29      for (y = edge_threshold; y < ysize - edge_threshold; y++) {
30        if (y % 4 == THIS->idx) {
31          int count;
32          fast9_detect (level, &THIS->line_p[THIS->idx], ..., &count);
33          cornerCount += count;
34
35          // Rotate FAST window
36          STREAM_IN_RELEASE (FAST_PORT_IN, 4);
37          for (i = 0; i < 2; i++) {
38            THIS->line_p[i] = THIS->line_p[i+1];
39          }
40          for (i = 0; i < 4; i++) {
41            THIS->line_p[i+2] = (Line_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
42          }
43        }
44      }
45
46      STREAM_IN_RELEASE (FAST_PORT_IN, 4+2);
47      header->... = cornerCount;
48      STREAM_OUT_PUSH(FAST_PORT_OUT, cornerCount+1);
49    }
50  }
```

Listing 2.15: Data-parallel version of the FAST actor.

It is worth noticing that the StreamDrive offers great flexibility in connecting and synchronizing the data parallel actors. By buffering the input and output tokens, actors data parallel instances do not need to start and stop processing simultaneously, thus benefiting from the efficiency of pipelined execution. Finally, creating a few data parallel actors, we remain within the scope of a small-scale data parallelism (as opposed to the massive data parallelism with hundreds or thousands of parallel instances) matching well with the scale of the target architecture cluster.

2.4.6 Optimizing Scheduling via Firing Rules

Execution of the refined and parallelized dataflow graph can be optimized by introducing dataflow *firing rules*.

In KPN *execution mode*, software actors require the ability to suspend an actor on a blocked *pop* (or *reserve*), and to resume its execution when sufficient tokens (or empty FIFO entries) are available. Suspending and resuming actors implies costly context-switching. In the *dataflow execution mode* the *firing rules* give preconditions for actor execution by ensuring that there are enough input tokens (or room in output FIFOs) for the actor not to be blocked. Thus, dataflow mode allows the context-switch free, *cooperative*, scheduling.

In the dataflow mode, actor's WORK function is subdivided into a sequence of *firings* [45, 155]. During a firing, the actors *reserve* and *pop* tokens similar to the KPN mode, but the *firing rules* ensure that the actor is never blocked during the firing. When a firing is completed, actor returns control to the scheduler without requiring a context switch via the STREAM_YIELD call. The dataflow actor WORK function is "fired" by the scheduler until the STREAM_EXIT call signals the scheduler that actor completed its execution and does not require anymore firings.

Introducing firing rules requires to once more change actor's WORK function. Listings 2.16 and 2.17 show the ORB FAST actor converted to the dataflow mode.

```
1   typedef struct {
2     uint8_t    idx;
3     uint8_t    state;
4     uint8_t    level;
5     uint32_t   cFastThreshold;
6     uint8_t *  line_p[6];
7     Keyp_t * header;
8     uint16_t cornerCount;
9     uint16_t y;
10  } fast_t;
11  STREAM_DECLARE_ACTOR_TYPE(FAST,fast_t);
12  // States
13  #define FAST_IDLE         0   // actor initial state
14  #define FAST_LEVEL        1   // one iteration of the outermost loop
15  #define FAST_LEVEL_END    2   // iteration control
16  #define FAST_LINE         3   // one iteration of the second level loop
17  #define FAST_LINE_END     4   // iteration control
18  // Ports
19  #define FAST_IN_ESIZE             (sizeof(Line_t))
20  #define FAST_OUT_ESIZE            (sizeof(Keyp_t))
21  #define FAST_PORT_IN              0
22  #define FAST_PORT_OUT             1
23  #define FAST_PORT_COUNT           2
```

Listing 2.16: ORB FAST dataflow actor definition.

```
1   STREAM_CONSTRUCTOR (void *arg) { ... }
2   STREAM_DESTRUCTOR (...) { ... }
3   STREAM_INIT(void *arg) {
4     SET_PORT_QUOTA (FAST_PORT_IN, 3+(N-1));
5     SET_PORT_QUOTA (FAST_PORT_OUT, MAX_IMAGE_WIDTH/2);
6     THIS->state = FAST_IDLE;
```

```
7   }
8   STREAM_WORK() {
9     int16_t fastThreshold = THIS->_cFastThreshold;
10    uint8_t n_levels = cfg->n_levels;
11    switch (THIS->state) {
12    case FAST_IDLE:
13      THIS->level = 0;
14      SET_PORT_QUOTA (FAST_PORT_IN, 3+(N-1));
15      THIS->state = FAST_LEVEL;
16      // Fallthrough to LEVEL
17    case FAST_LEVEL:
18      THIS->header = (Keyp_t*)STREAM_OUT_RESERVE (FAST_PORT_OUT, 1);
19      for (i = 0; i < 3-1; i++) {
20        THIS->line_p[i] = (uint8_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
21      }
22      THIS->cornerCount = 0;
23      THIS->y = edge_threshold;
24      SET_PORT_QUOTA (FAST_PORT_IN, N+1);
25      THIS->state = FAST_LINE;
26      break;
27    case FAST_LINE:
28      int xsize = cfg->img_width[THIS->level];
29      int ysize = cfg->img_height[THIS->level];
30      int count;
31      fast9_detect (level, &THIS->line_p[THIS->idx], ..., &count);
32      THIS->cornerCount += count;
33      // Rotate FAST window
34      STREAM_IN_RELEASE (FAST_PORT_IN, N+1);
35      for (i = 0; i < 3-1; i++) {
36        THIS->line_p[i] = THIS->line_p[i+1];
37      }
38      for (i = 0; i < N+1; i++) {
39        THIS->line_p[i+2] = (Line_t*)STREAM_IN_POP(FAST_PORT_IN, 1);
40      }
41      THIS->y += N+1;
42      if (THIS->y < ysize) break;
43      // Fallthrough to LINE_END
44    case FAST_LINE_END:
45      STREAM_IN_RELEASE (FAST_PORT_IN, 3+(N-1));
46      THIS->header->... = THIS->cornerCount;
47      STREAM_OUT_PUSH(FAST_PORT_OUT, THIS->cornerCount+1);
48      THIS->level += 1;
49      if (level < n_levels) {
50        SET_PORT_QUOTA (FAST_PORT_IN, 3+(N-1));
51        THIS->state = FAST_LEVEL;
52        break;
53      }
54      // Fallthrough to LEVEL_END
55    case FAST_LEVEL_END:
56      SET_PORT_QUOTA (FAST_PORT_IN, 0);
57      STREAM_EXIT();
58    }
59    STREAM_YIELD();
60  }
```

Listing 2.17: ORB FAST actor with dataflow firing rules.

The KPN version of the actor from Listings 2.14 and 2.15 consisted of a loop-nest with the outermost level iterating over the images in the image pyramid, the second level over the image lines inside each image, and the innermost level iterating over the image pixels. In order to transform the KPN code to firings, all loops in the loop-nest above the *granularity* level need to be converted to a state machine.

For our FAST actor, we have chosen the second level loop, iterating over input image lines as the granularity level. The conversion is straightforward. The state machine states correspond to the loop-nest levels of the KPN actor: the FAST_IDLE corresponds to the initial state, the FAST_LEVEL and FAST_LEVEL_END to the outermost level loop, and the FAST_LINE and FAST_LINE_END to the second level loop. Before the WORK function yields the control to the scheduler, a transition to the next state needs to be specified by setting the private THIS->state variable. In addition, the firing rules can be given for the next firing via the STREAM_SET_PORT_QUOTA API function. The STREAM_SET_PORT_QUOTA function takes two arguments, the input or output port id and the number of tokens to expect in that port before the firing can take place. The initial state and the initial firing rules can be specified inside the actors' STREAM_INIT function. Notice that by default, unless set by the actor, the firing rules are not set and the actors' *reserve* and *pop* calls become blocking similar to the KPN execution mode.

One important point about converting the graph into the dataflow form is that all variables alive across multiple actor firings need to be saved by the actor before the end of the firing and restored in the next firing. For this, such variables need to be added to actors' private state, similar to local variables level, header, y, and cornerCount from the FAST actor example.

2.4.7 Further refinement and optimization

In an embedded system, the cost of the system and the power consumption are directly related to the system memory size [156], and therefore reducing application memory footprint is important. The dataflow program memory footprint depends on the communication buffers size and is finally related to the actors' granularity. The coarser the actor granularity, the bigger is the memory footprint. On the other hand, when the granularity of a program is very fine, the overhead of the runtime synchronization and communication has a negative impact on efficiency. Thus, the optimization objective consists in finding the best trade-off between the communication buffer sizes and the parallelization overhead.

This step is the most time-consuming of the entire transformation process since the developer needs to choose from many different possibilities leading to different trade-off results. For example, we have noticed that processing the NONMAX, HARRIS, or CULL one keypoint per firing is inefficient because the amount of work per keypoint is small relative to the actor invocation overhead. One possibility that we explored was to combine the three actors together thus creating larger workload per keypoint. While this works well with smaller number of processing resources (less than 4 processing elements), when the number of processing resources increases, the resulting bulky actor

is difficult to efficiently schedule and balance with other actors. On the other hand, several keypoints are usually simultaneously available for processing by the above actors. This allows the above actors process multiple keypoints (tokens) per firing leading to noticeable reduction in parallelization overhead. In StreamDrive, increasing the working set granularity is an optimization task within a well defined reference frame - number of tokens per actor firing. At the same time, preserving tokens *natural* granularity allows optimized application keep algorithmic description close to the original code.

As a general rule, the optimization process should first search for the possibility to combine actors together - this has additional benefits of reducing the overall buffer requirements since intermediate buffers between the combined actors can often be eliminated, and of reducing the schedulers' workload since fewer actors are active in the system. Then, the optimization should work to increase the number of tokens used in actors firings until an acceptable trade-off between the performance and the memory footprint is found.

This section's example illustrates several important points from the StreamDrive:

- The StreamDrive successive refinement flow facilitates parallelization of sequential applications into the DPN implementation. For example, the original `fast9_detect` code incrementally undergoes relatively simple modifications during the transformation process: addition of the StreamDrive communication primitives; using the rotating window of image lines instead of the full image; etc.

- The StreamDrive does not impose any specific language restrictions on reference code in order to be parallelized.

- Unlike standard dataflow implementations, the StreamDrive allows usage of shared global variables. Shared variables are very efficient way of communicating in a shared memory environment and it facilitates porting existing sequential reference code. In our example, the `FAST` actor relies on global `cfg` for retrieving parameters such as image width and height, etc. These parameters are also used by other actors and, instead of duplicating the set of these parameters for each actor, they are implemented as a shared data structure. The coherent use of the shared data remains developer's responsibility.

- The StreamDrive runtime simultaneously supports two execution modes, the KPN and the DPN execution. This is essential for enabling our incremental transformation flow.

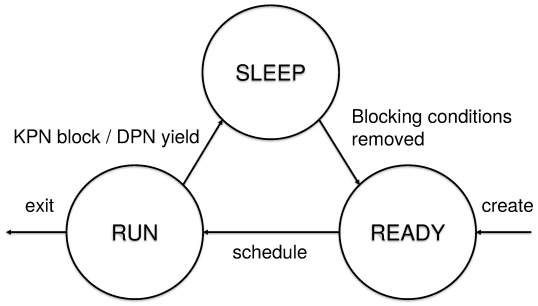


Figure 2.9: Transitions between dataflow actor states. A sleeping actor is unblocked when last KPN blocking condition has been removed or when all firing rules are satisfied.

2.5 THE STREAMDRIVE RUNTIME SYSTEM

The StreamDrive runtime system provides application with a communication layer and the dynamic scheduler. It is implemented as a user-level library avoiding costly system calls and enabling optimization such as inlining function calls, etc.

2.5.1 Actor Management and Scheduling

The actor management and scheduling component (scheduler) orchestrates the whole life-cycle of a dataflow execution: the startup phase, the running phase and the shutdown phase. In the startup phase, the scheduler invokes the user-supplied *dataflow graph construction* routine which creates actors and communication channels. While the dataflow network is running, the scheduler balances the load by choosing which software actors will be executed, when, and on which processing element. Software actors can represent DPN actors with firing rules or KPN processes with blocking conditions. All DPN firing rules and KPN block and unblock operations are managed by the scheduler, so it is able to keep track of the ready actors and shut down the dataflow execution when this number reaches zero. The scheduler also keeps all information necessary for resolution of runtime deadlock. In this section, we present the scheduler's main control-flow, i.e., startup, shutdown and management of the actor's life-cycle.

Figure 2.9 depicts the life-cycle of a StreamDrive actor, both software and hardware-implemented. An actor is created in the ready state where it is eligible to be selected by the scheduler for execution if it is a software actor, or for starting execution if it is a hardware-implemented function. Once selected, it enters the run state from which it can either (1) yield (a DPN actor) or be blocked (KPN actor), entering the sleep state; or (2) exit and release resources associated with it. When a KPN actor (process) is blocked, it is not eligible for

execution until another actor removes the blocking condition (by supplying or consuming the required number of tokens), i.e., an actor cannot unblock itself. After the blocking condition has been removed, the software actor enters the ready state and is again eligible for dispatch, and the hardware-implemented actor continues its execution. When a DPN actor yields control to the runtime scheduler, it is not eligible for execution until all associated firing rules have been satisfied.

The StreamDrive scheduler is cooperative, which means that an actor runs uninterrupted until it yields control to the scheduler, encounters a blocking point, or exits. In our implementation, the only potentially blocking points are currently the *reserve* and the *pop* operations.

StreamDrive scheduler implements a number of *runners* equal to the number of programmable PEs. The scheduler also contains static data shared by all runners, which includes:

- Descriptor tables for all instantiated actors, ports, and channels.
- Synchronization objects for controlling access to critical code sections.
- Immutable data related to deadlock detection mechanism.
- Complete bookkeeping data.
- Detailed accounting information.

The data private to each runner includes an actor queue, a pointer to the currently running actor, a pointer at runner's runtime stack, and the runner accounting information. The actor queue is organized as doubly linked lists of actor instances organized by actor priority levels. The pointer to the currently running actor is needed so that the scheduler can find the actor context when actor yields, blocks or resumes the execution.

The scheduler assigns unique integer identifiers to each newly created actor or communication port, which are created using the StreamDrive graph construction API described earlier. The identifiers start at 0 and increase by 1 for each new actor or port. These IDs are used to map actors and channels to corresponding *actor context descriptor* and *port descriptor* tables. There is one `dsched_acd_t` per actor instance in the current dataflow graph. The `dsched_acd_t` contains actor's scheduler related data and the data needed for user-mode context switch. The *port descriptor* serves as the interface between the user-provided actor code and the runtime communication layer.

2.5.2 Control Flow

The StreamDrive application starts with the *main* routine executed in one of the PEs (typically the PE0). Among other actions, the main

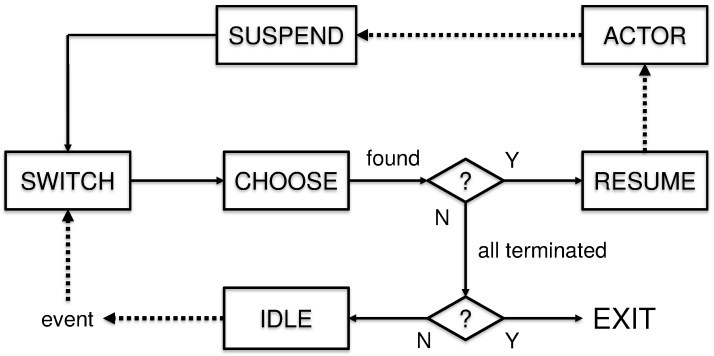


Figure 2.10: Runner control flow. When all actors reach the TERMINATED state, the scheduler exits the loop.

routine invokes the dataflow graph construction and configuration procedures via the StreamDrive graph API. The runners are created during the graph construction procedure, one per programmable PE. Applications may give affinity hints to software actors to tell the scheduler on which subset of PEs each actor will execute. The affinities are set at graph configuration time as well as actor priorities for the priority-based scheduler. This dataflow preparation phase as well as the shutdown phase after the dataflow graph has completed its execution, are executed in the starting PE.

The dataflow graph execution is started by a call to the `STREAM_GRAPH_EXEC` API method, which needs a dataflow graph to have been built and configured. This method reads the number of PEs to use and sets the *bootstrap* context and the current runtime stack pointer for each PE. It also sets up necessary bookkeeping, starts the deadlock detection timer and starts the runner in each active PE. This code section is protected by a single global mutex, and runners are waiting on the associated condition variable during this time. At the end of the setup process, the condition variable is signaled and the mutex is unlocked.

The scheduler is fully distributed with regard to the PEs - there is no one process dedicated to the runtime system duties. Instead, each PE concurrently (1) performs its own scheduling and (2) handles synchronization actions related to the actor being executed by it. As a result, the StreamDrive dataflow scheduler is fully dynamic: it schedules but also assigns actors to PEs dynamically at runtime. The StreamDrive runtime system is still centralized from the point of view of the memory because the runtime system uses a single, global scheduling list.

Each runner has the same startup procedure, which initializes per-runner data structures (the actor queue). Once the global mutex has been unlocked, all runners proceed simultaneously and enter the scheduling loop.

The scheduling loop performs the following steps (see the Figure 2.10):

1. It starts in an initial state [SWITCH].
2. It calls the policy method to choose the next actor to dispatch [CHOOSE]. If the policy method returns NULL, there are no ready processes, so the runner checks whether all actors are in ZOMBIE (terminated execution) state. If so, the scheduler exits the control loop [EXIT]; otherwise the scheduler waits for the synchronization event to be generated [IDLE].
3. The synchronization event can be generated by actors via the *push* or the *release*, when an actor terminates execution, and by the DMA on a data transfer completion. All this events correspond to a change in the dataflow graph, wake up the scheduler, which goes to the initial state [SWITCH].
4. If the policy method finds a ready actor, new actor's state is restored [RESUME].
5. The actor is dispatched on the processing element [ACTOR].
6. The actor has returned control to the scheduler because it has blocked, yielded or exited. The actor's state is saved [SUSPEND].
7. Go to step 1.

An actor may be acquired and executed by at most one runner. In order to ensure this, actors are protected by locking the mutex associated with the actor when the actor is acquired by the policy method [CHOOSE]. The mutex is unlocked when, either the policy method fails to choose this actor, or when actor is suspended, or when actor terminates. The mutex is accessed from runner's context and if the mutex is already taken by another runner, the policy method moves to the next actor in actor queue.

StreamDrive uses a priority-based scheduler where higher priority actors are selected for execution before actors of lower priority, which are dispatched when all higher-priority actors are sleeping. Contention among processes of equal priority is resolved by a round-robin policy. Actor priorities are application-specific and statically assigned at dataflow graph construction time.

All runners share a counter that reflects the total number of active runners. The counter is set to the total number of PEs in the startup procedure, and it is decremented every time a runner is exiting the scheduler loop. The counter decrementing is also protected by a special mutex. When all runners exit their scheduler loops, the main PE invokes the dataflow graph termination procedure. Among others,

the graph termination procedure checks that all actors and the synchronization counters have been left in a consistent state (for example, no tokens remain un-consumed in the communication channels), cleans up the global state, and writes the collected accounting data to the stream given as the argument, default being standard output.

2.5.3 User-Mode Context Switch

Context-switch is a mechanism that enables time-multiplexing of several KPN actors (processes) over one runtime runner. Context-switch is usually performed by the OS kernel as part of scheduling threads. However in order to gain efficiency or predictability, we have implemented our own user-mode context switch mechanism optimized for our platform programmable cores. The context-switch is written in target architecture assembly language. The actor's context is stored as the `dsched_ctx_t` structure shown in Figure 2.11. The register area stores registers necessary to resume the actor after it has been suspended. Along with stack pointer and program counter registers, we store only those registers that the calling convention mandates to be saved across function calls. When a context switch occurs, the actors' register context is saved to a location in the off-cluster memory. Together with saving actors' register context, the current runtime stack is also *spilled* to the actors' external stack location pointed at by the `stack` field in the `dsched_ctx_t` data structure.

As explained earlier, StreamDrive supports two execution modes, the DPN and the KPN. In the DPN mode, actors *run-to-completion* and therefore all actors can share a single runtime stack per PE during the execution. These runtime stacks can be fixed for the entire execution because the stack size for each software actor is specified in actor declaration and is therefore known at compile-time for all actor instances. These runtime stacks are also reasonably small to fit inside the shared cluster memory because typical signal processing actors do not use recursion, and large stack-allocated variables are easily avoided. Additionally, the stack hungry print and file manipulation routines are not typically used with high-performance embedded code [5]. Thus, in practice a small stack of 1-2KB per actor is largely sufficient.

In the KPN mode, actor execution can be suspended on a blocking condition (they do not *run-to-completion*) - therefore each actor requires its own dedicated runtime stack. Placing too many actor stacks inside the cluster memory raises an important difficulty because this memory is relatively small. For example, given a stack size of 2K per actor and a relatively small number of actor instances, eg. 20, the total

5 During the development, it is often convenient to use the printing routines even though they are usually removed from the final performance version. In this case, bigger stack can be allocated and placed in larger off-cluster memory with an associated performance penalty.

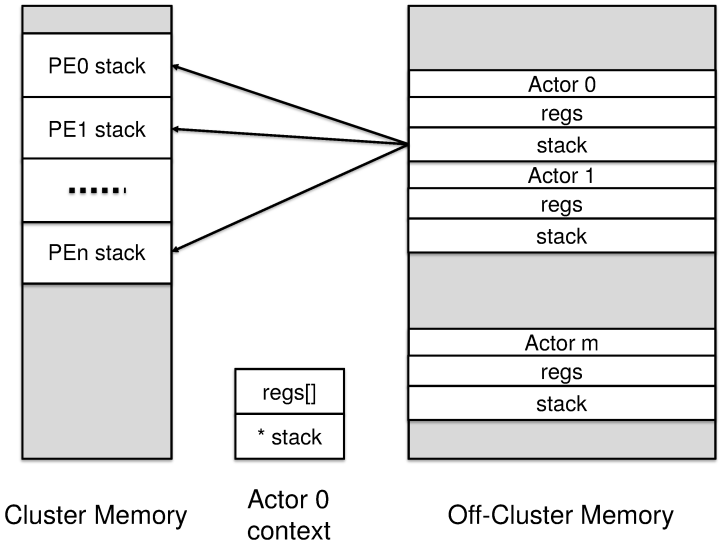


Figure 2.11: Actor context descriptor and stack spilling.

stack memory requirement would be 40KB, a heavy overhead for a relatively small cluster memory. But most importantly, the number of actors would have to be considered along with the dataflow communication buffer sizes and other memory related trade-off parameters. Alternatively, the individual actor stacks can be allocated in the larger off-cluster memory. However, placing the runtime stack inside the external memory, with long access latency, leads to an inefficient, low performance execution.

We address this difficulty by implementing a novel stack *spilling* strategy. For this, we allocate one runtime stack per PE inside the shared cluster memory. This runtime stacks are called *current runtime stack* and are used during execution in both execution modes (the total number of these stacks is independent of the number of application actors). The individual actor stacks are also allocated in the larger off-cluster memory and serve as placeholders for saving sleeping KPN actors' stacks. When a KPN actor gets blocked during the execution, together with saving actors' register context, the current runtime stack is saved or *spilled*. When the blocked actor resumes execution, its register context is restored, and also its stack content is loaded to current runtime stack from the external stack location. In the DPN execution mode, actors do not get blocked and no context switch and stack spilling are necessary.

There are two consequences to the stack spilling. First, KPN actor assignment to PEs cannot be changed when actor is resumed after having been suspended. Indeed, because different PE's runtime stacks point at different addresses in the cluster shared memory, a suspended actor can only resume its execution in the same PE (address space) where it has been suspended. As a result, each KPN actor keeps execution in the same PE where it has begun its execution.

Second, using the above stack-spilling strategy increases the cost of a context-switch: in addition to usual saving and restoring registers, the stack contents need to be saved and restored as well. In order to alleviate the problem, StreamDrive optimizes the KPN execution as follows: (1) our runtime scheduler minimizes the context-switch occurrences, (2) we have implemented optimized, hand-crafted code for the context switch routines. Moreover, in our experiments we have observed that, typically, the total number of bytes of stack that need to be spilled is quite small, and penalty for stack spilling is comparable to that of register context switching. In general, we consider the KPN execution as an intermediate step during the incremental transformation process from a sequential reference code into a DPN implementation with no context switching.

2.5.4 Communication Layer

In the dataflow model, the FIFO channels are the only means of communication between actors. The communication layer must not only fulfill the dataflow semantics, but it should also be efficient.

The StreamDrive communication layer defines three types of communication channels: `s_buffer_t`, `s_broadcast_t`, `s_collect_t`. The communication channels use circular buffers to store tokens. All three types are hidden from the developer and are represented by opaque handles. In addition to the standard dataflow communication *buffer* type, we have extended communication channels with support for the *broadcast* and the *collect* functionality. The *broadcast* allows a single communication channel to be shared by several destination dataflow actors. Sharing the channel removes the necessity to *copy-forward* tokens to all destinations. The important operation in the *broadcast* channel is *release*. The *broadcast* needs to ensure that all destination actors have released the token before the release can be seen by the source actor. Similarly, the *collect* allows a single communication channel to be shared by several source dataflow actors. Sharing the channel removes the necessity to *copy-join* tokens from all sources. The *collect* needs to ensure that all source actors have *push*ed the token before the push can be seen by the destination actor. The *broadcast* release and the *collect* push operations are internally supported by the StreamDrive communication layer. Therefore, there is no need to schedule these operations - the runtime knows when the *release* or the *push* needs to be executed. For example, a *broadcast* release will only be executed if the broadcasting output port is blocked on a FIFO full condition. Such dedicated support to the *broadcast* and the *collect* connections ensures optimal runtime execution.

StreamDrive actors do not access the communication channels directly; they define instead instances of ports, which ensure unidirectional communication. Four types of communication ports are

defined: `stream_out_t`, `stream_in_t`, `sync_out_t`, and `sync_in_t`. The *port descriptor* structure holds common bookkeeping data, independent of what type of port it is, and some port-specific fields. The common part makes it possible to treat all ports as a single object and write code that treats all ports uniformly when necessary. The StreamDrive API methods on ports implement StreamDrive communication protocol. The `sync_in/out` ports implement dataflow synchronization in cases where no data between the two actors need to be communicated. In particular, these ports implement the *push* and the *pop* methods but do not have associated channels and do not use associated *reserve* or *release* functions. All StreamDrive API communication methods take a communication port `id` and the number of tokens involved in the communication as parameters. The StreamDrive graph API provides methods for connecting the `stream_out_t` and `stream_in_t` ports, as well as the `sync_out_t` and `sync_in_t` ports together.

The *reserve* and *pop* operations use `WAIT` function to implement KPN blocking semantics. The implementation of StreamDrive API port methods is optimized such that no penalty is incurred when the channel is not blocking: ensuring that the DPN execution is not penalized by the KPN blocking behavior. The `WAIT` function is a *hook* into the runtime scheduler: it gives hand to the runtime scheduler when a blocking condition is encountered, and when the blocking condition is removed, the scheduler returns to a point in the `WAIT` function where the actor has been interrupted. Semantically, the `WAIT` function corresponds to a KPN execution mode; the runtime scheduler performs required context-switch when entering from and returning to the `WAIT` function.

The *push* and *release* operations are coordinated: when an actor releases a token from a full channel, it must unblock the actor on the sending side (if it was blocked). Similarly, when an actor pushes a token to an empty channel, it must unblock the actor on the receiving side (if it was blocked).

2.5.5 Deadlock Detection

The dataflow semantics requires that receiving a token from an empty channel shall block the receiving actor (process) until a token arrives. Token sending should always succeed without blocking, but such definition of the send operation would require infinite channel capacities, which any real implementation obviously cannot provide. In practice, computers have only a limited memory, so the assumption about infinite channel capacities is never fulfilled. Real implementations assign finite capacities to channels and redefine the send operation to block if the output channel is full. With such modification of operational semantics, an artificial deadlock can occur at runtime. This is a situation where a subset of actors is blocked in a

deadlock cycle, with at least one actor being blocked on send. The deadlock is artificial because it would not occur with non-blocking send, i.e. infinite channel capacities.

Generally, execution of arbitrary dataflow networks in limited memory requires runtime detection and resolution of artificial deadlocks, which was first addressed by Parks [157]. It is impossible to statically compute channel capacities that are sufficiently large to ensure that artificial deadlock cannot occur at runtime. Parks' algorithm resolves the problem when a global deadlock occurs, that is, when all actors in the network have become blocked. If at least one actor is found to be blocked on a send, the deadlock must be artificial, and it is resolved by increasing the capacity of the smallest full channel in the deadlock cycle. Geilen and Basten [158] have proposed a deadlock detection and resolution algorithm for deadlock that is local, i.e. involve only a part of the network.

The artificial deadlock resolution is impractical in a low-cost, low-power resource constrained embedded platform. Instead, in a practical implementation we treat the deadlock occurrence as error condition, i.e. detecting a deadlock should terminate the network execution. Indeed, an artificial deadlock can occur in two cases: (1) the application is executable from bounded memory but communication channels are not sufficiently large for the application working set, and (2) the application is not executable from bounded memory.

From a practical standpoint, applications whose memory requirements are not bound have limited interest in our target computing domain, therefore detecting the deadlock and raising the error condition is sufficient for our purposes. In order to detect an eventual deadlock, we are using the timer mechanism available in each cluster in our platform. A timer interrupt is thus programmed at the beginning of the dataflow graph execution. If the execution takes longer time, the interrupt is triggered terminating the execution. Usually, developers have a good idea of how long the execution should take. For example, when targeting a video sequence processing at 30 frames per second, each frame shall have been processed in at most 1/30 of a second. Generally, setting the deadlock detection interrupt to a few seconds is enough for detecting eventual deadlock even with low-performance debug version of the application.

2.5.6 Accounting Options

Accounting, i.e., collection of detailed performance statistics data, is a StreamDrive compile-time option because it causes noticeable extra overhead in terms of performance and memory footprint. The overhead in the accounting mechanism stems from measurements of runners and per-actor CPU time, as well as saving collected informa-

tion. When enabled, the accounting mechanism collects the following information:

- The time (consumed CPU cycles) spent inside the runners, as well as CPU time used by actors.
- Idle time used by the runners. The idle time is the total real time that runners spend in the IDLE state waiting for a synchronization event to be raised.
- Number of actor yields and context switches; each dispatch of an actor counts one.
- Dataflow statistics, i.e. number of tokens that have been sent and received on each actor port.

Additionally, StreamDrive can generate full VCD traces of the execution including waveforms describing execution of all actors showing the detailed state of the actor at any given point in time. These VCD traces are useful for performance evaluation and debugging.

2.6 SUMMARY

In this chapter, we have presented the StreamDrive framework including implementation details of its components: the hardware platform, the programming API, the runtime scheduler, and the communication layer.

The StreamDrive platform allows integration of streaming tightly-coupled application-specific hardware elements as actors of a dataflow application. The platform also supports the efficient dataflow synchronization via extending the programmable cores instruction set with special synchronization even handling instructions, via the dataflow-enabled DMA engine and via the dedicated synchronization event network.

The StreamDrive runtime scheduler is fully distributed allowing automatic load balancing. The StreamDrive runtime scheduler supports simultaneous execution of two types of dataflow actors, the KPN blocking processes and the DPN actors with firing rules. Supporting the two types of actors has two objectives. First, it allows optimal execution in a heterogeneous context because software actors are most efficiently executed under the control of a DPN scheduler, while the hardware implemented functions are most efficient running as independent KPN processes. Second, it enables the incremental successive refinement application development flow, starting with a sequential reference C algorithm and proceeding to a highly optimized DPN implementation. The KPN execution of software actors is intended then as an intermediate, sub-optimal way of executing dataflow applications.

The simultaneous support for the KPN and DPN actors and the possibility to refine a KPN process into a DPN actor during the application design space exploration raises a number of implementation difficulties. Among others, the StreamDrive optimizes the blocking behavior of KPN actors vs the firing rules of DPN actors. Another difficulty is the runtime stack management. In order to optimize the runtime stack footprint vs performance, the StreamDrive implements the novel stack spilling technique.

The StreamDrive communication protocol is designed to allow copy-free data exchange and is based on lock-free synchronization counters. Both, the software actors and the hardware implemented blocks support this common protocol. To further improve the dataflow communication efficiency in a limited shared memory clusters, the StreamDrive also implements a new broadcast and collect communication channels that allow different dataflow actors to share data buffers instead of duplicating them. This broadcast and collect channels also allow elegant and efficient implementation of data-parallel actors which is typically cumbersome in the standard dataflow execution model. To enable execution in limited memory space, the communication layer imposes finite channel capacities and redefines the dataflow send operation to block on full channels. This modification of dataflow semantics opens for the possibility of artificial deadlock, a problem which we address by a centralized runtime deadlock detection timer.

Finally, we have equipped StreamDrive with extensive runtime accounting of various performance data. This mechanism has overhead, so it can be completely disabled at application compile-time.

In this chapter we have discussed how the dataflow model can be used to develop parallel applications and how low-overhead dataflow runtime can be implemented in a shared memory cluster. In chapter 4, we will extensively evaluate performance of StreamDrive applications as well as performance of the StreamDrive itself.

3 COMPUTER VISION ACCELERATOR

Normal people believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet.

—Scott Adams

This chapter focuses on developing the *Computer Vision Engine* (CVE), an implementation of the StreamDrive platform targeting the computer vision application domain. The central element of the CVE is the HWC tightly-coupled convolution hardware block, which can efficiently execute CNN convolutional workloads in addition to the standard image processing convolution.

The computer vision is the automated extraction of information from images. Traditional approaches to computer vision date back to the past 10-20 years and are characterized by extracting human-engineered features (edges, corners, color) deemed to be relevant in vision tasks, such as image classification, object detection, face recognition, etc.). The techniques stemming from deep neural networks learn these features via an automated procedure using non-linear statistical models (deep nets). Particularly popular are Convolutional Neural Networks (CNN) that are widely used for solving artificial intelligence problems, such as object and voice recognition, scene labeling and others [159].

There has been a significant amount of research into hardware acceleration of traditional feature detection algorithms. The examples include the GPU-accelerated [160, 161], and FPGA-accelerated [162, 163, 164, 165, 166] solutions. However, these solutions are either too general and too costly for being used in constrained embedded platforms, or are too inflexible. For instance, Bouris *et al.* [163] and Svab *et al.* [162] both presented a fixed FPGA implementations for the SURF algorithm. Although there are many opportunities for creating application-specific hardware, specializing it for particular feature detectors and feature generators, this would lead to over-specialization. Such dedicated hardware is either too complex, supporting dozens of different functions and configurations options, or too inflexible, i.e. useful in a single image processing application.

One example of a common image processing operation that is used by virtually all image processing algorithms is constructing the *image pyramid*. The image pyramid helps detecting image patterns that appear at different scales. The image pyramid consists of a sequence of copies of an original image in which pixel density and resolution are

decreased in regular steps. The feature detection is then applied over a series of images representing scaled down original image. Computing the image pyramid involves pixel interpolations and it is possible to develop a tightly-coupled pixel interpolator for this task. However, a special hardware pixel interpolator is usually readily available as part of most embedded image processing systems. Therefore, we decided to use the existing off-cluster hardware block for building the image pyramid.

Another common basic operation used by different computer vision and image processing algorithms is the convolution. Convolution is also the most expensive operation in terms of computing requirements due to the large amounts of data that needs to be processed. For example, best commonly used feature descriptors in terms of precision, such as the Scale-Invariant Feature Transform (SIFT) [167] and the Speeded Up Robust Features (SURF) [168] require computing the gradient of the image in the region of each feature by convolving it with a 2D filter, which is the most time consuming part of processing. Convolution is ubiquitously used in image processing algorithms for filtering to remove useless details and noise from images, for edge or gradient detection, for blurring and sharpening images, embossing, for local range detection and local standard deviation calculations, as well as for other transformations.

Recently, many research and development efforts have focused on CNN inference accelerators in order to meet their high computational requirements with reasonable energy and cost efficiency [169]. In the CNN computation, convolution accounts for up to 90% of the processing. Thus, in the development of a flexible solution for the CNN acceleration, a critical keypoint is designing computationally efficient means of performing the CNN convolution.

The straightforward approach to accelerate the convolution processing is to use the 2D convolvers. This works well with image filtering and traditional computer vision implementations. However, fast implementations of a 2D convolution core are necessary but not sufficient to accelerate the CNN workload. In CNNs, storing significant amount of intermediate data to the off-accelerator memory and the detailed orchestration of complex control and data flows by the controlling processor quickly offset performance gains obtained by using fast hardware only for 2D convolutions.

3.1 DESIGNING CNN CONVOLUTION ACCELERATOR

The main bottleneck for implementing efficient and cost effective convolution hardware for the CNN inference is the memory system. Large volume of data accessed (weights) and produced (feature maps)

during CNN inference makes it difficult to simultaneously buffer the input feature maps, the output feature maps, and the filter weights in limited internal accelerator memory. This generates high memory traffic between the accelerator internal memory and off-accelerator memory, resulting in lowered performance and increased power consumption. One way to alleviate this issue is to use large SRAM buffers (up to a few MBytes may be used) in order to completely eliminate main memory traffic [32, 170]. When massive accelerator area budget is available, this may be an acceptable approach. However, large amounts of memory are not affordable in deeply-embedded markets, such as mobile or IoT clients, for example. Furthermore, even with abundant memory resources, relying solely on large internal buffers is an expensive approach in terms of area and power.

While completely absorbing all CNN memory traffic in internal accelerator storage is usually not possible, the memory bandwidth requirement for a given accelerator storage capacity can be significantly reduced if sufficient data reuse happens. The data reuse pattern of a computation, in time and space, is determined by the *computation schedule*. Generally, the efficiency and performance impact of the *computation schedule* varies with CNN topology and size making it difficult to adapt the accelerator architecture to different CNNs.

This Chapter describes an analytical model for finding the best *computation schedule* for a CNN convolutional layer, such that computation working set fits limited internal accelerator memory while the number of accesses to the off-accelerator memory is minimized. It is shown that proposed model is more accurate than previously published models in the case of application-managed scratchpad memories, which are used in the majority of computing platforms dedicated to CNNs [27, 28, 29, 30, 31, 32, 33, 34, 35, 36]. We have applied this model for designing a tightly-coupled convolution hardware block, the HWC, that we integrate with the StreamDrive shared memory cluster creating a complete domain-specific computer vision platform, the CVE. The *computation schedule* found using our model is not trivial and achieves significant bandwidth reductions with respect to previously published accelerators based on a similar architecture [29]. We show that this *computation schedule* is implementable in practice by designing the HWC using the CatapultC high-level synthesis tool [171]. We have verified that our analytical model is accurate by comparing the number of memory accesses predicted from our model with the real number of memory accesses measured from the HWC implementation. Finally, based on our analytical model we develop a methodology for optimal convolution loop-nest tiling with respect to the cluster shared memory size.

3.1.1 Background on CNNs

The success of the AlexNet [172] CNN in the 2012 ImageNet recognition competition has established it as one of the most promising machine learning techniques.

As a classical supervised learning algorithm, CNN employs a feed-forward process for recognition, or *inference*, and a backward path for training. In industrial practice, application designers train CNN off-line and use the off-line trained CNN to perform the inference. So in this thesis, we focus on implementing the forward inference computation.

A typical CNN for image recognition is composed of two components: a feature extractor and a classifier. The feature extractor is used to filter input images into *feature maps* that represent various features of the image. These features may include corners, lines, circular arch, etc., which are relatively invariant to position shifting or distortions. The working principle of CNN is to extract the local features from high-resolution feature maps and combine them into more abstract low-resolution feature maps. These are realized by two alternating types of layers: convolutional and pooling layers. The last few layers are often fully-connected layers that combine all local features together to produce the abstracted classification results. The output of the feature extractor is a low-dimensional vector containing these features. This vector is then fed into the classifier, which is usually based on traditional artificial intelligence algorithms. The purpose of this classifier is to decide the likelihood of categories that the input (e.g. image) might belong to.

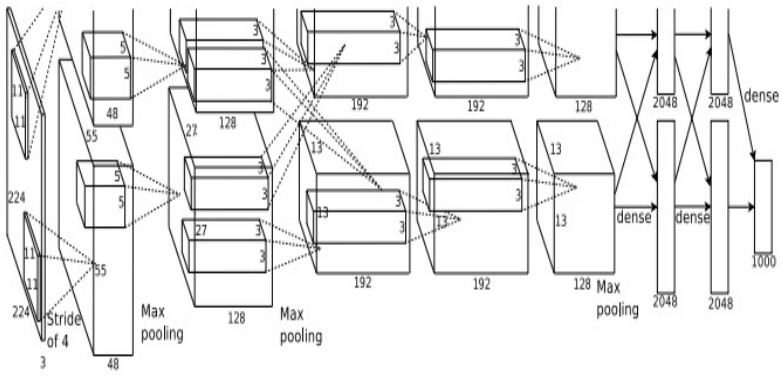


Figure 3.1: AlexNet CNN that won the ImageNet 2012 contest [May look weird because there are two different processing "streams". This is because the training process was so computationally expensive that they had to split the training onto 2 GPUs.].

Figure 3.1 shows AlexNet, a CNN application, taken from [172]. This CNN is composed of 8 layers. The first 5 layers are convolutional layers and layers $6-8$ are fully connected. The algorithm receives

three 224x224 input images that are from an original 256x256 three-channel RGB image. The output vector of 1000 elements represents the likelihoods of 1000 categories. As is shown in Figure 3.1, Layer 1 receives 3 input feature maps in 224x224 resolution and generates 96 output feature maps in 55x55 resolution. The output of Layer 1 is partitioned into two sets, each sized 48 feature maps (this is the result of partitioning over two GPU units in the original implementation). Layer 1's kernel size is 11x11 and the sliding window shifts across feature maps in a stride of 4 pixels. The following layers also have a similar structure.

From the inference perspective, previous studies have shown that convolution operations will occupy up to 90% of the computation time [173, 174]. So the hardware acceleration must focus on accelerating convolutional layers.

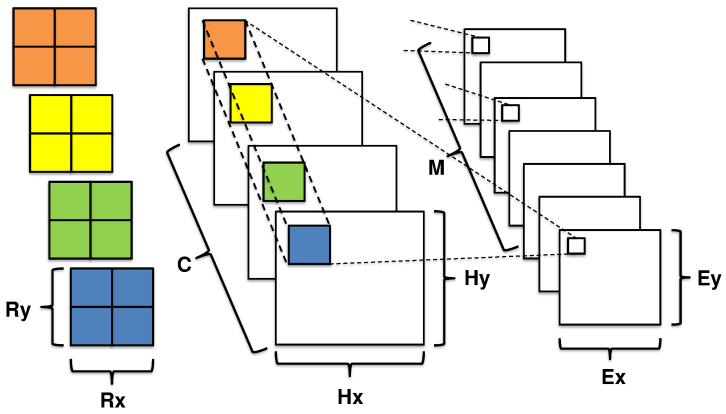


Figure 3.2: CNN convolutional layer illustration.

Figure 3.2 illustrates the computation of a CNN convolutional layer. The convolutional layer receives feature maps as input. Feature maps are 3D volume data with three dimensions: width H_x, height H_y and depth C. Each input feature map is convolved by a shifting window with a $R_x \times R_y$ kernel to generate one pixel in one output feature map. The stride of the shifting window is S, which is normally smaller than R_x and R_y. Generally, if the convolution stride is greater than 1, the output feature map dimensions are different from the input feature maps, i.e. of width E_x and height E_y. A total of M output feature maps will form the set of input feature maps for the next convolutional layer. The computation in the convolution stage, that is repeated for each pixel in each $E_y \times E_x$ output feature map over all M output feature maps, is shown in Equation 3.1:

$$O[m][y][x] = \sum_{c=0}^{C-1} \sum_{k=0}^{R_y-1} \sum_{l=0}^{R_x-1} I[c][y \cdot S + k][x \cdot S + l] * W[m][c][k][l], \quad \forall m \in M, \forall y \in E_y, \forall x \in E_x \tag{3.1}$$

