Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Ingegneria Elettronica, delle Comunicazioni e tecnologie dell'informazione

Ciclo XXIX°

Settore Concorsuale di afferenza:
Area 09 (Ingegneria Industriale e dell'Informazione) – 09/E3 Elettronica

Settore Scientifico disciplinare:
Area 09 (Ingegneria Industriale e dell'Informazione) – ING-INF/01 Elettronica

OPTIMIZATION TECHNIQUES FOR PARALLEL PROGRAMMING OF EMBEDDED MANY-CORE COMPUTING PLATFORMS

Presentata da: Dott. Giuseppe Tagliavini

Coordinatore Dottorato

Prof. Alessandro Vanelli Coralli

Relatore

Prof. Luca Benini

Correlatori

Dott. Andrea Marongiu

Esame finale anno 2017

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Dipartimento di Ingegneria

dell'Energia Elettrica e dell'Informazione

Tesi di Dottorato

OPTIMIZATION TECHNIQUES FOR PARALLEL PROGRAMMING OF EMBEDDED MANY-CORE COMPUTING PLATFORMS

Relatore:

Chiar.mo Prof. Luca Benini

Candidato:

Dott. Giuseppe Tagliavini

Correlatore:

Dott. Andrea Marongiu

Anno Accademico 2015-2016

I dedicate all the work and efforts of last years to my grandfather Domenico, who prematurely passed away leaving me without my personal superhero.
Wherever you are now, one day I will be there with you.

Contents

Introduction **3**

I Background 3

I.I The roadmap to many-core architectures 3

I.II Programming models for parallel platforms 9

II Thesis contributions 12

1 Overview of embedded many-core platforms **19**

1 Platform template 19

2 STMicroelectronics STHORM 21

3 Kalray MPPA-256 23

4 PULP 26

2 Enabling fine-grained OpenMP tasking on many-core accelerators **29**

1 Introduction 29

2 Related work 31

3 Design of an embedded tasking runtime 32

3.1 Work queue 33

3.2 Runtime layer 34

4 Task schedulers 38

5 Untied tasks 39
6 Experimental results 44
6.1 Applications with a linear generation pattern . . . 45
6.2 Applications with a recursive generation pattern . . 46
6.3 Applications with mixed patterns 48
6.4 Impact of cutoff on linear and recursive applications 49
6.5 Real applications 50
6.6 Comparison with other tasking models 51
7 Conclusion 53

3 Extending the OpenCL model for efficient execution of graph-based workloads on many-core accelerators 54
1 Introduction 54
2 OpenVX programming model 58
3 Extended OpenCL run-time 62
4 Optimization framework for many core accelerators 65
4.1 Data access patterns 66
4.2 Graph partitioning 71
4.3 Node scheduling 72
4.4 Tile size propagation 74
4.5 Buffer allocation and sizing 75
4.6 Run-time graph 77
4.7 Nested graphs 78
4.8 User-defined nodes 80
5 Experimental results 80
5.1 Comparison with OpenCL 85
5.2 Execution efficiency 88
5.3 Comparison with similar tools 89

6 Related Work 92
7 Conclusion 95

4 An OpenVX environment to optimize embedded vision applications on parametric parallel accelerators 96

1 Introduction 96
2 Alternative tools 98
3 ADRENALINE internals 98
3.1 Virtual platform 99
3.2 OpenVX run-time 103
4 Experimental results 104
4.1 Comparison with a real platform 104
4.2 Application partitioning 106
4.3 Architectural configuration 107
5 Conclusion 109

5 Providing lightweight OpenVX support for resource-constrained mW-scale parallel accelerators 110

1 Introduction 110
2 Background 114
2.1 Platform template 114
2.2 OpenVX advanced features 116
2.3 OpenVX execution model and RTE 119
3 Code Generation-based RTE 122
3.1 Extension for static graph support 124
3.2 *milliVX* framework 133
4 Experimental results 135
4.1 Setup 136

4.2 Memory footprint . . . 138
4.3 Execution time . . . 140
4.4 Energy efficiency . . . 141
4.5 Bandwidth reduction . . . 142
5 Related work . . . 143
6 Conclusion . . . 146

6 A new frontier: supporting approximate computing on mW-scale parallel accelerators 148
1 Introduction . . . 148
2 Related work . . . 152
3 Hardware Architecture . . . 155
3.1 PULP with hybrid L1 memory . . . 156
3.2 TCDM Reliability Extensions . . . 157
4 Software Stack . . . 161
4.1 Programming Model . . . 161
4.2 Runtime Extensions . . . 164
4.3 Compile-time Optimizations . . . 165
5 Experimental Evaluation . . . 169
5.1 Simulation Setup and Methodology . . . 171
5.2 Software Stack and Benchmark Suite . . . 172
5.3 Energy Consumption . . . 173
5.4 SoC Area . . . 179
5.5 Application accuracy . . . 181
5.6 Comparison with other approaches . . . 183
6 Conclusion . . . 185

Conclusion 187

List of Publications **189**

Acknowledgments **193**

List of Figures

1 Trend of processor frequency over time (Data source: http://cpudb.stanford.edu). 5

2 Two cores computing two elements in parallel. 6

3 Thesis organization. 12

1.1 Reference architectural template for many-core computing platforms. 20

1.2 STHORM architecture. 22

1.3 STHORM evaluation board. 23

1.4 MPPA-256 block diagram (Source: Kalray Corporation). . 24

1.5 MPPA-256 compute cluster (Source: Kalray Corporation). 25

1.6 PULP cluster architecture. 27

2.1 Design of tasking support. 33

2.2 Design of task scheduling loop. 35

2.3 Example of tied and untied task scheduling. 38

2.4 *untied* task suspension with *task contexts* and per-task stacks. 40

2.5 Speedup of the LINEAR benchmark (no cutoff). 45

2.6 Speedup of the RECURSIVE benchmark (no cutoff). . . . 46

2.7 Structure of the MIXED benchmark. 48

2.8 Speedup of the MIXED benchmark. 48

2.9 Speedup of the LINEAR benchmark (with cutoff). 50
2.10 Speedup of the RECURSIVE benchmark (with cutoff). . . 51
2.11 Speedup of BOTS. 52
2.12 Comparison to other tasking models (LINEAR). 52
2.13 Comparison to other tasking models (RECURSIVE). . . . 53

3.1 Edge detector (DAG). 62
3.2 OpenCL mapping for STHORM (source: STMicroelectronics). 63
3.3 Execution of a single kernel on CLE run-time. 65
3.4 Structure of a tiling descriptor. 67
3.5 Image tiling schema for a single kernel. 68
3.6 Classes of image processing kernels. 69
3.7 Application graph partitioning. 72
3.8 Node scheduling algorithm. 73
3.9 Scheduling order for edge detector. 74
3.10 Example of tile size propagation. 75
3.11 Buffer allocation for edge detector. 76
3.12 Buffer sizing algorithm. 77
3.13 Example of CLE run-time schedule. 77
3.14 OpenVX graph of Canny benchmark with tiling annotations. 83
3.15 Speed-up of OVX CLE w.r.t. standard OpenCL approach. 85
3.16 Bandwidth reduction using OVX CLE. 86
3.17 Bandwidth required by applications for both OVX CLE and OpenCL. 87
3.18 Breakdown analysis of accelerator efficiency. 88
3.19 Efficiency of Edge detector and Retina preprocessing with memory bridge at 400 MHz (simulation). 89

4.1 Speed-up of OpenVX compared to OpenCL (ADRENALINE). 105
4.2 Speed-up of OpenVX compared to OpenCL (STHORM board). 105
4.3 Mapping of Canny and FAST9. 107
4.4 Canny edge detector. 107
4.5 Evaluation of Canny edge detector. 108

5.1 Heterogeneous architecture model. 114
5.2 Example of an OpenVX graph and its expansion after tiling. 120
5.3 Developer workflow using our approach. 124
5.4 Control function generation (pseudo-code). 126
5.5 Control function code. 128
5.6 Graph representation. 129
5.7 Example of graph modifications. 131
5.8 Code footprint (GB-RTE vs CG-RTE). 138
5.9 Total memory savings (CG-RTE vs GB-RTE). 139
5.10 Execution time (GB-RTE vs CG-RTE). 140
5.11 Energy efficiency of the STM32-L476 host compared to the PULP accelerator. 141
5.12 Frame memory bandwidth. 142

6.1 PULP architecture with hybrid L1 memory. 155
6.2 Breakdown analysis of the PULP SoC area. 156
6.3 Hybrid TCDM organization. 159
6.4 Reconfigured address space for a word level access in the split memory area. 160
6.5 Evolution of variables in different program regions. 162
6.6 A sparse matrix computation with a tolerant directive. . . 163

6.7 Layout of the PULP chip used for memory reliability and power characterization. 171
6.8 Normalized energy consumption of the TCDM (average energy reduction is reported in percentage). 174
6.9 Normalized components of the energy consumption. 176
6.10 Normalized energy consumption of the full SoC. 177
6.11 Normalized energy consumption for two solutions (SCM with tiling VS hybrid memory with our approach). 180
6.12 Energy consumption and area compared to Split-VS. . . . 183

List of Tables

3.1 OpenVX framework objects. 58
3.2 OpenVX data objects. 58
3.3 Details on OpenVX benchmarks. 81
3.4 Comparison with KernelGenius 90

4.1 Computed speed-up. 108
4.2 Effect of DMA latency on execution time. 109

5.1 OpenVX program phases. 119

6.1 Probability of bit-flip errors in 6T-SRAM. 157
6.2 Memory layout changing the 6T-SRAM voltage domains. . 158
6.3 Energy consumption (pJ) of a 32-bit read/write access. . . 166
6.4 Characterization of the benchmark suite. 170
6.5 Leakage energy (pJ per cycle) of SCM and 6T-SRAM cuts. 172
6.6 Total number of core cycles (in millions). 178
6.7 Area and energy/area values. 179
6.8 Mean squared error for zeroing and flip-bit error. 183
6.9 Normalized energy-area product (NEAP). 185

Abstract

Nowadays multi- and many-core computing platforms are widely adopted as a viable solution to accelerate compute-intensive workloads at different scales, from low-cost devices to HPC nodes. It is well established that heterogeneous platforms including a general-purpose host processor and a parallel programmable accelerator have the potential to dramatically increase the peak performance/Watt of computing architectures. However the adoption of these platforms further complicates application development, whereas it is widely acknowledged that software development is a critical activity for the platform design, as it affects development cost and time-to-market. The introduction of parallel architectures raises the need for programming paradigms capable of effectively leveraging an increasing number of processors, from two to thousands. In this scenario the study of optimization techniques to program parallel accelerators is paramount for two main objectives: first, improving performance and energy efficiency of the platform, which are key metrics for both embedded and HPC systems; second, enforcing software engineering practices with the aim to guarantee code quality and reduce software costs. This thesis presents a set of techniques that have been studied and designed to achieve these objectives overcoming the current state-of-the-art. As a first contribution, we discuss the use of OpenMP tasking as a general-purpose programming model to support the execution of diverse workloads, and we introduce a set of runtime-level techniques to support fine-grain tasks on high-end many-core accelerators (devices

with a power consumption greater than 10W). Then we focus our attention on embedded computer vision (CV), with the aim to show how to achieve best performance by exploiting the characteristics of a specific application domain. In this context we introduce an extension of the OpenCL programming model to support execution of graph-based workloads, which are common in CV applications, and we design a runtime tailored for many-core accelerators with a power envelope in the range 2-5 W. To enhance the programmability of this approach, we take into account the use of Domain Specific Languages (DSLs), and we design a runtime based on a standard API for embedded CV, namely OpenVX. However experimental results show that a standard OpenVX runtime has severe limitations when applied to mW-scale parallel accelerators, in particular when we impose strict constraints on local memory size and off-chip bandwidth. To overcome these limitations, we extend the OpenVX programming model to support offline code generation, and we design a lightweight OpenVX runtime tailored for resource-constrained mW-scale parallel accelerators. To further reduce the power consumption of parallel accelerators beyond the current technological limits, we describe an approach based on the principles of approximate computing, which implies modification to the program semantics and proper hardware support at the architectural level.

Introduction

I Background

I.I The roadmap to many-core architectures

Following the trend predicted by *Moore's law* [1] in 1965, the number of transistors in an integrated circuit doubles at each generation of semiconductor technologies, approximately every two years. The feature size of MOS transistors, corresponding to the minimum length of the channel between drain and source terminals, is scaled by around 30% (i.e., by a factor $S \simeq \sqrt{2}$) in a new generation, so that the area is accordingly scaled by 0.5 ($1/S^2$). This effect also reduces the delay by 30% (0.7×) and therefore increases operating frequency by about 40% (1.4×). *Dennard scaling* [2] is another important law first enunciated in 1974, and it states that power consumption stays proportional to area by scaling down both voltage and current with transistor size. Dennard scaling can be easy explained from the formula to compute the power consumption of a generic CMOS device,

$$P = \alpha C V_{dd}^2 f + V_{dd} I_{sc} + V_{dd} I_{leak}$$

where α is the transistor activity rate, C the load capacitance of transistors, V_{dd} is the operating voltage, f is the operating frequency of the chip, I_{leak} is the leakage current and I_{sc} is the short-circuit current. The assumption of Dennard scaling is that I_{sc} and I_{leak} are not significant for the total power balance, and the corresponding terms can be neglected. In this case the load capacitance is related only to area, when the size of the transistors shrunk and the voltage is reduced circuits could operate at higher frequencies at the same power level. For three decades (roughly from 1975 to 2005) Moore's law and Dennard scaling were the panacea of computing architectures, since they guaranteed that every two years circuits became 40% faster while total costs did not increase. Moreover, the same software targeting previous generations could be executed on new architectures taking automatic advantage of the improved speed without any code refactoring.

Dennard scaling broke down around 2005, with the introduction of feature sizes below 65 nm. With new technology nodes, scaling rules could no longer be sustained because of the exponential growth of leakage current. As transistors get smaller, power density increases because both leakage current and threshold voltage do not scale with size. This created an effect called *power wall* [3] that has limited practical processor frequency to around 4 GHz since 2006, as depicted in Figure 1. In the last years several techniques have been introduced to lessen the contribution of leakage current, such as the switching process used for 3D FinFET transistors [4], initially introduced on the market in 22 nm Haswell processors by Intel. However no further scaling of the operating voltage is possible at the current frequencies, and so its value is remained constant for several generations of mainstream processors (at around 1

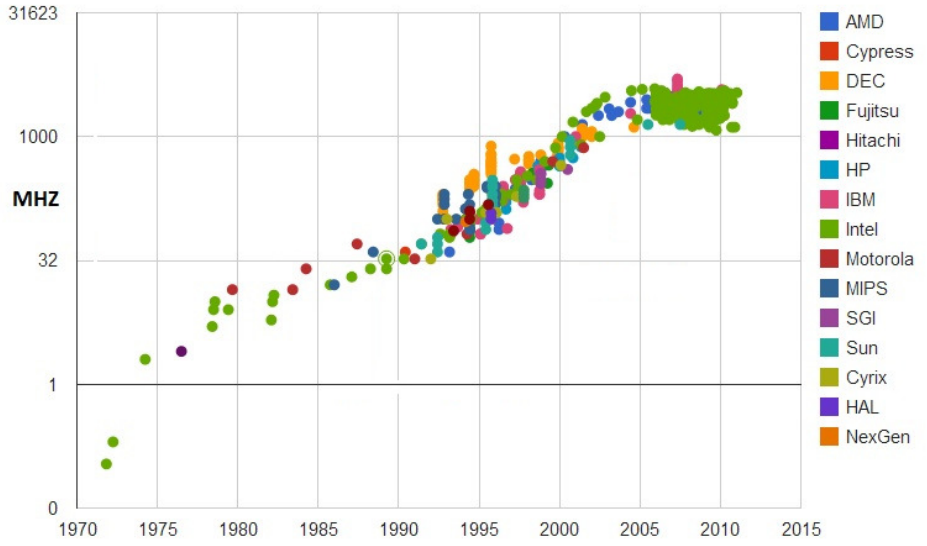


Figure 1: Trend of processor frequency over time (Data source: http://cpudb.stanford.edu).

V). In Post-Dennard scaling era, the increasing number of transistors yet provided by Moore's law leads to an equivalent power increase of $S^2 = 2$ (for a fixed die area). To meet the constraints of a target *thermal design power* (TDP), that is the maximum amount of heat generated by a computing system, two guidelines emerged from academic and industrial research: (i) organizing transistors in different and more energy efficient ways or (ii) switching off a subset of transistors to reduce instant power consumption. The introduction of multi-core processors is a solution adopting the first principle.

A *multi-core processor* is a computing system with two or more actual processing units (cores), with the capability to run multiple instructions at the same time. Basically, a multi-core processor implements multiprocessing on a single physical package. The main idea of this approach is to use the increasing transistor count to add more computing cores to

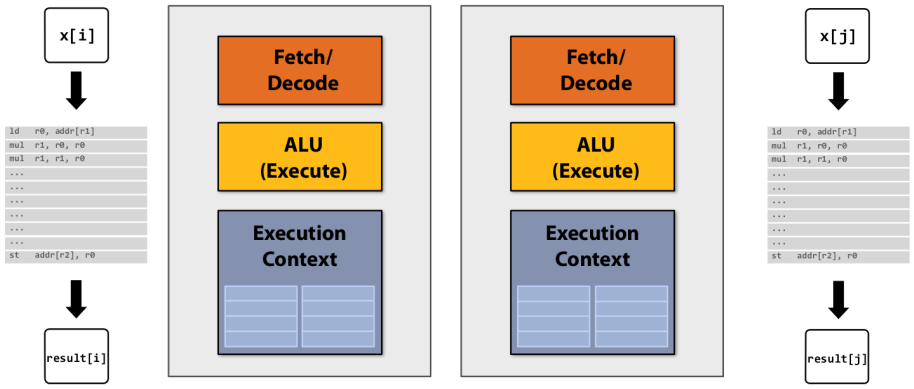


Figure 2: Two cores computing two elements in parallel.

a processor rather than use transistors to further increase sophistication of processor logic that accelerates a single instruction stream. In order to reduce power consumption, each core is slower on a single instruction stream than old-style monolithic cores (e.g., 25% slower), but concurrent execution on multiple cores provides a potential speed-up (e.g., having 2 cores that are 25% slower, the speedup would be $0.75 \times 2 = 1.5$). Figure 2 depicts the case of two cores performing the same computation on different elements in parallel. This specific configuration is typical of modern CPUs, where fetch/decode logic, execution units and execution contexts are replicated per core.

The multi-core design also mitigates two additional technical problems emerged in computer architectures, that are known as *memory wall* and *instruction-level parallelism (ILP) wall*. The memory wall is a limiting effect of computing speed due to the disparity of speed between CPU and off-chip memory, because of the limited communication bandwidth beyond chip boundaries. Until 2005, CPU speed improved at an annual rate of 55% while the improvement of memory speed was stuck to 10%. Given these trends, research studies [5] forecasted that memory

latency would become a limiting factor in achieving better performance, and this was confirmed on market products by Intel in 2005 [6]. Multi-core architectures address the memory wall by implementing deep and distributed cache hierarchies, which reduce the average requests on the memory subsystem and hide the access latency. The ILP wall [7] is an effect of diminishing returns on instruction-level parallelism, due to the increasing difficulty of extracting additional parallelism from a single instruction stream. At the ILP wall each increment in the clock rate has a corresponding clock-per-instruction (CPI) increase, and accordingly more instructions cannot be fetched and decoded per clock cycle. In addition, poor data locality further limits ILP and adversely affects memory bandwidth. Multi-core architectures address the ILP wall by increasing granularity of parallelism at thread level, introducing thread-level parallelism (TLP). This choice has serious implications on software programming, which are addressed in the next section.

The term *many-core architectures* is typically used to describe multi-core architectures with an especially high number of cores, from tens to hundreds. Today these platforms are considered a viable solution to accelerate compute-intensive workloads at different scales, from low-cost devices to HPC nodes. Nevertheless, the ever-increasing on-chip power density leads to an emerging scenario in which the ongoing many-core evolution is again limited by the power wall. Applying the second guideline discussed to reduce TDP, only a small fraction of a chip is powered at the same time. The term *dark silicon* is exactly used to define the amount of silicon of an integrated circuit that cannot be powered-on at the nominal operating voltage to fulfill TDP constraints. The main consequence of dark silicon is a technology-imposed *utilization wall*

[8] that limits the fraction of the chip we can use at full speed at one time. The heterogeneous architecture design, where a large number of different accelerators can be build on the same chip and can be woken up only when needed and for the specific task that was design for, is one of the most promising solutions to bypass the utilization wall [9]. Heterogeneous architectures including general-purpose host processors and programmable many-core accelerators are already available on the market, including: general purpose architectures like AMD APUs, which integrate multiple x86 cores with a general-purpose GPU on the same die [10]; mobile-centric products, like Samsung Exynos [11]; architectures for signal-processing, like Texas Instrument Keystone II [12] and NVidia X1 [13]; large many-core accelerators like Kalray MPPA 256 [14], PEZY-SC [15], ST Microelectronics STHORM [16] and Parallella Epiphany-V [17].

In this thesis we focus our attention on a significant class of embedded parallel accelerators, which are *clustered accelerators* sharing on-chip fast memory and communicating via low-latency, high-throughput on-chip interconnections. These devices differ from GPGPUs in two main traits. First, cores are not restricted to run the same instruction on different data, in an effort to improve execution efficiency of branch-rich computations and to support a more flexible workload distribution. Second, embedded many-core accelerators do not rely on massive multithreading to hide memory latency, but they rely instead on DMA engines and double buffering, which give more control on the bandwidth vs. latency tradeoff, but require more programming effort. Our target devices are described in further detail in Chapter 1.

I.II Programming models for parallel platforms

Parallel accelerators have the potential to dramatically increase the peak performance/Watt of embedded computing architectures, however their adoption further complicates application programming, whereas it is widely acknowledged that software development is a critical activity for the platform design, as it affects development cost and time-to-market [18]. The introduction of parallel architectures raises the need for programming paradigms capable of effectively leveraging an increasing number of processors, from two to thousands. According to software engineering principles, programming models should expose high-level constructs for outlining the available parallelism in applications, without the need for programmers to handle performance scalability issues by expertising on low-level hardware details [19]. In the next paragraphs we briefly introduce the programming models that have been mostly adopted in this context.

OpenCL [20] is a standard which introduces platform and execution models that are particularly suitable for heterogeneous platforms including many-core accelerators, and it has been widely adopted by industry in the last years. OpenCL offers a conjunction of task parallel programming model, with a run-to-completion approach, and data parallelism, supported by global synchronization mechanisms. An OpenCL application runs on the host processor and distributes kernels on computing devices, and programmers must write separate code for the host and the accelerator. In the OpenCL platform model, each device is composed by one or more *compute units*, each one composed of one or more processing elements and a local memory. Kernels are programmed in the

OpenCL C language, which is based on the C99 standard. Host applications are written in C/C++ language, and invoke standard application programming interface (API) calls to orchestrate the distribution and execution of kernels on devices, using a mechanism based on command queues. The execution of a kernel invoked on a device by a command on a single processing element is called a *work-item*, while a collection of related work-items that execute on a single compute unit is called a *work-group*. Using OpenCL model, program execution is explicitly orchestrated by the host code, including data transfers and synchronization points. The last version of the standard (OpenCL 2.0) also enables dynamic parallelism on device side, but most programming environments do not support it yet. OpenCL offers a conjunction of task parallelism (with a run-to-completion approach) and data parallelism, supported by global synchronization mechanisms. In addition, OpenCL defines a memory abstraction model that is common to all computing devices implementing the standard. There are four virtual memory regions (global, constant, local, private), which are mapped to actual memory hierarchies of target devices. An host-side API defines how data is stored and also enables the orchestration of data transfers. The most common critic to OpenCL is due to the fact that it offers a very low-level programming style, and existing applications must be rewritten entirely to comply to programming practices that could be tedious and error-prone.

OpenMP [21] is a programming model for shared memory multiprocessing architectures. It is mainly based on preprocessing directives for C/C++ and Fortran, which are resolved at compile time by means of source-to-source code transformations into low-level calls to a target-specific runtime. OpenMP was initially used for multi-core CPUs, but

latest versions of the standard provide specific mechanisms to execute code on parallel accelerators. A similar approach is used by OpenACC [22], another annotation-based programming model which has been widely adopted in recent years. The use of directives does no alter exiting sequential code, which enables rapid and maintainable code development thanks to an incremental parallelization style coding. OpenMP provides a rich set of constructs which enable to exploit parallelism at different levels, but their usage is often limited to applications exhibiting work units 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 the embedded computing systems which are the main target of this dissertation.

To enhance the programmability of accelerators, programmers often take into account the use of Domain Specific Languages (DSLs). The adoption of a DSL enables hardware vendors to implement and optimize low-level domain-specific primitives. OpenVX [23] is a cross-platform C-based API which aims at enabling hardware vendors to implement and optimize low-level image processing and computer vision (CV) primitives. In the context of image processing, applications can be easily structured as a set of vision kernels (i.e. basic features or algorithms) that interact on the basis of input/output data dependencies. Relying on this consideration, OpenVX promotes a graph-oriented programming model, based on Directed Acyclic Graphs (DAGs) of kernel instances.

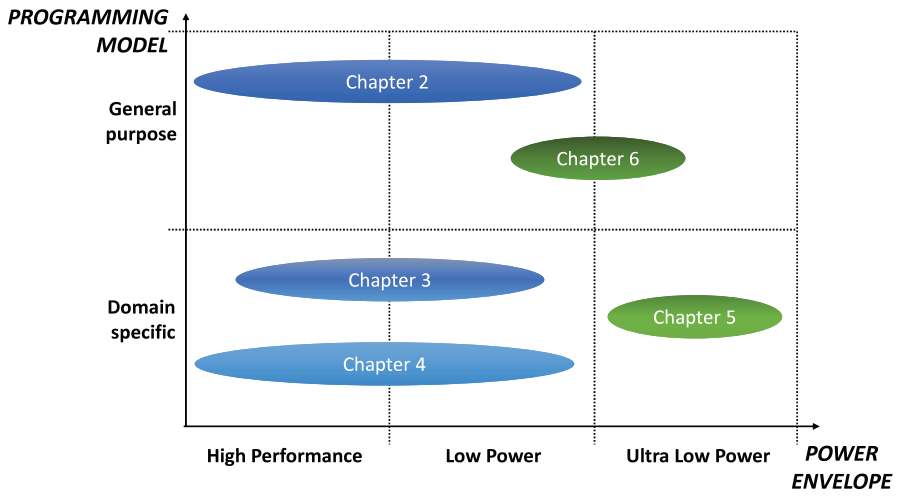


Figure 3: Thesis organization.

II Thesis contributions

Considering the depicted scenario, the study of optimization techniques to program parallel accelerators is paramount for two main objectives: first, improving both performance and energy efficiency of parallel platforms, which are key metrics for embedded computing systems; second, enforcing software engineering practices with the aim to guarantee code quality and reduce software costs. This thesis presents a set of techniques and designs that have been studied to achieve these objectives. In the dissertation we consider a range of devices with a more and more stringent power budget, since the emergence of new application areas (e.g., Internet-of-Things applications) makes a common requirement that devices combine complex processing capability with ultra-low-power operation. The next paragraphs summarize the major contributions of this thesis, and Figure 3 depicts an overview of the chapters in terms of target power envelope (x-axis) and programming model classification (y-axis).

Enabling fine-grained OpenMP tasking on many-core accelerators (Chapter 2). As a first contribution, we discuss the use of OpenMP tasking as a general-purpose programming model to support the execution of diverse workloads, and we introduce a set of runtime-level techniques to support fine-grain tasks on many-core accelerators. The tasking abstraction provides a powerful conceptual framework to exploit irregular parallelism in embedded applications, but its practical implementation requires sophisticated runtime support, which typically implies important space and time overheads. The applicability of this approach is often limited to applications exhibiting work units 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 computing systems. Minimizing runtime overheads is thus a primary challenge to enable the benefits of tasking on these systems. We designed an optimized runtime environment supporting the OpenMP tasking model on an embedded shared-memory cluster [24] [25], validating our work on Kalray MPPA. Experimental results on compute-intensive applications show that this approach can achieve the *maximum speed-up with an average task granularity of 7500 cycles*, while previous approaches require about 100000 cycles to achieve the same performance level.

Extending the OpenCL model for efficient execution of graph-based workloads on many-core accelerators (Chapter 3). Data transfers and memory management are major challenges in programming a heterogeneous platform. OpenMP provides programmers a convenient way to express code parallelism, but the standard does not address these

issues. To overcome this limitation we explored the use of OpenCL, that defines a unified memory abstraction model and provides a standard API to handle data transfers. Our preliminary experiments highlighted that a common issue of using OpenCL in the context of embedded parallel accelerators is related to the mandatory use of the global memory space to share intermediate data. Global memory is available to any work-item for both read and write, and must be fully accessible (and cacheble) by the host CPU, so it is usually mapped on off-chip memory (e.g. a DDR). When increasing the number of interacting kernels, the off-chip memory bandwidth required to fulfill data requests originated by work-items is much higher than the available one. Moreover, data must be transferred into local memory (on-chip) prior to computation to take advantage of low-latency accesses, and this has to be done explicitly; consequently, code modularity and execution efficiency become conflicting constraints. We focused our attention on embedded computer vision (CV) benchmarks, since this class of applications is particularly sensitive to memory effects [26]. In this context we introduced an OpenCL extension to support graph-based workloads, which are common in a large class of applications (including the CV domain), and we designed a runtime tailored for multi- and many-core accelerators [27]. This runtime extends the OpenCL semantics by enabling the creation of low-level graphs which contains nodes of different types (data allocations, DMA transfers and processing kernels), with the aim to allow the separation of concerns between algorithm code and data management. To enhance the programmability of this approach, we took into account the use of Domain Specific Languages (DSLs) as a programming front-end and we designed a runtime based on a standard API for embedded CV, namely

OpenVX [28]. Experiments performed on STHORM showed that our solution provides huge benefits in terms of speed-up compared to the execution of standard OpenCL code. This approach achieves up to *9.6× speed-up w.r.t. OpenCL*, mainly due to a drastic bandwidth reduction. This also implies an higher *execution efficiency*, having more than *95% of the accelerator time spent in active processing.*

An OpenVX environment to optimize embedded vision applications on parametric parallel accelerators (Chapter 4). OpenVX has been introduced with the aim to raise significantly the level of abstraction at which CV applications should be coded. Based on a standard plain C API, it is easy to use and fully transparent to architectural details. The details of the hardware platform are hidden in the underlying run-time environment (RTE) layer. This approach enables the portability of vision applications across different heterogeneous platforms, delegating the performance tuning to hardware vendors, who provide an efficient RTE with architecture-specific optimizations. In this context we designed ADRENALINE [29], a framework for fast prototyping and optimization of OpenVX applications on heterogeneous platforms. ADRENALINE consists of an optimized OpenVX runtime, based on streamlined support for a generic heterogeneous platform, and a virtual platform modeling the hardware architecture, using a template which can be easily configured along several axes (e.g., core type, memory timings). The runtime system includes several optimizations for the efficient exploitation of the explicitly managed memory hierarchy, mainly *tiling* and *double-buffering*. A low-level application graph is automatically generated by the high-level OpenVX code, which implies a consistent reduction of source

code complexity (*25% reduction of source code lines*). In addition, this tool can support several end users: (i) *researchers and SDK vendors* can explore various platform-specific optimizations, scheduling policies and algorithms for the implementation of the OpenVX support layer; (ii) *application developers* can explore different partitioning solutions (host vs accelerator, parallelization) for different applications; (iii) *platform engineers* can quickly evaluate different architectural configurations for a target CV application.

Providing lightweight OpenVX support for resource-constrained mW-scale parallel accelerators (Chapter 5). The typical power envelope of commercial parallel accelerators is in the range 2–15 W. For applications with a more constrained power budget, which are more and more common in the IoT domain, we take into account a new class of ultra-lower-power heterogeneous platforms, which include a microcontroller unit (MCU) coupled to a mW-scale parallel accelerator. A key trait of these platforms is the strongly limited amount of available on-chip memory, which requires dedicated memory management techniques to enable the efficient execution of CV graphs of arbitrary large size, also considering a limited off-chip bandwidth. Experimental results show that a standard OpenVX runtime has severe limitations when applied to on this class of devices, since the scarce amount of on-chip memory can be insufficient to contain data and code. To overcome these limitations, we extended the OpenVX programming model to support offline code generation, and we designed a lightweight OpenVX runtime (milliVX) tailored for MCU hosts. This approach takes advantage of the static structure extracted by an OpenVX program to optimize the execution stage in

terms of memory footprint and execution time, but at the same time it fully preserves the dynamic features of the original OpenVX standard, namely graph updates and node callbacks [30]. To assess our approach we designed a reference implementation for the OpenVX extension and the milliVX specification. This approach achieves *68% memory footprint reduction and 3× execution speed-up* compared to a baseline implementation. At the same time, *memory bandwidth used for data transfers is reduced by 10% and energy efficiency is improved by 2×.*

A new frontier: supporting approximate computing on mW-scale parallel accelerators (Chapter 6). To further reduce the power consumption of parallel accelerators beyond the current technological limits, we introduce an approach based on the principles of approximate computing, which implies modification to the program semantics and proper hardware support at the architectural level [31]. We designed a hybrid memory system for the L1 scratchpad, including both Standard Cell Memory (SCM) and six-transistor Static RAM (6T-SRAM). Although 6T-SRAM provides a much better storage density than SCM, its minimum operating voltage is much higher (0.8 V as opposed to 0.5 V for the considered technology node). Accessing 6T-SRAM below 0.8 V results in a flip-bit error with a certain probability, but the operating voltage of 6T-SRAM can be safely lowered whenever data located in this region are not accessed. In addition, to leverage approximation tolerance in applications in a controlled manner we provide a transparent mechanism to split multi-byte data into multiple banks at the architecture level (tolerant memory region). The most significant bits (MSB) of a word are stored in the SCM, while the least significant bits (LSB)

are stored in the 6T-SRAM. This allows to access 6T-SRAM cells at low voltage for error-tolerant computations, bounding the error to the LSB. Our results demonstrate that this approach provides much better precision than just dropping the LSB. At the programming model level, we provide constructs for specifying regions of code and data that are tolerant to approximation in a C program. A compiler pass places data into different memory areas according to error tolerance, activating the tolerant memory region and inserting voltage switch points for 6T-SRAM when it is safe. A greedy allocation algorithm uses a heuristic to minimize the power consumption due to data accesses when other policies are not possible. Using cycle accurate simulation models of the target platform annotated with power numbers extracted from a silicon implementation, we demonstrate that this architecture can *reduce by 25% on average the energy consumption*, very close to the savings achievable by an ideal platform including SCM cells only.

Chapter 1

Overview of embedded many-core platforms

1 Platform template

Figure 1.1 shows a block diagram of the architectural template for embedded many-core computing platforms that is the main reference for this thesis. It consists of a general-purpose host processor coupled with a clustered many-core accelerator (CMA) inside an embedded system-on-chip (SoC) platform. The multi-cluster design is a common solution applied to overcome scalability limitations in modern many-core accelerators, such as STMicroelectronics STHORM [16], Kalray MMPA-256 [14] and PULP [32]. Clusters are typically interconnected via a network-on-chip (NoC).

The processing elements (PEs) inside a cluster are fully independent reduced instruction set computing (RISC) cores, supporting both single-instruction/multiple-data (SIMD) and multiple-instruction/multiple-data (MIMD) parallelism. Each PE is equipped with a private instruction

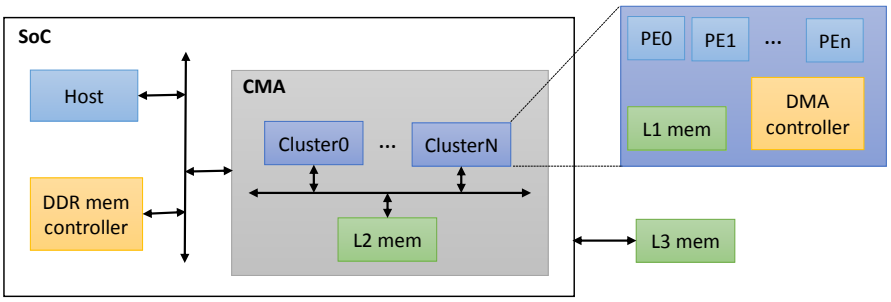


Figure 1.1: Reference architectural template for many-core computing platforms.

cache. To avoid memory coherency overhead and increase energy efficiency, the PEs do not have private data caches. All PEs share a L1 memory (L1 mem) acting as a scratchpad. L1 memory is typically implemented with a set of SRAM banks to which processors are connected through a low-latency, high-bandwidth data interconnect. Communication between the cores and the L1 memory is based on a fast interconnect, implementing a word-level interleaving scheme to reduce the access contention to L1 banks. Multiple concurrent reads at the same address happen in the same clock cycle (broadcast). A conflict takes place only when multiple processors try to access different addresses within the same bank. In this case the requests are sequentialized on the single bank port. To minimize the probability of conflicts the interconnection implements address interleaving at the word-level and the number of banks is M times the number of cores ($M = 2$ by default). Processors can synchronize by means of standard atomic operations (e.g., test-and-set), which read the content of the target memory location and updates it.

The architectural template also includes a L2 scratchpad memory at SoC level and an external L3 (e.g., using DDR or Flash technology) accessible by means of a memory controller. Both host cores and PEs

can access the whole memory space, that is modeled as a partitioned global address space (PGAS). Since the L1 memory has a small size, the software must explicitly orchestrate data transfers from other memory levels to L1 to ensure that the most frequently referenced data at any time are kept close to the processors. A direct memory access (DMA) unit enables communication with other clusters, L2 memory and external peripherals.

2 STMicroelectronics STHORM

STHORM accelerator by STMicroelectronics, previously known as P2012 [16], is a many-core computing accelerator fabricated in 28 nm bulk CMOS technology. Its design is based on clusters interconnected by an asynchronous network-on-chip (Figure 1.2). The accelerator fabric also includes a fabric controller (FC) core, intended to manage fabric-level run-time and interaction with the host processor.

Each cluster features 16+1 cores, 16 PEs to perform general-purpose computation and a cluster controller (CC) to handle cluster-level run-time. All the cores (FC, CC, PEs) are dual-issue STxP70 processors, supporting multiple-programs multiple-data (MPMD) instruction streams. This means that each core can execute independent programs on independent data. Moreover, each cluster contains a multi-banked one-cycle access L1 scratchpad memory, connected by a multi-level logarithmic interconnect, and a dual-channel DMA engine that can handle both linear and rectangular transfers. A L2 scratchpad memory is shared by all clusters. The architecture has no data cache, as it is designed to minimize SoC size and energy consumption.

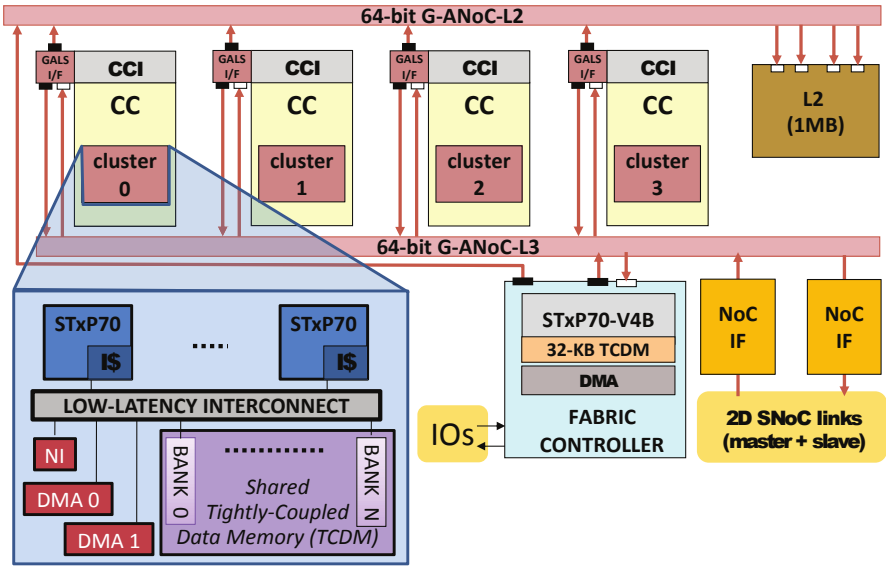


Figure 1.2: STHORM architecture.

