Alma Mater Studiorum - University of Bologna

ARCES - Advanced Research Center on Electronic Systems
for Information and Communication Technologies E.De Castro

PhD Course in Information Technology

XXVI CYCLE - Scientific-Disciplinary sector ING-INF /01

Heterogeneous Multi-core Architectures for High Performance Computing

Candidate:
Matteo Chiesi

Advisors:
Prof. Roberto Guerrieri
Prof. Eleonora Franchi Scarselli

PhD Course Coordinator:
Prof. Claudio Fiegna

Final examination year: 2014

Abstract

This thesis deals with low-cost heterogeneous architectures in standard workstation frameworks.

Heterogeneous computer architectures represent an appealing alternative to traditional supercomputers because they are based on commodity hardware components fabricated in large quantities. Hence their price-performance ratio is unparalleled in the world of high performance computing (HPC).

In particular in this thesis, different aspects related to the performance and power consumption of heterogeneous architectures have been explored. The thesis initially focuses on an efficient implementation of a parallel application, where the execution time is dominated by an high number of floating point instructions. Then the thesis touches the central problem of efficient management of power peaks in heterogeneous computing systems. Finally it discusses a memory-bounded problem, where the execution time is dominated by the memory latency.

Specifically, the following main contributions have been carried out:

- A novel framework for the design and analysis of solar field for Central Receiver Systems (CRS) has been developed. The implementation based on desktop workstation equipped with multiple Graphics Processing Units (GPUs) is motivated by the need to have an accurate and fast simulation environment for studying mirror imperfection and non-planar geometries [1].

- Secondly, a power-aware scheduling algorithm on heterogeneous CPU-GPU architectures, based on an efficient distribution of the computing workload to the resources, has been realized. The scheduler manages the resources of several computing nodes with a view to reduc-

ing the peak power. The two main contributions of this work follow:

  - the approach reduces the supply cost due to high peak power whilst having negligible impact on the parallelism of computational nodes.
  - from another point of view the developed model allows designer to increase the number of cores without increasing the capacity of the power supply unit [2].

- Finally, an implementation for efficient graph exploration on reconfigurable architectures is presented. The purpose is to accelerate graph exploration, reducing the number of random memory accesses.

Contents

List of Figures

List of Tables

Introduction

Heterogeneous multi-core architectures represent the future of high performance computing (HPC). Traditionally, to increase the performance of Supercomputers, designers just scale up the number of CPUs (Central Processing Unit). This because CPUs are general purpose and easy to program. However a large portion of the CPU area is exploited to realize a powerful control unit. This control capability is usually wasted on many compute-intensive problems. Since a lot of scientific, engineering and financial applications require that a single instruction is executed on multiple data, heterogeneous architectures are potentially more efficient than homogeneous architectures that require to fetch and decode one instruction for each data. This has led to equip large scale computing systems with heterogeneous accelerators based on commodity hardware components to speed-up data-intensive workloads [10]. For example, the new supercomputer Titan, developed by ORNL [11], uses an heterogeneous architecture composed of conventional 16-core CPUs and GPU accelerators to overcome the computational power achieved by the previous generation of supercomputers, leading to the development of powerful heterogeneous High Performance Computing systems.

Nevertheless, different applications have different needs towards the computing systems. Designing an architecture to address a particular computational problem leads to excellent result, but it would be difficult to design an architecture to address different types of problems and so far no "one size fits all" architecture has been designed [12]. For instance, some architectures offer several simple computational cores vs. fewer complex processors, some depend on multi-threading and some even replace caches with explicitly addressed local stores [4].

This thesis explores the capabilities and the limitations of heterogeneous ar-

chitectures based on GPUs and FPGAs (Field Programmable Gate Array), to address some real-world "computationally hungry"problems. As target applications this work focuses on data-intensive computational problems. Initially the thesis focuses on the acceleration of complex 3 dimensional optical problems. These problems are based on intensive floating-point operations which can be efficiently implemented on a multi-GPU platform, since GPUs have evolved in highly parallel floating-point processors. In particular the purpose of this work is to speed up the computation in the design and analysis of concentrating tower power plants which represent one of the best ways to harness solar energy on a large scale. The design and optimization of these systems is quite complex and time-consuming, because it requires several design parameters to be considered at different time steps. The contribution of this work is a new simulation environment based on heterogeneous multi-GPU systems. The framework supports tuning of the trade-off between accuracy and computational time to obtain an analysis consistent with the precision required. The parallelism level in multi PCI-express workstations is usually limited by the power consumption. These systems could theoretically host from 2 up to 4 PCI-express devices (e.g. GPUs) for each CPU. However, to equip these systems with a number of GPUs equal to the number of PCI-express slots, could lead to system failure caused by power capacity overload. In particular, the multi-GPU workstation used could theoretically host 4 GPU cards, since 4 PCI-express slots are available. However the system is equipped with 3 GPU cards because with the fourth GPU the specification on the maximum power consumption provided by the power supply unit will be exceeded. Therefore the analysis has been moved to techniques to increase the parallelism on multi-GPU platforms limiting the maximum power consumption of the system. In this scenario, the attention has been focused on a job-level scheduling algorithm that aims to reduce the worst case power condition below a predetermined budget. The main idea is a redistribution of the workload between nodes of the system in order to avoid concurrent execution of the most power-consuming jobs on the same node. The approach used in this work aims to limit the total power consumption of the system under a prefixed budget and allows designers to increase the number of cores without increasing the capacity of the power supply unit.

Finally the thesis discusses a reconfigurable approach for efficient graph exploration. Differently from the previously studied class of problems, characterized by an high ratio between arithmetic operations and memory accesses, this kind of problems (i.e. breath-first search (BFS)) is largely dominated by memory latency, therefore not suitable for GPU architecture. The methodology adopted in this work is based on a partitioning of the algorithm in different levels. This in order to exploit the fast on-chip memory available in FPGAs and avoid multiple and inefficient random memory accesses.

The rest of this work is organized as follows. Chapter 1 discusses the two heterogeneous architectures based on GPU and FPGA used during this research activity. The goal of this chapter is to explain the concept of heterogeneous systems, comprising systems set-up, architectures and programming models. Chapter 2 provides a brief outline of solar fields explored in this work, pointing out the parallelism of the computational problem, where the previously described architectures based on GPU promise significant improvements. The chapter also describes in details how the simulation environment has been implemented on a multi-GPU platform and the results obtained. Chapter 3 focuses on the GPU power consumption and the techniques used in literature to limit the total power consumption of the system under a prefixed budget. Then the approach proposed to reduce the peak power and to increase the parallelism without increasing the capacity of the power supply unit is illustrated. Chapter 4 discusses a new techniques for efficient graph exploration. This section describes in details how the algorithm could be implemented on reconfigurable architectures and the limitations of this approach. Section 5 contains a general discussion and some conclusions are drawn in Section 6.

Chapter 1

Heterogeneous Architectures

Microprocessors are the basic blocks of the information technology. Their performance has made incredible progress over the last 20 years, driven by transistor speed and energy scaling, as well as by the progress in system integration density. Transistor density increases by about 35% per year. Increases in the die size are less predictable and slower, ranging from 10% to 20% per year. The combination of these two effects leads to a growth rate in transistor count on a chip of about 40% to 55% per year. This trend is known as Moore's law [13] and it is shown in figure 1.1. While transistor density increases with the Moore's law, device speed scales more slowly and is governed by Pollack's Rule, which states that performance increases as the square root of increase in complexity [14]. Figure 1.2 shows integer performance increase of new micro-architectures against area increase from the previous generation micro-architecture, in the same process technology. In particular the figure highlights that if designer doubles the logic in a processor core, then it delivers only 40% more performance. In addition, a further increase of frequency could lead to power dissipation issues. Better results can be obtained with a multi-core micro-architecture. Two smaller processor cores can provide 70-80% more performance, as compared to 40% from a large monolithic core [3].
Therefore chip manufacturers start to build multi-core microprocessors. Each core delivers lower performance than a large complex core; however the total compute throughput of the system is much higher. Nowadays general purpose processors integrate 2 to 16 cores [15], and it is expected that in the future a single multi-core processor will host up to hundreds or

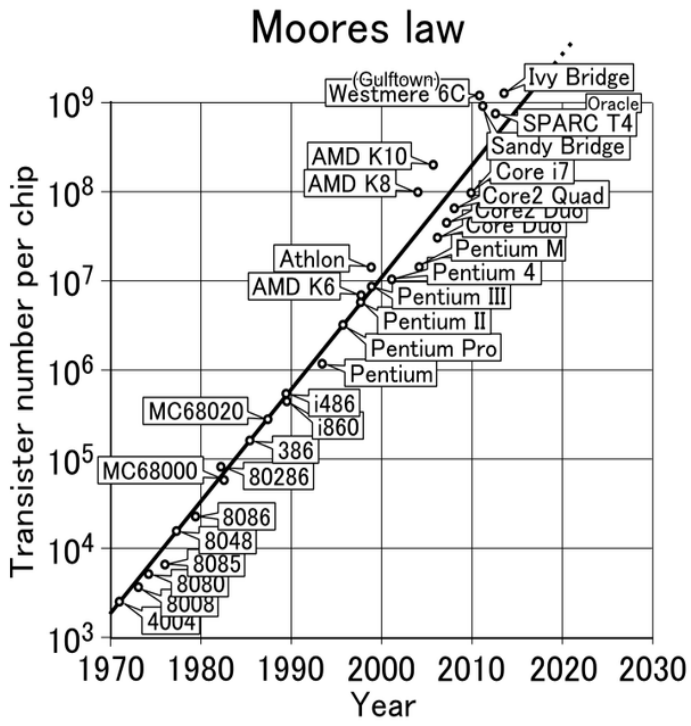


Figure 1.1: Transistor counts for integrated circuits plotted against their dates of introduction. (source: Wikipedia).

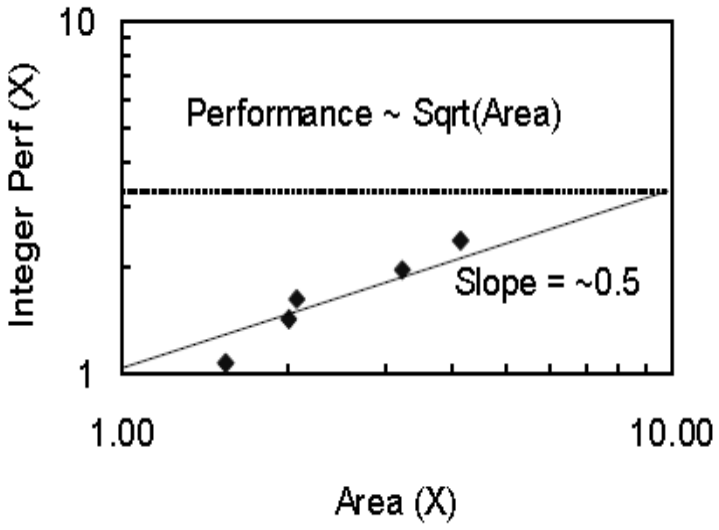


Figure 1.2: Pollack's Rule [3].

thousands of cores [16].

1.1 Classification of Parallel Architectures

Many scientific applications present a lot of concurrency, since the real world itself is massively parallel. Although multi-core systems deliver higher compute throughput than monolithic core systems, it may be difficult to harvest the performance. The performance gain that can be obtained by improving some portions of an application is limited by the Amdahl's law [17]. It states that the parallel speed up is inherently limited by the serial code in a program according to the equation 1.1 :

$$SpeedUp_{overall} = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.1}$$

where P represents the fraction of parallel code and N the number of computational cores. The speed-up scales with the number of cores as shown in figure 1.3. If the serial part of an application is large (i.e. 50%) the parallel speed-up saturates with a small number of cores (i.e. 16 cores).

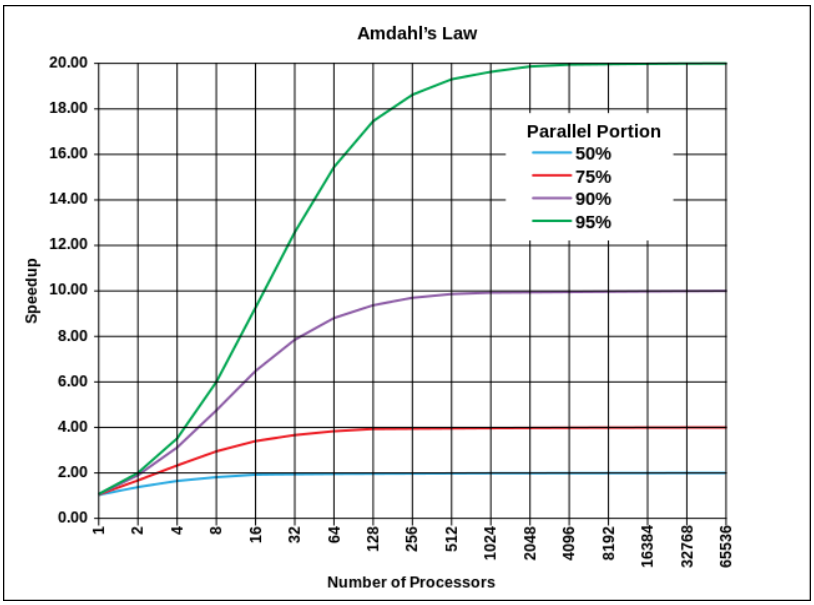


Figure 1.3: Amdahl's Law. (source: Wikipedia).

Few applications offer performance which scales with the number of processors. This usually happens when there are no data dependency between different streams. However most applications require communication between the computational cores. In this scenario data traffic and synchronization between nodes can be expensive. In general different computer

architectures can provide different performance according to the class of problems that programmers need to study. Homogeneous computer architectures are usually classified according to their processing units (PU) or their memory architectures. Concerning their processing units, the most common classification is the Flynn's Taxonomy [18]. This classification is based upon the number of concurrent instructions and data streams available in the architecture:

- Single Instruction Single Data (SISD)
- Multiple Instruction Single Data (MISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Multiple Data (MIMD)

Figure 1.4 shows the general principles of these architectures.

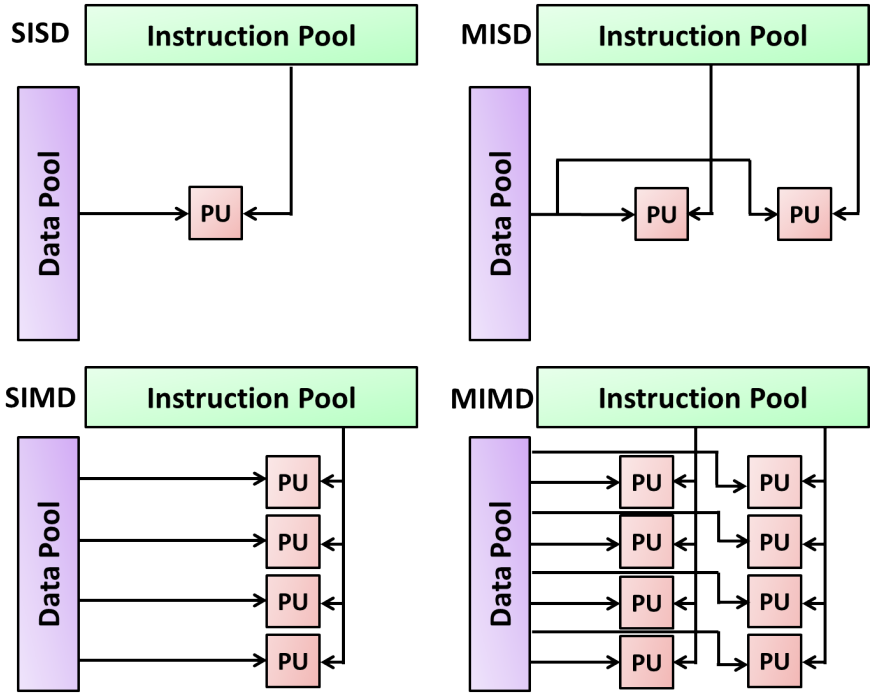


Figure 1.4: Flynn's taxonomy.

SISD represents a sequential computer which exploits no parallelism in either the instruction or data stream. A single control unit fetches single instruction stream from memory. MISD is a parallel computing architecture where many functional units perform different operations on

the same data. MISD is an uncommon architecture which is generally used for fault tolerance, and not many instances of this architecture exist. The classes of interest in computing are SIMD and MIMD. Multi-core general purpose processors for example are MIMD architectures. Multiple autonomous cores simultaneously execute different instructions on different data. They are suitable to perform parallel tasks as shown in figure 1.5 (each core independently and concurrently processes a serial task).

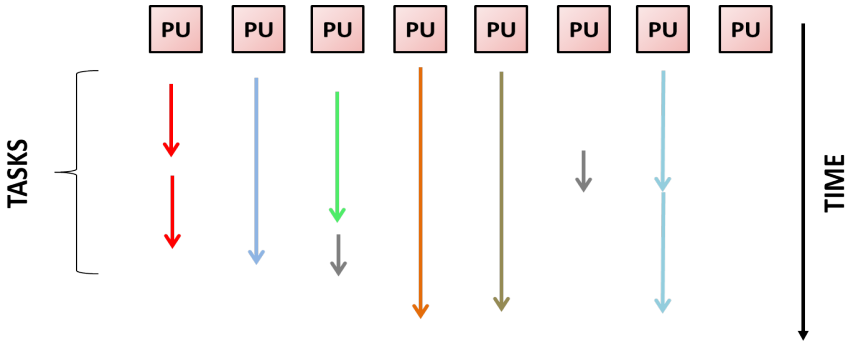


Figure 1.5: Multiple tasks executed on MIMD architecture.

Task-parallel processing is very useful to control different independent parts of a system. However a wide set of applications has significant data-level parallelism (e.g. matrix-oriented applications, media-oriented images, graphics processing and linear algebra). In this case an application can be decomposed into a large number of concurrent instruction streams which perform the basic operation on different data sets as shown in figure 1.6.

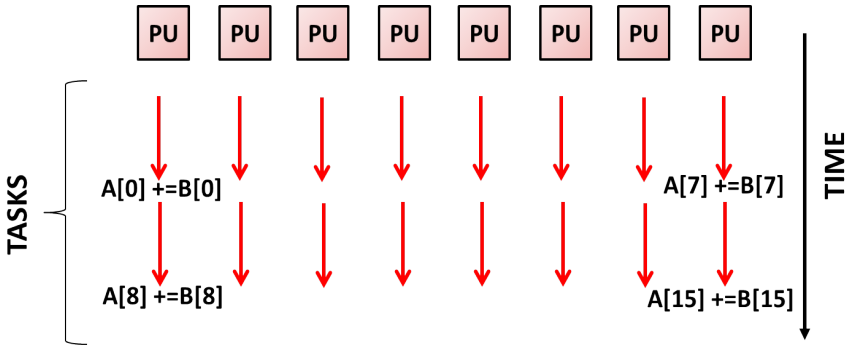


Figure 1.6: Multiple tasks executed on SIMD architecture.

Moreover, since a single instruction can lunch many data operations, SIMD is potentially more energy efficient than MIMD which needs to fetch and execute one instruction per data operation. In the past, this has led to

develop vector architectures which essentially are pipelined execution of many data operations [19] and later to SIMD instruction set extension for CPUs [20].
Computer architectures can also be classified according to their memory organization. Memory architectures can be distributed, shared and hybrid as shown in figure 1.7.

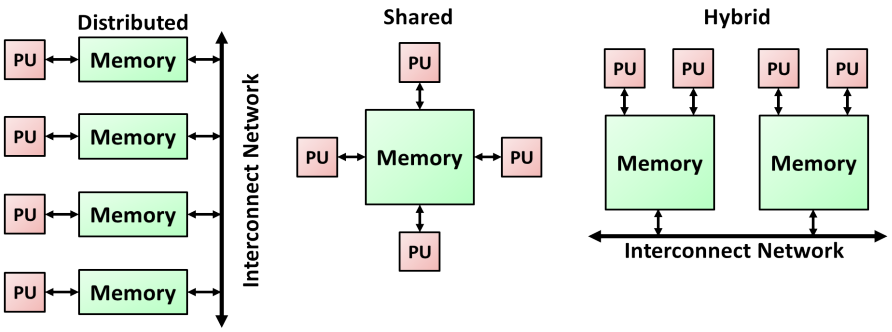


Figure 1.7: Memory Architectures.

Distributed memory provides a portion of private local memory to each core. Hence this kind of architecture ensures fast local memory accesses and is suitable for applications which present independent streams of data. However, communication between processors occurs through an interconnect network, which can become quickly the bottleneck for most applications. Moreover, when multiple cores are working on the same set of data, each of them has to do its own copy, increasing memory consumption and traffic. On the contrary in the shared memory model, the communication between computational cores is the fastest and multiple cores can share the same set of data. However this model does not scale well with the number of cores since data traffic and competition for resources can scale up geometrically with this number. The result is that most architectures adopt a hybrid mix of shared and distributed memory [21].

1.2 Heterogeneous Parallel Computing

Since different applications have different needs towards the computing systems, no universal multi-core accelerators have been designed so far. Usually CPUs are considered the closest architectures to universal multi-

core solutions because they present an high versatility. However in a CPU, a large portion of the overall area is used for control unit and cache, leaving a restricted area to computational unit.
In the past supercomputers were built just scaling up the number of CPUs and clustering them together using an high bandwidth interconnect. Recently power consumption has become the most critical issue. Supercomputers can reach up to tens of MW and in 2012 the supply cost over their useful life exceeded the initial capital investment [22]. For that reason heterogeneous architectures with combined traditional general purpose multicore processors and accelerators with a view to reducing the power consumption have become an appealing alternative.
Heterogeneous Architectures are designed to maximize throughput for specific application fields. In heterogeneous architecture a large part of the control logic is cut down. This means that they need an host processor which provides input data and then collects the results from the accelerators output.
The most common approach is therefore to combine few general purpose processors with a uniform and large array of accelerators [23]. The block diagram of a typical heterogeneous system is depicted in figure 1.8.

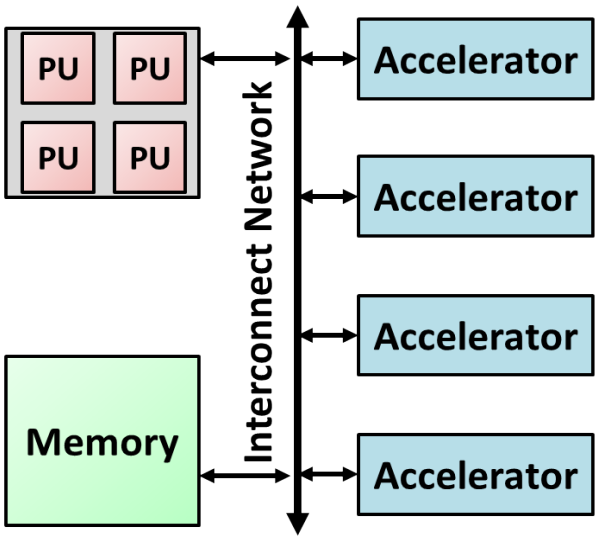


Figure 1.8: Block Diagram of a typical heterogeneous architecture

The most common general purpose accelerators are graphics processing

unit (GPU) and field programmable gate array (FPGA).
GPUs are massively parallel coprocessors for floating-point operations. They are usually integrated on a board with their own memory. The board communicates with the host system through PCI-express bus. Currently there are two manufacturers of GPUs: AMD [24] and NVIDIA [25]. In this work only NVIDIA GPUs have been used. The architecture of a GPU cluster is reported in figure 1.9.

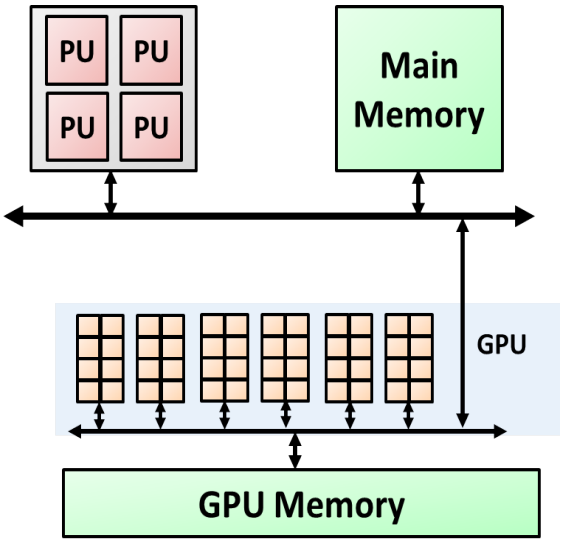


Figure 1.9: Heterogeneous architecture coupling CPU and GPU; in yellow are GPU computational cores

In a GPU application, a source code is transformed into a list of instructions for the computational cores, which is then loaded into the memory as shown in figure 1.10. Processors contain caches, forwarding and prediction logic to improve the efficiency of this paradigm. Performance of these architectures (i.e. called control-flow architectures) depends on the latency of the memory access and the clock speed.
Also FPGAs have a significant mark share in high performance computing. FPGAs can be reconfigured many times to address different problems. This characteristic makes them the most versatile accelerators. However they are hard to program, which is usually done using hardware description languages (HDLs). Rarely FPGAs are used as massively parallel floating-point units, but excellent results using these devices have been reported for pattern matching [26], encryption [27], and signal processing applications

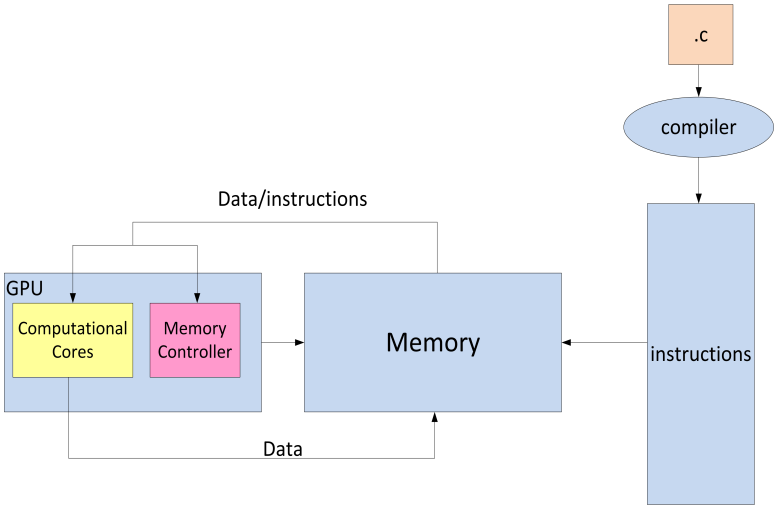


Figure 1.10: Control Flow Architectures (GPU)

[28]. A system using an FPGA as accelerator is depicted in figure 1.11.