```
// M output fmaps loop
LOF: for (m = 0; m < M; m++)
  // C input fmaps loop
  LIF: for (c = 0; c < C; c++)
    // spatial loops (ExE)
    LSY: for (y = 0; y < E; y++)
      LSX: for (x = 0; x < E; x++)
        // filter loops (RxR, stride S)
        LFY: for (k = 0; k < R; k++)
          LFX: for (l = 0; l < R; l++)
          {
            p = I[c][y*S+k][x*S+l];
            w = W[m][c][k][l];
            O[m][y][x] += p*w;
          }
```

Figure 3.3: Canonical form of the CNN convolution layer loop-nest.

Symbol	Description
H	Input feature map size (H × H)
E	Output feature map size (E × E)
R	Convolution kernel size (R × R)
S	Convolution kernel stride
C	Number of input feature maps
M	Number of output feature maps

Table 3.1: Symbols used to discuss the CNN convolution in this thesis.

Figure 3.3 shows a "canonical form" of a generic CNN convolutional layer expressed as a 6 level loop nest. For simplification, the initialization of output feature maps to 0 before the accumulation is omitted from the figure. Throughout the explanation, square feature maps are used to avoid overburdening the mathematical notation, although the discussion can be easily extended to general rectangular feature maps. Table 3.1 lists symbols used in discussing the CNN convolutional layer and their meaning. In the Figure 3.3, there are 3 arrays referenced in the loop-nest: the I[C][H][H] holds the C input feature maps of size H × H, the *W*[M][C][R][R] holds M × C convolution kernels, with R × R weights each, and the O[M][E][E] holds M output feature maps of size E × E.

Data reuse occurs when a reference within a loop accesses the same data element in different iterations. Convolutional loop-nest shown in Listing 3.3 contains several opportunities for data reuse:

- **Convolution reuse**: Each input feature map pixel is reused R^2 times within each input feature map.
- **Weight reuse**: Each kernel weight is reused E^2 times.

Carries?	LFX	LFY	LSX	LSY	LIF	LOF
I	✓ or ✗	✗ or ✓	✓ or ✗	✗ or ✓	✗	✓
W	✗	✗	✓	✓	✗	✗
O	✓	✓	✗	✗	✓	✗

Table 3.2: Carrying loops of array references of the CNN loop-nest.

- **Imap reuse**: Each input feature map pixel is reused across M output feature map computations.
- **Omap accumulation**: Each output feature map pixel is reused across accumulations of partial results from C input feature maps.

We say that *reuse* of a reference is *carried* by a loop if the same memory location is used by different iterations of that loop [175]. In Figure 3.3, reuse of the $W[m][c][i][j]$ reference is *carried* by two loops, **LSX** and **LSY**; reuse of the $I[c][y*S+i][x*S+j]$ reference is carried by a combination of loops, (**LFX**,**LSX**), and (**LFY**,**LSY**), as well as by loop **LOF**; and reuse of the reference $O[m][y][x]$ is carried by the loops **LFX**,**LFY**, and **LIF**. The reuse of the I reference is slightly more complex because it is carried by a combination of two loops: which loop in a pair of loops, the (**LFX**,**LSX**) and the (**LFY**,**LSY**), is carrying the reuse of array I depends on relative ordering of the loops - the reuse is carried by the outer loop in each pair. Table 3.2 resumes which loops are carrying each array reference in the CNN loop-nest.

In order to take full advantage of the reuse, such that every data is loaded from main memory only once (or stored for the O array), large local buffering capacity is necessary. In Figure 3.3, the entire set of input feature maps needs to be stored in the local buffer to reuse the input data across the loop **LOF**. To reuse the accumulated partial output feature maps O across loop **LIF**, one full output feature map needs to be stored in local buffer. To put this into perspective, the total amount of buffering required for the second convolution layer of AlexNet shown in Figure 3.1, with $H = 55, E = 27, C = 96, M = 256, R = 5$, exceeds 284 KB if we consider that each element in the input and output feature maps and the kernel weights is 1 byte in size.

If large local buffer is not available, optimal data reuse cannot be achieved and some data must be accessed from the external level in memory hierarchy multiple times. For example, the standard 2D convolvers buffer a few lines of input feature maps in the local memory. Particularly, a delay line of R lines of input feature maps are buffered locally [173]. Thus, every input feature map pixel needs to be reloaded M times, once for each iteration of loop **LOF** in Figure 3.3. The problem is that, although there is plenty of data reuse in the CNN

loop-nest, unless the entire working set fits the local buffer, this reuse cannot be taken full advantage of.

3.1.2 State-of-the-Art

The most straightforward way of accelerating the CNN convolution processing is to use the 2D hardware convolvers initially designed for standard image processing and filtering. For a pipelined implementation of such convolvers, an input delay buffer is used [173]. For an image of $H_y \times H_x$ and a $R_y \times R_x$ convolution kernel, the size of this buffer capacity needs to be able to hold at least $H_x \times R_y$ input pixels, with some additional room to allow reading new input data while processing the convolution. After a delay that is equal to the time required to fill the buffer, output pixels are available every clock cycle. In such 2D convolvers convolution weights are stored in local registers for the duration of processing one input image. Although simple to implement, the 2D convolvers are very inflexible and do not scale well for bigger workloads. These implementations usually limit the maximum convolution kernel size; at the same time when a CNN requires a smaller kernel, the accelerator efficiency drops. They support limited variety of CNN shapes and layers. Their input buffering capacity limits the size of images that can be processed leading to loss of efficiency with larger images. The *CNP processor* [173] was the first CNN specific FPGA implementation based on the 2D convolver operation. The CNP architecture was designed to execute the operations for all common CNN layers. The CNP evolved into *NeuFlow* [176] that used multiple CNP convolution engines together. The NeuFlow architecture was further evolved into the *nn-X* [28] taking advantage of a more powerful FPGA chip.

Numerous other research were based on similar 2D convolvers focusing on computational performance or on communication infrastructure issues [173, 177, 178, 28, 179, 29, 180, 181]. These CNN accelerators are inflexible (which makes them inefficient with most recent CNN topologies (like Google's Inception or ResNets) and do not scale well to higher performance applications.

Reuse-Driven Accelerators

The large data sizes of CNN convolutional layers require the computation to be tiled to avoid using very large on-chip buffers. What makes the tiling hard is the fact that all the data, the input feature maps, the kernel weights, and the output feature maps, are reused multiple times. This data reuse needs to be efficiently exploited in order to reduce off-chip memory bandwidth. Several specialized accelerators attempted to overcome the 2D convolvers limitations by

scheduling the CNN convolution computations using different trade-off on which data reuse to favor and which to sacrifice.

Chakradhar *et al.* [182] proposed a convolution engine in which a number of input feature maps and a number of output feature maps can be processed simultaneously, therefore exploring a mix of different reuse opportunities. The basic computational element of this architecture is a bank of convolvers all computing one output feature map at a time. Each bank is composed of several 2D convolvers whose outputs are accumulated - therefore omap reuse is achieved inside the bank. If more input feature maps need to be combined than there are convolvers in the bank, then the aggregated output from the convolver bank may only be a partial output (i.e. partial sums) that must be stored in off-chip memory. Each convolver bank can store a number of convolution kernels internally achieving weight reuse. Multiple convolver banks can be put together so that one input feature map can be simultaneously used to compute more than one output feature map - imap reuse. By varying the number of convolvers in a convolver bank, as well as the total number of convolver banks, the application can control the extent to which the different reuse dimensions are explored to match the exact computational workload of a given CNN layer. Each convolver bank also has a specialized hardware pipeline to compute non-linearity and sub-sampling functions used in CNN processing.

Origami [183] architecture tiles the convolutional layer into blocks with a fixed number of input and output feature maps. The block of input feature maps is fed in stripes of configurable height into the processing engine and stored in an internal SRAM buffer, which keeps a spatial window of the input data. The processing engine is composed of a number of sum-of-product (SoP) units which process the same input feature map, but different weight kernels, such that each SoP computes the partial sum for a different output feature map. Thus, the imap reuse is exploited inside the processing engine. The processing engine iterates over the tiled input feature maps while the partial sums are accumulated internally to compute the complete result - omap reuse. During the processing, on-chip filter bank holds all the weights required for processing a tile of input and output feature maps, exploiting the weight reuse. Because Origami needs all input data to be stored in on-chip buffer before the processing, the processing engine idles while the hardware is configured and while these data are transferred to the SRAM.

The DianNao work [184] analyzed in details the data reuse properties of the different layers encountered in CNNs and the more general Deep Neural Networks (DNN). The authors then proposed a neural network accelerator architecture consisting of three data reuse structures: an input buffer (NBin), an output buffer (NBout) and a weight buffer (SB) connected to a Neural Functional Unit (NFU). The

NFU has a fixed size tailored for processing a tiled image of T_i input feature maps producing T_n output feature maps. The input feature maps are split into chunks which fit in NBin, and they are reused by implementing NBin as a circular buffer. The partial output sums of T_n output pixels are computed for a chunk of input pixels contained in NBin. Then, the same input pixels are used for another chunk of Tn output pixels, etc., exploring the imap reuse. When the input pixels in NBin are reused for a new set of Tn output pixels, the previous chunk partial sums are stored in the NBout buffer. Naturally, the loop iterating over output chunks must be tiled so that all simultaneously computed partial sums fit the capacity of NBout. DianNao also implements non-linear and pooling functions in hardware. In DianNao architecture, the *computation schedule* and buffer sizes have been determined experimentally. The authors acknowledged that, like many processing architectures, DianNao's efficiency and scalability remained severely limited by memory bandwidth constraints. Chen *et al.* [184] acknowledge this issue by observing that their neural network accelerator loses at least an order of magnitude in performance due to memory accesses. DianNao's successor, ShiDianNao accelerator [170] was directly integrated with an image processor sensor fully eliminating the off-chip DRAM. This approach is not scalable as only a few small CNNs can be accommodated.

One more work that must be mentioned is the Eyeriss accelerator [82, 32]. In their paper, the authors first propose a taxonomy that classifies existing CNN *computation schedules* from different research. Thus, CNN architectures have been classified as *weight-stationary*, therefore favoring the convolutional and weight reuse; *output-stationary* favoring the omap reuse, and *no-local-reuse* exploring the imap and omap reuse. Eyeriss also proposed the new *row-stationary* architecture enabling a trade-off between different types of reuse. The CNN convolution computation is mapped onto an array of processing elements (PEs) by tiling the feature maps and the input and output feature maps volume. Each PE uses the local scratchpads memory for both convolutional data reuse and partial sums *psum* accumulation. Filter rows, input pixels and *psums* are also reused within a feature map tile via the inter-PE communication network. By exploiting the low-cost memory levels, the PE scratchpads and the inter-PE communication, Eyeriss minimizes data accesses to the high-cost levels, including the large on-chip global buffer and the off-chip DRAM.

More similar architectures include Angel-Eye [185], DLAU [186], Orlando [36], and others. The above reuse-driven architectures reduce accesses to external memory by exploiting convolution data reuse. However, they use dedicated buffering schemes leading to sub-optimal amount of reuse because there is no flexible memory hierarchy with such buffers. Given different CNN layers, there is always

a choice to increase the data reuse of some of the input feature maps, the weights, or the partial sum accumulations, while sacrificing the data reuse for the others. The buffering design choice and the necessity to resort to complex arbitration and routing logic to share inputs and connect outputs of the convolvers to other resources limit the extent to which the data reuse trade-off can be exploited.

The implementation flexibility can be increased using the FPGA based architecture template because the buffering organization can be adjusted to exploit data reuse in memory access patterns of a particular CNN layer. Thus, Peemen *et al.* [27] developed a memory-centric configurable FPGA accelerator template for CNN, with flexible data reuse buffers. The template supports different compute patterns in the CNN workload and can be configured with the number of PEs, connectivity, supported addressing modes, buffer configuration, and buffer depth. The buffer configuration parameters influence the hardware instantiation and are fixed after synthesis. By reconfiguring the memory resources, Peemen reported up to 13 times reduction in required buffer resources compared to accelerators with fixed buffering scheme, while maintaining the performance. Peemen's accelerator also supports a variety of layer configurations. Thus, multiple parameters of a CNN can be programmed, such as convolution kernel size, feature map size, number of input feature maps, and sub-sample size.

Zhang *et al.* [38] propose a CNN accelerator template for FPGA with multiple parallel processing elements (PE) and dedicated on-chip buffers. As with previous architectures, the CNN convolutional layer computation is divided into tiles of T_n input and T_m output feature maps. The feature maps are also tiled such that the entire combination of (T_m, T_n, T_y, T_x) pixels fits with accelerator internal buffers. The accelerator implements T_m concurrently executing computation engines. Each computation engine accepts T_n inputs from input feature maps and T_n inputs from weights and one input from bias, which is stored in the buffers of output feature maps. The accelerator is producing partial results for T_m output feature maps every clock cycle. On-chip buffers are built upon a basic idea of double-buffering, in which double buffers are operated in a ping-pong manner to overlap data transfer time with computation. Therefore, they are organized in four sets: two for input feature maps and weights and two for output feature maps. Every buffer set contains several independent buffer banks. The number of buffer banks in each input buffer set is equal to T_n (tile size of input feature maps volume). The number of buffer banks in each output buffer set is equal to T_m (tile size of output feature maps volume). Zhang used polyhedral-based data dependence analysis [187] to derive a series of CNN implementations through loop unrolling, pipelining, and the tile size enumeration with the objective to find the best performing one. Taking into consideration additional design constraints, such as available resources, communication topol-

ogy, etc., Zhang proposed a particular shape of the CNN tiled loop-nest. Zhang's architecture template allows adjusting the tile sizes of the CNN computation in order to optimize the data reuse. Based on experimental observation of several typical CNN layers, the authors implemented the memory subsystem with $T_m = 64$ and $T_n = 7$; this clearly favors omap reuse.

Shared Memory Accelerators

An alternative to dedicated buffering and communication topology is to use shared memory subsystem in order to facilitate the flexibility in exploiting data reuse in memory access patterns. Surprisingly, very few research have explored this possibility.

Sankaradas *et al.* [188] proposed a convolution co-processor with functional units consisting of parallel 2D convolution operators and programmable units performing sub-sampling and non-linear functions specific to CNNs. The co-processor is connected to distributed off-chip memory banks with large data bandwidth. The convolution weights are kept in dedicated register file during the convolution processing, while the off-chip memory is used as a scratchpad for intermediate data. Significant amount of intermediate data are transferred over the off-chip link requiring multiple DDRx channels, which are expensive in terms of power and cost (chip area or pin count).

DaDianNao [189] is a successor of DianNao discussed earlier, developed with the objective to overcome the size limitations of the DianNao architecture. For this, DaDianNao employed a sufficiently large on-chip memory for storing large CNNs close to the datapath. The accelerator is composed of multiple chips, each chip containing computation logic together with enough RAM that the sum of the RAM of all chips can contain the whole neural network, requiring no off-chip memory. This way the convolution weights can reside in on-chip memory, and the intermediate results do not need to be frequently written back and read from off-chip memory, thus removing the off-chip memory access bottleneck. This architecture requires that the total memory footprint of CNN dataset, however large (up to tens of GB), is fully mapped to on-chip storage (in a multi-chip system with a reasonable number of chips). This puts the system size and power consumption outside of the embedded domain constraints. DaDianNao needs to use large on-chip eDRAM buffers exceeding embedded systems requirements in terms of power and cost.

Neurocube [190] and TETRIS [191] architectures leverage on recent through-silicon-via (TSV) technology 3D memory technology in order to reduce the size of on-chip SRAM buffers. Compared with the conventional DRAM technology, 3D memory provides an order of magnitude higher bandwidth with better energy efficiency. Using the 3D memory allows to devote more chip area for processing elements and less area for on-chip SRAM buffers. For example, TETRIS

integrates 8 DRAM dies on top of a logic layer that contains 16 neural network (NN) engines used to process a CNN layer in parallel. Each NN engine is similar to a single Eyeriss accelerator [82]: hundreds of PEs are connected through a dedicated network into a 2D array. A global buffer is shared by all PEs to store and reuse data from memory. TETRIS reduces the global buffer size arguing that the cost of accessing the 3D stacked DRAM is not much higher than accessing the on-chip SRAM in terms of bandwidth and power. Therefore, using the on-chip buffers can be bypassed in favor of in PE registers which then capture most of the data reuse.

Above shared memory CNN accelerators require chip area, power consumption and pin count that fall outside of the embedded computing domain.

We are aware of only one embedded accelerator which uses shared memory for implementing the CNN computation. Conti *et al.* [29] proposed to connect a Hardware Convolution Engine (HWCE) to a PULP shared memory cluster [192, 193]. The HWCE communicates with processing elements (PE) via the same cluster shared memory (a scratchpad) used by the PEs themselves. The tight coupling of the HWCE to shared memory allows gaining additional flexibility within the convolutional layer (using sub-sampling, non-linearity, etc.) operations, as well as across multiple network layers of the CNN. However, the HWCE engine itself is a classical 2D convolver with a delay line buffer for the input feature maps and a local memory for storing the current convolution kernel weights. The reported HWCE implementation supports convolution kernel size of 5x5 and the input feature map line width of 32, 64, or 128 bytes. Similar to all limited memory size accelerators, the HWCE requires that CNN convolution computation space is tiled into a number of smaller blocks. The overhead of orchestrating larger convolutions or computing larger images can quickly offset performance gains obtained by using fast hardware convolver. Most of all, as shown in section 4.2, the *computation schedule* implemented using standard 2D convolvers is sub-optimal.

3.2 MEMORY PERFORMANCE OPTIMIZATION

We consider the problem of optimizing the CNN convolutional layer memory access pattern for a two-level memory hierarchy. Figure 3.4 shows a generic view for an accelerator consisting of computing datapath, *local reuse buffer* optimized for this datapath, and *off-accelerator main memory* external to the computing datapath. The problem consists in minimizing the number of memory accesses to the off-accelerator memory given a limited local buffer capacity.

A generic 2-level memory hierarchy such as that exemplified in Figure 3.4 exposes two memory levels: *off-accelerator memory*, where

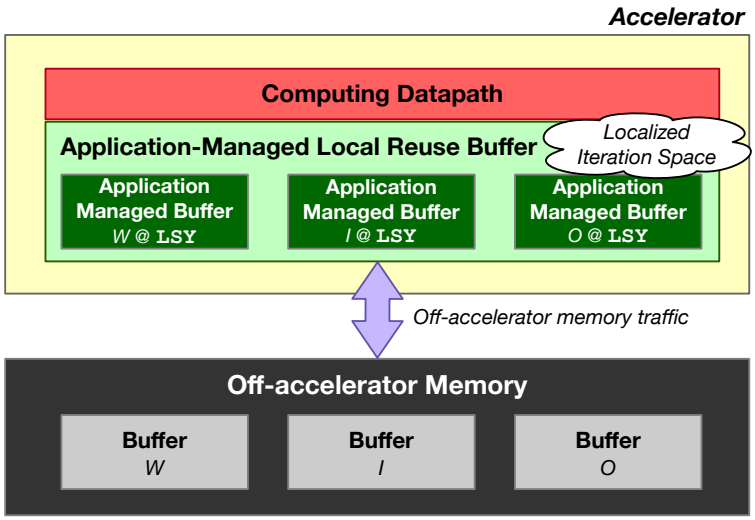


Figure 3.4: Generic view of a CNN accelerator combining a computing datapath with an optimized application-managed *local reuse buffer*, and off-accelerator memory external to the datapath.

we assume all data involved in the CNN loop-nest is resident, and a *local reuse buffer* that usually cannot host the entirety of the data, but is vastly faster and more energy-efficient than off-accelerator memory. This buffer is used to host data that is reused multiple times. While data reuse is inherent in the computation and not dependent on a particular shape of the loop-nest, the usage of a local reuse buffer of any kind implies that the reuse only translates into a reduction of memory accesses if there is enough *data locality*, i.e. if data inside the local buffer are reused within a short period of time and are not replaced between reuse accesses.

Data caches

Existing reuse evaluation methods derive from cache behavior and build a *localized iteration space*, i.e. a set of k innermost loops of a loop-nest where data locality is exposed [40]. It is assumed that all array references inside the *localized iteration space* need to be simultaneously stored in a local reuse buffer [175, 27, 38, 39]. Indeed, in the context of a data cache, every array reference is mapped to a unique location in the cache. If the cache capacity is smaller than required for holding all array references, reused data can be displaced from the cache and are not guaranteed to remain in the cache in every iteration that they are referenced. Thus, all data touched inside the *localized iteration space* need to be cached in order to benefit from the data reuse. For example, in order to reuse elements of array I across the loop **LSY** in Figure 3.3, the *localized iteration space* will have to include loops **LFX**, **LFY**, **LSX** and **LSY**. The data cache would need to hold $H \times H$ elements of I - otherwise referencing I could displace some other reused data,

O for example. The data cache would also need to hold $E \times E$ elements of the output feature map array O - otherwise, referencing the array O may displace elements of array I from the cache before they have been reused.

Application-managed scratchpad memories

Dedicated hardware accelerators, including CNN accelerators, commonly use application-managed scratchpad memories as local reuse buffers instead of caches, as these are deemed to provide better performance, predictability, and energy efficiency [147, 169]. In this case, data placement, reuse, and transfer have to be managed explicitly by partitioning the local reuse buffer in a set of *application-managed buffers*, one per each array referenced in the loop-nest. Application-managed buffers can be partitioned statically (i.e. by using physically separate memory to implement them) or dynamically (i.e. by partitioning a single piece of memory so that all array references fit). Instead of a single *localized iteration space*, each array reference can have its own data locality scope. Thus, utilization of the local reuse buffer can be optimized by choosing, for each array reference, a nested loop level at which data are buffered for reuse. We call this level the *buffering level* of the array reference.

The number of loop iterations, d, that separates two consecutive accesses to the same data element, is called the *dependence distance of data reuse* or simply the *reuse distance* [40]. Only the elements of the array touched by d loop iterations need to be buffered in application-managed local buffer in order to ensure that reused data is preserved across loop iterations. For example, in Figure 3.3, the reuse distance of the reference to array I across the loop **LSY** is $H \times R$ iterations (H elements of I are touched iterating over the **LSX** loop). Therefore, if an implementation chooses to buffer the array I at the **LSY** level, enabling I data reuse across this loop, $H \times R$ elements of I need to be buffered in the local buffer. Similarly, buffering the array O at the **LFY** level, i.e. reusing array O data across the two innermost loops, requires only a single element of the array O to be buffered inside the local buffer.

3.2.1 Data Locality Optimization

The memory performance optimization can be formulated as a loop-nest optimization problem [40]. For example, methods such as [175] or [27], use *reordering* and *tiling* of the loop-nest to maximize the data locality. Loop reordering places a subset of the k loops that form the *localized iteration space* at the innermost position. Loop tiling partitions the loop-nest iteration space into a number of smaller blocks, such that data used inside the *localized iteration space* stays in the local buffer until it is reused. Figure 3.5 shows a general form of tiled