The STHORM evaluation board (Figure 1.3) is based on ZedBoard open-hardware design [33]. It includes a Xilinx Zynq 7020 chip, featuring an ARM Cortex A9 dual core host processor operating at 667 MHz plus FPGA programmable logic, and a STHORM chip clocked at 430 MHz. The ARM subsystem on the Zynq is connected to an AMBA AXI interconnection matrix, through which it accesses the DRAM controller. The latter is connected to the on-board DDR3 (500 MB), which is the third memory level in the system (L3) for both ARM and STHORM cores. To allow transactions generated inside the STHORM chip to reach the L3 memory, and transactions generated inside the ARM system to reach internal STHORM L1 and L2 memories, part of the FPGA area is used to implement an access *bridge*. The FPGA bridge is clocked very conservatively at 40 MHz; consequently, the main memory bandwidth available to the STHORM chip is limited to 250 MB/s for the read channel and

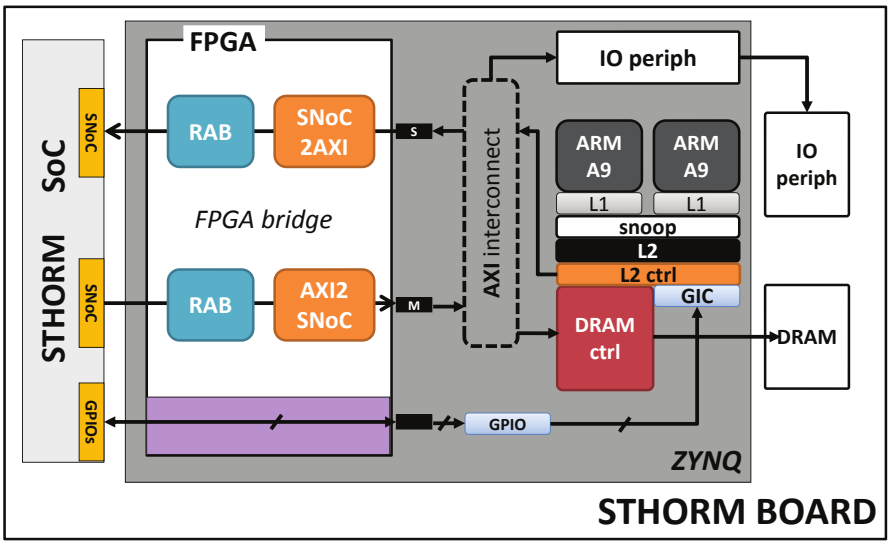


Figure 1.3: STHORM evaluation board.

125 MB/s for the write channel, with an access latency of about 450 cycles. Compared with ARM processor on the Zynq, which access DDR3 through a bus at 533 MHz, STHORM chip is severely penalized. Clearly, in a full production SoC scenario host and accelerator share the same silicon die, and consequently the accelerator gets a much larger share of the main memory bandwidth (e.g.. 1/2 instead of 1/12 as in the evaluation board). Hence, the evaluation board represents a very challenging and interesting scenario for any software optimization focusing on reducing main memory bandwidth needs.

3 Kalray MPPA-256

The Kalray MPPA-256 [14] is a single-chip many-core processor manufactured in 28 nm CMOS technology for compute intensive embedded applications, and it is based on MPPA (Multi-Purpose Processor Array)

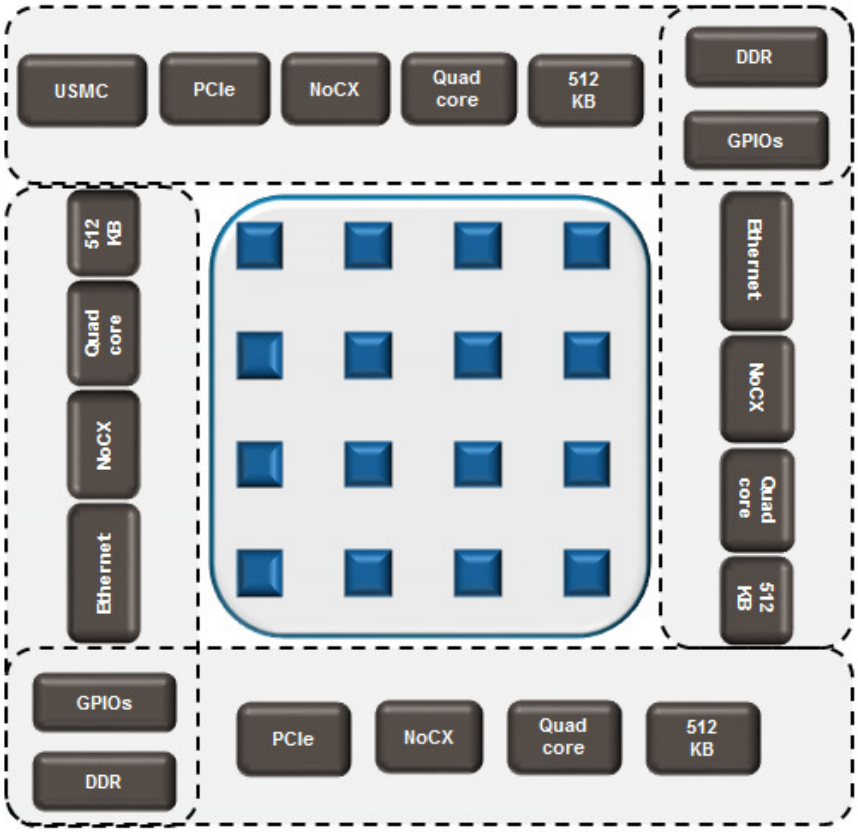


Figure 1.4: MPPA-256 block diagram (Source: Kalray Corporation).

technology by Kalray. This product features 256 processors on a single die, and it is composed of an array of 16 clusters connected through a high-speed NoC. Figure 1.4 depicts the cluster-based structure of a MPPA-256 chip.

Each compute cluster is composed of 16 identical cores, plus a system core (with private FPU and MMU) and a shared memory (2 MB). The cluster is equipped with Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Switch off (DPS) support plus a DMA. Processors communicate through shared memory. Specifically, the cores are connected to a multi-bank memory enabling low latency access or

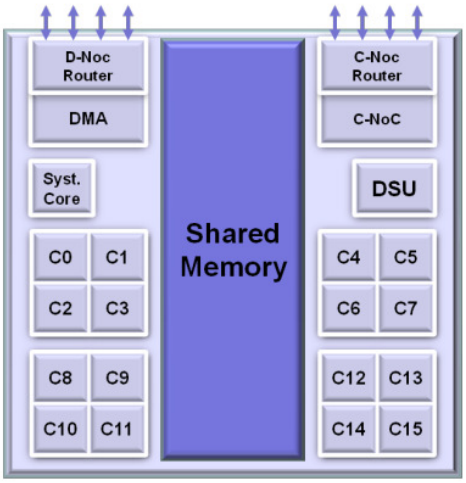


Figure 1.5: MPPA-256 compute cluster (Source: Kalray Corporation).

bank private access depending on the configuration. The cluster also features one Debug Support Unit (DSU). Figure 1.5 shows the structure of a MPPA-256 cluster.

The core architecture within MPPA clusters is a 32-bit Very Long Instruction Word (VLIW) processor including: a Branch/Control Unit; two Arithmetic Logic Units; a Load/Store Unit (LSU), including simplified ALU; a Multiply-Accumulate (MAC) / FPU, including a simplified ALU; a Standard IEEE 754-2008 FPU with advanced Fused Multiply-Add (FMA) and dot product operators; a Memory Management Unit (MMU). This architecture enables to execute up to five 32bit RISC like integer operations every clock cycle. In addition, every core is equipped with private instruction and data L1 caches.

Multiple clusters are interconnected through a NoC, providing a full duplex bandwidth up to 3.2 GB/s between each adjacent cluster. The NoC implements a Quality of Service (QoS) mechanism, thus guaranteeing predictable latencies for all data transfers.

The MPPA-256 processor communicates with external devices through I/O subsystems located at the periphery of the NoC. The I/O subsystems implement various standard interfaces: two DDR3 channels (64-bit with optional ECC, up to 12,8GB/s); two PCIe Gen3 X8; two Ethernet controllers; a universal Static Memory Controller; two banks of 64 General Purpose I/Os. Moreover, the NoC eXpress interfaces (NoCX) provides an aggregate bandwidth of 40 Gb/s, and enables the possibility to scale the number of cores by connecting multiple processors on the same board, or alternatively to couple an external FPGA used as a co-processor or interface bridge.

The I/O subsystems are controlled through a quad-core processor. The cores are based on the same VLIW architecture adopted within MPPA clusters. These processors operate as controllers for the MPPA clusters. Any program is started on the I/O cores, which are then responsible to properly offloading computation to the clusters.

The MPPA 256 processor can also be optionally connected to a market standard host processor, like in Kalray EMB01 board which includes a x86 host (AMD E-series processor).

4 PULP

Currently ultra-low-power (ULP) embedded systems are largely based on microcontrollers featuring simple, cache-less cores (e.g., Cortex M0 or M4), coupled to a simple support for power management and a standard set of peripherals. The parallel processing ultra-low-power platform (PULP) [32] [34] aims at providing a significant boost to the peak performance that ULP systems can achieve by coupling the multi-core

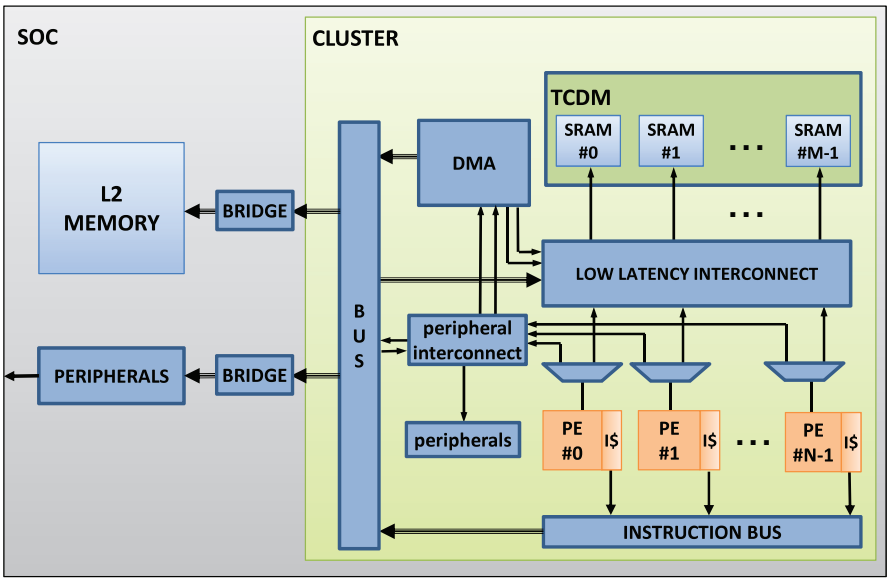


Figure 1.6: PULP cluster architecture.

paradigm to the most advanced fully-depleted silicon-on-insulator (FD-SOI) design technology and associated techniques for energy efficiency, mostly near-threshold computing and body biasing [35].

PULP is a scalable, many-core computing fabric, organized as a set of clusters. Figure 1.6 shows the main building blocks of single-cluster PULP SoC. Multiple clusters can be interconnected at the top level to share L2 memory and peripherals for off-chip communication. A cluster includes a parametric number of processing elements (PEs) consisting of an optimized microarchitecture based on the OpenRISC ISA [36], each equipped with a private instruction cache. To avoid memory coherency overhead and increase energy efficiency the PEs do not have private data caches, but they share a L1 multi-banked tightly coupled data memory (TCDM). The TCDM is configured as a shared scratchpad memory, featuring as many R/W ports as the number of memory banks. This allows concurrent access to memory locations mapped on different banks,

via a one-cycle-latency logarithmic interconnect implementing word-level interleaving to reduce contention. The whole memory space (L1, L2, memory-mapped peripherals) is visible to all the cores of the cluster. A lightweight DMA enables fast and flexible communication with other clusters, the L2 memory and external peripherals [37]. The DMA unit guarantees a fixed programming latency (10 cycles), featuring up to 16 outstanding transactions and multiple physical control ports (one per core, with the aim to limit port contention). The DMA unit has also a direct connection to the TCDM, using dedicated ports on the interconnect. This eliminates the need for data buffering in the DMA engine, which is very expensive in terms of area and power. The arbitration and protocol adaptation necessary for the processors to communicate to the TCDM and peripheral interconnect is implemented by the DEMUX block connected to the data interface of each PE.

To enable the SoC to achieve high energy efficiency, each cluster can provide on-demand shutdown of cores by means of fine-grain partitioning into regions with separate clock trees and isolated wells. The polarization of the P-well and N-well of each domain can be selected using a set of body biasing multiplexers, choosing between two couples of global voltages (forward or reverse body biasing). In contrast with other approaches such as DVFS and power gating, this architecture has minimum overhead in terms of area (less than 1%) and does not require level shifters and power grid isolation.

Chapter 2

Enabling fine-grained OpenMP tasking on many-core accelerators

1 Introduction

In this chapter we describe the design of an optimized runtime environment supporting the OpenMP tasking model on embedded many-core accelerators. OpenMP historically relied on a *fork/join* parallel execution model guided by directives. The program starts with a single thread of execution (called *master*); when a `parallel` construct is encountered, $n-1$ additional threads (n can be specified with the `num_threads` clause) are recruited into a parallel *team*. Several work-sharing constructs are provided to specify how the parallel workload is distributed among threads. Since the specification version 3.0, on top of the fork-join model OpenMP provides support for task-based parallelism, which is our focus.

When a thread encounters a `task` construct, a new *task region* is generated from the code contained within the task. Additional data-sharing clauses specify an associated data environment, while the execution of the new task can be assigned to one of the threads in the team, based on additional *task scheduling* clauses that specify i) dependencies among tasks; ii) (conditional) immediate or deferred execution; iii) task type, between `tied` and `untied` (referred to the thread that first encounters them).

Tied tasks are the default in OpenMP, as they attempt to establish a trade-off between ease of programming and scheduling flexibility (and thus, performance) [38]. If a *tied* task is suspended, it can later only be resumed by the same thread that originally started it. `Untied` tasks are not bound to any thread and so in case they are suspended they can later be resumed by any thread in the team. Using `untied` tasks has the potential for significantly increasing the achievable parallelism, but comes at the cost of a higher programming effort (the programmer is responsible for avoiding issues such as deadlock, thread-private memory, etc.). All tasks bound to a given parallel region are guaranteed to have completed at the implicit barrier at the end of the parallel region, as well as at any other explicit `barrier` construct. Synchronization over a subset of explicit tasks can be specified with the `taskwait` construct, which forces the encountering task to wait for all its first-level descendants to complete before proceeding.

The tasking abstraction provides a powerful conceptual framework to exploit irregular parallelism in embedded applications, but its practical implementation requires sophisticated runtime support, which typically implies important space and time overheads. The applicability of this

approach is often limited to applications exhibiting work units 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 computing systems. Minimizing runtime overheads is thus a primary challenge to enable the benefits of tasking on these systems.

2 Related work

Tasking model is a well known paradigm in the domain of general-purpose computing and in last decade it has been successfully adopted to program multi-core architectures. Cilk [39], Intel TBB [40], Wool [41], Apple GCD [42] and the current OpenMP specification [21] are notable examples of task-based programming models. While OpenMP has recently gained much attention also in the embedded domain [43] [44] [45], not much work has been done on demonstrating the benefits of tasking for fine-grained embedded workloads or for proposing lightweight and efficient tasking implementations for embedded multi-core platforms. The main limitation of most tasking-enabled runtimes is the lack of efficient support for *untied* tasks and nested parallel patterns, which are the ones for which task-based parallelism is most beneficial. Our work addresses these shortcomings and proposes a lightweight tasking runtime capable of enabling near-ideal speedups for recursive parallel patterns employing very fine-grained tasks.

3 Design of an embedded tasking runtime

The `task` construct can be used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When a thread encounters the `task` construct, it prepares a *task descriptor* containing a pointer to the code to be executed, plus a data environment inherited from the enclosing structured block. *shared* data items point to the variables with the same name in the enclosing region. New storage is created for *private* and *firstprivate* data items, and the latter are initialized with the value of the original variables at the moment of task creation. The execution of the task can be immediate or deferred until later by inserting the descriptor in a work queue from which any thread in the team can extract it. This decision can be taken at runtime depending on resource availability and/or on the scheduling policy implemented (scheduling policies are discussed in detail in Section 4). However, a programmer can enforce a particular task to be immediately executed by using the `if` clause. When the conditional expression evaluates to false the encountering thread suspends the current task region and switches to the new task. To provide work-unit based synchronization, the `taskwait` directive forces the current thread to wait for the completion of every tasks generated from the current task region. *Task scheduling points* (TSP) specify places in a program where the encountering thread may suspend execution of the current task and start execution of a new task or resume a previously suspended task.

Figure 2.1 shows our layered approach to designing the primitives for the tasking constructs. These constructs are depicted in the top layer blocks (in black). To manage OpenMP tasks we rely on a main work queue where units of work can be pushed to and popped from (bottom

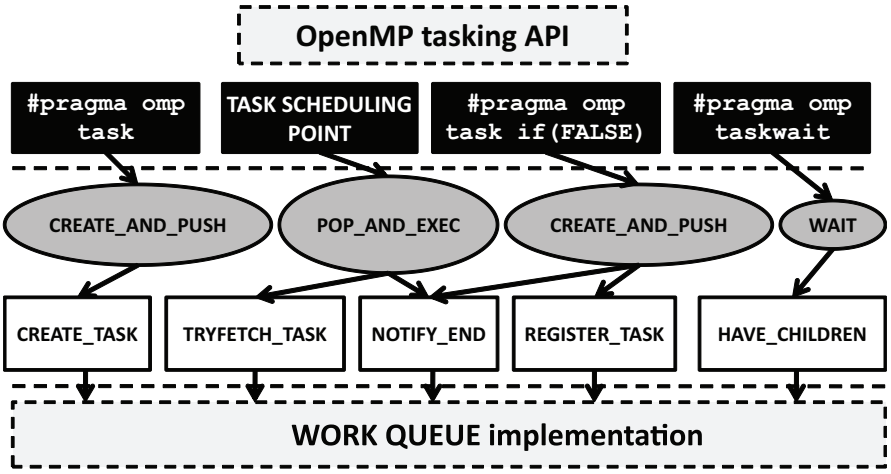


Figure 2.1: Design of tasking support.

layer block). The gap between OpenMP directives and the work queue is bridged by an intermediate runtime layer (gray blocks), which operates on the queue through a set of basic primitives (white blocks) to implement the semantics of the tasking constructs.

3.1 Work queue

Our design relies on a centralized queue with breadth-first, LIFO scheduling. Tasks are tracked through descriptors which identify their associated task regions and which are stored in the work queue. The two basic operations on the queue are task insertion and extraction. Inserting a task has two effects: i) creating a new descriptor for it, and ii) registering it as a child of the executing task (its parent). We formalize these semantics as a primitive that we call `CREATE_TASK`.

Extracting a task from the work queue retrieves its descriptor for execution. To this aim we consider a `TRYFETCH_TASK` primitive, which returns the task descriptor in case of successful extraction, or a NULL pointer if the work queue is empty. Task extraction should only return

the descriptor to the caller, not detach it from the work queue until the task has completed execution. This is necessary for correctly supporting synchronization (`taskwait` construct). We thus designed a separate `NOTIFY_END` primitive to dispose of the descriptor, which acts as an epilogue to task execution.

Note that since the `TRYFETCH_TASK` primitive does not remove the task descriptor from the work queue, it is necessary to mark it as running to avoid multiple extractions of the same descriptor. Thus, the `CREATE_TASK` inserts a waiting task in the work queue and the `TRYFETCH_TASK` changes its status to running. `NOTIFY_END` marks it as ended. To support undeferred tasks (e.g., whose if condition is evaluated to false) we introduce a `REGISTER_TASK` primitive which inserts a descriptor marked as running. Finally, the `HAVE_CHILDREN` primitive allows to determine if a task has children not yet assigned to a thread (i.e., in the waiting state). As we will explain in the next section, this is necessary to implement task switching capability in presence of a `taskwait`.

3.2 Runtime layer

Let us consider the simple example of the `task` construct in the code snippet of Figure 2.2. The `parallel` directive creates a team of worker threads, then only one of them executes the `single` block. This thread acts as a work producer, since it is the only one encountering the `task` construct. The control flow for the rest of the threads falls through the parallel region to the implied barrier at its end.

An important of the tasking execution model is related to TSPs. Parallel threads are allowed to switch from one task to another:

1. at `task` constructs;

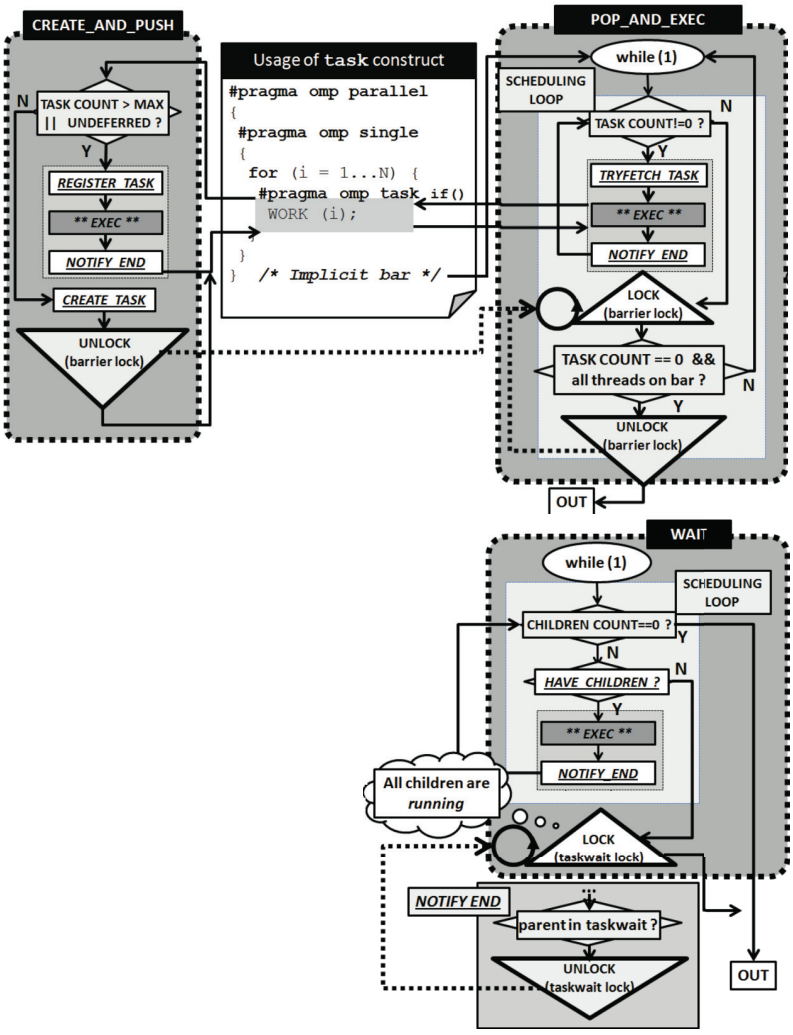


Figure 2.2: Design of task scheduling loop.

2. at implicit and explicit barriers;

3. at the end of the current task;

4. at `taskwait` constructs.

The first point prevents system oversubscription in cases where a thread is required to generate a very high number of tasks (e.g., the `task` directive is nested inside a loop with a huge number of iterations).

Placing a TSP on a `task` construct allows the producer thread to switch to executing some of the tasks already in the queue. Task creation is resumed once the queue has been depleted to a certain level.

To keep the implementation of task scheduling as simple as possible we deal with this issue in the following manner. Upon encountering a `task` directive, threads calls the `CREATE_AND_PUSH` runtime function, depicted on the left part of Figure 2.2. Here, the caller first checks for the number of tasks already in the queue. If this number exceeds a given threshold the thread does not insert the task in the queue, but it immediately executes it instead. Note that this can not be implemented through a simple jump to the task block code. Executing a task without creating a descriptor and connecting it to the others will in fact result in ignoring its existence, which may lead to incorrect functioning of the `taskwait` directive due to bad internal representation of the task hierarchy. Thus we create and insert in the queue a descriptor for a running task through the `REGISTER_TASK` primitive. Similarly, we signal task execution termination through a call to `NOTIFY_END`. This same solution is adopted when an undeferred task is explicitly generated by the user through the `if(FALSE)` clause. In all the other cases, a call to `CREATE_AND_PUSH` will result in regular creation of a team descriptor and insertion in the queue (`CREATE_TASK`). After that, the producer thread signals the presence of work in the queue by releasing a barrier lock on which consumer threads wait.

This brings us to the second TSP. As explained before, threads not executing the `single` block are trapped on the barrier implied at the end of the region. This is implemented through a call to the `POP_AND_EXEC`

function (central part of Figure 2.2 Here, threads first check for the presence of tasks in the queue. If there are tasks available the encountering thread initiates an execution sequence. First, the task descriptor is extracted from the queue with the `TRYFETCH_TASK` primitive. Then, the associated task code is executed. Finally, notification of task completion is signaled through the `NOTIFY_END` primitive. If the queue is empty, the encountering thread busy waits on the *barrier lock* (note that this lock is initialized as *busy* at system startup). When the lock is released by a producer pushing a task in the queue, the current thread checks for the presence of tasks in the queue and for the number of threads waiting on the lock (annotated in a counter). If all threads are on the lock and there are no tasks in the queue, this indicates that the end of the parallel region has been reached. Otherwise, there may still be work left to do, so the thread jumps back to the scheduling loop. Note that upon task termination we execute again an iteration of the scheduling loop, thus implementing the third TSP.

Finally, a TSP is also implied at a `taskwait` construct. However, in this specific case the specification only allows to switch execution to a task that was directly created by the current one to prevent deadlocks. We implement this semantics in the `WAIT` runtime function. Each task keeps track of its children. The `HAVE_CHILDREN` primitive allows to fetch the descriptor of a child task in the *waiting state*. If a valid task descriptor is returned, the thread can be rescheduled on that task. Otherwise, all the children are in the *running* state and the thread will have to stay idle waiting for their completion. In this case, the last terminating child notifies the parent through the `NOTIFY_END` primitive.

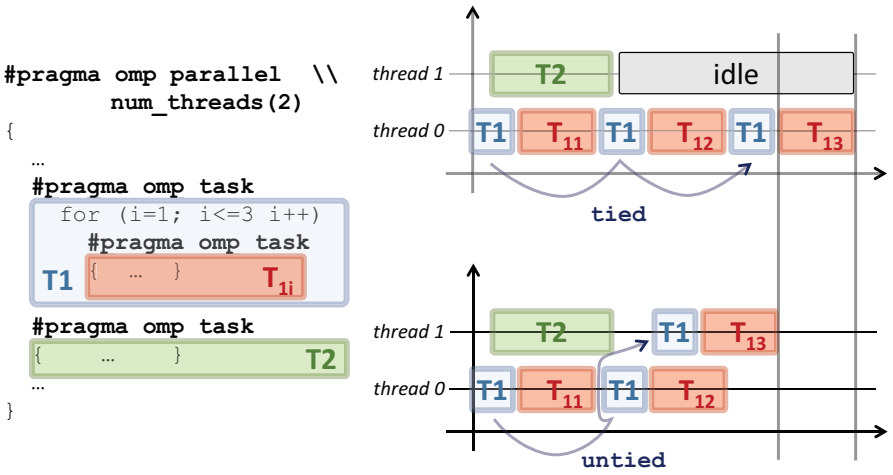


Figure 2.3: Example of tied and untied task scheduling.

4 Task schedulers

The two most widespread scheduling approaches for task-based programming models are *breadth-first scheduling* (BFS) and *work-first scheduling* (WFS). Upon encountering a task creation point: i) BFS will push the new task in a queue and continue execution of the parent task; ii) WFS will suspend the parent task and start execution of the new task. BFS tends to be more demanding in terms of memory, as it creates all tasks before starting their execution (and thus all tasks coexist simultaneously). This is an undesirable property – in general – and in particular for the resource-constrained systems that we target in this work, which makes WFS a better candidate. WFS also has the nice property of following the execution path of the original sequential program, which tends to result in better data locality [46]. However, since *tied* tasks are the default in OpenMP, RTE implementations typically use BFS.

Figure 2.3 shows the behavior of WFS if used in combination with *tied* and *untied* tasks. If all the tasks are generated from a parent task T_0,

untied tasks will be distributed among threads in a balanced manner thanks to the capability of the system to resume a suspended task on a different thread. If *tied* tasks are used, at each creation point the parent task will be suspended and the hosting thread will be rescheduled to execute the child task. The suspended parent, however, cannot be resumed on a different thread, which will lead to a sequential execution.

5 Untied tasks

The key extension required to support *untied* tasks is the capability of allowing to resume a *suspended* task on a different thread than the one that started and *suspended* it. To achieve this goal we rely on lightweight co-routines [47]. Co-routines rely on cooperative tasks which publicly expose their code and memory state (register file, stack), so that different threads can take control of the execution after restoring the memory state. Every time that a thread suspends or resumes a suspended cooperative task a context switch is performed. We place the required metadata to support task contexts (TC) in the shared TCDM, which ensures fast context switch (any thread can access the shared stacks with the same latency of just 1 cycle) and we use inline assembly to minimize the cost of the routines to save and restore architectural state.

Figure 2.4 shows how task suspension works in our approach for *untied* tasks (WFS is assumed). Initially the thread on which the code shown in figure is executing uses its own private stack (in gray). When the outermost task region (T_0) is encountered the context of the current task is saved in the TC (including the current SP), then the thread is rescheduled to executing the new task T_0. The SP of the thread is

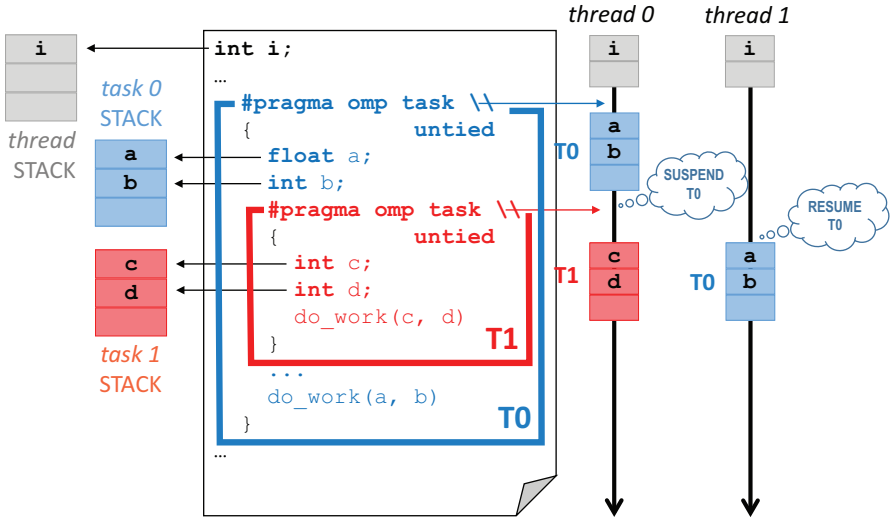


Figure 2.4: *untied* task suspension with *task contexts* and per-task stacks.

updated to the stack of T_0 (in blue) and the new task is started. When the creation point of the innermost task T_1 is reached an identical procedure is followed. The context of T_0 is saved in its TC, which is *push*ed back in the queue, then thread 0 is pointed to the stack of T_1 (in red). Now the suspended T_0 can be *pull*ed out of and restarted by thread 1. On top of this basic mechanism, a number of other design choices were made to minimize the cost of our runtime support.

Task hierarchy Supporting nested tasks requires to keep in the runtime a *tree* data structure that represents the task hierarchy. A parent task has a link to its children and vice versa, to facilitate exchange of information about execution status. For example, a parent task needs to be informed about execution completion of its children to support `taskwait`. When a parent task completes execution its children become orphans, and should not care to inform the parent. The fastest solution to handle parent task termination in terms of bookkeeping would be not

to delete the descriptor, but just to maintain the task in a *zombie* status until all children have completed. This operation would require a simple update to the descriptor, which can be executed in very short time. However, this solution brings to a memory occupation that is not acceptable for our constrained platform. Thus, we opt for a costlier removal of the descriptor from the *tree*. As a consequence, all child tasks must receive an update from the parent to avoid dangling pointers to a deallocated descriptor.

Taskwait construct Task level synchronization is widely used in recursive-based parallel patters. Here typically a fixed number of tasks is created at every recursion level, and their execution is synchronized with a `taskwait` directive. When a parent task encounters a `taskwait` it should wait until all the children (first-level descendants) have completed, but typically for performance the thread hosting the parent task is allowed to switch to executing one of the children tasks. In the baseline implementation this feature is implemented by just traversing the list of children tasks in the *tree* data structure, and inspecting their status to verify that it is set to *WAITING*.

We changed this mechanism to rely on two queues per task, to directly reference children in the *WAITING* and *RUNNING* states, respectively. Upon creation, a task is inserted in the *WAITING* queue. Every time that a task starts to execute, the runtime moves this task from the *WAITING* queue to the *RUNNING* queue, and vice versa in case of suspension.

Decoupling waiting and running tasks requires a costlier bookkeeping upon task insertion and extraction, but allows faster support for `taskwait`, as it is no longer required to search the tree for *WAITING*

tasks. In the baseline implementation this benefit was not evident, as the `taskwait` is virtually useless for flat parallel patterns. On the contrary, in recursive parallel patterns it is extensively used, and this design choice pays off.

Task dependencies The runtime design rely on a centralized queue where all tasks in the *WAITING* state are ready for extraction and execution. Suspended tasks are also pushed back in this queue. We found that in presence of recursive parallel patterns it is important to distinguish between suspended tasks that could be resumed at any time, and tasks that are suspended due to a scheduling constraint that needs to be unblocked. A typical example is, again, tasks suspended upon a `taskwait` (or due to a data dependence). As already mentioned, recursive parallelism extensively relies on such form of synchronization, thus hosting this type of suspended tasks in the central queue used to lead to a situation where we would repeatedly *pop* from there a task just to realize that the scheduling constraint was still unsatisfied. We would then have to *push* back the task in the queue and retry. Checking the status of the task before extracting it does not entirely solve the problem, as it requires time-consuming search operations. To deal with this problem we changed the implementation so as to not re-insert in the queue suspended tasks with a unresolved dependence. Such tasks are kept floating instead, and it is up to the task that will eventually resolve the dependence to *push* them back into the queue. This modification requires some additional checks to deal with the above mentioned case, but greatly improves the performance of recursive parallel programs.

Allocation of runtime metadata To minimize the overhead for dynamic resource allocation (memory, locks, task descriptors, ..) we have extensively used pools of pre-allocated resources. This is significantly faster than `malloc`-like primitives and does not require lock-protected operations, as we adopt thread-private resources. The downside of this approach is memory occupation. Since our architectural target relies on a shared cluster memory with limited size, we have to wisely use the available space. A reasonable design solution would be to dedicate roughly 5-10% of this memory to hosting tasking support data structures. If we consider the Kalray MPPA-256 platform used for experiments, the basic task descriptor has a size of 174 bytes, while the extension to support untied tasks require another 98 bytes for the contexts, plus the stacks. Private thread stacks are configured to be 1 KB (a common choice for embedded systems), while task stacks are by default 1/4 of that size. Clearly all those values are parameters in our design, and can be changed depending on specific application requirements.

Cutoff mechanisms With 10% of the cluster's shared memory allocated to task descriptors, the runtime can host simultaneously 750 pre-allocated tied tasks or 400 untied tasks. If the queue of available task descriptors is depleted during the program execution, a mechanism (known in literature as *cutoff* [48]) is triggered. When this condition is met, the creation of new task descriptors must be suspended to avoid that runtime resources saturate when task production rate is greater than execution rate. Our runtime supports two different cutoff variants: *yield* and *work-first*. In the first case, the producer task is stopped and pushed at the end of the READY queue, with the aim to re-schedule the core to executing

pending tasks instead of generating new ones. Using the second variant, the producer task starts working in work-first mode by executing the new tasks in-place via a standard function call: in this case task descriptors are not required, as the synchronization is enforced by serializing tasks on the same thread. Cutoff mechanisms are introduced to avoid an unbounded consumption of runtime resources, but recursive applications can cause additional problems. Using untied tasks, task stacks typically end up to be oversized to fit the worst case (i.e., the maximum recursion level reached in cutoff state) to the detriment of runtime memory footprint. To avoid this case we introduced a specific optimization for untied tasks using work-first policy, which forces the producer task to swap its current stack with a special one that is the only one sized for worst-case recursive execution.

6 Experimental results

As already pointed out, supporting the untied execution model is usually subject to large overheads. While such overheads can be tolerated by large applications exploiting coarse-grained tasks, this is usually not the case for embedded applications, which rely on fine-grained workloads. To study this effect, our plots show speedup (parallel execution on 16 cores versus sequential execution) on the y axis, comparing the original Kalray runtime (based on a standard OpenMP implementation) to tied and untied tasks in our runtime (the ”OPT tied” and ”OPT untied” series). For all the experiments except the one in Section 6.5 we use a set of microbenchmarks in which tasks only consist of ALU operations (e.g.,

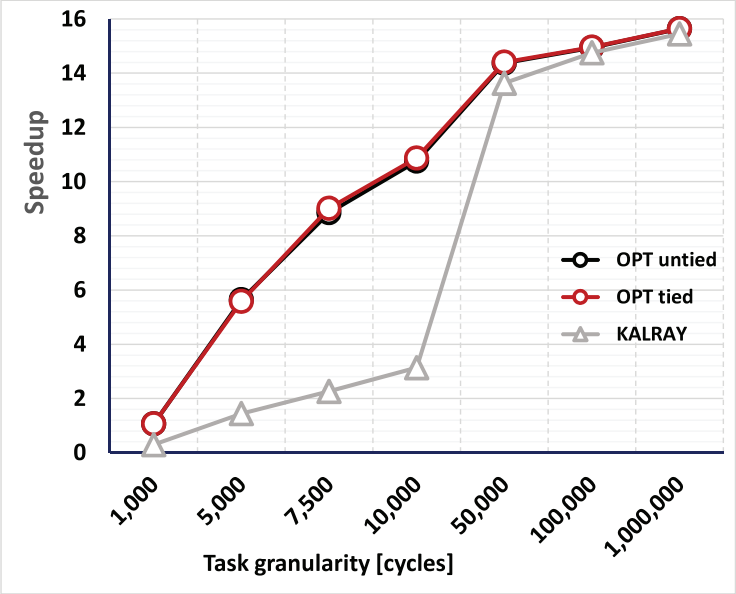


Figure 2.5: Speedup of the LINEAR benchmark (no cutoff).

add on local registers) and no load/store operations, which allows to explore the maximum achievable speedups. The number of ALU operations within the tasks can be controlled via a parameter, which allows to study the achievable speedup for various task granularities, which we report on the x axis of each plot (task granularity is expressed in duration in clock cycles, roughly equivalent to the number of ALU operations that each task contains).

6.1 Applications with a linear generation pattern

Figure 6.1 shows results for the LINEAR microbenchmark, a simple loop from which 512 tasks are created (one per loop iteration). Focusing on the results for the Kalray SDK (”KALRAY” line), ideal speedups can be achieved only for tasks larger than 100 KCycles. For smaller tasks the maximum achievable speedup is 3×. In this fine-grain task area,

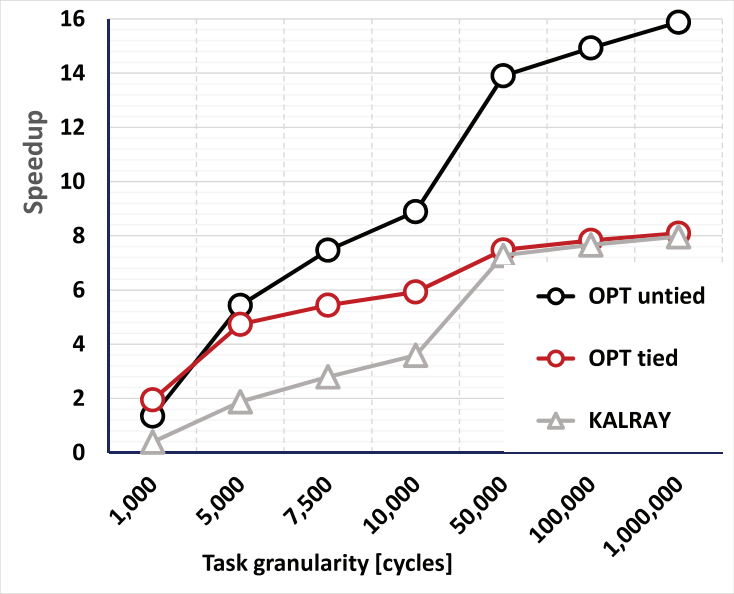


Figure 2.6: Speedup of the RECURSIVE benchmark (no cutoff).