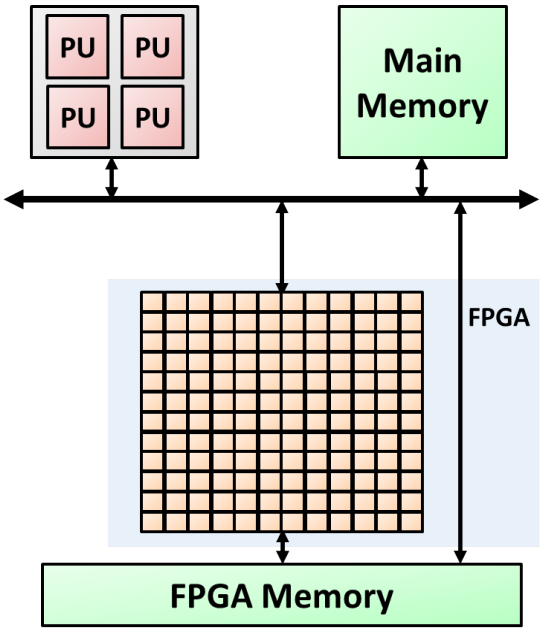


Figure 1.11: Heterogeneous architecture coupling CPU and FPGA

In an FPGA, data streams from memory into the chip where data is forwarded directly from an arithmetic unit to another until the chain is complete as shown in figure 1.12 according to a data-flow paradigm. Once a program has been executed the FPGA can be reconfigured for a new application. Performance of this architecture depends on how the arithmetic

units are connected together.

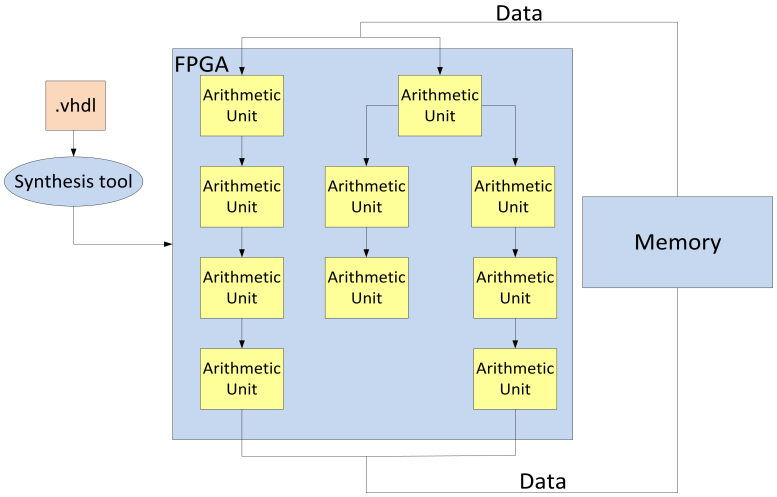


Figure 1.12: Data Flow Architectures (FPGA)

The next sections explain more in details how GPUs and FPGAs work in heterogeneous systems. Performance of GPUs and FPGAs are related to the algorithm characteristics. An FPGA is in general superior over a GPU for algorithms requiring large number of regular memory accesses, while a GPU is superior for algorithms with variable data reuse [29].

1.3 Graphics Processing Unit (GPU)

Everybody with few hundreds of euros can buy a GPU which transforms a standard machine desktop in a powerful heterogeneous computing system. GPUs were initially designed as hardware accelerators for 3D applications. These applications involve intensive low precision floating-point operations with a very modest need for control flow and data caching. The market of GPUs has exponentially grown up, when their potential have been combined with a programming language that made GPUs easier to program. AMD and NVIDIA developed their application programming interfaces (API): "Close to Metal" now called "Accelerated Parallel Processing"(APP) [30] and Compute Unified Device Architecture (CUDA) [31]. They also provided new hardware features to make GPUs more appealing for general purpose computing. To date, these include:

- *IEEE-compliant Fast Double-Precision Floating-Point Arithmetic* – The lat-

est GPUs match the relative double-precision speed of conventional processors at roughly half the speed of single precision.

- *Caches for GPU memory* – While the GPU philosophy is to have enough thread to hide the DRAM latency, there are variables that are needed across threads, such as local variables. New GPU architectures include both an L1 data cache and L1 instruction cache for each multiprocessor. In addition an L2 cache shared by all multi-processors is available.

- *64-Bit Addressing and Unified Address Space for All GPU Memories* – This innovation makes it much easier to provide the pointers needed for C and C++.

- *Error correction codes* – to detect and correct errors in memory and registers.

- *Faster Context Switching* – The new architectures have hardware support to switch contexts much more quickly.

- *Faster Atomic Instruction* – New architectures improve performance of Atomic instructions. Hardware interrupts and other processors cannot read or store the same location concurrently.

Driven by this scenario, GPUs have evolved in highly parallel, multi-thread, many-core processors with an high computational power as shown in figure 1.13.
Throughput is measured in GFLOP/s that means billion of Floating-point operations per second. Values reported in the plot indicates the maximum theoretical throughput of the device, assuming that all computational units are continuously busy with multiply-add operations. Thanks to these features, GPUs are nowadays computational accelerators not only used for graphics, with capabilities far exceeding their original purpose. As an example the NVIDIA Tesla series of graphics cards does not even have a graphics output [32].
The following section covers different aspect related to GPUs, with particular emphasis to the devices used for the experimental researches described in chapters 2 and 3

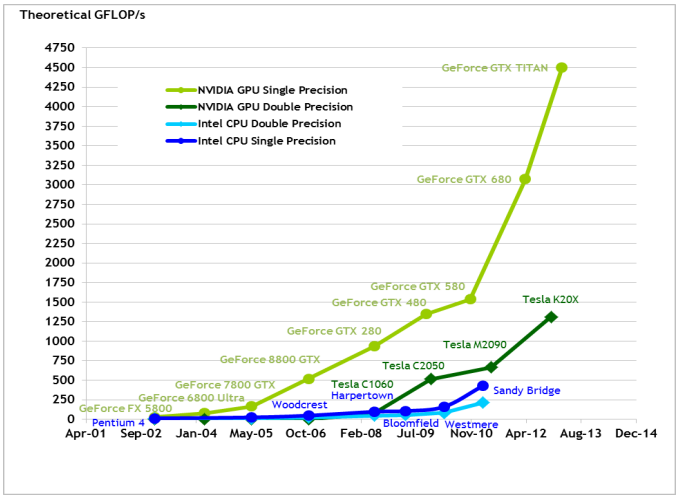


Figure 1.13: Floating-point operations per second for the CPU and GPU. (source: NVIDIA).

1.3.1 NVIDIA Fermi Architectural overview

GPUs work well only to address data-level parallel problems, since they are SIMD architectures. This means that instructions must be executed on many data elements in parallel. The reason behind this is that more transistors are devoted to data processing rather than data caching and flow control. Figure 1.14 shows the architecture of NVIDIA GeForce 500 series (codename: *Fermi*) used in the designs described in chapter 2 and 3.

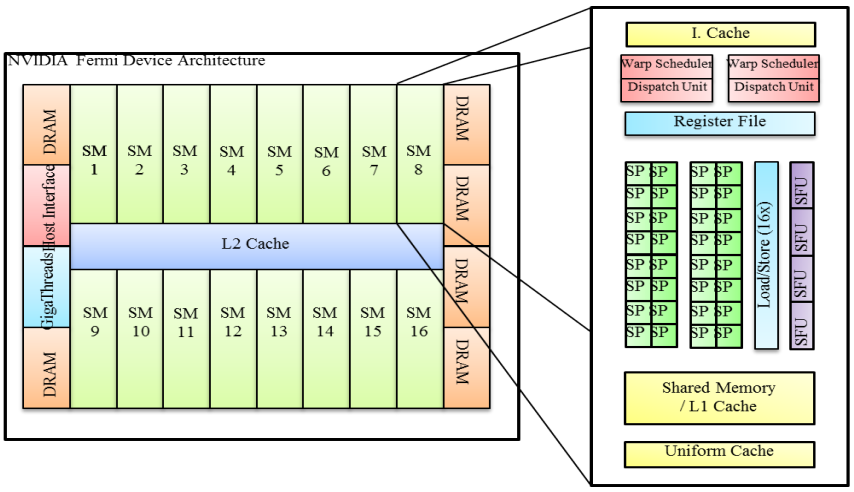


Figure 1.14: Block diagram of the NVIDIA GeForce 500 series.

It is composed of 512 cores, organized in 16 Streaming Multi-Processors

(SM) composed of 32 cores (SP), able to perform a total of 512 single precision or 256 double precision instructions per GPU clock. In addition each SM has four special function units (SFU) which perform transcendental instructions such as sine, cosine, square root, etc. To increase hardware performance with respect to the previous generation of GPUs, each SM has two thread schedulers and two instruction dispatch units.
This GPU has 3 different levels of memory, with different access times. For each SM there is an L1 cache of 64 KB that allows programmer fast data access. This memory can be divided into 2 sub-blocks, one of 16 KB and one of 48 KB, and is used as the shared memory of SM or local cache L1. The second level memory is a 768 KB L2 cache memory. Both L1 and L2 caches are used to cache accesses to local or global memory, including temporary register spills. Finally, the last level is called the device (or global) memory and is a 1536 MB off-chip memory, connected via a high-bandwidth interface. This memory has higher latency than the other two. The GPU cannot reach memory levels further down this memory hierarchy. Hence data must be uploaded to and downloaded from the GPU device memory via the PCI-express port. All control is exerted by the CPU side since the GPU cannot initialize any kernel launch or data transfer on its own. Care in using the on-chip memory is very important because it allows one to limit access to the off-chip device memory, since it is usually the bottleneck of GPU computation performance (see also section 1.3.3).

1.3.2 NVIDIA GPU Computational Structures

The CUDA programming model envisions hierarchical space decomposition to describe natural application parallelism. The basic element of the CUDA computation is the thread. It represents the smallest sequence of instructions that can be managed independently by the scheduler. A CUDA code (usually called kernel) is organized as a *Grid* of *Thread Blocks*, each one executing a certain number of parallel data threads.
As an example, suppose a programmer has to multiply two vectors, each 16384 elements long. The GPU kernel that works on all 16384 elements is called *grid*. The grid is organized in more manageable size, the *thread blocks*, each with up to 512 elements. With 16384 elements in the vectors, this example thus has 32 thread blocks since $16384 \div 512 = 32$. An overview of

the mapping is reported in figure 1.15.

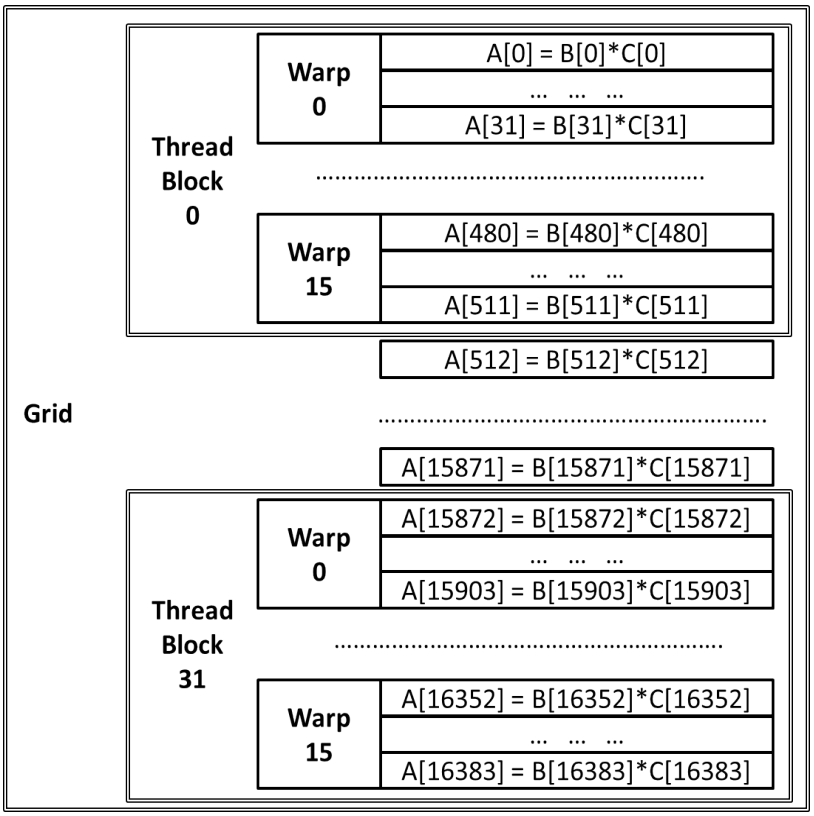


Figure 1.15: The mapping of a Grid [4].

These abstractions help programmer to organize the code. Thus a kernel performs multiple thread blocks, so that the total number of threads (i.e. 16384) is equal to the number of threads per block (i.e. 512) times the number of blocks (i.e. 32). Each SM processes the threads inside these blocks as an isolated problem, no synchronized communication between these blocks being provided. In this example, it would send 32 thread blocks to SMs to compute all 16384 elements.

GPUs have 2 levels of hardware schedulers; the thread block scheduler assigns thread blocks to SMs and ensures that thread blocks are assigned to the SMs whose local memories have the corresponding data. The second scheduler is within each SM which schedules when threads of instructions should run. In order to optimize the computation, the CUDA environment introduces *warp* meaning a group of 32 parallel threads which are simultaneously assigned to a SM. By contrast, thread blocks are assigned to different SMs. Hence, a thread block is a stream of vector instructions, and scalar

threads are the vector elements. In order to massively exploit GPU capabilities, the number of threads within a thread block should be composed of an integer number of warps. Each warp consists of 32 thread. In the Fermi architecture (see also figure 1.14) the number of cores is 32 so each warp takes one clock cycle to complete. In the previous example, *thread blocks* would contain $512 \div 32 = 16$ warps (see also figure 1.15). Since by definition warps are independent, the thread scheduler can pick whatever warp is ready. Figure 1.16 shows the Scheduler picking warps in a different order over the time. The assumption is that GPU applications have so many warps that multi-threading can both hide the latency and increase utilization of SMs.

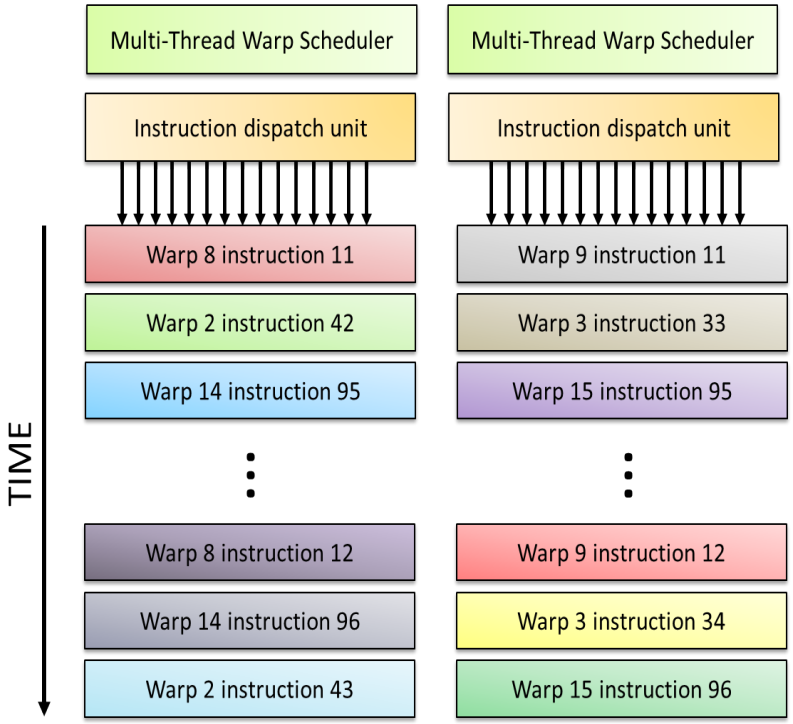


Figure 1.16: Block Diagram of Fermi's Dual Warp Thread Scheduler [4].

1.3.3 NVIDIA GPU Memory Structures

Figure 1.17 shows the GPU memory organization.
Understanding the memory architecture is essential because all GPU applications are inherently bandwidth limited. Since most of the area is devoted to arithmetic units, the amount of on-chip memory is limited. As shown in

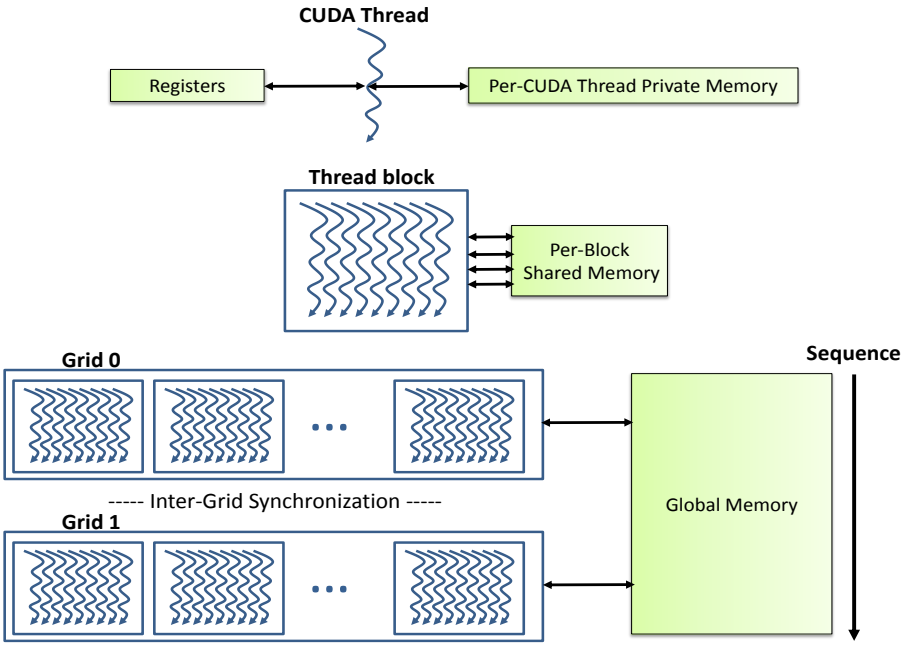


Figure 1.17: GPU Memory structures [4].

figure 1.17, three layers of memory are available:

- private memory;
- shared memory;
- global memory.

Each thread has some registers and a private memory. Registers can be accessed in two clock cycles. Private memory consists in a private section of off-chip DRAM and it is used to stack frame, spilling registers, and for private memory variables that do not fit in the registers. Recent GPUs cache this private memory in the L1 and L2 caches to aid register spilling and to speed up function calls. Private memory is freed as soon as the thread finishes.

Threads inside a block can access the same shared on-chip memory that each SM is equipped with. Resources allocated in the shared memory are freed once the block is completely finished. The shared memory can be accessed in two clock cycles if no access conflict occur. To increase bandwidth shared memory is divided into equally sized banks. Hence full parallel access is possible if threads from a warp access data without bank conflicts.

The condition of bank conflict vary depending on the GPU series.
Since the local memory is limited and the accesses to the device memory are slower, it is necessary to organize the code in order to find the best trade-off between the kernel length and the number of accesses to the global memory. Only data in global memory remain consistent for the lifetime of the CUDA context. Therefore, communications between blocks of the same kernel or between different kernels are obtained allocating space for results in the global memory using a shared memory computation model. Global memory has a latency between 400-600 cycles, which can usually be hidden by warp scheduling. The accessing pattern to global memory should be structured in a particular way to allow coalescing of the access, which means that data requested by a *warp* are transferred in a single memory transaction. Failing in this could lead to a performance drops up to 90% [33]. As for the bank conflict, the condition for coalescing vary depending on the GPU generation.

1.3.4 Power consumption

In heterogeneous CPU/GPU computing systems, GPUs are the most power consuming devices. Nowadays GPUs cannot be supplied via the PCI-express interface and they usually require additional power supply lines. In particular each card used in the application described in chapter 3 requires a power supply of 375W [2]. Therefore when multiple cards are packed on the same platform the power requirement rises to order of KW.

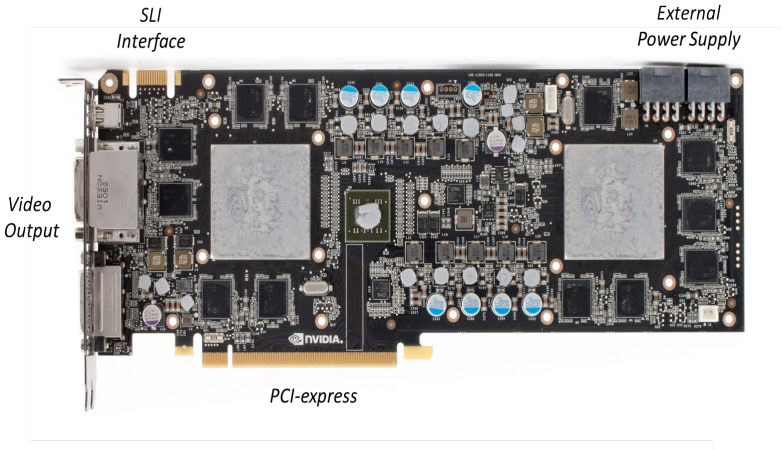


Figure 1.18: NVIDIA GTX590 Graphics card.

1.3.5 Programming the NVIDIA GPU

The heterogeneous programming model supported by GPU implies a system composed of a host (CPU) and one or more GPUs each with their own separate memory. Kernels operate out of GPU memory, so the runtime provides functions to allocate, deallocate and copy GPU memory, as well as transfer data between host memory and GPU memory [34]. A typical application flow is shown in figure 1.19.

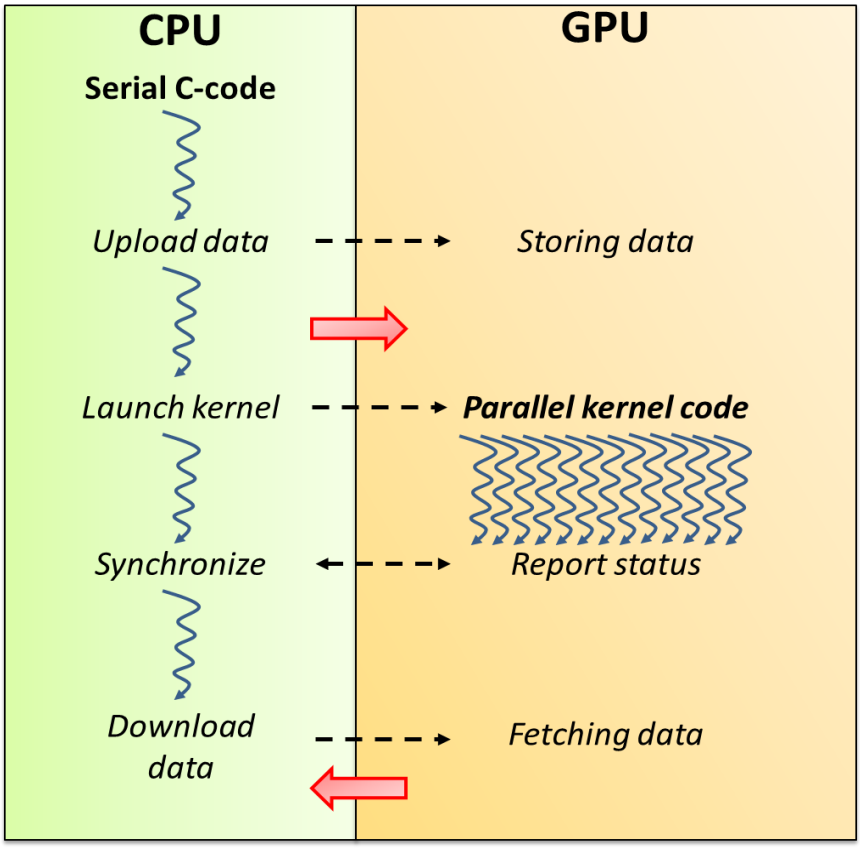


Figure 1.19: Heterogeneous programming of a CPU/GPU system.

1.3.6 Multiple GPUs

Multiple GPUs can work in parallel if the motherboard allows users to accommodate multiple graphics cards, as shown in figure 1.20.
Starting from the Fermi-class each GPU can map memory belonging to the other GPU into its global address space. To fully exploit the performance capabilities of a Multi-GPU system, CUDA has been used in concert

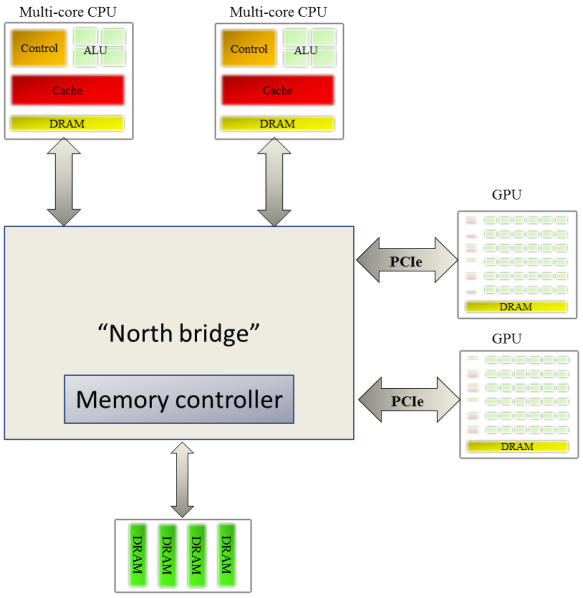


Figure 1.20: GPUs in Multiple Slots [5].

with OpenMp [35]. OpenMP is a multi-threading library based on shared memory principles. OpenMP runtime support, allows programmer to split the code into multiple threads via preprocessors directives. At the end of the region threads are joined again with the master thread. An example is shown in figure 1.21.

```
Float a[4], b[4], c[4];
int i;

#pragma omp parallel (private i)
{
   i = omp_get_thread_num();
   c[i] = a[i] + b[i];
}

printf(" c= %f %f %f %f\n ", c[0], c[1], c[2], c[3]);
```

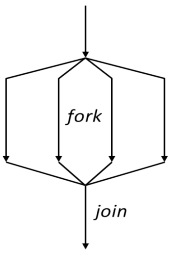


Figure 1.21: Parallel processing with OpenMP.

CUDA provides the application programming interface to the GPU, while OpenMP manages CPU multi-threading. Each GPU has a dedicated CPU core for handling and controlling. Other solutions are available, but OpenMP is portable, lightweight and it fits perfectly the needs of the designs presented in chapter 2 and 3. The software hierarchy is depicted in figure 1.22.