```
// output fmaps -- loop on tiles
LTOF: for (mm = 0; mm < M; mm += mss)
  // input fmaps -- loop on tiles
  LTIF: for (cc = 0; cc < C; cc += css)
    // spatial -- loops on tiles
    LTSY: for (yy = 0; yy < E; yy += yss)
      LTSX: for (xx = 0; xx < E; xx += xss)
        // output fmaps -- tile loop
        LOF: for (m=mm; m<min(mm+mss,M); m++)
          // input fmaps -- tile loop
          LIF: for (c=cc; c<min(cc+css,C); c++)
            // spatial -- tile loops
            LSY: for (y=yy; y<min(yy+yss,E); y++)
              LSX: for (x=xx; x<min(xx+xss,E); x++)
                // kernel -- tile loops
                LFY: for (k=0; k<R; k++)
                  LFX: for (l=0; l<R; l++)
                  {
                    p = I[c][y*S+k][x*S+l];
                    w = W[m][c][k][l];
                    O[m][y][x] += p*w;
                  }
```

Figure 3.5: Tiled CNN convolution layer loop-nest.

convolution loop-nest, where the choice of tile sizes mss, css, yss, xss leads to different data locality results.

With caches, the order in which loops from the *localized iteration space* execute is not important because the ability to reuse data depends solely on the total number of data elements loaded between reuses, i.e. only on the size of the *localized iteration space*. Conversely, with application-managed buffers, the order of loop execution has a substantial effect on data locality. To see this, consider reusing elements of array I across the **LIF** loop in the Figure 3.5. Let h be the number of elements of I touched across iterations of each loop, **LSX** and **LSY**. It is easy to see that $h \times R$ elements need to be buffered in the local buffer to ensure the reuse of I across the loop **LIF**. Consider reordering loops **LIF** and **LOF**. All elements of array I touched by loops **LSX** and **LSY** are reused in every iteration of the **LOF** loop (i.e. $h \times h$ elements of I). A local buffer of $h \times h$ elements of I is thus necessary in order to ensure the reuse of I across the loop **LIF**.

With limited buffering capacity it is not possible to fully exploit the full amount of data reuse for all array references at the same time. In general, the problem of optimally scheduling computation in arbitrary loop-nest is complex, requiring an exhaustive search of a large solution space. Given the large possible number of schedules — the computation is a multiple level loop-nest - computation scheduling needs to consider all possible loop splits and loop orders for each

split. Several research proposed analytical memory models for optimizing loop-nest computations.

3.2.2 Previous Work

Wolf and Lam [40, 175] used loop blocking and reordering techniques to capture data locality and reduce the memory traffic in scientific computations. They used a combination of loop interchange, skewing, and reversal (altogether called *unimodular transformations*) with loop tiling to improve data locality of loop-nests in the context of memory hierarchies with caches. Wolf's algorithm builds a *localized iteration space* such that all data references inside the space can fit within the data cache; the objective is to find the localized iteration space that maximizes data reuse. Such model is inaccurate in the context of memory hierarchies with application-managed buffers. As a result, Wolf's method overestimates the real buffering requirements and required memory bandwidth of a computation and leads to sub-optimal *computation schedules*.

Peemen *et al.* [37] proposed an accelerator model where a computation loop nest is split into two parts: an *innermost tile*, (similar to M.Wolf's *localized iteration space*), for execution on the accelerator, and outer *controlling loops* that run on a host processor. Peemen's model improves on M.Wolf's cache model significantly by taking into account that with application managed buffers some data can be reused between consecutive executions of the *innermost tiles* (for example, in computations involving prologue, steady state, and the epilogue). Authors proposed a design flow for selecting the best *computation schedule* to maximize data reuse given a buffer size restriction. This is achieved by exhaustively exploring all possible tilings and controlling loop orders of a given computation loop-nest. However, Peemen's algorithm estimates accelerator buffer requirements as the number of distinct data elements accessed in the *innermost tile* of the computation. Thus, the buffer capacity estimation remains inaccurate and the solution is sub-optimal, as we show later in this section.

Zhang *et al.* [38], cited earlier, proposed an analytical approach for analyzing computing throughput and required memory bandwidth of a CNN design on an FPGA platform. Similar to Peemen's method, the CNN loop-nest is tiled with the innermost tiled loops executing in the FPGA, while controlling loops execute in the host. All data necessary for the computation of the *innermost tile*, eg. the input feature maps and the weight kernels need to be transferred into the FPGA BRAM buffers before the computation; resulting output feature maps and partial sums are transferred back to off-FPGA memory after the execution. In order to reduce the main memory traffic authors use local memory promotion [187] for placing out-of-FPGA communication operations optimally. The local memory promotion allows moving a

transfer of data from array X across the innermost controlling loop, when this loop carries full reuse of X, i.e. this loop's iterator does not appear in any reference to array X. Similarly, to Peemen's method, Zhang *et al.* explore 4 possibilities for the 4 different innermost controlling loops in the convolution loop-nest. The resulting *computation schedules* are less efficient than in Peemen's approach because only a subset of array references benefit from data reuse across consecutive executions of the innermost tile.

The TETRIS system [191], also described earlier, analytically derived optimal *computation schedules* for the CNN convolution by simplifying the problem. They proposed the *bypass ordering* where internal on-chip storage is bypassed for two out of the three input streams in the convolution layer using a large register file for exploiting data reuse from the two *bypassed* streams instead. TETRIS explored three variants: I*W* bypass (avoid the local buffer for the input feature maps and weights), O*W* bypass (avoid the local buffer for the output feature maps and weights), and IO bypass (avoid the local buffer for the input and output feature maps). As explained earlier, the main idea of TETRIS architecture is to reduce the on-chip buffering leveraging on high bandwidth of the 3D memory system. Bypass ordering is significantly simpler than the general CNN computation scheduling problem, and it is possible to analytically derive the optimal loop-nest shape without recurring to an exhaustive search of the solution space.

SmartShuttle [194] propose an adaptive layer partitioning and computation scheduling scheme to minimize the off-chip memory accesses for CNN accelerators. The scheme is based on an analytical framework to quantify the off-chip memory access volume for different layer partitioning and scheduling configurations. Unlike the Peemen's and Zhang's approach, which exhaustively searches the loops-nest tiling and reordering space, the SmartShuttle proposes to choose the best CNN *computation schedule* using an empirical rule of thumb. Based on this rule, the SmartShuttle can configure different tiling factors for each CNN layer individually, and dynamically match different layers by switching among three scheduling schemes, the *input reuse oriented* (IRO), the *output reuse oriented* (ORO), and the *weight reuse oriented* (WRO). Similar to previously discussed methods, the SmartShuttle does not take into account the possibility to share the application-managed buffer memory inside the localized iteration space.

Yang *et al.* [39] published a method for the convolution loop-nest tiling for multi-level memory hierarchies. The authors focus on improving the total energy consumption in such systems. In Yang's work, one loop *blocking* is performed for each target memory hierarchy level, building one *localized iteration space* per memory level. Each *blocking* builds a *loop order string* that indicates a particular loop order

with the 4 loops (corresponding to iterating over the output feature maps, input feature maps, feature maps' height and width) tiled at each memory hierarchy level. Blocking the convolution loop-nest at each level can be though of as tiling the loops in the loop-nest, and then exchanging the order in which the controlling loops are executed. Yang *et al.* acknowledge that optimal application of their algorithm to multiple memory hierarchy levels is quite computationally costly. In order to achieve a reasonable computation time, instead of optimal multi-level blocking, the authors then propose to apply a 2-level blocking repeatedly starting from lower memory hierarchy level to the upper, while adjusting the lower level results at each new level. However, for the 2-level blocking, since the memory level buffering requirements are estimated as the sum of all data elements touched inside the level's *localized iteration space*, this approach results in *computation schedule* quality essentially similar to the Peemen's method.

Existing memory performance evaluation methods build a unique *localized iteration space* of the CNN computation and require that internal computation buffer be dimensioned to simultaneously hold all data elements in this *localized iteration space*. We have observed that under application control, different data may be buffered at different loop-nest level, i.e. there is no one single *localized iteration space*. As a result, our memory performance model results in a more accurate buffer size estimation for application-managed buffers and in better *computation schedules*.

Among related loop-nest optimization work, memory optimization models for stencil computational kernels were published in [195] and in [196]. Stencils differ from CNN convolutional layers in that they do not need to handle a large amount of convolution kernel weights. Therefore, these models cannot be used to optimize the CNN computation.

Finally, it is worth mentioning that one important research direction that aims at alleviating the CNN memory bottleneck is data compression. Various data compression techniques have been proposed in the literature: quantization [197, 198, 199], binarization [200, 201, 202, 203, 204], and other [205]. A survey on CNN data compression can be found in [206]. The work presented in this chapter is orthogonal to these techniques and can be used on top of them. Indeed, in our HWC implementation we have used a dynamic fixed-point data quantization technique from [198]. We shall also point out that recently sparse neural networks emerged as one solution to reduce the amount of computation and memory required for the CNN processing [207, 208, 209]. This is beyond the scope of this thesis.

3.2.3 Memory Performance Model

Given a limited local reuse buffer capacity, memory performance optimization consists in finding a *computation schedule*, i.e. the computation order, such that *i)* the working set of the computation, fits in the *local reuse buffer*; and *ii)* traffic to off-accelerator memory is minimized, reducing energy and time dedicated to data movement. Therefore, using the notation introduced above, building a *computation schedule* involves specifying the loop-nest shape via loop tiling and reordering and choosing a *buffering level* for each array reference.

This section describes an analytical model for evaluating the *computation schedules* that is more accurate than previously published models in the case of application-managed buffers. Given a *computation schedule*, this analytical model computes the local buffer size and the number of bytes accessed from the off-accelerator memory required for this schedule execution. Note that though the model is developed in a context of the CNN convolution computation, in itself it is generic and applicable to other types of computation organized in loop-nests.

Computing Local Buffer Requirements

As explained earlier, the *reuse distance* of an array reference with respect to a loop corresponds to the number of iterations during which the corresponding array element is used by the computation. Table 3.3 lists the *reuse distances* of CNN convolution array references with respect to all loops in the loop-nest.

	LFX	**LFY**	**LSX**	**LSY**	**LIF**	**LOF**
I	1 or R[5]	R or 1[5]	1 or R[5]	R or 1[5]	1	M
W	1	1	E	E	1	1
O	R	R	1	1	C	1

Table 3.3: Reuse distance of arrays in CNN convolution loop-nest.

We call the portion of an array that is touched, while iterating over a loop, the *footprint* of an array inside that loop. The footprint of an array X in loop **L** measures the number of distinct elements of X used inside **L**. We need to compute the footprint of each array reference in the CNN loop-nest in order to compute the local buffering requirements. Computing footprint of an array in the convolution loop-nest is straightforward: it is a function of the number of loop iterations and takes into account any data reused over different loop iterations.

Assume $\mathbf{L}_0 \ldots \mathbf{L}_{N-1}$ are the loops in the current *computation schedule*, ordered from the innermost to the outermost one. If we call $n(\mathbf{L}_i)$ the number of iterations of loop $\mathbf{L}_i$, and $d_X(\mathbf{L}_i)$ the *reuse distance* of the

5 The reuse of array I is carried by a combination of two loops: which pair of loops – (**LFX**,**LSX**) or (**LFY**,**LSY**) – is carrying the reuse depends on relative ordering of the loops.

array reference X with respect to loop $\mathbf{L}_i$, the *footprint* $F_X(\mathbf{L}_i)$ of array X in loop $\mathbf{L}_i$ can be computed as follows:

$$F_X(\mathbf{L}_i) = F_X(\mathbf{L}_{i-1}) \cdot \frac{n(\mathbf{L}_i)}{d_X(\mathbf{L}_i)}, \forall i \in \{0, \cdots\} \tag{3.2}$$

with $F_X(\mathbf{L}_{-1}) \triangleq 1$. The *footprint* takes into account any reuse of data elements that exists in a loop. Thus, in order for the elements of array X to be reused across iterations of a loop $\mathbf{L}_i$, the local buffer must be big enough for holding the full *footprint* of one iteration of the loop. Furthermore, if the loop does not carry reuse of the array, the application-managed buffer can be shared by data elements from multiple loop iterations. This means that the actual required size for the application-managed buffer is computed as follows:

$$B_X(\mathbf{L}_i) = \begin{cases} F_X(\mathbf{L}_{i-1}) & \text{if } \mathbf{L}_i \text{ carries reuse of X} \\ B_X(\mathbf{L}_{i-1}) & \text{if } \mathbf{L}_i \text{ does not carry reuse of X} \end{cases} \tag{3.3}$$

Given a reordered and tiled loop-nest, Equation 3.3 allows to recursively compute local buffer requirements for all array references, starting from the innermost loop in the loop-nest. Therefore, it allows to evaluate the buffering requirements of a *computation schedule*, given its loop-nest shape with *buffering levels* annotated for all data references.

Off-accelerator memory traffic

Equation 3.3 allows to evaluate which *computation schedules* are feasible given a certain buffer size, as well as to compare them based on the minimum local buffer size they require, but does not give any indication on its quality in terms of off-accelerator memory traffic. Let us call T_X the memory traffic computed as number of bytes accessed in off-accelerator memory for array X, and P_X the numerical precision (in bytes) used for its storage. At *buffering level* $\mathbf{L}_i$, the number of memory accesses to X is given by the footprint of X with respect to $\mathbf{L}_i$ multiplied by the total number of times that loop $\mathbf{L}_i$ is executed:

$$T_X = P_X \cdot F_X(\mathbf{L}_i) \cdot \prod_{j=i}^{N-1} n(\mathbf{L}_j) \tag{3.4}$$

Note that T_X does not depend explicitly on the size of the local buffer, but only on the *computation schedule*, through the footprint $F_X(\mathbf{L}_i)$ and the iterations of the outermost loops. Whether the schedule fits with respect to a given local buffer capacity depends only on Equation 3.3.

Memory accesses to the O array constitute a special case as they include two distinct contributions: $E \times E \times M$ writes of the final fully computed output feature maps, and memory accesses corresponding to the accumulation of intermediate partial results (each accumulation composed of two accesses, 1 write + 1 read). Storage of accumulated partial results typically uses a different (higher) numerical

precision than that used for the final O array. The total traffic is therefore the sum of the traffic of the three I, W, O arrays:

$$T = T_I + T_W + T_{O,acc} + T_{O,final}. \tag{3.5}$$

Equations 3.4 and 3.5 enable a quantitative comparison of *computation schedules* in terms of memory traffic, which is known to be strongly correlated with energy consumption, and of course with system cost [169, 204].

Computation schedule selection procedure

Unless all input or all output feature maps along with convolution weights fit within the local buffer, it is impossible to choose a loop-nest shape such that all data reuse is exploited. The best *computation schedule* is a trade-off between loop ordering, loop tiling, and the choice of buffering level for each array reference. Furthermore, the best *computation schedule* depends on CNN layer shape: convolution kernel size, number and size of the feature maps, feature map and kernel numeric precision.

The selection of the optimal CNN convolution *computation schedule* proceeds in two steps. In the first step, the local buffering requirements for each array reference are computed, at different loop levels across an enumeration of different loop orders and loop tile sizes. This step is independent of a particular CNN layer shape because the local buffering requirements at any loop-nest level depend only on loop order and tile sizes of different loops. In the second step, given a CNN layer, a local buffer capacity and the pre-enumerated buffer requirements for the three CNN arrays, the layer is analyzed exhaustively searching for a best combination of buffering levels for these arrays.

The first step requires the enumeration of 6! = 720 loop-nest permutations. We can reduce this number by not considering permutations between the two kernel loops (**LFX** and **LFY** in Figure 3.5), and between the two image loops (**LSX** and **LSY** in Figure 3.5). These permutations can be omitted without affecting our conclusions because they result in symmetric *computation schedules*. In order to reduce the enumeration size further, tile sizes are enumerated selectively. We want to quantify how different loop orderings and tile sizes affect the number of memory accesses. For this it is not necessary to enumerate all possible tile sizes; instead we examine a sequence of monotonically increasing, power of two, tile sizes as well as tile sizes that correspond to common CNN layer configurations. The first step results in buffering requirements for each CNN array reference at each loop level for different loop-nest shapes.

With above search space reduction, the first *computation schedule* selection step yields 180 possible convolution loop-nest permutations with multiple tiling shapes each. In the second step, we search

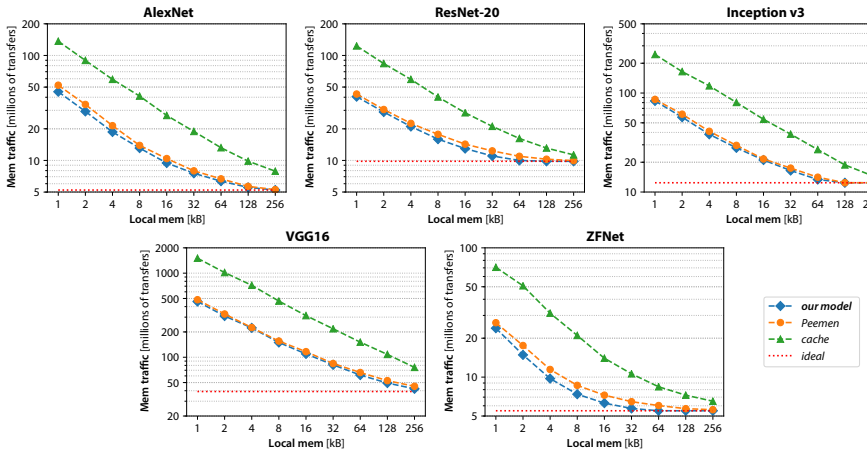


Figure 3.6: Memory traffic comparison of the best *computation schedules* identified by our model, the model proposed in Peemen *et al.* and the cache model on a set of representative CNNs, while sweeping the local memory constraint from 1 to 256 KB. Both axes are in logarithmic scale. Results aggregate traffic contributions from all convolutional layers; the dashed red line indicates the ideal memory traffic when all data reuse is fully exploited.

within this pre-enumerated loop-nest permutation space for *computation schedules* that fit given local reuse buffer capacity, while minimizing required off-accelerator memory access bandwidth. This search results in the optimal *computation schedule*, i.e. loop-nest order, tiling sizes, and *buffering levels* for CNN memory references for the given convolution layer and local buffer capacity.

3.2.4 Comparison vs Existing Models

The main use of the memory performance model is to evaluate the quality of *computation schedules* pending a set of local buffering constraints, to determine which schedule minimizes memory traffic. Therefore, to compare our model with the current state-of-the-art, we first investigated whether it can identify better *computation schedules* than those that can be extracted by previously published models [40, 37, 38, 39]. We selected two representative models, CACHE and PEEMEN, to compare with our own proposed model. Similarly to what happens with proposed model, the selection of the best *computation schedules* involves exhaustive search over a large solution space, considering all possible loop tile sizes and different loop orderings.

CACHE

M. Wolf *et al.* [40] were first to apply loop-nest reordering and tiling in order to find the *localized iteration space*. The method conservatively assumes that each data element touched in the *localized iteration space*

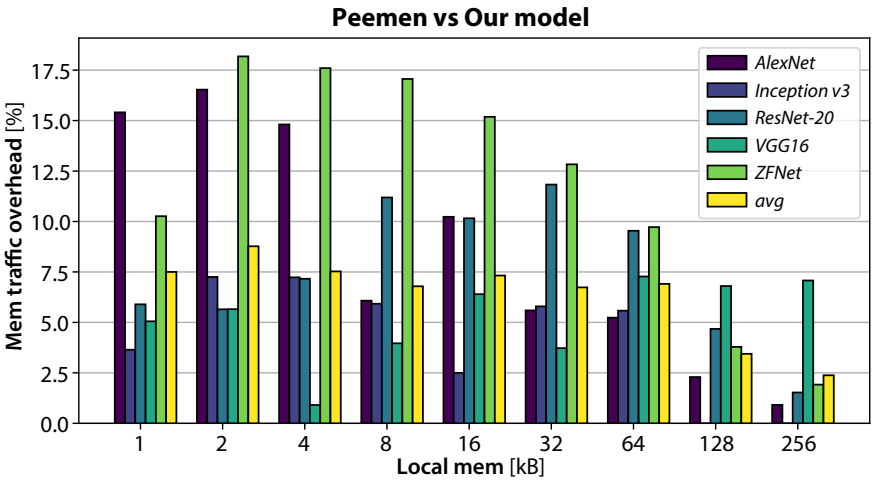


Figure 3.7: Memory traffic overhead with respect to our proposed model when using the model proposed in Peemen *et al.* to select the best *computation schedule*, while sweeping the local memory size from 1 to 256 kB.

needs to be allocated a place in the cache, thus overestimating the required memory footprint. It is also assumed that the entire *localized iteration space* working set needs to be transferred between the memory and the cache for each new *localized iteration space* execution because it cannot be guaranteed that data from a previously execution is still present in the cache. The original paper [40] also proposed a heuristics for trimming the number of tiling possibilities, guided by the cache behavior in scientific computations.

Peemen

The memory performance model proposed by Peemen *et al.* [37] and independently by Zhang *et al.* [38], improves on the cache model significantly by taking into account that with application managed buffers some data can be reused between consecutive executions of the *localized iteration space*, which they call the *innermost tile*. By changing the order of controlling loops, different quality solutions are obtained. Yang *et al.* [39] published a similar model extended for optimal CNN loop-nest tiling for multiple levels of memory hierarchy. As explained earlier, models developed by Zhang *et al.* and by Yang *et al.* yield *computation schedules* essentially similar to the schedules generated by the Peemen *et al.* method, therefore we implemented the latter as a representative model for the three approaches[6]. For a detailed explanation of Peemen's model the reader is referred to [37].

In all these methods, the ordering of the loops inside the *innermost tile* is not important because the ability to reuse data depends solely

6 Appendix B describes Peemen's memory performance model that we derived for the loop-nest from the Figure 3.5.

on the total number of data loaded to the local reuse buffer between reuses. In Peemen's and Zhang's models, only the innermost controlling loop affects the data reuse across consecutive iterations of the *innermost tile*. Therefore, fewer loop-nest permutations need to be explored reducing the solution search space.

We have compared the *computation schedules* generated by our model with *computation schedules* computed by previously published models [40], [37], [38], and [39] over the convolutional layers from five state-of-the-art CNN topologies: AlexNet, ZFNet, VGG16, Inception v3, and ResNet-20[7]. Although the following comparison is based on generated *computation schedules*, it is shown in Section 4.2 that our method is exact with respect to the real execution. Therefore, this estimation corresponds to the actual amount of memory traffic generated by these CNN layers.

Figure 3.6 plots the total number of data transfers to and from off-accelerator memory, using the best *computation schedules* estimated by the three models, while sweeping the size of the local reuse buffer from 1 to 256 KB. We aggregate the memory traffic from the best *computation schedule* identified for each layer, and we also show the ideal result given by the "essential" memory traffic that is present when all data reuse is exploited. From the plot, it is clear that the cache model largely overestimates the memory traffic requirements compared to the two application managed buffer models, and always results in sub-optimal *computation schedules* for any buffer size and for all CNN convolution layers that we tested, by a factor of up to 3.5×.

The advantage of our model when compared with Peemen's method is more subtle, as both exploit the characteristics of application-managed buffers to yield a better schedule. To analyze the difference between the two models, Figure 3.7 plots the relative overhead in memory traffic of Peemen's model with respect to our model. Figure 3.7 plots the memory traffic requirements estimated from Peemen's model as a percentage overhead compared to our model, given local reuse buffer capacities between 1KB and 256KB, for the same CNN convolution layers. Our model finds *computation schedules* with between 2.5% and 17.5% lower memory traffic, with over 10% memory traffic reduction for several CNNs, especially when targeting smaller local reuse buffer sizes. Even for a relatively large local buffer size of 128KB and 256KB, our method results in *computation schedules* with more than 5% memory traffic reduction over the set of convolution layers for several CNN networks.

For most of evaluated CNN convolution layers, our method results in some reduction of memory traffic due to its ability to exploit data reuse across all levels of the CNN convolution loop-nest. With rare exceptions, Peemen's method is able to find similar schedules only

7 Configurations of the CNN layers that we've used in our evaluations are listed in the Appendix C.

when a full volume of the input or the output feature maps can be stored in the local reuse buffer. We have noticed the following points that contribute to these results:

- Our method's footprint calculation better takes into account the data reuse because it considers independent buffering for the I, *W*, O arrays. As a result, our buffer requirements are systematically lower for a given loop-nest shape, and allow room for bigger tiles to be placed in local reuse buffers.
- Due to independent buffering of the 3 arrays, our method always places memory transfers optimally with respect to the total memory traffic.
- In Peemen's method, unless **LTIF** is the innermost controlling loop, the memory traffic for the O array is multiplied by 2 to account for 1 read and 1 write of partial accumulations. This happens even when the **LIF** loop is not tiled.

Moreover, for a given CNN convolution layer, the memory traffic overhead from Peemen's method does not necessarily decrease when the local reuse buffer capacity is increased. Increasing the local buffer capacity allows, in the first place, to generate larger tiles. However, at some buffer capacity points, our method is able to find a completely different loop ordering and buffering levels for the data references, such that more important memory traffic reduction can be achieved than by simply increasing the tile sizes.

It is worth noticing that Yang's algorithm can be modified to achieve the same quality *computation schedules* as the ones using our approach. Intuitively, it is possible to generate different localized iteration space loop orders and buffer data at different loop levels using Yang's loop order string approach. Indeed, it can be verified that applying a 4-level blocking at each memory hierarchy level - essentially enumerating the permutations of the 4 loops, **LSX**, **LSY**, **LIF**, and **LOF** from the Figure 3.5, would lead to schedules equivalent to our method. However, such 4-level blocking is quite computationally expensive: Yang *et al.* reported that a 4-level blocking of a single CNN layer takes 24 hours on a Xeon E5645 processor. Our method is significantly faster: the first step in our *computation schedule* selection procedure is performed only once for many different CNN layers, whereas in Yang's method, a solution search needs to be performed for each convolution layer individually. Even with densely sampled tiling space, for example exploring all tile sizes multiple of 2, our first step takes ~6 minutes on a simple Intel i7 processor running at 3,4 GHz. With such sampling of the tile size space, our second step applied to 71 different CNN convolution layers took less than 2 minutes per layer on the same processor.

3.3 THE COMPUTER VISION ENGINE

The CVE is a specialization of the StreamDrive dataflow architecture template along three dimensions:

1. Each programmable processor core is extended with specialized instructions common to many image processing algorithms.
2. The StreamDrive cluster is extended with a number of specialized HWC hardware elements for accelerating convolution computation.
3. Higher performance can be achieved by adding multiple CVE clusters together.

Figure 3.8 shows a generic CVE fabric with multiple clusters and several programmable PEs and several HWC elements in each cluster. The CVE can be configured at design time with respect to the number of clusters, the number of PEs and application-specific HWC blocks, the TCDM memory capacity, and the number of TCDM memory access ports. Different configurations can be adapted to target different computer vision applications and performance vs cost requirements.

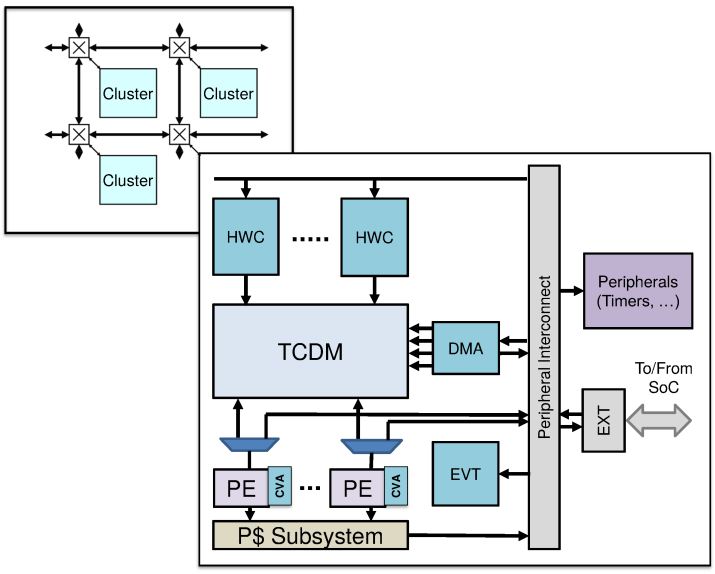


Figure 3.8: The Computer Vision Engine Clusters.

In CVE, each PE is equipped with a small dedicated set of specialized instructions (CVAx) resulting in a $2-4\times$ acceleration of common image processing functions. This set includes relatively generic instructions, such as a `MAC4CLIP`, which performs Single Instruction Multiple Data (SIMD) multiplication on bytes of two input operands, saturates the two 16-bit results, and accumulates them with the result operand; as well as instructions dedicated to specific image processing functions, such as a `XORSBCW` instruction that calculates the

Hamming distance between two vectors, it is used in Support Vector Machine (SVM) implementation. The CVAx extension development has not been part of this thesis. The PEs and application-specific HWC units are connected to the TCDM shared memory via 32-bit wide logarithmic interconnect ports. The use of shared memory enables efficient exchange of data between PEs and the HWC blocks, achieving high degree of flexibility by computing non-convolutional functions, such as CNN pooling, normalization, etc., and any other processing in software. The CVE can be configured at design time with the TCDM memory between 64KB and 512KB. The DMA engine is connected to 4 independent logarithmic interconnect ports for off-cluster access capacity of 16 bytes per processor cycle.

3.3.1 The HWC Hardware Block

The HWC is an application-specific hardware element dedicated to processing the CNN *convolutional layer*, which accounts for most of the computational work and most of the memory bandwidth in existing CNNs [169]. The HWC is meant to be connected to the shared memory in the CVE cluster. Such tight coupling of the HWC to shared memory allows usage of complex memory access patterns, such as sliding kernels, repeated re-fetch of data, etc., enabling flexible *computation schedule*. The HWC implementation also leverages on StreamDrive rotating buffer mechanism from chapter 2 for efficient handling of the application managed buffers inside the TCDM memory. The HWC block diagram is shown in Figure 3.9.

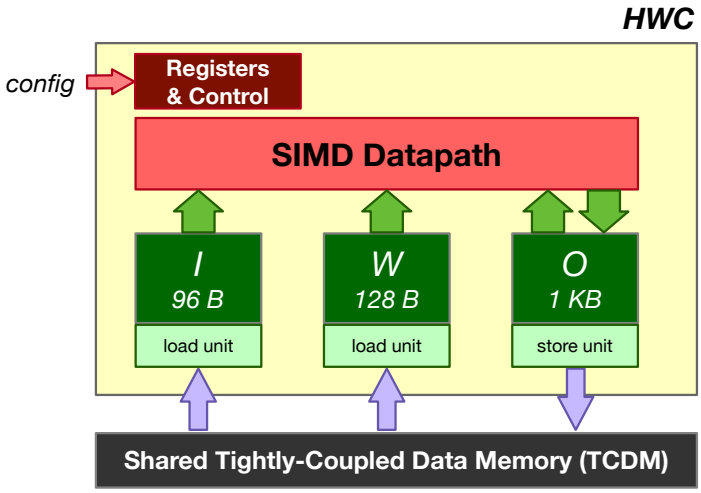


Figure 3.9: HWC block internal architecture.

As we explain in Section 4.2, we specified a HWC prototype directly in C language, using the CatapultC high-level synthesis (HLS) tool to derive a Verilog design. The architecture exposes one slave configuration port and 3 master ports towards the TCDM (one for each

array reference) to separate control for memory accesses for each array, simplifying the high-level synthesis process. The two *load units* read the input image data and the convolution kernel weights, while the *store unit* is writing the results to the TCDM shared memory. These units are responsible for generating the streaming accesses to the HBB, described in chapter 2, as well as for buffering the input and output data. The HWC does internal partial sum accumulation and does not implement an additional input for the partially accumulated CNN sums. The *Registers and Control* module implements the HWC registers and control logic. These registers are accessible from the system memory map.

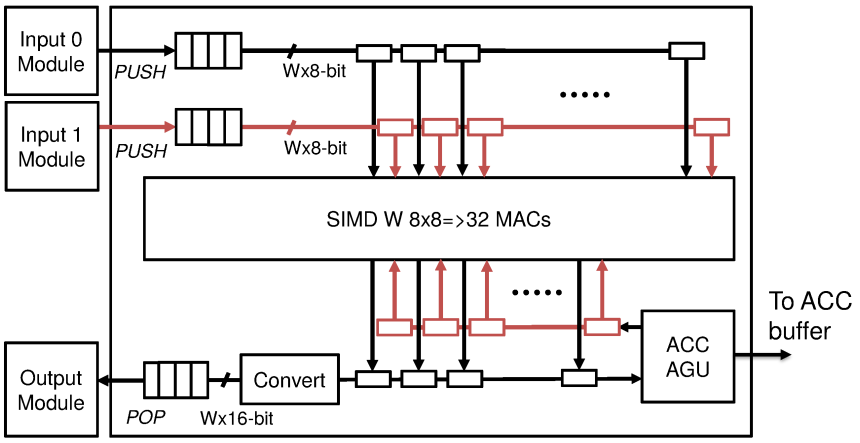


Figure 3.10: The SIMD datapath block diagram.

Figure 3.10 shows the block diagram of the main HWC module, the Single Instruction Multiple Data (SIMD) datapath. The SIMD processing ensures a steady datapath utilization independent on the convolution kernel size. The SIMD datapath can be dynamically configured to perform W 8-bit $\times$ 8-bit or $W/2$ 16 $\times$ 16 fixed-point MAC (Multiply-ACcumulate) operations per clock cycle, with accumulation precision of 32-bits. The HWC is dimensioned to handle convolution kernel sizes between 1×1 and 11×11 with stride and unconstrained input/output feature map sizes.

The SIMD datapath width W directly impacts the HWC performance in terms of MACs per clock cycle. However, a wide SIMD datapath may be underutilized: a SIMD datapath of W MAC units has a 100% *utilization* when working on input feature maps with line sizes multiple of W pixels. A SIMD processing is loosing its efficiency working on shorter lines, such as used in smaller CNN layers, or at the end of a line when there are not enough remaining pixels to fill the W MAC units in parallel. From this prospective, a narrower SIMD datapath can achieve a more efficient utilization. An advantage of proposed tightly-coupled shared memory cluster is that multiple HWC units can be added to the cluster. Therefore, each HWC can implement a relatively narrow SIMD datapath, relying on combining several HWCs

to deliver necessary performance. From our simulations, we have found that SIMD width of 16 MAC units results in the average utilization of 80% across a large set of representative CNN layers. A wider SIMD suffers from noticeable utilization drop across evaluated CNN layers, while a narrower SIMD leads to cost overhead as more HWCs need to be instantiated within a cluster, and less efficient TCDM ports usage.

Thanks to its SIMD operation and flexibility of communication system built around the shared memory, neither the convolution kernel, no feature map size does influence the HWC efficiency.

3.3.2 Optimizing HWC Memory Subsystem

A critical point in the design of the HWC, like most accelerators, is what data has to be internalized within a local buffer and what is accessed from outside the accelerator, in this case from the cluster shared TCDM. Minimizing local HWC buffer capacity is important because the HWC internal storage includes scratchpad memory and FIFOs that are implemented as dedicated physical resources distinct from the TCDM. In practice, all PEs and hardware blocks share most of the data used in processing. While PEs physically share the data via the TCDM, the hardware blocks increase the area devoted to dedicated (and redundant) buffers with each new block integrated. We make the observation that the proliferation of private buffers inside the HWC leads to significant area inefficiency and an over-provisioned memory subsystem.

Keeping shared memory bandwidth requirements as low as possible is also important. The bandwidth to cluster shared memory remains a scarce resource because multiple actors in the system are accessing it simultaneously: PEs, HWC hardware blocks, and the DMA engine. Without any local storage at all, every data access from HWC would be done to the TCDM memory. Resulting bandwidth requirement for transferring data between the TCDM and the HWC exceeds the local interconnect capacity leading to a drop in performance and to high energy consumption. Lowering the HWC to shared memory bandwidth allows reducing the number of ports used to connect the HWC to the TCDM, which impacts the logarithmic interconnect area and performance. Finally, accessing the cluster shared memory is more expensive in terms of energy consumption than accessing internal HWC storage [204].

Reducing local storage and TCDM bandwidth are generally conflicting objectives. This optimization problem is readily mapped to the conceptual view of our memory performance model (Figure 3.4), where internalized memories constitute the *application-managed local reuse buffer*, whereas the TCDM is the *off-accelerator memory*. It is therefore straightforward to apply this memory performance model as a

tool for design space exploration, with the objective to find the best trade-off between the HWC local storage capacity and the required TCDM bandwidth.

In general, the trade-off between local storage and memory bandwidth depends on the shape of the specific CNN layer: convolution kernel size, number and size of the feature maps, feature map and kernel numeric precision. Therefore, for a general-target HWC we want to build a CNN loop-nest *computation schedule* that, given a local reuse buffer capacity in the order of a few KBs, minimizes the TCDM memory traffic – and therefore bandwidth – across a large number of CNN layers taken from different CNNs.

We conducted the HWC design space exploration in two steps as described in Section 3.2. We analyzed 71 different representative CNN layers chosen from the AlexNet, ZFNet, VGG, Inception v3 and ResNet topologies, exhaustively searching for a best *computation schedule* for each of 180 different loop-nest permutations under different local buffer capacities.

In the first step of the HWC design exploration, we have enumerated CNN convolution loop-nest permutations and tile sizes, generating a set of buffer requirements for all consistent loop-nest shapes as described in section 3.2. In the second step, using these pre-enumerated buffer requirements, we analyzed 71 different representative CNN layers, exhaustively searching for a best combination of buffering levels for the three CNN arrays under different local buffer capacities.

Figure 3.11 shows the distribution of the *computation schedule* quality across the 180 loop-nest permutations obtained with different local memory constraints, binned according to the amount of memory traffic they generate. The Y-axis shows, for each local buffer capacity, the percentage of loop-nest permutations that result in optimal traffic, or add up to 10%, 20%, etc. of overhead to the optimal traffic, or exceed 2 times the optimal traffic, respectively. For small local buffer capacities, less than 20% of loop-nest permutations can achieve optimal bandwidth. On the other hand, with a large local buffer ~50% of loop permutations can be tiled in such a way that optimal bandwidth is achieved.

By analyzing *computation schedules* obtained in this experiment, we confirmed the intuition that best *computation schedules* are those that allow the output feature maps to be fully accumulated locally by buffering the partial sums, especially when the local buffer capacity is small. Although in these schedules the input feature maps and weights are read from memory multiple times, they still result in fewer total memory accesses compared to *computation schedules* where the output feature maps are swapped out to memory before being fully accumulated through all of the input feature maps. Swapping and re-fetching the output feature maps to complete the accumulation generates twice the traffic compared with the read-only input

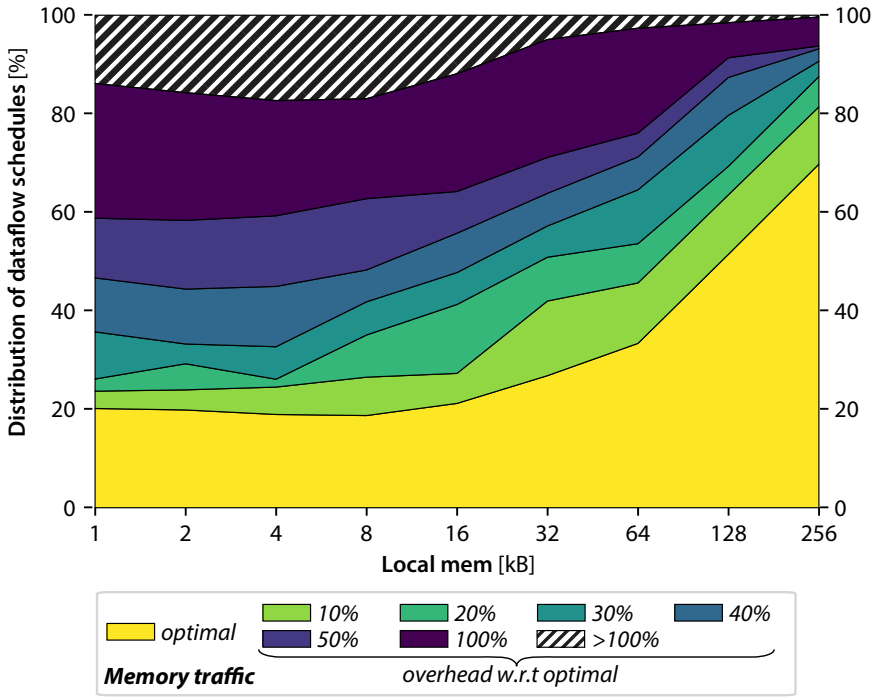


Figure 3.11: Distribution of *computation schedules* across different loop-nest permutations, binned depending on the amount of memory traffic they generate.