OPT tasks can consistently achieve 4× higher speedup. Since in the LINEAR microbenchmarks there is no task nesting, there is no significant difference between tied ("OPT tied" line) and untied ("OPT untied" line) tasks. We thus explore a new configuration where tasks are recursively created to appreciate the difference.

6.2 Applications with a recursive generation pattern

Figure 6.2 shows the efficiency of our runtime for the recursive parallel pattern, considering tied and untied tasks. The RECURSIVE microbenchmark builds a binary tree of depth N = 9 (512 tasks) recursively. This is similar to a classical Fibonacci algorithm, where each of the two recursive calls is enclosed in a `task` directive. A `taskwait` directive is

placed after the creation of the two tasks. The first result that we observe is that only untied tasks can achieve the maximum speedup. Tied tasks have a maximum speedup of 8. This effect is due to the behavior of `taskwait` in presence of tied tasks. If a tied task is stuck on a `taskwait` and there are no children tasks in the WAITING state (e.g., few tasks generated at each recursion level, like in the binary tree), that task is bound to wait until the children have finished. Using a binary tree, this leads to exactly half of the threads getting stuck, which explains the maximum speedup observed in this configuration. This problem is circumvented by untied tasks, which can reschedule the threads hosting the stuck tasks to other ready tasks. Similar considerations to what we discussed in the previous section hold for the comparison between KALRAY tasks and OPT tied tasks (Kalray implementation only supports tied tasks, so a comparison to untied is not directly feasible).

In general, it is possible to see that RECURSIVE implies much higher overhead than LINEAR. This is justified by a significantly increased contention for shared data structures (queues, trees, etc.), as in this pattern multiple threads are concurrently creating tasks. Even if we have struggled to make the lock-protected operations to operate on shared data structures as short as possible, their serialization over multiple requestor is evident. As a result, it takes an order of magnitude coarser tasks (around 100K) than in the LINEAR case to achieve nearly-ideal speedups. This is a typical situation where cutoff policies can help in significantly reducing the runtime overheads. We explore the adoption of cutoff policies in Section 6.4.

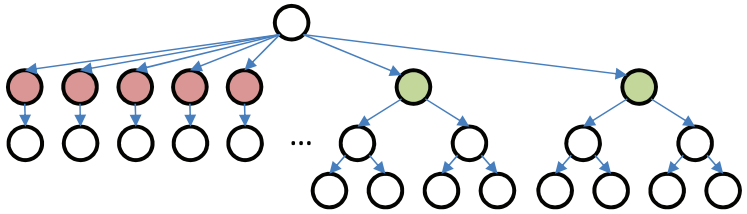


Figure 2.7: Structure of the MIXED benchmark.

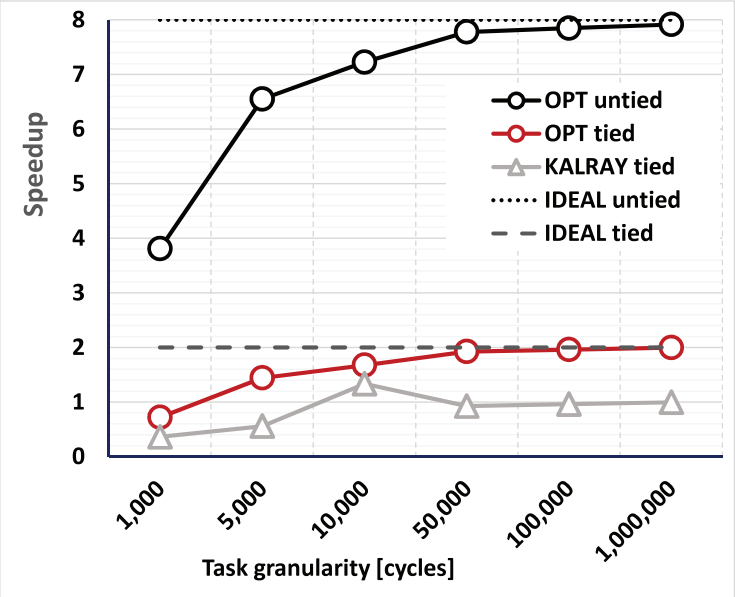


Figure 2.8: Speedup of the MIXED benchmark.

6.3 Applications with mixed patterns

The advantage of using untied task is particularly evident for applications presenting a mixed structure which includes both LINEAR and RECURSIVE task creation patterns. The MIXED microbenchmark depicted in Figure 2.7 is aimed at studying the behavior of such applications. A root task generates 7 tasks in a LINEAR manner, each one spawning a single child with a long execution time and then performing a taskwait, and another two tasks from within RECURSIVE binary trees of depth 5. Figure 2.8 shows the results for this benchmark. Using tied tasks, 14

threads are allocated to execute the linear part of the application, 7 of which are blocked by the `taskwait` directive. The ideal speedup of the application is 2×, which OPT tied tasks reach for granularities of around 10000 cycles. Using untied tasks only 7 threads are allocated to the LINEAR part, which brings the ideal speedup to 9×. The maximum speedup achieved by OPT untied tasks is 8× due to a limitation of the tracing (performance monitoring) of the Kalray platform. The root task of the hierarchy is the one performing time measurement and we were forced to declare this as a tied task to gather coherent clock values (allowing this task to migrate to other cores results in incoherent measurement). This limits the maximum achievable speedup to 8×, which OPT untied tasks achieve for granularities above 10000 cycles. Overall, untied tasks enable 4× faster execution than tied tasks for application featuring mixed task creation patterns.

Note that this result holds for any runtime implementation. Our solution makes this result visible for smaller tasks compared to other OpenMP tasking implementations. The Kalray implementation never surpasses a speedup of 1× in the considered range of task granularities (up to one million cycles) for this experiment.

6.4 Impact of cutoff on linear and recursive applications

We repeated the experiments with LINEAR and RECURSIVE microbenchmarks considering a higher number of tasks (2048). This configuration saturates the runtime data structures and activates cutoff mode. Figure 2.9 and Figure 2.10 show the results for this experiment.

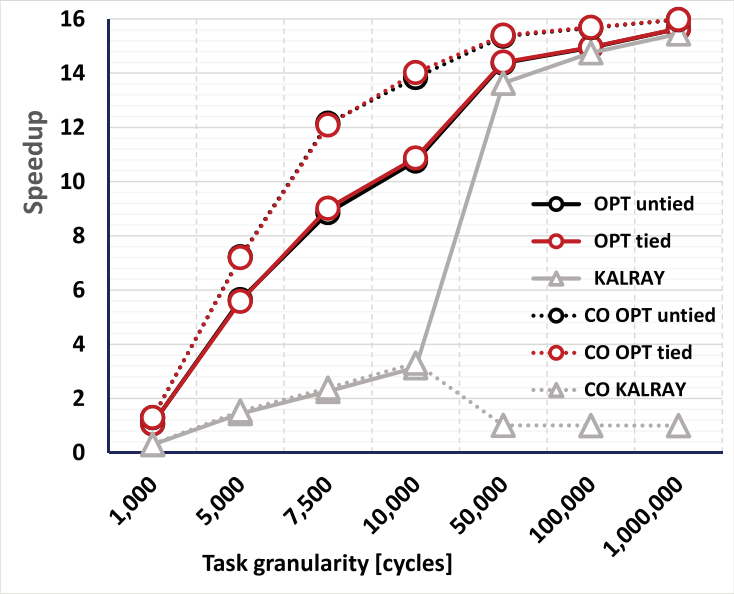


Figure 2.9: Speedup of the LINEAR benchmark (with cutoff).

Focusing on the LINEAR pattern, the adoption of cutoff greatly mitigates overhead effects, and we can achieve nearly-ideal speedups for an order of magnitude smaller tasks compared to Kalray tasks. It also has to be noted that cutoff mode is not properly supported for LINEAR patterns in the original Kalray runtime. Enabling cutoff mode in this configuration simply seems to disable parallelism completely.

Focusing on the RECURSIVE pattern the use of cutoff policies proves extremely beneficial, with nearly-ideal speedups for very fine-grained tasks (in the order of thousand cycles).

6.5 Real applications

To assess the performance of out runtime on real applications, we execute the benchmarks from the Barcelona OpenMP Task Suite (BOTS), which includes a wide set of real-life applications parallelized with OpenMP

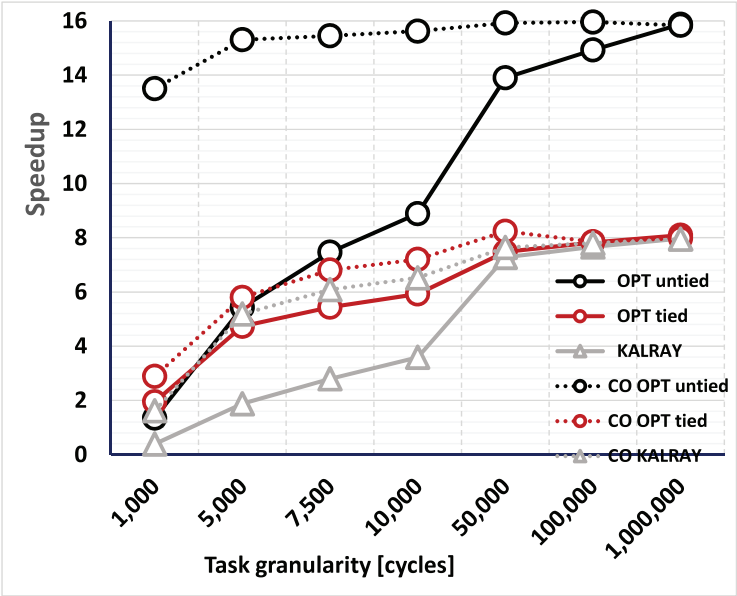


Figure 2.10: Speedup of the RECURSIVE benchmark (with cutoff).

tasks. Figure 2.11 shows the speedup of applications for different configurations, comparing the Kalray SDK (KALRAY) with different configurations of our runtime, using tied tasks (OPT tied), untied tasks (OPT untied) and untied tasks with cutoff (OPT untied CO). On average, our runtime has a 12× speedup compared to 8× for Kalray SDK. The benefits of cutoff are minimal, since the bottleneck is limited parallelism in the application rather than runtime overhead. The marginal improvements, where present, are usually due to better memory usage (tasks in cutoff use less memory for the runtime, which is used for application data instead).

6.6 Comparison with other tasking models

For completeness, we compare our implementation to other representative commercial and academic ones, targeted at general purpose and

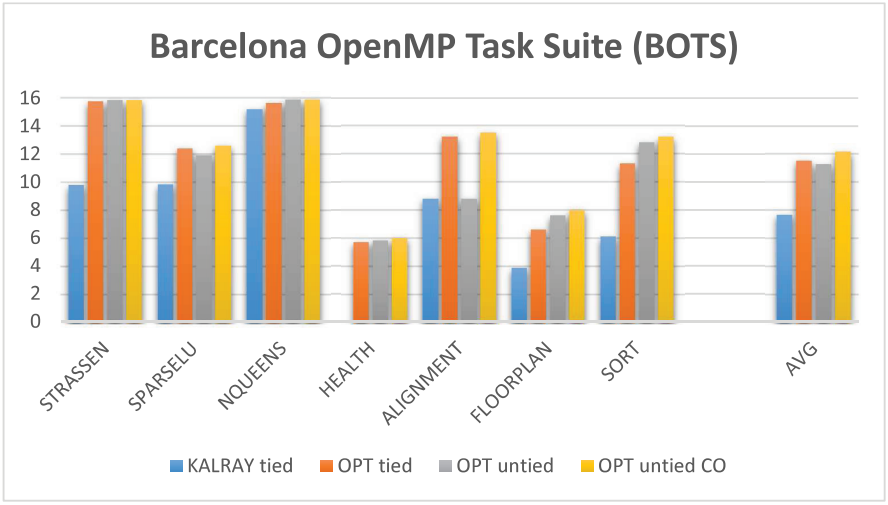


Figure 2.11: Speedup of BOTS.

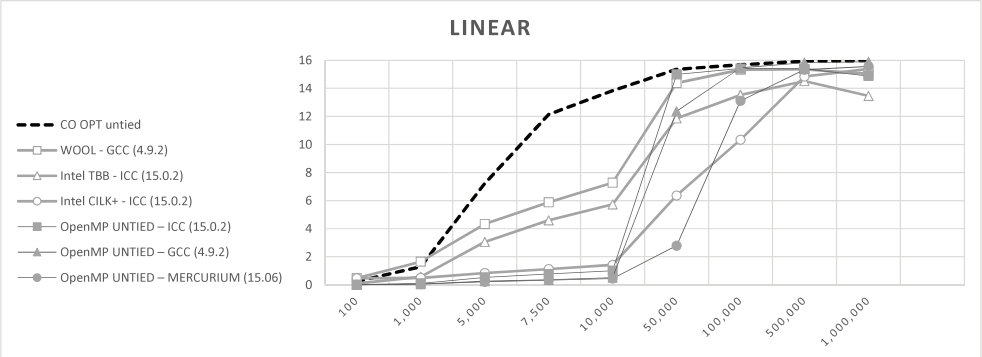


Figure 2.12: Comparison to other tasking models (LINEAR).

high performance computing systems: the GNU OpenMP implementation (GCC 4.9.2); the Intel OpenMP implementation (ICC 15.0.2); the OpenMP runtime of the Barcelona Supercomputing Center (Mercurium 15.06 running on Nanos++); the CILK+ programming model (ICC 15.0.2; the WOOL tasking model (GCC 4.9.2). The LINEAR and RECURSIVE microbenchmarks have been used for this experiment, considering untied tasks and a BFS policy. As a target platform for these experiments we used a compute server equipped with two Intel Haswell with 8 cores @ 2.40 GHz. Figure 2.12 and Figure 2.13 show that our

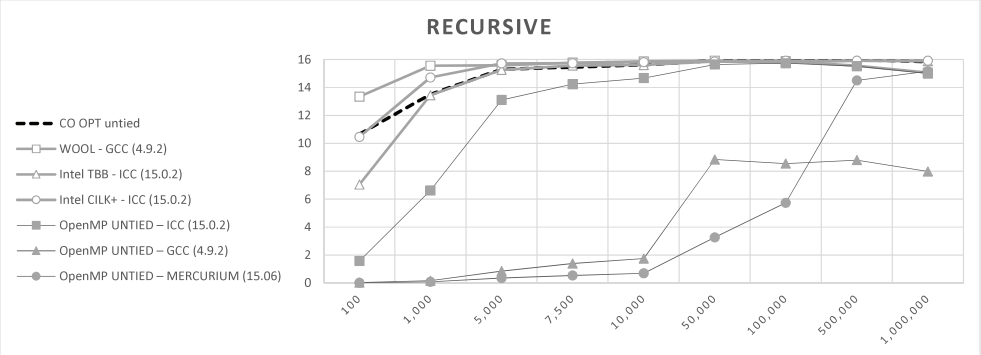


Figure 2.13: Comparison to other tasking models (RECURSIVE).

implementation allows to achieve near-ideal speedups for one order of magnitude smaller tasks compared to the others for the LINEAR case, and outperforms other OpenMP implementations for the RECURSIVE case.

7 Conclusion

Task-based parallelism has the potential to provide efficient exploitation of many-core accelerators, offering flexible support to the fine-grained and irregular parallelism found in embedded applications. In this chapter we have presented an optimized implementation of the OpenMP tasking model for embedded parallel accelerators. The proposed design enables support for *untied* tasks and recursive parallel patterns for the targeted class of computing systems. When compared to OpenMP implementation for embedded computing systems, our design achieves near-ideal speedups for one order of magnitude smaller tasks.

Chapter 3

Extending the OpenCL model for efficient execution of graph-based workloads on many-core accelerators

1 Introduction

As introduced in Section I.II of this thesis, the OpenCL standard introduces platform and execution models which are particularly suitable for heterogeneous platforms. Data transfers and memory management are major challenges in programming a heterogeneous platform, and OpenCL defines a unified memory abstraction model and provides a standard API to handle data transfers. Nevertheless, a common issue of using OpenCL on embedded systems is related to the mandatory use of global memory space to share intermediate data between kernels. When increasing the number of interacting kernels, the main memory bandwidth

required to fulfill data requests originated by PEs is much higher than the available one, causing a bottleneck. In addition, unlike accelerators for desktop computing environments (e.g. Intel Many Integrated Core architecture [49]), SoCs have unified host and global memory spaces, and have a common data path connecting host processor and accelerator with L3 memory [50]. As a direct consequence, applications experience high contention for off-chip memory access, that may severely limit the final speed-up. For instance, we consider a platform with an accelerator and a DDR3-1600 memory (6400 MB/s per channel). If we reasonably assume that the accelerator has half of the available bandwidth (3200 MB/s), and our application need to process a 1920×1080 video source at 60 fps, then a single image access uses 123.12 MB/s. Hence, after accessing 26 image buffers in one frame time the available bandwidth is saturated, but this could not be enough for a complex application that instantiates many kernels and requires many intermediate results.

Overall DMA is difficult to use, but it can be managed quite easily with additional assumptions on data layout, for instance considering images and limiting the DMA to perform single frame copies. However this approach cannot be generalized, because (i) there is not enough internal memory even to keep a single image, and (ii) the memory bandwidth is easily saturated in presence of multiple kernels. DMA can be used for *tiled transfers*, using the *async_work_group_copy* function to handle asynchronous copies between global and local memory and vice versa. In this case programmers need to interleave DMA and computation and this gets much more complicated. For basic kernels, the lines of code used for DMA orchestration and subsequent workload distribution are over 50% of the total [51]. Considering applications composed by multiple kernels,

this normally implies multiple tile sizes, because each kernel may require a different tile. In this case, the process to manage this orchestration by hand becomes totally unmanageable, and programmers need tools. This is exactly what our approach provides. Other programming paradigms have been proposed to implement image processing applications on embedded systems, based on data-flow graphs [52] [53] or functional models [54]. Overall, these solutions tackle the issue of memory bandwidth using a tiling-based approach, but in most cases the related execution models are not suitable to build complex applications which include irregular algorithms and data access patterns.

In this chapter we introduce a framework that implements a set of optimizations specifically targeted to accelerate the execution of graph-based image processing applications on many-core accelerators. The evolution of imaging sensors and the growing requirements of applications are pushing hardware platform developers to incorporate advanced image processing capabilities into a wide range of embedded systems, ranging from smartphones to wearable devices. In particular, we focus our attention on three main classes of computationally intensive image processing tasks: Embedded Computer Vision (ECV) [55], brain-inspired visual processing [56], and computational photography [57]. Considering the actual market trend toward HD formats and real-time video analysis, these algorithms require hardware acceleration. In parallel embedded accelerators, the cores are simpler w.r.t. common multi-core architectures and offer a good trade-off between highly parallel computation and power consumption, so they are a promising target for running image processing workloads. The framework front-end is based on the OpenVX standard [23]. OpenVX is a cross-platform API which aims at enabling

hardware vendors to implement and optimize low-level image processing primitives, with a strong focus on mobile and embedded systems. In our framework, data accesses are performed on local buffers in the L1 scratchpad memory of the reference architecture, that is what we call a *localized execution*. To satisfy this condition, we have taken into account all the data access patterns that can be found in the OpenVX standard-defined kernels (e.g., local and statistical operators), and that are the most common in image processing algorithms, and then we have defined a set of techniques to support automatic image *tiling*. Coupling tiling with double-buffering, we achieve a good overlap between data communication and kernel execution on the accelerator, that guarantees a higher efficiency in terms of PEs usage.

The novelty of this approach derives from three main contributions: (i) the introduction of a low-level OpenCL extension, that can be effectively used to support efficient execution of graph-structured workloads with explicit DMA transfers; (ii) the automatic mapping of an OpenVX program to a list of host kernels and OpenCL low-level graphs; (iii) an algorithm for computing optimal tile sizes for the kernels, taking into account on-chip memory size limitations, while minimizing main memory utilization. Our framework supports applications of any complexity level, with no limitations related to data access patterns. An OpenCL low-level graph is used on the accelerator side, with the aim to achieve better timing performance yet minimizing the required bandwidth. The use of host kernels is limited to irregular access patterns, and the overall orchestration is provided by the framework without any hint by the programmer.

Context	Container for all object instances
Kernel	Vision kernel implementation, used to create nodes
Graph	DAG of nodes implicitly connected by data usage
Node	Instance of a kernel inside a specific graphs
Parameter	Reference to a data object used as node parameter

Table 3.1: OpenVX framework objects.

Scalar	Scalar type (integer, floating point, enum)
Array	Array of scalar or structured types
Image	Image (including one or ore data planes)
Matrix	MxN matrix
Convolution	MxN matrix with an associated scaling factor
Distribution	1D or 2D histogram
Pyramid	Set of images with a fixed scale ratio
LUT	Lookup table
Remap	Map of source points to destination points
Threshold	Set of thresholding values
Delay	Time-delayed set of images or arrays

Table 3.2: OpenVX data objects.

2 OpenVX programming model

OpenVX [23] is a cross-platform C-based API which aims at enabling hardware vendors to implement and optimize low-level image processing and CV primitives. The final OpenVX 1.0 framework specification is available as an open, royalty-free standard ratified by the Khronos Group. Most image processing applications can be easily structured as a set of vision kernels (i.e. basic features or algorithms) that interact on the basis of input/output data dependencies. Considering this usage scenario, OpenVX promotes a graph-oriented execution model, based on Directed Acyclic Graphs (DAGs) of kernel instances.

The standard introduces the software abstractions that are mandatory for an OpenVX execution environment, defining *framework objects* (Table 3.1) and *data objects* (Table 3.2). Framework objects are model entities, while data objects are input/output parameters which are processed at execution time. The first step of an OpenVX program is the creation of a valid context by calling `vxCreateContext`, followed by the declaration of the data objects which are required as node parameters. Data objects are created via `vxCreate<Object>` or retrieved via `vxGet<Object>`. To enforce consistency, access to data objects is regulated by an acquire/release protocol, via `vxAccess<Object>` and `vxCommit<Object>` functions. Data objects exist at the context level, they have transparent reference counts and are not destroyed until their reference count is zero. Ina any case, data objects are forcibly destroyed at context destruction.

A key feature is the possibility to declare *virtual data objects*. These objects are not guaranteed to reside in main memory on a permanent basis, and they may have null size and undefined format (`VX_DF_IMAGE_VIRT`) at declaration time. Basically, virtual data are used to set a dependency between adjacent kernel nodes, and are not associated with any memory area accessible by read/write operations.

OpenVX graphs are composed of one or more nodes that are added by calling node creation functions (in the form `vxCreate<Kernel>Node`). Nodes are linked together via data dependencies, without specifying any explicit ordering. The OpenVX standard defines a library of *predefined vision kernels* which can be used to create nodes, but it also supports the definition of *user defined kernels*. The standard defines 41 predefined kernels, which are fully supported in our implementation.

Graphs must be verified calling `vxVerifyGraph` before execution, with the aim to guarantee some mandatory properties:

- Input and output requirements must be compliant to the node interface (data direction, data type, required vs optional flag).
- No cycles are allowed in the graph.
- Only a single writer node to any data object is allowed.
- Writes have higher priorities than reads.

During the verification stage, a *validator callback* is called for each node parameter to verify the above-mentioned properties. This is a function defined at kernel level, and it is also responsible to set dimension and format of virtual data with the aim to respect all functional constraints.

Graphs can be processed as many times as needed after their verification. Changes are possible but require a further verification. A graph can be processed in two modes: (i) *synchronous blocking mode*, which blocks the program execution until the graph processing is completed; (ii) *asynchronous single-issue mode*, which is non blocking and enables the parallel execution of multiple graphs.

To introduce OpenVX programming, we consider a basic edge detector. The OpenVX code for this application is shown in Listing 3.1. In this example, all the referenced kernels are contained in the OpenVX standard. Note that an OpenVX program is much more abstract and concise than an OpenCL program, and at the same time the underlying OpenCL run-time used in our approach is completely hidden to the programmer. Moreover, we require no additional information from the programmer to guide the accelerator tuning.

```
1 vx_context ctx = vxCreateContext();
2 vx_graph graph = vxCreateGraph();
3 vx_image imgs[] = {
4   vxCreateImage(ctx, width, height, VX_DF_IMAGE_RGB),
5   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8),
6   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
7   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
8   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
9   vxCreateImage(ctx, width, height, VX_DF_IMAGE_U8),
10 };
11 vx_node nodes[] = {
12   vxColorConvertNode(graph, imgs[0], imgs[1]),
13   vxSobel3x3Node(graph, imgs[1], imgs[2], imgs[3]),
14   vxMagnitudeNode(graph, imgs[2], imgs[3], imgs[4]),
15   vxThresholdNode(graph, imgs[4], thresh, imgs[5]),
16 };
17 status = vxVerifyGraph(graph);
18 while (/* input images? */) {
19   /* capture data into imgs[0] */
20   status = vxProcessGraph(graph);
21   /* use data from imgs[5] */
22 }
23 vxReleaseContext(ctx);
```

Listing 3.1: Edge detector (C code with OpenVX API calls).

The program follows these steps:

- A context is initially created (line 1) and then released at the end (line 23).
- A graph is created (line 2).
- Images are defined (lines 4-9), some of them as *virtual* (lines 5-8).
- A set of nodes is created and added to the graph as instances of vision kernels (lines 12-15).

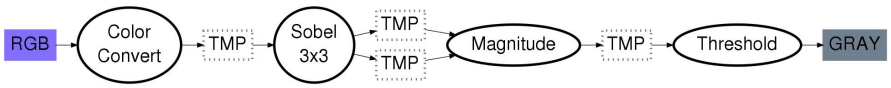


Figure 3.1: Edge detector (DAG).

- The `vxVerifyGraph` function (line 17) checks the graph consistency and propagates constraints on virtual images; for instance, the format of the virtual images defined in lines 6-7 is set to `VX_DF_IMAGE_S16`, as required by Sobel kernel validators after a `VX_DF_IMAGE_U8` input image.

- The `vxProcessGraph` function (line 19) executes the graph in synchronous blocking mode inside a loop, that is a typical programming pattern to process an incoming stream of input images.

Figure 3.1 shows the DAG derived by this program, outlining input and output images of the program.

3 Extended OpenCL run-time

In most cases (see [16], [58], [59] and [60]) the programming environments for many-core accelerators support OpenCL 1.1 [61]. Figure 3.2 shows the comparison between the OpenCL logical model and the STHORM architecture.

In this scenario, data must be transferred into shared local memory prior to computation in order to take advantage of low-latency accesses, and it has to be done explicitly using OpenCL built-in functions for asynchronous work-group copy. STHORM cores are hardware mono-threaded, hence the best performance is achieved by exactly matching any OpenCL ND-Range with the STHORM architectural parameters to

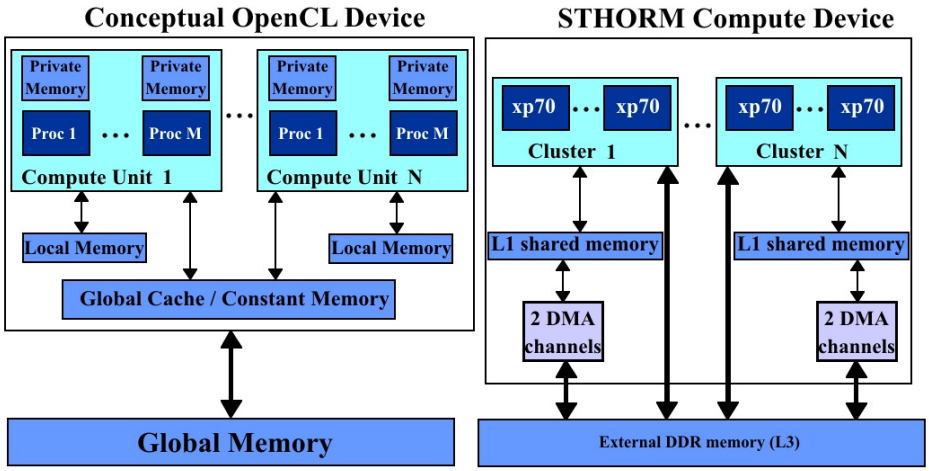


Figure 3.2: OpenCL mapping for STHORM (source: STMicroelectronics).

avoid expensive context switches: that is, programmers should use as many work-groups as the number of clusters, and as many work-items as the number of processing elements in a cluster.

We introduce an *extended OpenCL run-time* (referred as CLE), which enables the creation of low-level graphs containing nodes of different types:

- `CreateBuffer` – a node that allocates a buffer at the specified memory level (L1, L3) associated with a numerical identifier.
- `CopyBuffer` – a node that enqueues a DMA transfer to copy data from a source buffer to a destination buffer, specifying their numerical identifiers.
- `ExecKernel` – a node that enqueues the execution of an OpenCL kernel.
- `ReleaseBuffer` – a node that releases the specified buffer.
- `EndGraph` – a node that triggers the end of the graph execution.

For each node, the programmer has to specify the actual kernel parameters and a set of dependencies: if there is a dependency edge between node A and node B, node A must terminate its execution before node B can start. OpenCL kernels intended for CLE graphs directly access buffer parameters (identified by `global` access modifier in the kernel source code) without managing data transfers and local buffers in the kernel body.

The memory management is totally explicit, including the allocation of the stack area used by cores to execute. The `cleGraphSetBuffer` primitive associates a standard OpenCL buffer to a graph using a numerical identifier, which can be used by CLE functions to address input/output data provided by the host using L3 memory space. Using `clEnqueueGraph`, a CLE graph is pushed in OpenCL command queues for execution. Each graph is executed on a single cluster, while other clusters can serve different requests (different graphs or standard OpenCL tasks). When the `EndGraph` node terminates, a notification is sent back to the host side, and the concerned cluster is made available. An example of a CLE graph that includes a single kernel is depicted in Figure 3.3. In real applications, images do not fit entirely in L1 memory. In the context of our framework, CLE graphs contain multiple kernel nodes, and the same set of nodes is executed multiple times on complementary data subsets. This mechanism is explained in greater detail in Section 4.6.

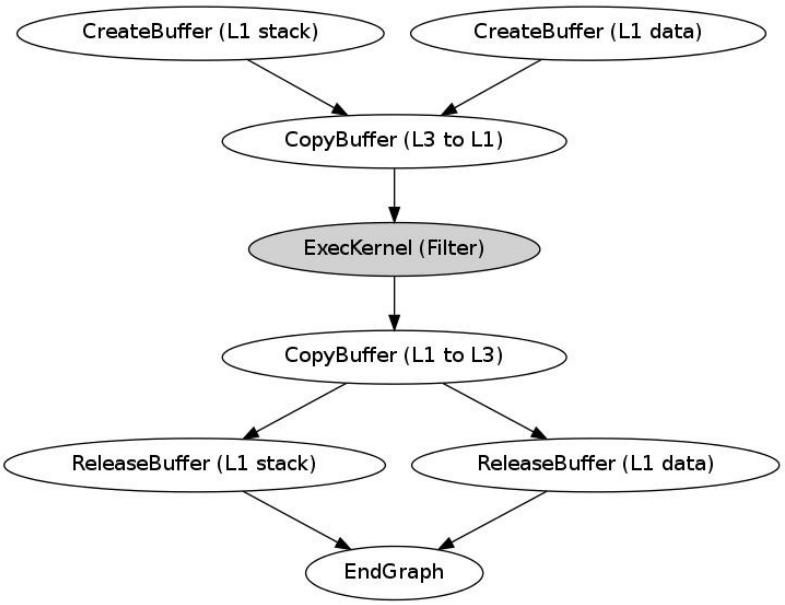


Figure 3.3: Execution of a single kernel on CLE run-time.

4 Optimization framework for many core accelerators

Our main goal is the maximization of *execution efficiency* of the accelerator, which is defined as the amount of time that PEs spend to execute kernel code over the total execution time. This goal implies to minimize the total waiting time due to memory transfers to/from L3 memory. For this purpose, virtual images are not allocated in L3 memory, but they are partly allocated in a set of L1 buffers managed by the framework. After the verification steps required by OpenVX semantics, we integrate a set of algorithms for graph partitioning and scheduling, that provide buffer allocation, buffer sizing and CLE graphs creation. The verification stage of an OpenVX graph is performed at run-time, that gives the capabilities of changing the application graph with an adaptive approach and supporting dynamic hardware resources. To limit the time of on-line

partition and scheduling phases, we have privileged the use of heuristics algorithms, but different policies can be easily plugged in.

Overall, nodes can be executed by the host processor or by the accelerator. On the host side, the kernels are implemented as plain C functions, using the OpenVX API to access data objects (see Section 3). Our implementation for host kernels is based on the reference implementation provided by Khronos. On the accelerator side, we provide a set of CLE kernels which follow the guidelines introduced in Section 3. All the boilerplate code to enable localized execution is managed by the run-time on the basis of kernel data structures and graph verification steps.

All the presented algorithms are focused on single-cluster optimization. The framework supports the execution of OpenVX programs on multi-cluster accelerators by applying two methodologies: (i) the use of different clusters to compute different input sets, and (ii) the partition of an input set into multiple parts that can be computed independently. The second approach can use the function `vxCreateImageFromROI` to create a new image object referencing a rectangular region of another image. In both cases, to take effective advantage from the multi-cluster execution OpenVX programmers must orchestrate a global execution schema using asynchronous single-issue mode, and then synchronize the execution status with the `vxVerifyGraph` API.

4.1 Data access patterns

In an OpenVX run-time targeting the host processor of an embedded platform, such as the reference implementation provided by Khronos, images normally reside in main memory. In our framework, images which represent an input/output for the graph are partitioned into smaller

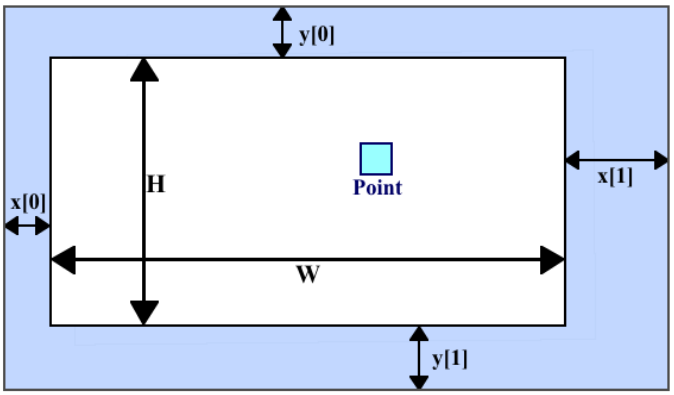


Figure 3.4: Structure of a tiling descriptor.

blocks, called *tiles*, to fit in L1 buffers. Using this approach, virtual images representing an intermediate result are not allocated to L3 memory, at least whenever it is possible without breaking other constraints. The allowed size for tiles strictly depends on the data access patterns used by kernels.

To describe these patterns, we associate a tiling descriptor to each input/output port of OpenVX kernels. For input data, this structure specifies the minimum set of points necessary to compute an output value, in terms of both *computing area* and *neighboring area*. For the sake of illustration, we consider the common case of a single point per output value, but in some cases the input set could contain more adjacent points (e.g., scaling operator), and this condition is managed by the framework. Figure 3.4 describes the structure of a tiling descriptor. W and H are the dimensions of the computing area, that is the set of points used to compute a single output value; for output tiles, these values represent the minimum number of output points generated by a single iteration. x and y values describe the neighboring area, that is the set of additional points contributing to the computation of a single output value, but that

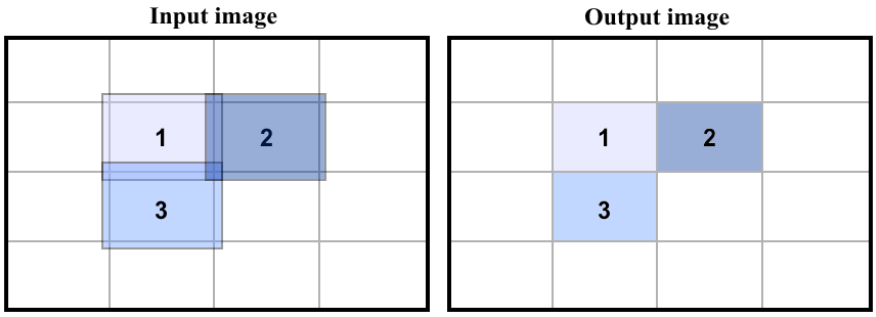


Figure 3.5: Image tiling schema for a single kernel.

may belong to other computing or neighboring areas.

The distinction between computing and neighboring area is really important, since it has a major impact on data partitioning. The partition of input images into tiles is correct when the juxtaposition of the output tile produces a complete output image, equivalent to execute the graph in one step, that is when the tiles have the same size of the images. In this context, x and y values exactly correspond to the horizontal/vertical overlap between adjacent tiles which is required to compute all the points in the output tile. Taking into account a single kernel, the size of output tiles is implied by input tiles, and output tiles do not require any overlap (see Figure 3.5).

Referring to the literature on image processing functions [62], we have identified five different classes of operators (see Figure 3.6), which cover 100% of the OpenVX standard defined kernels:

A) *Point operators* (e.g. color conversion, threshold) compute the value of each output point from the corresponding input point. These operators do not require any tile overlap by construction ($W = 1, H = 1, \forall ix[i] = 0$).

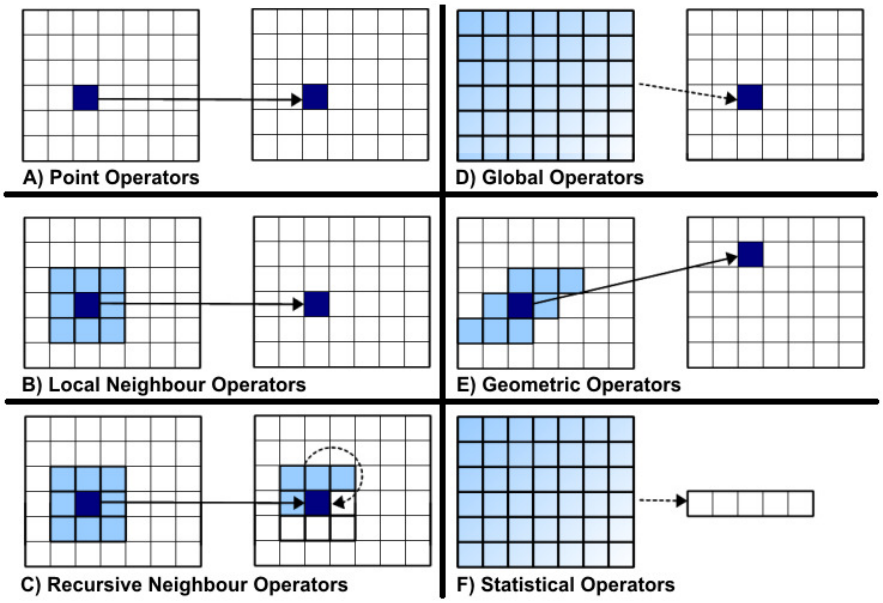


Figure 3.6: Classes of image processing kernels.

B) *Local neighbor operators* (e.g. linear operators, morphological operators) compute the value of a point in the output image that corresponds to the input tile. Local neighbor operators require a complete handling of the tile overlap, based on the parameters of the kernel neighboring area ($W = 1, H = 1, \exists ix[i] >= 1$).

C) *Recursive neighbor operators* (e.g. integral image) are similar to the previous ones, but in addition they also consider the previously computed values in the output tile. The managing of tiling is equivalent to local neighbor operators, but we also need to save state data between tiles (in common cases the borders of previous output tiles).

D) *Global operators* (e.g. DFT) compute the value of a point in the output image using the whole input image. In this case it is impossible to apply tiling to input data.

E) *Geometric operators* (e.g. affine transforms) compute the value of

a point in the output image using a non-rectangular input area. In the most general case, we cannot apply a classical input tiling due to the generic shape of the neighboring area. For some transformations we can specify a tile defining a bounding box, even if this causes an overhead in terms of data that are transferred and not used, and we can derive an equivalent local neighbor operator.

F) *Statistical operators* (e.g. mean, histogram) compute statistical functions of image points. Tiling can be activated on input images, and we can use a persistent buffer to implement a reduction pattern "walking" through the tiles.