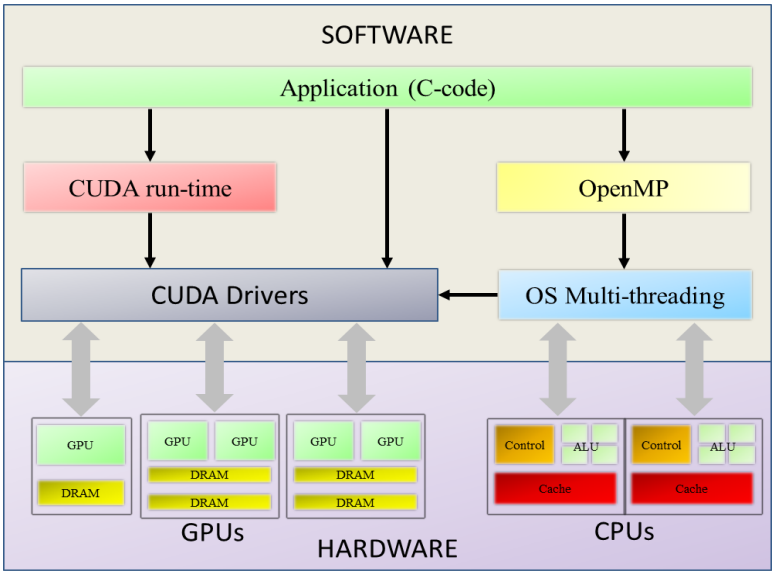


Figure 1.22: Software hierarchy and interaction with hardware.

A typical application starts a single master thread. Then a fork is created to assign one CPU thread per GPU. Within each thread a CUDA context is created; the context sets up the synchronization loop and initializes the runtime library. Then all CUDA API calls will refer to the selected GPU. Host Random memory access (RAM) is shared among CPU threads and with them also the CUDA contexts which can be used as buffer to exchange data between GPUs.

1.3.7 GPU applications

The introduction of CUDA has changed the world of high performance computing. New applications began to report speedups in the order of ten to hundred times over CPUs. However, not all applications can be efficiently accelerated on GPU. Basically, any application involving an high number of concurrent floating-point operations can greatly benefit from GPU acceleration.

One intuitive way to compare potential performance of a variation of architectures is the Roofline model [36]. It links together floating-point performance and memory performance introducing a parameter called arithmetic intensity. Arithmetic intensity is defined as the ratio of floating-point operations per byte of memory accessed [4].

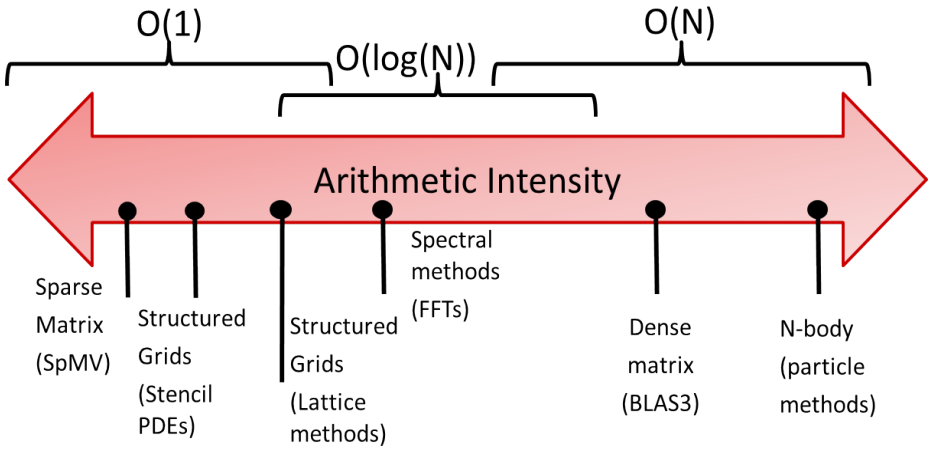


Figure 1.23: Arithmetic Intensity, specified as the number of floating-point operations to run the program divided by the number of Bytes accessed in main memory [4].

Figure 1.23 shows the arithmetic intensity of several computational problems. Some of these have an arithmetic intensity that scales with problem size (such as dense matrix) while some others have an arithmetic intensity independent of problem size (such as sparse matrix).
As an example, the Roofline model for a GPU NVIDIA TESLA C2050 (Fermi class) is shown in figure 1.24 [6].

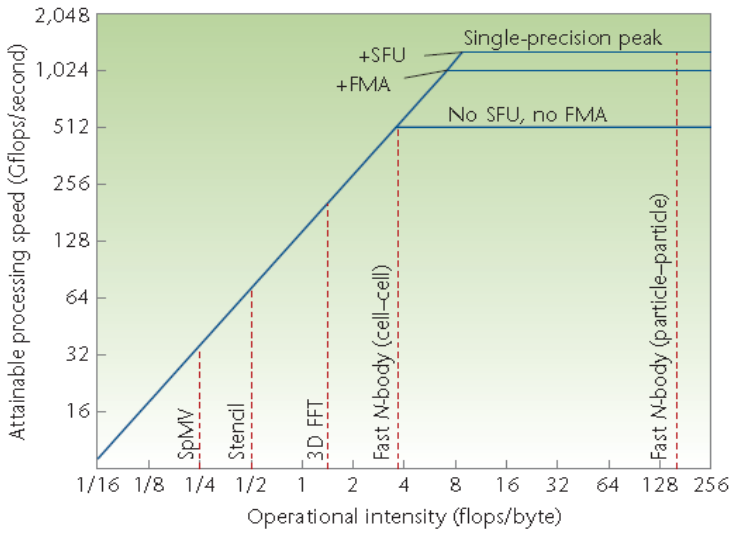


Figure 1.24: Roofline model of several kernels on an NVIDIA C2050 GPU [6].

The graph shows how the throughput changes, changing the application. The SFU label indicates the use of special function units and FMA indicates the use of multiply-add instructions. A sparse matrix-vector multiplication is labelled *SpMV*, a multigrid method with a seven-point-stencil is labelled *Stencil* and a 3D fast-Fourier transform is labelled *3D FFT*. As shown in the graph, only applications which present an high arithmetic intensity (also called operational intensity in figure 1.24) are capable to exploit the throughput provided by GPUs. This because the intensive computation hides the memory latency.

1.4 Field Programmable Gate Array

FPGAs were introduced 30 years ago and since then, they have become a popular implementation for digital circuits. Field Programmable Gate Arrays are devices which can be electrically programmed to became a digital circuits or systems. These devices represent a cheaper solution as compared to Application Specific Integrated Circuits (ASICs). In fact, FPGAs cost around a few tens to a few thousand dollars and they take less than a second to configure while ASICs require time and money in order to obtain first device. Moreover a portion of FPGA can be reconfigured while the rest of an FPGA is still running [37]. This flexibility is the main advantage of FPGA but at the same time the major cause of its drawback. This because flexibility makes FPGA larger, slower and more power consuming than their ASICs counterparts [37]. In general ASICs are convenient for large markets while FPGAs for small markets. An example of FPGA is depicted in figure 1.25.

It is composed of an array of programmable logic blocks (including general logic blocks, digital signal processors, multipliers and memory), connected by a programmable routing network. The routing network of an FPGA occupies 80-90% of total area, while the logic area occupies only 10-20% [38]. The entire array is surrounded by programmable input/output blocks which make possible the off-chip connections. The "reconfigurable" term in FPGAs means the ability to program a function into the chip after the fabrication process. This is made possible by the programming technology, which is a method that can cause a change in the behaviour of the chip after

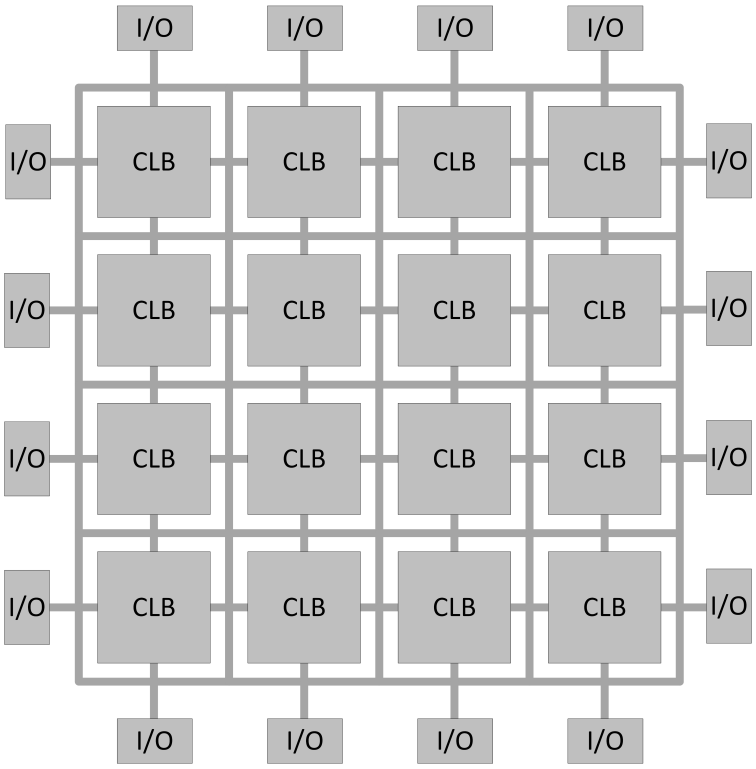


Figure 1.25: Overview of FPGA architecture [7]

the fabrication process [39].

1.4.1 Programming Technologies

An FPGA is configured using electrically programmable switches. Several programming technologies have been used so far and their proprieties dictate the trade-off in reconfigurable architectures. The most important are SRAM (Static Random Memory Access) programming technologies [7], flash [40] and anti-fuse [41].

1.4.2 Configurable Logic Block

The basic block of an FPGA is the configurable logic block (CLB), which provides logic and and storage for the target application. The purpose of a configurable logic block is to provide the computation and storage elements needed in digital circuits and systems. The CLB could theoretically be either a single transistor or an entire processor. Both the cases present their advantages and disadvantages. When a CLB is very fine-grained, it

requires a lot of programmable interconnect to create any typical logic function. This results in an area-inefficiency (because programmable routing is expensive in terms of area), low performance (each routing hop is slow) and high power consumption (since each interconnect has an high capacitance that must be charged and discharged). At the other extreme, when the CLB is very coarse-grained, it cannot perform small function without a massive wastage of resources. Usually intermediate solutions to implement efficient CLBs are used. Some of them includes basic logic blocks made of transistors [42], NAND gates [43], interconnection of multiplexers [44], lookup tables (LUTs) [45], and PAL-style wide-input gates [46].

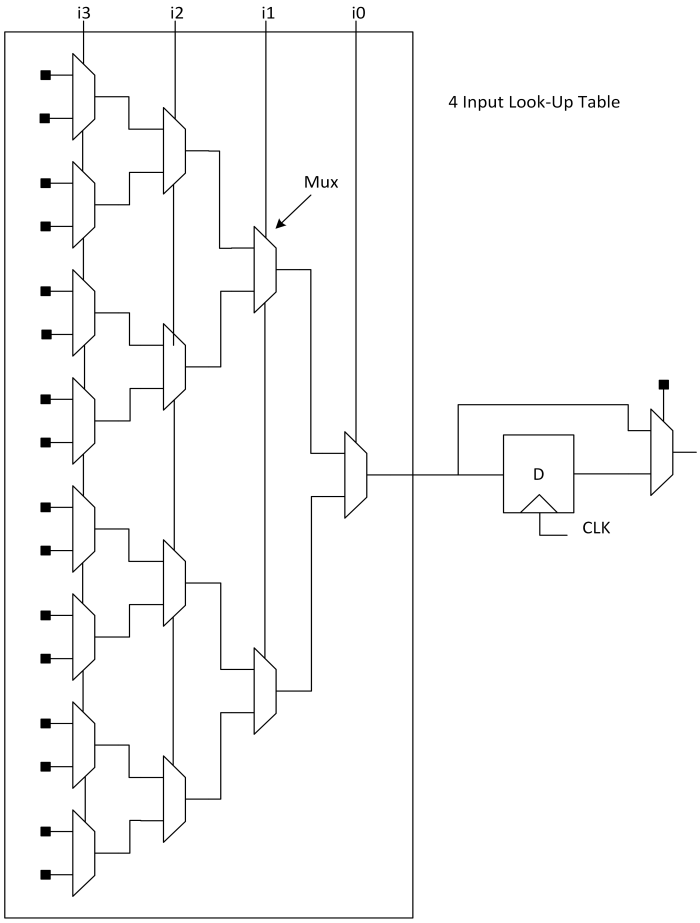


Figure 1.26: Basic Logic Element (source: Tree-Based Heterogeneous FPGA Architectures.)

The market's leader Xilinx [47] and Altera [48] use LUT-based CLBs. LUT-based CLBs provide a good compromise between fine-grained and coarse-

grained blocks. Figure 1.26 shows the architecture of a LUT-based CLB. It consists of a look-up table and a flip-flop. The LUT has k boolean inputs (i.e. 4) and 2^k configuration bits (16 SRAM bits). It can implement any k-input function. A multiplexer selects the basic logic element (BLE) to be either the output of the flip-flop or the output of the look-up table. Figure 1.27 depicts a cluster of BLEs. The output of each BLE is accessible to other BLEs through a local interconnect network. Usually CLB packs together 4 to 10 BLEs in a single cluster.

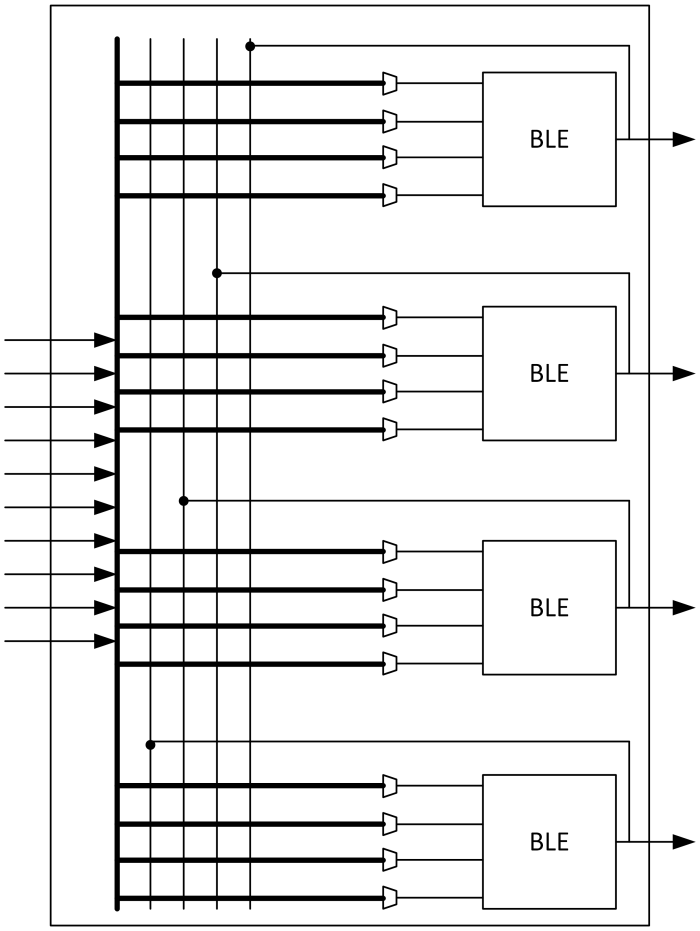


Figure 1.27: A configurable logic block (CLB) (source: Tree-Based Heterogeneous FPGA Architectures.)

In addition to standard CLB, modern FPGAs contain a heterogeneous mixture of blocks to address specific functions. These specific purpose blocks are DSPs, multipliers or dedicated memory blocks.

1.4.3 Routing Architecture

Configurable logic blocks are connected to each other through programmable routing network. This interconnect consists of wires and programmable switches that connect the logic elements. These switching are configured using the programmable technology.
The interconnect structure must be flexible to accommodate a high variety of circuits keeping the design speed as high as possible. Although different circuits need different connections, there are a lot of common characteristics of digital circuits which can be used to optimally design the routing network of FPGA architecture.
Most circuits exhibit locality, hence they require a high number of short wires, while at the same time they need at least some longer wires to support more distant connections. In addition, signals such as clock and reset must be widely distributed across the FPGA. Hence, care in designing routing interconnect is very important because routing has to address both flexibility and efficiency. The macroscopic arrangement of wires with no focus on the more microscopic switching between wires is called global routing architecture. The global routing architecture can be categorized as either hierarchical [49] or island-style [38, 50]. Currently, most commercial FPGA architectures use island-style architectures [51, 52, 53, 54]. This routing structure offers several advantages. First of all, an efficient connection for most designs can be achieved, since routing wires are in close physical proximity to logic blocks. In addition, the physical layout for each logic block can be optimized to form a single tile. As a result, the minimum feasible routing delay between blocks can be quickly evaluated.

1.4.4 Software Flow

One of the main aspect of FPGA research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. The software flow, also called CAD flow, takes an application described using a Hardware Description Language and converts it to a stream of bits used to configure the FPGA. This process can be divided into five steps, namely: synthesis, technology mapping, packing, placement and routing. Finally a bitstream that configures the state of the memory bits in an FPGA is pro-

duced. The state of the bits determines the logic function mapped on the FPGA.

Logic Synthesis

The first step in the software flow is the logic synthesis [55, 56] which transforms an HDL description into a set of Flip-Flops and boolean gates. Several technology-independent techniques are applied to optimize the boolean network.

Technology Mapping

The purpose of technology mapping consists in finding a network of logic elements which implements the boolean network. In particular in an FPGA, technology mapping problem involves transforming the boolean network in k-input look-up tables and registers. The most common used tools for FPGA technology mapping are based on the FlowMap algorithm [57]. This algorithm is capable to find an optimal solution in polynomial time.

Packing

Usually the configurable logic blocks in a Mash-based FPGA are organized in two levels of hierarchy. The first level involves logic blocks which are k-input LUTs and flip-flops, while the second level consists of k logic blocks together to form a cluster. The packaging phase of the flow consists of forming groups of k logic blocks. Hence these clusters can be directly mapped to a logic element of an FPGA.

Placement

The purpose of placement algorithms is to determine which logic block should implement a logic function required by the circuit. The goal of these algorithms is to connect logic blocks close together in order to balance and minimize the wiring across the FPGA, or to maximize circuit speed.

Routing

The routing is the process where nets are assigned to routing resources avoiding to share one routing between more nets. The state-of-the-art rout-

ing algorithm is *Pathfinder* [58]. This algorithm operates according a graph abstraction G(V,E) of the routing resources. The routing problem consists in finding a direct tree in G which connects the source and the sink terminals. This process is quite complicate since the routing resources are limited.

Timing Analysis

The purpose of the timing analysis is to determine the speed of the circuits which have been placed and routed and to estimate the slack [59] between source-sink connections. This is done in order to find and fix the connections which slow down the circuit.

Bitstream Generation

Bitstream information is generated from the netlist after the placement and routing processes. The bitstream contains information as to which SRAM bit be programmed to 0 or 1. It is configured on the FPGA using a bitstream loader.

1.5 FPGAs for High Performance Computing: The Maxeler Solution

Recently FPGAs have expanded their application area to HPC. Several HPC vendors offer FPGA-based high performance computing solutions. These boards are socket compatible with Intel and AMD processors using high-speed bus. To improve the bandwidth, these boards offer on-board memory. In the design presented in chapter 4, the HPC system based on CPUs and FPGAs proposed by Maxeler [60] is used. It consists of a combination of synchronous dataflow, vector and array processors. Figure 1.28 depicts the architecture of a Maxeler dataflow system which comprises FPGAs with their own local memories, connected to a host through PCI-express or Infiniband. Each FPGA can access two types of memory: on-chip memory which can store several Mega-bits of data accessible with an high bandwidth and a large on-board memory which can store many Giga-bytes of data off-chip. However, one important limitation is that the on-board memory works in burst mode. This means that at least a block of 384 byte of data

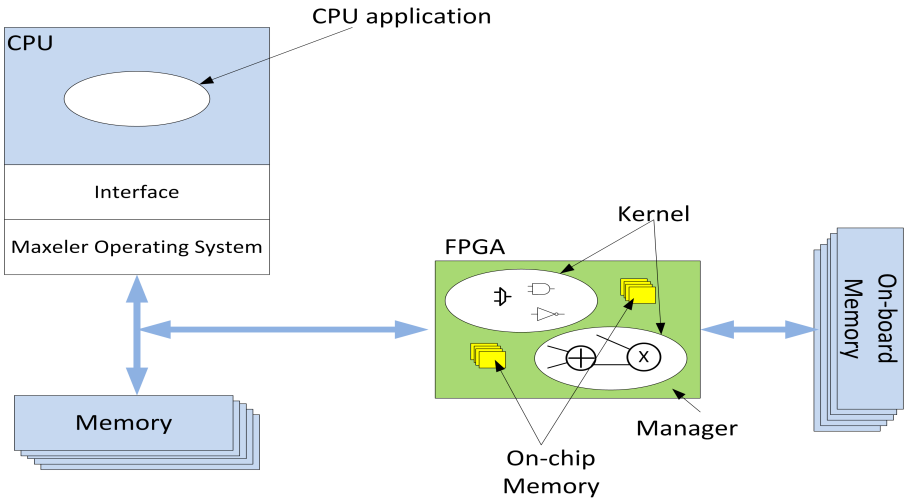


Figure 1.28: FPGA-based computing architecture (source: Maxeler.)

must be read even if just 1 bit must be modified. This memory access provides excellent performance when data are read in sequential order, but poor performance when data are read randomly.

1.5.1 Programming the FPGA using MaxCompiler

A Maxeler supercomputer consists of CPUs and FPGAs. The CPUs run executable files while FPGAs run a configuration file which contains the bitstream information. In particular, a general application consists of a small pieces of source code running on CPU and a large amount of data running on FPGA. In order to create the FPGA configuration file, a domain specific language called MaxCompiler is used. It is a Java-cased programming language with additions for FPGA memory and data access. Therefore an additional preliminary level is added to the software flow described in section 1.4.4. The FPGA is programmed through one or more *kernels* and a *manager* file. Kernels are graph of pipelined arithmetic units, which implement the computation. Without loops in the graph, data flows from inputs to outputs. With loops in the graph, data flows in a physical loop inside the FPGA, in addition to flowing from inputs to outputs. The manager orchestrates data movement between kernels, FPGA memory and other input/output interconnects. The first step when using MaxCompiler is to design, debug and simulate the application using a standard computer. Then given kernels and a manager, MaxCompiler generates dataflow implemen-

tation which can be called from the CPU.

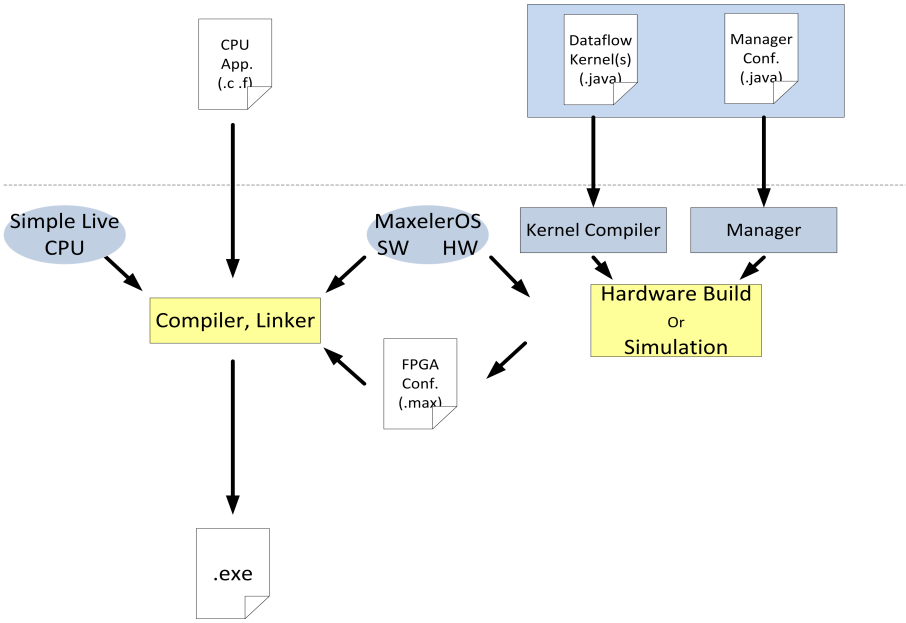


Figure 1.29: MaxCompiler chain (source: Maxeler.)

Figure 1.29 illustrates the development tool provided by MaxCompiler and how it works to build an accelerated application. Usually in a Maxeler supercomputer, multiple FPGA are connected together via a high-bandwidth interconnect as shown in figure 1.30.

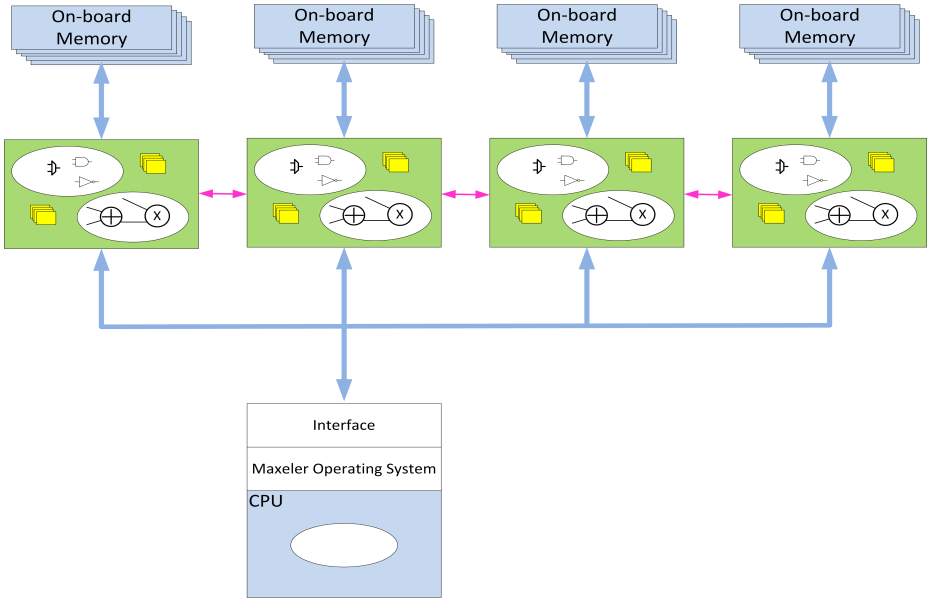


Figure 1.30: Maxeler dataflow system architecture (source: Maxeler.)

Chapter 2

Optical Model for Design and Analysis of Solar Field on Multi-GPU platform

This is the first in a series of three chapters presenting the major contributions of this thesis. They address the implementation of specific algorithms using different configurations of the systems described in chapter 1. The purpose of this chapter is to present an efficient implementation of optical model for design and analysis of solar field on Graphics Processing Units. The work presented in this chapter was carried out as part of the ERG project supported by ENIAC JU. This chapter follows the structure of [1].