```
LTOF: for (mm = 0; mm < M; mm += mss)
  LTIF: for (cc = 0; cc < C; cc += css)
    LTSY: for (yy = 0; yy < E; yy += yss)
      LTSX: for (xx = 0; xx < E; xx += xss)
        // Buffering O
        LIF: for (c=cc;c<min(cc+css,C);c++)
          // Buffering I
          LSY: for (y=yy;y<min(yy+yss,E);y++)
            LFY: for (k = -R/2; k < R/2; k++)
              // Buffering W
              LOF: for (m = mm; m < mss; m++)
                LSX: for (x=xx;x<min(xx+xss,E);x++)
                  LFX: for (l = -R/2; l < R/2; l++) {
                    p = I[c][y*S+k][x*S+l]
                    w = W[m][c][k][l]
                    O[m][y][x] += p*w
                  }
```

Figure 3.12: HWC *computation schedule*: gives best bandwidth trade-off with local storage capacity less than 4KB.

feature maps and weights. Furthermore, the partially accumulated output feature maps require higher numerical precision and therefore are more costly in terms of required bandwidth. It is interesting to notice that given less than 512KB buffering capacity, no one per-

```
LCOF: for (mmm = 0; mmm < M; mmm += Tm)
  LCIF: for (ccc = 0; ccc < C; ccc += Tc)
    LCSY: for (yyy = 0; yyy < E; yyy += Ty)
      LCSX: for (xxx = 0; xxx < E; xxx += Tx)

        LTOF: for (mm=mmm; mm<min(mm+Tm,M); mm+=mss) {
          LTIF: for (cc=ccc; cc<min(cc+Tc,C); cc+=css) {
            LTSY: for (yy=yyy; yy<min(yy+Ty,E); yy+=yss) {
              LTSX: for (xx=xxx; xx<min(xx+Tx,E); xx+=xss) {