To handle the computation of recursive neighbor operators, the framework supports the use of *state buffers*. Each kernel implementation must specify the amount of bytes needed to maintain its state across tiles, as a constant value or as a linear combination of the tile border size. After computing the final tile size (this will be explained in detail in Section 4.4), the framework allocates state buffers of the proper size and provide to the kernel function a pointer to the memory area (in case of borders, a single pointer for each border state). Henceforward, each node is responsible to handle the content of its state buffer. To satisfy their goal, state buffers are persistent for a specific node over multiple tile executions. For instance, we have used the state buffers to implement the integral image kernel. In this case the area sum of the borders is propagated to the adjacent tiles using a state buffer with a number of elements exactly equal to the border size. The first PE is responsible to update the content of the buffer corresponding to left and bottom state.

State buffers are also used to handle reduction patterns when executing statistical operators. For example, the mean kernel can use different

accumulator variables (sum of values, number of points) for each executing core, and after the last tile a single core is responsible to compute the reduction and perform a division. At the current stage of development, the framework provides a pointer to the state buffer and some flags (first tile/last tile), and the vendor which provides an accelerated kernel must implement the reduction patterns directly in the kernel code. Overall, the tiling approach is totally transparent to the OpenVX final user.

4.2 Graph partitioning

There are applications for which the resultant graph cannot be executed allocating all the intermediate tiles in L1 buffers, basically for two reasons: (i) the graph contains a kernel of classes D, E, or F, or (ii) the buffer sizing algorithm fails to fit all buffers in L1 memory (see Section 4.5). In these cases. the graph is automatically partitioned into multiple sub-graphs, each one corresponding to a CLE graph, and the intermediate images that connect different graphs are saved into L3 memory. Hence, the execution of an OpenVX graph is divided into multiple stages at run-time level, and the tiling is applied at each stage independently. This process is totally transparent to the programmer.

To support graph partitioning, a *memory boundary* is added after a kernel of class F, as it presents a cumulative output which is not compatible with a tiling approach. Kernels of classes D and E do not support any tiling scheme, consequently they are executed on the host processor and memory boundaries are forced before and after. Additional memory boundaries are added to switch the memory domain of image parameters, in particular: (i) a boundary from L3 to L1 is required in input for kernels of class A,B,C or F, (ii) a boundary from L1 to L3 is required in

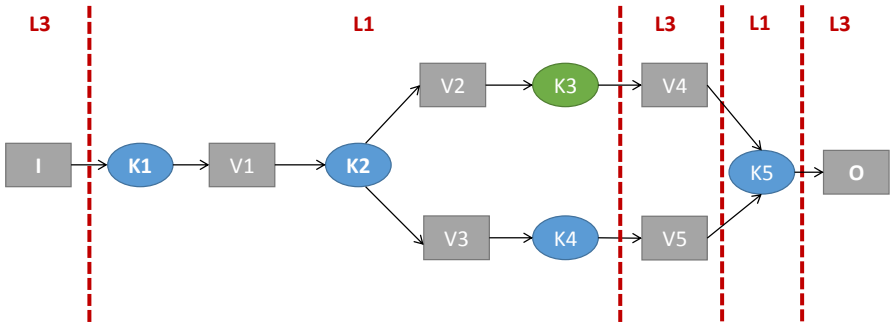


Figure 3.7: Application graph partitioning.

output for graph final results, and (iii) all input/output parameters of a node must reside in the same memory domain.

For instance, in the application graph depicted in Figure 3.7 we suppose that K3 is a statistical kernel (e.g., a histogram). Tiling cannot be used on its output image V4, because each input tile contributes to sparse data in the result set (the histogram bins). Consequently, a memory boundary is added to its output, and this includes all the images read by K5. In this example K5 is a kernel of classes A, B or C, so a memory boundary is inserted to switch back its input images to L1 domain. Finally, I and O must reside on L3 domain.

4.3 Node scheduling

For each sub-graph extracted at the partitioning stage, a node scheduling must be determined. The current version of the framework forces the processing of a graph on a single cluster, and allocates all the processing elements to a single running node. Consequently, the node scheduling algorithm selects a single node at each iteration, and its final schedule is an ordered list.

Experimental results on STHORM show that the contention for L1

```
1: while active set is not empty do
2:    – Select from the active set the kernel with more
   input dependencies;
3:    – Append the selected kernel to the schedule list;
4:    – Remove the selected kernel from the active set,
   and add it to the visited set;
5:    if active set is empty then
6:       – Compute a new active set, including all the
   nodes that are not in the visited set and are con-
   nected to graph input data or to visited nodes;
7:    end if
8: end while
```

Figure 3.8: Node scheduling algorithm.

memory is very limited when all the cores are active, due to the low-latency of the logarithmic interconnect and the address interleaving across a large number of memory banks. In this scenario, having all the PEs executing the same kernel guarantees that precedence constraints bound to active kernel can be satisfied faster, and the time gaps in the schedule are minimized. At the same time, the number of output buffers that are currently active is the lowest schedulable, accordingly to the policy described in Section 4.5.

To compute the schedule, the algorithm in Figure 3.8 considers an active set of nodes, that initially contains all the kernels connected to the input data (head nodes). In the example of Listing 3.1, we have a single active node at each iteration, and the resulting schedule is trivial (Figure 3.9).

$$ColorConvert \longrightarrow Sobel3x3 \longrightarrow Magnitude \longrightarrow Threshold$$

Figure 3.9: Scheduling order for edge detector.

4.4 Tile size propagation

To respect all the constraints imposed by node access patterns, the final tile size for each image must be determined taking into account the effect of *redundant re-computation*. To provide data to kernels which follow in the schedule, a kernel could be required to compute the same values multiple times for adjacent tiles. The final tile size for all images is computed into two passes: (i) the first pass analyzes the graph forward, simulating an execution (based on computed scheduling) and simultaneously collecting the tiling constraints for each kernel; (ii) the second pass performs a backward analysis, starting from the last simulated node, and sets the buffer final overlap according to all collected constraints. For instance, considering the code of Listing 3.1 and an output tile size of 160x120, we get the results depicted in Figure 3.10. The `Sobel 3x3` kernel specifies a neighboring area of one point ($W = 1, H = 1, \forall ix[i] = 1$), while other kernels are simple point operators ($W = 1, H = 1, \forall ix[i] = 0$). On the backward pass, the connection between `Color Convert` and `Sobel 3x3` adds a constraint on the intermediate tile B1; in practice, a tile with no overlap and a tile with a fixed overlap deal with the same image. To satisfy this inter-kernel constraint, tile B0 is enlarged of the absolute difference between overlapping areas and the color conversion kernel is called multiple times to re-compute the points on the borders (exactly

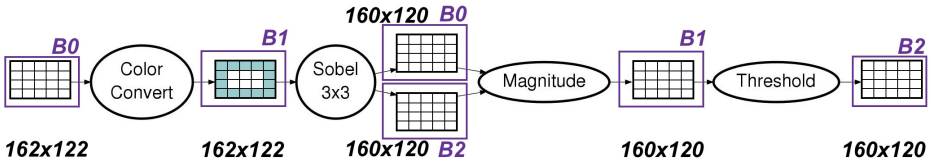


Figure 3.10: Example of tile size propagation.

once per including tile).

When applied, redundant re-computation enforces data locality for intermediate buffers at the cost of transferring and computing the tile borders multiple times, but we have observed with the experiments that this aspect does not affect the general benefits of data locality.

4.5 Buffer allocation and sizing

The *buffer allocation policy* specifies the maximum number of buffers that are allocated in L1 memory and their association to input/output kernel image parameters. The visit order used by the scheduling algorithm guarantees that allocated buffers are used as soon as possible, and nodes that release more data references are executed first, so that we can promote buffer reuse to save L1 memory space.

1. The number of L1 buffers that are initially allocated is equal to the number of input images to the graph.

2. When a kernel is added to the schedule list, we allocate output images to buffers. If there is a buffer that is no longer used, we reuse it, otherwise we increment the buffer count; Due to the double buffering policy, buffers that have been use for inputs cannot be reused for outputs.

3. Using a reference counter, we verify whether images are used by

```
1: – The first buffer is allocated for RGB graph input;
2: –    The second buffer is allocated for Color Convert
   output;
3: – The first buffer enters the free list;
4: –  Sobel 3x3 can reuse the first buffer for its first
   output, then allocates a third buffer for its second
   output;
5: – The second buffer enters the free list;
6: – Magnitude can reuse the second buffer;
7: – First and third buffers enter the free list;
8: –   Threshold can reuse the third buffer for its out-
   put (the first buffer has been initially allocated
   to graph input, so it cannot be reused for graph out-
   put);
```

Figure 3.11: Buffer allocation for edge detector.

other nodes; if there is no further reference, we can reuse it, hence we add the buffer to the free list. Buffers associated with graph outputs can not be reused, so they are never added to the free list.

In addition, a buffer usage data structure is allocated for each element in the associative map, saving information for next steps. The graph in the example (Listing 3.1) requires three buffers, adding a single buffer to fulfill data requirements of virtual images (Figure 3.11).

The *buffer sizing algorithm* computes the maximum size for allocated buffers in L1 memory. The heuristic algorithm that is currently used is depicted in Figure 3.12. This approach differs from the typical buffer sizing problem for data-flow graphs presented in [63]. Fixing a point on the time axis, each buffer contains a tile of a specific image, but the buffer is unique and the referred image changes due to the buffer reuse policy.

```
1: – Set the size of each buffer equal to its upper
   bound (maximum image size);
2: – Compute the total memory footprint of the buffers;
3: while total memory footprint > available L1 quota do
4:    if iteration is even then
5:       – Halve the width of each buffer when the result
   is greater than minimum tile width;
6:    else
7:       – Halve the height of each buffer when the re-
   sult is greater than minimum tile height;
8:    end if
9:    if No changes to buffer size then
10:      – Infeasible (backtracking);
11:   end if
12: end while
```

Figure 3.12: Buffer sizing algorithm.

4.6 Run-time graph

The generation algorithm for CLE graphs interprets all the decision made in previous steps to build a CLE graph for sub-graph to be executed on the accelerator. In addition to preceding constraint and buffer policies, this algorithm applies double buffering to achieve a good overlap between data transfer and kernel execution. Figure 3.13 depicts the execution

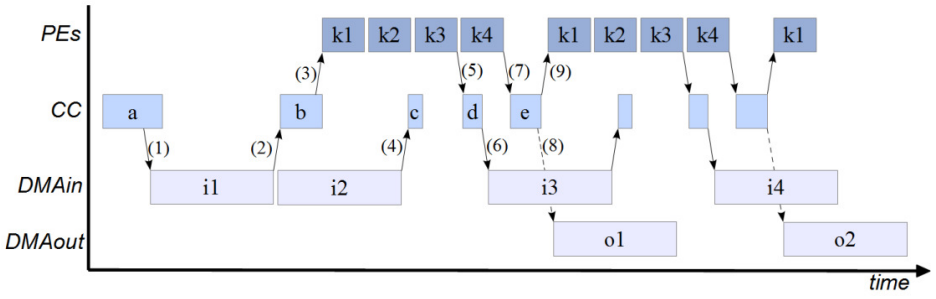


Figure 3.13: Example of CLE run-time schedule.

schema of an OpenVX application equivalent to Listing 3.1. When considering a target architecture without a cluster controller processor, the same tasks could be performed by a thread running on the host side without any loss of generality.

The Cluster Controller (CC) initializes the execution environment (a), in particular the allocation of the L1 buffers, and then program the DMA engine to transfer the first two input tiles tiles from L3 memory to L1 buffers (1). When the first transfer is completed (2), the CC is notified, and then the computation of the kernels is triggered (3). The CC is notified one more time when the second transfer is completed (4), but it does not take any immediate action at this time (c), because the PEs are still executing the kernels for the first tile. When the input buffer is no more occupied by any intermediate results, the CC (d) is notified (5), and a new input data transfer is triggered (6). When the last kernel terminates its execution, the CC is notified again (7), and the DMA engine is programmed to transfer an output tile from L1 buffer to L3 memory (8); in the most general case, each kernel can produce an output image, and so this block could be triggered at the end of any kernel. If a DMA input transfer is completed, that is our case, a new tile computation is triggered (9); however, the next kernel execution is triggered when both events have occurred.

4.7 Nested graphs

Our framework supports the definition of *nested graphs*. Each OpenVX kernel can be associated to a child graph, and the execution of an associated node implies its processing. A nested graph can be created at node execution or initialization time. When created at execution time, nested

graphs do not require any additional API support w.r.t. standard graphs. The creation of framework and data objects is performed inside the kernel function, and this enables the dynamic execution of different graphs based on parameter run-time values. Nested graphs which are created at initialization time require two additional features, *initialization callbacks* and *graph parameters*.

Initialization callbacks are kernel specific functions which are automatically called at node creation time. They can be used to create and validate a nested graph, in this case the node execution just includes the graph processing. This solution is less dynamic than the previous one but at the same time it is much more efficient, since graph creation and validation are performed once in the application time-line, regardless of the number of graph executions. Graph parameters are a reference to a specific node parameter within the graph. These parameters are created with a specific API call (`vxAddParameterToGraph`), and they can be modified between executions (`vxSetGraphParameterByIndex`). Using graph parameters no knowledge of the internal structure is required to set node parameters, and this approach is used to support nested graphs. A graph can be created in the initialization callback using exclusively virtual data types, and then a set of graph parameters is defined representing inputs and outputs at graph level. As a further step, graph parameters are associated to actual parameters already defined in the context of the external graph node, which are passed to the initialization callback. After the type match between actual parameters and virtual data, the graph can be verified and its processing is finally demanded to node execution.

Since the use of nested graphs implies to call a set of OpenVX API

functions, this feature is limited to the kernels that are executed by the host. In turn, nested graphs can be executed (in part or totally) on the accelerator, and this methodology can be also applied recursively. Overall, nested graphs are a viable solution to implement OpenVX kernels that provide higher level features, also facilitating their reuse as software components from a software engineering perspective.

4.8 User-defined nodes

OpenVX enables a programmer to specify custom nodes, and our framework supports this feature. On the host side, programmers have to specify the validator callbacks used at verification stage, and a data descriptor specifying image parameters, tiling behavior and state requirements. For the kernels that are intended to execute on the host side, a C function implementing the kernel must be provided. On the accelerator side, programmers must implement a CLE kernel that accesses image parameters directly in global memory space, without using any intermediate local buffer (as described in Section 4). The benchmarks defined in Section 5 reference several user-defined kernel.

5 Experimental results

We have implemented the full framework described in the previous section to target the STHORM evaluation board (see Section 2). The framework supports the data access patterns described in Section 4.1, enabling the execution of all the kernels included in the OpenVX standard. In addition, we have implemented a library of user-defined kernels.

To assess the benefits of our approach on real applications, we have

Benchmark	Nodes (acc./host)	Accelerator sub-graphs	Images (in/out/virtual)
Random graph	10 / 0	1	1 / 1 / 10
Edge detector	4 / 0	1	1 / 1 / 4
Object detection	4 / 0	1	2 / 1 / 3
Super resolution	8 / 0	1	3 / 1 / 5
FAST9	4 / 0	1	1 / 1 / 3
Disparity	5 / 0	1	2 / 1 / 6
Pyramid	6 / 1	1	1 / 1 / 4
Canny	4 / 1	1	1 / 1 / 5
Optical	4 / 4	4	1 / 1 / 2
Diparity S4	20 / 0	1	2 / 1 / 28
Retina preproc.	165 / 0	8	1 / 4 / 120

Table 3.3: Details on OpenVX benchmarks.

selected a set of benchmarks representing the main fields of image processing domain, with the addition of a single synthetic benchmark:

- *Random graph* is a synthetic benchmark which includes 10 morphological nodes, exposing a wider branching schema compared to real applications with the specific aim to stress allocation and scheduling algorithms;
- *Edge detector* is a basic edge detector including 4 kernels (RGB to gray-scale conversion, Sobel 3x3 filter, magnitude and thresholding);
- *Object detection* is an algorithm to detect objects that have been abandoned/removed in a set of adjacent video frames, and it is based on NCC background subtraction and morphological operators (as described in [64]);
- *Super resolution* represents the recombination phase typical of a computational photography algorithm, which is used to increase

the quality of an image using multiple overlapping pictures of a scene [65];

- *FAST9* implements the FAST9 corner detection algorithm [66];
- *Disparity* computes the stereo-matching disparity between left and right images;
- *Pyramid* creates a set of 4 images which are derived from the input one, weighted using a 5x5 Gaussian kernel and then scaled down;
- *Canny* implements a standard Canny edge detector [67];
- *Optical* is an implementation of the Lucas-Kanade algorithm [68], used to measure optical flow field for a set of keypoints on two adjacent video frames;
- *Disparity S4* is an extension of *Disparity*, using four shifted versions of the right image to support a wider range for disparity.
- *Retina preprocessing* implements the retina preprocessing filter described in [56].

Table 3.3 reports some implementation details about these benchmarks. It specifies the number of nodes executed by accelerator and host, the number of sub-graphs executed on the accelerator and the number of involved images (organized by type). *Pyramid*, *Canny* and *Optical* are implemented as a single host node including a nested graph which is executed multiple times. In this case, the values reported in Table 3.3 are related to the cumulative executions of the inner graph.

As an example, Figure 3.14 depicts the OpenVX graph of *Canny* benchmark. The nested graph contains five nodes, which are *Sobel 3x3*

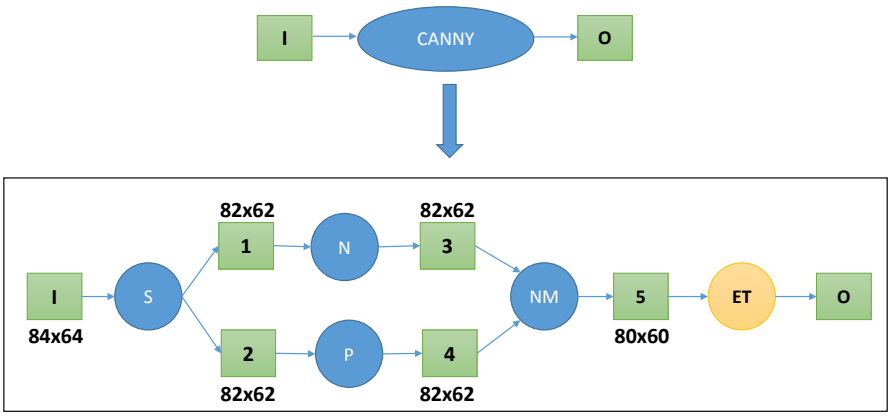


Figure 3.14: OpenVX graph of Canny benchmark with tiling annotations.

(S), *elementwise norm* (N), *phase* (P), *non-maxima suppression* (NM) and *edge tracing* (ET). The last node (ET) is executed by the host, while the other ones are scheduled on the accelerator. ET scans the full image and inserts into a stack data structure the coordinates of the points that are over a high threshold (with a YES status), and then the points between low and high thresholds (with a MAYBE status). Then, it traverses this stack and evaluates the MAYBE points considering the status of their neighborhood. Having a low number of edge points w.r.t. the full image size, this kernel allows to limit the memory bandwidth in the final algorithm stage, and it is computed by the host due to its irregular access pattern. The images are annotated with the tile size computed by our algorithm (see Section 4.4), that is not the same for all nodes. Tile 5 is the output of the accelerator sub-graph, and its size (80×60) is an exact multiple of the full output image (640×480). Tiles 1-5 are two points wider (82×62) to support the application of non-maxima suppression kernel, which requires a 3×3 window. Tile I is even wider (84×64), with the aim of providing proper input to Sobel kernel in accordance with its output size. The host kernel (ET) reads

the full graph output to produce the final image O , directly accessing L3 memory through Cortex-A9 cache hierarchy. This type of result cannot be obtained by fusing kernels and optimally tiling the generated loop, as in the most general case tiling requirements of different kernels are not homogeneous. Overall, the use of OpenVX graphs enables a global level of optimization which is not possible under a single-function paradigm.

Retina preprocessing is a brain-inspired visual processing algorithm described in [56]. It is composed by 24 building blocks, which can be represented with an OpenVX graph including both standard and user-defined nodes. The final graph size is much larger than typical OpenVX applications, but we have included it to give an idea of how our approach scales very well also to future applications that will emerge when OpenVX will start to be heavily utilized in industry. In our implementation, some of these blocks (complementary, sum, subtraction) are directly mapped on OpenVX nodes, while the other ones have been implemented as the composition of multiple nodes. In particular, we have defined two functions to describe specific building blocks (`retina_opponency` and `LGN_opponency`), and another one (`LGN`) to instantiate a sequence of identical blocks that are recurring in the last algorithm stage. Each function call adds a set of nodes to the OpenVX graph, and overall they are invoked multiple times to build the full application. This example demonstrates a major benefit of our approach, that is the support to *composability*. Our framework provides an extendable set of software components which can be assembled in various combinations to satisfy specific functional requirements. This is a key feature to implement complex applications, nevertheless it is often disattended by many tools.

Using our approach, the orchestration of multiple kernel nodes and

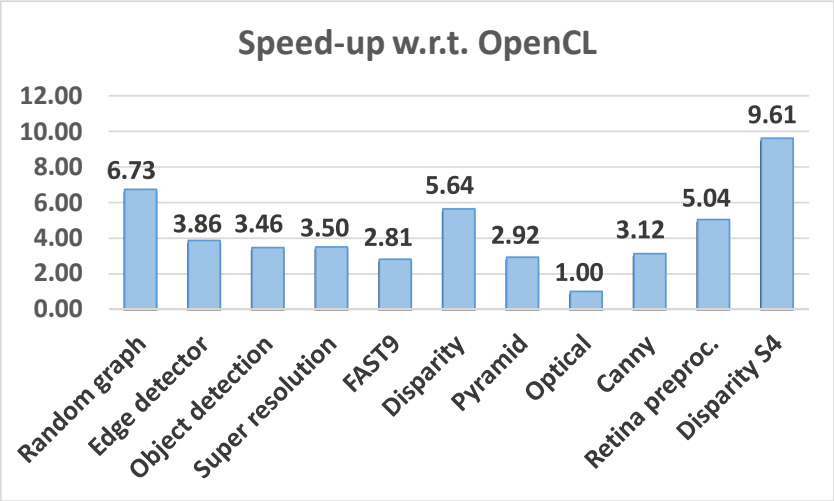


Figure 3.15: Speed-up of OVX CLE w.r.t. standard OpenCL approach.

accelerator sub-graphs is totally transparent to the programmer. In *Optical*, a single host kernel executes a nested graph multiple times. In *Retina preprocessing*, the presence of statistical kernels used for image normalization (mean and standard deviation) induces the partition of the OpenVX graph into eight sub-graphs at CLE run-time level. In this specific case, all the referenced kernels are member of classes A, B and C (see Figure 3.6), and our framework orchestrates the execution on the accelerator to get the maximum speed-up.

All tests are performed with an input image size of 640×480 pixels. This setup is sufficient to already see the effects of memory bandwidth, due to the limited main memory bandwidth available for the STHORM accelerator on the evaluation board.

5.1 Comparison with OpenCL

Figure 3.15 shows the speed-up of the OpenVX accelerated versions w.r.t. the same applications implemented on the standard OpenCL 1.1 run-time

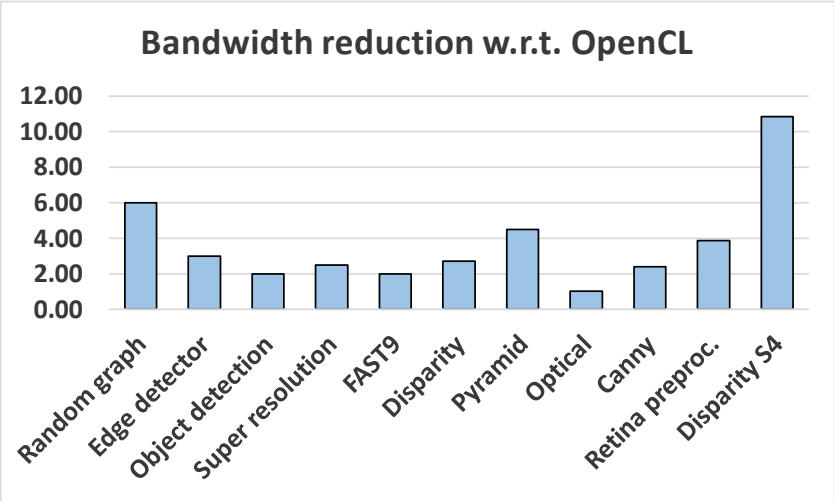


Figure 3.16: Bandwidth reduction using OVX CLE.

provided by STHORM software environment. "OVX CLE" denotes the version executed using our framework. Each OpenCL application is built using a library of image processing kernels, with the aim to mimic the component-like approach promoted by OpenVX.

Figure 3.16 depicts the *L3 bandwidth reduction*, which is computed as the ratio between the amount of data transferred by the two implementations (OVX CLE/OpenCL). Since each OpenCL kernel copies its outputs to L3 memory in order to pass data to the next one, the trend of the speed-up is closely related to the L3 bandwidth reduction. This is particularly evident for *Optical*, which is composed by a single kernel (Scharr 3×3) executed multiple times. In this case we get no advantage from bandwidth reduction, and the related speed-up is very close to one. This is an expected result, because the baseline OpenCL implementation exploits parallelism as effectively as our run-time if main memory effects are not important.

Figure 3.17 reports the bandwidth requirements of the benchmarks

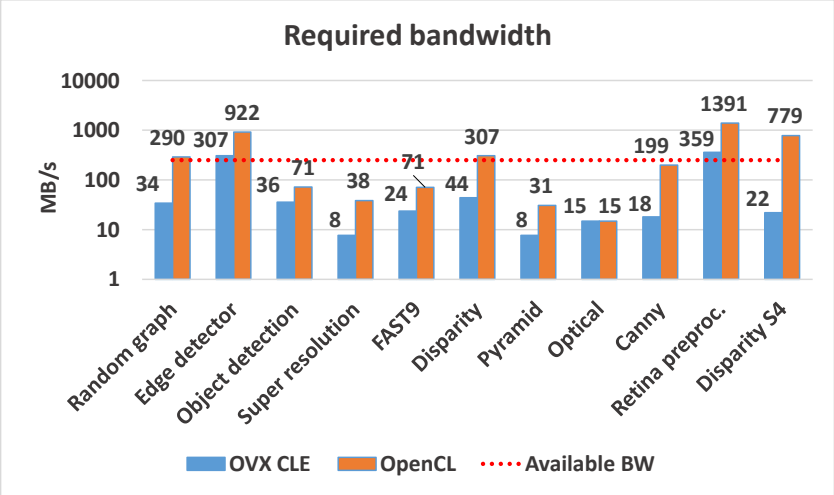


Figure 3.17: Bandwidth required by applications for both OVX CLE and OpenCL.

for both OVX CLE and OpenCL. These values have been computed as a ratio between the total transferred bytes and the related computation time required by benchmarks. This is an upper-bound case, which suppose a total overlap between data transfer and computation. The bandwidth required by an the OpenCL application exceeds the available one in four cases (`Edge detector`, `Disparity`, `Retina preprocessing`, `Disparity S4`), limiting the speed-up of the OpenCL solution. In OVX CLE benchmarks, the bandwidth threshold is exceeded twice for a limited amount (`Edge detector`, `Retina preprocessing`), and this effect impacts on execution efficiency (see next section).

All the measures do not include the initialization time for OpenVX and OpenCL contexts. In OVX CLE, the algorithms which verify the graph are polynomial and the computational complexity is $O(n*e)$, where n and e are the number of nodes and edges in the CLE graph. These algorithms are executed on the host side and they have not shown any relevant impact into the execution of the selected benchmarks.

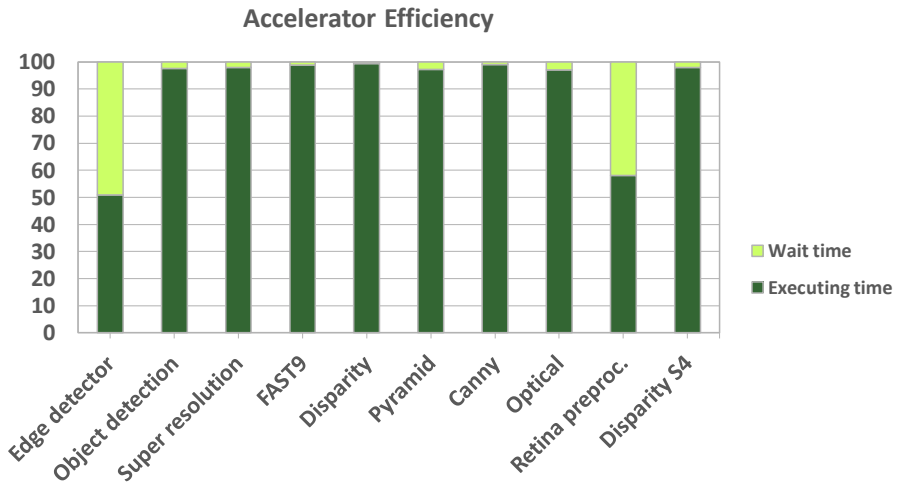


Figure 3.18: Breakdown analysis of accelerator efficiency.

5.2 Execution efficiency

Figure 3.18 depicts the execution efficiency related to benchmarks' execution on the accelerator, that is computed as the percentage of total graph execution time. In *execution time*, cores are actually executing kernel instructions, while the *wait time* is the time required for transfers and not overlapped to the execution time.

Edge detector and *Retina preprocessing* are characterized by a significant level of inefficiency. In both cases the wait time is high, and the execution on the accelerator is dominated by data transfer times (see Figure 3.17). In *Edge detector* this is due to the low computational intensity of the algorithms. In *Retina preprocessing* this effect is related to the high number of statistical kernels that imply multiple splits in the graph, and the traffic to/from L3 memory is increased w.r.t. an application with the same number of nodes and a single run-time graph. These outcomes are not limitations of our framework, since in both cases it guarantees the maximum overlap for data transfers and computation, but they are due to the limited accelerator bandwidth of the STHORM evaluation

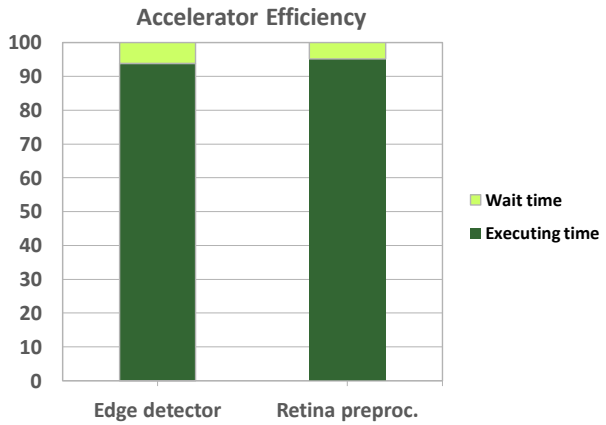


Figure 3.19: Efficiency of Edge detector and Retina preprocessing with memory bridge at 400 MHz (simulation).

board. As explained in Section 2, this is a worst case scenario compared to a fully integrated SoC, where the accelerator will have a much greater share of L3 bandwidth. Nevertheless, we can simulate a more realistic scenario using the simulation tool in the STHORM SDK. Figure 3.19 reports the efficiency of *Edge detector* and *Retina preprocessing* when the L3 memory interface is clocked at 400 MHz. As expected, the wait time is drastically reduced in both cases. This approach is equivalent to move up the reference line in Figure 3.17.

5.3 Comparison with similar tools

KernelGenius [69] is a tool that enables the high-level description of vision kernels using a custom programming language. Starting from an application described as a set of vision kernels, KernelGenius aims at generating an optimized OpenCL kernel for the STHORM platform, with a totally transparent management of the DMA data transfers. The structure of the tiling problem for a single kernel is analogous to the formulation we are using in our work, but there is no support for most data patterns

Benchmark	KernelGenius (ms)	OVX CLE (ms)
Sobel 3×3	22.2	20.42
Convolution 5×5	30.1	25.58
Erode/Dilate 5	25.6	12.84
FAST9	5.4	5.9
Canny	182.0	58.2

Table 3.4: Comparison with KernelGenius

(classes C, D, E and F of Figure 3.6). Table **??** reports a comparison between OVX CLE and KernelGenius for a subset of our benchmarks that are compatible with KernelGenius current limitations.

Halide [54] is a tool specifically designed to describe image processing pipelines, with a strong focus on computational photography. To implement an algorithm with Halide, the programmer must specify a functional description using a domain-specific language embedded in Python or C++. Halide defines a model based on stencil pipelines, with the aim to find a trade-off between locality, exploitation of parallelism and redundant re-computation. Compared with our approach, Halide presents the following limitations:

- Irregular algorithms and data patterns are not supported by the language.
- Composability of software modules is limited, as programmers can just express single algorithms or independent pipelines.
- Schedule and tile size are always explicit, there is no default to implicitly select the best choice for a specific target.

Halide targets high-end platforms such as Intel multi-cores and high performance GPGPUs, while OVX CLE is focused on many-core accelerators for the embedded market. Since Halide and OVX CLE do not

include a common target platform, a comparison based on benchmark timings cannot be reported.

HIPAcc [70] is a framework for image processing that includes a domain specific language embedded in C++ and a source-to-source compiler. Its model of computation includes a reduction pattern, which can be used to compute statistical and recursive neighbor operators. Host code can be used to implement other unsupported patterns, so the coverage is equivalent to our solution. HIPAcc supports multiple architectural targets, generating the image processing code through a visit of the Abstract Syntax Tree (AST) using the compiler front-end. This approach is different from the one promoted by OpenVX, which is based on run-time steps. Some relevant differences are:

- An OpenVX graph can be modified at run-time based on external parameters, while HIPAcc code has to be completely re-compiled.

- An OpenVX framework can transparently support load balancing using dynamic resource management, while in HIPAcc this feature is not target agnostic and must be guided by the programmer .

- The validation step in OpenVX programs requires an initial overhead, but it simplifies debugging and enables additional features (such as nested levels of execution, which are not supported by HIPAcc).

Both Halide and HIPAcc include OpenCL in their target list, but the generated code is not intended to run on a many-core accelerator. Future extensions of these works would enable us to compare them with our OpenVX framework.

6 Related Work

The role of OpenVX for performance optimization has been introduced in [71]. The authors are active actors of the standardization process, and they discuss how OpenVX run-times could provide both kernel and system level optimizations. In the context of embedded vision system, in the last years many FPGA-based solutions has been presented (e.g., [72] and [73], and [74]). We can also find several examples of domain-specific architectures. CHARM [75] and AXR-CMP [76] propose a framework for composable accelerators assembled from accelerator building blocks with dedicated DMA engines, while NeuFlow [53] is a special-purpose data-flow processor tailored for vision algorithms. Both FPGA and domain-specific architectures have emerged to satisfy demands for power-efficient and high-performance multiprocessing. Our solution is based on a general-purpose accelerator, using a programming model which takes into account the specific needs of CV domain but still promoting a general-purpose programming paradigm. Darkroom [77] is a tool that synthesizes hardware descriptions for ASIC or FPGA, or optimized CPU code using an optimally scheduled pipeline. The tiling approach is similar to our solution, but there are some limitations. They consider just two access patterns (point-wise and stencil), that are the only to be compatible with the pipeline execution.

OpenCV [78] is an open-source and cross-platform library featuring high-level APIs for Computer Vision. OpenCV is the de-facto standard in desktop computing environment, its mainstream version is optimized for multi-core processors but it is not suitable for acceleration on embedded many-core systems. Some vendors provide accelerated versions of OpenCV which have been optimized for their hardware (e.g. OpenCV for

Texas Instruments embedded platforms [79] or OpenCV for Tegra [80]). As an alternative to OpenCV, Qualcomm provides a specific library for ECV which includes the most frequently used vision processing function. This library is called FastCV [81], and it is optimized for ARM-based processors and tuned to take advantage of Qualcomm's Snapdragon processors. As a matter of fact, OpenCV needs a lower-level middleware for accelerating image processing primitives. This is precisely the goal of OpenVX, which aims at providing a standardized set of accelerated primitives, thereby enabling platform agnostic acceleration.

OpenCL [82] is a very widespread programming environment for both many-core accelerators and GPGPUs, and it is supported for an increasing number of heterogeneous architectures; for instance, Altera supports OpenCL on its FPGA architecture [83]. The OpenCL memory model is too constrained for SoC solutions, hence many extensions have been proposed by vendors. For instance, AMD provides a zero-copy mechanism to share data between host and GPU in Fusion APU products, also enabling the access to GPU local memory by host side through a unified northbridge with full cache coherence [84]. In a many-core accelerator we need even more control on data allocation, because cores are not working in lock-step. In addition, we need the possibility to map the logical global space at different levels of the memory hierarchy, to efficiently maintain state between kernels.

Graph-structured program abstractions have been studied for years in the context of streaming languages (e.g. StreamIt [52]). In these approaches, static graph analysis enables stream compilers to simultaneously optimize data locality by interleaving computation and communication between nodes. However, most research has focused on 1D streams,

while image processing kernels can be modeled as programs on 2D and 3D streams. The model of computation required by image processing is also more constrained than general streams, because it is characterized by specific data access patterns. Some good results have been achieved with special-purpose data-flow processors targeted for vision algorithms (e.g. NeuFlow [53]). Our model takes into account the specific characteristics of image processing domain, but we target a general-purpose accelerator.

Stencil kernels are a class of algorithms applied to multi-dimensional arrays, in which an output point is updated with weighted contributions from a subset of neighbor input points (called window or stencil). Our definition of tiles is equivalent to a 2D stencil. Many optimization techniques have been proposed to execute stencil kernels on multi-core platforms [85], but an effective solution for many-core accelerators executing heterogeneous vision kernels has not been proposed yet. Such a solution has to consider all the data access pattern specific of this domain, handling the possible overlapping of input windows and providing a solution for the access patterns that are not properly describable in terms of stencil computation.

Many previous works have proposed specific optimizations for architectures with explicitly managed scratchpad memories, such as Cell BE [86–88]. In this chapter we propose a totally transparent approach for the well-defined scope of image processing applications. We do not introduce any new programming model which programmers should learn in addition to the standard OpenVX API, and overall we also manage all the cases in which the localized execution is not directly possible.

7 Conclusion

In this chapter we have presented a framework which aims at improving the execution efficiency of image processing algorithms on many-core accelerators. The proposed solution applies a set of algorithmic steps to map an OpenVX application into an OpenCL low-level graph. Experimental results on the STHORM platform show that our approach provides huge benefits in terms of speed-up, considering both a sequential version and an accelerated OpenCL version, and also in terms of execution efficiency and bandwidth reduction.

Chapter 4

An OpenVX environment to optimize embedded vision applications on parametric parallel accelerators

1 Introduction

To increase the productivity of application designers, it is important that the programming model exposes high-level constructs for the exploitation of parallel and heterogeneous resources. In this way, experts of the application can focus on its partitioning and deployment, without the need for expertise on the hardware details. This is particularly true in the CV domain, where the expertise of application designers is typically on the algorithms. OpenVX [23] has been introduced as a cross-platform standard for imaging and vision application domains, with the aim to raise significantly the level of abstraction at which CV applications should be

coded. Using OpenVX, the details of the hardware platform are hidden in the underlying run-time environment (RTE) layer. This approach enables the portability of vision applications across different heterogeneous platforms, delegating the performance tuning to hardware vendors, who provide an efficient RTE with architecture-specific optimizations. The work presented in the previous chapter moves in this direction, but it still lacks generality on architectural aspects.

In this chapter we introduce *ADRENALINE*, a framework for fast prototyping and optimization of OpenVX applications on heterogeneous SoCs with many-core accelerators. ADRENALINE consists of an optimized OpenVX run-time system, based on streamlined OpenCL support for a generic heterogeneous SoC template like the one introduced in Section 1. The tool comes with a virtual platform modeling the target architecture template, which can be easily configured along several axes. The run-time system includes several optimizations for the efficient exploitation of the explicitly managed memory hierarchy adopted in the targeted SoCs, but it can be easily extended to consider other optimization opportunities. Similarly, the virtual platform can be expanded to model additional architectural blocks in a simple manner. Finally, we provide relevant use cases for our tool, showing how it can support the needs of several users:

- *Researchers and/or SDK vendors* can explore various platform-specific optimizations, scheduling policies and algorithms for the implementation of the OpenVX support layer.

- *Application developers* can explore different partitioning solutions (host vs accelerator, parallelization) for different applications;

- *Platform engineers* can quickly evaluate different architectural configurations for a target CV application;

2 Alternative tools

In the previous chapter we have introduced several tools that represent an alternative to OpenVX, mainly KernelGenius [69], Halide [54] and HIPAcc [70]. Even if these tools enforce good software engineering practices and successfully reduce the time-to-market of vision-based applications, they are not intended to improve performance and energy efficiency of embedded many-core accelerator. Moreover, they do no provide the full set of tuning capabilities that are offered by ADRENALINE.