2.1 Motivation and background

The depletion of fossil fuels, the increase in energy demand and the attention of public opinion to environmental problems are moving the production of electrical energy towards renewable sources. The sun is the most abundant source of energy on Earth and every year delivers more than 10,000 times the amount of energy that humans currently use [61]. Hence, the global solar electricity market is currently more than $10 billion/year, and the industry is growing at more than 30% each year [62]. One of the best ways to harness solar energy on a large scale is based on concentrating tower power plants [63, 64]. This system, also called Central Receiver Sys-

tem (CRS), consists of a field of highly reflective mirrors called heliostats which focus the solar radiation on an absorbent surface, positioned at the top of a tower. Solar radiation heats a fluid which is used in thermal or thermo-electrical processes as as shown in figure 2.1 and has been published in [1].

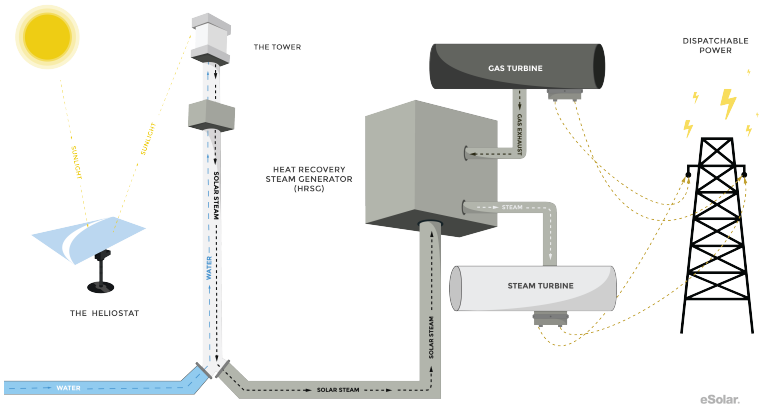


Figure 2.1: Block diagram of a Solar tower power plant (source: eSolar).

The design and optimization of these systems is quite complex and time-consuming, because it requires several design parameters to be considered at different time steps [65]. For example codes like MIRVAL [66] and SolTRACE [67] provide a detailed description of power reflected using a ray-tracing method, but they do not support field optimization. Accuracy of ray-tracing improves, increasing the number of rays, but unfortunately the computation time itself increases. UHC [68] and DELSOL [69] which use the convolution method contain more approximations in the peak flux computation, but they provide a quicker evaluation of the annual performance of a field [70]. These examples show that there is a trade-off between accuracy and simulation time in the existing design tools.

This chapter presents a simulation framework capable of supporting the design and analysis of solar fields for tower power plants with high accuracy and speed. The work is motivated by the need to analyze and optimize solar fields taking into account degradation factors such as shadowing, blocking, atmospheric attenuation, cosine effects, spillage, mirror reflectivity [71, 72] , in addition to mirror imperfections and non-planar geometries.

The need to reduce the computational time has been recognized by other

groups who have proposed novel algorithms for this purpose [73]. However better improvements can be obtained by algorithms that exploit parallel computing architectures based on graphic processing units, which, at a lower cost than standard servers, provide an accurate and fast simulation environment.

2.2 Mathematical model

The sun can be considered as a Lambertian surface appearing from the Earth as a disk of constant radiance [74]. However, considering a solar concentrating system on the Earth's surface, solar irradiation appears different from the ideal case due to the effect of the atmosphere on the sun's shape. In order to simulate major atmospheric effects, the model describes the sun as a Lambertian plane with a defined irradiation distribution. For simplicity's sake, a normal distribution will be considered, as suggested in [75], because it is simple to implement and introduces a limited error in the sun description. Solar distribution, supported by data available in databases (such as PVGIS [76] for Europe and Africa or TMY3 [77] for the USA), provides site-specified solar irradiation.
Let us consider the subsystem sun-mirror-absorber as shown in figure 2.2.

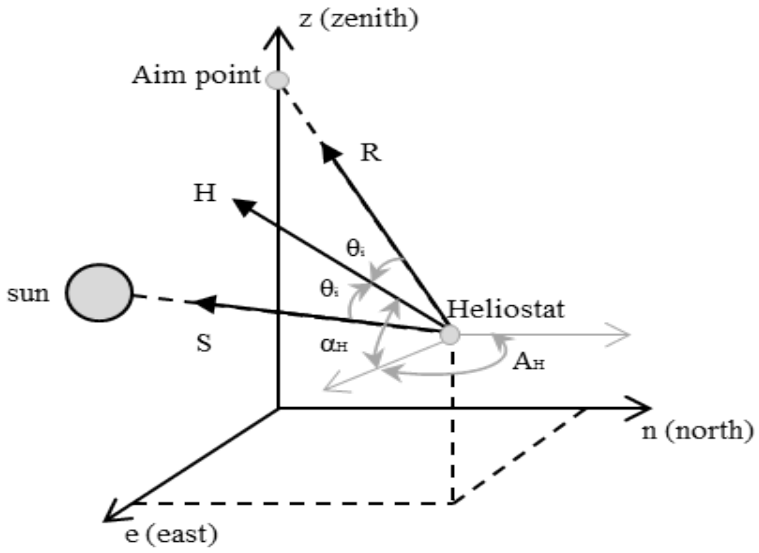


Figure 2.2: Reflection of the sun's rays by a heliostat to a single aim point.

Since Snell's law requires that the angle of incidence θ_i be equal to the angle

of reflection, the cosine of this angle can be derived from the scalar product between sun ray unit vector S and aim point unit vector R as reported in equation 2.1:

$$cos(2\,\theta_i) = S \cdot R \tag{2.1}$$

where vectors S and R can be written as:

$$S = S_z\widehat{i} + S_e\widehat{j} + S_n\widehat{k} \tag{2.2}$$

$$R = R_z\widehat{i} + R_e\widehat{j} + R_n\widehat{k} \tag{2.3}$$

S_z, S_e, S_n and R_z, R_e, R_n, represent the direction cosines, where i, j and k are unit vectors along z, e, and n axis. The direction cosines of S may be written in terms of solar altitude (α) and azimuth (A) as:

$$\begin{cases} S_z = sin\,\alpha \\ S_e = cos\,\alpha\, sin\,A \\ S_n = cos\,\alpha\, cos\,A \end{cases} \tag{2.4}$$

The reflection surface unit normal (H) can be found by adding the incidence and reflection vector and dividing by appropriate scalar quantity as shown in Equation 2.5:

$$H = \frac{R+S}{2cos\,\theta_i} = \frac{(R_z+S_z)\widehat{i} + (R_e+S_e)\widehat{j} + (R_n+S_n)\widehat{k}}{2cos\,\theta_i} \tag{2.5}$$

In the same way as shown in equation 2.4 it is possible to describe the direction cosines of unit vector H in terms of altitude (α_H) and azimuth (A_H) of the reflecting surface.

$$\begin{cases} H_z = sin\,\alpha_H \\ H_e = cos\,\alpha_H\, sin\,A_H \\ H_n = cos\,\alpha_H\, cos\,A_H \end{cases} \tag{2.6}$$

Heliostat tracking angles may be derived from equation 2.5 as shown in equations 2.7 and 2.8:

$$sin\,\alpha_H = \frac{R_z + sin\,\alpha}{2cos\,\theta_i} \tag{2.7}$$

$$sin\, A_H = \frac{R_e + cos\,\alpha\, sin\, A}{2cos\,\theta_i cos\,\alpha_H} \tag{2.8}$$

Once the correct tilt angles (α_H, A_H) have been found, the observer located on the receiver surface, looking in the direction of the mirror will see the sun reflected at an apparent position, called the apparent sun plane, placed behind the mirror as shown in figure 2.3 [78].

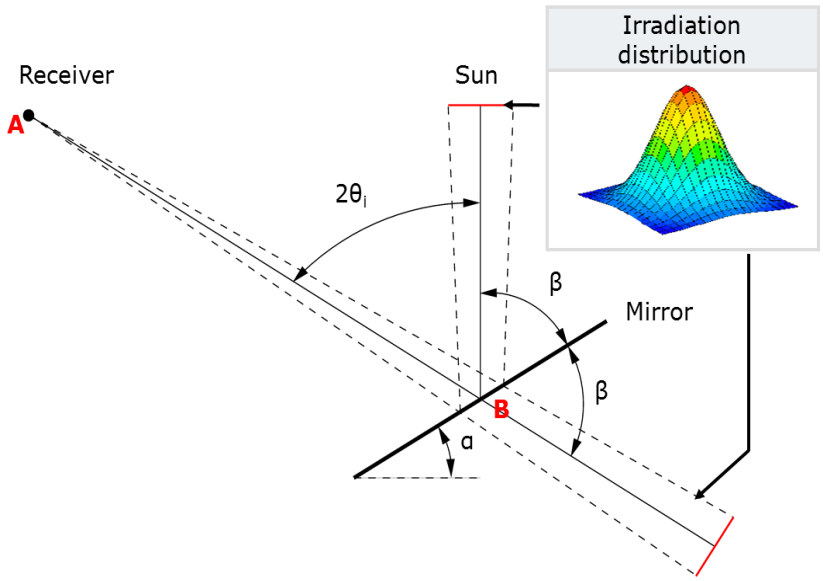


Figure 2.3: Geometrical model for optical simulation.

This plane is calculated as the plane located at a distance equal to the earth-sun distance, orthogonal to the line that passes through the reflection point and the mirror centre.

An observer sees the sun through the mirror and consequently sees the mirror edges as delimiting the sun disk. Therefore, to find the solar radiation collected by the mirror, it is necessary to project the mirrors corners onto the apparent sun plane as shown in figure 2.4.

Normally, central receiver systems are composed of hundreds or thousands of heliostats. Consequently, in order to project the mirror corners on the sun plane, it is necessary to verify the blocking and shadowing effects introduced by adjacent heliostats. In order to evaluate these two effects, each mirror is discretized into a matrix of square sub-mirrors. In evaluating the shaded area, the projection of each potentially occluding mirror onto the plane of the mirror under analysis is calculated, taking as observation point the center of the solar dish. A sub-element is then considered in the shad-

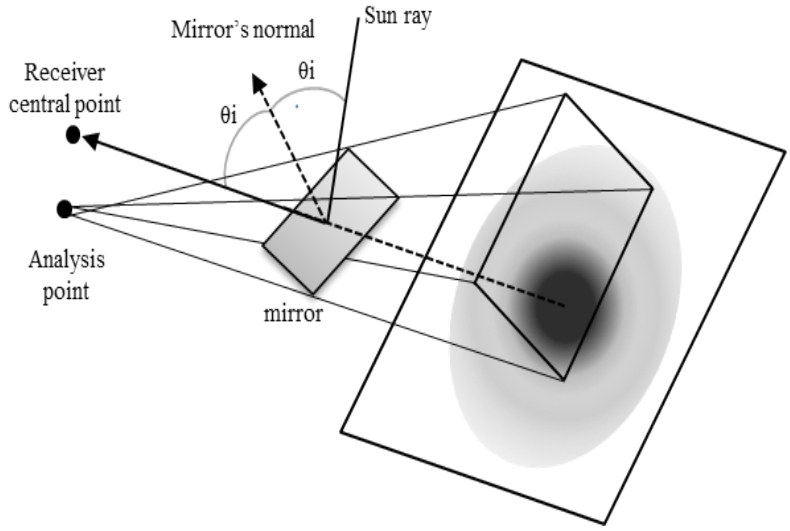


Figure 2.4: Simulation model geometry for a single receiver point analysis.

owing area if the neighbour mirrors' projection falls on it. Evaluation of blocking is similar to the detection of shadowing. The only difference is in the observation position which in this case coincides with the receiver's.
As mentioned above, the proposed model allows one to quickly evaluate field performances, composed of planar mirrors, or to obtain a more detailed description, considering mirror imperfections or non-planar geometry. In the study of planar square mirrors, the adjacent non-shadowed and non-blocked squares are joined up in larger squares, in order to calculate the minimum number of integrals. Otherwise, if a more accurate analysis is needed, it is possible to describe imperfections at the sub square level by introducing two local angles which tilt the sub-mirror from the ideal position. This model feature can also be used to describe non-planar geometries of mirrors. In this case the apparent sun plane must be calculated for each sub-element taking into account the different reflection point as shown in figure 2.5.
The solar radiation collected on the receiver by a heliostat will be given by the integral over the apparent sun plane of the flux density function. The limits of integration are the projected mirror corners on the plane.

$$P = \iint\limits_{\substack{Mirror \\ Projection}} F(x_r, y_r)\, dx_r\, dy_r \tag{2.9}$$

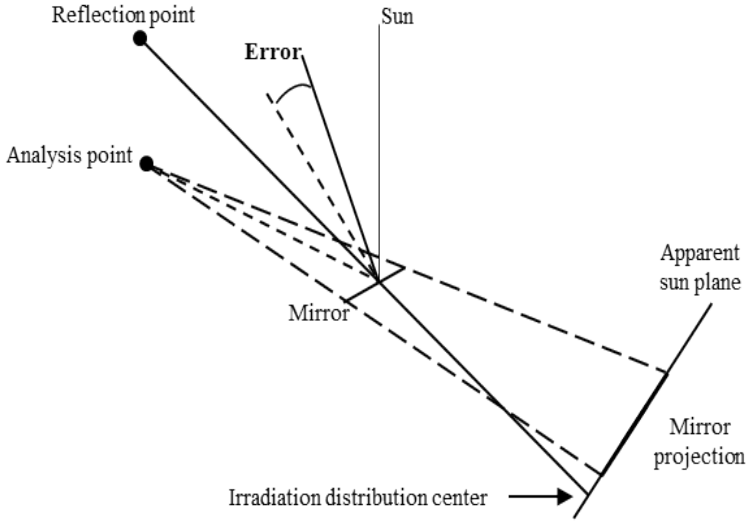


Figure 2.5: Geometric model for simulation with non-idealities in mirror surface.

where $F(x_r, y_r)$ is the flux density function.
Performing this operation for each heliostat, the total solar radiation on the absorber surface is achieved.

2.3 Computing System

The algorithm is implemented to run on a standard desktop machine with one or more GPUs. In particular, the configuration of the computing node used for this work is detailed as follows:

- Motherboard SuperMicro X8DTG-QF;
- Two Intel Xeon E5520 CPUs @ 2.27 GHz;
- 24 GB RAM;
- Two NVIDIA GTX 590 graphics cards.
- One NVIDIA GTX 480 graphics card.

The three GPUs and the two quad-core Intel CPUs share the same motherboard.

2.4 Implementation

The mathematical model for the system emphasizes a high level of parallelism due to the fact that each mirror is considered as a single element in the system. Thus, the solar flux distribution on the absorber surface is obtained by adding the flux collected by each heliostat. Depending on the specific target, the algorithm presents a different level of parallelism, as shown in figure 2.6.

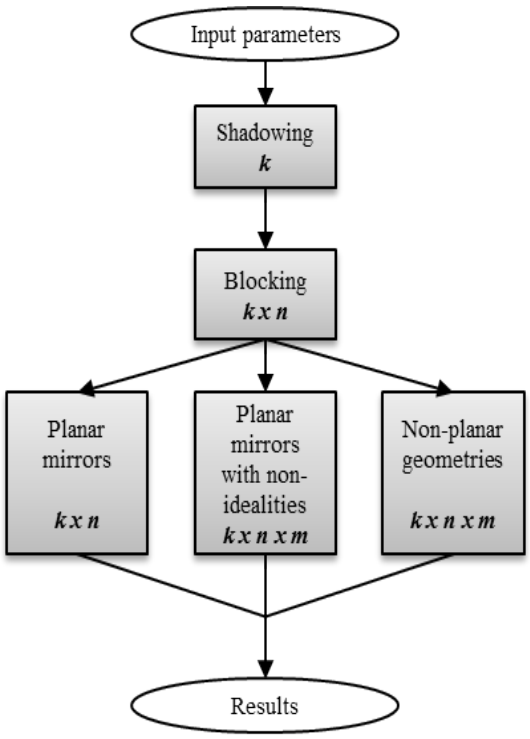


Figure 2.6: Flow chart of the algorithm.

To analyze the degree of parallelism of each stage in the overall algorithm, three parameters are introduced as shown in table 2.1: k represents the number of heliostats in the solar field, n the number of sub-elements used to discretize the receiver surface, and m the number of elements used to discretize each mirror surface.

Table 2.1: Nomenclature table of the algorithm

Heliostats k	Number of heliostats in the field
Heliostat grid $m = m_x \times m_y$	Number of sub-elements used to discretized each mirror surface
Receiver grid $n = n_x \times n_y$	Number of nodes used to discretized the receiver surface

2.4.1 Programming Model

The main objective of the first part of the algorithm is to study the interactions between adjacent heliostats such as shadowing and blocking. As described in Section 2.2, the first step is to calculate heliostat altitude and azimuthal angles which allow one to obtain the correct position of each mirror. This can be done if one knows the day of the year and the coordinates of each heliostat according to Equation 2.1, 2.7 and 2.8. Once the angles are known, the algorithm calculates the corner positions of each mirror (p_1, p_2, p_3, p_4) which define the heliostat outer edge.
In detection of shaded areas, the equation of the plane of the mirror being analyzed is calculated solving Equation 2.10:

$$\begin{vmatrix} x - x_1 & y - y_1 & z - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{vmatrix} = 0 \tag{2.10}$$

where x_i, y_i and z_i are the three components of the corner p_i. The outlines of other mirrors, which potentially cover the mirror being analyzed, are projected onto the plane previously calculated, taking the center of the sun as the observation point. The heliostat being analyzed is then discretized into a matrix of $m = m_x \times m_y$ elements. If the center of a matrix sub-element is contained in one of the neighbor mirror projections, the entire sub-element is considered to lie in the shadowed region and is not included in the peak flux computation, as shown in figure 2.7. This part, which is a computation of the positioning and shadowing interaction of a single heliostat in the field, is repeated for each heliostat. The parallelism level in this part is k.
Detection of a blocked area is similar to the shadowed case. The only difference is the observation point which in this case coincides with that of the

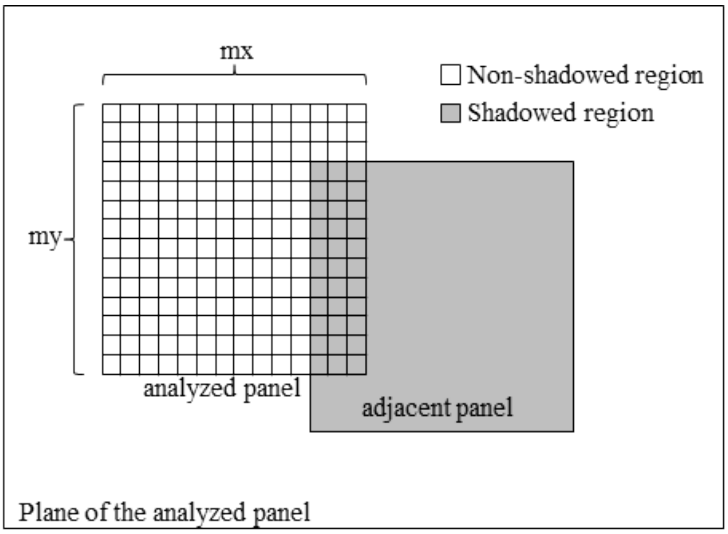


Figure 2.7: Shadowing evaluation.

receiver. Since the receiver is discretized in order to accurately assess the solar flux distribution over the entire surface, the analysis is repeated for each receiver point for each mirror. The level of parallelism for this part is $k \times n$.

Once the part of each mirror which actually contributes to collecting solar energy has been identified, the effective mirror outline is projected onto the apparent sun plane and the integration of the irradiation distribution is performed. As stated in Section 2.2, the apparent sun plane is calculated as the plane located at a distance equal to the earth-sun distance, orthogonal to the line that passes through the reflection point and the mirror (sub-mirror) center as shown in 2.3.

Having defined the equation of the apparent sun plane, the algorithm finds parameter values that fit the equations of the four lines that start from the analysis point on the receiver and pass through the four corners of the mirror (sub-mirror) as previously shown in figure 2.4. The cross points between the apparent sun plane and these lines define a polygon on the plane. This polygon represents the integration area. Once the integration polygon is defined in 3D space, the algorithm performs a coordinate space change. The goal of this operation is to pass from a 3D space to a 2D space in order to ease calculation of the integral in 2.9.

The solar flux distribution is numerically evaluated by Gauss-Legendre

quadrature. This part is characterized by a major level of parallelism, the analysis being carried out for each receiver point and for each mirror. In addition, the level of parallelism increases still further when the simulation involves geometries different from the flat one or when mirror non-idealities are introduced. In this case, all the instructions are repeated for each receiver point, for each heliostat and for each sub-element of the heliostats. Thus the number of integrals that can be evaluated in parallel increases to $k \times n \times m$.
The solar irradiance on a single mesh element is then calculated by multiplying the integration value by the incident radiation. This value is reduced by the cosine effect estimated considering the solid angle between the normal vector and receiver element, and the line that passes through the mirror center and the analysis point on the receiver. In order to describe the effect of mirror light absorption and non-specular reflectance, a scalar attenuation parameter is then introduced. Another scalar factor is also considered in order to model the attenuation that the light undergoes in crossing the path between the mirror and the receiver due to air scattering and absorption. This last term is dependent on the distance between each mirror and the receiver.

2.4.2 GPU kernels

The algorithms can be efficiently implemented on GPUs utilizing the data parallel programming model provided by the CUDA environment [31]. In order to extend the model to a multi-GPU environment OpenMP is used as multithreading library [35]. As mentioned in section 2.3 the computing system consists of 3 GPUs that are massively parallel processors, equipped with a large number of arithmetic execution units and 8 CPU cores.
For this algorithm the best solution found is to divide the GPU code into three different kernels as shown in figure 2.6, organized as a suite of interconnected routines, which make the code modular and easy to modify. This is due to the different degrees of parallelism in the code and the need to free the memory space used by the local variables.
Communications between different kernels is obtained allocating space for results in the device memory using a shared memory computation model. Since the parallelism is at the heliostat level and each mirror contributes in-

dependently to collect solar energy on the receiver aperture, each block of the same kernel can be executed independently of each other without any need of additional communication.
The purpose of the first kernel is to evaluate the shadowing effect between adjacent heliostats, while the second one evaluates blocking effects. This is done following the algorithm described in section 2.4.1. Shadowing is evaluated once for each heliostat (k), while blocking is evaluated for each receiver point, for each heliostat ($k \times n$), since the receiver is discretized in a matrix of n elements.
Results of the shadowing kernel consist in a set of k matrixes (once for each heliostat) of dimension m. Each matrix entry is set to 0 when the corresponding sub-element is covered by another heliostat and to 1 when the associated sub-element reflects the solar radiation towards the receiver. Since the blocking effects are evaluated for each one of the n receiver points and for each heliostat, results produced by the blocking kernel amounts to ($k \times n$) matrixes of dimension m. The effects of shadowing and blocking are combined in order to obtain the portion of reflective surface that collects the solar energy on the receiver aperture.
In order to limit the usage of device memory space, the output results are stored in vectors of 32 bit integers, where each bit corresponds to an element of the shadowing matrix. This mapping allows to reduce the memory space used to store the shadowing and blocking results of a factor 32 when compared to the outcome of using a memory location to store a binary number, as supported by the existing compiler.
Once these effects are evaluated, the third kernel that performs the integration of the irradiation distribution is executed.
The peak flux density on the receiver surface is stored in a matrix of floating point numbers of size (n). After the execution of the third kernel, the results are copied from the device to the host memory.
For the execution of the kernels, a thread block of size 64 is chosen. Results show that this is the best size to minimize pipeline latency as mentioned in [79]. In summary, the launching parameters for the three kernels are:

- Threads per block = 64
- Number of blocks

$k \div 64$ for shadowing kernels;

$k \times n \div 64$ for blocking and planar geometries integration kernels;

$k \times n \times m \div 64$ for other integration kernels.

2.4.3 Application Flow

The framework is executed in a MATLAB environment, which makes it easy to use and caters for plotting solar flux distribution on the receiver. As an input it takes a set of fields parameters describing the heliostats, tower and simulation parameters. Heliostat parameters consist of the coordinate of each heliostat, the support height, the size of each mirror and a mirror reflectivity coefficient. The code also provides two scripts that allow one to generate test fields. The first one generates a rectangular field based on a spatial distribution parameter, while the second one is based on radial stagger distribution [80]. Tower data consist of the spatial coordinates, size and tilt angle of the receiver aperture. The last data structure provides simulation parameters, such as the number of the receiver analysis points n, the heliostat matrix m, and the days and hours used to perform simulations. The user can choose whether to run a standard analysis or to perform an accurate simulation, introducing errors on each sub-element of the mirrors.

2.5 Validation

The code has been validated performing simulations of the PS10 field and comparing the results with the published data [81]. PS10 is a solar tower power plant, located near Seville and composed of 624 heliostats having an area of 121 m^2. The reflectivity of the mirrors is 0.88. The receiver is a cavity type one. Its center is located at 100.5 m with a rectangular aperture of 13.78 m $\times$ 12 m. Results are shown in table 2.2. Heliostat positioning was graphically obtained from [81].

2.6 Computational benchmarking

The framework is benchmarked on the target system and its performance is compared to that of a CPU-based system using a C language optimized implementation. To completely exploit the capabilities of the system, OpenMP

Table 2.2: Comparison between the GPU model and PS10 published data.

	PS10 declared	GPU evaluation
Concentrated flux 21-March 12:00 Irradiance $981 W/m^2$	51,953 kW	51,325 kW
Concentrated flux 21-June 16:00 Irradiance $831 W/m^2$	34,456 kW	34,403 kW

is used to share out the computation among eight different CPU cores (C language implementation) and among three GPUs (CUDA C implementation). All computations were performed in single precision floating point. A dense rectangular field layout, similar to that proposed by eSolar [82], has been chosen as target scenario. The field analyzed is composed of 12,000 heliostats of 1 m^2 without any physical non-ideality, located in the Libyan Desert (lat. $2585N$). A summary of the setup parameters is shown in table 2.3.

Table 2.3: Setup parameters of the simulation.