                  HWC (mss,css,yss,xss)
```

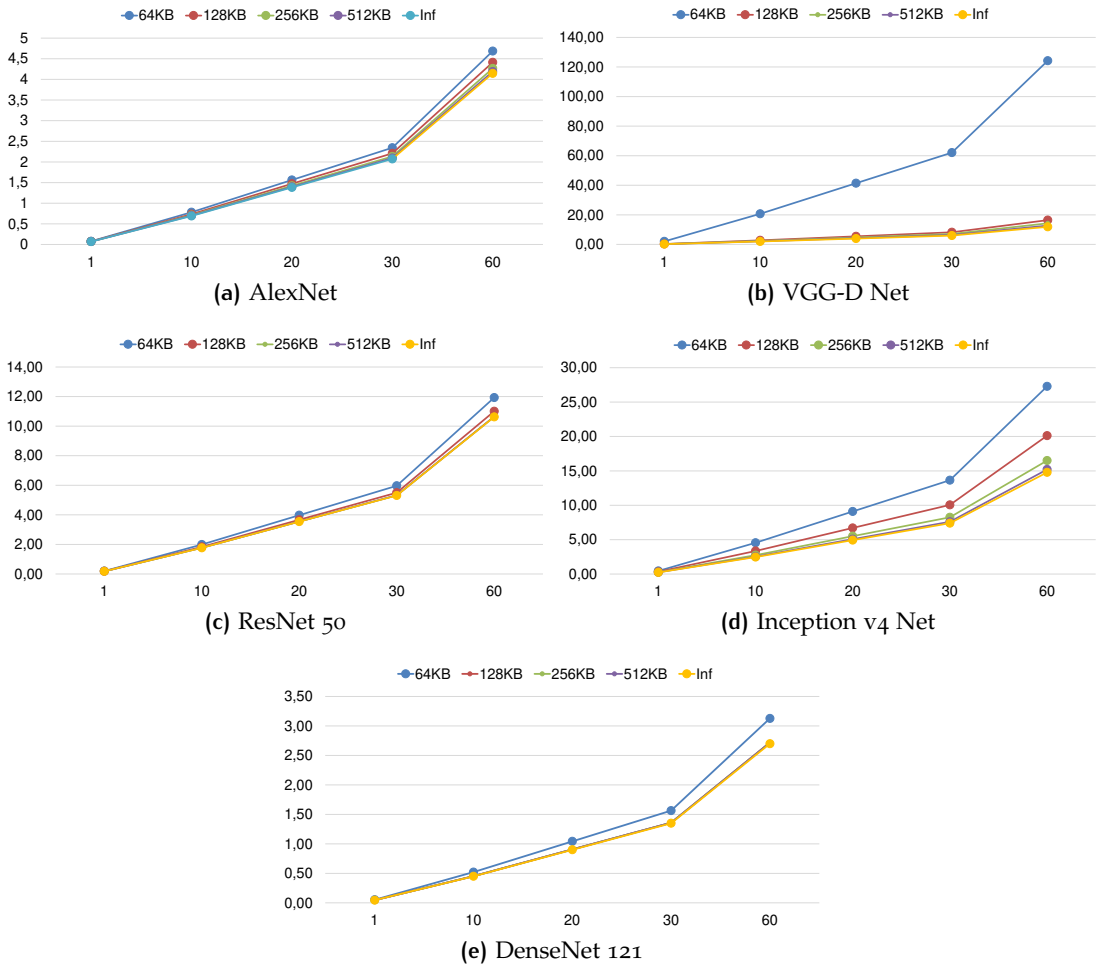
Figure 3.13: Cluster-level CNN convolution loop-nest.

mutation resulted in a *computation schedule* with optimal bandwidth across the entire set of tested convolution layers.

For the HWC implementation, among several small footprint schedules, we selected one that, for most tested CNN layers, leads to minimal memory bandwidth requirements for local buffer sizes from 1KB to 4KB. This selection was also guided by several hardware implementation criteria, such as the number of required simultaneous local buffer accesses, access alignment, etc. and the compatibility with a SIMD datapath. Figure 3.12 shows the *computation schedule* selected for the HWC implementation, with the *buffering level* for each array shown as a comment on top of the corresponding loop. The HWC main loop executes an innermost tile in order **LIF** - **LSY** - **LFY** - **LOF** - **LSX** - **LFX**. The relative order of loops **LSX**, **LFX** and **LFY** ensures that the partial sum accumulation remains internal to the HWC as much as possible. In the actual implementation, the tiling factor xss for the **LSX** loop is fixed and equals the SIMD datapath width. The remaining tile dimensions: the number of output feature maps, mss, the input feature maps, css, and the number of output lines in a tile, yss, are computed for each particular convolution layer also using our performance model. In practice, over all tested CNN layers, the input feature maps volume was never tiled ($css = C$), allowing a complete accumulation of the partial sums inside the HWC local buffer.

3.3.3 Optimizing Off-Cluster Memory Access

At the cluster level, the TCDM shared memory is not big enough for most CNN convolution layers to be completely stored. The convolution computation volume needs to be tiled in order to fit the available TCDM memory while ensuring that off-cluster memory accesses are minimized. With respect to our analytical memory model, we consider the TCDM cluster shared memory as *local reuse buffer*, while the off-cluster memory as the *off-accelerator main memory*. Figure 3.13 shows the cluster level tiling of the convolution loop-nest.



(a) AlexNet (b) VGG-D Net

(c) ResNet 50 (d) Inception v4 Net

(e) DenseNet 121

Figure 3.14: Off-cluster bandwidth requirements vs performance of different CNN networks under various TCDM memory capacities.

As explained earlier, for each particular CNN layer configuration, the tile sizes mss, css, yss, xss for the HWC controlling loops need to be computed. We also need to search for a cluster level *computation schedule*: the ordering of HWC controlling loops **`LTSX`**, **`LTSY`**, **`LTIF`**, and **`LTOF`**, the T_m, T_c, T_y, T_x tile sizes, and the data *buffering levels*. Thus, the computation of the HWC configuration (tile sizes) and the cluster level *computation schedule* are interdependent. If the HWC tile size is fixed, it creates an additional constraint on the cluster level *computation schedule*, and vice-versa. Because optimizing off-cluster memory traffic has bigger impact in terms of performance and power consumption than reducing the internal HWC to TCDM memory traffic, we first compute the cluster level *computation schedule*, setting the $T_m \times T_c \times T_y \times T_x$ volume, then we compute tile sizes for the HWC loop-nest corresponding to this volume.

Thus, overall, an implementation of a CNN network requires computing two sets of parameters for each CNN convolution layer: (1)

a *computation schedule* for the cluster level loop-nest, and (2) the tile sizes for the HWC loop-nest.

We have applied the above methodology for tiling a number of well-known CNN networks, the AlexNet [172], the VGG [210], the ResNet [211], the Google Inception v4 [212], and the DenseNet [213]. Figure 3.14 shows the resulting optimal off-cluster bandwidth requirements of these CNN networks vs the TCDM memory capacity. The Y-axis plots the bandwidth (GB/sec.) required by a particular network vs the frame rate (frame/sec.) along the X-axis. Achieving the frame rate of 30 frames/sec. requires only 10 GB/sec. for all but the VGG-D network with less than 256KB of TCDM memory. This methodology can also be used for determining the adapted off-cluster memory technology for a given CNN network class.

3.4 SUMMARY

In this chapter, we have described the Computer Vision Engine (CVE), which is a specialization of the StreamDrive platform for image processing applications.

The main element of the CVE is a tightly-coupled convolution block, HWC, designed to efficiently support CNN convolution processing. The HWC implements non-trivial *computation schedule* which allows minimizing the usage of the TCDM bandwidth with very little internal HWC buffering capacity. We have derived this *computation schedule* using a novel analytical memory performance model for CNN computation optimization. Our analytical model is more accurate than existing models when applied to applications that manage data buffers explicitly. Such application-managed buffers are the most common architecture template with embedded convolution accelerators. Furthermore, our analytical model is general and can be applied to computations other than CNN convolution. Finally, based on this analytical model we have developed a method for determining the optimal convolution *computation schedule* for minimizing the off-cluster memory bandwidth under a limited TCDM memory capacity.

4 PERFORMANCE EVALUATIONS

Experience serves not only to confirm theory, but differs from it without disturbing it, it leads to new truths which theory only has not been able to reach.

— Dalembert

In previous chapters, we have described the StreamDrive framework and its *Convolution Vision Engine* (CVE) implementation. In this chapter, we will use the CVE for a performance evaluations of computer vision applications.

In section 4.1, we investigate traditional image processing applications in order to characterize the dynamic dataflow execution model in the context of a shared memory cluster. Particularly, we evaluate the StreamDrive parallelization and scheduling overhead, and performance scalability of the implementation. We investigate the effects of the trade-off between cluster shared memory capacity versus performance and scalability. With the latter evaluation, we want to find out how well StreamDrive scales along two dimensions: the number of processing elements in a CVE cluster, and the latency of off-cluster memory access. We *quantify* the performance vs memory capacity trade-off by considering performance in the context of Amdahl's law [214]. In section 4.2, we characterize the cluster shared memory traffic generated by the HWC convolution hardware blocks. We quantify the HWC memory bandwidth requirement using several well-known CNN networks.

In order to evaluate different CVE configuration options, we conducted performance characterization using the time-approximate platform simulator. In our simulator programmable cores are simulated with less than 10% inaccuracy compared to a real execution. The HWC model is built directly from the CatapultC code, where the CatapultC interfaces have been adapted to communicate with the rest of the platform. The simulator also models memory and interconnect conflicts giving a very reasonable simulation accuracy at decent simulation speed. We then have derived a CVE configuration targeting mobile camera applications with resolutions not exceeding VGA quality for an FPGA implementation.

4.1 CASE-STUDY 1: IMAGE PROCESSING

In this section, we answer the question what are the characteristics of traditional image processing applications implemented using the dynamic dataflow model of computation in the context of a parallel shared memory cluster with limited memory capacity. We use our time-approximate platform simulator to evaluate different CVE configuration options.

In general, computer vision algorithms implement two types of processing, the feature detection and the descriptor generation. The feature detectors examine every pixel in an image in order to determine if it meets the criteria of a feature. A good example is the *Features from Accelerated Segment Test* (FAST) feature detector [215] used with the ORB application. FAST determine keypoints of interest by finding rapid changes in direction on image edges. Feature detectors are highly data parallel but this parallelism is often unbalanced; the image pixels that satisfy the feature criteria require more processing than "uninteresting" pixels.

Feature descriptor generation requires computation across independent image patches. For example, with binary descriptors, such as the BRIEF [216], a patch centered around a detected keypoint needs to be described as a binary string. Binary detectors use a sampling pattern, pick N pairs of points on the pattern and determine whether the first element or the second element of the pair is greater than the other and define the pair as binary 1 or 0 correspondingly. The resulting N-bit vector is the feature descriptor for the patch to be used for feature matching. This requires computations with higher complexity and less data parallelism compared to the feature detection, with irregular memory access patterns to non-contiguous image patches. On the other hand, the processing is usually well balanced across different patches of interest.

Given above general observations, an efficient implementation of a computer vision algorithm requires pipelining unbalanced tasks of the feature detection, pipelining the feature detection with feature descriptors generation, and pipelining different pyramid levels (avoid a barrier between the pyramid levels). Our performance evaluation shows that the dataflow execution model has significant benefits:

- The dynamic dataflow implementation leads to efficient pipelining of application tasks, efficiently hides off-cluster access latency, and achieves near optimal load balancing between parallel processing elements.
- We were able to perform extensive application exploration with respect to parallelism, its granularity, pipelining, etc., in a very short period of time.
- Debugging dataflow implementation is simple.

- The dynamic dataflow implementation is scalable with respect to the number of processing elements, TCDM memory capacity, and off-cluster memory access latency.

We proceed with detailed performance characterization of two typical image processing applications: the Oriented FAST and Rotated Brief (ORB) [25] already introduced in section 2.4, and the Face Detection (FD) [26]. Since our first implementation targets low-resolution applications, we perform the performance characterization on a 640 $\times$ 480 VGA image containing about 12.000 FAST keypoints for the ORB, and a 320 $\times$ 240 QVGA image containing 10 human faces for the Face Detection. These two images are representative of the most demanding processing requirements for both applications.

Using the StreamDrive framework, we parallelized the reference sequential implementation, optimized the processing pipeline, and performed extensive performance exploration of the two applications. Each application dataflow graph has been parameterized with respect to the target platform configuration. Thus there is a dataflow graph tuned for each platform configuration tuple $(P : M)$:

1. P is the number of available processing elements, which affects how many dataflow actors are instantiated.

2. M is the capacity of cluster TCDM memory, which determines the buffering size of dataflow channels.

Using the StreamDrive successive refinement development flow, in less than 6 weeks, we have parallelized both applications and explored different parallelization strategies and dataflow graph parameters for tuples with P from 1 to 16 processing elements and M in 64KB, 128KB, 256KB, and 512KB.

4.1.1 ORB

The ORB algorithm tries to identify a set of objects inside an image and computes their BRIEF descriptors. Figure 4.1 shows the functional block diagram of the reference ORB algorithm.

The processing is applied to an image pyramid of 8 scaled down images, generated by *Scale*. At each pyramid level, the objects are identified by first detecting the *keypoints* of interest via the FAST algorithm [217]. Irrelevant FAST keypoints are dropped in *Nonmax*, and the remaining keypoints are sorted by *Cull* function based on *Harris* scores [218]. For each "good" keypoint, the algorithm computes orientation of the object around the keypoint, *Angle*, and the object's BRIEF descriptor. The *Angle* computation inspects a N $\times$ N patch around the keypoint from the scaled image. The BRIEF computation requires a M $\times$ M patch from the Gaussian blurred image produced by the *Gauss* function.

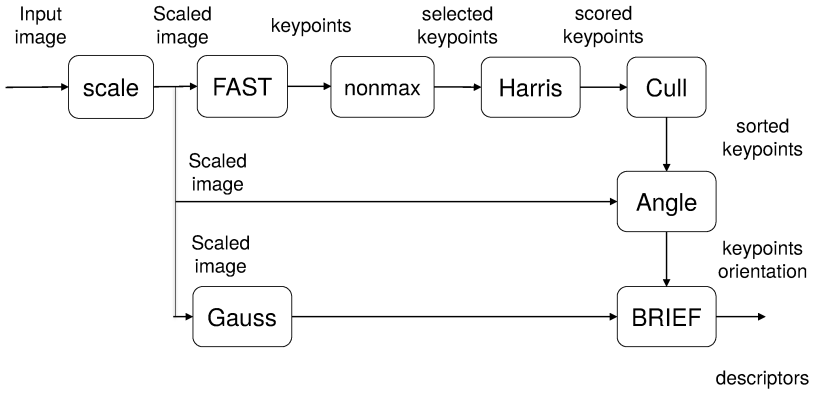


Figure 4.1: Functional blocks from reference ORB algorithm.

As explained earlier, the image pyramid scaling is done outside of the CVE cluster by a specialized scaler hardware block. Therefore, the input to our ORB implementation is a sequence of scaled images. Along with several obvious parallelization choices, eg. FAST, Harris, Angle, and BRIEF computations are naturally data-parallel, ORB also puts to evidence several parallelization difficulties:

- ORB computation is largely spatially unbalanced - some parts of the image may not have any keypoints, while others contain many; this is challenging for the runtime load balancing strategy.
- The Nonmax and the Cull computations are sequential and serialize the processing; they account for non-parallelizable part from the Amdahl's law standpoint and limit performance scaling.
- The Cull that performs sorting of keypoints creates a synchronization barrier - Angle and Brief computations cannot start until all keypoints in an input image have been detected and sorted; efficient pipelining of consecutive image pyramid images is necessary in order to fill processing elements with work while sorting is performed.
- The access pattern to image patches from the Angle and the Brief functions is irregular and cannot be predicted in advance; this requires efficient dynamic pipelining in order to hide memory transfers latency with computations.
- Smaller scaled-down images require much less work compared to larger images at lower scaling factors; this requires low-overhead runtime implementation and scheduling in order to efficiently handle actors with small workloads.

Figure 4.2 shows the ORB StreamDrive dataflow graph. The FAST, Harris, Angle, and Brief actors are data-parallel such that several

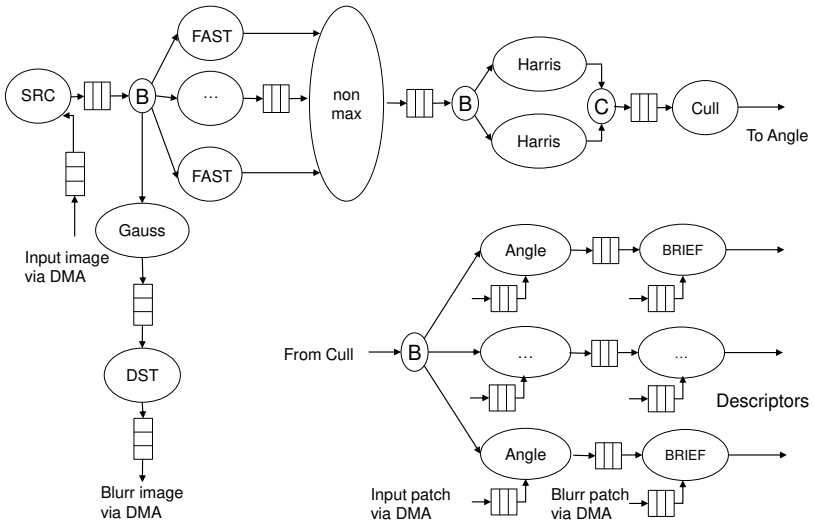


Figure 4.2: ORB StreamDrive dataflow graph.

instances of each can be created depending on number of available processing elements. The Gaussian blur is performed using the HWC convolution block. The DMA is used to perform several memory transfers: (1) reading input scaled images from off-cluster memory, the DMA is managed by a special SRC actor; (2) writing the Gaussian blur image to off-cluster memory, the DMA is managed by a special DST actor; (3) reading input scaled image patches used by the *Angle* actor, the DMA is managed by the actor itself; and (4) reading blur image patches used by the *Brief* actor, the DMA is also managed by the actor itself.

Table 4.1 reproduces the Table 2.3 from chapter 2 showing ORB actors token granularities. Thus, the FAST actor accepts a window of 7 input image lines and generates a number of keypoints corresponding to that window. All remaining actors's inputs are keypoints generated by previous actors. In addition to keypoints, the *Harris*, the *Angle* and *Brief* actors require small image patches of 9×9, 31×31 and 41×41 pixels, respectively. In our implementation, the *Harris* is processing keypoint line-by-line in raster scan order, and therefore accepts a window of 9 lines of input image in addition to the FAST keypoints. The *Angle* and *Brief* actors process keypoints in order sorted by the *Cull*, they take small input image patches as additional input.

The StreamDrive ORB implementation relies on the HWC convolution block for computing the Gaussian blurred image. The Gaussian function implements a 7×7 filter over a 640×480 image. At 16 MAC/cycle that the HWC is able to perform, the *Gauss* processing can be done in a single HWC block, in parallel to the rest of the ORB computation.

# Actor	Port	Token size
FAST	IN	One image line
	OUT	One keypoint
NONMAX	IN	One keypoint
	OUT	One keypoint
HARRIS	IN	One keypoint
	REF	One image line
	OUT	One keypoint
CULL	IN	One keypoint
	OUT	One keypoint
ANGLE	IN	One keypoint
	REF	One image patch
	OUT	One keypoint
GAUSS	IN	One image line
	OUT	One image line
BRIEF	IN	One keypoint
	BLUR	One image patch
	OUT	One descriptor

Table 4.1: Granularity of actors in ORB dataflow graph.

Altogether, our ORB implementation scales up to 54 actors, including up to 16 FAST, 16 Angle, and 16 Brief actors, as well as 2 Harris actors.

4.1.2 Face Detection

The Face Detection application detects faces in an input image.

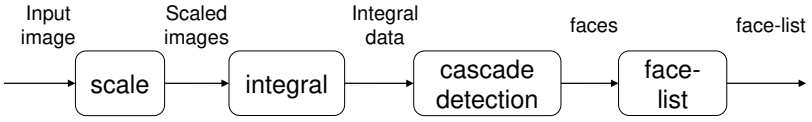


Figure 4.3: Functional blocks from reference Face Detection algorithm.

As the Figure 4.3 shows, reference FD algorithm describes a pipeline of four main functions: the Scale that creates the image pyramid with 16 scaling levels; the Intg generating an *integral image* and a square *integral image* for each scaled input; the Cascade function that scans the integral image *patch-by-patch* (a small rectangle), analyzing these patches through a cascade database of features; when a candidate face is found, the patch is sent to the List function, which checks and sorts all candidate faces in order to produce a list of found faces. The Cascade performs the pattern-matching and is by far the most time-consuming of all functions. The main difficulty in efficiently implementing the FD is that the Cascade processing is very unbalanced: the patch workload varies greatly (100 times or more) from one im-

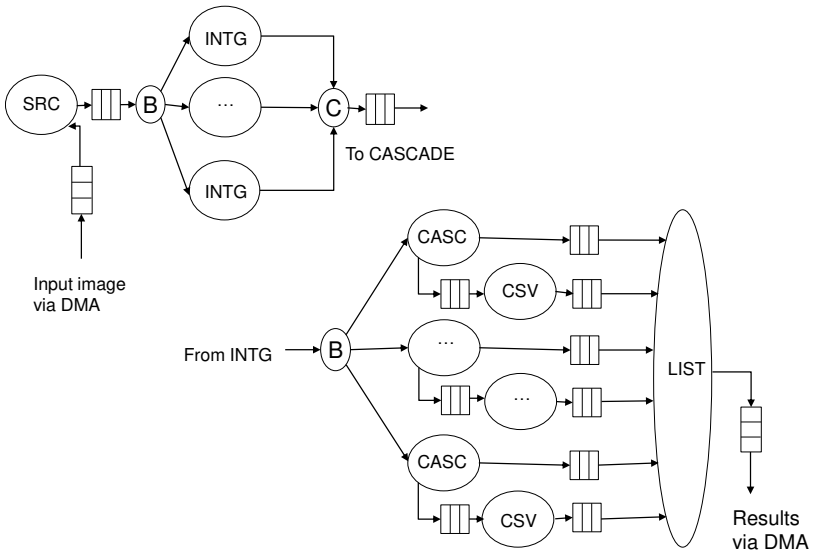


Figure 4.4: Face Detection StreamDrive dataflow graph.

age patch to other. Thus, while one of the actors blocks the processing pipeline with a heavy workload patch, other actors risk to remain waiting for the processing pipeline to unblock. We have implemented a special *Cascade Slow Vehicle* (CSV) actor, which handles the heavy workload patches while more numerous low-workload patches continue to be processed by the regular Cascade actor. The CSV actors allow the out-of-order patch processing enabling the regular patches that follow a heavy workload patch to be completed before the heavy workload patch completes.

Figure 4.4 shows the Face Detection StreamDrive graph. Similar to ORB, the image pyramid is built in a pre-processing step outside of the FD processing. The three main actors, the Intg, the Cascade, and the CSV are implemented as data-parallel actors. There are no application-specific hardware elements used by the FD implementation, this would require very narrowly specialized hardware. Therefore, all processing is done is software. The DMA is used to read the input scaled images from off-cluster memory, and to write the resulting face list to the off-cluster memory at the end of processing. The DMA transfers for the input images are managed by special SRC actor, while writing back of results is handled by the List actor itself.

Table 4.2 shows FD actors token granularities:

The FD actors token granularity is straightforward. The Intg accepts the input image one line at a time and generates integral image patches. These patches are then processed by the Cascade and by the Csv actors that generate face descriptors for those patches that contain faces. The List takes the face descriptors, matches them with the face database and outputs the matching faces as result.

The FD implementation is a full software implementation (except for the image pyramid scaling) with no application-specific hardware

# Actor	Port	Token size
INTG	IN	One image line
	OUT	One integral image patch
CASCADE	IN	One integral image patch
	OUT	One face descriptor
CSV	IN	One integral image patch
	OUT	One face descriptor
LIST	IN	One face descriptor
	OUT	One face

Table 4.2: Granularity of actors in FD dataflow graph.

elements used. Our Face Detection implementation scales up to 42 actors, including up to 8 Intg, and up to 16 Cascade and CSV actor instances.

4.1.3 StreamDrive Parallelization Overhead

The parallelization overhead is a penalty paid for parallelizing an application. The StreamDrive parallelization overhead results from (1) the communication overhead including the *reserve*, *push*, *pop*, and *release* functions; (2) the DMA management for moving the data between the off-chip memory and the cluster TCDM; and (3) the runtime scheduler overhead including the *broadcast* and *collect* synchronization. The communication and the DMA management overhead is *scalable*, i.e. from Amdahl's law perspective it contributes to the parallelizable part of the application. The runtime scheduler overhead, on the other hand, grows with the number of actors and communication channels. It is important that the scheduler has as low-overhead as possible because from Amdahl's law standpoint, it contributes to the non-parallelizable part of the application. In order to characterize the StreamDrive parallelization overhead, we measure the performance of the ORB and FD dataflow implementations configured for 1 PE and 512KB of TCDM with off-cluster latency of 1 processor cycle. In this configuration the off-cluster data transfers are completely hidden and do not affect measured application cycle count.

Figure 4.5 shows the breakdown of ORB actors execution time into times spent in computation, in the communication API, and performing the DMA management tasks. The overheads from the FAST, the Angle and the Brief actors are small compared to the computation part and, most importantly, these actors are parallelizable including this overhead. The Harris actor performs relatively little computation per input and suffers from higher communication overhead, 24.2%. However, this overhead is also parallelizable. The Nonmax and Cull actors also have heavy communication overheads of 35.0% and 18.7% respectively. These actors, however, cannot be parallelized and there-

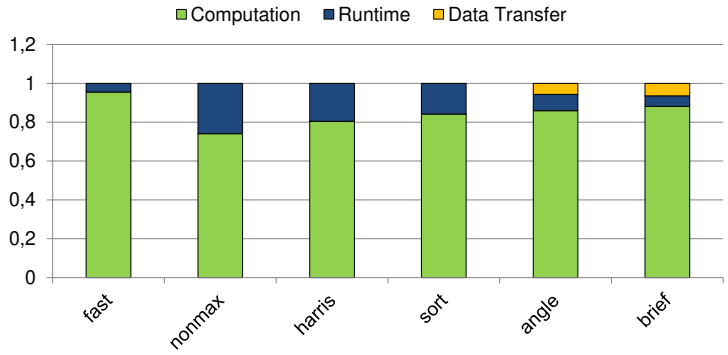


Figure 4.5: StreamDrive parallelization overhead: ratio of time spent in computation vs. data transfer vs communication API, for the ORB application.

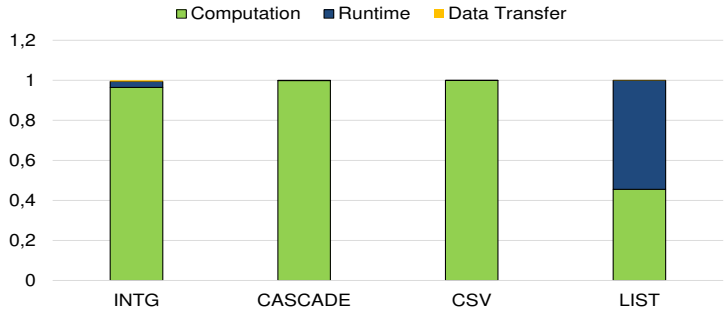


Figure 4.6: StreamDrive parallelization overhead: ratio of time spent in computation vs. data transfer vs communication API, for the FD application.

fore contribute to performance scaling degradation (explained later). Altogether, the three actors, the Nonmax, the Harris and the Cull, perform each relatively little computation per input. One possibility that we explored is to merge these three actors into a single bigger actor. However, this only works well when the total number of actors is low because the Nonmax and the Cull cannot be parallelized and the resulting merged actor is difficult to load balance with the rest of the application. The better performance is achieved by not combining these actors in favor of a better load balancing. We have observed that even though the relative overhead penalty seems high, the combined total overhead of the three actors in terms of application processing time is less than 0.5% (not counting the Gaussian filter). The Angle and the Brief actors overhead count includes both, the communication and the DMA management overhead, because they manage the DMA for transferring reference image patches around each keypoint from off-cluster memory to the TCDM. Their data transfer management overhead is 6.2% and 6.7%, respectively. This overhead corre-

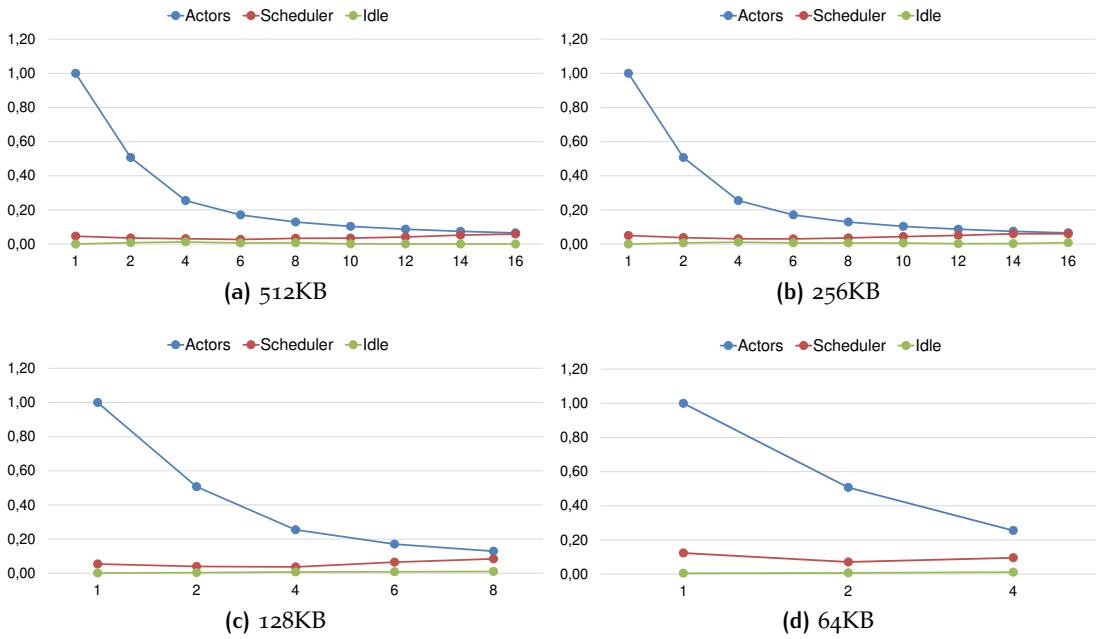


Figure 4.7: Ratio of time spent, on average, by each PE in actor computation, runtime scheduler, and the idle, for the ORB application.

sponds to many relatively small DMA transfer requests for reference image patches.

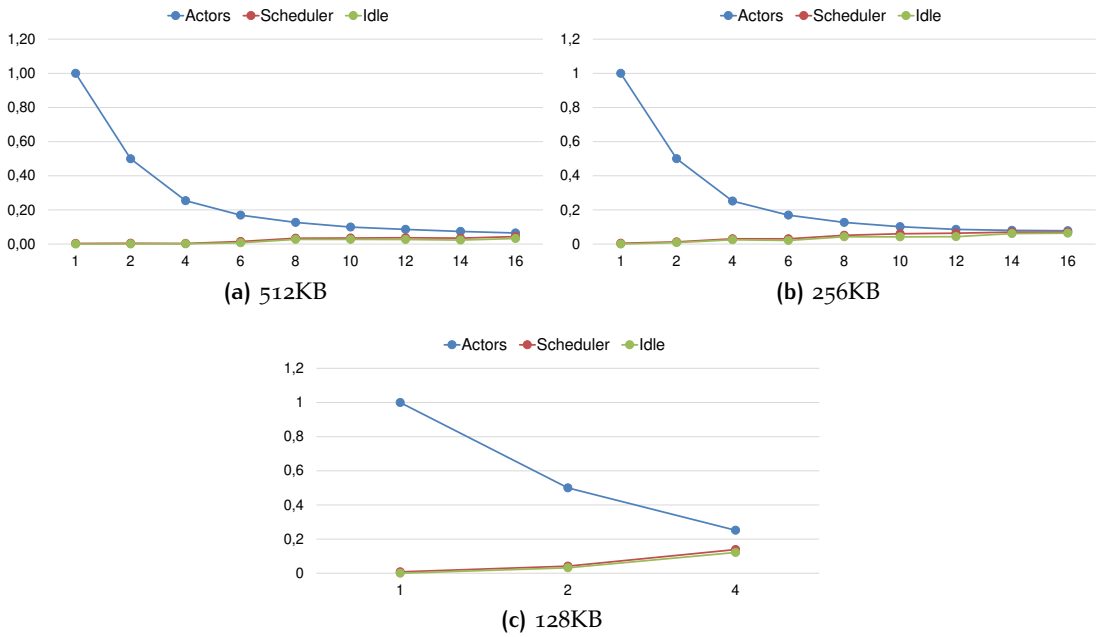
Figure 4.6 shows similar breakdown of FD actors execution time. The StreamDrive communication overhead is only noticeable with the List actor which performs very little work per input face. There is no need to further optimize this since the List only account for 0.000332% of the FD total execution time.

It is interesting to consider our results in a context of existing state-of-the-art runtime environments. Compared to the KPN implementation by Haid, the StreamDrive synchronization is faster: less than 40 processor cycles per blocking access (a *reserve* or a *pop*) on average versus 150 reported in [23].

The scheduling overhead is affected by the number of actors and the number of communication channels in the application (including the number and size of the *broadcast* and the *collect* connections).

Figures 4.7 and 4.8 show the percentage of time spent on average by each processing element in actor computation, runtime scheduler, and the idle time, for different dataflow graph configurations. The Y-axis show the percentage of time that each PE spends on average in different parts of the processing; the X-axis plots the number of PEs active in various dataflow graph configurations. There are plots for dataflow graph configurations with 64KB[8], 128KB, 256KB, and 512KB of available TCDM memory. For each TCDM capacity, the number of

8 The smallest FD configuration requires 128KB, therefore there is no 64KB plot for the FD application.



(a) 512KB

(b) 256KB

(c) 128KB

Figure 4.8: Ratio of time spent, on average, by each PE in actor computation, runtime scheduler, and the idle, for the FD application.

dataflow actors increases with the number of PEs. For example, a 1 PE ORB configuration contains 8 actors, while a 16 PE configuration contains 54 actors. Similarly, the 1 PE FD configuration contains 5 actors, while the 16 PE configuration has 42 actors. The StreamDrive runtime scheduler is very efficient: the time spent in the scheduler remains low over different number of actors, under 6% for the largest ORB configuration and under 4% for the largest FD configuration. These numbers also show how efficient our *broadcast* and *collect* connections are - the scheduling overhead remains low going from none (1 PE) to heavy broadcast and collect operation (16 PEs).

The StreamDrive distributed scheduler results in near-optimal *load balancing*. There is virtually no idle time with the StreamDrive ORB implementation. With the FD implementation, the idle time grows up to 12% per processing element in a 128KB TCDM and 4 PE configuration. The reason is that occasionally there is a very long latency *Cascade Slow Vehicle* task that blocks the pipeline sufficiently longtime for all dataflow buffers to fill, freezing all PEs but one. The problem is alleviated with increased TCDM buffering capacity but cannot be completely hidden even with large amount of buffering with TCDM capacity 512KB. The StreamDrive scheduler's efficiency is not affected by the buffering capacity - the scheduler overhead remains virtually constant under different TCDM memory sizes.

4.1.4 Memory Footprint

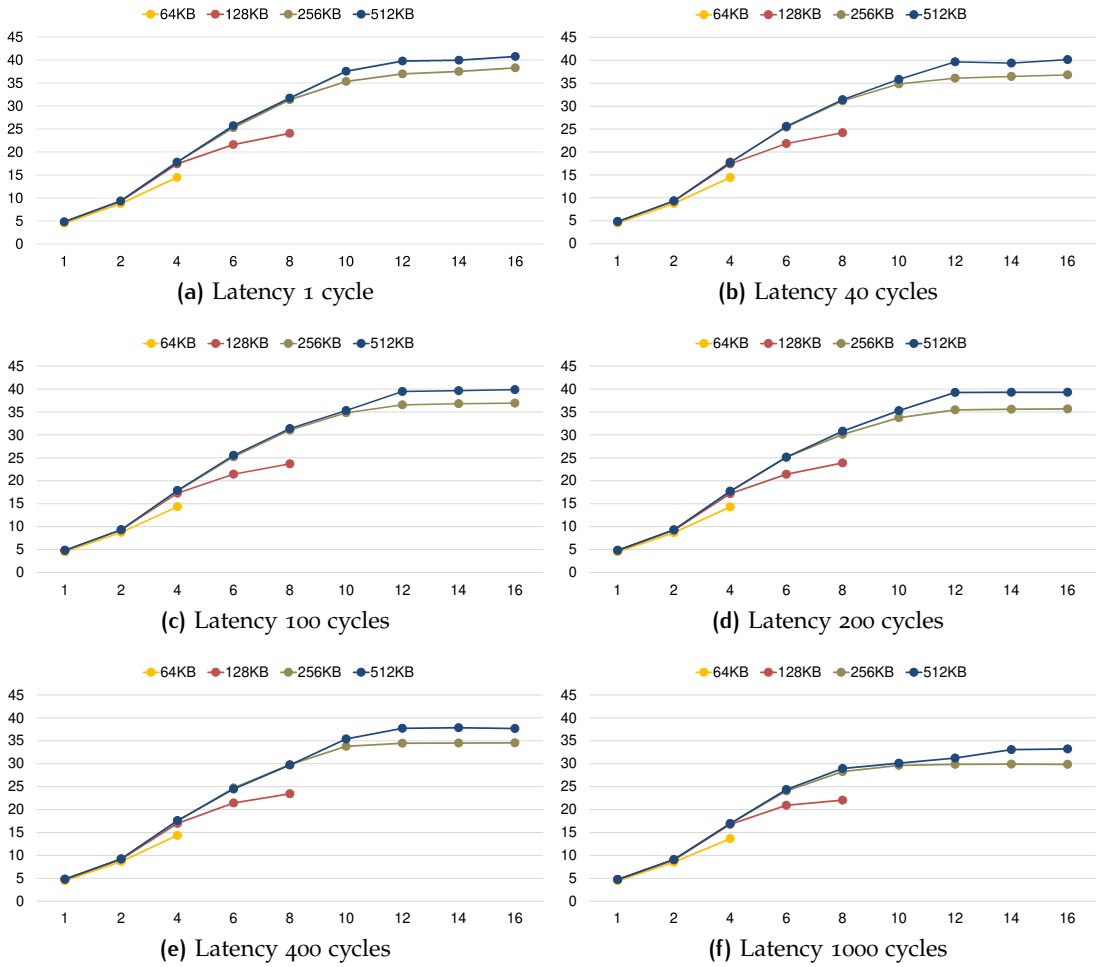
Application memory footprint determines how much memory the application needs for execution. The StreamDrive application memory footprint includes the application data, the runtime system including the runtime stack, and the dataflow buffers. There is an important design point trade-off between the application performance and its memory footprint. While the application data and runtime stack are specific to each particular implementation, the increase in dataflow buffers capacity leads to higher performance. On the other hand, large memory sizes may not be affordable for low-cost embedded implementations.

In terms of the run-time system memory requirements, the debug version of the StreamDrive library uses 944 bytes of static data. It also needs 64 bytes of memory per actor in addition to actor private data, and up to 60 bytes per communication channel, depending on channel type. In comparison, an image line of a VGA image has a size of 640 bytes. While the smallest ORB keypoints buffer requires almost 300 bytes, the buffer capacities in FD implementation all exceed several KBytes. The stack contribution is application-specific and depends on the size of biggest stack that any one actor may require. As explained in section 2.5.3, the StreamDrive implementation allocates one runtime stack per processing element inside the TCDM memory. In our ORB and FD implementations, 2KB of the stack space per actor is enough even for a debug version of applications with `printfs`. Thus, a maximal 16 PE dataflow graph configuration required 32KB of stack space for the 16 processing elements.

The application buffering requirements are determined by the actor granularity along with the capacity of the dataflow communication channels. Every dataflow channel requires a minimal FIFO buffer size that ensures a deadlock free execution [9]. Additional buffer capacity beyond such minimal size helps improve performance by reducing scheduler overhead and by absorbing communication peaks when actor computation is irregular and unpredictable.

Figures 4.9 and 4.10 summarize the performance improvement associated with increasing the buffering capacity for the ORB and the FD applications. Each figure plots the frame rate, in terms of the number of frames/sec. achieved by different dataflow graph configurations at operating frequency of 500 MHz. There is one figure for each of 6 different off-cluster access latencies. For example, with off-cluster latency of 1 processor cycle, the ORB implementation achieves the frame rate of 15 frames/sec. with 4 PEs and only 64 KB of TCDM memory. With the same off-cluster latency, the ORB implementation with 16 PEs and 512KB of the TCDM memory can achieve more than 40 frames/sec. performance. Some dataflow graph configurations do

9 Unless there is uncontrolled accumulation of tokens in a channel.



(a) Latency 1 cycle

(b) Latency 40 cycles

(c) Latency 100 cycles

(d) Latency 200 cycles

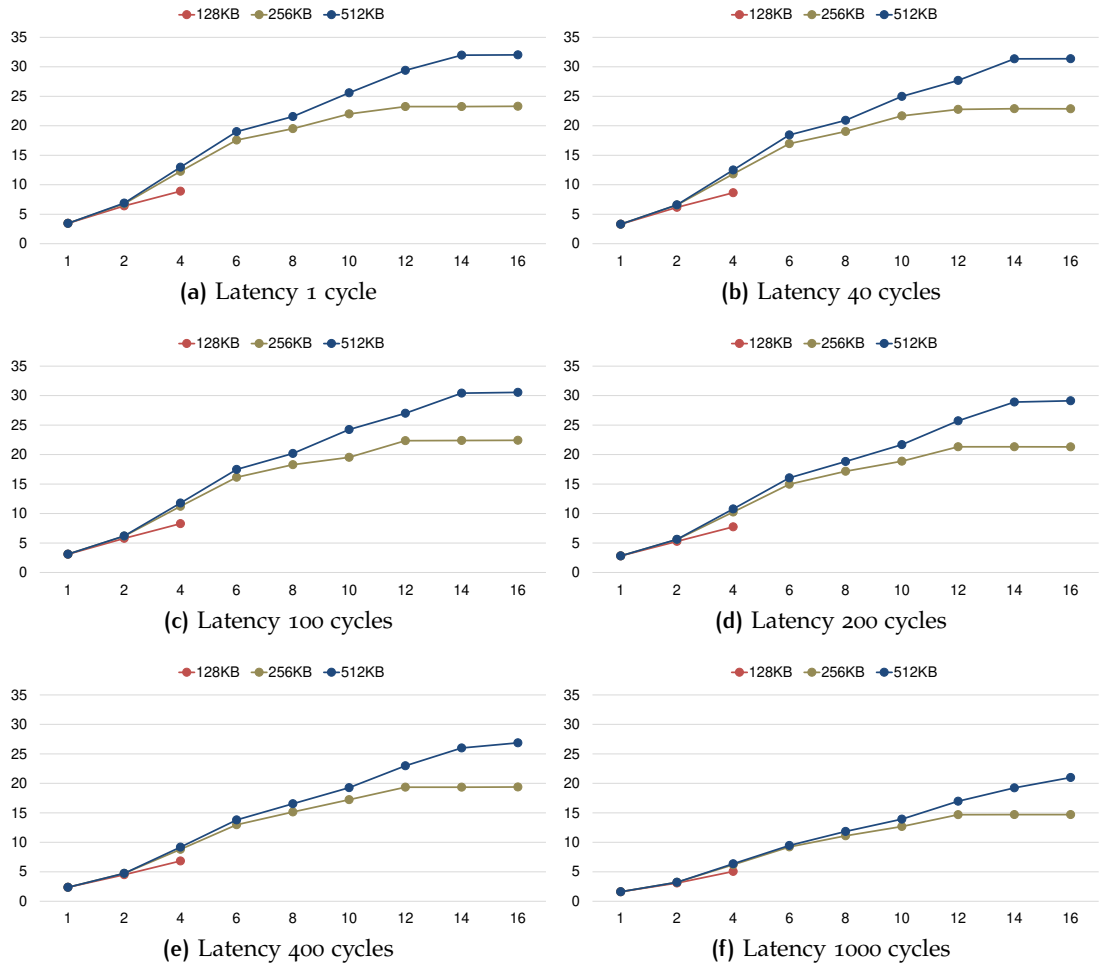
(e) Latency 400 cycles

(f) Latency 1000 cycles

Figure 4.9: Performance gain vs. TCDM memory capacity for different ORB dataflow graph configurations. A minimum of 64KB of the TCDM memory is necessary for smaller configurations. Starting with larger configurations (8PEs and bigger) a minimum of 256KB of TCDM memory is necessary.

not fit with smaller memory sizes, thus there are no points plotted for the ORB with more than 4 PEs and with 64KB TCDM capacity. Similarly, the FD implementation requires the minimum of 128KB of the TCDM. From these Figures, it can be seen that performance gain due to having more memory starts to be noticeable when number of PEs is greater than 4 in both applications. The performance gain from TCDM increase between 256KB and 512KB is more important in the FD application because in FD the long CSV actor's latency can only be hidden by increasing the communication channel buffering capacity.

With the number of PEs greater than 8, it is important to have the TCDM memory capacity of at least 256KB due to minimal applications buffering requirements. As a rule of thumb, the memory requirement of this type of image processing applications is 32KB of the TCDM memory per PE.



(a) Latency 1 cycle

(b) Latency 40 cycles

(c) Latency 100 cycles

(d) Latency 200 cycles

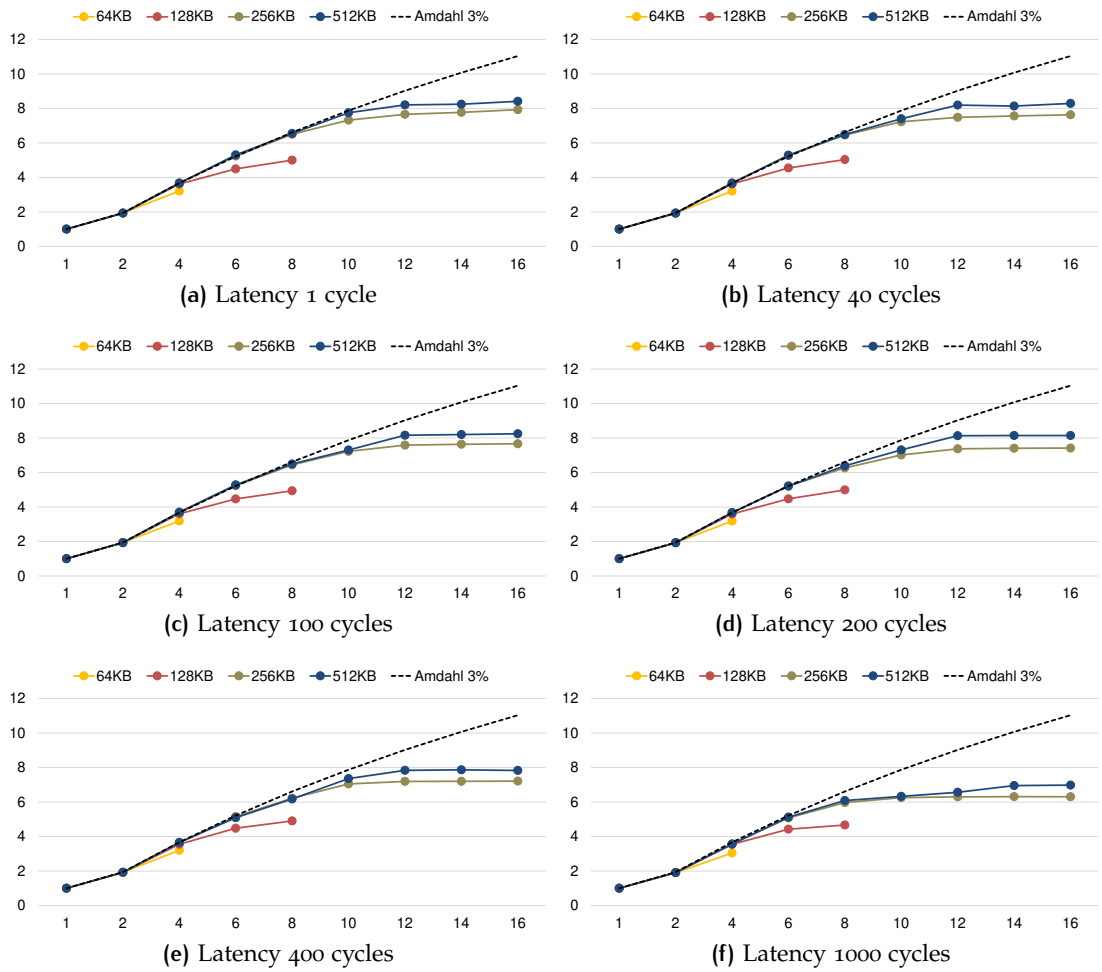
(e) Latency 400 cycles

(f) Latency 1000 cycles

Figure 4.10: Performance gain vs. TCDM memory capacity for different FD dataflow graph configurations. A minimum of 128KB of the TCDM memory is necessary for our implementation. Starting with 8PEs and bigger configurations a minimum of 256KB of TCDM memory is necessary.

4.1.5 Performance Scaling

The performance scaling of a parallel application indicates how much performance increases when more processing elements are added. Performance scaling is important, as the more an application scales the larger number of PEs it can take advantage of. In practice, application nature and overheads limit the scalability of real applications. Amdahl's law models these limits in terms of serial and parallel fractions: *"The performance improvement to be gained by parallelization is limited by the proportion of the code which is serial"* [214]. Generally, the performance scaling also shows how efficient the parallel runtime system is. In order to gain insight into the StreamDrive performance scaling and determine the optimal number of PEs in a CVE cluster, we evaluated the performance of the ORB and the FD applications config-