Considering market products, Cadence IVP processor [89] exploits a comparable approach. It is based on the proprietary Xtensa CPU/DSP technology, which allows to configure and build application-specific processors coupled with DSP accelerators, with a software toolchain that supports OpenVX. This solution targets a configurable technology, but the flexibility of configuration is more restricted than ADRENALINE one. Due to its totally open-source approach, ADRENALINE is definitily a better solution for researchers.

3 ADRENALINE internals

ADRENALINE comes with a *virtual platform* modeling the generic architectural template described in Section 1 and an *OpenVX run-time* optimized for this target. This tool has been developed following two main objectives, application benchmarking/profiling and architecture tuning.

3.1 Virtual platform

The virtual platform is written in Python and C++. Python is used for the architecture instantiation and configuration, and also for the high-level execution management. C++ is used for implementing the models in an efficient manner, so that only binary code is called during normal execution.

A key functionality is the possibility of describing a block as a combinatory network. For instance, it is used to describe the interconnection between the cores and the TCDM. A library of basic components is available, but custom blocks can also be implemented and assembled. At run-time, any request that arrives at the block boundary is enqueued and stays inside the block until it is elected to an output boundary. For that at each cycle, a resolution cycle is executed in the network. At each resolution cycle, all the enqueued requests are handled from the compoenents at the input boundaries and propagated to the output boundaries, and are at the same time affected by each component that they go through. Some components will just select one request amongst all the ones that has been propagated to this component (e.g. the logarithmic trees), others will apply a remapping. When the resolution cycle ends and a new one is started, all the requests that are still enqueued in the network starts again from the beginning, as the resolution policies might have been updated since the last cycle.

In the following paragraphs we report some additional details about the most relevant blocks.

OpenRISC core. It is modeled with an Instruction Set Simulator (ISS) for the OpenRISC ISA [36], extended with timing modeling to consider the various sources of pipeline stalls. Adopting the same interface,

an ISS for a different architecture can be plugged.

Memories. The memory blocks use a simple timing model, with a fixed latency for each reported access.

L1 interconnect. This block has an important impact on data accesses. The timings of an application can really differ depending on how the data buffers are accessed from memories. In our model, each target memory bank can accept one request per cycle, as provided by the architectural template.

Other interconnects. In the interconnect model a single request is never split, it traverses all the interfaces to the final target not allowing fine-grained arbitration. However there is a bandwidth model which is applied on each request, with the aim to report realistic timings.

DMA. The DMA sends a single synchronous request to the interconnect for each line to be transferred. The interconnect reports a latency for the whole transfer. As the DMA is able to fully stream several input requests while writing output requests, another input request can be scheduled after the latency of the first input request.

Shared instruction cache. The shared instruction cache model is made of an interconnect model and a set of cache banks. Each cache bank is a classic cache model that is able to report the latency to a request, depending on the fact it is a hit or a miss. The interconnect model is quite similar to the memory interconnect model with a support for multicast and a model of the L0 prefetch buffer. The L0 prefetch buffer is modeled as a classic cache with a single entry, so that only the misses are propagated to the L1 interconnect.

The current version supports the simulation of a single cluster and a single-core host. The host used for experiments is an OpenRISC core,

with the aim to have comparable metrics for results. To instantiate a fully heterogeneous system, the host could be configured as an ARM or a x86 core. The OpenRISC core has a simple pipeline which allows modeling all the following sources of stalls. When an instruction finishes or when the core goes to running model, an event is enqueued at the time the instruction must be executed. Once the event is scheduled, the ISS is called to execute the instruction model. This execution can trigger accesses in the platform (for instruction fetching or data accesses) which report timing to be taken into account for the calculation of the instruction cost. The instruction is executed in one call to the ISS, and this call reports to the platform the cost of the instruction that has been estimated. The platform uses this information to enqueue the event for the next instruction at the appropriate timing. Other details of the core model include:

- *Instruction cache.* The instruction cache model is external to the ISS, as we can model more complex beaviors inside the virtual platform. The ISS is calling this model each time it needs to fetch a new instruction, through a synchronous request. The ISS stalls the core by the extra amount of cycles compared to the normal case where the instruction is fetched in one cycle.

- *Data dependency.* A few instructions have a latency above 1 cycle. This means the destination register is not available for a number of cycles correponding to the latency minus one. This status does not stall the pipeline unless an instruction uses the register before it is available, in which case the pipeline is stalled for the remaining number of cycles. To model this behavior, the ISS maintains a scoreboard that tells at which time each register is available. Any

instruction with a latency can update this scoreboard, and any instruction with input registers can check this scoreboard. The most important example is the load instruction that has a latency of 2 cycles, and leads to an important error when it is not modeled in a scenario where the compiler is not able to hide the latency with other instructions. Other instructions like hardware floating-point unit instructions has also an associated latency.

- *Load/store latency.* The pipeline during a load or a store is not stalled for accesses with the minimal latency as it is the case for accesses to the local shared memory without any contention. However, if the access has an extra-latency for any reason, like a contention or the fact it is a remote memory, the pipeline is stalled for the corresponding amount of extra cycles. The ISS models this behavior by just accounting the extra-cycle to the instruction cost in order to simulate the stall behavior.

ADRENALINE provides the following tunable parameters: number of cores; hardware FPU enabled/disabled; available memory at L1, L2 and L3 levels; DMA bandwidth/latency. Moreover, the virtual platform can be extended by defining new modules. A module is a Python class which declares the input and output ports that can be connected to other modules to specify the connections between the architectural blocks (e.g., a router to a memory). Then each Python class has a corresponding C++ class that implements the block model. When the platform is started, each C++ class receives all the configuration from the Python class (how ports are connected, property values), so that only C++ code is running during simulation. Overall, writing a new module requires a limited effort to write the Python class, as it mainly contains declarations. Then

the difficulty of writing the C++ model usually depends on the timing behavior complexity of the block.

3.2 OpenVX run-time

The OpenVX runtime provided in ADRENALINE is a generalization of the implementation for STHORM described in the previous chapter. An extension to OpenCL is used as low-level runtime, but it hides all the low-level details of OpenCL host code and automatically handles data partitioning that would otherwise be required explicitly in the kernel code. To provide more flexibility in kernel specification, we map all the supported access patterns (see Section 4.1) into three kernel classes:

- `CLE_KERNEL_NORMAL` is used for point and local neighbor operators. Tiling is used on both input and output images.
- `CLE_KERNEL_STAT` is used for statistical operators. Tiling can be activated just on input images, and we can use a persistent buffer to implement a reduction pattern ”walking” through the tiles.
- `CLE_KERNEL_HOST` is used when it is impossible to apply tiling to input data, and this implies the kernel is executed by the host processor. This is true for global operators, and also in case of many geometrical operators we cannot apply a classical input tiling due to the irregular shape of the neighboring area.

A `CLE_KERNEL_HOST` kernel is executed by the host processor. This could be the best solution for global operators, and also in case of many geometrical operators we cannot apply a classical input tiling due to the irregular shape of the neighboring area. More in general, this option can

be used to force kernel execution on the host side, that could increase overall performance in some specific cases (see Section 4).

The use of *state buffer* has been introduced in Section 4.1 to handle the computation of recursive neighbor operators. In the ADRENALINE runtime we further generalize these policies, with the aim to provide more flexibility to kernel programmers:

- `CLE_STATE_NONE`: no state is required;
- `CLE_STATE_SCALAR`: an amount of state proportional to a scalar.
- `CLE_STATE_BORDER`: an amount of state proportional to the border size.
- `CLE_STATE_TILE`: an amount of state proportional to the whole tile size.

4 Experimental results

In this section we illustrate some use cases for ADRENALINE, and we show a comparison with a real platform.

4.1 Comparison with a real platform

To validate the virtual platform, we have compared ADRENALINE with a STHORM-based board. Figure 4.1 shows the speed-up of the OpenVX accelerated versions compared to their OpenCL implementation using ADRENALINE. In these tests we have considered a configuration for the virtual platform that resembles STHORM: 16 cores with hardware floating point support, 256 KB L1 memory, 2 MB L2 memory, 500 MB

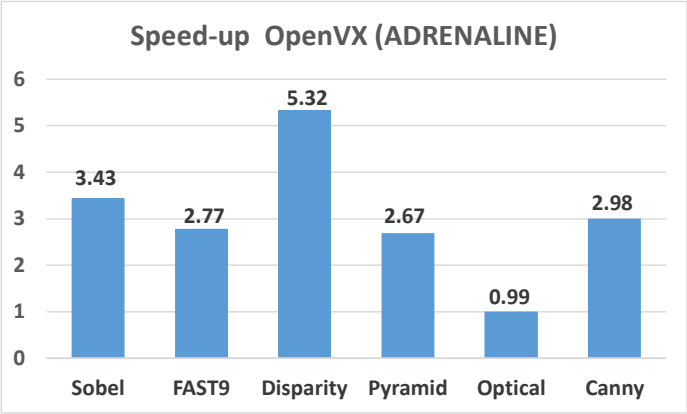


Figure 4.1: Speed-up of OpenVX compared to OpenCL (ADRENALINE).

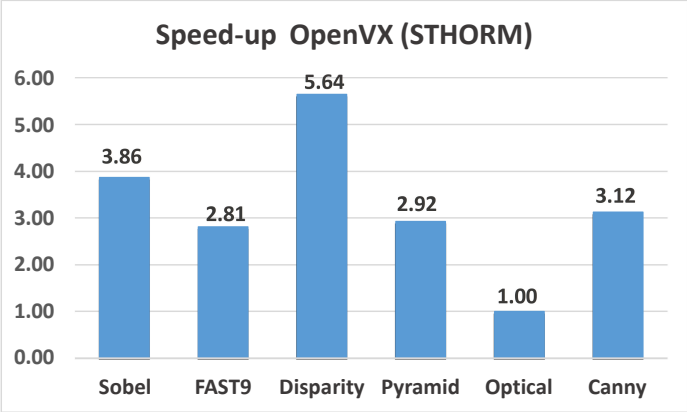


Figure 4.2: Speed-up of OpenVX compared to OpenCL (STHORM board).

L3 memory, 250MB/s for DMA bandwidth, 450 cycles for DMA latency. Figure 4.2 reports the speed-up values measured on the STHORM platform. The comparison with Figure 4.1 shows that the speed-up trend is coherent, with an average margin of 6%. This implies that ADRENALINE models the bandwidth effects on a real embedded system with very good accuracy, even when the bandwidth is a severely constrained resource.

4.2 Application partitioning

As an example of partitioning optimization, we have considered a single application graph containing both a Canny edge detector and a FAST9 corner detector.

Figure 4.4 shows the graph representation of a Canny edge detector. This algorithm applies a Gaussian filter to de-noise the input image, and then apply a Sobel operator to find horizontal and vertical gradients. Starting from gradients, it computes the edges in terms of magnitude and phase. Finally, it performs *non-maxima suppression* to thin the edges and *hysteresis thresholding* to remove loosely connected points.

FAST9 corner detector [66] extracts corners by evaluating pixels on the Bresenham circle around a candidate point. If N contiguous pixels are brighter or darker than the central point by at least a threshold value then this point is considered to be a corner candidate. For each detection, its strength is computed and finally a non-maxima suppression step is applied to reduce the final points. The OpenVX standard library provides a specific node for FAST9 algorithm (`vxFastCornersNode`), which reads an input image and outputs an array of corner coordinates. This version differs from the one used in the previous sections, which is split into multiple kernels to run on the accelerator but provides as output a full image.

Figure 4.3 shows the timings resulting from the different mappings of FAST node w.r.t. Canny edge detector. For these specific implementations, the best solution is the one executing Canny on the accelerator and FAST9 on the host side. This is a non trivial result that fully highlights the benefits of this kind of analysis.

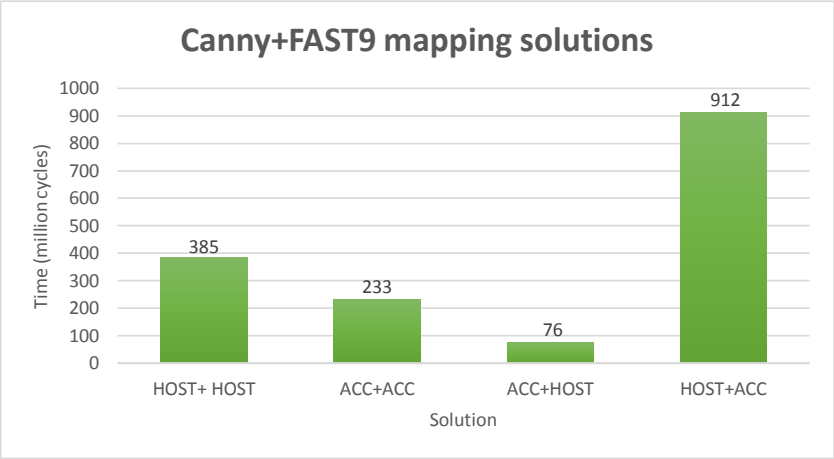


Figure 4.3: Mapping of Canny and FAST9.

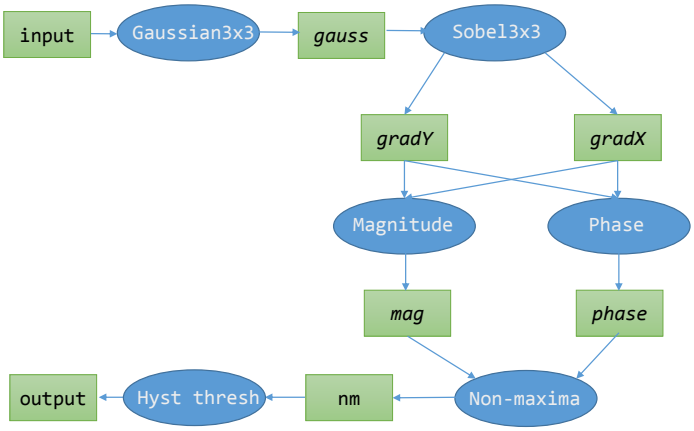


Figure 4.4: Canny edge detector.

4.3 Architectural configuration

To evaluate different platform parameters, we have executed Canny edge detector measuring the execution time when the number of cores is set to increasing values. Figure 4.5 depicts the execution time in cycles. The bar on the left represents the time required to execute the whole application on the host using the reference implementation provided by Khronos. For this application, the lack of optimizations and the overhead due to OpenVX data access functions make more profitable the execution

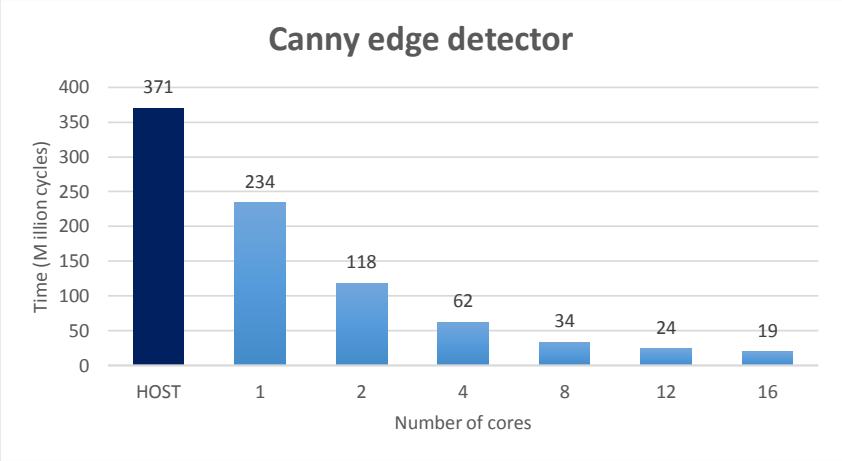


Figure 4.5: Evaluation of Canny edge detector.

Cores	Speed-up
1	1.00
2	1.98
4	3.79
8	6.97
12	9.70
16	12.00

Table 4.1: Computed speed-up.

on the accelerator, even in case of a single core (1.58x).

Table 4.1 reports the speed-ups obtained using different configurations of the many-core accelerator. The efficiency decreases with the number of cores, and using this table a designer can choose a valid trade-off. Table 4.2 reports the effect of DMA latency on execution time, with a bandwidth set at 8 bytes/cycle. In this case our optimization approach based on localized execution and double buffering hides the variations of platform parameters related to DDR connection. In the most general case, this conditions can be verified for any application using our tool.

Latency (cycles)	Execution time (Mcycles)
100	61.75
200	61.98
400	62.70

Table 4.2: Effect of DMA latency on execution time.

5 Conclusion

In this chapter we have introduced ADRENALINE, a framework for fast prototyping and optimization of OpenVX applications that includes an OpenVX run-time and a virtual platform. The use cases proposed in Section 4 show how ADRENALINE can be used to highlight optimization opportunities for a wide range of end users, from hardware designers to CV researchers.

Chapter 5

Providing lightweight OpenVX support for resource-constrained mW-scale parallel accelerators

1 Introduction

Market forecasts [90] [91] report that there will be more than 25 billion Internet-of-Things (IoT) devices by 2020. A key enabler for the IoT is the development of next-generation *smart sensors*, that combine standard analog/digital transducers and communication interfaces with powerful data-processing capabilities. At the beginning of its story, the IoT paradigm was characterized by the critical role of cloud infrastructures as providers of computational power, and its terminal constituent devices were relatively unsophisticated. With the advent of smart sensors this trend is evolving rapidly [92] towards the *edge computing* paradigm, that

moves the execution environment of applications away from centralized nodes and to the logical extremes of a network. This is made possible by recent designs for IoT devices, which combine complex processing capability with ultra-low-power operation. Sophisticated data manipulation is thus enabled at the edges of the cloud, significantly reducing the amount of data to be sent through bandwidth-limited communication interfaces (e.g., serial peripheral interface). In a nutshell, a strong requirement for smart sensors is the compliance with a limited power budget combined with high performance demands and high energy efficiency.

To deliver the required performance/watt targets, the most promising platforms available on the market are focusing on a class of heterogeneous systems including a microcontroller unit (MCU) coupled to a *programmable, mW-scale parallel accelerator* [89] [93] [94] [95] [96]. The adoption of ultra-low-power parallel accelerators as a co-processor provides hundred-fold increase in OPS/W [97] compared to state-of-the-art microcontrollers. The advent of such devices, combined with the widespread diffusion of miniaturized cameras [98], is becoming the key enabler for building sensor nodes capable of running computer vision (CV) workloads, which will be at the heart of tomorrow's most ambitious frontier of the IoT (smart cities, the internet of vehicles [99]).

Similar to what has happened to high-end heterogeneous systems, programmability will become a key issue in this domain as well. The adoption of mainstream programming paradigms is an appealing solution, but it is complicated by the constrained nature of IoT designs (power, memory, etc.). On the other hand, OpenVX [23] represents the state-of-the-art for embedded vision programming, as witnessed by its widespread adoption in commercial products. In the most common case, an OpenVX

framework relies on a *graph-based RTE*, assuming that a data structure describing the graph is allocated in the accelerator memory [29]. A key trait of mW-scale accelerators is the strongly limited amount of available on-chip memory, which requires dedicated memory management techniques to enable the efficient execution of graphs of arbitrary large size. To this aim OpenVX supports *data tiling* [100], a well-known technique that exploits spatial locality in a program by partitioning large data structures into smaller chunks that are brought in and out of the target memory via DMA transfers. When tiling is applied, the number of nodes in the graph data structure becomes much larger than the number of application kernels, due to the tiles and to computation artifacts generated by their presence (e.g., image borders, corners or inner tiles, double-buffering). Consequently, the scarce amount of on-chip memory can be insufficient to contain both application and RTE data and code.

In this chapter we propose a novel approach to provide OpenVX support in mW-scale parallel accelerators. Experiencing with real-life applications, we realized that standard data management techniques (e.g., tiling and double buffering) are not enough to guarantee the stringent memory constraints that have been outlined for this category of devices, and we had to re-think the way such execution model could be supported. Our proposal leverages a new API to support static management of application graphs. An OpenVX graph can be created, verified and executed on the developer workstation using the standard OpenVX execution flow, which guarantees full portability on any platform an OpenVX framework is available for. Once a program has been developed and tested following this standard flow, the proposed API extension allows to save the

resulting graph into a static representation as a platform-dependent binary file. The OpenVX runtime data structure management is replaced by a control-code generation approach, which reduces the total RTE footprint (code + data) to a few Kilobytes. This approach drastically reduces also the platform energy consumption, by replacing costly and frequent accesses to the runtime data with cheaper control instructions (ALU operations). It also makes effective usage of low-bandwidth data channels by means of software techniques aimed at maximizing the computation locality (thus minimizing the request for external bandwidth). The control-code generation is complemented by a minimal RTE design for the target platform, called *milliVX*, which includes a small subset of the OpenVX specification to read a generated binary and offload its execution to the accelerator. It is important to underline that our approach takes advantage of the static structure extracted by an OpenVX program to optimize the execution stage in terms of memory footprint and execution time, but at the same time it fully preserves the dynamic features of the original OpenVX standard, namely *graph updates* and *node callbacks*. Graph updates are dynamic modifications to the graph data structure, generating a different graph that requires a further verification stage before its execution. Node callbacks are used to control the graph execution flow by calling functions upon termination of a particular node. *milliVX* provides low-cost yet full-fledged support to those features.

To assess our approach, we provide a reference implementation for the OpenVX extension and the *milliVX* specification using a publicly available research tool [29] for OpenVX development on ultra-low-power parallel accelerators [96]. Experimental results show that our approach

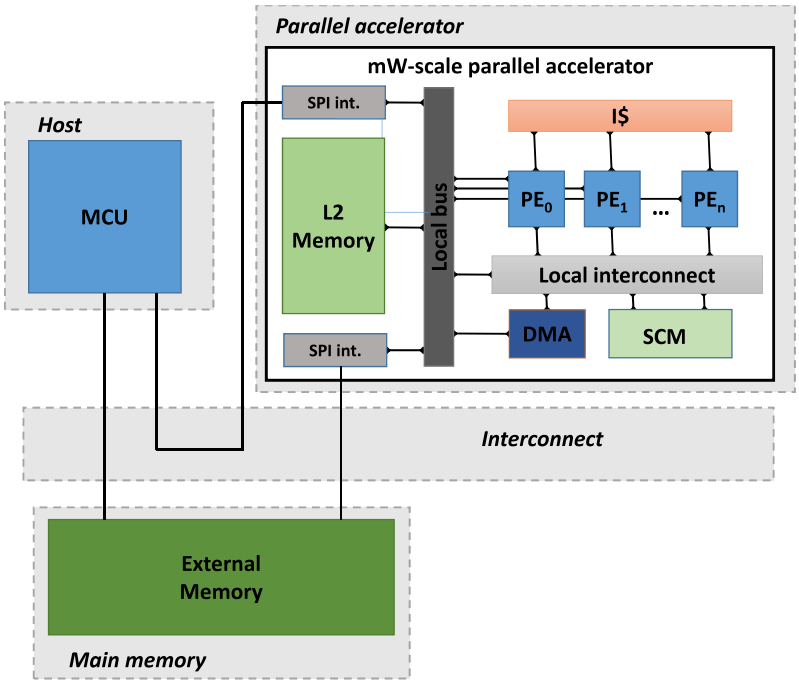


Figure 5.1: Heterogeneous architecture model.

achieves 68% memory footprint reduction and about 3× execution speed-up compared to a baseline. Moreover, the data memory bandwidth towards off-chip memory is further reduced by 10% and energy efficiency is improved by 2×.

2 Background

2.1 Platform template

Today the market offers several products that can be included in the class of mW-scale parallel accelerator, mostly in the segment of licensable IP cores [89] [93] [94], but also hardware platforms from research institutions are available [95] [96]. The size, performance, and power consumption of these solutions greatly depend on the core configuration,

synthesis flags, physical-IP libraries, technology node, and other variables. Overall, these accelerators are characterized by a common design. An architecture provides a set of homogeneous PEs, that are CPUs or general-purpose DSPs, commonly supporting a VLIW instruction set or a vector extension. Each architecture also includes an L1 code memory, an L1 data memory and a DMA engine to enable data transfers with greater memory levels. Peripherals (e.g., SPI) and greater memory levels (e.g., L2 or DDR) are accessible via off-chip communication channels.

Figure 5.1 shows the generalized architecture model considered in the rest of this work. It consists of a MCU host coupled with a multi-core mW-scale accelerator. The host is the main control unit of the full system, and it has the option to offload computation-intensive workloads to the accelerator. The link between MCU and accelerator uses the SPI protocol, a common interface for off-the-shelf MCUs which fully satisfies mW-scale power constraints. External sensors and communication channels are managed by the MCU, while data to/from the accelerator are stored in the external memory.

The accelerator is a parallel platform featuring a number n of PEs, that are fully independent cores supporting MIMD parallelism, in accordance with the template introduced in Section 1. In addition, the external memory is intended to store input/output data. A typical example of such IP could be a generic flash memory or a more specific frame buffer where various sensors place sampled data. It is accessible through the SPI, which provides a low-bandwidth, long-latency serial IO channel.

2.2 OpenVX advanced features

To recap what hast been introduced in Section 2, Listing 5.1 shows the typical structure of an OpenVX application:

- an execution context is created (line 1);
- a graph instance is created (line 2);
- each image is defined with a call to `vxCreateImage` (lines 3-4) or `vxCreateVirtualImage` (5-6), specifying size and type (e.g., RGB or grayscale);
- each kernel is added to the graph as a node with a call to `vx<KernelName>Node` (lines 8-9), specifying a list of one or more input images, a list of scalar parameters (e.g., a threshold) and a list of output images;
- the `vxVerifyGraph` function (line 13) checks the graph consistency;
- the `vxProcessGraph` function (line 15) executes the graph on the target device, using a specific framework.

```
1 vx_context ctx = vxCreateContext();
2 vx_graph graph = vxCreateGraph();
3 vx_image img0 =  vxCreateImage(ctx, <width>,
4                                <height>, <type>),
5 vx_image vimg0 =  vxCreateVirtualImage(ctx, <width>,
6                                <height>, <type>),
7 ...
8 vx_node node0 = vx<KernelName>Node(graph,
9                                <input0>,...,
10                               <param0>, ...,
11                               <output0>, ...);
12 ...
13 status = vxVerifyGraph(graph);
```

```
14 ...
15 status = vxProcessGraph(graph);
```

Listing 5.1: OpenVX program template.

OpenVX includes advanced features that allow for more dynamic behavior, such as *graph updates* and *node callbacks*.

Graph updates are dynamic modifications to a graph data structure following its first execution. This can be accomplished by nesting graph creation constructs (`vx<KernelName>Node` and `vxRemoveNode`, which removes a node from its parent graph) within conditional control flow in the program. When the original execution path is altered due to control flow, then the OpenVX standard requires a further verification stage before executing a modified graph. Listing 5.2 shows the typical structure of an OpenVX application using graph updates, extending the code of Listing 5.1 with additional lines. The graph is executed multiple times in a loop structure, and when an application specific condition is met (e.g., environment conditions are changing) the graph is modified accordingly (e.g., a set of nodes implementing a specific algorithm is replaced with a different one). An `else` clause or additional `if` blocks can apply alternative or additional modifications.

```
12 ...
13 status = vxVerifyGraph(graph);
14 while(<running_condition>)
15 {
16   status = vxProcessGraph(graph);
17   ...
18   if(<graph_update_required_condition>)
19   {
20     vx<KernelName>Node(graph, ...);
21     ...
22     vxRemoveNode(<node_var>);
23     ...
```

```
24     status = vxVerifyGraph(graph);
25   }
26   else
27     ...
28 }
```

Listing 5.2: Graph modifications

Node *callbacks* represent a mechanism to control the graph execution flow on the host side by specifying a function to be called after the execution of a particular node. Node callbacks are set through the `vxAssignNodeCallback` function with two parameters, the graph node and the callback function. Callbacks are intended to provide simple early exit conditions, based on the return value of the passed function. If `VX_ACTION_CONTINUE` is returned, the graph execution on the target device will continue. If `VX_ACTION_ABANDON` is returned, execution is unspecified for all nodes whose execution must follow the callback owner. In practice, the execution of the graph on the device may be aborted due to application constraints (e.g., a deadline was missed or an intermediate result was sufficient to take some decisions). Listing 5.3 shows the typical usage of a node callback.

```
12   ...
13   vxAssignNodeCallback(<node_var>, func);
14   status = vxVerifyGraph(graph);
15   status = vxProcessGraph(graph);
16   ...
17 }
18
19 vx_nodecomplete_f func(vx_node n)
20 {
21   ...
22   if(<exit_condition>)
23     return VX_ACTION_ABANDON;
24   return VX_ACTION_CONTINUE;
25 }
```

Listing 5.3: Node callback

2.3 OpenVX execution model and RTE

OpenVX was conceived for heterogeneous systems including a host processor and a parallel accelerator, however, it can be compiled and executed on any machine. Typically, programmers develop and test their vision applications on a development platform (e.g., an x86 workstation). Once the application code is debugged and tuned, it can be executed as-is on the target OpenVX installation for heterogeneous systems. When deployed to a heterogeneous system, an OpenVX graph is created and verified on the host, while graph execution is offloaded to the accelerator.

Generally, OpenVX relies on a *graph-based runtime environment (GB-RTE)*, which leverages a *graph interpreter* running on the accelerator side, that is a small software layer capable of reading an OpenVX graph description and orchestrating the execution of the corresponding kernels. Since local memory in mW-scale parallel accelerator is a scarce resource, it is important to design the OpenVX RTE with minimal memory footprint (RTE code and metadata). Table 1 reports the contributions to the memory footprint associated to the three main operations in an OpenVX program. The main contribution to RTE metadata footprint is the *graph data structure*, whose size depends on the application.

Stage	**Code footprint**	**RTE metadata**
Creation	API support	Graph
Verification	API support	Graph
Execution	Interpreter	Graph

Table 5.1: OpenVX program phases.

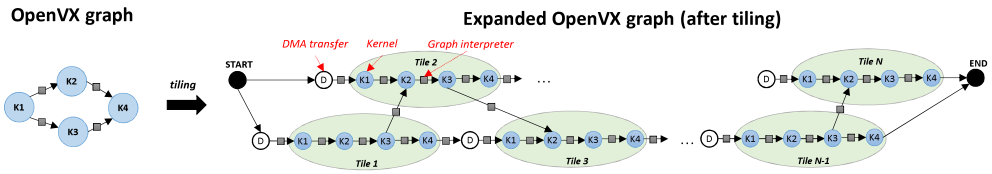


Figure 5.2: Example of an OpenVX graph and its expansion after tiling.

Data tiling is a common optimization for scratchpad-based architectures, used to enable high-locality computation on the fast L1 memory via explicit DMA transfers. The OpenVX standard incorporates tiling as the main technique to manage diverse memory hierarchies with a single program source. This technique is essential to execute a full graph with any image size on the target accelerators, but at the same time tiling policies affect the size of an OpenVX graph.

Figure 5.2 shows an example of how data tiling increases the size of a graph. Two node types are shown, DMA transfers and executable kernels. Points where the graph interpreter is invoked are also highlighted.

After tiling, the graph includes a number of nodes much larger than the number of application kernels, because the graph structure is replicated to consider different configurations (e.g., image borders, corners or inner tiles) and different target buffers (e.g., double-buffering) for each sub-graph generated by applying the tiling policies. In addition, nodes to program DMA transfers are added to the graph.

To further complicate the scenario, there are applications for which no tiling scheme is feasible, as they contain kernels that cannot be processed in parts (e.g., histograms) or because the tiling algorithm can't fit all the buffers in the SCM. In these cases the graph is automatically partitioned by the RTE into multiple sub-graphs during the verification stage. Inside each sub-graph tiling is applied and intermediate results at the sub-graph

frontier are saved on a temporary buffer out of the small L1 SCM (e.g., the L2 memory). An estimation of the graph footprint inflation due to data tiling can be achieved with this formula:

$$a * N_{intiles} * N_{nodes} + b * N_{tiles} + c \tag{5.1}$$

The terms are:

- a is the average size of a kernel node.
- $N_{intiles}$ is the number of input tiles.
- N_{nodes} is the number of nodes in the application graph.
- b is the average size of a transfer node.
- N_{tiles} is the total number of tiles (input + output).
- c is the total size of additional helper nodes.

Concerning the code footprint, the graph interpreter represents the main contribution. The graph interpreter is invoked each time a node is completed, marking the output dependencies as satisfied and looking for the next node to execute. A node is ready for execution when all its input dependencies are satisfied; to support a generic topology, a full graph visit is performed at each interpreter call, and this implies a full iteration on the node set. As a consequence, the graph expansion due to tiling does not have an impact on the code footprint for the interpreter. However, it does have an impact on the time overhead (and associated energy) for executing the interpreter more often.

3 Code Generation-based RTE

Applying Formula 5.1 to existing GB-RTE OpenVX implementations for multi- and many-core accelerators [29], the size required to represent a kernel node can be computed as the total memory requirement of actual kernel parameters plus a number of words (4 bytes) equal to the output dependencies. The size required to represent a DMA transfer node is a fixed amount (40 bytes) plus a number of words (4 bytes) equal to the output dependencies. Additional helper nodes (start, end) require from 96 to 128 bytes. As a quantification, applying this formula to the Sobel filter used in Section 4, which has 3 kernels and 64 input tiles, the final graph size is 29.68 KB.

Our alternative to GB-RTE is to replace the graph data structure and its management (i.e., the graph interpreter) with *control-code generation (CG-RTE)*. This has the potential to reduce the total RTE footprint (metadata) and to reduce the management overheads. The binary footprint of the generated code can be computed using this formula:

$$(a * N_{images} + b * N_{nodes} + c) * d \tag{5.2}$$

The terms are:

- a is the number of C lines required per each input or output tile, and its average value is 8.
- N_{images} is the number of defined images.
- b is the number of C lines required for each node. It can be computed as the average of $p_i + 2$, where p_i is the number of parameters required by node i.

- N_{nodes} is the number of nodes in the application graph.
- c is a constant number of C lines, and its value is 35.
- d characterizes the average density of assembly instructions per C line on the target platform. An average value valid for the experiments of Section 4 is 11.9.

Applying this formula to the same Sobel filter considered for GB-RTE, the final graph size is 25.88 KB, that is a 15% reduction in code size also for a very small graph. In practical case, $N_{intiles}$ is much greater than (N_{images}, and so the result of Formula 5.1 is greater than the result of Formula 5.2

Figure 5.3 describes the main steps of our approach. The program source is compiled on the developer workstation, and linked with a standard OpenVX RTE (libopenvx.so) and a node implementation (libXYZ.so) targeting the workstation environment. Testing and debug are performed on the developer workstation. These initial steps are totally equivalent to the standard OpenVX workflow.

In addition to that, to deploy the program on the target architecture we require the call to `vxProcessGraph` to be replaced with a call to `vxSaveGraph`. With this new function, the source is processed by a cross-compiling toolchain to generate two binaries: (i) a program for the target host, obtained by the original program linking a lightweight runtime called milliVX (libmillivx.a); (ii) a program for the target accelerator, obtained by linking a static version of the program graph generated via a code-generation approach to the kernel library implementation for the target accelerator (libXYZ.a). Dynamic features of OpenVX are preserved by this approach, since a proper support for graph updates and

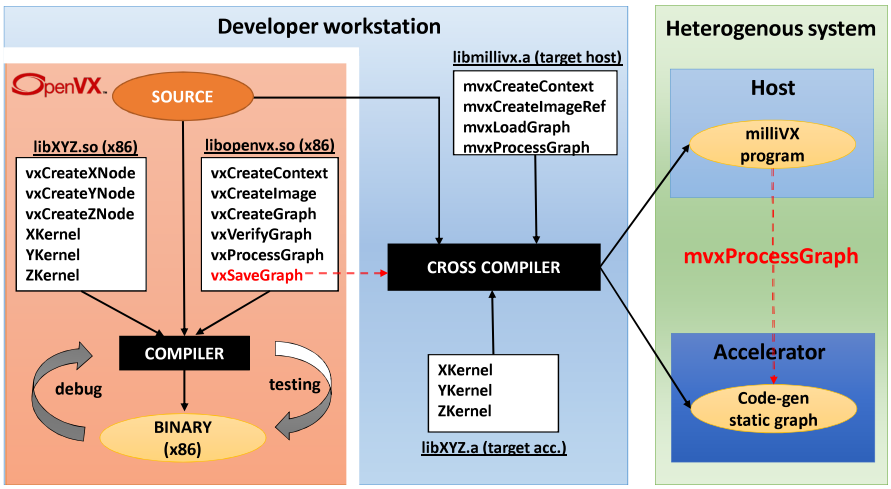


Figure 5.3: Developer workflow using our approach.

callbacks is provided by the milliVX RTE.

3.1 Extension for static graph support

Using the API extension, a call to `vxSaveGraph` produces a binary file, using a compilation toolchain to generate a sequence of intermediate artifacts:

1. *graph control function* – A C function is created for each OpenVX graph, including the code required to orchestrate the kernel executions and the DMA transfers specific of the graph instance.

2. *entry point function* – A C function is created as an entry point for the execution of the OpenVX application. It orchestrates the execution of multiple sub-graphs deriving from a single application.

3. *intermediate linked object* – An object file is generated by linking the single artifacts (i.e., control functions, entry point function, and kernel functions).

Code generation, compiling and linking steps are executed transparently by `vxSaveGraph`. This solution is based on a set of standard features common to modern compilation toolchains. In our implementation, we use the LLVM [101] toolchain libraries (libClang and libLLVM) to generate C code and translate it into LLVM intermediate representation (IR) artifacts.

Graph control function

A graph control function includes the capabilities provided by the union of a graph data structure and a graph interpreter from the GB-RTE. This approach enables the use of different algorithms for data tiling and kernel scheduling. Basically we reuse the algorithms provided by the verification stage of the baseline GB-RTE with minimum modifications, with the aim to perform a fair comparison limited to the execution phase. The code of a graph control function is generated on the basis of a common template, which is described by Algorithm 1.

Lines 1-6 contain the initialization phase of the algorithm. The local buffers are allocated in SCM, the first set of input DMA transfers is performed and the second one is programmed targeting a set of shadow buffers. The double-buffering technique enables the overlap between data transfer and computation. The loop in lines 7-15 drives the computation on all input sets, using data tiling. Line 8 updates the data structures containing the actual parameters for the kernel executed at iteration i, and it involves a limited number of fields changing between adjacent iterations. In lines 10-13, the algorithm flow awaits the completion of the required input transfers (input set $i+1$) and the output transfers related

```
1: – Allocate memory for SCM buffers
2: – Program the first set of input DMA transfers
3: – Initialize the data structures containing the kernel function
   parameters
4: – Wait the first set of input DMA transfers
5: – Program the second set of input DMA transfers
6: – Initialize to 0 the double-buffering state
7: for input tile set i do
8:   – Update kernel parameters for the current set i
9:   – Execute kernels (respecting the scheduling order)
10:  – Wait the previous set i − 1 of output DMA transfers
11:  – Program the current set i of output DMA transfers
12:  – Wait the next set i + 1 of input DMA transfers
13:  – Program the future set i + 2 of input DMA transfers
14:  – Update the double-buffering state for set i + i
15: end for
16: – Wait the last set of output DMA transfers;
```

Figure 5.4: Control function generation (pseudo-code).

to the buffers to be reused (output set $i - 1$). The resulting code implements the control logic of the graph for which it has been generated, and does not require complex data structures since all the instance-specific constants are encapsulated. Moreover, for each node there is a single copy of the data structures containing the kernel actual parameters, and the required fields are updated when executing different tiles. This requires a limited amount of stack memory area, just proportional to the number of nodes in the longest graph schedule. In practice, the average stack requirement for the standard OpenVX kernels is limited to 64 bytes per graph node.