Field parameters	
Heliostat size	1 m $\times$ 1 m
Tower height	40 m
Receiver tilt angle	10 degree
Mirror reflectivity	0.93
Simulation parameters	
n-point Gaussian quadrature rule	8

Once the target scenario has been selected, it is necessary to adjust simulation parameters in order to ensure validity of results at the lowest computational cost. For this purpose the simulation environment allows one to set the number of rays used to discretized heliostats (m) and the number of nodes used to explore the receiver surface (n). These two parameters affect the accuracy of the simulation results in two different ways. The heliostat matrix determines the number of sub-elements used to discretize each mirror. Shadowing and blocking effects are evaluated once for each sub-element in the center point. Therefore an increase of m will enhance the accuracy in the assessment of power collected. On the other hand, the number of points used to discretize the receiver surface provides a measure

of how the solar radiation is distributed on the aperture. Thus an increase of n leads to higher accuracy in the evaluation of the solar flux distribution. As reported in [65], typical field performance optimizations are carried out varying the field parameters (e.g. tower height, heliostat dimension, etc.) on an interval of $\pm 30\%$ around baseline values. This leads to a variation in the optical efficiency of about 10%. In order to appreciate the effects of the tuning of field parameters it is necessary to ensure that the error margin introduced by simulation parameters m and n will be smaller than the variation introduced by field parameters. Consequently an acceptable error margin can be imposed at 10% of such value obtaining a value of 1%. Simulations parameters have been chosen following the procedure described below:

- The value of n has been set to 40×40 nodes, which is the same resolution used to evaluate the PS10 absorber surface in [81];
- The value of m has been initially set to 20×20 which is the maximum achievable discretization;
- A set of simulations has been performed at different time steps on the GPU based-system, in order to estimate the reference power values collected on the receiver;
- Simulations for the same field configuration have been repeated with lower values of the heliostat discretization matrix in order to obtain the minimum value of m that ensures an acceptable error margin.

Table 2.4: Error in the flux collected on the receiver aperture as a function of m.

Number of simulations performed	Heliostat mesh m	Maximum error %
72	20×20	0
72	16×16	0.4
72	10×10	0.7
72	8×8	1.4
72	5×5	3.8

Table 2.4 shows the trend of the error in the amount of power collected on the receiver surface as a function of the parameter m. The simulation parameter m has been set to 10×10, in order to obtain an error on the power

collected less than 1% for all time steps analyzed.
Table 2.5 shows that in this case a speed-up of 52× as compared to the 8-core CPU-based model is obtained for a field of 12,000 heliostats.

Table 2.5: Comparison between CPU and GPU execution times.

CPU-based model	GPU parallel model	Speed-Up
226.98 s	4.33 s	52 ×

Although analysis of solar fields performed with a CPU-based solver needs a tuning of the number of rays for each different field layout, the high computational power provided by GPUs allows to use high accuracy regardless of the size of the field analyzed. Therefore simulations for different field dimensions have been carried out with the highest value of the heliostat discretization matrix m (20×20) and the number of ray n previously set, in order to analyze how the execution times of the parallel model vary with respect to the problem size.

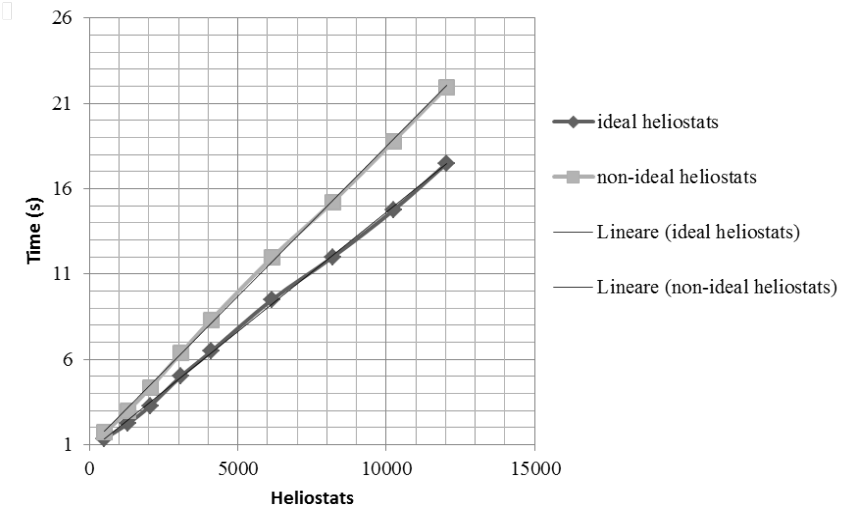


Figure 2.8: Comparison of GPU parallel model in a case of planar and non-planar geometry.

Figure 2.8 shows the run time with respect to the number of heliostats and the linear trend of experimental data for ideal mirrors and for heliostats with non-idealities. The GPU execution time includes every step of the overall computation, such as data upload and download from/to the central memory system. For ideal mirrors, the execution time of the GPU-

based model for fields up to 500 heliostats is constant because the parallelism provided by graphic cards is not fully exploited. For larger fields, the execution time increases linearly with the number of heliostats.
Contrary to the ideal mirrors case, where the matrix m (created to discretize each mirror) is only used to evaluate shadowing and blocking effects, in the analysis of mirrors with non-idealities, when increasing the value of m, the number of integrals that must be performed also increases as previously shown in figure 2.6. In this case the GPU resources are fully exploited even for small fields due to the high computational cost of the algorithm. In addition for this kind of analysis the arithmetic intensity (the ratio between floating-point operations and memory accesses) is higher with respect to the previous case since for each heliostat the number of integrals that must be performed grows to the size of m. This leads to a better exploitation of graphic cards which results in the small increase of time with respect to the previous case.
Rather than the execution time, the throughput in terms of floating-point operation per second is often considered a more expressive measure of the quality of an algorithm implementation as it gives insight into how well the hardware is being utilized. This is calculated from the amount of floating point operations required to process the given problem size divided by the execution time.
The scaling of GPU throughput is shown in figure 2.9.

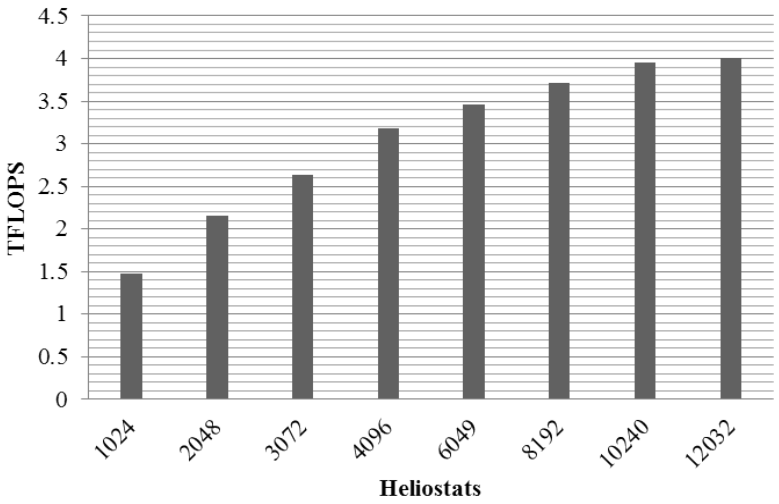


Figure 2.9: Comparison of throughput of our application in TFLOPS.

To evaluate the quality of our implementation, a comparison with the theoretical peak throughput of the hardware is helpful. The 2 GTX 590 and the GTX 480 have a theoretical peak throughput of up to 6.3 TFLOPS. This value indicates the theoretical maximum of the 3 devices, calculated by pretending that all computational units of the corresponding level of precision are continuously busy with fused multiply-add operations. A peak performance of 4 TFLOPS is obtained by our application. This number represents 64% of the theoretical peak and is based on execution times that include all overheads.

2.7 Application cases

2.7.1 Performance field analysis and optimization

Simulations geared toward optimization of a solar field were performed requiring an analysis repeated on several days of the year and at different hours of a day. Three different configurations of a radial stagger distribution field layout composed of 5000 Heliostats are studied. Each one of these fields was made up of 4 m^2 heliostats and a 16 m^2 receiver aperture. Even though these parameters can also be used in our optimization loop, a specific size of mirrors and receiver is presented. Analysis was performed on the 21st day of each month every hour from 7 am to 12 pm. Exploiting the symmetry of the day, the results obtained were replicated from 12 pm to 5 pm as described in [69].
The optimization loop is carried out by varying the radial staggered field aperture and tower height by $\pm$ 30% as suggested in [65]. Thus, the total number of simulations was 1296. A summary of the parameter analysis is shown in table 2.6.

Table 2.6: Parameter space used in the optimization.

Tower height	50 m; 60 m; 70 m; 80 m; 90 m; 100 m
Field Configuration	$(2/3)\pi$; $\pi/2$; $\pi/3$
Day	21; 52; 80; 111; 141; 172; 202; 233; 264; 294; 325; 355
Hour	7; 8; 9; 10; 11; 12

Since the simulations provide instantaneous flux distribution on the receiver aperture at a given time of the day (figure 2.10), by repeating the

simulation at each hour and interpolating the results the daily power on the receiver aperture is obtained, as shown in figure 2.11. Integrating this

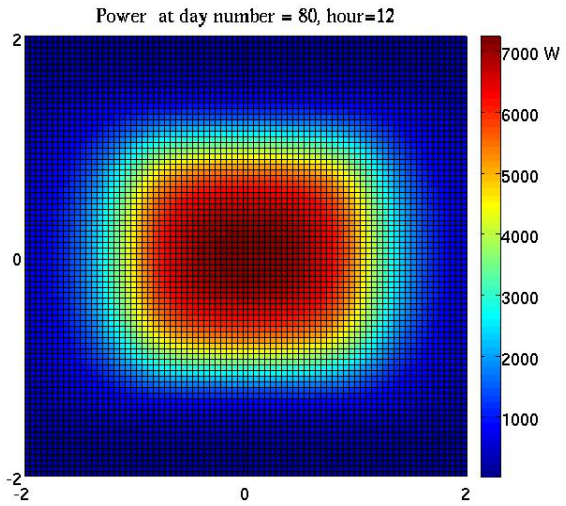


Figure 2.10: Power on the receiver surface at noon of 21st March.

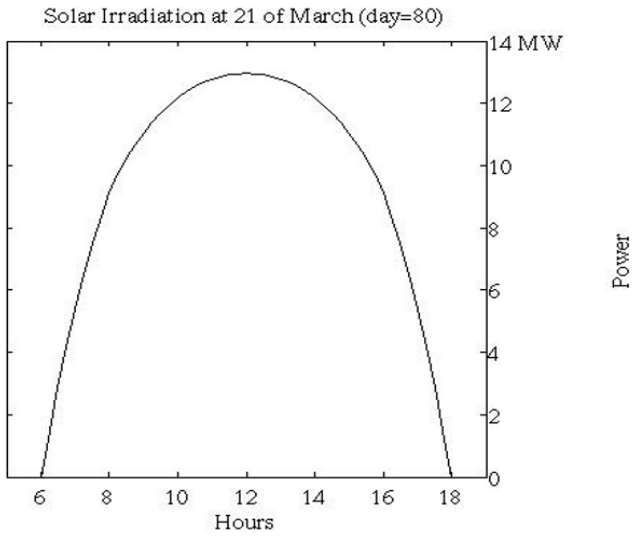


Figure 2.11: Solar irradiance collected by the receiver on March 21st

curve, the amount of daily energy collected by the field is computed. Annual field performance was obtained repeating the steps described above on the 21st of each month representing the average direct irradiance of the month. The time taken to perform these 1296 simulations was 48 min. Annual field performances for all configurations reported in table 2.6 are col-

lected in figure 2.12. Simulations show that the field with 100 m tower height and aperture π collects the highest energy. The variation in annual performance between the best and the worst simulated cases is about 6%.

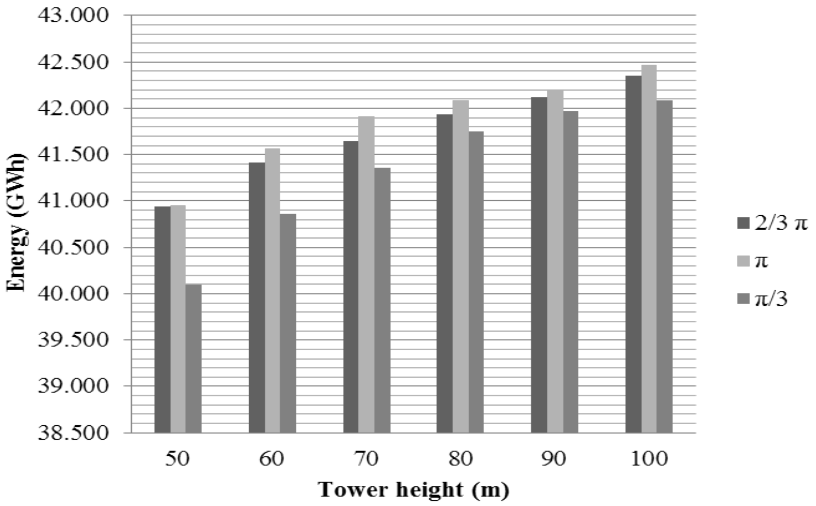


Figure 2.12: Annual field performance.

2.7.2 Analysis of mirror non-idealities

The analysis carried out in section 2.7.1 allows one to change field parameters and consequently to find the field with the best performance. However, in the real case, it is necessary to consider some non-idealities which characterize heliostats. Thus, once the best plant configuration is found, it is important to repeat the simulations for that field configuration, introducing these effects. In order to do this, statistical local slope errors are assigned to each sub-element of the $m_x \times m_y$ heliostat mesh with a deviation range of ± 5 mrad [83]. Results are reported in figure 2.13 and 2.14. Figure 2.13 shows an enlargement of the mark on the receiver. Figure 2.14 shows a performance degradation of about 6% obtained comparing annual energy produced by real and ideal heliostats. The computing time required to obtain these results is 15 min at a 6-fold increase vis-vis the ideal case.

2.7.3 Stretched membrane mirrors

The approach discussed so far can be extended to very general mirror shapes. In this section an example based on stretched membrane mirrors is shown.

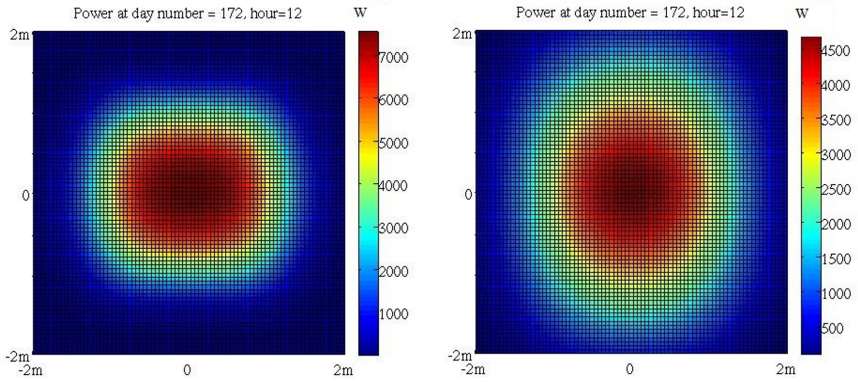


Figure 2.13: Power on the receiver surface for ideal (left) mirrors and non-ideal (right) mirrors.

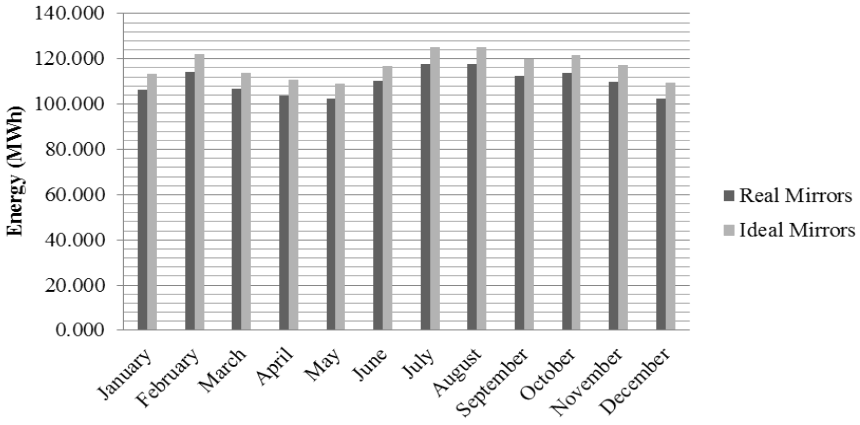


Figure 2.14: Comparison of energy collected by the field on a typical day, using ideal and real mirrors.

This mirror allows one to reduce the concentration area and increase the concentration ratio, making the approach applicable to photovoltaic concentration systems as well. As stretched membrane mirrors are simpler and cheaper than ideal parabolic ones, their performance is explored in this context. Simulations were performed for a radial stagger field layout composed of 100 stretched membrane mirrors obtained by stretching a high-reflectivity aluminum membrane such as "Miro reflective 90" produced by Alanod Solar [84]. The surface properties are reported in table 2.7. The field is composed of 4 rings of mirrors that reflect the solar radiation to the top of a 30 m high tower. Since the rings have a different radius, mirrors belonging to each ring have a different focal distance from

Table 2.7: Material properties.

Tensile strength (MPa)	160 - 200
Yield strength (MPa)	140 - 180
Elongation A 50%	≥ 2
Bending radius	≥ 1.5 fold thickness
Total solar reflectance %	90
Thickness in mm	0.3 - 0.8
Heat resistance	250 degree

the receiver. Hence, considering the ideal parabolic case, the depth d of the generic dish is related to its distance from the receiver F and its radius R by equation 2.11 :

$$d = \frac{(2R)^2}{16F} \tag{2.11}$$

In order to make the stretched membrane mirror approximate a parabolic dish the outer edge of the circular surface is constrained to a fixed position leaving it free to bend, and a tensile strength to the central ring S is applied in order to produce an elastic deformation equal to d at the center of the surface, as shown in figure 2.15a. Using a structural simulation software (Comsol 3.5a), the numerical model describing this 2D circular plate is solved for a radius R of 0.5 m and a thickness of 0.5 mm, obtaining the results shown in figure 2.15b. These structural simulations were carried out

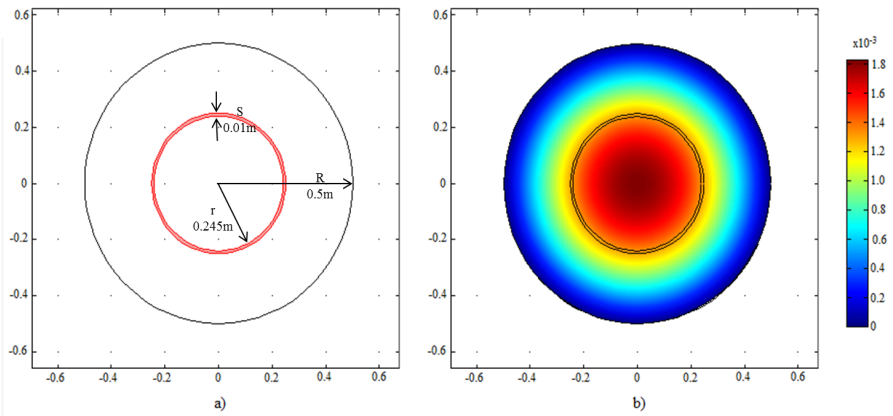


Figure 2.15: Simulated bending of a mirror realized with aluminum stretched membrane.

for different values of the tensile strength so as to find the different configurations suitable for each ring. Once the surface was simulated, the first

derivatives of the surface are approximated. Interpolation coefficients to the data structure of the heliostats are added and the procedure described in [85] is followed to find the correct tilt angle for each square. While a thorough analysis of key issues arising in the field of sun power concentration, such as uniformity of power distribution at the receiver, is not within the scope of this work, an example of simulation results is reported in figure 2.16. These simulations show that with such a change to the mirrors geometry, the total amount of power collected remains the same (53.7 kW in both the cases), but the spot on the receiver aperture is 8 times smaller. The computational time required to perform this simulation is 4 s.

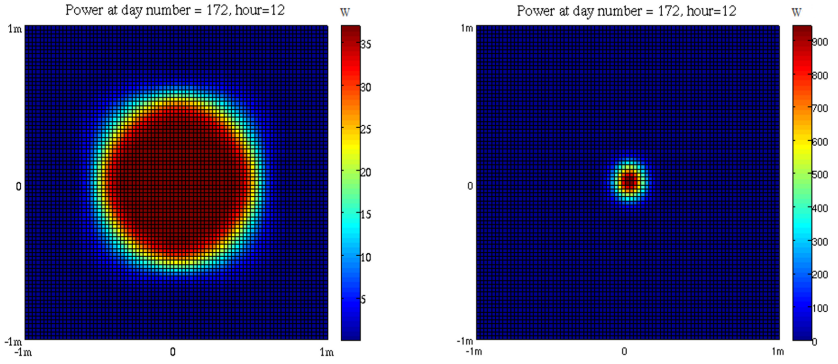


Figure 2.16: Receiver spot using flat and stretched membrane mirrors.

2.8 Conclusions

In this chapter a new algorithm and simulation environment for CRS based on heterogeneous multi-GPU systems is presented, built with low cost commodity hardware components. The model supports tuning of the trade-off between accuracy and computational time to obtain an analysis consistent with the precision required. Compared to an efficient OpenMP-based reference code running on two quad-core CPUs, a speedup of up to 52 is obtained with our implementation running on two GTX 590 and one GTX 480 graphics cards. Processing times for problem sizes that usually take several hours on efficient CPU-based solvers can be reduced to a matter of minutes, at the cost of an affordable hardware upgrade, making it possible to carry out an optimization and highly accurate physical description of solar fields.

Chapter 3

Power-Aware Job Scheduling

The desktop supercomputer used to solve the problem described in section described in section 2.3 is equipped with a power supply unit of 1400 W. Since each GPU consumes up to 365 W, add another GPU to the computing system could lead to system failure caused by power capacity overload. In this chapter an algorithm to increase the parallelism of the system without increase the capacity of the power supply unit is presented. A paper has been accepted with minor revision [2]

3.1 Motivation and background

As discussed in chapter 1 the computational power required by scientific, engineering and financial applications is unattainable on todays most advanced multi-core CPUs [86] and GPUs have been proposed as accelerators for intensive workloads in large scale supercomputers [10]. Although in heterogeneous computing systems the ratio between floating-point instructions and power consumption is higher with respect to homogeneous computing system, GPUs are power "hungry" devices and they require high external power supply in the range range of several hundreds of watts. Hence, the limitations of the power consumption is a key factor. In addition to supply cost abatement and positive environmental implications, the limitation of the worst case power scenario leads to a cost saving due to the lower complexity and capacity of the cooling systems needed.
Traditionally, supercomputers were designed to sustain the worst case operating condition. However this scenario is very rare and oversized power

supply and cooling systems involve additional costs. Thus in order to hold down the costs, supercomputers are nowadays designed with a better-than-worst-case policy [87]. In this situation the power consumption is constantly monitored and if the operating condition overlaps the predetermined power threshold, the power budget required by each node is adjusted to run safely under the maximum physical limitation.

Power capping defined as a strategy to limit peak power under a predetermined threshold is strongly influenced by the jobs activated on the computing system nodes. Hence techniques are needed which allow one to dynamically control the peak power while keeping system performance as high as possible [88]. In particular simultaneous execution of jobs (concurrency) leads to a performance enhancement effect, but also to an increase in power consumption. On the contrary, when concurrency is decreased, both performance and power consumption decrease. In this framework, this chapter discusses a job-level scheduling algorithm that aims to limit the worst case power condition below a predetermined budget during the concurrent execution of jobs in a heterogeneous computing system coupling CPU cores and GPU accelerators. The need for power-saving policies allowing control of power consumption, depending on the jobs being activated on the nodes, has already been recognized [89]. The open challenge is to find an effective way to reduce peak power while keeping parallelism as high as possible.

Several studies have been carried out addressing this issue. Some of them exploit dynamic voltage and frequency scaling (DVFS) in order to achieve a power reduction in HPC systems [90]. In [91] the approach presented was to reduce the clock frequency on nodes which had been assigned small computation load. The algorithm developed in [92] presents a power-aware DVFS run-time system that performs power reduction with small performance loss. Another work [93] provides a power-aware scheduling algorithm for applications with deadline constraint. In this approach DVFS is used to minimize power consumption meeting the deadline specified by users. However none of these approaches are designed to keep the power consumption under a preset threshold. Other methods exploit DVFS in order to keep the maximum power lower than a predetermined power constraint. In [94] power is shifted between resources, observing how they are

being used, while keeping the total power consumption lower than a given budget. Since frequency assignment is performed at a very fine grain, applying this approach to large scale systems could involve high overheads. In [87] the technique presented uses feedback control to keep the system within predetermined power constraints managing the CPU performance. The scheduling algorithm developed in [95] uses integer linear programming to assign a CPU frequency before executing a selected job in order to remain below the predetermined budget.

Even though DVFS is common for CPU-based infrastructures [96], it is relatively new for heterogeneous systems based on GPUs. For example the official driver provided by NVIDIA [34] is blind and it does not allow user to adjust the voltage and the frequency levels for computational kernels. Once the driver sees an application that is ready to be activated, the driver will increase the clock speed up to the highest level set by the user [97]. For enabling the DVFS users have to use open-source drivers such as Nouveau [98]. Unfortunately Nouveau driver do not allow hardware acceleration of 3D operations. Concerning AMD GPUs [24] the official driver supports OpenCL [99], but does not support voltage and frequency scaling. On the contrary the open-source driver called RadeonDriver [100] supports voltage and frequency scaling but does not support OpenCL [101].

It must also be noted that scheduling algorithms based on dynamic voltage and frequency scaling deliver a suboptimal response for short-time workloads because they rely on reaction instead of prediction [102] and for short workloads this reaction can occur after the transition. In this case the amount of performance loss is related to the number of transitions in the workload and the lag between request and capacity [102].

Approaches that aim to reduce power consumption according to the jobs being activated on the node have been explored. In order to do this, the power consumption of several library functions may be characterized for different CPU performance. For example, in [103] a power performance comparison between LAPACK [104] and PLASMA [105] libraries was made using the setup developed in [89]. The work presented in [106] uses the same measure setup [89] in order to develop a job-centric model. The purpose of these works is to understand how a program can be modified to improve performance with respect to system run-time and power consump-

tion. In [107] a method of profile-based power-performance optimization is presented. In this work a program is split into several regions and for each one the frequency which minimizes the power-performance ratio is selected.

All of these works are based on multicore CPU and not on heterogeneous CPU-GPU architectures. In addition, most of the algorithms described in the literature do not consider the concurrent execution of several jobs on different cores as a target to be optimized in order to keep peak power under a predetermined budget. However, in heterogeneous computing systems where GPUs are the most power-consuming devices, the simultaneous execution of GPU kernels may lead to overlapping high power profiles, causing generation of power absorption peaks which could be avoided with a smart distribution of the workload to the resources.

This chapter presents a predictive power-aware scheduling algorithm which provides a real-time allocation of computationally-intensive jobs to the nodes of a heterogeneous computing system, with a view to keeping the peak power under a predetermined budget, mitigating the worst case power condition. The algorithm can be used as a level of adjustable power state software services [108, 88] in order to provide an efficient solution during high-demand periods.