(a) Latency 1 cycle

(b) Latency 40 cycles

(c) Latency 100 cycles

(d) Latency 200 cycles

(e) Latency 400 cycles

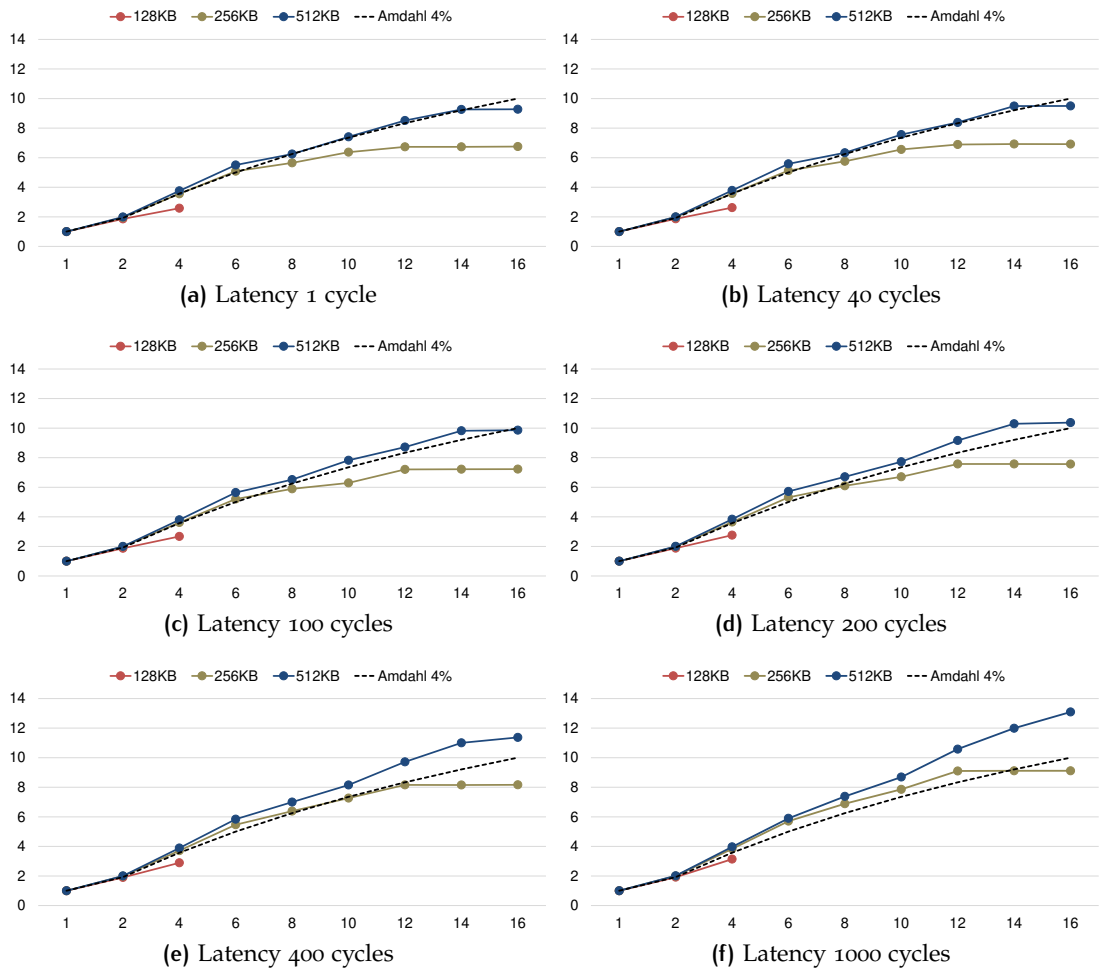
(f) Latency 1000 cycles

Figure 4.11: ORB performance scaling: speed-up vs. number of PEs.

ured for 16 PEs with different TCDM memory sizes, while varying the number of active PEs.

Figures 4.11 and 4.12 plot the resulting Amdahl's curve for the two applications. In each figure, the Y-axis plots the speedup, i.e. the ratio of performance improvement, versus the number of processing elements along the X-axis, for dataflow graph configurations with different TCDM memory sizes. In order to put these numbers into a context, we also plot the "fitting" Amdahl curves corresponding to 3% serial application part for the ORB, and to 4% serial part for the FD. These Amdahl curves fit well the performance of our implementations up to 8 PEs.

Generally, we are observing good speedup up to 8 PEs with a clearly diminishing return when the number of PEs grows further. It is specifically true for the ORB implementation where the speedup does not exceed 8 times independent of the number of PEs or the TCDM memory capacity. Particularly, the speedup with the number of PEs exceeding 8 corresponds to Amdahl curves when increasing



(a) Latency 1 cycle

(b) Latency 40 cycles

(c) Latency 100 cycles

(d) Latency 200 cycles

(e) Latency 400 cycles

(f) Latency 1000 cycles

Figure 4.12: FD performance scaling: speed-up vs. number of PEs.

application serial part. The ORB speedup is limited by about 2% of non-parallelizable processing that corresponds to the Nonmax and Sort functions. Further, the runtime scheduler overhead grows close to 5% as we have shown in Figure 4.7. Altogether, this limits the potential speedup of the ORB application.

The FD speedup is not limited by a non-parallelizable algorithmic part but instead a "non-parallelizable" part is created by an extremely long latency Csv actors that may block the pipeline. The impact of this long latency actor can be reduced by increasing the dataflow buffering capacity, which allows other actors to execute while the Csv is blocking the pipeline. That explains why the FD speedup is higher with longer off-cluster access latencies compared to the low latencies.

For comparison, Yviquel [24] reported performance scaling numbers of their dataflow implementation for an MPEG-4 video decoder. With 10 processors their reported speedup is less than 6 times. The authors explain this relatively low speedup numbers by the limit of

functional parallelism in the application. The StreamDrive is able to exploit data parallelism in addition to the functional parallelism.

As a side note, we observed that if these applications are parallelized using only data-parallelism, eg. with OpenMP parallel section pragmas, the obtained speedup corresponds to an Amdahl curve with at least 10% of serial application part. Although the data-parallelism is often the most obvious to exploit and may require minimal parallelization effort, pipelining implementation is important for achieving high performance.

One important quality of a parallel implementation is how well it performs when off-cluster access latencies are increased. In a System-on-Chip (SoC) environment, where multiple IPs compete for the access to the DDR memory, the off-cluster access latency may grow to tens or hundreds of cycles. Particularly demanding in terms of off-cluster access efficiency are dataflow configurations with many PEs because they require data to be available simultaneously for a large number of actors. Additionally, the relatively small ORB actor granularity results in many modest size DMA transfers, such as one image line of only 640 bytes, or the *Angle* reference image patch of 31x31 bytes.

Figures 4.9 and 4.10 above characterize the impact of increasing the off-cluster access latency from 1 to 1000 processor cycles, in terms of real-time performance, on the ORB and the FD application. Figures 4.11 and 4.12 characterize this impact in terms of performance scaling. Figures 4.9 and 4.10 show that there is almost no performance impact up to off-cluster of 400 processor cycles. Even with unrealistically long latency of 1000 cycles, the performance impact is not very significant. From Figures 4.11 and 4.12, there is no impact on performance scaling up to the very long latency of 1000 cycles. The dynamic dataflow performance holds well even under long off-cluster memory latencies.

4.1.6 KPN vs. DPN Trade-off

Considering that the biggest parallelization effort is required for optimizing the dataflow graph after having introduced the firing rules, it is interesting to characterize the performance achievable with the KPN execution vs. the optimized DPN execution. For this, we compare the KPN and the DPN implementation of the ORB application.

Figure 4.13 shows the comparison of the performance of the two ORB implementations, the DPN and the KPN implementations, for different dataflow graph configurations and as a function of the off-cluster access latency. The Y-axis plots the frame rate (frames/sec.) vs the number of PEs for two TCDM memory capacities, 256KB and 512KB. The DPN implementation frame rate is up to 15% better that the KPN rate with low off-cluster access latency, and up to double the

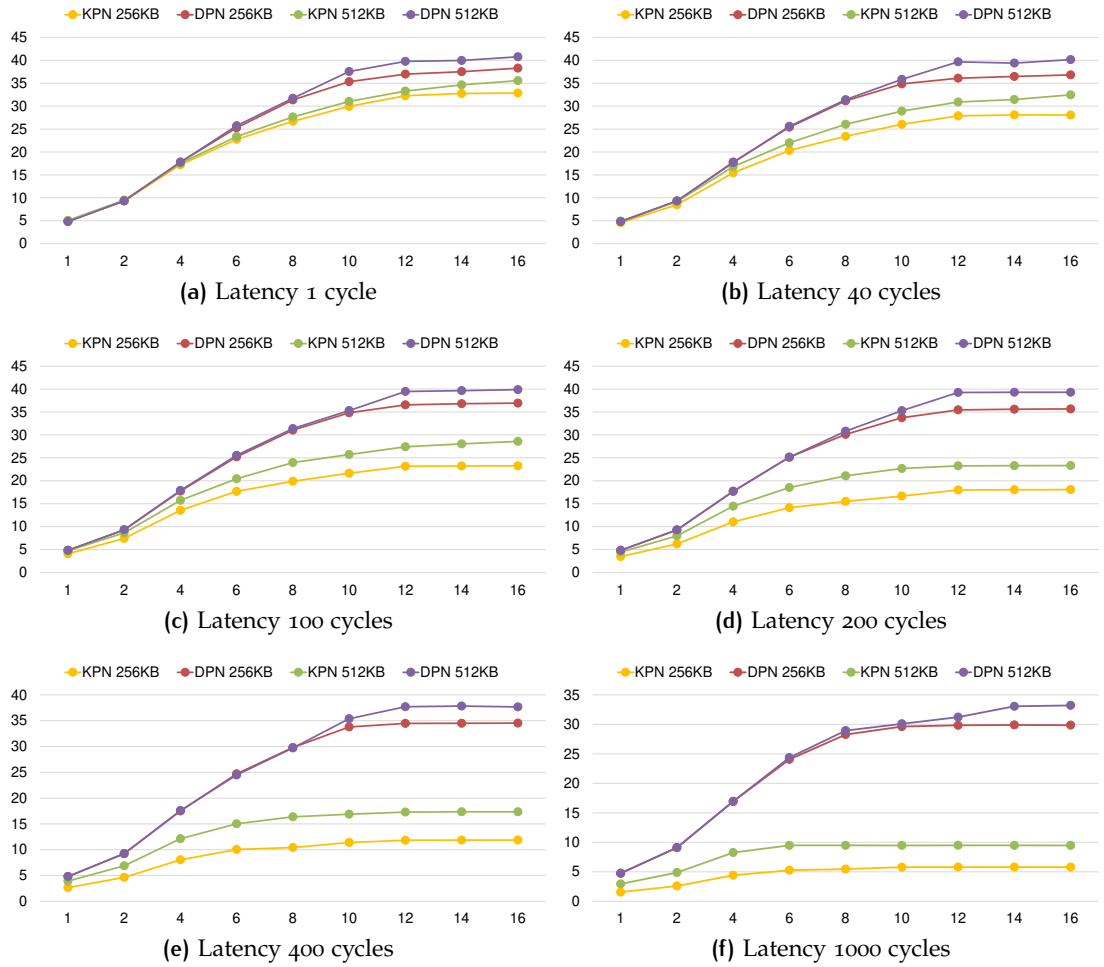


Figure 4.13: DPN vs KPN performance for different ORB dataflow graph configurations.

KPN frame rate even with a reasonable 200 cycles latency. While the DPN implementation achieves a real-time rate of 30 frames/sec. with 8 PEs under all latencies, except unrealistically high 1000 cycles, the KPN implementation does not achieve this rate with latencies greater than 40 cycles and even with such low off-cluster access latencies requires more than 10 PEs. Thus, the effort spent in optimizing the dataflow graph is certainly necessary in order to achieve the target real-time objective.

In both implementations, the frame rate improves with increasing the capacity of communication buffers. However, in the KPN mode the impact of buffering capacity increase is more significant. Unlike the DPN implementation, the KPN implementation is also significantly impacted by the off-cluster memory latency. The cost of a KPN context switch is the reason for these impacts. The number of KPN context switches during the application execution is related to the available buffer sizes: the bigger are dataflow channel buffers, the fewer are there context switches in the KPN mode. Similarly, in the DPN mode,

the number of times that the runtime scheduler switches actors is also related to the dataflow buffers sizes. However, a DPN actor switch is much less penalizing than a KPN context switch. Thus, the performance impact of the number of actor suspensions and resumptions is much more significant in KPN mode. On the other hand, the cost of the KPN context switch is directly related to the off-cluster memory access time because it is not possible to hide the context saving and restoring by performing it in parallel with other computation work. For example, there were 202 context switches during the KPN execution of the first scaling pyramid ORB image in 1 PE. With off-cluster latency of 1 processor cycle, they account for less than 2% of the image processing time. When the off-cluster memory latency increases to 40 cycles, these context switches account for 10% of the image processing time. The KPN implementation performance is much more affected by the off-cluster memory latency than the performance of the DPN implementation. The DPN would be a better choice for real embedded systems, where the off-cluster memory latency is often a bottleneck.

Finally, the KPN implementation performance does not scale as well as the DPN implementation performance when the number of processing elements increases. The reason is twofold: (1) the dynamic DPN assignment of dataflow actors to PEs outperforms the fixed KPN assignment, and (2) the relative contribution of the KPN scheduler is increasing faster than the contribution of the dataflow scheduler with more parallel actors.

4.2 CASE-STUDY 2: CONVOLUTIONAL NEURAL NETWORKS

In this section, we evaluate the internal TCDM memory bandwidth requirements of the HWC hardware block. The HWC bandwidth needs to be supported by sufficient internal HWC buffering and by sufficient number of logarithmic interconnect ports to the TCDM shared memory. We also compare the HWC bandwidth requirements to the state-of-the-art tightly-coupled convolution hardware unit from [29].

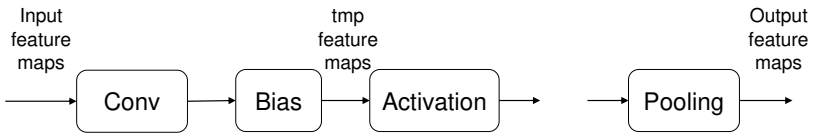


Figure 4.14: Functional blocks from a typical CNN layer.

Figure 4.14 shows a typical CNN layer structure. As explained in chapter 3, the main function blocks are the `Conv` (convolution), the `Activation` and the `Pooling`. We are performing addition of the convolution layer *Bias* outside of the HWC block and therefore have pic-

tured this as a separate function block. There may be any number of additional functions depending on a particular CNN network, such a the *Local Response Normalization* (LRN), or others.

Embedded versions of CNN networks are typically *quantized*, i.e. the floating-point computations are replaced with the fixed-point computations, which are much more efficient in terms of performance, area and power. We are using the *dynamic fixed-point* data quantization technique from [198].

StreamDrive parallelization of the CNN layer over multiple HWC blocks and processing elements is straightforward. The output feature maps volume is split into N sets processed in parallel. The CNN processing is very regular and we can statically assign each parallel output feature map set to a processing element eliminating the dynamic actor assignment overhead. The token granularity of all actors is also straightforward - one feature map line of pixels. Figure 4.15 shows the resulting StreamDrive dataflow graph.

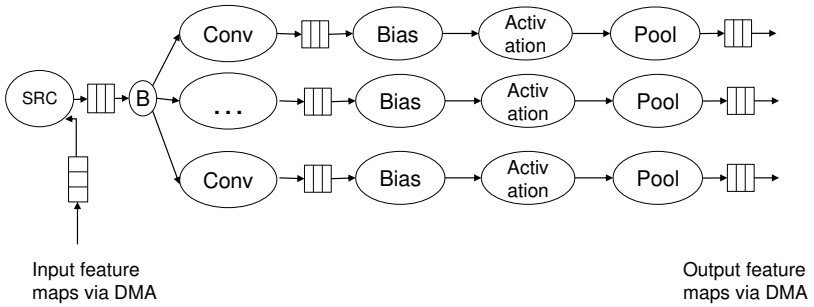


Figure 4.15: StreamDrive dataflow graph for a typical CNN layer.

The input feature maps are read from off-cluster memory via the DMA by a special SRC actor. All parallel output feature map computation sets share the input feature map buffer. The Conv actors use the HWC blocks, while other CNN layer functions are implemented in software. In order to eliminate the runtime scheduler overhead, all software functions are re-grouped in a single dataflow actor. This actor also handles the DMA management for writing the output feature maps back to the off-cluster memory. The StreamDrive runtime ensures the synchronization and dynamic pipelining between the DMA transfers, the HWC blocks, and the software implemented functions.

In order to perform the bandwidth requirements evaluation, we have configured our simulator with a single HWC and run simulations of a large number of convolutional layers coming from following CNNs: the AlexNet [172], the VGG [210], the ResNet [211], the Google Inception [212], and the DenseNet [213]. The complete list of CNN layers used in our evaluation and their shapes can be found in Appendix B. We have then measured the number of bytes transferred between the HWC and the TCDM memory for each tested CNN layer. Figures 4.16 and 4.17 summarize the results.

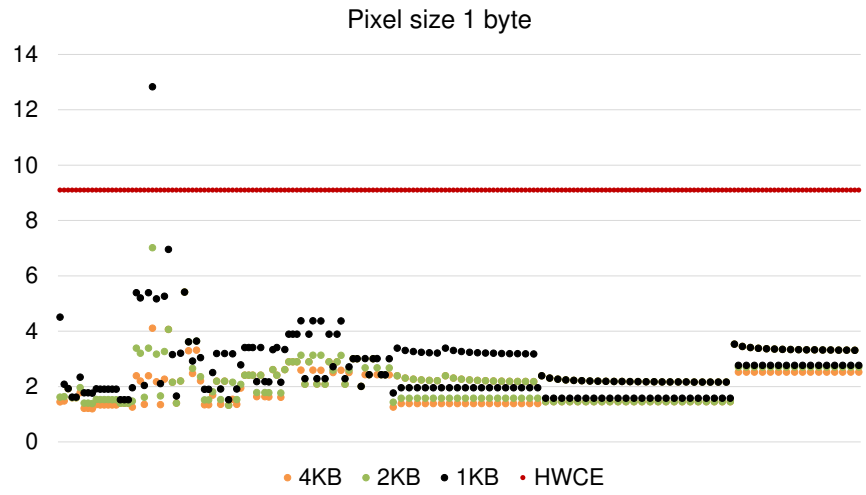


Figure 4.16: Bandwidth (in terms of Bytes/cycle) required by the HWC for a number of different CNN layers: the input feature maps, output feature maps and weights are assumed to be 8-bit values, the accumulation is done in 32-bits. The red line corresponds to the standard 2D convolver bandwidth.

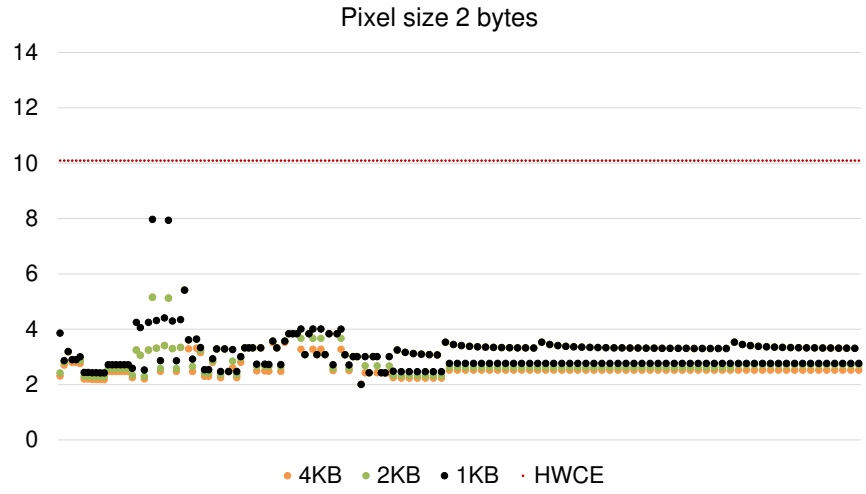


Figure 4.17: Bandwidth (in terms of Bytes/cycle) required by the HWC for a number of different CNN layers: the input feature maps, output feature maps and weights are assumed to be 16-bit values, the accumulation is done in 32-bits. The red line corresponds to the standard 2D convolver bandwidth.

The Figures show the memory bandwidth requirements (in terms of Bytes/cycle) necessary to sustain a 16 MAC wide HWC datapath for the different CNN layers. The Y-axes plot this bandwidth over dozens of CNN layers marked along the X-axes. There are dots of 4 different colors corresponding to different HWC internal buffering capacities, namely 1KB, 2KB, and 4KB of HWC internal storage. The required HWC to TCDM bandwidth is increasing for lower HWC internal buffer capacities. The Figure 4.16 assumes that the feature maps and kernel weights are all 8-bit values while the accumulation is done in a 32-

bit precision; the Figure 4.17 assumes the feature maps and kernel weights of 16-bits with the same accumulation precision.

To put these numbers in a context, we have also plotted the TCDM bandwidth requirements of a state-of-the-art tightly-coupled convolution accelerator, HWCE [29] - the red line. Both hardware units, the HWC and HWCE, target low-cost and low-power applications and implement very little internal buffer storage. Both are constrained by the performance of the shared memory logarithmic interconnect. The HWCE implements a 2D convolver with linear input buffering. The HWCE runs a steady pipeline reading one input pixel per clock cycle and reading and writing a partially accumulated sum every clock cycle [10].

The HWC computation schedule results in lower bandwidth requirements in all cases, but one: with minimal 1KB internal buffering, the decimated 1×1 convolution with the stride 2 in ResNet results in slightly higher bandwidth requirements. Furthermore, the vast majority of CNN layers access less than 6 bytes of data per operating clock cycle. This allows us to reduce the number of logarithmic interconnect ports to 2 ports per HWC element, still allowing an uninterrupted operation even considering occasional conflicts in shared memory access.

4.3 PUTTING IT ALL TOGETHER

We have implemented the HWC hardware block with the CatapultC High-Level Synthesis tool. The design also includes the StreamDrive hardware block bridge described in chapter 2. This HWC version implements slightly over 1KB of internal storage including small input, weight, and partial sum buffers. Table 4.3 shows the breakdown of the local storage capacity for the three CNN arrays:

	I	W	O
Buffer size	96B	128B	1KB

Table 4.3: The HWC buffering size for the three CNN arrays.

In 28 nm technology, the routed HWC together with the hardware bridge occupies the 0,11 mm^2 of the silicon area. For comparison, the HWC with computing capacity of 16 MAC operations per clock cycle, occupies 1,7 times area of our RISC core, capable of only one arithmetic operation per clock cycle. Total area of a shared memory cluster, with 4 PEs (enhanced with the image processing CVAx extension) and 4 HWCs, 256KB of shared TCDM memory, and with 64KB of shared program cache, is $3,0mm^2$, i.e. while accounting for the vast

10 We aligned the HWCE required bandwidth with our precision assumptions; in actual HWCE, the accumulation is performed in 16-bit instead of the 32-bit precision.

majority of cluster performance capacity, the 4 HWCs occupy less than 15% of the total cluster area.

The evaluations conducted in this chapter lead to following observations:

- Given a reasonable off-cluster memory access latency, 8 PEs per CVE cluster allow achieving the frame rate of 30 frames/sec. with the ORB application and 20 frames/sec. with the FD application.
- Given that the HWC required bandwidth can be sustained with 2 logarithmic interconnect ports, 4 HWC per CVE cluster can be integrated for the total of 16 master IPs on the logarithmic interconnect.
- 256KB of the TCDM memory is sufficient for supporting this number of computing elements. However, higher performance implementations should integrate 512KB.

Based on this observations, we propose three CVE implementations: (1) the low-end implementation with 2 PEs, 1 HWC, and 128KB TCDM, (2) the middle range implementation with 4 PEs, 4 HWCs, and 256 KB of TCDM memory, and (3) the performance implementation with 8 PEs, 4 HWCs, and 512 KB of the TCDM memory. The characteristics of each implementation are summarized in Table 4.4:

Config	PEs	HWCs	TCDM,KB	Perf.,GOPs	Cluster Area,mm2
Low-end	2	1	128	12	1,06
Middle	4	4	256	72	2,14
Performance	8	4	512	80	3,54

Table 4.4: Proposed CVE cluster configurations.

The number of GOPs corresponds to the maximal number of 8-bit MAC operations that the cluster is able to performed per second, assuming the 500 MHz operating frequency. The area estimation is a result of the topographical mode synthesis, with a 62% density adjustment applied on top of the synthesis results.

The low-end configuration is able to run ORB at 20 frames/sec. and FD at 5 frames/sec., which is a good performance for the extremely low-cost device. It can also run a small CNN, such as the Traffic Sign Recognition [219]. The middle range configuration runs ORB at 15 frames/sec., FD at 10 frames/sec., and is able to run a CNN similar to AlexNet at 10 frames/sec. as well. Finally, the performance configuration can achieve the real-time execution of 30 frames/sec. for many real life applications.

5 CONCLUSIONS AND FURTHER DIRECTIONS

One of the symptoms of an approaching nervous breakdown is true belief that one's work is terribly important.

— Bertrand Russell

To achieve high performance at low cost and energy consumption, the application-specific hardware acceleration has surfaced as the prevailing approach across embedded application domains. Flexible accelerators, where modifications and enhancements can be done by software changes rather than by re-spins of the hardware present multiple advantages: they are applicable across a variety of applications, allow bug fixes, and support new product requirements over the lifetime of the initial hardware. A general and structured approach for designing flexible accelerators is to build heterogeneous multi-core shared memory clusters. Such clusters combine programmable processor cores and application-specific hardware elements that communicate through a common shared data memory, achieving high performance, cost- and energy-efficiency, while also remaining flexible.

Many modern embedded applications are streaming in nature lending themselves to very efficient hardware implementations through exploitation of parallelism.

To take advantage of the potential performance offered by a shared memory clustered platform, applications need to manage multiple programmable cores, application-specific hardware elements, limited on-chip memory, and explicit memory hierarchy. Parallel programming techniques, however, have developed at a much slower pace compared to the hardware platforms, and most parallel computation models are targeted towards workstation-class machines. Even though industry-backed models, such as OpenMP, have been developed, they have several drawbacks in our context: they lack sufficient abstraction for helping to achieve the right trade-off between various optimization constraints; they do not take advantage of the streaming nature of many embedded applications; they do not manage heterogeneous processing elements; they initially target large-scale systems instead of embedded platforms and incur significant space and time overhead. We have therefore proposed the use of dataflow computation model as one way of addressing the disparity between embedded platform hardware and software developments. The dataflow computation model proposes many properties desirable in the context of embedded application development:

- A simple parallelism model relieves the developers from thinking "in parallel". Dataflow actors can only communicate via the FIFO communication channels; there are no locks, mutexes, critical sections, or race conditions.
- Determinism guarantees that the program will behave identically on each execution. Potential problems can therefore be reliably reproduced, diagnosed and fixed, which is an otherwise notoriously difficult with parallel applications.
- Dataflow actors are written in the usual sequential style. Developers can therefore quickly become productive in the parallel programming domain using their existing knowledge.
- A simple memory model simplifies dealing with limited local memories. There is a clear link between the buffer capacity and the application performance helping the application optimization.
- The asynchronous and pipelined dataflow execution tolerates memory and synchronization latency efficiently.
- The dataflow execution model matches well the streaming nature of many applications. Heterogeneous dataflow actors are seamlessly integrated together; particularly, the application-specific hardware elements are naturally handled.

Decidable dataflow models can be efficiently implemented (even in software only) but lack necessary expressiveness to handle modern applications. The dynamic dataflow models, such as Kahn Process Networks (KPN) and Dataflow Process Networks (DPN) have the necessary expressiveness but existing implementations failed to achieve efficiency required with tightly constrained embedded solutions. Therefore, we have developed StreamDrive, a dynamic dataflow framework consisting of a shared memory cluster architecture with support for efficient dataflow communication and synchronization, the lightweight dataflow programming API, and the efficient dataflow runtime system. During implementation phase, we have focused on simplicity and performance. These properties, together with collection of detailed run-time statistics, distinguish StreamDrive from other existing dynamic dataflow frameworks.

While the dataflow computation model offers a more flexible development framework than other frameworks, it may incur increased complexity on the developer. Being able to reasonably quickly derive a highly optimized implementation from a sequential reference algorithm is "must have" for the adaption of the dataflow computation model by the industry. Addressing this issue we have proposed a successive refinement process for deriving an optimized dataflow implementation from a sequential reference algorithm. The successive

refinement process is based on StreamDrive double support for the two dynamic dataflow models, the KPN and the DPN. The resulting application transformation process consists of a sequence of well-defined and straightforward elementary steps, which significantly reduces the application development costs.

Review of Problem Statement and Contributions

Our initial idea has been to investigate how the dataflow computation model can be used to address the limitations of existing shared memory clustered platforms. In Chapter Introduction, we have therefore raised 4 research questions:

- What architecture support is necessary for efficient execution of dataflow applications in tightly-coupled clusters with streaming hardware elements ?

  To answer this question, we have extended previous shared memory cluster [15] with several hardware elements supporting the dataflow execution model. First, we designed a special hardware block bridge that allows tightly-coupled hardware blocks to communicate via the cluster shared memory. The hardware block bridge takes care of dataflow synchronization letting the hardware elements execute as KPN processes without directly synchronizing with the software scheduler. The hardware block bridge also makes a connection between the streaming *send* and *receive* requests of the hardware element and a *load* and *store* accesses to the cluster shared memory. Second, although dataflow communication via shared memory is efficient, the synchronization based on shared memory polling is not. We have also extended the shared memory cluster with a special *event* network which allows dataflow synchronization via lightweight events rather than memory polling or processor interrupts. Third, we have extended the DMA with event generation ability so that DMA transfers can be seamlessly handled within the general dataflow framework. We have thoroughly analyzed two typical image processing applications, the oriented FAST and Rotated Brief (ORB), and the Face Detection (FD). We have shown that StreamDrive communication and synchronization overhead is very low due to this dedicated hardware support. We have measured that the cost of a single communication operation (a dataflow send or a receive) takes less than 40 processor cycles in our implementation.

- As part of answering the previous question, we have also designed the *Computer Vision Engine* (CVE). The CVE is a specialization of StreamDrive platform targeting computer vision and image processing applications. The CVE development has fo-

cused on two axes: (1) designing a tightly-coupled convolution hardware block, HWC, which can efficiently execute the Convolutional Neural Networks (CNN) workloads, and (2) the CVE configuration in terms of the number of processing elements and the TCDM memory size. The HWC architecture exploration lead us to develop an analytical memory performance model that is more accurate than existing models in applications that manage data scratchpad buffers.

- How do we transform a sequential reference code into an optimized dynamic dataflow implementation and what is the required effort of doing so ?

  To answer this question, we have developed a successive refinement transformation flow. The transformation process starts with a sequential reference algorithm, and proceeds with well-defined elementary steps towards a highly optimized dataflow implementation. Each elementary step is straightforward and allows full functional and performance verification of the application. For example, the very first step transforms the reference code to the KPN form. This step requires minimal changes to the original code and can be done one function at a time. Similarly, going from the KPN form to the DPN requires dividing KPN processes into firings and introducing the *firing rules*. The transformation can be done one function or loop at a time, and one firing rule at a time. The application remains executable throughout all the intermediate steps. We have used the successive refinement process for implementing the ORB and the FD applications. In less than 6 weeks, we have parallelized both applications, optimized the implementation by exploring different parallelization strategies, and fine-tuned each application for executing in 180 target platform configurations.

- How can we make an efficient runtime system for executing dataflow applications on top of a memory limited embedded platform ?

  To answer this question, we have implemented StreamDrive runtime supporting efficient execution of dynamic dataflow applications. StreamDrive implements an innovative zero-copy communication protocol, as well as special *broadcast* and *collect* communication channels that allow efficient sharing of dataflow buffers by dataflow actors. These features allow StreamDrive implementations to have an extremely low memory footprint. Another StreamDrive memory saving feature is the novel stack spilling technique which significantly reduces the application runtime stack requirements. StreamDrive implements distributed low-overhead combined KPN and DPN scheduler able to simultaneously handle both types of dataflow

actors. We have shown that applications under control of this scheduler scale very close to optimal up to 8 processing elements, which is better than other user-space scheduler implementations: Yviquel [24] reported performance scaling less than 6 times with 10 processors on their dataflow implementation for an MPEG-4 video decoder.

- What are the performance characteristics of applications implemented within our dataflow framework?

  To answer it, we have implemented the ORB and the FD applications, as well as a parameterized Convolutional Neural Network (CNN). Our implementation clearly demonstrates the advantages of using the dynamic dataflow framework. The dataflow implementations suffer low overhead, require low memory footprint, scale well with more processing elements, and are almost unaffected by the increase in off-cluster memory access latency.

- As part of answering the previous question, we have also evaluated the bandwidth requirements of different CNN networks versus available HWC internal storage and the cluster shared TCDM memory. With the HWC, with as little as 1KB of internal storage, the CNN convolutional layers require less than 6 bytes of data per clock cycle, dropping to around 2 bytes for most layers with 4KB of HWC internal storage. Such low TCDM bandwidth results from the HWC *computation schedule*, which we derived using our analytical memory performance model. On the other hand, we also used our analytical performance model to compute optimal CNN computation loop-nest tiling in order to minimize off-cluster bandwidth. We show that for most well-known networks the total off-cluster memory bandwidth requirements remain relatively low while achieving real-time execution frame rates.

5.1 A CRITICAL VIEW

A perfect work is rarely found, and it is next-to-impossible to achieve perfection in the limited time available for fulfilling a PhD degree. We have identified certain limitations of our work, which addressing is left for future work.

We have run our experiments using mostly a cycle-approximate simulator. This, however, was not our intention, but a consequence of circumstances – we did not have sufficient time to develop a full FPGA prototype that could be used instead. We have, however, implemented an initial FPGA prototype of our framework with 8 processing elements and a single HWC hardware block.

We consider that using a simulator instead of real hardware would suffice for a proof-of-concept. One particular point that is raising question is that the TCDM memory accesses are only modeled approximately and it is uncertain how well it accounts for actual shared memory conflicts. Further validation of the point using real hardware prototype is left for future work.

Many of our conclusions on dataflow computation model would be stronger if we could perform a quantitative comparison against the OpenMP implementation, for example. We have some comparison points but it is never really clear whether a given OpenMP implementation can be considered efficient enough to serve as a point of reference for this computation model.

Finally, one of the most important design constraints for embedded systems is their power consumption. Although our initial intent was to investigate the power-related aspects of the dataflow computation model, we discovered that it is difficult to report a meaningful power measures in absence of an actual hardware implementation. We have therefore reported results in terms of the number of internal and off-cluster memory accesses, which we consider proportional to the IP power requirements.

5.2 FUTURE DIRECTIONS

We judge the overall results presented in this thesis as positive, and we are considering a full hardware implementation of our Computer Vision Engine.

A real implementation opens at least two new challenges: the assumptions on how well the logarithmic interconnect can handle shared memory conflicts under very busy conditions will need to be validated; the assumptions on how well the logarithmic interconnect can handle the DMA off-cluster memory traffic will need to be validated. The real hardware implementation will also allow to put forward solid cost and power numbers for our computer vision platform.

The StreamDrive also needs to evolve. The aspects of StreamDrive that need to be further investigated are primarily related to automating the optimization of the dataflow graph, and improvements to the runtime scheduler. Many elementary steps during the successive refinement transformation or during the parallelization strategy exploration can be automated. For example, dividing a KPN process loop into firings (iterations) could be helped by an appropriate code rewriting tool. As another example, actor merging is a fairly straightforward but verbose task that might be facilitated by an appropriate tool. On runtime scheduler improvement side, we would focus on providing hints from compile time, such as actor priorities,

for example, to the runtime scheduler in order to further reduce the StreamDrive runtime overhead.

Another future development direction is related to the CNN networks. CNNs are typically developed using open source tools such as Caffe, TensorFlow, or others. Our work with these tools revealed that it is fairly straightforward to integrate our analytical memory performance model with them. Thus, the future work in this area includes the automatic generation of CNN computation schedules from the standard CNN descriptions.

A STREAMDRIVE API REFERENCE