Figure 5.5 shows an example of generated code, corresponding to the graph depicted in Figure 5.2. The regions are colored to highlight the same steps described in the code generation template. All

the uppercase identifiers are constant values, computed at code generation time on the basis of graph analysis results (e.g., considering a 80×80 tiling schema with overlapping borders `IMG0_TILE_WIDTH` could be 82). Consequently the resulting code is highly optimized for a specific instance, as the compiler can apply constant value optimization passes. Region 1 includes the code to allocate the required space in the SCM, and each single buffer is computed in terms of offset. Region 2 includes the first set of DMA transfers, one for each input image. The `dma_memcpy_2d` function program the DMA to perform a 2D transfer from the external memory to the SCM (`EXT2LOC`) specifying the full size of data (`IMG0_TILE0_SIZE`), the stride between adjacent lines (`IMG0_STRIDE`) and the line width (`IMG0_TILE0_WIDTH`). Region 3 contains the instructions to initialize the kernel-specific parameters, including width and height of the tile to compute. Region 4 includes a cumulative wait instruction for the DMA transfers of region 2. Region 5 includes the second set of DMA transfers, one for each input image. Region 6 initializes the variable `buffer_index`, used to maintain the double-buffering state. The subsequent regions (7-13) contain the code of the tiling loop, whose iterations correspond to distinct set of tiles as provided by the tiling algorithm. Regions 7 includes the code to initialize the parameters that change at every iteration (e.g., buffer location), so that in Region 8 all the kernels are invoked in the exact order provided by the scheduler algorithm. Regions 9-12 include the management code for the DMA transfers: (9) await the previous set of output transfers to guarantee the availability of the corresponding output buffers, (10) program the output DMA transfers for the last computed result, (11) await the input transfers of data required by the next cycle and (12) schedule

```
unsigned char *scm_memory = scm_malloc(BUFFERS_SIZE);                          1
unsigned char *scm_buffers[N_BUFFERS];
scm_buffers[0] = scm_memory;
scm_buffers[1] = scm_memory+B1_OFFSET;
...
dma_in[0] = dma_memcpy_2d(ext_data[0], ext_data[0],                            2
   scm_buffers[0], EXT2LOC,
   IMG0_TILE0_SIZE, IMG0_STRIDE, IMG0_TILE0_WIDTH);
...
K1_args_t args_node_0;                                                         3
args_node_0.src0Width = NODE0_TILE_WIDTH;
...
dma_wait(dma_in[0]);                                                           4
dma_in[1] = dma_memcpy_2d(ext_data[0]+IMG0_BUFFER_SIZE,                        5
    scm_buffers[0]+IMG0_TILE1_OFFSET, EXT2LOC,
    IMG0_TILE1_SIZE, IMG0_STRIDE, IMG0_TILE1_WIDTH);
buffer_index = 0;                                                              6
for(tile_index=0; tile_index<NTILES; ++tile_index) {
  args_node_0.src0 = scm_buffers[0] + ...;                                     7
  ...
  K1(&args_node_0);                                                            8
  ...
  if(tile_index > 1) dma_wait(dma_out[next_buffer_index]);                     9
  dma_out[buffer_index] = dma_memcpy_2d(ext_data[1] + out_offset[0],           10
    scm_buffers[1] + (buffer_index? IMG1_BUFFER_SIZE: 0), LOC2EXT,
    IMG1_TILE_SIZE, IMG1_STRIDE, IMG1_TILE_WIDTH);
  ...
  dma_wait(dma_in[next_buffer_index]);                                         11
  dma_in[buffer_index] = dma_memcpy_2d(ext_data[0] + in_offset[0],             12
      scm_buffers[0] + (buffer_index? IMG0_BUFFER_SIZE: 0), EXT2LOC,
      w*h*IMG0_CHANNEL_SIZE, IMG0_STRIDE, w*IMG0_CHANNEL_SIZE);
  ...
  buffer_index = next_buffer_index;                                            13
}
dma_wait(dma_out[1]);                                                          14
```

Figure 5.5: Control function code.

the next set of input transfers. Region 13 updates the double-buffering state. Finally, region 14 waits for the remaining DMA output transfers programmed in the last iteration of the tiling loop.

Figure 5.6 shows a graph representation of the control function code, which can be compared to the graph-based approach depicted in Figure 5.2. Multiple calls to the graph interpreter are replaced with two application-specific control code blocks, and the tiling policy is enforced using a loop.

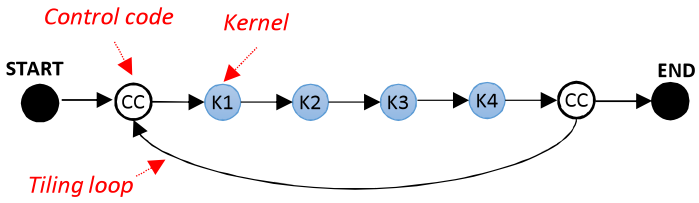


Figure 5.6: Graph representation.

Entry point function

An entry point function orchestrates the execution of the multiple sub-graphs derived by the tiling policies. In addition, the entry point function manages graph updates and node callbacks.

In practical cases, graph updates affect a limited portion of an OpenVX graph, otherwise it would be more convenient to create a totally new graph instance. Starting from this assumption, we extended the OpenVX context to be aware of modifications to the graph structure applied after one or more executions. When a single node or a connected subset is removed and replaced with another node or connected subset, our algorithm generates additional sub-graphs. All alternative paths in the original OpenVX graph must be explored and verified to generate all the code variants. For instance, Listing 5.2 is modified as depicted in Listing 5.4.

```
12 ...
13 status = vxVerifyGraph(graph);
14 #ifdef CODE_GENERATION
15 while(<running_condition>)
16 #endif
17 {
18 #ifndef CODE_GENERATION
19   status = vxProcessGraph(graph);
20 #endif
21   ...
22 #ifndef CODE_GENERATION
23   if(<graph_update_required_condition>)
24 #end
25   {
26     vx<KernelName>Node(graph, ...);
27     ...
28     vxRemoveNode(<node_var>);
29     ...
30     status = vxVerifyGraph(graph);
31   }
32 #ifndef CODE_GENERATION
33   else
34 #end
35     ...
36 #ifdef CODE_GENERATION
37   vxSaveGraph(graph, <filename>, <options>);
38 #endif
39 }
```

Listing 5.4: Graph modifications

A call to `vxSaveGraph` following all the modifications creates the additional graph control functions corresponding to the new sub-graphs and generates a new entry point function introducing *control flow variables*. These are integer variables that are passed as an input parameter to the entry point function. The actual value of a control flow variable discriminates what version of the related sub-graph must be executed. This

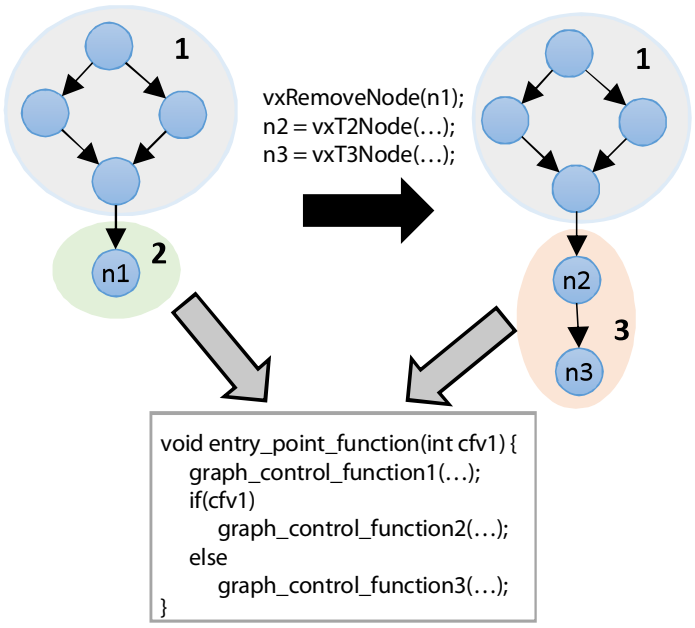


Figure 5.7: Example of graph modifications.

behavior is equivalent to having more variants of the same graph, and each control flow variable select a specific variant. Figure 5.7 shows an example of graph modification. Node n1 is removed, and nodes n2 and n3 are added. Consequently, two alternative sub-graphs are generated, the common sub-graph 1 and the alternative ones 2 and 3. The generated code for the entry point functions enables to switch between 2 and 3 on the basis of the control flow variable `cfv1` (the next section describes how to practically use this mechanism in the *milliVX* framework). When the algorithm fails in generating alternative sub-graphs (i.e., sub-graph deriving from distinct graph updates intersect), the framework generates totally distinct graphs. In this case a warning is generated to inform the programmer that graph modifications were too pervasive and subgraph generation was not possible, since this condition could increase the final binary size. This particular condition is related to complex polymorphic

behaviors that are not common of OpenVX applications, but nevertheless this corner case is correctly supported by our framework. Overall, this methodology totally preserves the semantics of the original program and ensures full code portability.

In our execution model, the support to callbacks is provided by means of a communication protocol between the host and the accelerator. A notification is fired to the host in the entry point function exactly after the execution of the sub-graph that includes the involved node, and the execution is suspended waiting for a message back from the host. The response message contains a Boolean value representing the continuation status; if false, the execution of the current binary on the accelerator is aborted. Since the node abstraction is not available in the *milliVX* RTE, a numerical identifier is returned for each callback. The described behavior is fully compliant with the OpenVX standard, which specifies that callbacks are not guaranteed to be called immediately after the node completes.

Intermediate linked object

An intermediate linked object is the final step in the binary generation flow. It is compiled by the toolchain back-end for the target architecture into a single binary file, applying link-time optimization (LTO) passes, such as basic inlining and dead-code elimination. In addition, we designed a new LTO pass to maximize the execution performance yet limiting the binary size. This pass forces the inlining of a kernel function ker_i when (i) it is invoked once (whatever its size) or (ii) it is invoked n times and

this property holds:

$$(n-1) \times memsize(ker_i) < \alpha \sum_{j=1}^{n} memsize(ker_j) \tag{5.3}$$

The parameter α is the percentage of the total kernel footprint (supposing no other inlining) that we could not exceed to inline the current kernel.

3.2 *milliVX* framework

In the context of our target mW-scale architecture, *milliVX* is a lightweight framework available to the MCU host to load a program binary corresponding to an OpenVX static graph and then offload its computation to the parallel accelerator. The *milliVX* API specification includes the following functions:

- `mvxCreateContext` – Create a lightweight context for the RTE. The implementation details are strictly dependent on the target platform.

- `mvxCreateImageReference` – Create a reference to an image location, providing a pointer. The address value is required to enable the graph binary to access input/output images. An image reference must be instantiated for each concrete image in the original OpenVX program. Virtual images just represent dependencies, and the generated code already handles these dependencies internally.

- `mvxLoadGraph` – Load a graph into the accelerator L2 memory, in the format provided by `vxSaveGraph`. This function returns a handle to the loaded graph.

- `mvxProcessGraph` – Start the execution of the graph on the accelerator. The required parameters are the graph handle returned by `mvxLoadGraph`, the graph input/output image references and the control flow variables. The control variables are set by the program logic with the aim to execute a specific graph variant.

- `mvxAssignNodeCallback` – Set a function callback. The required parameters are the callback identifier (provided by the extended OpenVX RTE) and the function to execute. The function could be the same provided in a standard OpenVX RTE, with the only difference that API function must be replaced with their equivalent in *milliVX*; in most cases, the access to a framework object is replaced with a direct memory access. The communication protocol between the host and the accelerator, used to manage the callback behavior, is handled by the *milliVX* RTE. The communication internals are based on platform-specific mechanisms (e.g., shared memory or communication channels), and also the synchronization can be achieved using alternative mechanisms (e.g., a software interrupt or a polling thread).

The structure of a *milliVX* program is much simpler than an equivalent OpenVX program. Creation and verification stages are performed by the extended RTE producing a static graph, and *milliVX* only handles the execution stage. A *milliVX* program can be automatically derived from the corresponding OpenVX code. Pointers to input/output images are provided as global external variables keeping the original names, and these symbols must be resolved at link time. The actual parameters for control flow variables when invoking `mvxProcessGraph` is derived by a static control flow analysis of the source code. Listing 5.5 shows the

structure of a *milliVX* application using callbacks.

```
14   mvx_context ctx = mvxCreateContext();
15   mvx_graph graph = mvxLoadGraph(<binary_location>);
16   mvx_image img0 =  mvxCreateImageReference(ctx,<ptr>);
17   mvxAssignNodeCallback(<callback_id>, func);
18   ...
19   status = mvxProcessGraph(graph, <image_references>,
20                            <control_flow_variables>);
21 }
22
23 mvx_nodecomplete_f func(int callback_id)
24 {
25   ...
26   if(<exit_condition>)
27     return MVX_ACTION_ABANDON;
28   return MVX_ACTION_CONTINUE;
29 }
```

Listing 5.5: *milliVX* callback support

4 Experimental results

This section describes a set of experiments performed to compare our approach to a standard dynamic graph-based framework. First, we measure the impact of code and runtime data footprint when comparing CG-RTE and GB-RTE. Second, we show how CG-RTE enables performance speedups due to link-time optimizations enabled by the approach. Third, we discuss how the reduced memory accesses due to graph removal enable important energy savings for CG-RTE. Finally, we discuss the impact of tiling on reducing the bandwidth pressure on the slow SPI channel.

4.1 Setup

The benchmarks used for experiments include a set of representative CV kernels for constrained embedded systems:

- *Sobel* is a gradient-based edge detector (nodes: Sobel 3×3, gradient magnitude, thresholding);
- *FAST9* implements FAST9 algorithm [66] (nodes: FAST9, find maxima, non-maxima suppression);
- *Pyramid* creates a set of scaled and blurred images (nodes: Gaussian blur, half scale, ... repeated 3 times);
- *Canny* implements an edge detector algorithm (nodes: Sobel M×N, element-wise norm, phase, non-maxima suppression, edge tracking by hysteresis);
- *Harris* implements a corner detector algorithm (nodes: Sobel M×N, Harris score computation, Euclidean non-maxima suppression, corner lister);
- *NCC* is an algorithm to detect abandoned/removed objects in a set of adjacent video frames [64] (nodes: NCC filter, erode);
- *Disparity* computes the stereo-matching disparity between two images (nodes: subtraction, multiplication, integral image, area sum, disparity computation).
- *CNN* is a convolutional neural network [102] including four layers made of 48 total nodes (node types: convolution, add, max-pooling).

The reference image size used for experiments is VGA, typical of ultra-low cost imagers.

As a reference platform for our experiments we use accurate simulation models for a heterogeneous system coupling a MCU host with a PULP multi-core accelerator, based on the open source tool ADRENALINE [29]. In our target platform, we include an external frame memory (FM) intended to store input/output images, while L2 is dedicated to code and run-time data. FM is an off-chip component, as its size could scale up to several MBs depending on the target image format. The system is configured as follows:

- **Host:** STM32-L476 MCU [103] – number of cores = 1 (ARM Cortex M4), core frequency = 26 MHz
- **External memory:** memory size = 1 MB, access latency = 50 cycles, bandwidth = 0.125 bytes/cycle
- **PULP v3 cluster:** [96] number of cores = 4 (OpenRisc OR10N), core frequency = 200MHz, SCM size = 64 KB, L2 size = 128 KB, L2 access latency = 5 cycles, L2 bandwidth = 4 bytes/cycle

The power consumption of a PULP cluster in 28 nm FDSOI technology [104] (running at 200 MHz, with a voltage supply of 0.7 V) is 9mW. The power consumption of the SoC periphery, IO pads and support circuits is around 2mW. The bandwidth and latency to the external memory are modeled after those of a SPI interface providing 100 Mbit/s transfers @ 100 MHz. For realistic performance and power consumption measurement, the simulation platform has been augmented with a performance monitoring unit that is used to measure active and idle cycles

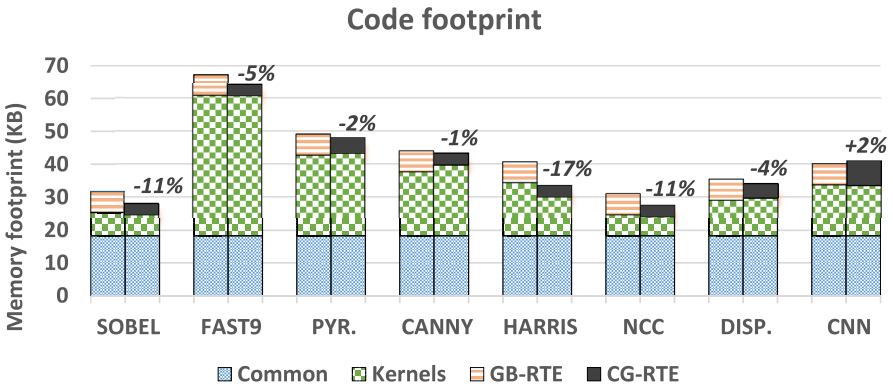


Figure 5.8: Code footprint (GB-RTE vs CG-RTE).

for cores, DMA, interconnects and external memory access. Power numbers for the host + SPI are pre-characterized with real measures on a STM32-L476 MCU. Leakage and dynamic power numbers for the PULP accelerator are extracted from a post-layout back-annotated timing and power analysis (PULP v3 chip @ 200 MHz). To compile the benchmarks, we used the ARM Sourcery Linux GNU toolchain (version 4.8.2) for the ARM Cortex-M target and the OR10N LLVM/Clang toolchain (based on LLVM 3.7) for PULP.

4.2 Memory footprint

Figure 5.8 compares the code footprints on GB-RTE and CG-RTE, highlighting the percentage savings of the second solution. Both RTEs allocate code and runtime data structures on the L2 on-chip memory. *Common* includes a set of low-level primitives for parallelism management that are used by both runtime environments. The variations on kernel footprint are related to the degree of code optimization enabled at compile and link time by the two approaches. In GB-RTE the kernels are standalone shared objects, and they are dynamically loaded at execution

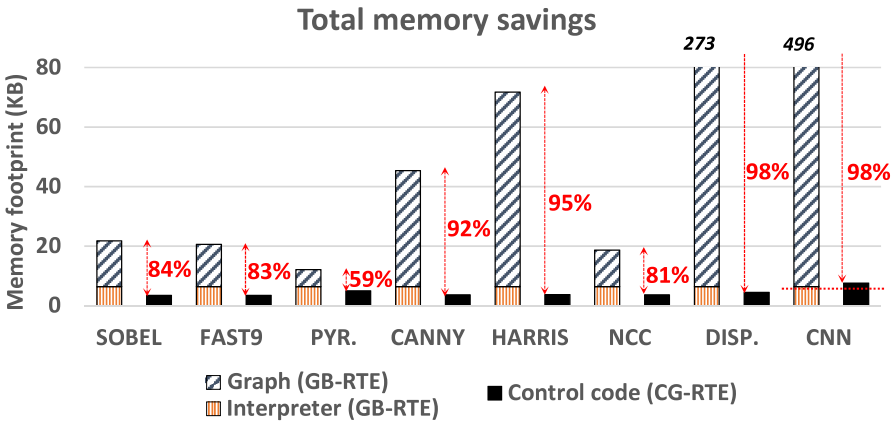
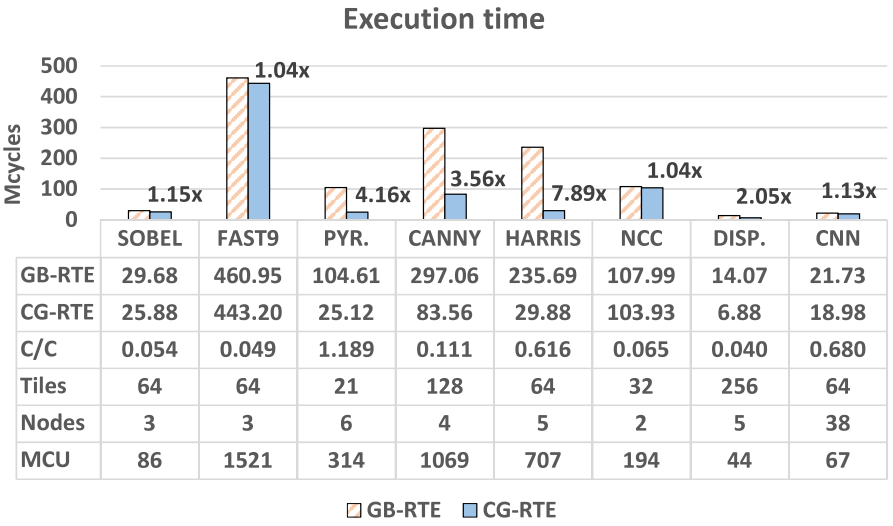


Figure 5.9: Total memory savings (CG-RTE vs GB-RTE).

time. Consequently, their total memory footprint is the sum of all kernel binaries. In CG-RTE the kernels are merged with the generated control code at link time to produce a single binary for the accelerator. This allows to enable aggressive LTO passes in the toolchain, with an impact on the code size. In the benchmarks the code size is increasing in *Canny*, which contains multiple inlined instances of the same kernels.

Figure 5.9 compares the memory footprint of CG-RTE and GB-RTE. The figure also reports percent L2 memory savings using CG-RTE instead of GB-RTE (these gaps are highlighted by vertical dashed lines). GB-RTE includes the graph interpreter (Section 2.3), that is independent of the executed benchmark. CG-RTE includes the control code generated for the specific benchmark (Section 3.1). The minimum difference between the runtime supports is variable, and in general CG-RTE could exceed GB-RTE. This effect is evident for CNN, which has a high number of kernels whose orchestration requires more lines of generated code w.r.t. other benchmarks. A horizontal dotted line highlights this overhead (about 1.30 KB). In any case its value is always lower than the sum of GB-RTE and the corresponding runtime graph. Overall the removal



	SOBEL	FAST9	PYR.	CANNY	HARRIS	NCC	DISP.	CNN
GB-RTE	29.68	460.95	104.61	297.06	235.69	107.99	14.07	21.73
CG-RTE	25.88	443.20	25.12	83.56	29.88	103.93	6.88	18.98
C/C	0.054	0.049	1.189	0.111	0.616	0.065	0.040	0.680
Tiles	64	64	21	128	64	32	256	64
Nodes	3	3	6	4	5	2	5	38
MCU	86	1521	314	1069	707	194	44	67

Figure 5.10: Execution time (GB-RTE vs CG-RTE).

of the RTE runtime graph is the major contribution to on-chip memory saving, that is a primary goal of our work.

4.3 Execution time

Figure 5.10 reports the execution time of the benchmarks for both runtime versions. The speed-up of CG-RTE over GB-RTE varies from 1.04 to 7.89, and this behavior depends on three main factors. First, it is proportional to a *C/C factor*, which includes the impact of the computation intensity of the kernel in terms of computation/communication ratio. Second, it is proportional to the *number of tiles*, as the overhead introduced by graph interpretation in GB-RTE is higher. Third, it is inversely proportional to the *number of nodes*, as the overhead of generated control loop in CG-RTE increases with this metric. The data table in Figure 5.10 reports these factors, and the resulting speed-up can be computed by the formula $C/C * Tiles/Nodes$. The resulting speed-up is

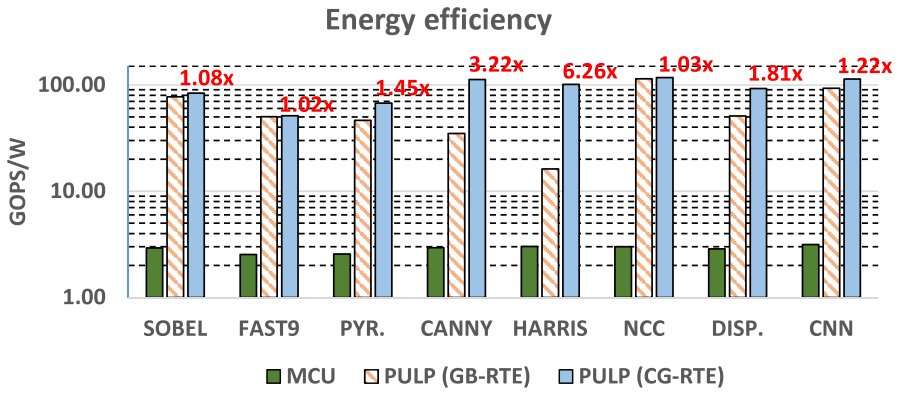


Figure 5.11: Energy efficiency of the STM32-L476 host compared to the PULP accelerator.

an additional benefit of our approach, mainly due to the aggressive link-time optimizations that are possible in the CG-RTE runtime (described in Section 3.1). The execution time on the MCU is also reported for comparison.

4.4 Energy efficiency

The reported MCU setup implies an average power consumption of 8.64 mW, close to the 8.10 mW value computed for the PULP accelerator. Considering this operating point, Figure 5.11 shows the energy efficiency of both OpenVX RTEs on the mW-scale accelerator with respect to the execution on the STM32-L476 MCU. To make a fair comparison, we compiled the optimized code generated by CG-RTE for the MCU target, using an intermediate C representation which includes advanced inlining. In addition, we used a basic instruction set for both cores to avoid the effects of vectorization or special-purpose acceleration. Overall, the energy efficiency of the MCU is two order of magnitude less than the one measured on the accelerator. Figure 5.11 also reports the ratio between the energy efficiency of CG-RTE and that of GB-RTE. On average, CG-RTE

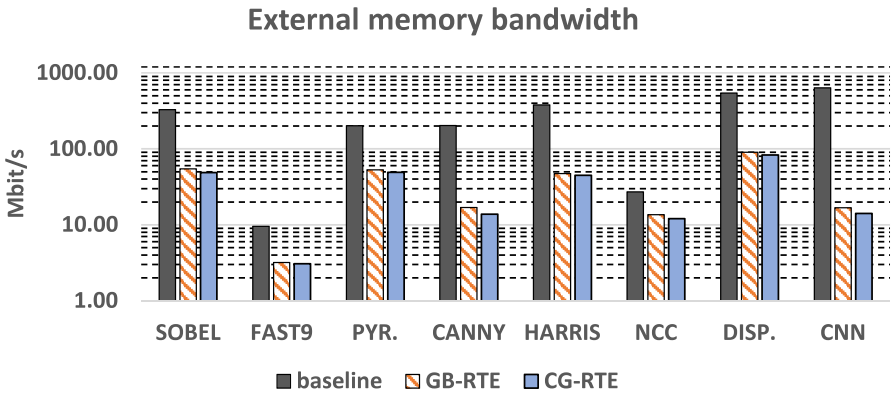


Figure 5.12: Frame memory bandwidth.

improves energy efficiency by 2.14×. This increase in energy efficiency is mainly due to the lower number of executed instructions and to the reduced number of L1/L2 accesses to runtime data which are replaced with cheaper control instructions (ALU operations).

4.5 Bandwidth reduction

Figure 5.12 shows the bandwidth required by both RTEs compared to a baseline implementation that access the external memory to store all the intermediate results. The bandwidth is computed as a ratio between the memory traffic required to compute a single image and its corresponding execution time on the accelerator. This consistent reduction enables the use of a low bandwidth SPI serial memory, while the access latency is hidden by double-buffering. There is no significant difference between GB-RTE and CG-RTE regarding the access patterns on the external memory. The bandwidth requirements of CG-RTE are lightly lower since the tiles are typically greater (L1 memory does not contain a reserved area for RTE data) and consequently the border effects are less evident.

Conversely, the bandwidth requirements of L2 and L1 memories are

affected. In GB-RTE a runtime graph must be read from L2 and written to L1 to be used by the graph interpreter. This amount of data is equal to the size of the runtime graphs, as reported by Figure 5.9. Using code generation these accesses to L1 and L2 are totally removed, as runtime parameters are directly encoded as instruction immediates with no additional redundancy.

5 Related work

The role of OpenVX to achieve performance optimization at system level has been initially highlighted in [71]. Its execution model assumes a graph-based application description. This is a very common approach in literature, and it has been extensively used to derive foundational models such as task graphs [105] and data-flow graphs [106]. The semantic of OpenVX defines a *dependency graph*, that is a structure that describes a partial evaluation order among kernel nodes. This approach is common to other modern programming models, such as OpenMP 4.0 tasking [107] and Intel TBB library [108]. The OpenVX standard has been supported by several major industries interested into CV acceleration on parallel computation devices, such as NVIDIA, AMD and Synopsys [109].

Alternatives to OpenVX are Halide [54] and HIPAcc [70], which allow programmers to specify a functional description with a domain-specific language. However these solutions present major limitations when applied to generic CV algorithms, in particular (i) irregular data patterns are not supported, (ii) composability of software modules is limited to pipeline patterns, and (iii) schedule management requires to write platform-specific code. OpenCL [61] allows applications to use pre-built

binaries, or alternatively to load and compile the program source at runtime. We apply the first method on the host side to load a pre-built OpenVX application, but in our optimized approach the source code for the accelerator is automatically generated from an OpenVX program.

The principle of code generation has been applied effectively in several contexts. In the context of Domain Specific Embedded Languages, code generation is commonly used to transform high-level patterns and structures into efficient parallel code for different architectures, such as CPUs [110], GPUs [111] or DSPs [112]. Machine learning techniques can be also used to generate efficient code for a specific algorithm [66]. Another technique which is strictly connected to code generation is *partial evaluation*. A computer program can be modeled as a mapping of input data into output data. A new mapping (i.e., a new program) can be obtained by removing from the input space all the dimensions corresponding to static inputs that are totally known at compile time. This is the principle that we have considered in this chapter, initially introduced by the first Futamura projection [113] in the context of code interpreters. We have extended this basic principle in two ways, by generating the code at the runtime level of a meta-model (that is the OpenVX program executing on the developer workstation) and by supporting the dynamic aspects of the original execution model with specific control code.

OpenVX support has been provided for different devices. The Khronos website [23] provides a sample implementation of the OpenVX specification targeting x86 architectures. VisionWorks [114] is software development toolkit that implements the OpenVX standard, targeting CUDA-capable GPUs and SOCs. Another OpenVX implementation supports

the PAAG array processor (Polymorphic Array Architecture for Graphics and image processing) [115], a polymorphous architecture specifically tailored for graphics rendering and image processing. An OpenVX framework tailored for low-power many-core accelerators has been presented in [27]. Most of these solutions are characterized by a power consumption from 500 mW up to 5 W with no specific memory limit, while the techniques described in this chapter are intended for heavily-constrained mW-scale devices.

State-of-the-art mW-scale MCUs (e.g., STMicroelectronics *STM32-L476* [103], SiliconLabs *EFM32* [116] and Texas Instruments *MSP430* [117]) already target a power budget lower than 50 mW, but they cannot guarantee high computing performance for the embedded vision domain. To bridge this gap some MCUs provide fixed-function hardware blocks [118] [119] or partially programmable accelerators [120], but their programmability is very limited and they cannot support a full OpenVX framework. In this chapter we have preferred a more general approach, coupling to the MCU a fully programmable accelerator able to execute diverse and complex workloads with a limited budget for silicon area and power consumption. Some multi-core MCUs [121] [122] are available, but they employ multiple cores with the objective to save power distributing heterogeneous tasks to specialized units.

Today the market offers several products that can be included in the class of mW-scale parallel accelerator, mostly in the segment of licensable IP cores. DesignWare EV52 and EV54 processors [94] by Synopsys integrate two or four 32-bit ARC HS cores with up to eight programmable accelerators optimized for CV and convolutional neural networks. CEVA-XM4 processor [93] is based on a general-purpose DSPs, with a VLIW

support up to eight parallel operations on 4,096 bits. Cadence IVP processor [89] is based on the configurable Xtensa CPU/DSP, supporting three parallel operations on 512 bits. Considering platform proposed by academic institutions, possible candidates to the role of multi-core accelerator are *Centip3de* [95] and *PULP* [96]. Centip3de is a clustered-based fabric of Cortex M3 cores, while PULP presents a similar design based on OpenRISC cores. These solutions rely on near-threshold and parallel computing to increase performance and energy efficiency. We used PULP as a target, for two main reasons: (i) its architecture is representative of our device class, and (ii) a virtual platform with an OpenVX RTE was already available for tests and comparisons. To the best of our knowledge, no optimized OpenVX support is provided for any platform including a MCU and a mW-scale parallel accelerator.

6 Conclusion

In this chapter we propose an alternative and novel approach to provide OpenVX support in heterogeneous systems including a MCU and a parallel accelerator. Our main contributions are an extension to the original OpenVX model to support static management of application graphs, and the definition of the *milliVX* specification, which provides a lightweight support to execute static graphs in a resource-constrained environment, without renouncing the dynamic features provided by OpenVX. Experimental results show that our approach drastically reduces the memory footprint (-68%) and the required bandwidth (-10%). Moreover, there is an average 3× execution speed-up and a 2× energy efficiency compared to a baseline implementation. From a theoretical point of view

our approach is fully scalable w.r.t. the number of nodes and processing elements in the system, with the only limitation given by the system resources.

Chapter 6

A new frontier: supporting approximate computing on mW-scale parallel accelerators

1 Introduction

Embedded systems targeting ultra-low power (ULP) markets are mostly implemented with single-core microcontrollers [123] [124] [125]. Usually, the computational power delivered by such systems within the admitted power envelope is sufficient for satisfying the very low demand of the target applications. This assumption is less and less valid for today's deeply embedded sensing applications [126], whose computation requirements grow to match the complexity of increasingly sophisticated workloads. To match conflicting requirements for energy consumption and performance, near-threshold multi-core systems have been recently proposed [127] [95] [128] in application fields such as industrial automation, wearable consumer electronics, human-computer interfaces and pervasive

video infrastructures. By joining parallelism with near threshold computing, these systems are able to provide more than one order of magnitude increase in energy efficiency [129] preserving the performance target.

In the context of energy efficient computing platforms operating in near-threshold, the memory system emerges as one of the most critical components, burning more than 50% of the total chip power [130] [32]. While standard voltage scaling techniques can be to some extent applied to reduce energy, their aggressive use is not possible for memory energy reduction as operations on standard six-transistor static RAM (6T-SRAM) cells become unreliable at low voltages due to the lack of sufficient static noise margin and read/write stability [131]. On these premises, the design must take into account a specific voltage domain to be kept at higher voltage that includes the memory system, and energy requirements become even more memory-dominated. Advanced design techniques have been proposed to improve the performance of SRAM banks at low voltage [132] [133] [134], but overall their adoption is difficult due to area and cost considerations [135]. A promising solution consists of adopting a *heterogeneous memory architecture* [136] [34], which includes both 6T-SRAM and standard-cell memory (SCM) banks.

The key idea of this approach is that SCM banks and 6T-SRAM banks can be powered at different voltage levels. At high voltage both memories operate reliably, but the SCM consumes significantly less energy. At low voltage, the 6T-SRAM has an associated probability of error (flip-bit) if it is read or written, while the SCM remains reliable and more energy efficient. Effectively managing a hybrid memory system requires techniques to partition data in a way that minimizes energy consumption. For example, greedy-allocation heuristics can be applied at the compiler level

to place most frequently accessed data into the memory that maximizes the metric of interest, e.g., energy efficiency [31], predictability [137] or performance [138].

In recent years approximate computing has emerged as a promising approach to design energy-efficient digital systems working at unreliable voltage levels, ranging from functional units [139] to interconnect [140] and memory systems. The notion of approximate computing [141] refers to a set of techniques ranging from programming language- to transistor-level, with a common aim to allow computing systems to save energy to the detriment of the *quality* of the computed results. Approximate computing is a promising approach in the ULP domain when applications exhibit inherent tolerance to errors [142] [143] [144].

In this chapter we propose a novel HW/SW approach to design energy-efficient ULP systems which combine the key ideas of hybrid memory designs and approximate computing. At the architecture level, we design a mechanism to split multi-byte data that is "tolerant" to approximation into multiple memory banks. The most significant bits (MSBs) of a word are stored in the SCM, while the least significant bits (LSBs) are stored in the 6T-SRAM. Both memories can be safely powered at the lower voltage level: here the SCM operates reliably, while the probability of error on the 6T-SRAM is guaranteed to be tolerated by the computation (i.e., the error is bound to the LSBs). Our experiments demonstrate that this approach provides much better precision than just dropping the LSBs. More specifically, we provide an upper bound to the accepted error for each application included in the benchmark suite, and we show that our approach complies with these bounds with a safety margin.

To expose the hardware extensions at the software level, we introduce

appropriate source code annotations used to specify what regions of code and what variables are tolerant to approximation. We choose to extend the OpenMP programming model, since this approach enables programmers to simply handle both parallelism and approximation efforts by annotating a C program with pragma directives. The annotations are processed by a compiler pass that implements an allocation heuristic which places data into one of the available logical memory areas (SCM, 6T-SRAM and split) according to their tolerance to errors. In addition, since different variables might be accessed in different program regions, we extend this heuristic to also take into account live ranges of tolerant variables. At the hardware level the 6T-SRAM memory is partitioned into multiple, independent voltage domains (1, 2 or 4), and the heuristic allocates variables with non-overlapping live ranges into distinct domains to allow for lowering the voltage when a variable is not accessed in the program.

The proposed techniques have been implemented in the parallel ultra-low-power platform (PULP) [128]. PULP is a scalable, clustered parallel computing platform that features a parametric number of processing elements (PE) per cluster, sharing a multi-banked tightly coupled L1 data memory (TCDM), acting as a scratchpad. At the HW level, our proposal focuses on energy saving techniques for the TCDM, while the PEs work at the most energy efficient operating point. To implement and validate the HW extensions required by our approach we used cycle-accurate simulation models, back-annotated with energy and performance numbers taken from a silicon implementation of the baseline system-on-chip (SoC) with 28 nm ultra-thin body and buried oxide fully depleted silicon-on-insulator (UTBB FDSOI) technology. Experimental results demonstrate

that our approach can reduce the energy consumption of the memory system by 47% on average (for a set of real-world benchmarks) compared to the baseline SoC. Focusing on the whole-system energy, our technique allows on average 27% savings and outperforms other solutions. We finally show that our approach is fully compliant with a set of realistic accuracy levels deriving from practical constraints, which assesses the effective usability of the described techniques in real-life applications.

2 Related work

In the past few years approximate computing has been considered a promising approach in different research areas [145] [146]. Many studies have also pointed out that approximate computing is an amenable solution for applications in the ULP domain [142] [143] [144], and this facilitates the translation of such algorithms into energy-efficient hardware implementations.

Circuit level [147] [148] and *architecture level* [149] [141] techniques have been used to to reduce overall energy consumption to the detriment of numerical accuracy. However, different research works observed that most of the energy consumed by ULP systems is spent on on-chip memory [130] [32]. In addition, memory is the primary source of faults, while logic components are typically more robust. Starting from these considerations, our approach considers the use of two voltage levels for a set of 6T-SRAM domains, with the aim to achieve a significant power reduction with a disciplined relaxation to approximate results. The rest of our architecture is designed to work safely at low voltage, and this assumption simplifies the overall model with a minimum loss of generality.

Raising power concerns to the *software level* enables a range of energy savings opportunities at OS [150] and application level [151] [152] [153] [154]. These approaches require approximate code transformations with the aim to modify the application code to support the management of performance/accuracy tradeoffs. The extensions to the compiler toolchain required by our framework can be implemented at a higher level of abstraction w.r.t. the cited approaches. A specific support by the OS is not required, conversely the requested features can be provided by a lightweight runtime layer. Moreover, we introduce a minimum set of preprocessor directives to write error-tolerant parallel programs using an annotation mechanism that is of immediate use to embedded C/C++ programmers. Note that our approach could be considered orthogonal to other approximation techniques, since the programming model frontend can be decoupled by the approximation support, which includes the hardware design, the runtime layer and the compiler support. In this perspective the cited tools could be extended to leverage our approximation support by providing additional directives/keywords.

Approaches based on hardware design of *approximated memories* can be classified into two main categories, that are custom [155] and domain-specific [156] [136]. These solutions are tailored for specific applications or algorithms and achieve significant energy savings, but overall they lack the hardware and software support required to implement a general-purpose application. Using our platform we provide dedicated hardware, runtime features and compiler support to fulfill this goal.

Different approaches have been proposed to improve the performance of *SRAM* at low voltage [132] [133] [157]. A different approach to the problem is to implement low-voltage memory structures relying on

standard-cell memory (*SCM*) [134]. All the described approaches cause serious overheads in terms of area, leakage and dynamic power consumption (at same supply voltage) with respect to standard 6T-SRAM [135]. Hence, their adoption for the implementation of the whole memory system is impractical, due to area and cost considerations. Frustaci et al. [158] explore the use of approximate SRAM banks in the context of error-tolerant applications, at the cost of the occurrence of read/write errors in the least significant bits of data. Although this technique is effective, it requires the design of custom SRAM banks featuring deep circuit-level optimizations, which leads to a low technology portability. Our approach leverages standard 6T-SRAM cells that can be realized with any memory generators provided by silicon vendors, and SCM that can be implemented with standard semi-custom design flows relying on industrially qualified standard-cells for implementation.

Recent works [159] [160] propose statistical techniques to measure the program response to injected approximation and derive the behavior of code and data. Compared to a programmer-annotated version, these techniques can lead to significant errors in some use cases, by marking as approximable a variable that is not tolerant for specific execution paths. Other works propose the adoption of *emerging technologies* to realize approximate memory cells, such as RRAM [161] and memristors [162]. These are promising approaches for the future, but today their application is limited to specific domains (e.g., neural networks) and requires a custom design. The solution proposed in this chapter is totally general and can be directly applied using standard design flows provided by tool vendors.