The basic idea behind the algorithm is to adopt a two-step approach. First, the power consumption of a GPU kernel library is characterized. Jobs activated on the system nodes utilize these kernels to accelerate intensive computational cores. From the user viewpoint this characterization does not affect the programming model at all. However, each time a new kernel is added to the library, its power consumption must be characterized. Second, this characterization is then used to develop a model capable of adjusting the start time of a job depending on its GPU kernel calls, and selecting the node on which to activate it, taking into account the jobs that are already running on the system. This approach limits peak power requirements and enables the system not to exceed the predetermined budget. This is achieved without performance reductions caused by frequency and voltage scaling as proposed in [87], since it is obtained by considering the different profiles associated with each kernel in order to avoid concurrent execution of the most power-consuming jobs on the same node. The

specific contributions of this chapter may be summarized as follows:

- A low-cost measurement system has been developed to extract the power profile of jobs running on heterogeneous computer architectures. This system has been designed to make up for the lack of standard hardware sensors in the computing nodes used as basic blocks of high performance systems [89].
- A power-aware scheduling algorithm to manage the resources of several computing nodes has been developed. The scheduler manages the start times and the nodes on which to run the jobs. The goal is to minimize peak power absorption (such as may happen during simultaneous execution of several jobs) while keeping concurrency as high as possible.
- A quantitative analysis has been carried out in order to demonstrate that the algorithm significantly reduces peak power requirements during parallel job execution, mitigating the worst case power condition.

3.2 Power Measuring System

In order to develop a job-level power model it is necessary to characterize the power consumption of the jobs performed on the system nodes. Common power profiling techniques allow one to measure the consumption of a single computer component such as a processor or a co-processor [109] [110] [111] [112]. Other setups have been proposed to perform fine grain profiling analysis of an entire system [89] [113]. However, they are based on expensive measuring instruments.
In [111] three options are discussed to measure the power consumption of a job running on heterogeneous CPU-GPU platform:

1. Measuring the power consumption of the GPU card;
2. Measuring the input of the power supply unit (PSU);
3. Measuring the output of the power supply unit.

Measuring the power consumption of the GPU card is the most direct way, although it is not precise because the GPU is a co-processor and needs a

CPU as a host. Thus the power measured is only a part of the power required to run the job. Measuring the PSU input includes the power loss by the power supply unit which may be 20% or more of the total power dissipation [111]. However, measuring the total PSU output will not allow one to understand how each computer component contributes to the total power consumption, whereas in order to develop a job-level model one needs to measure the power consumption of each system component.

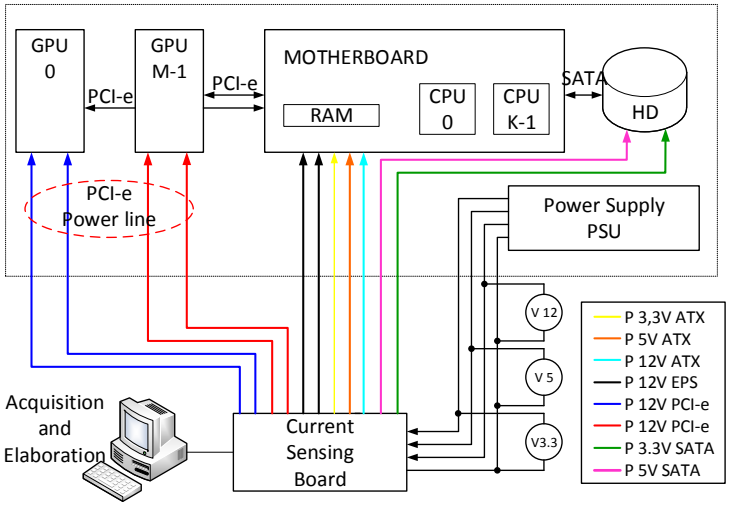


Figure 3.1: Measuring setup for the generic computing node.

Fig. 3.1 shows the acquisition system used in this work. It has been designed to measure the power consumption of each target system component:

- the motherboard (ATX - 12 V, 5 V and 3.3 V);
- the additional power supply for CPU and PCI Express (EPS - 12 V);
- the GPUs (PCI-e - 12 V);
- the hard disk (SATA - 5 V and 3.3 V).

The current-sensing board is composed of 16 Hall effect current sensors capable of measuring currents in the range 0-30 A (Allegro ACS713) and 0-50 A (Allegro ACS758). The sensor outputs provide voltage values proportional to the currents measured. Low-pass filters are connected between sensor outputs and microcontroller inputs (STM32F) in order to improve

the signal-to-noise ratio. The bandwidth of the sensors is 120KHz, which is higher than the cut-off frequency of the ADC input filter. Analog-to-digital converters internal to the microcontroller sample the current data. The board is connected to a PC used to acquire data via USB. A Java interface is set up to manage data acquisition. When the connection between the board and the PC is set, the microcontroller starts to send one data packet per second. Using all 16 channels, the maximum sample rate is 1600 samples/s for each channel. The current samples are then multiplied by the voltage values measured with a Fluke device and data are post-processed by Matlab in order to obtain the power profile of each job.

3.3 Power-aware scheduler

High performance computing relies on computing nodes equipped with multi-core devices at each node and a distributed resource management system (DRMS). Users submit jobs which have to be assigned to the cores. The DRMS sorts and assigns jobs to the available resources following a scheduling policy. Hence by changing the DRMS policy it is possible to limit peak power consumption[114].
The proposed power-aware scheduler is developed and implemented starting from two common scheduling policies (First-in-first-out and Backfilling first fit).

- First in First Out (FIFO)
  First-in-first-out is a simple scheduling strategy. New jobs that must be executed are placed at the end of the queue. When a resource becomes available the first job in the queue is activated.

- Backfill algorithm (BFF)
  Backfill is a policy which allows the scheduler to run jobs out of arrival order. When there are not enough resources to run the first job in the queue, other jobs in the queue are checked in order to find a job that could be executed without exceeding the additional constraint imposed by the algorithm. Usually backfill allows the scheduler to start lower-priority jobs so long as they do not delay the first job in the queue. Execution is therefore limited to the resources available

and the time available before the expected start time of the first job. Various backfill strategies can be used. In this work a backfill first fit strategy has been implemented: the list of feasible jobs is filtered, selecting the first one which fits the constraints.

The proposed power-aware scheduler is capable of predicting the behavior of each node, every time a new application is ready to be activated. Each job discussed in the following pages is composed of a low computational part running on a CPU host, and a data-intensive computational kernel running on a GPU.

3.3.1 The scheduling algorithm

In order to simplify the design of the scheduling algorithm, three hypotheses were formulated:

- during the entire execution of a GPU kernel, the power profile is assumed constant and equal to the maximum value;
- power consumption of GPU kernels is not sensitive to changes in input data set values;
- different input data size for the same GPU kernel lead to the same peak power with different durations.

The validation of these hypothesis will be discussed in Section 3.4.1. Parameters used in the scheduling algorithm are defined in Table 3.1. The

Table 3.1: Nomenclature table of the algorithm

P_C	Power Capping : predetermined power constraint
$P_{job\,(m)}$	Maximum power consumption of *mth*-job
$P_{node\,(n)}$	Power consumption of *nth*-node
$P_{idle\,(n)}$	Power consumed by the *nth*-node in idle state
N	number of nodes
M	number of cores for each node

purpose of the scheduler is to assign jobs to the available resources, limiting the maximum power consumption of each node to below the predetermined constraint (P_C) while keeping the parallelism as high as possible. P_C is set via software and can be adjusted by the user.

Power consumption of all kernels comprising the library is characterized a priori using the measuring system described in Section 3.2. Thus, the scheduler already knows the maximum power consumption of each job (P_{job}) that could be activated on the nodes. As previously mentioned, these contributions are assumed constants and equal to their maximum values. The detailed description of how the maximum power consumption of each job has been obtained is reported in Section 3.4.1.
Each GPU in the system is a stand-alone device running an independent job. Hence, the total power consumption of the *nth*-node during the concurrent execution of M jobs is computed by adding up the power consumed by the node in idle state (P_{idle}) and the power consumption of jobs (P_{job}) which are running on that node, as shown in (3.1).

$$P_{node\,(n)} = P_{idle\,(n)} + \sum_{m=0}^{M-1} P_{job\,(m)} \quad \forall n \in N \tag{3.1}$$

Once the *mth*-job has been executed, the power consumption of the *nth*-node is updated as shown in (3.2).

$$P_{node\,(n)} = P_{node\,(n)} - P_{job\,(m)} \tag{3.2}$$

When a new job needs to be activated on the system the DRMS checks if there are nodes with resources available and if these nodes will meet the power constraint when executing the job, as reported in (3.3).

$$P_C \leq P_{node\,(n)} + P_{job} \tag{3.3}$$

Hence, if the condition shown in (3.3) is verified for some nodes, the scheduler assigns the application according to a minimum power-slot policy as shown in the following algorithm.
Fig. 3.2 explains graphically what is discussed above. The system shown is composed of 4 nodes ($N = 4$), each one equipped with 4 GPUs ($M = 4$). At time T_{start} the scheduler has to select the node on which the new job is to be activated. The first node (*NODE 1*) is already running 4 jobs so it has no resources available. The second node (*NODE 2*) has one GPU available. However, if the job were to be run on the node, the P_C threshold would be exceeded. The job needs to be executed on *NODE 3* or *NODE 4* if the

Algorithm 3.1. Node selection

Input: $N; P_C; P_{node}; P_{job}$
Output: $selected\ node\ (n)$

1: $n \leftarrow none$
2: $P_{min} \leftarrow P_C$
3: **for** $i = 0$ to $N - 1$ **do**
4: $\quad P_{TMP} = P_C - P_{node\,(i)} - P_{job}$
5: $\quad$ **if** $(0 \leq P_{TMP} \leq P_{min})$ **then**
6: $\quad\quad P_{min} \leftarrow P_{TMP}$
7: $\quad\quad n \leftarrow i$
8: $\quad$ **end if**
9: **end for**
10: **return** n

power budget is not to be exceeded. The DRMS performs the scheduling according to a minimum power-slot policy as illustrated in the proposed algorithm. The strategy is to activate the job in the node with the smallest "power-slot" able to keep the power consumption under the predetermined threshold. This is done in order to keep the largest "power-slot" free for a more power-consuming job.

In this example the job is activated on *NODE 4* while *NODE 3* is left free for a more power-consuming job. If all nodes are busy or the constraints are not met, the job will wait to be scheduled in the queue in accordance with the scheduling policy selected. Thus the scheduler manages both power and GPU as finite resources.

The algorithm can easily be extended to rack level instead of node level. In this case, each time a new job is activated, the power needs to be controlled at rack enclosure level [88] instead of node level. Equation (3.3) can be rewritten as:

$$P_{rack} \leq \sum_{n=0}^{N-1} P_{node_{(n)}} + P_{job} \tag{3.4}$$

Once the characterization of jobs has been completed, the scheduler makes its decision at run time (on-line), selecting the most suitable candidates out of the current set of tasks ready-to-run. The algorithm is non-preemptive in that the currently executing task will not be preempted until completion.

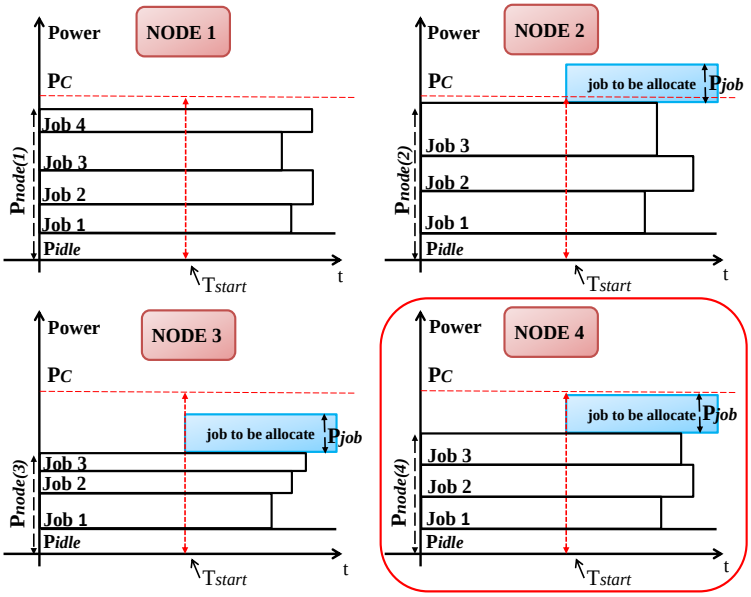


Figure 3.2: Example of scheduling: 4 nodes, each one composed of 4 cores

3.4 Performance and Evaluation

3.4.1 Job Characterization

Six jobs in the field of linear algebra (see Table 3.2) were developed, starting from the code samples available in [25, 115]. The power consumption of these jobs was characterized by changing the dimensions and the values of the input data.
The configuration of the computing node used for this work is detailed as follows:

- Motherboard SuperMicro X8DTG-QF;
- Two Intel Xeon E5520 CPUs @ 2.27 GHz;
- 24 GB RAM;
- Two NVIDIA GTX 590 graphics cards to a total of 4 GPUs.

Differently from the target system described in section 2.3, the GPU GTX 480 has been removed to avoid multiple algorithm characterizations on different GPUs. The contributions measured in order to obtain an estimation of the power consumption during the computation of a job are:

- the power consumption of the motherboard (ATX);

- the additional power supply for CPU and PCI-express (EPS);
- the power of the GPUs (PCI-e);
- the consumption of the hard disks (SATA).

The idle power consumption of the computing node (P_{idle}) is 240 W. Each job was characterized in the same operating conditions and during its execution no competing tasks were performed.
The heterogeneous programming model supported by GPU implies a system composed of a host (CPU) and a GPU each with their own separate memory. Kernels operate out of GPU memory, so the run-time provides functions to allocate, deallocate and copy GPU memory, as well as transfer data between host memory and GPU memory [34]. This architecture is reflected in the power profile of each job. A small increase in the job power consumption will be detected during data upload and download. However the most time- and power-consuming phase is the kernel execution.

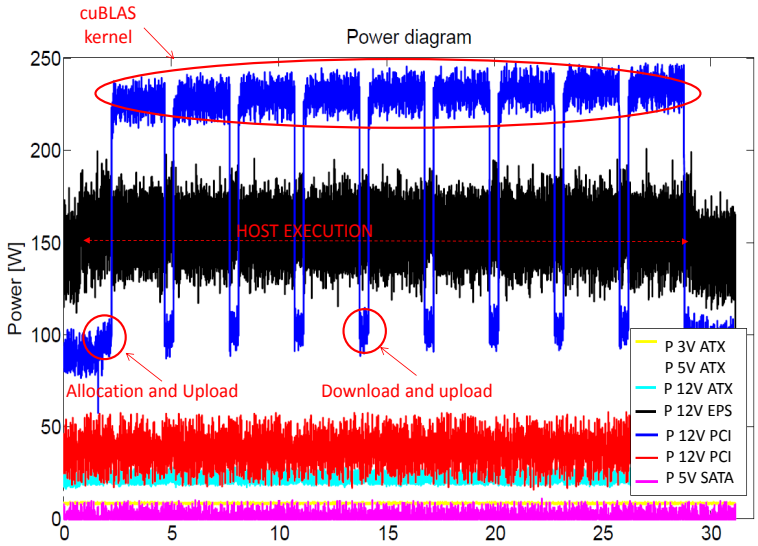


Figure 3.3: Power profile measured during a matrix multiplication on GPU

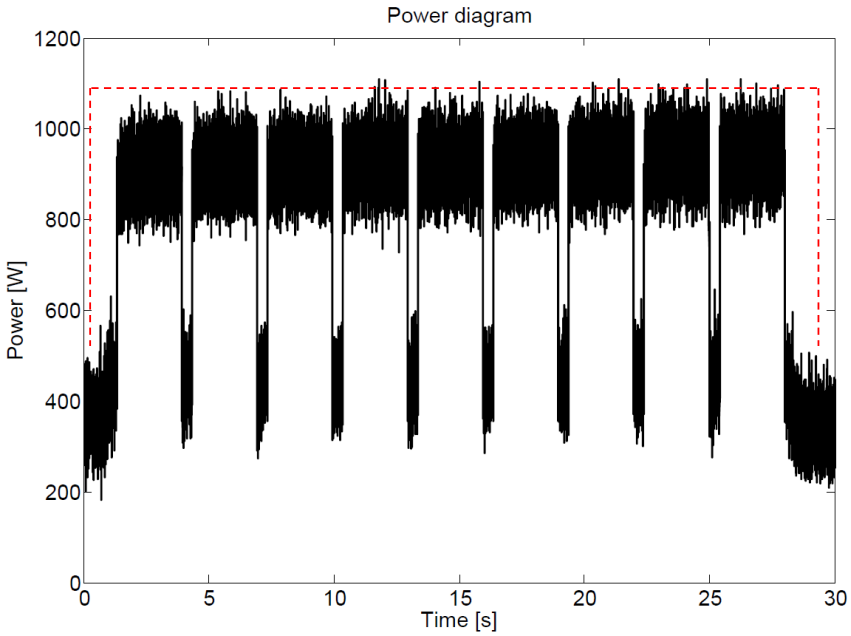
Fig. 3.3 shows an example of job power profile obtained with the monitoring system developed. The figure shows a characterization of a matrix multiplication performed by multiplying two input matrices comprising 30720x30720 elements. Since the GPU memory has limited space, the computation was carried out by decomposing the matrix into 9 different

sub-matrices of dimensions 10240x10240. Two main contributions can be observed: the black line which shows the consumption of the motherboard and the blue line which represents the power supply of a GPU. As shown in Fig. 3.3, a GPU job starts with allocation of the GPU memory and copying of data from host to GPU (contribution indicated in Fig. 3.3 with *allocation and upload*). Once the data have been allocated on the GPU memory, the kernel execution starts. This phase always coincides with the highest power in the job (contribution indicated in Fig. 3.3 with *cuBLAS kernel*). After computation of a kernel, results are copied from device to host memory and new input data are uploaded on the device memory (contribution indicated in Fig. 3.3 with *download and upload*). Once the job is finished, the device memory is released. The other contributions reported in Fig. 3.3 (e.g. SATA) are negligible. All the jobs studied in this work, performed with different sizes of input matrix and different input data set values, follow the trend discussed above and shown in Fig. 3.3. The characterization proves the hypothesis discussed in Section 3.3. The power profile of a kernel is not sensitive to changes in the input data set values. In addition, experiments demonstrate that different data sizes for the same kernel lead to power profiles with very similar power peaks which can be approximated to the same peak value. The different duration of these kernels depends on the computational complexity of the kernel selected. A summary of the GPU job power consumption is reported in Table 3.2. The table shows

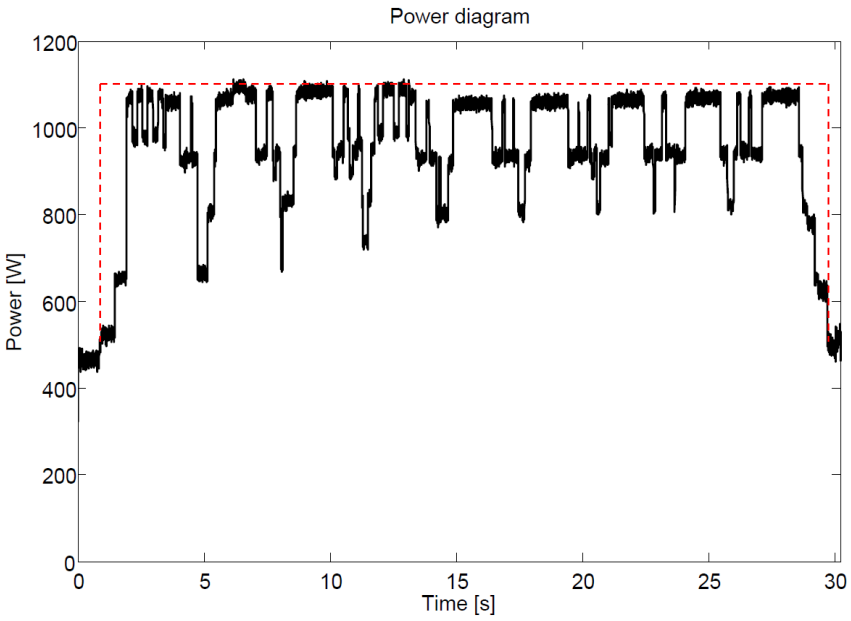
Table 3.2: Power consumption of GPU jobs

Job	P_{job}
Matrix Multiplication	160 W
Matrix Multiplication (cublas)	220 W
Eigenvalues	170 W
Triangular Matrix Inversion	190 W
Matrix Transpose	180 W
Scalar Product	110 W

that the jobs differ considerably in power consumption value. These values have been obtained by subtracting the idle power consumption of the GPU from its maximum power consumption during the execution of the job.
As previously described in the algorithm, characterization of jobs which can be activated on the target system allows the scheduler to predict what



(a) Theoretical



(b) Experimental

Figure 3.4: Comparison between theoretical and experimental evaluation of total power absorption during 4 concurrent matrix multiplications.

the power consumption will be, knowing which jobs are currently running on the node. An example of what has been discussed above is shown in

Fig. 3.4. The consumption profile is obtained by considering 4 concurrent executions of the previously characterized matrix multiplication. Fig. 3.4a shows the prediction obtained by adding the profile previously characterized, while Fig. 3.4b shows the consumption measured during run-time. Substituting the values in (3.1), the estimated peak power consumption is computed.

$$P_{node\,(n)} = P_{idle\,(n)} + \sum_{m=0}^{M-1} P_{job\,(m)}$$

$$= (240 + \sum_{m=0}^{3} 220)\ W$$

$$= 1120\ W$$

The dashed lines at the top of the two profiles show the approximation introduced in the algorithm (i.e. power constant and equal to the maximum power value). The two profiles show the same trend. The fluctuations are more evident in the theoretical case due to the fact that the plot represents the sum of four identical contributions, so that noise components are visibly amplified.

3.4.2 Experimental Setup

The scheduler was evaluated on 4 computing nodes ($N = 4$) equal to that described in Section 3.4.1. Hence the total number of GPUs used is 16. The algorithm was tested for generating, executing and measuring 10 workloads of 1000 job requests selected from the previously characterized jobs. In order to create the workload, a Markov chain model was used [116] [117]. Each job requires 1 GPU and it is assumed that there is no data dependence between any jobs. This hypothesis is used to simplify the experimental setup. However, no considerable benefits can be achieved from this assumption, since the scheduling policies used do not allow jobs to be executed out of the arrival order. Each measurement was ended after all jobs had finished.

The workloads were generated so as to have more concurrent jobs needing to be activated than resources available. This was done in order to verify

the performance of the algorithm during high-demand periods. Power profiles obtained using the proposed power-aware scheduling algorithm were compared with the results obtained by executing and measuring the same jobs without the power-aware characteristic, so as to evaluate the trade-off between performance and peak power reduction.
As shown in the previous Section, the worst case scenario of a node, (considering the previously characterized jobs) corresponds to four concurrent executions of matrix multiplication (*cublas*), bringing the total peak power up to 1120 W ($P_{peak\,\{WC\}}$) as shown in Fig. 3.4. Although this situation is very rare, the power supply has to be designed so as to sustain this condition.

3.4.3 Analysis of Results

Allocation of workloads to resources was evaluated for the two different policies (FIFO, BFF) while changing the constraint on the maximum power value attainable by the system (P_C). Since the scheduling is done according (3.3), changing P_C the maximum power consumed by the node and the execution time of the entire workload change as well.
Several indices are introduced to evaluate the performance of the proposed technique. A detailed description of these follows:
Peak reduction (PR)
The peak reduction is computed comparing the peak power values obtained executing the workload with and without the power aware characteristic as shown in (3.5):

$$PR = \frac{P_{peak\,\{ST\}} - P_{peak\,\{PA\}}}{P_{peak\,\{ST\}}} \cdot 100 \tag{3.5}$$

with

$$P_{peak} = \max(\{P_{node\,(n)} : n = 0, \ldots, N-1\})$$

where $P_{peak\,\{ST\}}$ is the peak power value measured during the execution of the algorithm without the power-aware characteristic while $P_{peak\,\{PA\}}$ is the peak power value measured during the execution of the power-aware version.
Peak reduction with respect to the worst case power scenario (PW)

Peak reduction with respect to the worst case power scenario is defined using (3.5) by substituting the power value measured during the execution of the algorithm without the power aware characteristic ($P_{peak\,\{ST\}}$) with the worst case power value ($P_{peak\,\{WC\}}$, in this case 1120W).

Increase in time (T)

The increase in computation time is obtained comparing the execution time of the workload with and without the power aware characteristic as shown in (3.6):

$$T = \frac{T_{max\,\{PA\}} - T_{max\,\{ST\}}}{T_{max\,\{ST\}}} \cdot 100 \tag{3.6}$$

with

$$T_{max} = \max(\{T_{W\,(n)} : n = 0, \ldots, N-1\})$$

where T_W is the workload execution time.

Increase in energy (EC)

The increase in the energy consumption is evaluated following (3.7):

$$EC = \frac{E_{\{PA\}} - E_{\{ST\}}}{E_{\{ST\}}} \cdot 100 \tag{3.7}$$

where the total energy consumption E is computed as shown in 3.8

$$E = \sum_{n=0}^{N-1} \left(\int_{0}^{T_{W\,(n)}} P_{node\,(n,t)} \cdot dt \right) \tag{3.8}$$

Peak power deviation from the average (MD)

The peak power deviation from the average is obtained as reported in (3.9):

$$MD = \frac{P_{peak} - P_{avg}}{P_{avg}} \cdot 100 \tag{3.9}$$

where the average power is computed following (3.10):

$$P_{avg} = \frac{E}{\sum_{n=0}^{N-1} T_{W\,(n)}} \tag{3.10}$$

Fig. 3.5 shows the experimental results measured by executing the workloads. As shown in Fig. 3.5a, by setting the power capping value at 800 W

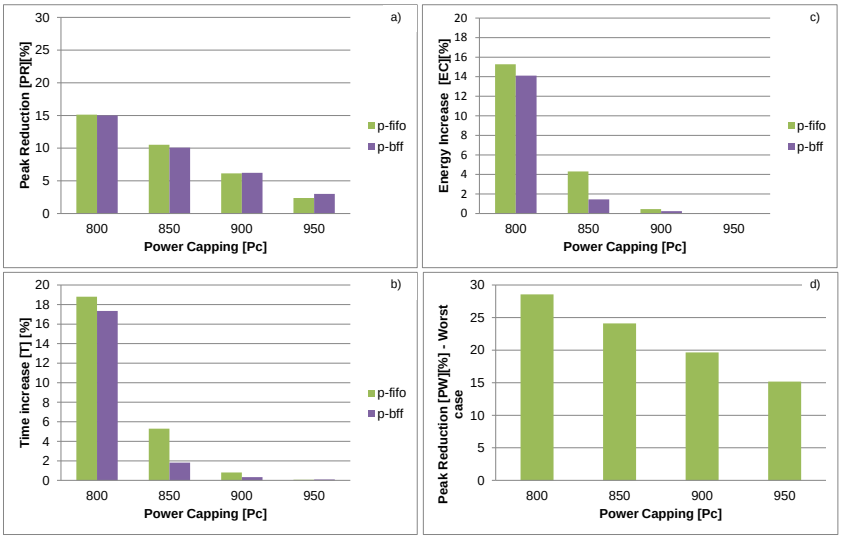


Figure 3.5: Average performance obtained using the power-aware scheduling algorithm. In a) one sees the average peak reduction obtained by the algorithm, while b) shows the increase in time. Fig. c) shows the increase in energy consumption and Fig. d) the power reduction with respect to the worst case scenario.

a peak power reduction of around 15% is measured. However this reduction is paid for by a time increase between 17 and 19% (Fig. 3.5b) and an increase in energy consumption between 14 and 15% (Fig. 3.5c), depending on the scheduling policy. This means that the threshold chosen (P_C) limits full exploitation of the computational parallelism available. By using a higher power budget (850 W), better results can be obtained. In this case the increase in time taken to compute the entire workload is less than 2% with the power-aware version of the BFF algorithm, with a measured peak power reduction of up to 10%. Using the power-aware FIFO approach, results are slightly worse because each time the power constraint is not met all jobs are delayed. On further increasing the P_C threshold, a peak power reduction between 6 and 7% is recorded without any impact on system performance. This means that the algorithm removes the sporadic peaks that take place during workload execution, thus avoiding power capacity overload.