```
================================================================================
                              Actor Declaration
================================================================================

STREAM_DECLARE_ACTOR_TYPE (a,type)

  Defines an alias <a>_actor_t to the <type>. The new type is used refering to
  actors' <a> instances.

STREAM_DECLARE_SW_ACTOR (a,n,s)

  Declares the software actor <a> with <n> ports and with runtime stack of <s>
  bytes. <n> and <s> are constants.

STREAM_DECLARE_HW_BLOCK (a,n)

  Declares the hardware actor <a> with <n> ports. <n> is a constant.

================================================================================
                         Graph Construction API
================================================================================

//
// Obfuscated type stream_bind_t is defined by the system
//

void STREAM_GRAPH_BEGIN ()

  Called from graph construction function. Start a dataflow graph construction.

void STREAM_GRAPH_END ()

  Called from graph construction function. Ends a dataflow graph construction.

<a>_actor_t * STREAM_ACTOR_MAKE (a, const char * n, void * p)
<a>_actor_t * STREAM_ACTOR_MAKE (a, uint8_t n, void * p)

  Called from graph construction function. Allocates an instance of actor <a>
  with name <n> for software actors, and with HW block id <n> for hardware
  actors. Calls actor CONSTRUCTOR. The argument <p> is passed to actor
  constructor.

void STREAM_ACTOR_TERM (<a>_actor_t)

  Called from graph termination function. Removes an instance of actor <a>. Calls
  actor DESTRUCTOR.

stream_bind_t STREAM_MAKE_BUFFER (uint32_t n, uint8_t p)

  Called from graph construction function. Allocates a dataflow buffer of size
  <n> bytes in memory hierarchy level <p>. Returns a handle to a buffer instance.

void STREAM_TERM_BUFFER (stream_bind_t p)

  Called from graph termination function. Deletes the buffer instance with handle
  <p>. Releases the dataflow buffer storage.
```

```
stream_bind_t STREAM_MAKE_BROADCAST (uint8_t out, uint32_t n, uint8_t p)

  Called from graph construction function. Creates a BROADCAST connection with
  fanout degree <out>. A dataflow buffer of size <n> bytes is also allocated in
  memory hierarchy level <p>. Returns a handle to a BROADCAST instance.

void STREAM_TERM_BROADCAST (stream_bind_t p)

  Called from graph termination function. Deletes the BROADCAST instance with
  handle <p>. Releases the dataflow buffer storage.

stream_bind_t STREAM_MAKE_COLLECT (uint8_t in, uint32_t n, uint8_t p)

  Called from graph construction function. Creates a COLLECT connection with
  fanin degree <in>. A dataflow buffer of size <n> bytes is also allocated in
  memory hierarchy level <p>. Returns a handle to a COLLECT instance.

void STREAM_TERM_COLLECT (stream_bind_t p)

  Called from graph termination function. Deletes the COLLECT instance with
  handle <p>. Releases the dataflow buffer storage.

stream_bind_t STREAM_MAKE_COLLECT_BROADCAST (uint8_t in, uint8_t out, uint32_t n,
     uint8_t p)

  Called from graph construction function. Creates a combined COLLECT-BROADCAST
  connection with fanin degree <in> and fanout degree <out>. A dataflow buffer
  of size <n> bytes is also allocated in memory hierarchy level <p>. Returns a
  handle to a COLLECT-BROADCAST instance.

void STREAM_TERM_COLLECT_BROADCAST (stream_bind_t p)

  Called from graph termination function. Deletes the COLLECT-BROADCAST instance
  with handle <p>. Releases the dataflow buffer storage.

void STREAM_BIND_OUT_TO_IN (
  <a1>_actor_t * src,
  uint16_t out,
  <a2>_actor_t * dst,
  uint16_t in,
  stream_bind_t b
)

  Called from graph construction function. Connects port <out> of a source actor
  <src> to port <in> of a destination actor <dst> via the dataflow buffer <b>.
  If <b> is NULL, the ports are assumed to be signal ports.

void STREAM_BIND_OUT_TO_BROADCAST (
  <a>_actor_t * src,
  uint16_t out,
  stream_bind_t b
)

  Called from graph construction function. Connects port <out> of a source actor
  <src> to the input port of a BROADCAST <b>.

void STREAM_BIND_BROADCAST_TO_IN (
  stream_bind_t b,
  <a>_actor_t * dst,
  uint16_t in,
  uint8_t i
)

  Called from graph construction function. Connects output port with index <i>
  of a BROADCAST <b> to input port <in> of a destination actor <dst>.
```

```
void STREAM_BIND_COLLECT_TO_IN (
  stream_bind_t c,
  <a>_actor_t * dst,
  uint16_t in
)

  Called from graph construction function. Connects output port of a COLLECT <c>
  to input port <in> of a destination actor <dst>.

void STREAM_BIND_OUT_TO_COLLECT (
  <a>_actor_t * src,
  uint16_t out,
  stream_bind_t c,
  uint8_t i
)

  Called from graph construction function. Connects output port <out> of a source
  actor <src> to the input port with index <i> of a COLLECT <b>.

============================================================================
                          Graph Execution API
============================================================================

void STREAM_GRAPH_SET_TIMEOUT (uint32_t t)

  Called at the time of initialization of the dataflow graph. Sets the execution
  timeout to <t> milliseconds. If <t> = 0, no timeout is set. Default 0.

void STREAM_GRAPH_SET_PROFILING (uint8_t l)

  Called at the time of initialization of the dataflow graph. Sets dataflow graph
  profiling level. The runtime scheduler then will collect various runtime
  statistics. They are dumped to stdout at the end of dataflow graph execution.
  This is intrusive and affects performance. Default 0.

void STREAM_GRAPH_SET_VERBOSITY (uint8_t l)

  Called at the time of initialization of the dataflow graph. Sets dataflow graph
  verbosity level. The runtime scheduler then will output messages on current
  execution state to stdout. This is intrusive and affects performance. Default
  0.

void STREAM_DUMP_PORT (<a>_actor_t * a, uint16_t p)
void STREAM_DUMP_ACTOR (<a>_actor_t * a)
void STREAM_DUMP_GRAPH ()

  Dumping routines. Can be called from the graph or actor functions.

void STREAM_ACTOR_ENABLE (<a>_actor_t * p)

  Called at the time of initialization of the dataflow graph. The actor <p> is
  enabled with all its connections in the following dataflow execution. The
  actor INIT function is called at this time.

void STREAM_ACTOR_SET_PRIORITY (<a>_actor_t * p, uint8_t n)

  Sets software actor scheduling priority to <n>.

void STREAM_ACTOR_SET_CORE (<a>_actor_t * p, uint16_t m)

  Sets software actor scheduling affinity. The actor <p> will be eligeable for
  execution in a processing core only if this core is set in the mask <m>.

void STREAM_ACTOR_ENABLE_VCD_TRACE (<a>_actor_t * p)
```

```
  Tells the scheduler to generate VCD traces of execution for actor <p>. This is
  intrusive and affects the performance.

================================================================================
                            Actor Construction
================================================================================

STREAM_CONSTRUCTOR (void * arg)

  Actor CONSTRUCTOR. Called at actor creation time. Inside the constructor, the
  actor ports need to be created. Any other actor specific allocations and
  initializations can be done. Any instantiation-specific parameters can be
  passed to the CONSTRUCTOR via <arg>. The THIS pointer points at the current
  actor instance.

STREAM_DESTRUCTOR ()

  Actor DESTRUCTOR. Called at actor deletion time. Inside the destructor, the
  actor ports need to be deleted. Any other actor specific allocations and
  initializations can be undone. The THIS pointer points at the current actor
  instance.

STREAM_INIT (void * arg)

  Called when the actor is ENABLED for a given dataflow graph execution. Any
  initializations, specific to this graph execution, can be done here. Any
  specific parameters can be passed to the INIT via <arg>. The THIS pointer
  points at the current actor instance.

STREAM_WORK ()

  Actor WORK function. Called every time that the actor is fired. Can use actor
  communication API and actor scheduler API. The THIS pointer points at the
  current actor instance.

void STREAM_ACTOR_MAKE_PORT_IN (uint16_t p, const char * n, uibt32_t t)

  Called from software actor CONSTRUCTOR. Allocates an input port with id <p>,
  debug name <n>, and token size <t>.

void STREAM_ACTOR_TERM_PORT_IN (uint16_t p)

  Called from software actor DESTRUCTOR. Removes the input port with id <p>.

void STREAM_ACTOR_MAKE_PORT_OUT (uint16_t p, const char * n, uibt32_t t)

  Called from software actor CONSTRUCTOR. Allocates an output port with id <p>,
  debug name <n>, and token size <t>.

void STREAM_ACTOR_TERM_PORT_OUT (uint16_t p)

  Called from software actor DESTRUCTOR. Removes the output port with id <p>.

void STREAM_ACTOR_MAKE_SYNC_IN (uint16_t p, const char * n)

  Called from software actor CONSTRUCTOR. Allocates an input signal port with
  id <p> and debug name <n>. A signal input port only signals the availability
  of "tokens" but does not have associated FIFO buffer storage.

void STREAM_ACTOR_TERM_SYNC_IN (uint16_t p)

  Called from software actor DESTRUCTOR. Removes the input signal port with id
  <p>.

void STREAM_ACTOR_MAKE_SYNC_OUT (uint16_t p, const char * n)
```

```
  Called from software actor CONSTRUCTOR. Allocates an output signal port with
  id <p> and debug name <n>. A signal output port only signals the availability
  of "tokens" but does not have associated FIFO buffer storage.

void STREAM_ACTOR_TERM_SYNC_OUT (uint16_t p)

  Called from software actor DESTRUCTOR. Removes the output signal port with id
  <p>.

void STREAM_ACTOR_MAKE_PORT_DMA (uint16_t p, const char * n, uint8_t c, uint16_t d
     )

  Called from software actor CONSTRUCTOR. Allocates a DMA port with id <p> and
  debug name <n>. A DMA port allows handling DMA records as if it were output
  tokens. To obtain a free record, it needs to be RESERVEd. The record is
  PUSHed when the DMA transfer is launched. The record is RELEASEd when the DMA
  transfer completes. The DMA port exposes when actors are waiting for the DMA
  records to become available to StreamDrive runtime scheduler.

STREAM_ACTOR_TERM_PORT_DMA (uint16_t p)

  Called from software actor DESTRUCTOR. Removes the DMA port with id <p>.

============================================================================
                          Actor Communication API
============================================================================

void * STREAM_IN_POP (uint16_t p, uint16_t n)

  Called from actor's WORK function. Pops <n> tokens from the input port with id
  <p>. This function blocks actor execution until the <n> tokens are available.
  It returns a pointer to the first available token.

void STREAM_IN_RELEASE (uint16_t p, uint16_t n)

  Called from actor's WORK function. Releases <n> tokens from the input port
  with id <p>.

void * STREAM_OUT_RESERVE (uint16_t p, uint16_t n)

  Called from actor's WORK function. Reserves <n> tokens in the output port with
  id <p>. This function blocks actor execution until enough room is available in
  the output channel for the <n> tokens. It returns a pointer to the first
  available token.

void STREAM_OUT_PUSH (uint16_t p, uint16_t n)

  Called from actor's WORK function. Pushes <n> tokens to the output port with
  id <p>.

============================================================================
                            Actor DMA API
============================================================================

//
// Obfuscated type stream_dma_node_t is defined by the system
//

stream_dma_node_t * STREAM_DMA_RESERVE (uint16_t p)

  Called from actor's WORK function. Reserves one DMA record in the DMA port
  with id <p>. This function blocks actor execution until the DMA record is
  available. It returns a pointer to the DMA record that can be then used for
  configuring a new DMA transaction.

void STREAM_OUT_PUSH_VIA_DMA (uint16_t p, stream_dma_node_t * r, uint16_t n)
```

```
  Called from actor's WORK function. Launches a DMA transfer configured with
  the DMA record <r>. When the DMA transfer is completed, the result is same as
  having PUSHed <n> tokens to the output port with id <p>. The DMA transfer
  should have been configured with destination address corresponding to the
  output channel assoiated with the port <p>.

void STREAM_OUT_SYNC_VIA_DMA (uint16_t p, stream_dma_node_t * r, uint16_t n)

  Called from actor's WORK function. Launches a DMA transfer configured with
  the DMA record <r>. When the DMA transfer is completed, a signal is send to
  the output signal port with id <p> with <n> tokens. The input signal port
  connected to <p> can be used to synchronize with the DMA transfer completion.

============================================================================
                          Actor Execution API
============================================================================

const char * STREAM_ACTOR_NAME ()

  Called from any of a software actor CONSTRUCTOR, DESTRUCTOR, INIT, or WORK
  function. Returns actor debug name.

uint8_t STREAM_GET_HWB_ID ()

  Called from any of a hardware actor CONSTRUCTOR, DESTRUCTOR, or INIT, function.
  Returns hardware actor ID.

void STREAM_PORT_SET_QUOTA (uint16_t p, uint16_t n)

  Called from a software actor CONSTRUCTOR, INIT, or WORK function. Sets the
       firing rule for actor's port <p> to <n> tokens.


void STREAM_PORT_SET_SKIP (uint16_t p, uint16_t n)

  Called from a software actor CONSTRUCTOR, INIT, or WORK function. The next
  firing of this actor will happen after <n> tokens have been POPed and RELEASEd
  from input port <p>. Firing rule is reset to '0' => fires even if there are
  no more tokens after the skip. This is usefull if we want to skip some number
  of tokens at the beginning or at the end of processing without having to fire
  the actor. More tokens than dataflow buffer capacity can be skept.

void STREAM_YIELD ()

  Called from actor WORK function. Yields execution control to the scheduler.

void STREAM_EXIT ()

  Called from actor WORK function. Terminates actor execution - this actor will
  no longer be fired.

void STREAM_ERROR (uint32_t x)

  Called from actor WORK function. Signals the error condition. The dataflow graph
  execution terminates with error code <x>.

void STREAM_TERMINATE (uint32_t x)