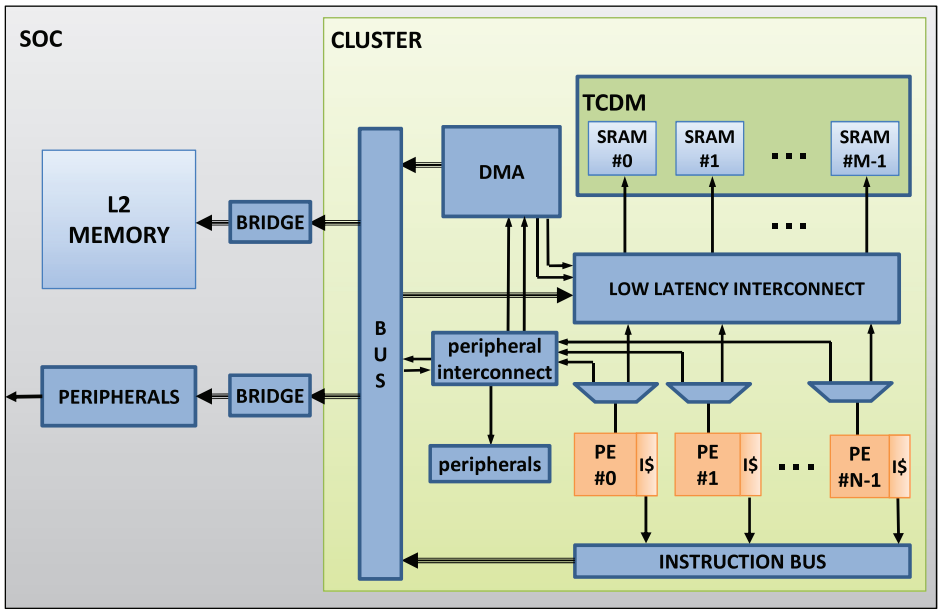


Figure 6.1: PULP architecture with hybrid L1 memory.

3 Hardware Architecture

ULP systems are largely based on microcontrollers featuring simple, cache-less cores (e.g., Cortex M0 or M4), coupled to simple support for power management and a standard set of peripherals. The parallel processing ultra-low-power platform (PULP) [32] [34] aims at providing a significant boost to the peak performance that ULP systems can achieve by coupling the multi-core paradigm to the most advanced FDSOI design technology and associated techniques for energy efficiency (near-threshold computing, body biasing, etc.) [35]. The following subsections describe the baseline PULP platform (Section 3.1) and the extensions to the memory system introduced by our work to support computation approximation (Section 3.2).

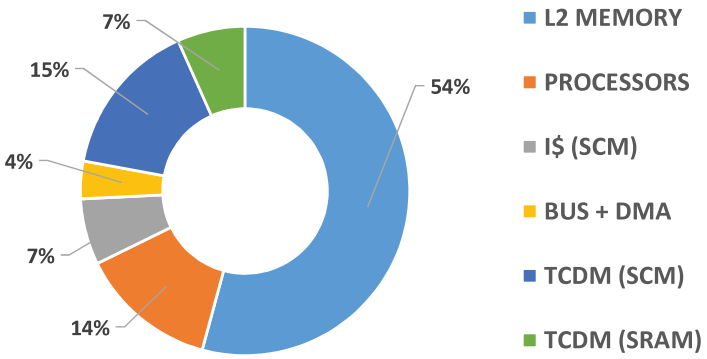


Figure 6.2: Breakdown analysis of the PULP SoC area.

3.1 PULP with hybrid L1 memory

The PULP instance considered in this chapter consists of a single cluster featuring 8 PEs and a TCDM composed by 16 6T-SRAM banks of 4KB each (64KB total) and 16 SCM banks of 1KB each (16KB total), plus 256KB of L2 memory. Each core features 1 Kbyte of I$ implemented with SCM, hence reliable down to 0.5 V. Three voltage domains are considered: i) the SoC domain includes the L2 memory and peripherals; ii) the cluster domain includes the PEs the SCM, the DMA and the cluster interconnect; and iii) the 6T-SRAM banks. A 28 nm UTBB FDSOI (STMicroelectronics technology) implementation of the platform in this configuration can operate at 20 MHz @ 0.5 V. We extend this baseline platform to support computation approximation in strict cooperation with the programming model. Figure 6.2 shows the contributions of the hardware components to the total area of a PULP SoC, that we consider a baseline configuration for this work.

Voltage [V]	0.50	0.55	0.6	0.65	0.7	0.75
P(bit-flip)	0.0037	0.0012	0.0003	5.24e-5	4.35e-6	6.16e-8

Table 6.1: Probability of bit-flip errors in 6T-SRAM.

3.2 TCDM Reliability Extensions

On-chip memory is traditionally implemented with 6T-SRAM banks working in super-threshold operating region. When operating close to the threshold voltage, SCM has demonstrated a better tradeoff between reliability, energy efficiency, area and portability among technology nodes [134]. In particular, although 6T-SRAM cells provide a much better storage density than SCM ($\sim 3\times$), SCMs are reliable over all the operating voltage range of the architecture (0.5 – 0.8 V) [155]. Accessing 6T-SRAM at a voltage lower than 0.8 V may results in a flip-bit error, as shown in Table 6.1. These values are derived executing the test patterns on PULP's silicon prototypes at different voltages using an Advantest SoCV93000 tester system. The program performs 10^{10} memory accesses on the 6T-SRAM memory area of PULP chips [34], and probability is computed from the error ratio.

SCM can thus operate at the same low voltage of the logic in a reliable way, with the key benefit of providing much smaller energy/access ($\sim 4\times$) [163]. Based on these observations and on the evidence that we cannot afford to build the entire TCDM with SCM, we propose a hybrid L1 memory design. We organize the TCDM in two different **physical memory areas**, including 16KB of SCM and 64KB of 6T-SRAM. Hereafter we consider alternative scenarios which provide a different number of voltage domains for the 6T-SRAM area, corresponding to the following memory layouts described in Table 6.2. Considering this size of the

6T-SRAM region (64KB), a further partitioning would produce a significant overhead, since the total area would be heavily dominated by the periphery and embedded power switches in small memory cuts.

A set of *reliability management units* (RMUs) are introduced in the path between the interconnect and the TCDM. These are simple combinational logic blocks that allow to remap the physical address range of the TCDM into three distinct **logical memory areas**:

- *SCM* – mapped in the SCM physical memory area (reliable at any operating point);
- *6T-SRAM* – mapped into the 6T-SRAM physical memory area (reliable at 0.8 V), include multiple regions corresponding to distinct voltage domains;
- *split* – MSBs are mapped in the SCM physical memory area and LSBs in the 6T-SRAM physical memory area.

Figure 6.3 shows the new TCDM design and the defined memory areas for the memory layout including four 6T-SRAM voltage domains. Level shifters are required at the boundaries between voltage domains, i.e., when the 6T-SRAM is operating at 0.8 V and the logic at a lower voltage (0.5 V). Considering the power breakdown of each memory bank, the overhead of the level shifters is $< 1\%$. The impact on critical path is also negligible, mitigated by the fact that delay is dominated by SRAM

6T-SRAM domains	**SCM cuts**	**6t-SRAM cuts**
1	64 128×16	16 2048×16
2	64 128×16	32 1024×16
4	64 128×16	64 512×16

Table 6.2: Memory layout changing the 6T-SRAM voltage domains.

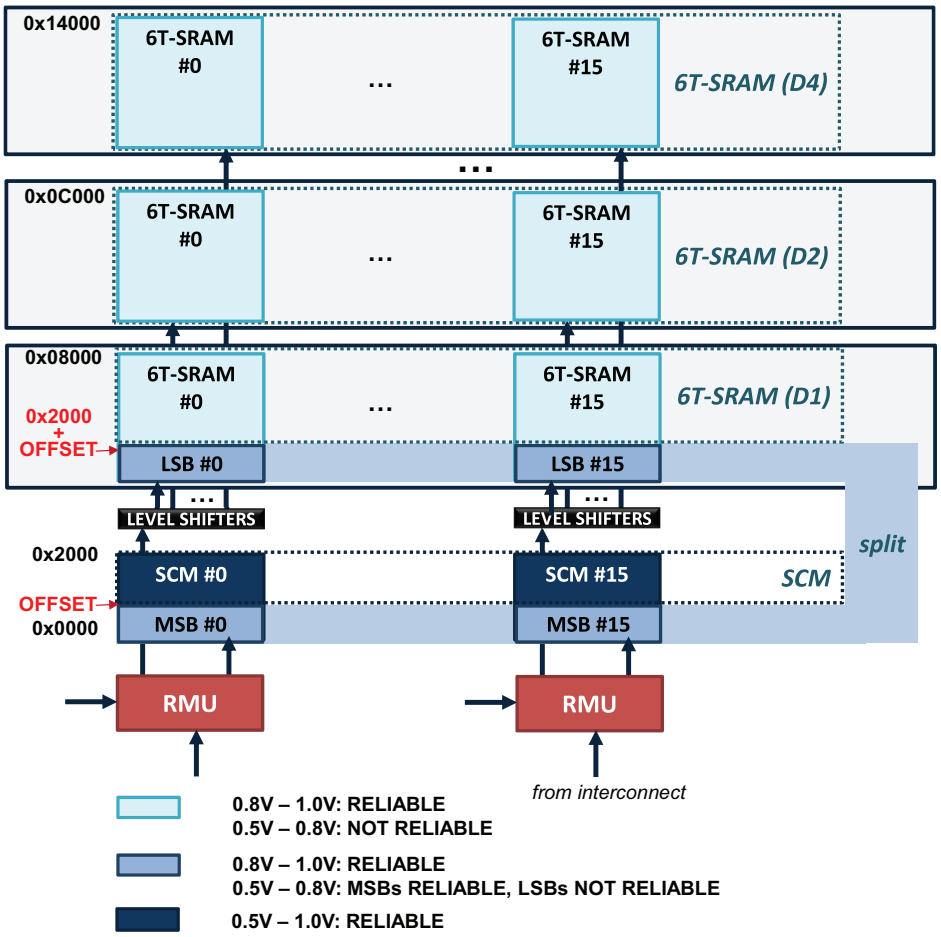


Figure 6.3: Hybrid TCDM organization.

banks when they operate at low-voltage (0.5 V), while the paths toward TCDM is not critical when the 6T-SRAM area operates at high voltage (0.8 V).

The RMUs provide access to the *split area* only at word or half-word level. In both cases, MSBs (upper 16 or 8 bits) are placed into SCM cells, while LSBs (lower 16 or 8 bits) are placed into 6T-SRAM cells. Figure 6.4 depicts the remapping of physical addresses when accessing the split area at word level. This implies that errors can show up only on LSBs at voltage levels below 0.8 V, guaranteeing a correctness threshold

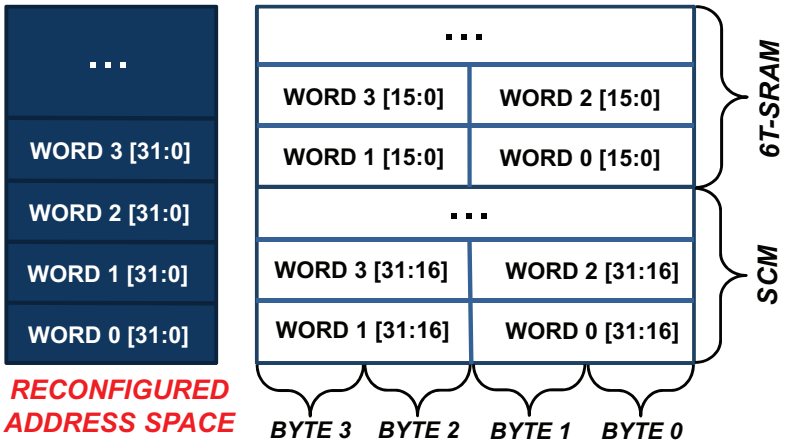


Figure 6.4: Reconfigured address space for a word level access in the split memory area.

for data when executing approximate code regions. In this design memory cuts are 16-bit wide. A word operation (32-bit) on SCM and 6T-RAM logical areas implies an access to two 16 bits banks in the corresponding physical area, while a word operation on the split logical area produces an access to a 16-bit bank in the SCM physical area and to a 16-bit bank in the 6T-SRAM physical area. The offset that defines the boundaries between the three logical memory areas can be configured by writing into a memory-mapped peripheral, accessible by every PE. Thus, the memory map can be re-configured on-the-fly by the software, enabling the optimization of application-specific policies. This design choice is due to the fact that 32-bit data types are common when programming ULP microcontrollers. A 64-bit operation in the split area is not supported, the compiler discards the tolerant flags related to 64-bits variables and reports a warning message.

4 Software Stack

4.1 Programming Model

Many works in literature have shown that a variety of applications are tolerant to errors [164] [165] [166]. In most cases, some code regions of the application are inherently tolerant, while others must be protected from errors. To exploit the hardware support to reliability management, we propose a programming model which is based on a small set of annotations involving program statements and variables, that is a common approach in the field of approximate computing [154] [141]. To fully exploit the parallel capabilities of our multi-core platform with a limited programming effort, we adhere to the OpenMP specification [21], which provides a model for parallel programming that is portable across different shared memory architectures. In the PULP platform the program code is stored in L2 and accessed via private I$. The latter is implemented with SCM cuts, and thus is always reliable. The focus of our techniques in thus only on data, which reside in the TCDM. The DMA unit introduced in Section 3.1 can be used to move data between the L2 and the TCDM. Based on these premises, we propose an extension to OpenMP including two constructs:

- `#pragma tolerant` – a directive applied to a program statement to assert that the related code is tolerant to approximation;

- `var_list(var1, var2, ...)` – a clause coupled to the tolerant directive to qualify what variables can be actually approximated.

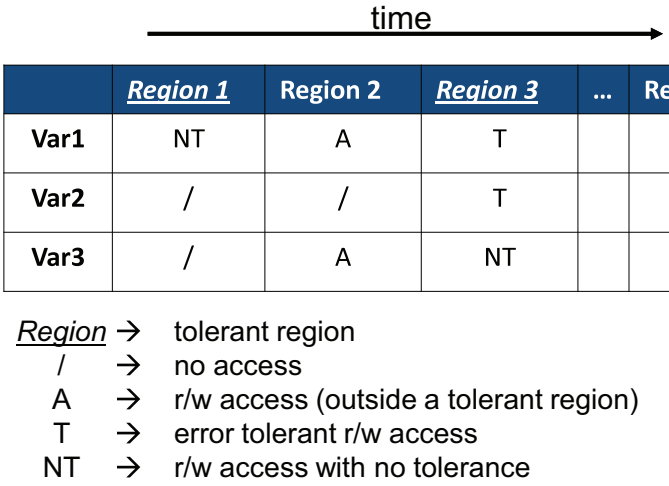


Figure 6.5: Evolution of variables in different program regions.

This model takes into account the full orthogonality between code and data in terms of approximation behavior. For analytical purposes the application code can be divided into multiple regions, each one including one or more statements. Each `#pragma tolerant` directive defines a *tolerant region*, while the rest of the code is grouped in *non-tolerant regions* (function starting/ending points and tolerant regions define their limits). At execution time, the program counter evolves through statements belonging to different regions, and each program variable can be accessed or not in a specific region (i.e., it is out of scope or not required). When a variable is accessed inside a tolerant region, it could be not tolerant to errors due to specific program constraints. Moreover, a variable can be error tolerant or not at different points of the execution flow, depending on the specific region it is accessed in. Figure 6.5 depicts the status of a set of variables when executing a use case that includes an alternation of tolerant and non-tolerant regions. In most practical cases programs start and end with non-tolerant regions with the precise aim to provide a consistent view to their execution environment.

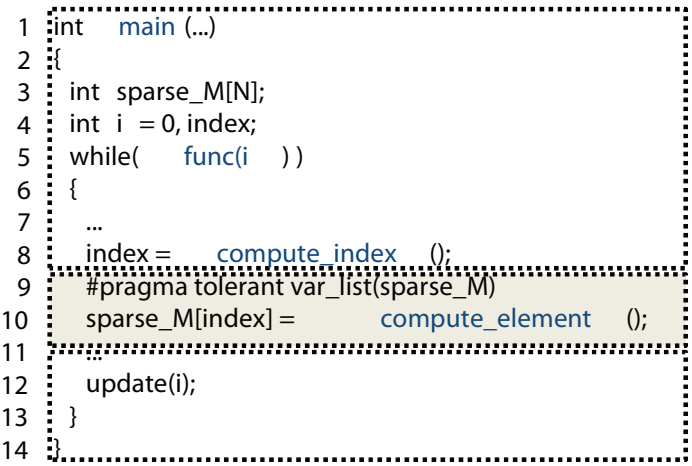


Figure 6.6: A sparse matrix computation with a tolerant directive.

Figure 6.6 shows an example of C code including a tolerant directive. In sparse matrix computation, matrix indexes cannot be approximated (a single error implies wrong element choice), while matrix elements may tolerate approximation. Thus, we declare the `sparse_M` array and the element computation as tolerant, while `index` and its computation are not tolerant. Three code regions are highlighted in the code, a tolerant region and two non-tolerant. Note that error tolerance is not a property of `sparse_M`, but of the code region. When a tolerant variable is copied to a non-tolerant variable, its value is automatically promoted to non-tolerant state after the read. Also the opposite copy is permitted, for instance the automatic variable containing the result of `compute_element()` (non-tolerant) is copied to `sparse_M` (tolerant).

In principle, approximate algorithms can "absorb" errors which occur with a probability lower than a given threshold. To lower this idea in the context of our approach, we need a rigorous methodology to identify the tolerant regions and evaluate the impact of flip-bit errors on data accuracy. Even if an extensive discussion of automatic techniques is out

of scope for this dissertation, we focus on describing the empirical process that we followed to derive suitable approximation thresholds for the various benchmarks. To annotate the applications described in Section 5, we first performed a set of accuracy tests on a x86 workstation. We simulated the effects of an unreliable memory by instrumenting variables with a macro (`APPROX`) which injects flip-bit errors on the LSBs with a configurable probability. Different code regions can be assigned to a specific accuracy level by modifying the flip-bit probability with runtime calls, which correspond to voltage switch operations in the final platform. Statistics on the result accuracy are collected for different configurations, each one corresponding to a bijective association among program variables and accuracy levels. The acceptable error threshold is application specific, and we provide details in Section 5.5.

4.2 Runtime Extensions

The framework described in this chapter requires a lightweight runtime support. The extensions that must be added to the OpenMP runtime can be summarized as follows:

- an interface to activate the split memory area and set its actual size;
- an interface to allocate data structures in a specific logical memory area;
- an interface to modify the voltage supply of 6T-SRAM domains.

In addition, the main modification to the OpenMP runtime involves the allocation of internal data structures. Whenever an OpenMP construct is

present inside a tolerant region, the data structures describing its behavior are not tolerant to errors, and they are allocated partly on the stack (e.g., the bound variables generated to distribute the workload among processing elements in loop constructs) and partly on the heap (e.g., the work-share descriptors). The heap area for these data structures is typically reserved using a dynamic allocator (a call to a malloc in POSIX API). To handle this aspect of the runtime environment, we introduced the concept of *allocation control variable*. This is an internal control variable, akin to others defined by the OpenMP standard, that specifies the logical memory region the runtime must allocate its dynamic data structures in. The positioning of the stack and the value of this variable is a further control knob for our architecture, as described in the next section.

4.3 Compile-time Optimizations

Our compile time optimizations are based on the analysis of the application *call graph*, that is the directed graph representing calling relationships between functions in the execution flow. After applying the *outlining* technique typically used for OpenMP constructs, each statement annotated by a `#pragma tolerant` directive is translated into an equivalent function, and so it follows that the statement in the program flow is replaced by a function call. After this source code transformation, each tolerant region corresponds to a single node in the call graph, while original nodes represent sets of non-tolerant regions. As a preliminary

6T-SRAM voltage	SCM	6T-SRAM	SPLIT
0.5 V	0.6 / 0.6	-	3.2 / 2.9
0.8 V	0.6 / 0.6	13.6 / 12.2	7.1 / 6.4

Table 6.3: Energy consumption (pJ) of a 32-bit read/write access.

step, an extended *use-defs* analysis is applied over the call graph to extract statistics about global and local variables[1]. For each variable the following attributes are collected:

- *scope* – a flag that specifies whether the variable is local or global;
- *dynamic* – a flag that specifies whether the variable is allocated statically (stack or global variable) or dynamically (call to malloc);
- *class* – the type of the variable (scalar/array/reference);
- *use-intervals* – live range of the variable within its entire scope;
- *tolerant-use-intervals* – live range of the variable restricted to tolerant code regions;
- *usedefs* – use-defs chains for the variable.

Table 6.3 reports the memory consumption (in pJ) of a single 32-bit read/write access to different logical memory regions. It considers the case of a single 6T-SRAM voltage domain, but the trend is similar when the number of domains increases. Taking into account the high voltage level (0.8 V), an access to SCM costs less than an access to 6T-SRAM, while an access to the split memory area costs more than an access to SCM but less than an access to 6T-SRAM; at the low voltage

[1]Static use-defs analysis limits the applicability of the approach to programs written as a single translation unit. Inter-Procedural Analysis and Link-Time Optimization approaches can be considered to avoid this limitation.

level (0.5 V), the cost of accessing the split memory area decreases, while 6T-SRAM cannot be accessed any more due to precision constraints. Accesses to the 6T-SRAM logical memory area are not allowed below 0.8 V. Considering the reported values, an approach that uses data tiling on the SCM could be considered the most viable solution to minimize the energy consumption. By leveraging the knowledge of the memory access pattern, data tiling exploits spatial locality in a program by partitioning large data structures into smaller chunks that are brought in and out of the target memory via DMA transfers. By reducing the granularity of data tiling it is theoretically possible to efficiently manage even very small memories. However, creating smaller tiles implies increasing the number of transfers required to process a given data structures, which is subject to increasing impact of DMA latency and programming overheads. Due to these considerations, the energy consumption of the whole system is indeed greater w.r.t. other solutions due to overhead effects, as explained in further detail in Section 5.

Taking into account the usage of both SCM and 6T-SRAM areas, the problem of allocating variables into logical memory areas minimizing the energy consumption can be solved using two general approaches, a formal integer linear programming (ILP) model or a heuristic algorithm. ILP models of the memory allocation problem are known in literature [167]. However in this context they cannot be fed with a complete set of parameters, since experimental evidence shows that access frequency and data size cannot be exactly determined in all practical cases by means of a static analysis.

On the basis of the previous considerations, variables are allocated into different logical memory regions using a *heuristic algorithm* based

on the following policies:

- tolerant variables are allocated in the split area, since SCM is a limited resource while using the 6T-SRAM logical area could miss a voltage switch opportunity;
- non-tolerant variables are allocated in SCM, with the aim to switch down the voltage of unused 6T-SRAM domains;
- variables accessed inside a non-tolerant region (if not considered by previous rules) are allocated in a single 6T-SRAM voltage domain;
- variables devoted to SCM and split memory areas are organized into separate lists, ordered on the basis of the most-frequent-accessed-first heuristic. The variables that eventually do no fit the related memory area are allocated in 6T-SRAM areas, applying (when feasible) the single-domain-per-area heuristic;
- the allocation control variable in the OpenMP runtime is set to use SCM (if available), or a previously allocated 6T-SRAM area with free space (in the worst case it could reference a new 6T-SRAM domain, which cannot be switched down in the region containing the current OpenMP directive);
- the voltage levels of unused domains (i.e., domains which are not used for allocation or contain variables that are not accessed) are switched down at the beginning of a region, with the aim to reduce the leakage power of the related memory banks.

All these transformations involving the source code are performed at compile time on the current application, but in any case they do no affect

the binary size since program data are moved but not increased. This algorithm takes into account all global variables and a subset of local variables, including: arrays, variables allocated with a malloc request, and local variables used in `var_list` clauses. Ultimately, what is not included are automatic variables that are left on the program stack. In the most common cases, the stack is accessed when executing tolerant regions and it typically contains non-tolerant variables (e.g., indexes and pointers), so it must be allocated in SCM or in a 6T-SRAM region kept at high voltage. The previous steps guarantee that the stack is reduced to a minimal set of scalar variables; since the experimental evidence shows that stack accesses are still frequent in tolerant regions after this process, in our solution the stack is preferably allocated in the SCM area. The size of the stack can be determined through static analysis in some cases (e.g., consider the restrictions to the standard C language imposed by OpenCL for kernels [20]), but this is not true in the most general case. For most applications included in our benchmark suite, the aforementioned heuristics enable the allocation of a limited stack area (512 bytes per core). A check for stack overflow is provided at hardware level to guarantee a controlled termination in all the cases that are not properly managed by the static analysis, and therefore require a manual stack sizing.

5 Experimental Evaluation

This section introduces the methodology adopted to validate our architecture, and then focuses on the implementation of the software stack and the reference benchmarks. The results for most relevant metrics

Application	Description	Regions	Tol. regions	Tolerant data %
Color Tracking	A sequence of image processing filters, with the aim to find the center of mass for objects of a specific color in a video	5	3	100% / 50% / 95%
HOG	Histogram of oriented gradients, that is a feature descriptor used to perform object detection	5	3	50% / 80% / 50%
CNN	A convolutional neural network including 6 layers	16	6	50% / 100% / 50% / 100% / 50% / 100%
Health	A signal processing algorithm to process ECG data series with the aim to predict seizures	5	2	50% / 50%
Navi	A navigation support for unmanned vehicles, including the Djikstra algorithm (to find the shortest path between known locations) and a heuristic to plan recharging stops (based on distance and power consumption)	6	2	50% / 25%

Table 6.4: Characterization of the benchmark suite.

are reported and commented (energy consumption in Section 5.3, area compared to energy in Section 5.4 and accuracy in Section 5.5).

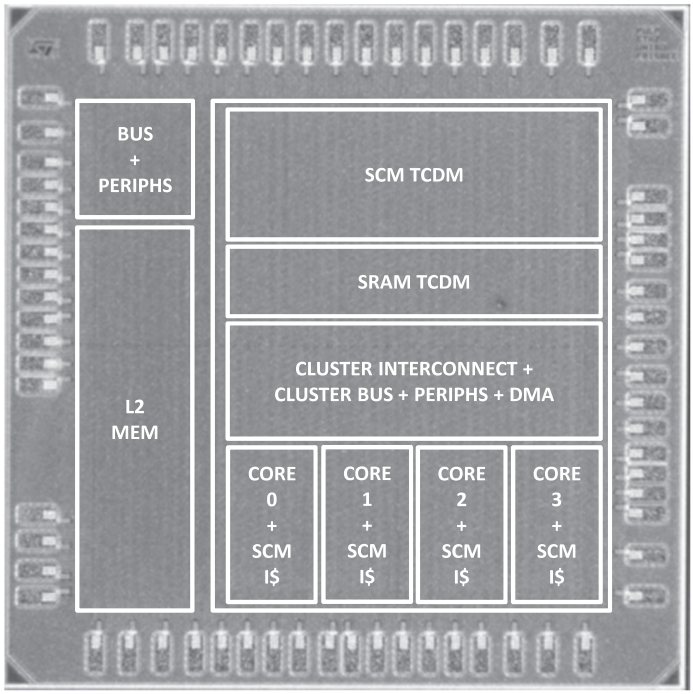


Figure 6.7: Layout of the PULP chip used for memory reliability and power characterization.

5.1 Simulation Setup and Methodology

The power consumption of the baseline architecture at different voltage levels has been measured on the first silicon implementation of PULP, realized with STMicroelectronics 28 nm UTBB FDSOI technology [32]. Figure 6.7 depicts the layout of this reference chip, highlighting its key subsystems. The power figures measured from real silicon have been then partitioned among the key components of the platform (cores, I$, SRAM banks, SCM banks, interconnect) taking into account back-annotated simulations performed on the post place&route netlist of the SoC. Finally, the energy numbers have been fed into the models of Virtual SoC [168], a SystemC-based cycle-accurate virtual platform for the simulation of massively parallel heterogeneous architectures, that is used to perform a

design exploration of the extensions proposed in this chapter. Table 6.3 reports the dynamic energy required for each read/write operation. Table 6.5 reports the leakage energy consumed by the memory cuts for a single clock cycle at 20 MHz, that is the operational frequency of our platform. The size of the memory cuts is the one described in Table 6.2 for alternative setups. This approach couples the advantages of very accurate power models with the simulation speed of the SystemC model, that allows to perform a wide exploration utilizing real-life benchmarks. Compared to a complete RTL simulation, the virtual platform guarantees a maximum error on the number of reported cycles that is between 5% and 6%. This error has a minimum impact on the leakage component of the total energy.

5.2 Software Stack and Benchmark Suite

To experimentally validate our framework, we set up a toolchain based on *Clang* [169] and *LLVM* [101]. The compiler frontend transforms the directives into annotated tokens in its intermediate representation (IR) format. These annotations are collected and interpreted by the heuristic algorithm described in Section 4.3, which is implemented as an IR optimization pass in the LLVM compiler infrastructure. To provide programming model and runtime support, we extended an OpenMP implementation tailored for embedded multicore systems [170], adding the features

	SCM @ 0.5 V	**6T-SRAM @ 0.5 V**	**6T-SRAM @ 0.8 V**
1 region	0.002	0.0056	0.021
2 regions	0.002	0.0043	0.013
4 regions	0.002	0.0022	0.0083

Table 6.5: Leakage energy (pJ per cycle) of SCM and 6T-SRAM cuts.

discussed in Sections 4.1 and 4.2. The benchmarks are implemented in C using standard OpenMP directives to split the workload over the 8 available cores. Considering the intrinsic data parallelism of the computational kernels, for the selected benchmarks we use a `omp parallel for` directive with a static chunking pattern. The data footprint of considered applications does not entirely fit into the TCDM, so we use the DMA engine to implement a *data tiling* technique; in addition, we adopt *double-buffering* with the aim to hide the DMA transfer latency. The applications included in the benchmark suite are described in Table 6.4, that also provides more information on single applications. *Regions* is the total number of code regions, while *Tol. regions* includes only the tolerant regions (Section 4.3). *Tolerant data %* reports the percentage of accessed data which are error tolerant in the corresponding region. For instance, in the first tolerant region of `Color Tracking` all the data are tolerant, in the second one half data are tolerant and the other half are not.

5.3 Energy Consumption

In the experiments that follow we take into account five alternative configurations:

- *Default* – No optimization is applied, program data are allocated into TCDM using standard compilation and linking policies;
- *Heuristic* – Program data are allocated in TCDM using a most-accessed-first heuristic;
- *Heur+VS* – Like *Heuristic*, but enabling low voltage operation for SRAM memory areas that are not used within a target code region;

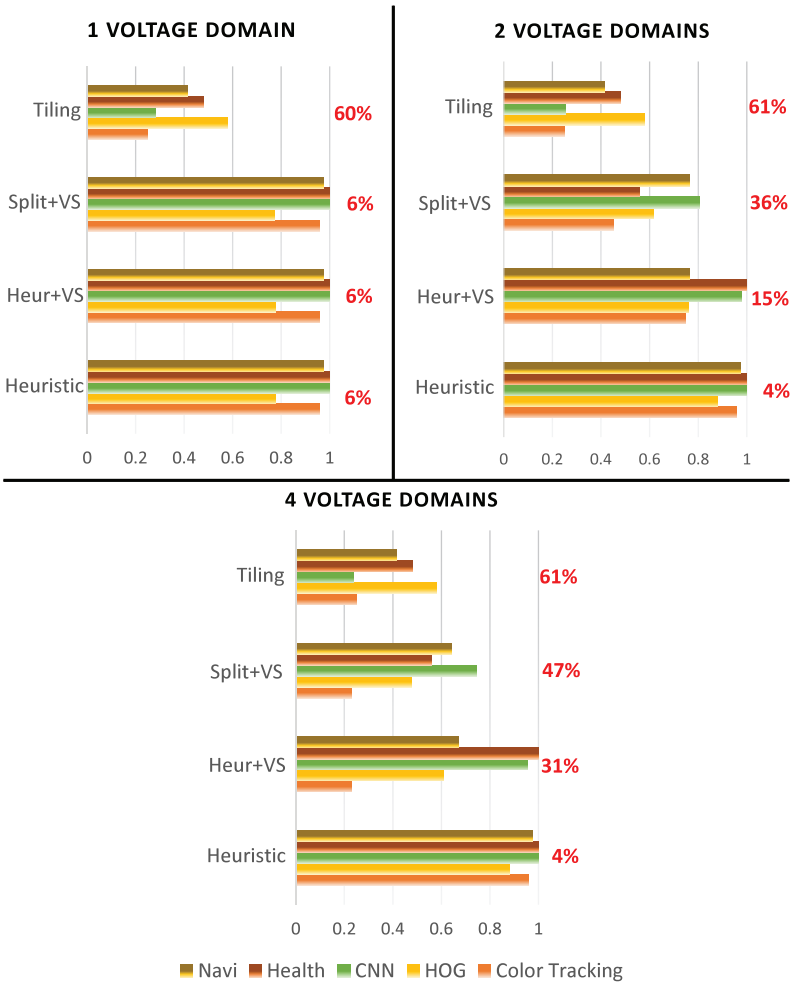


Figure 6.8: Normalized energy consumption of the TCDM (average energy reduction is reported in percentage).

- *Split+VS* – Tolerant data is allocated in the split memory area and the allocation heuristic maximizes low voltage SRAM memory operation;

- *Tiling* – All program data is accessed in small chunks from the SCM only, using DMA transfers to update continuously its content. SRAM is constantly powered at low voltage.

We first analyze the behavior of the memory system in isolation. Figure 6.8 depicts the energy consumption of the TCDM system, including the memory banks and the RMU logic block. The figure shows three plots, one for each explored configuration in terms of SRAM voltage domains: 1 (left), 2 (middle) and 4 (right). Each bar in a group depicts the energy consumption of a given application (normalized to the *Default* configuration), and different groups of bars represent a different configuration from the list above. The average energy reduction (% of the *Default* configuration) is also reported for each group. Considering a single SRAM voltage domain, *Heuristic*, *Heur+VS* and *Split+VS* are identical. This is expected, because unless all data fits in the SCM (which happens only for *Tiling*), it is impossible to lower the voltage of the SRAM without corrupting the correctness of the results. Increasing the number of voltage domains, the benefits of *Split* are more and more evident. With 2 voltage domains, the energy efficiency is increased by 15% on average by just enabling voltage switching on idle SRAM banks, and by 36% by allocating tolerant data on the split memory area. With 4 voltage domains, these values are respectively 31% and 47%. *Tiling* is ≈60% more efficient than the *Default* configuration, independent of the number of voltage domains (as the SRAM is constantly powered at a low voltage). This is not surprising, as SCM accesses are much more energy efficient than SRAM accesses, as reported in Table 6.3.

However, while every configuration requires DMA transfers to accommodate larger data structures than the whole TCDM in chunks, *Tiling* requires more DMA transfers that any other configuration. This is because the SCM is only a fraction of the total TCDM size (one fifth in our architecture), and part of it is devoted to data that always need to

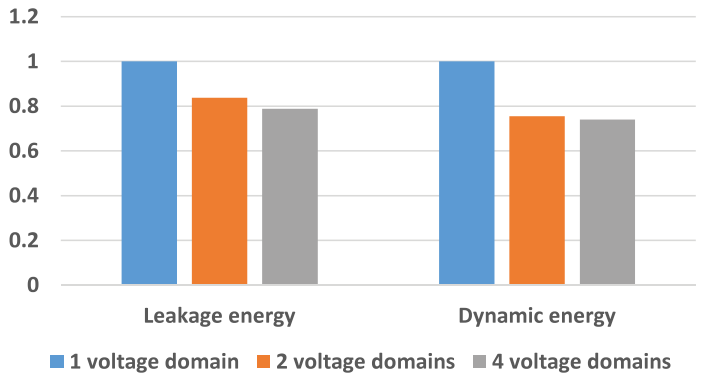


Figure 6.9: Normalized components of the energy consumption.

be reliably accessed, such as the stacks of the threads and the OpenMP runtime metadata. As a consequence, the main drawback of this configuration is that it is more sensitive to DMA management overhead (the finer and more numerous the DMA transfers, the higher the overhead for their management). Focusing on the whole system energy consumption in Figure 6.10 this effect becomes evident. In this plot, energy numbers are normalized to the *Default* configuration and broken down into the contributions of the TCDM system (what we already showed) and other SoC components (cores, I$, DMA engine, interconnect and L2). Again, the figure shows three plots, one for each explored configuration in terms of SRAM voltage domains: 1 (left), 2 (middle) and 4 (right). The higher energy values reported for `HOG` and `Navi` in the *Tiling* configuration are due to the fact that these applications require a bigger stack (768 bytes per core) and a larger footprint of the OpenMP metadata (due to the use of a higher number of dynamic parallelization constructs). Compared to other applications, this reduces the available SCM space to host data tiles, which will eventually be very small and numerous and ultimately imply a larger DMA overhead. Table 6.6 reports execution cycles for

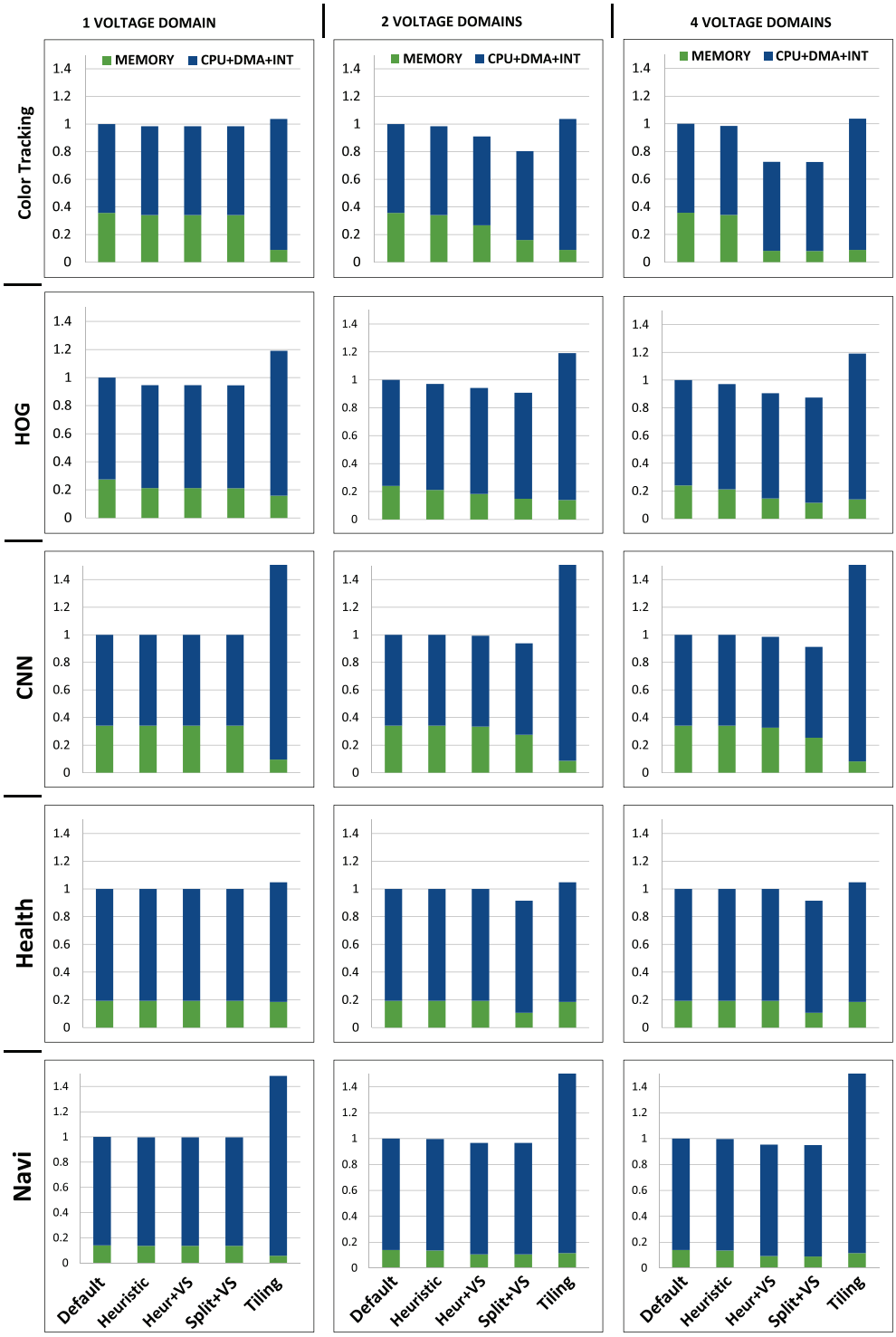


Figure 6.10: Normalized energy consumption of the full SoC.