Another advantage introduced by the algorithm is that it mitigates the worst case power scenario. As shown in Fig. 3.5d, by using the power-aware approach (with $P_C = 850W$) the worst case scenario is reduced by up to 24% with a negligible impact on performance.
Fig. 3.6 explains the benefit of the algorithm from another point of view. The plot shows the peak power deviation from the average (MD) as a function of the increase in computational time (T).

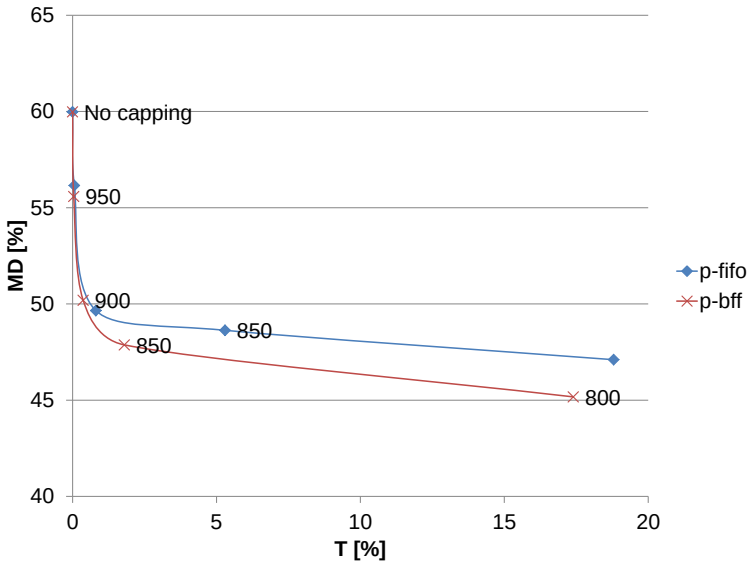


Figure 3.6: Power-Performance comparison.

When no capping is forced a peak power 60% higher than the average

Table 3.3: Power-Performance Comparison

	p-FIFO				p-BFF			
P_C	P_R [%]	T [%]	E [%]	MD [%]	P_R [%]	T [%]	E [%]	MD [%]
800	12.0-18.5	11.7-22.6	10.5-17.8	45.5-52.5	11.2-18.7	11.3-21.8	9.5-16.9	45.0-54.6
850	6.9-14.6	2.8-9.7	2.3-7.9	43.4-52.9	6.4-13.2	0-4.4	0-3.6	42.2-49.5
900	2.0-9.1	0-2.6	0-2.0	45.4-56.6	3.5-9.0	0-2.9	0-2.4	44.9-56.0
950	0-6.2	0-2.8	0-2.3	49.9-67.5	0-8.9	0-2.5	0-2.3	49.9-67.5

value is measured. The first part of the curve (i.e. when power capping is

set between 950 and 900 W) shows how the peak deviation from the mean can be reduced by 10% without any significant increase in the execution time of the workload, by scheduling jobs taking their power consumption into account. In the last part of the curve the value set in the algorithm is closest to the average power value of the workload, so that peak reduction is obtained at the cost of a significant time increase.

As expected, the BFF scheduling policy allows one to achieve better results than the FIFO policy because it introduces fewer constraints on queue management.

A summary of the measurements recorded is shown in Table 3.3. The increase in energy is due to the fact that jobs are delayed when the power constraint is not met. The system is therefore in the idle state for a longer time frame.

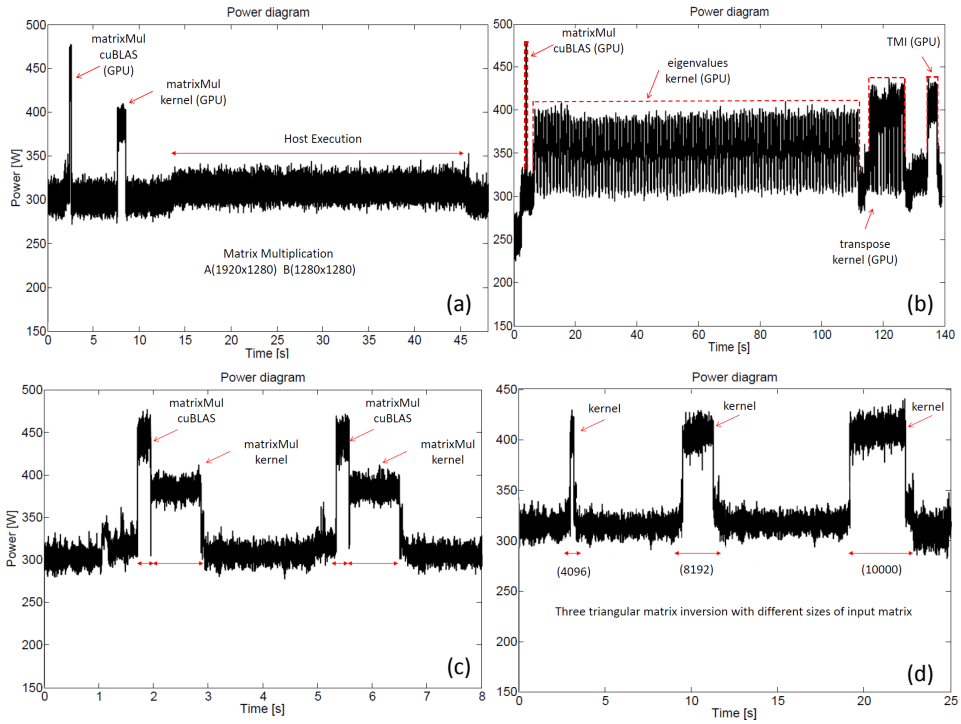


Figure 3.7: a) Comparison between GPU and CPU execution of matrix multiplication, b) Four different GPU kernels, c) Triangular matrix inversion with different sizes of matrix, d) Analysis of power requirements of different jobs.

3.5 Discussion

Although thorough modeling of GPU kernel power dissipation does not lie within the scope of this chapter, several important considerations can be drawn from the measurement of power profiles taken by the measuring system:

- Execution of a GPU job (for the architecture studied) is much more peak power-consuming than execution of the same job on a CPU (although the execution time is significantly reduced);
- during execution of a kernel the power profile can be assumed constant;
- power consumption of jobs is not sensitive to changes in the input data set values;
- different input data sizes for the same kernel lead to the same trend with different durations. The run-time depends on the computational complexity of the algorithm used in the kernel.

Fig. 3.7 shows what has been discussed above. The first plot (Fig. 3.7a) was obtained when computing the same matrix multiplication three times, using three different algorithms. The first two are mapped on GPU (*matrixMul cublas* and *matrixMul kernel*), while the third is obtained by computing the same operation on CPU (*host execution*). Since the computation on GPU is much more power consuming than on CPU, in order to reduce peak power one should focus on concurrent executions of GPU kernels rather than on CPU tasks. Since the difference between the idle power state and the maximum power state of a CPU is small (compared to that of a GPU), the CPU power consumption can be assumed to be equal to its maximum power consumption each time a new job is activated on CPU. Fig. 3.7b shows the execution of different jobs performed on GPU. A GPU job can be composed of a single kernel or multiple call to the same kernel performed with different sets of input data to overcome the limited space of the GPU memory. Fig. 3.7b makes it clear that the power consumption of a GPU job can be considered as a constant contribution (dashed line) that has to be added to the total power consumption. Fig. 3.7c and Fig. 3.7d provide

some additional considerations as to the values and sizes of input data. Fig. 3.7c shows four matrix multiplications performed with different input values. The graph shows that the power profile of a job depends only on the kernel computed and is independent of the input values. The last plot (Fig. 3.7d) shows that the same kernel, performed with different input data sizes, leads to different durations of the kernel, but with comparable peak power values. These considerations helped to streamline development of the algorithm (which is in fact based on these hypotheses) so as to make it general.

3.5.1 Application case

As previously pointed out, the purpose of the algorithm is not to save energy, which increases, albeit slightly. The approach aims to reduce the supply cost due to high peak power whilst having negligible impact on the parallelism of computational nodes.
From another point of view the developed model allows designers to increase the number of cores without increasing the capacity of the power supply unit. For example, each node used in this work is equipped with a power supply unit of 1400 W. As shown in Section 3.4 the worst case power scenario is around 1120 W. In this scenario, since each GPU GTX 590 used in this work consumes up to 365W [118], to equip the system with another GPU could lead to system failure caused by power capacity overload. Using the proposed approach, it would be possible to add one GPU GTX 590 to the system, without overloading the power capacity, thereby reducing supply costs, cooling systems and power distribution units.

3.5.2 Limitations of the approach

Experimental results shown in this chapter depend on the target architecture utilized in this work. The execution of these jobs on a different architecture could lead to a different peak power values. This is because the technique is based on an a priori characterization that is architecture dependent. If the target system changes, the characterization has to be repeated on the new target system. In addition a new characterization is needed each time a new kernel is added to the library. These considerations high-

light how the power-aware scheduling algorithm is related to the low-cost monitoring system proposed.

3.6 Conclusions

This chapter presented a new algorithm for parallel realtime scheduling, executed on GPU cluster nodes. The idea proposed is to manage both power consumption and GPUs as finite resources has been proposed in order to fully exploit parallelism which in heterogeneous CPU-GPU systems is limited by the power consumption. Since the power configuration may vary widely, there is the likelihood that job overlapping will result in power spikes high enough to exceed the specifications of the nodes, causing catastrophic failures in systems designed to a better-than-worst-case policy. In addition, peaks synchronized across several nodes could cause localized power outage. In addition the algorithm allows designers to increase the number of cores without increasing the capacity of the power supply unit. Compared to a system without any power-aware policy, the model allows one to obtain a peak power reduction of as much as 10%. Executing workloads that usually involve high power peaks can be avoided at the cost of a very slight time increase, making it possible to reduce the power supply cost.

Chapter 4

Heterogeneous System using a Reconfigurable approach for efficient graph exploration

This chapter present a graph exploration based on efficient use of memory which tries to overcame speedups achieved by the previous implementations. This exploration has been carried out during my 6-month visiting period at the Department of Computing - Imperial College London, under the supervisor of Prof. Wayne Luk. The implementation of this work is the subject of an ongoing research project.

4.1 Motivation and background

Many important areas of science such as astrophysics, artificial intelligence, genomics and national security require approaches to explore large scale graphs involving millions of vertices and billion of edges. Searching algorithm are used to explore vertices, and paths with specific properties. Among graph search algorithms, breadth-first search (BFS) is a simple one, often used as archetype for other important algorithms such as Prim's minimum-spanning-tree, Dijkstra's single-source shortest paths, best-first search, uniform-cost search, greedy search, etc [8].
BFS is widely used in protein-protein interaction problems [119], intelligence analysis [120], robotics [121, 122] and network analysis [123]. Moreover, the relationship between vertices in the analysis of scientific graph is

expressed by the properties of the shortest path, given by the BFS search [124].
A wide literature explores different BFS solutions, based on multicore processors [125, 126, 124] or heterogeneous architectures [127, 128, 9, 129]. In [124] a new sophisticated data structure to reduce cache coherence traffic between CPUs is presented. This implementation outperforms the previous ones, including other architectures such as cell processors [128], clusters [125] and shared memory supercomputers [130]. A simple and faster implementation of BFS on multicore CPUs has been proposed in [9]. The paper also proposes a new hybrid method based on CPU and GPU which selects the best execution methods among sequential and parallel approach, depending on the graph scale.
Unfortunately traditional software and hardware parallel implementations do not necessary work well for large scale graphs due to the graph properties [131]. For instance, many graphs are unstructured and highly irregular and they require fine-grained memory accesses to be explored. These characteristics lead to suboptimal performance in cache-based microprocessors due to poor spatial and temporal locality of memory accesses. Moreover, since no computation must be performed in BFS algorithm, the execution is dominated by the memory latency. Also on reconfigurable architectures, graphs with unstructured and irregular memory accesses cannot achieve high performance. The low memory bandwidth generates many pipeline stalls, resulting in a little FPGA-acceleration.
In this chapter a novel idea for an efficient graph implementation on reconfigurable architectures is presented.

4.2 Breadth-first search

Given a graph G(V,E) composed of a set of vertices V, a set of edges E and a source s in V, the BFS algorithm explores the edges of G to discover all the vertices reachable from s. It computes the distance from s to each reachable vertex in terms of smallest number of edges and it produces a breadth-first tree rooted at s. Vertices are visited in levels: when a vertex is visited at level l, it also said to be at distance l from the root.
For any vertex v reachable from s, the simplest path in the tree from s to v is

the path containing the smallest number of edges. The name breadth-first search comes from the fact that the algorithm expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. This means that the algorithm discovers all vertices at distance k from s before discovering vertices at distance $k+1$.

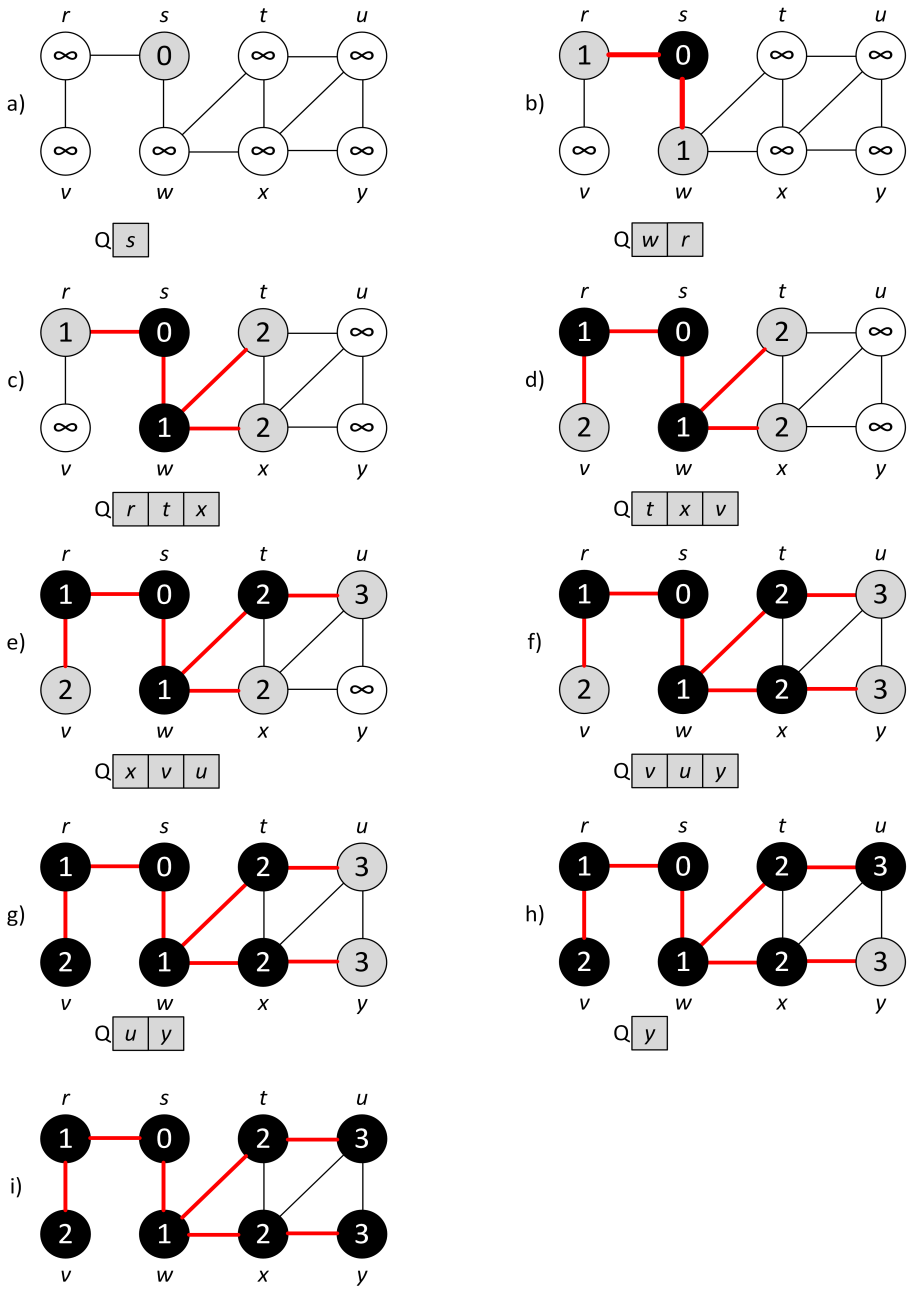


Figure 4.1: The operation of BFS on an undirected graph [8].

Figure 4.1 illustrates how BFS works on a sample graph. To keep track of progress, vertices in figure are white, gray and black. All vertices are initially white, except for the source vertex which is gray. When a vertex is encountered during the search, it becomes non-white. Therefore, gray and black vertices have already been discovered. However BFS distinguishes between them to ensure that the search proceeds in a breadth-first manner. Suppose for example that vertices w and t are connected by an edge (w, $t \in$ E) and vertex w is black, then vertex t can be gray or black (see also figure 4.1c and e). Grey vertices may have some adjacent white vertices which represent the frontier between discovered and undiscovered vertices (for example see vertices t and u in figure 4.1c). The algorithm also builds the breadth-first tree, which initially contains just the source vertex s. Every time a white vertex v is discovered, the vertex v and the correspondent edge are added to the tree. Since a vertex is discovered at most once, it has at most one parent. The sequential breadth-first search algorithm follows:
The graph representation usually adopted is the "Compressed Sparse Row" (CSR) format. It consists of three vectors:

- V O(N)-sized - Input/Output;
- Offset O(N)-sized - Input;
- Adj list O(M)-sized - Input.

The BFS level of each node is computed starting from a source vertex and stored in an O(N)-sized array (V). The graph representation merges the successors of all vertices into a single O(M)-sized array (Adj list) with the beginning location of each vertex's adjacency list stored in a separate O(N)-sized array (Offset).
Figure 4.2 shows the CSR representation relative to the previous example (see also figure 4.1 a). The source vertex is s (i.e. $v_s = 0$). The offset values in position s (i.e. 4) and s - 1 (i.e. r = 2), indicate that successors of vertex s are stored in the adjacency list in position 2 and 3 (for a generic vertex v, the successor are stored from $offset[v-1]$ to $offset[v]-1$). Then, the vertices vector must be updated according to the vertices stored in the adjacency list (red arrows in figure 4.2, $V_r = 1$ and $V_w = 1$).

Algorithm 4.1. Sequential BFS exploration of a graph

Input: G(V,E), graph;
source vertex, s;
level, exploration level;
Q, vertices to be explored in the current level;
Q_{next}, vertices to be explored in the next level;
E_v, set of edges connected to v
marked, array of booleans:
$marked_i \forall i \in [1 \cdots \mid V \mid]$

1: $\forall i \in [1... \mid V \mid] : marked_i = false$
2: $marked_s = true$
3: $level \leftarrow 0$
4: $Q \leftarrow \{s\}$
5: **repeat**
6: $Q_{next} \leftarrow \{\}$
7: **for all** $v \in Q$ **do**
8: **for all** $n \in E_v$ **do**
9: **if** $marked_n = false$ **then**
10: $marked_n \leftarrow true$
11: $Q_{next} \leftarrow Q_{next} \cup \{n\}$
12: **end if**
13: **end for**
14: **end for**
15: $Q \leftarrow Q_{next}$
16: $level \leftarrow level + 1$
17: **until** $Q = \{\}$

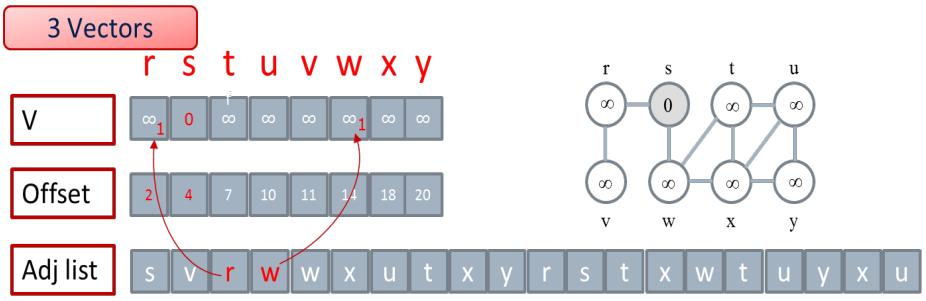


Figure 4.2: Compressed sparse row format

4.3 Irregular Graph and Parallel BFS Algorithm

The BFS algorithm guarantees that all vertices of the graph are traversed in a breadth-first manner. This means that all vertices in a certain level are visited before the vertices with a greater level.

In shared-memory systems the most common approach is *level synchronous BFS*. Vertices are divided in 3 sets; the visited set V, current-level set C and the next-level set N. All the vertices in the current set are visited and the next level vertices are identified. Following this approach synchronization is needed at each level and the parallelism is limited by the number of vertices in a given level as shown in algorithm 4.2.

Algorithm 4.2. Level Synchronous Parallel BFS

Input: G(V,E) Graph;
source vertex, *s*;

1: $Visited = Current = 0; Next = \{s\}$
2: $level \leftarrow 0$
3: **repeat**
4: $Current = Next$
5: $Next = 0$
6: **for** $Vertex\ v \in Current$ **do**
7: **for** $Vertex\ n \in E_v$ **do**
8: **if** $n \notin Visited$ **then**
9: $Next = Next \cup \{n\}; Visited = Visited \cup \{n\}$
10: **end if**
11: **end for**
12: **end for**
13: $level \leftarrow level + 1$
14: **until** $N = 0$

Recently new algorithms which optimize the simple parallel BFS implementation described in algorithm 4.2 have been proposed [124, 9, 129].
In [129] a reconfigurable computing solution which explores multiple vertices in parallel has been presented. The strategy used to take advantage of the hardware resources is to substitute the pipelining techniques usually implemented on FPGA with multiple application-specific graph-processing elements which tolerate the off-chip memory latency. This implementation outperforms the state of the art of 2 times for graphs with million of vertices and edges, using 4 FPGAs.
Other algorithms running on CPU use several optimizations to speed up the breadth-first exploration. For example in [124] (*Queue-based method* see also algorithm 4.3), the authors use a bitmap to compactly represent the visited set of vertices in concert with "test and set" atomic instruction when they update the bitmap. They also use local queues (one for each processor) before to update the global queue. Also this operation is efficiently imple-

mented using "fetch and add" atomic instruction. In addition they use a delicate queue implementation which minimize unnecessary traffic during queue operations. Although the queue optimization provides remarkable benefits for small input dataset, the performance of this approach decreases when the size of the input data set becomes very large.
However, real-world graphs are characterized by a property called "small-world phenomenon" which states that the number of vertices in each BFS level grows very rapidly. This property is not an observation valid for certain graphs but a fundamental characteristic of randomly-shaped real-world graphs [132]. Table 4.1 reports the number of vertices in each level obtained for a graph with 32 million vertices and 256 million edges generated using RMAT [133].

Table 4.1: Number of vertices in each BFS level: result from typical execution in an RMAT graph [9]

Level	Num. Vertices	$Fraction\ of\ Vertices(\%)^+$
0	1	$3.1 * 10^{-6}$
1	4	$1.3 * 10^{-5}$
2	749	$2.0 * 10^{-4}$
3	109239	0.34
4	7103690	22.20
5	9088298	28.40
6	130298	0.41
7	172	$5.3 * 10^{-4}$
total visited vertices	16432919	51.35
total visited edges	255962977	99.99

This large graph has a maximum level equal to 7. The execution time is dominated by the traversal of levels 4 and 5, since most of the vertices belong to these two levels. As stated in [9], many previous implementation of *level synchronous* BFS [128, 124, 129] do not take into account the small-world phenomenon. In this case an approach based on an efficient memory use provides better results. The approach [9] is implemented on GPU, and the use of shared queue is avoided due to the limitations of GPU memory architecture (see also section 1.3.3). This method, also called *Read-based* method, replaces local queues with one global queue that tells if a vertex belongs to the visited level, current level or next level. This array is accessed multiple times for each level. In this method, the data access pattern

Algorithm 4.3. Queue-based Parallel BFS algorithm

Input: G(V,E) Graph;
source vertex, s;
$Bitmap[v]$: bit set to 1 if vertex v is visited , otherwise 0
CQ : queue of vertices to be explored in the current level
NQ : queue of vertices to be explored in the next level
$LockedDequeue(Q)$: Returns the front element of the queue Q and updates the front pointer automatically;
$LockedReadSet(a, val)$: Returns the current value of a and sets it to val atomically;
$LockedEnque(Q, val)$: Insert val to the end of the queue and updates the pointer atomically;

Output: Array $P[1...n]$ with $P[v]$ holding the parent of v

```
1:  for all v ∈ V in parallel do
2:     P[v] ← ∞;
3:  end for
4:  for i ← 1...n in parallel do
5:     Bitmap[i] ← 0;
6:  end for
7:  P[r] ← 0;
8:  CQ ← Enqueue r;
9:  fork;
10: while CQ ≠ ∅ do
11:    NQ ← ∅;
12:    while CQ ≠ ∅ in parallel do
13:       u ← LockedDequeue(CQ);
14:       for each v adjacent to u do
15:          a ← Bitmap[i];
16:          if a = 0 then
17:             prev ← LockedReadSet (Bitmap[v], 1);
18:             if prev = 0 then
19:                P[v] ← u;
20:                LockedEnqueue(NQ, v);
21:             end if
22:          end if
23:       end for
24:    end while
25:    Synchronize
26:    Swap(CQ, NQ)
27: end while
28: join
```

allows a sequential read of the adjacency list, minimizing the access memory latency. In addition, in order to speedup the overall computation, a

combination of two methods is used. If the current level contains only few vertices the *Queue-based* algorithm presented in [124] is used, otherwise the *Read-based* method is used.
However, both these methods [124, 9] still require additional random accesses from the adjacency list to the destination vertices (red arrows of figure 4.2).