  Called from actor WORK function. Terminates dataflow graph execution with exit
  code <x>.
============================================================================
```

B PEEMEN EQUATIONS FOR THE CNN CONVOLUTION LOOP-NEST

The buffering requirements for the 3 array references in the CNN convolution loop-nest are computed as shown in Equation B.1:

$$\begin{aligned} B(I) &= css \times ((yss-1)*S+R) \times ((xss-1)*S+R) \\ B(W) &= mss \times css \times R \times R \\ B(O) &= mss \times yss \times xss \end{aligned} \tag{B.1}$$

The terms $((yss-1)*S+R)$ and $((xss-1)*S+R)$ compute the dimensions, in pixels, of the input feature map tile, given the tile size for the output feature map, $yss \times xss$, the convolution kernel size, $R \times R$, and the convolution stride S.

The total memory traffic is computed by multiplying the buffering requirements by the total number of tiles, with one improvement. Peemen noticed that using application-managed buffers, data can also be reused between consecutive innermost tile executions. This significantly improves the memory traffic estimation accuracy for the computations involving prologue, steady state, and the epilogue. Equation B.2 shows the general form of the memory traffic computation:

$$T = \left\lceil \frac{M}{mss} \right\rceil \left\lceil \frac{C}{css} \right\rceil \left\lceil \frac{E}{yss} \right\rceil \left\lceil \frac{E}{xss} \right\rceil (B(I) + B(W) + 2 * B(O)) \tag{B.2}$$

The 4 cases that need to be considered for the CNN loop-nest, corresponding each to one of the controlling loops, **LTSX**, **LTSY**, **LTIF**, and **LTOF**, being the innermost controlling loop, are shown in Equations B.3, B.4, B.5, and B.6 below.

1. Innermost **LTOF**:

$$T^1 = \left\lceil \frac{C}{css} \right\rceil \left\lceil \frac{E}{yss} \right\rceil \left\lceil \frac{E}{xss} \right\rceil (B^1(I) + B^1(W) + B^1(O)) \tag{B.3}$$

   with

$$\begin{aligned} B^1(I) &= css \cdot ((yss-1)*S+R) \cdot ((xss-1)*S+R) \\ B^1(W) &= M \cdot css \cdot R \cdot R \\ B^1(O) &= 2 \cdot M \cdot yss \cdot xss \end{aligned}$$

2. Innermost **LTIF**:

$$T^2 = \left\lceil \frac{M}{mss} \right\rceil \left\lceil \frac{E}{yss} \right\rceil \left\lceil \frac{E}{xss} \right\rceil (B^2(I) + B^2(W) + B^2(O)) \qquad (B.4)$$

with

$$B^2(I) = C \cdot ((yss - 1) * S + R) \cdot ((xss - 1) * S + R)$$
$$B^2(W) = mss \cdot C \cdot R \cdot R$$
$$B^2(O) = mss \cdot yss \cdot xss$$

3. Innermost **LTSY**:

$$T^3 = \left\lceil \frac{M}{mss} \right\rceil \left\lceil \frac{C}{css} \right\rceil \left\lceil \frac{E}{yss} \right\rceil (B^3(I) + B^3(W) + B^3(O)) \qquad (B.5)$$

with

$$B^3(I) = css \cdot H \cdot ((xss - 1) * S + R)$$
$$B^3(W) = mss \cdot css \cdot R \cdot R$$
$$B^3(O) = 2 \cdot mss \cdot E \cdot xss$$

4. Innermost **LTSX**:

$$T^4 = \left\lceil \frac{M}{mss} \right\rceil \left\lceil \frac{C}{css} \right\rceil \left\lceil \frac{E}{xss} \right\rceil (B^4(I) + B^4(W) + B^4(O)) \qquad (B.6)$$

with

$$B^4(I) = css \cdot ((yss - 1) * S + R) \cdot H$$
$$B^4(W) = mss \cdot css \cdot R \cdot R$$
$$B^4(O) = 2 \cdot mss \cdot yss \cdot E$$

In Equations B.3 - B.6, in order to account for the data reuse across consecutive innermost tile executions, the memory traffic computation is done as if the innermost controlling loop were not tiled. For example, with loop **LTOF** being the innermost controlling loop, the tile size of this loop is set to the total loop count, i.e. $mss = M$, inside the general equation B.2.

C CNN LAYERS CONFIGURATION USED IN THIS THESIS

C.1 ALEXNET, ZFNET, VGG

Layer	H × H	E × E	C	M	Conv.
AlexNet 1	224 × 224	55 × 55	3	96	11 × 11, 4
AlexNet 2	55 × 55	27 × 27	96	256	5 × 5, 2
AlexNet 3	27 × 27	13 × 13	256	384	3 × 3, 2
AlexNet 4	13 × 13	13 × 13	384	384	3 × 3, 1
AlexNet 5	13 × 13	13 × 13	384	256	3 × 3, 1
ZFNet 1	224 × 224	112 × 112	3	96	7 × 7, 2
ZFNet 3	13 × 13	13 × 13	256	384	3 × 3, 1
ZFNet 4	13 × 13	13 × 13	384	384	3 × 3, 1
ZFNet 5	13 × 13	13 × 13	384	256	3 × 3, 1
ZFNet 6	6 × 6	6 × 6	256	256	3 × 3, 1

C.2 VGG

A	A-LRN	B	C	D	E
11-layer	11-layer	13-layer	16-layer	16-layer	19-layer
Input 224x224 RGB image					
conv3x3, 64	conv3x3, 64 LRN	conv3x3, 64 conv3x3, 64	conv3x3, 64 conv3x3, 64	conv3x3, 64 conv3x3, 64	conv3x3, 64 conv3x3, 64
maxpool, 3x3, stride 2					
conv3x3, 128	conv3x3, 128	conv3x3, 128 conv3x3, 128	conv3x3, 128 conv3x3, 128	conv3x3, 128 conv3x3, 128	conv3x3, 128 conv3x3, 128
maxpool, 3x3, stride 2					
conv3x3, 256 conv3x3, 256	conv3x3, 256 conv3x3, 256	conv3x3, 256 conv3x3, 256	conv3x3, 256 conv3x3, 256 conv3x3, 256	conv3x3, 256 conv3x3, 256 conv3x3, 256	conv3x3, 256 conv3x3, 256 conv3x3, 256 conv3x3, 256
maxpool, 3x3, stride 2					
conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512 conv3x3, 512 conv3x3, 512
maxpool, 3x3, stride 2					
conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512 conv3x3, 512	conv3x3, 512 conv3x3, 512 conv3x3, 512 conv3x3, 512
maxpool, 3x3, stride 2					
FC-4096					
FC-4096					
FC-1000					
softmax					

C.3 GOOGLE INCEPTION

Layer	E × E	Inception v3
		Input 298x298 RGB image
conv	149x149	3x3, 32, stride 2
conv	147x147	3x3, 32
conv	147x147	3x3, 64
pooling	73x73	3x3, max pool, stride 2
conv	71x71	3x3, 80
conv	35x35	3x3, 192, stride 2
conv	35x35	3x3, 288
Inception A	35x35	[avg pool, 3x3 → 1x1, 64] [1x1, 64] [1x1, 48 → 5x5, 64] [1x1, 64 → 3x3, 96 → 3x3, 96] x3
Reduction A	17x17	[max pool, 3x3, stride 2] [3x3, 384, stride 2] [1x1, 64 → 3x3, 96 → 3x3, 96, stride 2]
Inception B	17x17	[avg pool, 3x3 → 1x1, 192] [1x1, 192] [1x1, 160 → 7x1, 160 → 1x7, 192] [1x1, 160 → 1x7, 160 → 7x1, 160 → 1x7, 160 → 7x1, 192] x5
Reduction B	8x8	[max pool, 3x3, stride 2] [1x1, 192 → 3x3, 320, stride 2] [1x1, 192 → 7x1, 192 → 1x7, 192 → 3x3, 192, stride 2]
Inception C	8x8	[avg pool, 3x3 → 1x1, 192] [1x1, 320] [1x1, 384 → 3x1, 384] [1x1, 384 → 1x3, 384] [1x1, 448 → 3x3, 384 → 3x1, 384] [1x1, 448 → 3x3, 384 → 1x3, 384] x2
	1x1, average pool, linear, softmax	

Layer	E × E	Inception v4
		Input 298x298 RGB image
conv	149x149	3x3, 32, stride 2
conv	147x147	3x3, 32
conv	147x147	3x3, 64
concat	73x73	[max pool 3x3, stride 2 conv 3x3, 96, stride 2]
concat	71x71	[conv 1x1, 64 conv 1x1, 64 conv 3x3, 96 conv 7x1, 64 conv 1x7, 64 conv 3x3, 96]
concat	35x35	[3x3, max pool, stride 2 conv 3x3, 192]
Inception A	35x35	[avg pool, 3x3 1x1, 96 1x1, 64 1x1, 64 1x1, 96 3x3, 96 3x3, 96] x5 3x3, 96
Reduction A	17x17	[max pool, 3x3, stride 2 3x3, 384, stride 2 1x1, 192 3x3, 224 3x3, 256, stride 2]
Inception B	17x17	[avg pool, 3x3 1x1, 384 1x1, 192 1x1, 192 1x1, 128 1x7, 224 1x7, 192 7x1, 256 7x1, 224 x10 1x7, 224 7x1, 256]
Reduction B	8x8	[max pool, 3x3, stride 2 1x1, 192 1x1, 256 3x3, 192, stride 2 1x7, 256 7x1, 320 3x3, 320, stride 2]
Inception C	8x8	[avg pool, 3x3 1x1, 256 1x1, 384 1x1, 384 1x1, 256 3x1, 256 1x3, 256 1x3, 448 3x1, 512 3x1, 256 1x3, 256] x5
		1x1, average pool, dropout, softmax

C.4 RESNET

Layer	E × E	18-layer	34-layer	50-layer	101-layer
		Input 224x224 RGB image			
conv1	112x112	7x7, 64, stride 2			
conv2_x	56x56	3x3 max pool, stride 2			
		[3x3, 64 3x3, 64] x2	[3x3, 64 3x3, 64] x3	[1x1, 64 3x3, 64 1x1, 256] x3	[1x1, 64 3x3, 64 1x1, 256] x3
conv3_x	28x28	[3x3, 128 3x3, 128] x2	[3x3, 128 3x3, 128] x4	[1x1, 128 3x3, 128 1x1, 512] x4	[1x1, 128 3x3, 128 1x1, 512] x4
conv4_x	14x14	[3x3, 256 3x3, 256] x2	[3x3, 256 3x3, 256] x6	[1x1, 256 3x3, 256 1x1, 1024] x6	[1x1, 256 3x3, 256 1x1, 1024] x23
conv5_x	7x7	[3x3, 512 3x3, 512] x2	[3x3, 512 3x3, 512] x3	[1x1, 512 3x3, 512 1x1, 2048] x3	[1x1, 512 3x3, 512 1x1, 2048] x3
	1x1	average pool, 1000-d fc, softmax			

Downsampling is performed by conv3_1, conv4_1, conv5_1 with stride 2.

C.5 DENSENET

Layer	E × E	DenseNet-121 (k=32)	DenseNet-169 (k=32)	DenseNet-201 (k=32)
Input 224x224 RGB image				
conv	112x112	7x7, 64, stride 2		
pooling	56x56	3x3 max pool, stride 2		
DenseBlock 1	56x56	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x6	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x6	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x6
Transition 1	56x56	1x1		
	28x28	2x2 average pool, stride 2		
DenseBlock 2	28x28	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x12	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x12	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x12
Transition 2	28x28	1x1		
	14x14	2x2 average pool, stride 2		
DenseBlock 3	14x14	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x24	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x32	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x48
Transition 3	14x14	1x1		
	7x7	2x2 average pool, stride 2		
DenseBlock 4	7x7	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x16	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x32	$\begin{bmatrix}1x1\\3x3\end{bmatrix}$ x32
Classification	1x1	7x7 global average pool		
		1000-d fc, softmax		

ABBREVIATIONS

ADF	Affine Dataflow	6
API	Application Programming Interface	xxi
BDF	Boolean Dataflow	7
BPDF	Boolean Parametric Dataflow	8
CFDF	Core Functional Dataflow	8
CNN	Convolutional Neural Networks	xxiii
CSDF	Cyclo Static Dataflow	6
CVE	Computer Vision Engine	xxiii
DIF	Dataflow Interchange Format	13
DMA	Direct Memory Access Unit	xxi
DNN	Deep Neural Networks	83
DPN	Dataflow Process Network	xx
EIDF	Enable-Invoke Dataflow	8
FD	Face Detection	xxiii
HBB	Hardware Block Bridge	xxii
HDF	Heterochronous Dataflow	7
HWC	Hardware Convolution Block	xxiii
HWPE	Hardware Processing Elements	28
IDF	Integer Controlled Dataflow	7
KPN	Kahn Process Network	xxii
ORB	Oriented FAST and Rotated Brief	xxiii
PE	Processing Elements	28
PiSDF	Parameterized and Interfaced Synchronous Dataflow	8
PSDF	Parameterized Synchronous Dataflow	7
SADF	Scenario-Aware Dataflow	7
SBF	Stream Based Function	8
SDF	Synchronous Dataflow	6
SIMD	Single Instruction Multiple Data	101
SPDF	Schedulable Parametric Dataflow	7
SRAM	Static Random Access Memory	xviii
TCDM	Tightly-Coupled Data Memory	28
TDIF	Targeted Dataflow Interchange Format	13

BIBLIOGRAPHY

[1] Computer vision hardware, software, and services market to reach $26.2 billion by 2025. [Online]. Available: https://www.tractica.com/newsroom/press-releases/computer-vision-hardware-software-and-services-market-to-reach-26-2-billion-by-2025

[2] H. Yue, Z. Wang, and K. Dai, "A heterogeneous embedded mpsoc for multimedia applications." in *HPCC*, ser. Lecture Notes in Computer Science, M. Gerndt and D. Kranzlmüller, Eds., vol. 4208. Springer, 2006, pp. 591–600.

[3] P. Lieverse, E. F. Deprettere, B. Kienhuis, and E. A. de Kock, "A clustering approach to explore grain-sizes in the definition of processing elements in dataflow architectures." *VLSI Signal Processing*, vol. 22, no. 1, pp. 9–20, 1999.

[4] P. Burgio, A. Marongiu, D. Heller, C. Chavet, P. Coussy, and L. Benini, "Openmp-based synergistic parallelization and hw acceleration for on-chip shared-memory clusters." in *DSD*, 2012, pp. 751–758.

[5] P. Burgio, A. Marongiu, R. Danilo, P. Coussy, and L. Benini, "Architecture and programming model support for efficient heterogeneous computing on tigthly-coupled shared-memory clusters." in *DASIP*, 2013, pp. 22–29.

[6] F. Conti, A. Marongiu, and L. Benini, "Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters." in *CODES+ISSS*, 2013, pp. 1–10.

[7] M. Dehyadegari, A. Marongiu, M. Kakoee, L. Benini, S. Mohammadi, and N. Yazdani, "A tightly-coupled multi-core cluster with shared memory hw accelerators," in *ISCAMOS*, 2012, pp. 96–103.

[8] C. Fajardo, Z. Fang, R. Iyer, G. Garcia, S. Lee, and L. Zhao, "Buffer-integrated-cache: a cost-effective sram architecture for handheld and embedded platforms." in *DAC*, 2011, pp. 966–971.

[9] NVIDIA, "Next generation cuda compute architecture: Fermi - white paper," http://www.nvidia.com, NVIDIA, 2010.

[10] Plurality, "Plurality hypercore," http://www.plurality.com, Plurality Ltd., 2011.

[11] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. Taylor, "Conservation cores: reducing the energy of mature computations." in *ASPLOS*, 2010, pp. 205–218.

[12] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. B. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future." *IEEE Micro*, vol. 31, no. 2, pp. 86–95, 2011.

[13] P. Wang, J. Collins, G. Ghinya, H. Jiang, X. Tian, M. Girkar, N. Yang, G.-Y. Lueh, and H. Wang, "Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system," in *PLDI*, 2007, pp. 156–166.

[14] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich SMPs," in *DAC*, 2012, pp. 843–849.

[15] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications." in *DAC*, 2012, pp. 1137–1142.

[16] F. Conti, A. Marongiu, C. Pilkington, and L. Benini, "He-p2012: Performance and energy exploration of architecturally heterogeneous many-cores." *Signal Processing Systems*, vol. 85, no. 3, pp. 325–340, 2016.

[17] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing." in *Parallel Computing in Science and Engineering*, ser. Lecture Notes in Computer Science, R. Dierstein, D. Müller-Wichards, and H.-M. Wacker, Eds., vol. 295. Springer, 1987, pp. 61–88.

[18] J. Landwehr, J. Suetterlein, A. Marquez, J. B. Manzano, and G. R. Gao, "Application characterization at scale: lessons learned from developing a distributed open community runtime system for high performance computing." in *Conf. Computing Frontiers*, G. Palermo and J. Feo, Eds. ACM, 2016, pp. 164–171.

[19] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space." in *PGAS*, A. D. Malony and J. R. Hammond, Eds. ACM, 2014, pp. 6:1–6:11.

[20] S. Bhattacharyya, S. Sriram, and E. Lee, "Optimizing synchronization in multiprocessor implementations of iterative dataflow programs," University of California Berkeley, Technical Report UCB/ERL M95/2, January 1995.

[21] O. P. Gangwal, A. Nieuwland, and P. E. R. Lippens, "A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems." in *ISSS*, R. Hermida and E. M. Aboulhamid, Eds. ACM / IEEE Computer Society, 2001, pp. 1–6.

[22] K. Huang, D. Grunert, and L. Thiele, "Windowed fifos for fpga-based multiprocessor systems," in *ASAP*, 2007, pp. 36–41.

[23] W. Haid, "Design and performance analysis of multiprocessor streaming applications," Ph.D. dissertation, ETH, Zurich, 2010.

[24] H. Yviquel, A. Sanchez, P. Jääskeläinen, J. Takala, M. Raulet, and E. Casseau, "Embedded multi-core systems dedicated to dynamic dataflow programs." *Signal Processing Systems*, vol. 80, no. 1, pp. 121–136, 2015.

[25] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *ICCV*, 2011, pp. 2564–2571.

[26] P. Viola and M. Jones, "Robust real-time face detection," *Int. Journal on Computer Vision*, vol. 57, no. 2, pp. 137–154, May 2004.

[27] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks." in *ICCD*. IEEE Computer Society, 2013, pp. 13–19.

[28] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks." in *CVPR Workshops*. IEEE Computer Society, 2014, pp. 696–701.

[29] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters." in *DATE*, W. Nebel and D. Atienza, Eds. ACM, 2015, pp. 683–688.

[30] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "Diannao family: energy-efficient hardware accelerators for machine learning." *Commun. ACM*, vol. 59, no. 11, pp. 105–112, 2016.

[31] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 a 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems." in *ISSCC*. IEEE, 2016, pp. 264–265.

[32] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." *J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[33] "Intel nervana neural network processor: Architecture update," https://ai.intel.com/intel-nervana-neural-network-processor-architecture-update, Intel, 2018.

[34] "An in-depth look at google's first tensor processing unit (tpu)," https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, Google, 2018.

[35] "Powervr series2nx neural network accelerators (nna)," https://www.imgtec.com/powervr/vision/series2nx, Imagination Technologies, 2018.

[36] G. Desoli, V. Tomaselli, E. Plebani, G. Urlini, D. Pau, V. D'Alto, T. Majo, F. D. Ambroggi, T. Boesch, S. pal Singh, E. Guidetti, and N. Chawla, "The orlando project: A 28 nm fd-soi low memory embedded neural network asic." in *ACIVS*, ser. Lecture Notes in Computer Science, J. Blanc-Talon, C. Distante, W. Philips, D. C. Popescu, and P. Scheunders, Eds., vol. 10016, 2016, pp. 217–227.

[37] A. Peemen, B. Mesman, and H. Corporaal, "Optimal iteration scheduling for intra- and inter-tile reuse in nested loop accelerators," Eindhoven University of Technology, Tech. Rep. EC Reports; ESR-2013-3, January 2013.

[38] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks." in *FPGA*, G. A. Constantinides and D. Chen, Eds. ACM, 2015, pp. 161–170.

[39] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A systematic approach to blocking convolutional neural networks." *CoRR*, vol. abs/1606.04209, 2016.

[40] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm." in *PLDI*, D. S. Wise, Ed. ACM, 1991, pp. 30–44.

[41] "Openmp application program interface - version 4.5," http://www.openmp.org/mp-documents/openmp-4.5.pdf, OpenMP Architecture Review Board, November 2015.

[42] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model." *Parallel Computing*, vol. 25, no. 13-14, pp. 1907–1929, 1999.

[43] E. A. Lee, "The problem with threads." *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[44] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP Congress*, August 1974.

[45] J. Dennis, "First version data flow procedure language," MIT Laboratory for Computer Science, Tech. Rep. MAC TM61, 1974.

[46] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2009.

[47] S. Ha and H. Oh, "Decidable dataflow models for signal processing: Synchronous dataflow and its extensions." in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Springer, 2013, pp. 1083–1109.

[48] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[49] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for dsp," vol. 47, no. 9, September 2000, p. 849–875.

[50] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *ICASSP*, vol. 5, 1995, pp. 3255–3258.

[51] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparison of synchronous and cycle-static dataflow," in *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, Oct 1995, pp. 204–210.

[52] A. Bouakaz, J.-P. Talpin, and J. Vitek, "Affine data-flow graphs for the synthesis of hard real-time applications." in *ACSD*, J. Brandt and K. Heljanko, Eds. IEEE Computer Society, 2012, pp. 183–192.

[53] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, no. 6, pp. 742–760, 1999.

[54] S. Stuijk, M. Geilen, B. Thelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *International Conference on Embedded Computer Systems*, 2011, pp. 404–411.

[55] B. J.T., "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, Department of Electrical Enginnering and Computer Science, University of California at Berkeley, 1993.

[56] J. T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," in *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*, Oct 1994, pp. 508–513.

[57] S. Neuendorffer and E. A. Lee, "Hierarchical reconfiguration of dataflow models." in *MEMOCODE*. IEEE, 2004, pp. 179–188.

[58] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems." *IEEE Trans. Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

[59] H. Kee, C.-C. Shen, S. S. Bhattacharyya, I. C. Wong, Y. Rao, and J. Kornerup, "Mapping parameterized cyclo-static dataflow graphs onto configurable hardware." *Signal Processing Systems*, vol. 66, no. 3, pp. 285–301, 2012.

[60] P. Fradet, A. Girault, and P. Poplavko, "Spdf: A schedulable parametric data-flow moc." in *DATE*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 769–774.

[61] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, "Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration." in *ICSAMOS*. IEEE, 2013, pp. 41–48.

[62] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "Bpdf: A statically analyzable dataflow model with integer and boolean parameters." in *EMSOFT*. IEEE, 2013, pp. 3:1–3:10.

[63] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional dif for rapid prototyping." in *IEEE International Workshop on Rapid System Prototyping*. IEEE Computer Society, 2008, pp. 17–23.

[64] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications." in *DATE*, L. Benini, G. D. Micheli, B. M. Al-Hashimi, and W. Müller, Eds. IEEE, 2009, pp. 111–116.

[65] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the sbf model of computation." *VLSI Signal Processing*, vol. 34, no. 3, pp. 291–300, 2003.

[66] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer." *Commun. ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[67] Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture." *IEEE Trans. Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[68] T. Yuba, K. Hiraki, T. Shimada, S. Sekiguchi, and K. Nishida, "The sigma-1 dataflow computer." in *FJCC*, S. A. Szygenda, Ed. ACM, 1987, pp. 578–585.

[69] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture." in *25 Years ISCA: Retrospectives and Reprints*, G. S. Sohi, Ed. ACM, 1998, pp. 398–407.

[70] R. Vedder and D. Finn, "The hughes data flow multiprocessor: Architecture for efficient signal and data processing," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 324–332, June 1985.

[71] F. H. Bloch, "The enhanced moduar signal processor," in *Proceedings of the Seventeenth Annual Pittsbursh Conference on Modeling and Simulation*, vol. 16, Part 3, 1986, pp. 829–836.

[72] R. Merritt, "Embedded software stuck at c," https://www.edn.com/design/other/4023198/Embedded-software-stuck-at-C, EDN Network, September 2007.

[73] P. J. Denning and J. B. Dennis, "The resurgence of parallelism." *Commun. ACM*, vol. 53, no. 6, pp. 30–32, 2010.

[74] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[75] B. Khailany, W. J. Dally, U. J. Kapasi, P. R. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams." *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[76] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, 2002, pp. 179–196.

[77] W. Thies, M. Karczmarek, J. Sermulins, R. M. Rabbah, and S. P. Amarasinghe, "Teleport messaging for distributed stream programs." in *PPOPP*, K. Pingali, K. A. Yelick, and A. S. Grimshaw, Eds. ACM, 2005, pp. 224–235.

[78] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures." in *MICRO*, A. Wolfe and M. S. Schlansker, Eds. ACM/IEEE Computer Society, 2000, pp. 159–170.

[79] S. M. S. A. Chiricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. A. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvptm)." in *MICRO*. ACM/IEEE Computer Society, 2003, pp. 141–150.

[80] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. C. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. L. Allmon, R. Rayess, S. Maresh, and J. S. Emer, "Triggered instructions: a control paradigm for spatially-programmed architectures." in *ISCA*, A. Mendelson, Ed. ACM, 2013, pp. 142–153.

[81] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration." in *ISCA*. ACM, 2017, pp. 416–429.

[82] Y.-H. Chen, J. S. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks." in *ISCA*. IEEE Computer Society, 2016, pp. 367–379.

[83] T. Goubier, R. Sirdey, S. Louise, and V. David, "ΣC: A programming model and language for embedded manycores," in *ICA3PP*, 2011, pp. 385–394.

[84] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, April 1994.

[85] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach." *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[86] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains)," University of California, Berkeley, Tech. Rep. UCB/EECS-2008-37, April 2008.

[87] D. Kim and S. Ha, "Static analysis and automatic code synthesis of flexible fsm model." in *ASP-DAC*, T. Tang, Ed. ACM Press, 2005, pp. 161–165.

[88] F. Kecelli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "First version of a dataflow interchange format," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2002-98, November 2002.

[89] C.-J. Hsu and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format." in *SCOPES*, ser. ACM International Conference Proceeding Series, K. M. Kavi and R. Cytron, Eds., vol. 136, 2005, pp. 37–49.

[90] C.-C. Shen, H.-H. Wu, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A design tool for efficient mapping of multimedia applications onto heterogeneous platforms," in *ICME*. IEEE Computer Society, 2011, pp. 1–6.

[91] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, 2008.

[92] A. D. Pimentel, T. Stefanov, H. Nikolov, M. Thompson, S. Polstra, and E. F. Deprettere, "Tool integration and interoperability challenges of a system-level design flow: A case study." in *SAMOS*, ser. Lecture Notes in Computer Science, M. Berekovic, N. J. Dimopoulos, and S. Wong, Eds., vol. 5114. Springer, 2008, pp. 167–176.

[93] S. Verdoolaege, H. Nikolov, and T. Stefanov, "Pn: a tool for improved derivation of process networks," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, pp. 19–19, 2007.

[94] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels." *IEEE Trans. Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[95] H. Nikolov, T. Stefanov, and E. F. Deprettere, "Multi-processor system design with espam." in *CODES+ISSS*, R. A. Bergamaschi and K. Choi, Eds. ACM, 2006, pp. 211–216.

[96] S. Verdoolaege, "Polyhedral process networks." in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Springer, 2013, pp. 1335–1375.

[97] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures." *EURASIP J. Emb. Sys.*, vol. 2007, 2007.

[98] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. M. Molnos, A. B. Nejad, A. Nelson, and S. Sinha, "Virtual execution platforms for mixed-time-criticality systems: the compsoc architecture and design flow." *SIGBED Review*, vol. 10, no. 3, pp. 23–34, 2013.

[99] S. Stuijk, M. Geilen, and T. Basten, "Sdf3: Sdf for free." in *ACSD*. IEEE Computer Society, 2006, pp. 276–278.

[100] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "Uml-based multiprocessor soc design framework," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 281–320, 2006.

[101] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *EDERC*, 2014, pp. 36–40.

[102] J. Heulot, M. Pelcat, K. Desnos, J. Nezan, and S. Aridhi, "Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps," in *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, Sept 2014, pp. 167–171.

[103] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, the University of Tennessee, Knoxville, 2012.

[104] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao, "Tideflow: The time iterated dependency flow execution model," in *Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, 2011, pp. 1–9.

[105] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, p. 53, 2013.

[106] E. A. de Kock, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink, "Yapi: Application modeling for signal processing systems," in *DAC*, 2000, pp. 402–405.

[107] Z. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen, "The nornir run-time system for parallel programs using kahn process networks on multi-core machines - a flexible alternative to mapreduce." *The Journal of Supercomputing*, vol. 63, no. 1, pp. 191–217, 2013.

[108] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *PASCO*, 2007, pp. 15–23.

[109] T. Gautier, F. L. Mentec, V. Faucher, and B. Raffin, "X-kaapi: A multi paradigm runtime for multicore architectures." in *ICPP*. IEEE Computer Society, 2013, pp. 728–735.

[110] V. Boulos, S. Huet, V. Fristot, L. Salvo, and D. Houzet, "Efficient implementation of data flow graphs on multi-gpu clusters." *J. Real-Time Image Processing*, vol. 9, no. 1, pp. 217–232, 2014.

[111] A. S. Sbîrlea, Y. Zou, Z. Budimlic, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms." in *LCTES*, R. Wilhelm, H. Falk, and W. Yi, Eds. ACM, 2012, pp. 61–70.

[112] S. A. Edwards and O. Tardieu, "Shim: A deterministic model for heterogeneous embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 854–867, 2006.

[113] S. A. Edwards, N. Vasudevan, and O. Tardieu, "Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling shim to pthreads." in *DATE*, D. Sciuto, Ed. ACM, 2008, pp. 1498–1503.

[114] N. Vasudevan and S. A. Edwards, "Celling shim: Compiling deterministic concurrency to a heterogeneous multicore," in *ACM Symposium on Applied Computing*, 2009, pp. 1626–1631.

[115] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems." in *ACSD*, T. Basten, G. Juhás, and S. K. Shukla, Eds. IEEE Computer Society, 2007, pp. 29–40.

[116] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos," in *ESTIMedia*, 2009, pp. 35–44.

[117] B. Kienhuis, E. F. Deprettere, K. A. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures." in *ASAP*. IEEE Computer Society, 1997, pp. 338–349.

[118] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation - a performance view." *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[119] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini, "Shapes: : a tiled scalable software hardware architecture platform for embedded systems." in *CODES+ISSS*, R. A. Bergamaschi and K. Choi, Eds. ACM, 2006, pp. 167–172.

[120] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "Mparm: Exploring the multi-processor soc design space with systemc." *VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, 2005.

[121] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. R. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view." in *SC*. IEEE, 2010, pp. 1–11.

[122] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Sensys*, 2006, pp. 29–42.

[123] C. Shen, W. Plishker, and S. S. Bhattacharyya, "Dataflow-based design and implementation of image processing applications," *Multimedia Image and Video Processing*, pp. 609–629, 2012.

[124] J. Eker and J. Janneck, "Caltrop—language report (draft)," Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, Technical Memorandum, 2002, http://www.gigascale.org/caltrop.

[125] J. Eker and J. W. Janneck, "Dataflow programming in cal – balancing expressiveness, analyzability, and implementability," in *Asilomar Conference on Signals, Systems and Computers*, 2012, pp. 1120–1124.

[126] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard [standards in a nutshell]." *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–167, 2010.

[127] S. S. Bhattacharyya, G. J. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "Opendf: a dataflow toolset for reconfigurable hardware and multicore systems." *SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2008.

[128] "Mpeg systems technologies - part 4: Codec configuration representation," ISO/IEC 23001-4:2009 Information technology, MPEG systems technologies, 2009.

[129] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, ser. MM '13. New York, NY, USA: ACM, 2013, pp. 863–866.

[130] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs - an mpeg-4 simple profile decoder case study." *Signal Processing Systems*, vol. 63, no. 2, pp. 241–249, 2011.

[131] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: An mpeg-4 simple profile decoder case study." in *SiPS*. IEEE, 2008, pp. 281–286.

[132] G. Roquier, E. Bezati, and M. Mattavelli, "Hardware and software synthesis of heterogeneous systems from dataflow programs." *J. Electrical and Computer Engineering*, vol. 2012, pp. 484 962:1–484 962:11, 2012.

[133] A. A.-H. B. A. Rahman, "Optimizing dataflow programs for hardware synthesis," Ph.D. dissertation, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2014.

[134] E. Bezati, "High-level synthesis of dataflow programs for heterogeneous platforms: design flow tools and design space exploration," Ph.D. dissertation, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2015.

[135] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck, "High-level system synthesis and optimization of dataflow programs for mpsocs." in *ACSSC*, M. B. Matthews, Ed. IEEE, 2016, pp. 417–421.

[136] C. Sau, P. Meloni, L. Raffo, F. Palumbo, E. Bezati, S. C. Brunet, and M. Mattavelli, "Automated design flow for multi-functional dataflow-based platforms." *Signal Processing Systems*, vol. 85, no. 1, pp. 143–165, 2016.

[137] C. Von Plate, "D2C: CAL ARM compiler," http://www.actors-project.eu, ACTORS Project, 2008-2011.

[138] E. Gebrewahid, M. Yang, G. Cedersjö, Z. U. Abdin, V. Gaspes, J. W. Janneck, and B. Svensson, "Realizing efficient execution of dataflow actors on manycores," in *EUC*, 2014, pp. 321–328.

[139] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with epiphany," in *Asilomar Conference on Signals, Systems and Computers*. IEEE, 2014, pp. 1719–1726.

[140] J. Janneck, *Actor Machines—a machine model for dataflow actors and its applications*. Lund Institute of Technology, Lund University, 2011, vol. 247.

[141] G. Cedersjo and J. W. Janneck, "Actor classification using actor machines." in *ACSSC*, M. B. Matthews, Ed. IEEE, 2013, pp. 1801–1804.

[142] G. Cedersjö and J. W. Janneck, "Software code generation for dynamic dataflow programs," in *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '14. New York, NY, USA: ACM, 2014, pp. 31–39.

[143] Z. Ul-Abdin and M. Yang, "A radar signal processing case study for dataflow programming of manycores," *Journal of Signal Processing Systems*, pp. 1–14, 2015.

[144] J. Janneck, G. Cedersjö, E. Bezati, and S. Casale Brunet, "Dataflow machines," in *[Host publication title missing]*. IEEE–Institute of Electrical and Electronics Engineers Inc., 2014, pp. 1848–1852.

[145] M. Michalska, E. Bezati, S. C. Brunet, and M. Mattavelli, "A partition scheduler model for dynamic dataflow programs." in *ICCS*, ser. Procedia Computer Science, M. Connolly, Ed., vol. 80. Elsevier, 2016, pp. 2287–2291.

[146] M. Michalska, N. Zufferey, J. Boutellier, E. Bezati, and M. Mattavelli, "Efficient scheduling policies for dynamic data flow programs executed on multi-core." in *11th International Meeting on Logistics Research*, 7-9 September 2016.

[147] F. Conti, C. Pilkington, A. Marongiu, and L. Benini, "He-p2012: architectural heterogeneity exploration on a scalable many-core platform," in *CASSAP*, 2014, pp. 114–120.

[148] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[149] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[150] K. Martin, M. Rizk, M. J. Sepulveda Florez, and J.-P. Diguet, "Notifying Memories: a case-study on Data-Flow Applications with NoC Interfaces Implementation," in *Design Automation Conference*, ser. Proceedings of the 53rd Annual Design Automation Conference, Austin, United States, 2016.

[151] S. Fischaber, R. F. Woods, and J. McAllister, "Soc memory hierarchy derivation from dataflow graphs." in *SiPS*. IEEE, 2007, pp. 469–474.

[152] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Memory analysis and optimized allocation of dataflow applications on shared-memory mpsocs - in-depth study of a computer vision application." *Signal Processing Systems*, vol. 80, no. 1, pp. 19–37, 2015.

[153] A. R. Mamidala, D. Faraj, S. Kumar, D. Miller, M. Blocksome, T. Gooding, P. Heidelberger, and G. Dózsa, "Optimizing mpi collectives using efficient intra-node communication techniques over the blue gene/p supercomputer." in *IPDPS Workshops*. IEEE, 2011, pp. 771–780.

[154] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl, "Sprint: A tool to generate concurrent transaction-level models from sequential code." *EURASIP Journal on Applied Signal Processing*, vol. 1, p. 213, January 2007.

[155] E. Lee, "A denotational semantics for dataflow with firing," Electronics Research Laboratory, U. C. Berkeley, Memorandum UCB/ERL M97/3, January 1997.

[156] M. Verma and P. Marwedel, *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[157] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California at Berkeley, 1995.

[158] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks." in *ESOP*, ser. Lecture Notes in Computer Science, P. Degano, Ed., vol. 2618. Springer, 2003, pp. 319–334.

[159] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning." *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[160] M. Schweitzer and H.-J. Wuensche, "Efficient Keypoint Matching for Robot Vision using GPUs," in *12th IEEE Internationall Conference on Computer Vision, 5th IEEE Workshop on Embedded Computer Vision*, October 2009.

[161] H. Xie, K. Gao, Y. Zhang, J. Li, and Y. Liu, "Gpu-based fast scale invariant interest point detector." in *ICASSP*. IEEE, 2010, pp. 2494–2497.

[162] J. Svab, T. Krajnik, J. Faigl, and L. Preucil, "Fpga-based speeded up robust features," in *2009 IEEE International Conference on Technologies for Practical Robot Applications*. Boston: IEEE Computer Society, November 2009, pp. 35–41.

[163] D. Bouris, A. Nikitakis, and I. Papaefstathiou, "Fast and efficient fpga-based feature detection employing the surf algorithm." in *FCCM*, R. Sass and R. Tessier, Eds. IEEE Computer Society, 2010, pp. 3–10.

[164] M. Schaeferling and G. Kiefer, "Object recognition on a chip: A complete surf-based system on a single fpga." in *ReConFig*, P. M. Athanas, J. Becker, and R. Cumplido, Eds. IEEE Computer Society, 2011, pp. 49–54.

[165] G. S. van der Wal, D. C. Zhang, I. Kandaswamy, J. Marakowitz, K. Kaighn, J. Zhang, and S. M. Chai, "Fpga acceleration for feature based processing applications." in *CVPR Workshops*. IEEE Computer Society, 2015, pp. 42–47.

[166] H.-Y. Chang, I. H.-R. Jiang, H. P. Hofstee, D. A. Jamsek, and G.-J. Nam, "Feature detection for image analytics via fpga acceleration." *IBM Journal of Research and Development*, vol. 59, no. 2/3, 2015.

[167] D. Lowe, "Distinctive image features from scale-invariant keypoints," in *International Journal of Computer Vision*, vol. 20, 2003.

[168] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Surf: Speeded up robust features," *Computer Vision and Image Understanding (CVIU)*, vol. 110, pp. 346–359, 2008.

[169] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey." *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[170] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: shifting vision processing closer to the sensor." in *ISCA*, D. T. Marr and D. H. Albonesi, Eds. ACM, 2015, pp. 92–104.

[171] M. Graphics, "Catapult high-level synthesis," https://www.mentor.com/hls-lp/catapult-high-level-synthesis, Menthor Graphics, 2018.

[172] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[173] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks." in *FPL*, M. Danek, J. Kadlec, and B. E. Nelson, Eds. IEEE, 2009, pp. 32–37.

[174] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks." in *ICANN*, ser. Lecture Notes in Computer Science, S. Wermter, C. Weber, W. Duch, T. Honkela, P. D. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, Eds., vol. 8681. Springer, 2014, pp. 281–290.

[175] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms." in *ASPLOS*, D. A. Patterson, Ed. ACM Press, 1991, pp. 63–74, sIGARCH Computer Architecture News 19(2), SIGOPS Operating System Review 25(Special Issue April 1991), and SIGPLAN Notices 26(4).

[176] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision." in *CVPR Workshops*. IEEE Computer Society, 2011, pp. 109–116.

[177] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 257–260.

[178] P. Merolla, J. V. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm." in *CICC*, R. Patel, T. Andre, and A. Khan, Eds. IEEE, 2011, pp. 1–4.

[179] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, "4.6 a1.93tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big-data applications." in *ISSCC*. IEEE, 2015, pp. 1–3.

[180] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz, "Convolution engine: balancing efficiency and flexibility in specialized computing." *Commun. ACM*, vol. 58, no. 4, pp. 85–93, 2015.

[181] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes." *CoRR*, vol. abs/1701.06420, 2017.

[182] S. T. Chakradhar, M. Sankaradass, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks." in *ISCA*, A. Seznec, U. C. Weiser, and R. Ronen, Eds. ACM, 2010, pp. 247–257.

[183] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator." *IEEE Trans. Circuits Syst. Video Techn.*, vol. 27, no. 11, pp. 2461–2475, 2017.

[184] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning." in *ASPLOS*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 269–284.

[185] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2018.

[186] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[187] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing." in *FPGA*, B. L. Hutchings and V. Betz, Eds. ACM, 2013, pp. 29–38.

[188] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, ser. ASAP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 53–60.

[189] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer." in *MICRO*. IEEE, 2014, pp. 609–622.

[190] D. Kim, J. Kung, S. M. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory." in *ISCA*. IEEE Computer Society, 2016, pp. 380–392.

[191] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory." in *ASPLOS*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 751–764.

[192] M. Gautschi, D. Rossi, and L. Benini, "Customizing an open source processor to fit in an ultra-low power cluster with a shared l1 memory." in *ACM Great Lakes Symposium on VLSI*, J. R. Cavallaro, T. Zhang, A. K. Jones, and H. H. Li, Eds. ACM, 2014, pp. 87–88.

[193] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "Energy-efficient vision on the pulp platform for ultra-low power parallel computing." in *SiPS*. IEEE, 2014, pp. 274–279.

[194] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators." in *DATE*. IEEE, 2018, pp. 343–348.

[195] V. Rana, I. Beretta, F. Bruschi, A. A. Nacci, D. Atienza, and D. Sciuto, "Efficient hardware design of iterative stencil loops." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2018–2031, 2016.

[196] J. Cong, P. Li, B. Xiao, and P. Zhang, "An optimal microarchitecture for stencil computation acceleration based on nonuniform partitioning of data reuse buffers." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 407–418, 2016.

[197] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision." *CoRR*, vol. abs/1502.02551, 2015.

[198] P. Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks." *CoRR*, vol. abs/1605.06402, 2016.

[199] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices." in *CVPR*. IEEE Computer Society, 2016, pp. 4820–4828.

[200] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations." *CoRR*, vol. abs/1511.00363, 2015.

[201] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1." *CoRR*, vol. abs/1602.02830, 2016.

[202] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks." *CoRR*, vol. abs/1603.05279, 2016.

[203] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[204] F. Conti, P. D. Schiavone, and L. Benini, "Xnor neural engine: a hardware accelerator ip for 21.6 fj/op binary neural network inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. Early Access, pp. 1–1, 2018.

[205] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and B. Dally, "Deep compression and eie: Efficient inference engine on compressed deep neural network." in *Hot Chips Symposium*. IEEE, 2016, pp. 1–6.

[206] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks." *CoRR*, vol. abs/1710.09282, 2017.

[207] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 806–814.

[208] S. Changpinyo, M. Sandler, and A. Zhmoginov, "The power of sparsity in convolutional neural networks," *CoRR*, vol. abs/1702.06257, 2017.

[209] X. Zhou, Z. Du, S. Zhang, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Addressing sparsity in deep neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. Early Access, pp. 1–1, 2018.

[210] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[211] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[212] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning." *CoRR*, vol. abs/1602.07261, 2016.

[213] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks." *CoRR*, vol. abs/1608.06993, 2016.

[214] G. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," in *AFIPS*, vol. 30, 1967, pp. 483–485.

[215] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision – ECCV 2006*, ser. Lecture Notes in Computer Science, A. Leonardis, H. Bischof, and A. Pinz, Eds. Springer Berlin Heidelberg, 2006, vol. 3951, pp. 430–443.

[216] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *Computer Vision – ECCV 2010*, ser. Lecture Notes in Computer Science, K. Daniilidis, P. Maragos, and N. Paragios, Eds. Springer Berlin Heidelberg, 2010, vol. 6314, pp. 778–792.

[217] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 1, pp. 105–119, 2010.

[218] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the 4th Alvey Vision Conference*, 1988, pp. 147–151.

[219] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks." in *IJCNN*. IEEE, 2011, pp. 2809–2813.