	Others	**Tiling**
Color Tracking	29.48	32.15
HOG	70487.50	134540.00
CNN	77.78	141.66
Health	1460.16	1563.77
Navi	54.12	99.49

Table 6.6: Total number of core cycles (in millions).

Tiling compared to other configurations. The effect of DMA management is clearly visible also in terms of execution time, with the obvious effect on overall energy consumption[2]. On average, the *Split+VS* approach allows a 13% reduction of the total SoC energy (up to 28% for the considered applications). The benefits of *Tiling* only become visible when the granularity of the technique is such that DMA management overhead becomes negligible (see Section 5.4). This approach never allows tangible benefits for the SCM size considered in our work. It increases the average energy consumption by 27% (by 54% in the worst case).

Figure 6.9 shows dynamic and leakage components of the energy consumption for the *Split+VS* approach. The reported values are the averages of all the applications and are normalized to the baseline case of a single voltage domain. Both leakage and dynamics components of the energy consumption are reduced by our approach. The leakage energy is reduced by switching down the voltage of the 6T-SRAM banks that are not accessed or are part of the split area. The dynamic energy is reduced accessing the 6T-SRAM banks at low voltage in the split area and maximizing the access to the SCM area.

[2]The execution time is not affected by the number of voltage domains, as the performance overhead of a voltage switch is just 2 additional cycles.

Note that the *Split+VS* approach can work in synergy with other approximation techniques, for instance arithmetic units [147] or interconnects [140], with the aim to increase the energy saving of the full system. In this perspective, our programming model can be also extended to support additional knobs.

5.4 SoC Area

Table 6.7 reports the area increase to implement our *Split+VS* technique when considering 2 and 4 SRAM voltage domains. The table also reports average and max energy savings enabled by the technique. *Normalized energy* and *Normalized area* are referred to the baseline SoC (with a single SRAM voltage domain). Looking at energy alone, 4 SRAM voltage domains are better than 2 only. If we consider a combined energy×area metric the difference is no longer visible (for both the average and the best case).

Another approach to use additional design area would obviously be that of increasing the size of the SCM. As we discussed earlier, this would have a beneficial effect on the *Tiling* approach, as it would allow to amortize the overhead of DMA programming. Figure 6.11 compares the effectiveness of *Split* and *Tiling* approaches at using additional chip area. Specifically, the X-axis reports the area increase (% of the original design;

	1 domain	**2 domains**		**4 domains**	
		Average	**Best**	**Average**	**Best**
Area (μm^2)	2943648	2968768		3062400	
Normalized energy	1.00	0.91	0.80	0.88	0.72
Normalized area	1.00	1.01		1.04	
Norm en. × area	1.00	0.92	0.81	0.92	0.81

Table 6.7: Area and energy/area values.

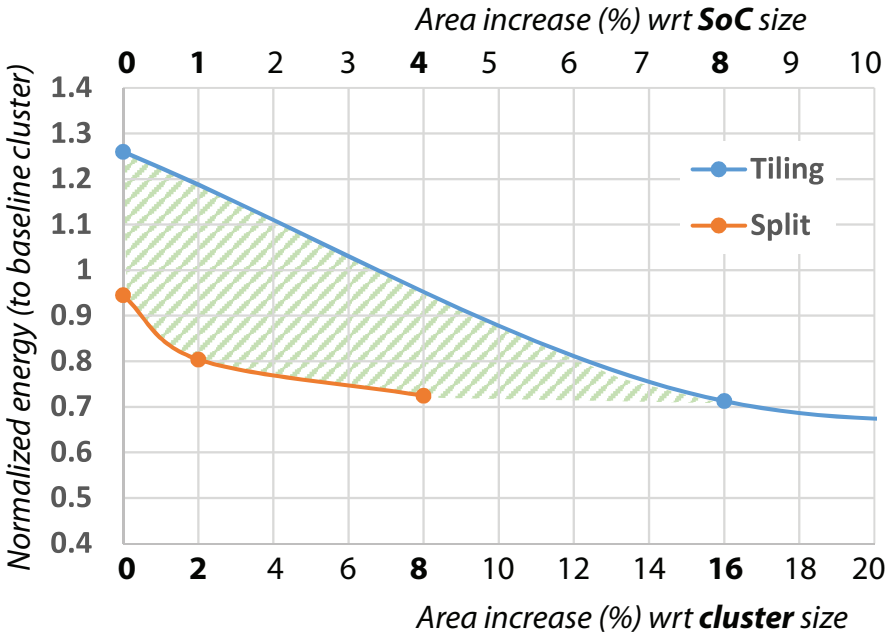


Figure 6.11: Normalized energy consumption for two solutions (SCM with tiling VS hybrid memory with our approach).

for the whole SoC on top and for the cluster only on bottom) and the Y-axis reports normalized energy (compared to the *Default* allocation). The plot shows that *Tiling* is capable of amortizing DMA overheads when the SCM size is increased to 24KB, which corresponds to 16% of the original cluster area. *Split* makes better use of additional chip area when the number of voltage domains is increased to up to 4. Beyond this number we reach a plateau in the energy saving curve. Further partitioning the 6T-SRAM into additional voltage domains, the total area would be dominated by the periphery and embedded power switches. The area between the two curves represent the design space where *Split* is more energy/area-efficient than *Tiling*.

5.5 Application accuracy

To assess the benefits of our approach, we compared the use of unreliable LSBs with a more drastic alternative, that is not computing them at all. To make the two solutions fully comparable, we used the same algorithms and data types, but in the second case we forced the LSBs to zero. Practically, the zeroing case represents an upper bound to the LSB error, since it totally discards the LSB part. All computations are performed on the integer range (32 bit words), and the values reported in the flip-bit column are the worst cases over 1000 executions. To be conservative, we used the highest flip-bit probability considered by our platform (0.0037, as reported in Table 6.1). Table 6.8 reports the value of the mean squared error (MSE) for both approaches, compared to a maximum accepted value. The maximum accepted MSE is not an inherent property of the algorithms, nevertheless it is a requirement of the application context. To derive meaningful values for our benchmarks, we made some hypotheses based on practical use cases. Most works on approximate computing use a similar approach, first checking a wide but yet limited subset of the output and then assuming that it is representative. These approaches have been shown to produce average errors that are acceptable for the considered use cases, but they cannot take corrective actions when large errors occur. Recent works [171] [172] tackle the result quality problem applying runtime checks; a lightweight application-specific metric could be used to derive a quality check of the result, with the aim to adapt the application behavior to an unacceptable quality loss. While this research topic is really promising, its discussion is beyond the scope of this dissertation. However, runtime techniques to support advanced quality management can be straightforwardly implemented on top of our

framework.

`Color Tracking` computes a point centered on the object of a specified color in the input image, and the error metric is based on the Chebyshev distance. Considering an industrial application with a 640×480 image source and a minimum object size of 60 pixels enforced by fixed camera positioning, a maximum error of 15 pixels guarantees that the midline between the center of mass and the object border is not exceeded by the approximated value. From these premises, the maximum MSE is 225. `HOG` computes a feature descriptor that counts discretized occurrences of gradient orientations in different regions of an input image. For human detection applications, the miss rate of HOG-based solutions is around 0.5. We verified that a 0.01% error rate on the computed descriptor does not affect this recognition rate. Using the difference between the binary checksums of feature descriptors as an error metric, this corresponds to a maximum MSE of 2814663 in our experimental setup. `CNN` computes a set of binary features filtered by multiple neural layers. Applied to face recognition, a CNN can achieve a 98% recognition rate. Using the same error metric of the previous benchmark, also in this case we verified that a 0.01% error rate does not affect the expected results. This error rate corresponds to a maximum MSE of 36864 in our experimental setup. `Health` computes the energy levels associated to its final wavelet transform. Considering the typical dynamics of energy levels in our experimental setup, an absolute error of 25 on a single energy level does not compromise the results. Using the average of the differences of energy levels to measure the error, this corresponds to a maximum MSE of 2500. `Navi` computes the travel plan and the required recharging stops of a unmanned vehicle. We consider the total travel time as a key metric

Benchmark	Zeroing	Flip-bit	Max MSE
Color Tracking	676	64	225
HOG	2.12E+13	58564	2814663
CNN	17114769	26244	36864
Health	6867734	378	25000
Navi	1681	36	100

Table 6.8: Mean squared error for zeroing and flip-bit error.

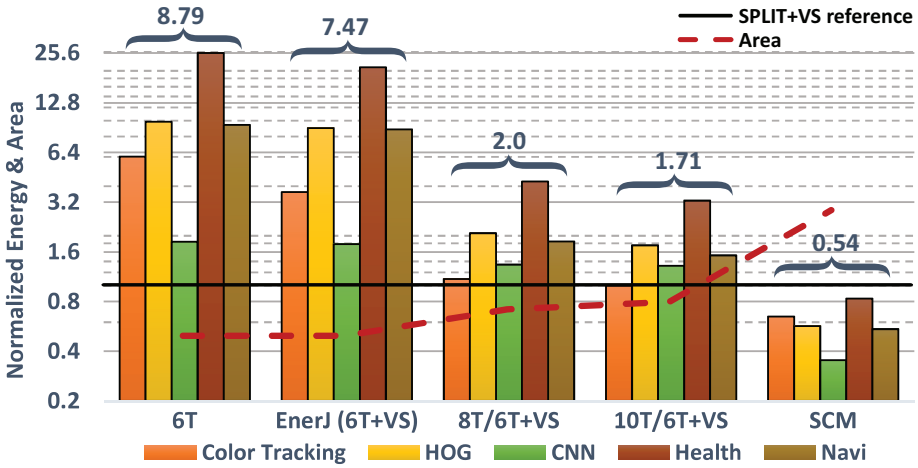


Figure 6.12: Energy consumption and area compared to Split-VS.

to evaluate errors, and then we consider 10 minutes as an upper bound for the maximum delay acceptable by an impatient human being. Finally the maximum MSE is 100.

5.6 Comparison with other approaches

In this section we compare energy and area of our solution to several alternative approaches: (i) *6T* uses 6T-SRAM without applying voltage scaling; (ii) *6T+VS* is the solution adopted by EnerJ [154], using 6T-SRAM and voltage scaling; (iii) *8T/6T+VS* and *10T/6T+VS* implement a hybrid memory system, using respectively 8T-SRAM [135] and 10T-SRAM [132]; (iv) *SCM* uses SCM cuts to implement the full SRAM.

For each of the proposed approaches we configure our simulation infrastructure (introduced in Section 5.1) based on energy numbers reported in the literature and we collect the results running the benchmarks previously described.

Since our technique is explicitly aimed at reducing energy spent in the memory, we first focus on energy and area numbers for the memory subsystem only. Results are shown in Figure 6.12 (bars represent memory energy and the dashed line represents memory subsystem area). Energy/area numbers for each approach are normalized to energy/area numbers for our technique (numbers below one are better than our solution, numbers above one are worse than our solution). The numbers on top of each group of bars show average normalized energy for each benchmark. On average, EnerJ consumes 8.9× higher energy than our solution. From the comparison with 8T/6T+VS and 10T/6T+VS architectures it is also evident that hybrid memory enables major energy savings.

Since from these results there is an obvious trade-off between energy-consumption and area, we also show results for a combined metric – normalized energy-area product (NEAP). We show NEAP for the memory part only (like in Figure 6.12) and for the whole cluster (the benefits of our approach are less evident). NEAP for these two configurations is shown in Table 6.9. Focusing on the memory part only, our solution always provides the best NEAP. When considering also the rest of the cluster components, the benefits are reduced, as expected. However, although the techniques *8T/6T+VS* and *10T/6T+VS* reach very close NEAP to ours, they never perform better.

	Our	6T	6T+VS	8T/6T+VS	10T/6T+VS	SCM
Memory	1.00	4.50	3.83	1.48	1.41	1.58
Cluster	1.00	1.74	1.58	1.03	1.03	1.80

Table 6.9: Normalized energy-area product (NEAP).

6 Conclusion

In this chapter we propose a novel HW/SW approach to design energy-efficient ULP architectures which combine approximate computing and hybrid memory systems featuring both SCM and 6T-SRAM. At the hardware level, we introduce a support to split error-tolerant data so to host MSBs in the SCM and LSBs in the 6T-SRAM. This allows to power the TCDM system at a low voltage while ensuring correct operation by binding potential flip-bit errors to the LSBs only. In addition, by organizing 6T-SRAM banks into multiple and independent voltage domains we enable fine-grained, software-controlled voltage switching policies. At the software level, we propose language constructs to specify what regions of code and what variables are tolerant to approximation. A compiler pass implements a heuristic algorithm which allocates data into available memory regions and leverages hardware knobs to maximize energy savings. Experimental results show that our hybrid memory architecture can reduce by 47% the energy consumption of the TCDM memory. Focusing on the whole-system, our technique allows on average 27% savings and outperforms other solutions. At the same time we can guarantee the exact level of accuracy required by real-life applications, since the MSE of our approach is always below a maximum reference value when a proper approximation policy is applied. Overall, these results encourage further research activities in the field of hybrid memory architectures, as these solutions represent a promising opportunity for the design of future ULP

systems.

Conclusion

Many-core computing platforms have emerged as a promising solution to tackle the major issues of architecture designers in the last years, namely the *power wall*, the *memory wall* and the *ILP wall*. However the technological evolution is going to hit these walls again, and this will require disruptive changes in the next decade. On the software side, it is paramount to provide programmers proper techniques and tools to take advantage of the increasing number of cores. In this scenario, there are two main objectives: first, improving performance and energy efficiency of the platform, which are key metrics for embedded computing systems; second, enforcing software engineering practices with the aim to guarantee code quality and reduce software costs.

This thesis introduces a set of techniques and tools that have been studied to achieve these objectives. In the dissertation we consider a range of devices with a more and more stringent power budget, since the emergence of new application areas (e.g., IoT applications) makes a common requirement that devices combine complex processing capability with ultra-low-power operation. The proposed solutions aim at ehnancing the state-of-the-art of optimization techniques for parallel programming of embedded parallel architectures, considering both general-purpose and domain-specific programming models. Even if it is not yet

totally clear which programming paradigm will prevail in the next years, our work brings out two main aspects that are an important subject for further research activities: first, the need for transparent mechanisms to handle processing and memory management in heterogenous platforms; second, the importance of providing synergistic hardware and software support to effectively exploit advanced techniques such as approximate and near-threshold computing.

List of Publications

1. Paolo Burgio, **Giuseppe Tagliavini**, Francesco Conti, Andrea Marongiu, Luca Benini. *"Tightly-coupled hardware support to dynamic parallelism acceleration in embedded shared memory clusters."* In Proceedings of the conference on Design, Automation & Test in Europe (DATE), p. 156. EDAA, 2014.

2. **Giuseppe Tagliavini**, Germain Haugou, Luca Benini. *"Supporting localized OpenVX kernel execution for efficient computer vision application development on STHORM many-core platform."* In Proceedings of the 11th ACM Conference on Computing Frontiers, p. 23. ACM, 2014.

3. **Giuseppe Tagliavini**, Germain Haugou, Luca Benini. *"Optimizing memory bandwidth in OpenVX graph execution on embedded many-core accelerators."* In 2014 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1-8. IEEE, 2014.

4. Davide Rossi, Igor Loi, Francesco Conti, **Giuseppe Tagliavini**, Antonio Pullini, Andrea Marongiu. *"Energy efficient parallel computing on the PULP platform with support for OpenMP."* In 2014

IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI), pp. 1-5. IEEE, 2014.

5. **Giuseppe Tagliavini**, Germain Haugou, Andrea Marongiu, Luca Benini. *"ADRENALINE: an OpenVX environment to optimize embedded vision applications on many-core accelerators."* In 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 289-296. IEEE, 2015.

6. Nandhini Chandramoorthy, **Giuseppe Tagliavini**, Kevin Irick, Antonio Pullini, Siddharth Advani, Sulaiman Al Habsi, Matthew Cotter, John Sampson, Vijaykrishnan Narayanan, Luca Benini. *"Exploring architectural heterogeneity in intelligent vision systems."* In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 1-12. IEEE, 2015.

7. **Giuseppe Tagliavini**, Germain Haugou, Andrea Marongiu, Luca Benini. *"A framework for optimizing OpenVX applications performance on embedded manycore accelerators."* In Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 125-128. ACM, 2015.

8. **Giuseppe Tagliavini**, Davide Rossi, Luca Benini, Andrea Marongiu. *"Synergistic Architecture and Programming Model Support for Approximate Micropower Computing."* In 2015 IEEE Computer Society Annual Symposium on VLSI, pp. 280-285. IEEE, 2015.

9. Andrea Marongiu, Alessandro Capotondi, **Giuseppe Tagliavini**,

Luca Benini. *"Simplifying many-core-based heterogeneous SoC programming with offload directives."* In IEEE Transactions on Industrial Informatics 11, no. 4 pp. 957-967. IEEE, 2015.

10. Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, **Giuseppe Tagliavini**, Alessandro Capotondi, Philippe Flatresse, Luca Benini. *"PULP: A parallel ultra low power platform for next generation IoT applications."* In Hot Chips 27 Symposium (HCS), pp. 1-39. IEEE, 2015.

11. **Giuseppe Tagliavini**, Germain Haugou, Andrea Marongiu, Luca Benini. *"Optimizing memory bandwidth exploitation for OpenVX applications on embedded many-core accelerators."* In Journal of Real-Time Image Processing, pp. 1-20. Springer, 2016

12. **Giuseppe Tagliavini**, Germain Haugou, Andrea Marongiu, Luca Benini. *"Enabling OpenVX support in mW-scale parallel accelerators."* In 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES), pp. 1-10. IEEE, 2016.

13. **Giuseppe Tagliavini**, Davide Rossi, Andrea Marongiu, Luca Benini. *"Synergistic HW/SW Approximation Techniques for Ultra-Low-Power Parallel Computing."* In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. IEEE, 2016.

14. **Giuseppe Tagliavini**, Andrea Marongiu, Davide Rossi, Luca Benini. *"Always-on motion detection with application-level error control on a near-threshold approximate computing platform."* In 2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS),

pp. 552-555. IEEE, 2016.

Acknowledgments

Questa tesi è il risultato di tre anni di lavoro intensi, che mi hanno richiesto molto lavoro ma mi hanno anche elargito diverse soddisfazioni.

Per prima cosa voglio ringraziare il prof. Luca Benini, che mi ha supportato in questo percorso e mi ha sempre fornito preziosi consigli su come migliorare il mio lavoro.

Un ringraziamento speciale va poi ad Andrea Marongiu, che ha cercato di insegnarmi uno stile di lavoro paziente e metodico al fine di moderare il mio approccio troppo sbrigativo: per me i suoi consigli sono stati molto preziosi e mi hanno guidato verso una metodologia di lavoro più equilibrata... anche se nel profondo rimarrò sempre "barroso"!!!

I would like to thank Prof. Philippe Coussy and Prof. Marco D. Santambrogio for their partecipation to the review process of this manuscript, I really appreciated their advices.

Voglio anche ringraziare i miei colleghi (attuali e passati) per l'enorme supporto che mi hanno dato in questi anni a livello lavorativo e personale, e per aver trasformarto in molteplici occasioni la fatica di svegliarsi la mattina dopo una notte di lavoro nel piacere di godersi una pausa caffé in un clima di rilassatezza e ilarità. Grazie a tutti voi: Andrea Bartolini, Christian Pinto, Daniele Bortolotti, Daniele Cesarini, Davide Rossi, Francesco Beneventi, Francesco Conti, Francesco Paci, Germain

Haugou, Igor Loi, Marco Balboni, Paolo Burgio, Simone Benatti, Thomas Bridi.

Ringrazio i miei genitori, Nadia e Angelo, che hanno sempre creduto che io potessi raggiungere ogni obiettivo che mi ero prefissato.

Infine il ringraziamento più importante va a Francesca, la mia compagna e l'amore della mia vita. Senza il suo supporto e la sua fiducia nelle mie capacità non avrei mai intrapreso questo percorso.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 5, pp. 33–35, 2006.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[4] G. G. Shahidi, "From 2D-planar to 3D-non-planar device architecture: A scalable path forward?" in *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*. IEEE, 2013, pp. 1–8.

[5] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[6] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," *Technology*, vol. 1, 2005.

[7] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[8] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 205–218.

[9] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[10] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *Ieee Micro*, vol. 2, no. 32, pp. 28–37, 2012.

[11] H. Chung, M. Kang, and H.-D. Cho, "Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big. LITTLEâĎć Technology," *Samsung White Paper*, 2012.

[12] Texas Instruments Inc. KeyStone II System-on-Chip 66AK2Hx. [Online]. Available: http://www.ti.com/lit/ds/symlink/66ak2h12.pdf

[13] Nvidia Inc. (2014) Nvidia Tegra X1 - NVIDIA'S New Mobile Superchip. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[14] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *HPEC*, 2013, pp. 1–6.

[15] PEZY Computing. (2014) PEZY-SC Many Core Processor. [Online]. Available: http://www.pezy.co.jp/en/products/pezy-sc.html

[16] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012.

[17] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC System-on-Chip," *arXiv preprint arXiv:1610.01832*, 2016.

[18] M. Jørgensen, "A review of studies on expert estimation of software development effort," *Journal of Systems and Software*, vol. 70, no. 1, pp. 37–60, 2004.

[19] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[20] Kronos Group, "OpenCL 2.1 Specification," https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf, 2015.

[21] "OpenMP 4.0 Specification," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2015.

[22] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC First Experiences with Real-World Applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012.

[23] Kronos Group, "The OpenVX API for hardware acceleration," http://www.khronos.org/openvx, 2015.

[24] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, "Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 1504–1509.

[25] P. Burgio, G. Tagliavini, F. Conti, A. Marongiu, and L. Benini, "Tightly-coupled hardware support to dynamic parallelism acceleration in embedded shared memory clusters," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 156.

[26] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. Al Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini, "Exploring architectural heterogeneity in intelligent vision systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 1–12.

[27] G. Tagliavini, G. Haugou, and L. Benini, "Optimizing memory bandwidth in OpenVX graph execution on embedded many-core accelerators," in *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*. IEEE, 2014, pp. 1–8.

[28] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Optimizing memory bandwidth exploitation for OpenVX applications on embedded many-core accelerators," *Journal of Real-Time Image Processing*, pp. 1–20, 2016.

[29] ——, "ADRENALINE: An OpenVX Environment to Optimize Embedded Vision Applications on Many-core Accelerators," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, 2015, pp. 289–296.

[30] ——, "Enabling OpenVX Support in mW-scale Parallel Accelerators," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. ACM, 2016, pp. 2:1–2:10.

[31] G. Tagliavini, D. Rossi, A. Marongiu, and L. Benini, "Synergistic Architecture and Programming Model Support for Approximate Micropower Computing," in *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*, 2015, pp. 280–285.

[32] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Bartolini, P. Flatresse, and L. Benini, "A 60 GOPS/W,- 1.8 V to 0.9 V body bias ULP cluster in 28nm UTBB FD-SOI technology," *Solid-State Electronics*, vol. 117, pp. 170–184, 2016.

[33] Zedboard.org, "Zedboard product page," http://zedboard.org/product/zedboard, 2015.

[34] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gurkaynak, A. Teman, J. Constantin, A. Burg, I. M. Panades, E. BeignÃĺ, F. Clermidy, F. Abouzeid, P. Flatresse, and L. Benini, "193 MOPS/mW

162 MOPS, 0.32V to 1.15V Voltage Range Multi-Core Accelerator for Energy-Efficient Parallel and Sequential Digital Processing," in *Cool Chips XIX*, 2016.

[35] N. Planes, O. Weber, V. Barral, S. Haendler, D. Noblet, D. Croain, M. Bocat, P. O. Sassoulas, X. Federspiel, A. Cros, A. Bajolet, E. Richard, B. Dumont, P. Perreau, D. Petit, D. Golanski, C. Fenouillet-BÃl'ranger, N. Guillot, M. Rafik, V. Huard, S. Puget, X. Montagner, M. A. Jaud, O. Rozeau, O. Saxod, F. Wacquant, F. Monsieur, D. Barge, L. Pinzelli, M. Mellier, F. Boeuf, F. Arnaud, and M. Haond, "28nm FDSOI technology platform for high-speed low-voltage digital applications," in *VLSI Technology (VLSIT), 2012 Symposium on*, June 2012, pp. 133–134.

[36] "OpenRISC project," http://www.opencores.org/or1k/.

[37] D. Rossi, I. Loi, G. Haugou, and L. Benini, "Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014, p. 15.

[38] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[39] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[40] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[41] K.-F. Faxén, "Wool-a work stealing library," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 93–100, 2008.

[42] K. Sakamoto and T. Furumoto, "Grand central dispatch," in *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 2012, pp. 139–145.

[43] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[44] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture," in *International Workshop on OpenMP*. Springer, 2014, pp. 202–214.

[45] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, "Simplifying many-core-based heterogeneous soc programming with offload directives," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 957–967, 2015.

[46] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *International Workshop on OpenMP*. Springer, 2008, pp. 100–110.

[47] C. D. Marlin, *Coroutines: a programming methodology, a language design and an implementation*. Springer Science & Business Media, 1980, no. 95.

[48] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 36.

[49] A. Heinecke, M. Klemm, and H. Bungartz, "From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture," *Computing in Science & Engineering*, 2012.

[50] P. Rogers and A. FELLOW, "Heterogeneous system architecture overview," in *Hot Chips*, 2013.

[51] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "Towards Transparently Tackling Functionality and Performance Issues across Different OpenCL Platforms," in *Computing and Networking (CANDAR), 2014 Second International Symposium on*, 2014.

[52] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*. Springer, 2002.

[53] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2011.

[54] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013.

[55] Embedded Vision Alliance, http://www.embedded-vision.com/, 2015.

[56] S. Park, A. A. Maashri, K. M. Irick, A. Chandrashekhar, M. Cotter, N. Chandramoorthy, M. Debole, and V. Narayanan, "System-on-chip for biologically inspired vision applications," *Information Processing Society of Japan: Transactions on System LSI Design Methodology*, 2012.

[57] S. Greengard, "Computational photography comes into focus," *Commun. ACM*, 2014.

[58] Plurality Ltd, "The HyperCore Processor," http://www.plurality.com/hypercore.html, 2015.

[59] Adapteva, Inc., "Epiphany-IV 64-core 28nm Microprocessor," http://www.adapteva.com/products/silicon-devices/e64g401/, 2015.

[60] 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 *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.

[61] Kronos Group, "The OpenCL 1.1 Specifications," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2015.

[62] M. Sonka, V. Hlavac, R. Boyle *et al.*, *Image processing, analysis, and machine vision*. Thomson Toronto, 2008.

[63] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the 42nd annual Design Automation Conference*, 2005.

[64] M. Magno, F. Tombari, D. Brunelli, L. Di Stefano, and L. Benini, "Multimodal abandoned/removed object detection for low power video surveillance systems," in *Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, 2009.

[65] F. Schubert, K. Schertler, and K. Mikolajczyk, "A hands-on approach to high-dynamic-range and superresolution fusion," in *Applications of Computer Vision (WACV), 2009 Workshop on*, 2009.

[66] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010.

[67] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[68] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision," in *IJCAI*, 1981.

[69] T. Lepley, P. Paulin, and E. Flamand, "A Novel Compilation Approach for Image Processing Graphs on a Many-core Platform with Explicitly Managed Memory," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2013.

[70] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Korner, and W. Eckert, "Hipacc: A Domain-Specific Language and Compiler for Image Processing," *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[71] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing System-Level Optimization with OpenVX Graphs," in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2014.

[72] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011.

[73] Y. Lei, Z. Gang, R. Si-Heon, L. Choon-Young, L. Sang-Ryong, and K.-M. Bae, "The platform of image acquisition and processing system based on DSP and FPGA," in *International Conference on Smart Manufacturing Application*, 2008.

[74] S. K. Gehrig, F. Eberli, and T. Meyer, "A real-time low-power stereo vision engine using semi-global matching," in *Computer Vision Systems*. Springer, 2009.

[75] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "CHARM: A Composable Heterogeneous Accelerator-rich Microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, 2012.

[76] J. Cong, C. Liu, M. A. Ghodrat, G. Reinman, M. Gill, and Y. Zou, "AXR-CMP: Architecture Support in Accelerator-Rich CMPs," 2011.

[77] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," in *Proceedings of the 41st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2014.

[78] OpenCV Library Homepage, http://www.opencv.com/, 2015.

[79] J. Coombs and R. Prabhu, "OpenCV on TIâĂŹs DSP+ ARM® platforms: Mitigating the challenges of porting OpenCV to embedded platforms," *Texas Instruments*, 2011.

[80] Tegra Android Development Documentation Website, http://docs.nvidia.com/tegra/index.html, 2015.

[81] Qualcomm, "Computer Vision (FastCV)," https://developer.qualcomm.com/computer-vision-fastcv, 2015.

[82] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, 2010.

[83] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[84] P. Boudier and G. Sellers, "Memory System on Fusion APUs," *AMD fusion developer summit*, 2011.

[85] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, 2009.

[86] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "COMIC: a coherent shared memory interface for Cell BE," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[87] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien, "Hybrid access-specific software cache techniques for the Cell BE architecture," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[88] A. Franceschelli, P. Burgio, G. Tagliavini, A. Marongiu, M. Ruggiero, M. Lombardi, A. Bonfietti, M. Milano, and L. Benini, "MPOpt-Cell: A High-performance Data-flow Programming Environment for the CELL BE Processor," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011.

[89] "Tensilica Customizable Processor IP," http://ip.cadence.com/ipportfolio/tensilica-ip.

[90] Gartner, "Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020," http://www.gartner.com/newsroom/id/2636073.

[91] ABIresearch, "More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020," https://www.abiresearch.com/market-research/product/1021642-edge-analytics-in-iot/.

[92] ——, "Edge Analytics in IoT," https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/.

[93] "CEVA-XM4 Intelligent Vision Processor," http://www.ceva-dsp.com/CEVA-XM4.

[94] "DesignWare EV Family of Vision Processors," https://www.synopsys.com/dw/ipdir.php?ds=ev52-ev54.

[95] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu *et al.*, "Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*. IEEE, 2012, pp. 190–192.

[96] D. Rossi *et al.*, "PULP: A Parallel Ultra-Low-Power Platform for Next Generation IoT Applications," in *HotChips 2015*.

[97] A. Y. Dogan *et al.*, "Power/performance exploration of single-core and multi-core processor approaches for biomedical signal processing," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*. Springer, 2011, pp. 102–111.

[98] S. Banerjee and D. O. Wu, "Final report from the nsf workshop on future directions in wireless networking," 2013.

[99] "IoT-From Research and Innovation to Market Deployment," http://www.internet-of-things-research.eu/.

[100] I. Kadayif and M. Kandemir, "Data Space-oriented Tiling for Enhancing Locality," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 2, pp. 388–414, 2005.

[101] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on.* IEEE, 2004, pp. 75–86.

[102] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, 1995.

[103] STMicroelectronics, *STM32L476xx Datasheet*, rev. 2.

[104] P. Flatresse *et al.*, "Ultra-wide body-bias range LDPC decoder in 28nm UTBB FDSOI technology," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International.* IEEE, 2013, pp. 424–425.

[105] T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[106] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 24–35, 1987.

[107] P. Virouleau *et al.*, "Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite," in *Using and Improving OpenMP for Devices, Tasks, and More*. Springer, 2014, pp. 16–29.

[108] A. Kukanov and M. J. Voss, "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[109] Khronos Group, "OpenVX resources," http://www.khronos.org/openvx/resources.

[110] A. T. Tan, J. Falcou, D. Etiemble, and H. Kaiser, "Automatic task-based code generation for high performance domain specific embedded language," *International Journal of Parallel Programming*, pp. 1–17, 2014.

[111] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 311–320.

[112] M. Püschel *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[113] Y. Futamura, "Partial evaluation of computation process–an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, no. 4, pp. 381–391, 1999.

[114] NVIDIA, "NVIDIA Jetson TX1 Supercomputer-on-Module Drives Next Wave of Autonomous Machines," http://devblogs.nvidia.com/parallelforall/nvidia-jetson-tx1.

[115] Z. Guo, J. Han, and T. Li, "Implementing OpenVX on a polymorphous array processor," in *2015 IEEE 16th International Conference on Communication Technology (ICCT)*, 2015, pp. 598–601.

[116] SiliconLabs, *EFM32G210 Datasheet*, rev. 1.90.

[117] Texas Instruments, *MSP430F161 Datasheet*, rev. G.

[118] J.-S. Yoon, J.-H. Kim, H.-E. Kim, W.-Y. Lee, S.-H. Kim, K. Chung, J.-S. Park, and L.-S. Kim, "A Unified Graphics and Vision Processor With a 0.89/spl mu/W/fps Pose Estimation Engine for Augmented Reality," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 2, pp. 206–216, 2013.

[119] J. Oh, S. Lee, and H.-J. Yoo, "1.2-mW Online Learning Mixed-Mode Intelligent Inference Engine for Low-Power Real-Time Object Recognition Processor," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, pp. 921–933, 2013.

[120] Carnegie Mellon University, "CMUcam," http://www.cmucam.org/.

[121] NXP, *LPC5410x Datasheet*, rev. 2.1.

[122] Freescale, *MC9S12XDP512 Datasheet*, rev. 2.21.

[123] "STM32L4 ultra-low-power MCUs," http://www.st.com/stm32l4.

[124] "Texas Instruments MSP Microcontrollers," http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/overview.page.

[125] "Ambiq Apollo," http://ambiqmicro.com/low-power-microcontroller.

[126] A. Y. Dogan, J. Constantin, D. Atienza, A. Burg, and L. Benini, "Low-power processor architecture exploration for online biomedical signal analysis," *IET Circuits, Devices Systems*, vol. 6, no. 5, pp. 279–286, Sept 2012.

[127] E. Krimer, R. Pawlowski, M. Erez, and P. Chiang, "Synctium: a Near-Threshold Stream Processor for Energy-Constrained Parallel Applications," *IEEE Computer Architecture Letters*, vol. 9, no. 1, pp. 21–24, Jan 2010.

[128] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu, "Energy efficient parallel computing on the pulp platform with support for openmp," in *Electrical & Electronics Engineers in Israel (IEEEI), 2014 IEEE 28th Convention of.* IEEE, 2014, pp. 1–5.

[129] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb 2010.

[130] D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J.-D. Legat, "A 25MHz 7μW/MHz ultra-low-voltage microcontroller SoC in 65nm LP/GP CMOS for low-carbon wireless sensor nodes," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International.* IEEE, 2012, pp. 490–492.

[131] B. H. Calhoun and A. Chandrakasan, "Analyzing static noise margin for sub-threshold SRAM in 65nm CMOS," in *Solid-State Circuits Conference, 2005. ESSCIRC 2005. Proceedings of the 31st European*, Sept 2005, pp. 363–366.

[132] M. E. Sinangil, N. Verma, and A. P. Chandrakasan, "A Reconfigurable 8T Ultra-Dynamic Voltage Scalable (U-DVS) SRAM in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 11, pp. 3163–3173, Nov 2009.

[133] B. H. Calhoun and A. P. Chandrakasan, "A 256-kb 65-nm Subthreshold SRAM Design for Ultra-Low-Voltage Operation," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 3, pp. 680–688, March 2007.

[134] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, "Power, Area, and Performance Optimization of Standard Cell Memory Arrays through Controlled Placement," in *ACM Transactions on Design Automation of Electronic Systems*, 2016.

[135] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues, "Benchmarking of Standard-Cell Based Memories in the Sub-V_T Domain in 65-nm CMOS Technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 2, pp. 173–182, June 2011.

[136] D. Bortolotti, A. Bartolini, C. Weis, D. Rossi, and L. Benini, "Hybrid memory architecture for voltage scaling in ultra-low power multi-core biomedical processors," in *Design, Automation and Test*

in Europe Conference and Exhibition (DATE), 2014, March 2014, pp. 1–6.

[137] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *26th IEEE International Real-Time Systems Symposium*, Dec 2005, pp. 10 pp.–232.

[138] J. Hu, C. J. Xue, Q. Zhuge, W. C. Tseng, and E. H. M. Sha, "Towards energy efficient hybrid on-chip Scratch Pad Memory with non-volatile memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[139] M. Gautschi, M. Schaffner, F. K. GÃijrkaynak, and L. Benini, "A 65nm CMOS 6.4-to-29.2pJ/FLOP@0.8V shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan 2016, pp. 82–83.

[140] A. Mineo, M. Palesi, G. Ascia, P. Pande, and V. Catania, "On-Chip Communication Energy Reduction through Reliability Aware Adaptive Voltage Swing Scaling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[141] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 301–312.

[142] S. H. Nawab, A. V. Oppenheim, A. P. Chandrakasan, J. M. Winograd, and J. T. Ludwig, "Approximate signal processing," *Journal*

of VLSI signal processing systems for signal, image and video technology, vol. 15, no. 1-2, pp. 177–200, 1997.

[143] J. T. Ludwig, S. H. Nawab, and A. P. Chandrakasan, "Low-power digital filtering using approximate processing," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 3, pp. 395–400, Mar 1996.

[144] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *International Symposium on Low Power Electronics and Design*, Aug 1999, pp. 30–35.

[145] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate Computing: A Survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb 2016.

[146] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.

[147] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, Jan 2013.

[148] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini, "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, Sept 2013, pp. 1–10.

[149] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June 2010, pp. 555–560.

[150] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "ECOSystem: Managing Energy As a First Class Operating System Resource," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. ACM, 2002, pp. 123–132.

[151] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," in *MIT CSAIL Tech. Reports*, 2009.

[152] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *9th IEEE/ACM International Symposium on Code Generation and Optimization*, April 2011, pp. 85–96.

[153] W. Baek and T. Chilimbi, "Green: A system for supporting energy-conscious programming using principled approximation," TR-2009-089, Microsoft Research, Tech. Rep., 2009.

[154] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *Proceedings of the 32Nd*

ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '11. ACM, 2011, pp. 164–174.

[155] I. J. Chang, D. Mohapatra, and K. Roy, "A Priority-Based 6T/8T Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, Feb 2011.

[156] M. Cho, J. Schlessman, W. Wolf, and S. Mukhopadhyay, "Reconfigurable SRAM Architecture With Spatial Voltage Scaling for Low Power Mobile Multimedia Applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 1, pp. 161–165, Jan 2011.

[157] N. Verma and A. P. Chandrakasan, "A 256 kb 65 nm 8T Subthreshold SRAM Employing Sense-Amplifier Redundancy," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 141–149, Jan 2008.

[158] F. Frustaci, M. Khayatzadeh, D. Blaauw, D. Sylvester, and M. Alioto, "SRAM for Error-Tolerant Applications With Dynamic Energy-Quality Management in 28 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 5, pp. 1310–1323, May 2015.

[159] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, May 2013, pp. 1–9.

[160] P. Roy, R. Ray, C. Wang, and W. F. Wong, "ASAC: Automatic Sensitivity Analysis for Approximate Computing," in *Proceedings of*

the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, ser. LCTES '14. ACM, 2014, pp. 95–104.

[161] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, "RRAM-Based Analog Approximate Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 12, pp. 1905–1917, Dec 2015.

[162] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ser. ISLPED '13. IEEE Press, 2013, pp. 242–247.

[163] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues, "Benchmarking of standard-cell based memories in the sub-domain in 65-nm CMOS technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 2, pp. 173–182, June 2011.

[164] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error Resilient System Architecture for probabilistic applications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, March 2010, pp. 1560–1565.

[165] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," *SIGPLAN Not.*, vol. 46, no. 3, pp. 213–224, Mar. 2011.

[166] V. Wong and M. Horowitz, "Soft error resilience of probabilistic inference applications," in *In Proceedings of the Workshop on System Effects of Logic Soft Errors*. SELSE, 2006.

[167] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 409–415.

[168] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini, "VirtualSoC: A Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 2182–2187.

[169] "clang: a C language family frontend for LLVM ," http://clang.llvm.org/.

[170] A. Marongiu and L. Benini, "An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 222–236, Feb 2012.

[171] B. Grigorian and G. Reinman, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, July 2014, pp. 248–255.

[172] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An Online Quality Management System for Approximate Computing,"

in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. ACM, 2015, pp. 554–566.