4.4 Parallel BFS implementation

Having a look to table 4.1, it is clear that there are 2 levels (level 4 and level 5) where up to the 30% of the vertices must be explored. This means that in these 2 levels a large part of the adjacency list and the vertices vector V must be read. In these levels multiple random reading of the same memory page executed at different time steps, could lead to a performance drop.
As an example, suppose we have to find the successor of vertices r and w in the previous example (see also figure 4.1b, c, d). For sake of simplicity we also suppose that the vertices vector V is memorized in 2 different bursts (or memory pages).

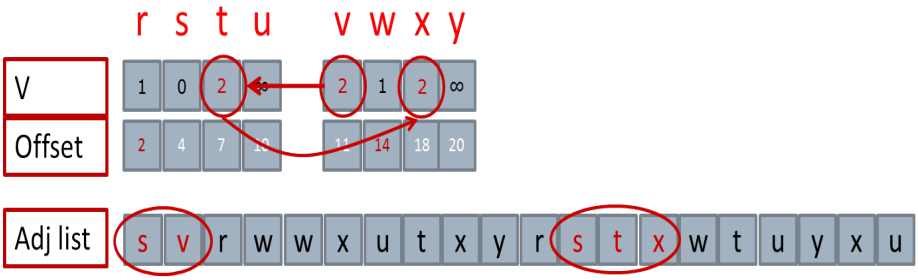


Figure 4.3: Data access pattern

As depicted in figure 4.1 the successor of vertex r is vertex v while the successors of vertex w are t and x. Hence we initially have to read the second burst in order to update the level of vertex v. Then the algorithm updates vertex t saved in the first burst and finally vertex x stored in the second burst.
Since vertices contained in the adjacency list are stored in sparse order, there is an high likelihood to modify elements in the same burst several times, at different time steps. This means that the same burst (or memory

page) must be loaded from the on-board memory to the on-chip memory multiple times. Since graph algorithms are memory-bounded a speed up can be obtained reducing this number of transfers. In this case instead of multiple random accesses of the vertices vector V, stored on the on-board memory, it is better to perform random accesses on the on-chip memory. Hence, the idea herein proposed to speed up the computation is to explore levels with an high number of vertices (such as levels 4 and 5 of table 4.1) using a different approach.
In order to do this the algorithm is decoupled in different stages. The CRS representation on reconfigurable architectures, must be changed in order to avoid as much as possible additional logic, necessary to manage sparse and shapeless graphs. The first step is to create a new O(M)-sized vector called Index on the host memory and copy this vector on the FPGA on-board memory. This vector is created at the beginning of each iteration level (level 1 in figure 4.4) starting from the vertices vector V and the offset vector as shown in figure 4.4. The vertices vector V is read and when the value is equal to the iteration level, the index vector must be updated according to the position indicated in the Offset vector. The Index vector will contain 1 if an element of the adjacency list must be read and 0 otherwise.

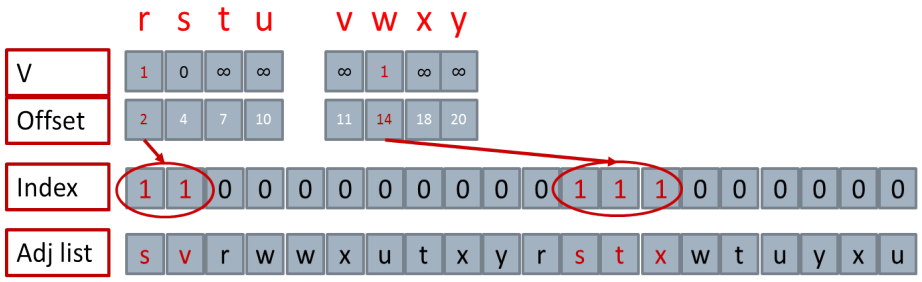


Figure 4.4: Additional Index Vector.

Although this vector represents an additional steps to the graph exploration, it is generated exploiting temporal and spatial locality of the caches, since the list is read sequentially. The Index vector can be thought as an enable. Its purpose is to increase the parallelism and avoid additional logic on FPGA since each vertex does not have a fixed number of successors.
As an example we consider the figure 4.5. In order to exploit the paral-

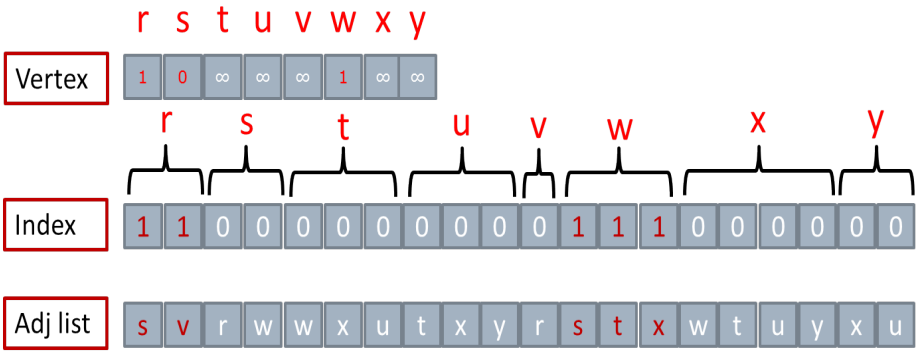


Figure 4.5: Relationship between Index and Vertex vectors.

lelism provided by FPGA, each vertex contained in the first burst should be read in parallel, and the successors stored in the adjacency list should be modified in parallel as well. However, if each vertex has a different number of successors, the speed up is limited by vertex with the higher number of successors (vertex x in the previous example, which has 4 successors). In a large scale graph this number can be huge. In the example shown in table 4.1 the number of successors can vary from 0 up to 300000. Therefore, once the Index vector is created, the graph can be processed in parallel at the adjacency list level. Then, a bitmap on the on-chip memory is created in order to store the vertices which must be updated at this iteration level. The size of this bit-vector is equal to the number of vertices of the graph (O(N)-sized). To update the bitmap a random memory access is still needed. However, in this case the random access is performed on the on-chip memory. Since on-chip memory is composed of several banks edges s and r can be written at the same cycle exploiting the parallelism provided by FPGA as shown in figures 4.7 and 4.6.

Once the bitmap has been completely updated the vertices vector can be updated in sequential order avoiding multiple overheads due to random memory access. The proposed implementation is shown in algorithm 4.4.

Obviously this methods can be used only when a large number of vertices must be explored. Hence, to address the inefficient processing of non-critical levels, a combination of two approaches can be adopted as proposed in [9]. The idea is to use the *Queue-based method* proposed in [124] when the current level contains only few vertices, and the proposed implementation when many vertices must be explored. Since all graphs start

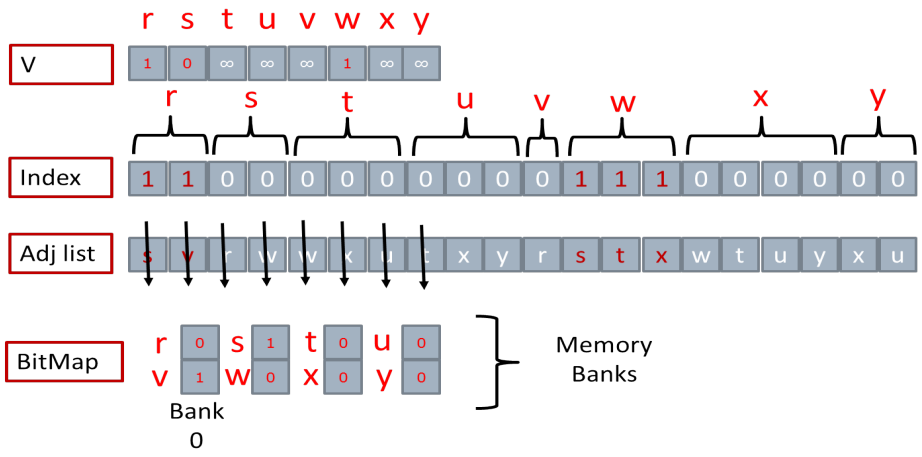


Figure 4.6: Bitmap update: first step in parallel.

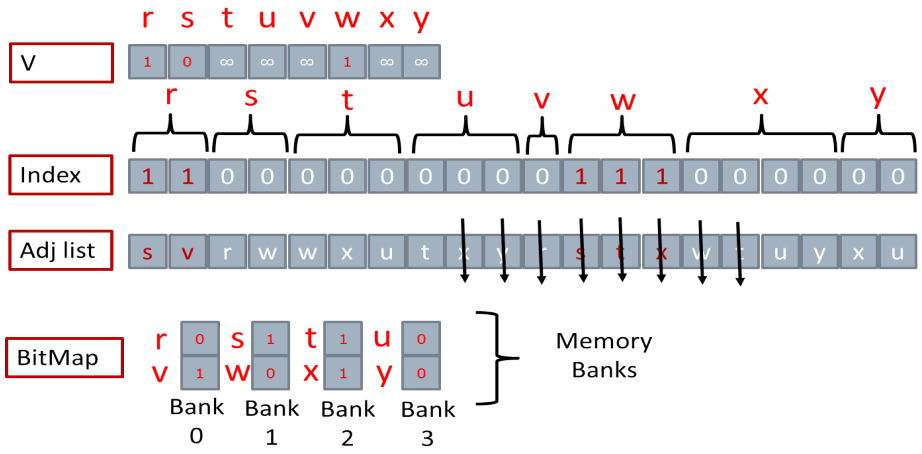


Figure 4.7: Bitmap update: second step in parallel.

from a single source vertex, the hybrid approach will start using the *Queue-based method*. Then when the size of the next level is larger than a threshold T, the parallel implementation herein proposed is used. Later, when the number of vertices decreases under the threshold, the algorithm switches once again to the previous *Queue-based* implementation.

4.5 Discussion

As explained in the previous sections due to irregular and shapeless nature of graphs, BFS algorithm requires a lot of random memory accesses in or-

Algorithm 4.4. Level-synchronous Parallel BFS

Input: $G(V,E); bfs\_level$
Output: $distance\,[1 \cdots n]\ \ with\ distance\,[i] = minimum\ distance(v_s, v_i)$

```
1:  repeat
2:    for all v_i ∈ V do
3:      if distance[i] = bfs_level then
4:        for offset[i] ≤ j < offset[i]) do
5:          (index[i] ← 1)
6:        end for
7:      else
8:        for (offset[i] ≤ j < offset[i]) do
9:          (index[i] ← 0)
10:       end for
11:     end if
12:   end for
13:   for all v_i ∈ V do
14:     (bitmap[v] ← 0)
15:   end for
16:   for all e_j ∈ E do
17:     if index[j] = 1 then
18:       (bitmap[adj[j]] ← 1)
19:     end if
20:   end for
21:   for all v_i ∈ V do
22:     if index[v] = 1&distance[v] = ∞ then
23:       distance[j] ← bfs_level + 1)
24:       done ← false
25:     end if
26:   end for
27:   bfs_level ← bfs_level + 1
28: until done
```

der to discover the minimum distance between the source and each vertex. Since the computation in BFS is absent, the only way to speed up the algorithm is to find an efficient solution capable of reducing the number of on-board memory accesses.

The proposed implementation allows designer to exploit the principle of locality in a cache based system, or to avoid multiple overheads due to random memory accesses in the system described in section 1.5.

The method proposed in this chapter replaces random memory accesses needed to updated the vertices vector (V), with an higher number of sequential on-board memory accesses. In order to roughly compare the per-

formance of the proposed method described in the algorithm 4.4, with the *Queue-based method* described in section 4.3, we consider the simple CPU execution time of level 5 of table 4.1. This is the most time consuming level, since 9000000 millions of vertices must be visited. (one third of the entire vertices vector).
Generally the execution time can be estimated considering the number of CPU clock cycles and the number of memory stall cycles as stated in equation 4.1

$$CPU_{Execution\,time} = (CPU_{Clock\,cycles} + Memory_{Stall\,Cycles}) \times Clock\,Cycle \tag{4.1}$$

where CPU clock cycles can be written as:

$$CPU_{clock\ cycles} = CPI \times IC \tag{4.2}$$

CPI means clock cycles per instructions and IC is the total instruction count. Also Memory stall cycles are evaluated following the equation 4.3

$$\begin{aligned} Memory_{stall\ cycles} &= Number\ of\ misses \times Miss\ Penalty \\ &= IC \times \frac{Misses}{Instruction} \times Miss\ Penalty \\ &= IC \times \frac{Memory\,accesses}{Instruction} \times Miss\ rate \times Miss\ Penalty \end{aligned} \tag{4.3}$$

Parameters used to compare the two algorithms are reported in table 4.2.
Since the architecture is the same for both the executions the CPI and the $Miss\ Penalty$ are equal for both cases. The ratio between $Memory\ Accesses$ and $Instruction$ is also equal for both cases. With the new implementation a double number of memory accesses have to be performed, since the data are initially stored in the bitmap and then in the central memory. However, in the proposed implementation the cache is massively used and therefore a smaller $Miss\ Rate$ is estimated (20% vs. 70%).
Combining equation 4.2 and equation 4.3, equation 4.1 can be rewritten as:

$$CPU_{execution\,time} = IC \times (CPI + \tfrac{Memory\ Accesses}{Instruction} \times Miss\ rate \times Miss\ Penalty) \tag{4.4}$$

Table 4.2: Parameters table

Parameter	Description	Queue based	New algorithm
CPI	number of clock cycles per instruction	1	1
IC	instruction count	9000000	$2 \times IC_{QB}$ = 18000000
$\frac{Memory\ Accesses}{Instruction}$	Number of memory access for each instruction	0.5	0.5
$Miss\ Rate$	Number of miss for each memory access	0.7	0.2
$Miss\ Penalty$	Number of additional clock cycles for each miss	30	30

Substituting the value for the Queue based method we obtain:

$$CPU_{QB} = IC_{QB} \times (1 + \times 0.5 \times 0.5 \times 30)$$

$$= IC \times (1 + 10.5) = IC \times 11.5$$

while considering the new algorithm we have:

$$CPU_{New} = IC_{new} \times (1 + \times 0.5 \times 0.2 \times 30)$$

$$= IC \times (1 + 3) = IC \times 4$$

Since $IC_{new} = 2 \times IC_{QB}$ the total speed up is:

$$SP = \frac{CPU_{QB}}{CPU_{New}} = \frac{IC_{QB} \times 11.5}{2 \times IC_{QB} \times 4} = 1.44$$

This is a simple estimation of a CPU based algorithm. However the same concepts can be applied to an FPGA implementation. In fact, the FPGA computing system developed by Maxeler presents an overhead of 40 clock cycles every time a burst is read randomly from the on-board memory and an overhead of 2 clock cycles for each other burst, which is read in sequential order.

The reconfigurable implementation of the algorithm is the subject of a current study.

4.6 Conclusions

In this chapter a scalable breadth-first search (BFS) algorithm has been presented. In spite of the highly irregular access pattern of the BFS, the proposed algorithm was able to enforce various degrees of memory, minimizing the negative effects of completely random memory accesses. Although this implementation has been thought for reconfigurable architectures, the algorithm can be efficiently implemented also on CPU. Theoretically this implementation allows programmer to achieve a speedup as compared to the previous implementations. Achieving a speedup in graph explorations is quite important since graphs are a core part of most analytic workloads. The implementation is currently under study.

Chapter 5

General Discussion

This thesis deals with the capabilities and limitations of some heterogeneous architecture in the field of high performance computing. In particular, this work explores the performance of standard workstation equipped with commodity hardware components to address some computational hungry problems. Heterogeneous computing systems are affordable, highly configurable and easy to upgrade. Given these characteristics, excellent results can be obtained from paring parallel software with parallel hardware because it takes parallel applications to access the potential of parallel hardware.

5.1 Summary of the contributions and results

The main contributions of this thesis can be summarized in three principal parts, which address selected computational challenges. Based on the individual demands of the task, different scales of heterogeneous systems have been employed. The set-ups range from a workstation supercomputers with multiple CPUs and several high-end graphics cards to a workstation equipped with multiple CPUs and FPGA.
Chapter 3 presented a multi-GPU based model for design and analysis of solar field, motivated by the need to have an accurate and fast simulation environment for studying mirror imperfection and non-planar geometries. A single-precision throughput of 4 TFLOP/s is achieved using two NVIDIA GTX 590 dual-GPU graphics cards and one NVIDIA GTX480, which is a high percentage of the total theoretical peak. It outperformed a

multiCPU reference by a factor of 52X.
Chapter 4 presented a scheduling algorithm for an efficient distribution of the workload to the resources. The scheduling algorithm yielded peak power reduction of as much as 10% compared with a system without any scheduling policy. The algorithm has been tested using 2 NVIDIA GTX 590 dual-GPU graphics cards.
Finally, Chapter 5 presented a graph exploration on reconfigurable architectures based on efficient use of memory. This work has been carried out during my 6-month visiting period at the Department of Computing - Imperial College London, under the supervisor of Prof. Wayne Luk.
This implementation tries to overcame the speedup achieved by the previous parallel implementations. Although the study is not finished yet, a preliminary estimation performed on CPU shows that the proposed implementation could overcome the state of the art by a factor of 1.5.

5.2 Performance and limitations of heterogeneous architectures

Looking at the results available in literature, it is easy to see that heterogeneous architectures offer remarkable performance in many scientific applications. Some implementations report amazing speedups up to 500x as a look in the "CUDA community showcase" [134] or in the "MAX-Up Publications Website" [135]. However, the theoretical throughput between GPU and CPU of comparable generations is an order of magnitude. Also the efficiency advantage of FPGA ranges from 10 to 50. The question is: how these applications achieve these high speedups over CPUs? In many cases, benchmarks hide several pitfalls. For instance fine-tuned multi-GPU implementations are compared with serial CPU executions, sometime unoptimized. This has become so common that some studies present quantitative and qualitative analyses which have the purpose to debunk these unexpected speedups [136, 137]. In addition, heterogeneous implementations are subjected to several limiting factors that are responsible for the gap between theoretical and real application performance. Some depend on the architecture while some others are inherent to parallel computing. As an example, theoretical performance declared by vendors are obtained in par-

ticular conditions that are usually far from real cases. GPU vendors advertise their products with peak throughput of some TFLOP/s. However this number is obtained by pretending that all arithmetic units are continuously busy with fused multiply-add operations which count as two FLOPs, but require only one operation on GPU. On the other hand, as stated in the Amdahl's law (see also section 1.1) if an algorithm contains steps that cannot be fully parallelized, parallel speedup vs. serial speedup stagnates at a certain level and cannot be improved by parallel computation.
Taking these factor in considerations, even highly parallel applications can be subjected to the scaling limitations described by Amdahl's low.

5.3 Applications and Algorithms

As mentioned before, several algorithms can benefit from heterogeneous architectures. Comparing individual performance of an algorithm can be difficult, since data movement between host and device could vary. Each of these architectures has its benefits and its drawbacks. In general, the GPU is the best architecture when executing single-precision floating-point operations. In this case its performance is an order of magnitude better as compared to others architectures [23]. However the best result can be obtained when the computation presents independent set of data, disfavoring applications which require communication and synchronization.
On the other hand, it is difficult to quantify FPGA performance in terms of floating-point operations, since floating-point is usually avoided [23]. FPGAs are well exploited for algorithms where massively fixed-point, integer or bit operations are required. For these tasks, FPGAs present outstanding results and optimal performance per watt ratio. The main drawback of FPGAs is the time to program them. However, new domain specific languages, such as Maxeler [60] offer promising abstractions.

Conclusions

The contributions of this thesis are very practical. They address challenging computational problems in different scientific fields and use affordable hardware upgrades to considerably reduce computation time spent on these problems.
As expected it can be stated that computational intensive floating-point applications can be efficiently mapped on heterogeneous multi-GPU platforms.
In addition, although the high power consumption limits the parallelism level in heterogeneous multicore systems, an efficient scheduling policy contributes to keep their performance as high as possible.
Finally, the thesis proposed an graph implementation on reconfigurable architecture. As shown in section 1.3.7, for memory bounded problems GPUs do not work well, do to their high memory latency. In such a scenario FPGAs can theoretically offer better performance.

Appendix A

Acronyms

API Application Programming Interface

APP Accelerate Parallel Processing

ASIC Application Specific Integrated Circuit

ATX Advanced Technology eXtended

BFF Backfill First Fit

BFS Breadth-first search

BLE Basic Logic Element

CB Connection Box

CLB Configurable Logic Block

CPU Central Processing Unit

CRS Central Receiver System

CSR Compress Sparse Row

CUDA Compute Unified Device Architecture

DAG Direct Acyclic Graph

DRMS Distributed Resource Management System

EPS Entry-Level Power Supply Specification

FIFO First In First Out

FLOPs Floating Point Operations per second

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HPC High Performance Computing

ISA Instruction Set Architecture

LUT Look-Up table

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

PC Power Capping

PCI Peripheral Component Interconnect

PSU Power Supply Unit

PTX Parallel Thread eXecution

PU Processing Unit

RAM Random Memory Access

SATA Serial Advanced Technology Attachment

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SLI Scalable Link Interface

SM Streaming Multi-Processor

SP Streaming Processor

SRAM Static Random Memory Access

SW Switch Boxes

Bibliography

[1] Matteo Chiesi, Luca Vanzolini, Eleonora Franchi Scarselli, and Roberto Guerrieri. Accurate optical model for design and analysis of solar field based on heterogeneous multicore systems. *Renewable Energy*, 55:241–251, 2013.

[2] Matteo Chiesi, Luca Vanzolini, Claudio Mucci, Eleonora Franchi Scarselli, and Roberto Guerrieri. Power-aware job scheduling on heterogeneous multicore architectures. *Parallel Distrib. Syst., IEEE Trans. on*, Under minor revision.

[3] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[5] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

[6] Rio Yokota and Lorena Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science & Engineering*, 14(3):30–39, 2012.

[7] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[8] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. The MIT press, 2001.

[9] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[10] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In

Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE Int. Conf. on, pages 1–8. IEEE, 2009.

[11] ORNL. Titan project timeline, December 2012.

[12] Florian Ries. *Heterogeneous Multicore Architecture for Digital Signal Processing*. PhD thesis, ARCES - University of Bologna, 2011.

[13] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[14] F Pollack. Pollack's rule of thumb for microprocessor and area. http://en.wikipedia.org/wiki/Pollack's_Rule.

[15] Jinuk Luke Shin, Kenway Tam, Dawei Huang, Bruce Petrick, Ha Pham, Changku Hwang, Hongping Li, Alan Smith, Timothy Johnson, Francis Schumacher, et al. A 40nm 16-core 128-thread cmt sparc soc processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 98–99. IEEE, 2010.

[16] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[17] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[18] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[19] Jack J Dongarra, Hans W Meuer, Horst D Simon, and Erich Strohmaier. Recent trends in high performance computing. *The Birth of Numerical Analysis*, page 93, 2009.

[20] Mahmoud Hassaballah, Saleh Omran, and Youssef B Mahdy. A review of simd multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal*, 51(6):630–649, 2008.

[21] Kai Hwang, A Ramachandran, and R Purushothaman. *Advanced computer architecture: parallelism, scalability, programmability*, volume 199. McGraw-Hill New York, 1993.

[22] Federal Energy Management Program. Data Center Energy Consumption Trends. Technical report, U.S. Department of Energy, 2009.

[23] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[24] AMD. Amd graphics cards, April 2013.

[25] NVIDIA CORPORATION. Nvidia graphics cards, December 2013.

[26] Christopher R Clark and David E Schimmel. Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 249–257. IEEE, 2004.

[27] Chang Shu, Soonhak Kwon, and Kris Gaj. Fpga accelerated tate pairing based cryptosystems over binary fields. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 173–180. IEEE, 2006.

[28] Roger Woods, John McAllister, Gaye Lightbody, and Ying Yi. *Front Matter*. Wiley Online Library, 2008.

[29] Ben Cope, Peter YK Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *Computers, IEEE Transactions on*, 59(4):433–448, 2010.

[30] AMD. Accelerated parallel processing (app) sdk, December 2013.

[31] NVIDIA. Compute unified device architecture (cuda) sdk, December 2013.

[32] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[33] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 31. IEEE Press, 2008.

[34] NNVIDIA. Cuda c programming guide, December 2013.

[35] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[36] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[37] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-Based Heterogeneous FPGA Architectures*. Springer, 2012.

[38] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic Publishers, 1999.

[39] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture*. Now Publishers Inc., 2008.

[40] Daniel C Guterman, ISAM H Rimawi, Te-Long Chiu, RICHARD D Halvorson, and DJ McElroy. An electrically alterable nonvolatile memory cell using a floating-gate structure. *Electron Devices, IEEE Transactions on*, 26(4):576–586, 1979.

[41] J al Birkner, A Chan, HT Chua, A Chao, K Gordon, B Kleinman, P Kolze, and R Wong. A very-high-speed field-programmable gate array using metal-to-metal antifuse programmable elements. *Microelectronics Journal*, 23(7):561–568, 1992.

[42] David Marple and Larry Cooke. An mpga compatible fpga architecture. In *Custom Integrated Circuits Conference, 1992., Proceedings of the IEEE 1992*, pages 4–2. IEEE, 1992.

[43] QuickLogic Corporation. Eclipse ii family data sheet. http://www.quicklogic.com/assets/pdf/data_sheets/, 2013.

[44] Abbas El Gamal, Jonathan Greene, Justin Reyneri, Eric Rogoyski, Khaled A El-Ayat, and Amr Mohsen. An architecture for electrically configurable gate arrays. *Solid-State Circuits, IEEE Journal of*, 24(2):394–398, 1989.

[45] William Carter, Khue Duong, Ross H Freeman, H Hsieh, Jason Y Ja, John E Mahoney, Luan T Ngo, and Shelly L Sze. A user programmable reconfigurable gate array. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1986.

[46] Sau C Wong, HC So, Jung H Ou, and John Costello. A 5000-gate cmos epld with multiple logic and interconnect arrays. In *Custom Integrated Circuits Conference, 1989., Proceedings of the IEEE 1989*, pages 5–8. IEEE, 1989.

[47] Xilinx. http://www.xilinx.com/, 2013.

[48] Altera. http://www.altera.com/, 2013.

[49] Aditya A Aggarwal and David M Lewis. Routing architectures for hierarchical field programmable gate arrays. In *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*, pages 475–478. IEEE, 1994.

[50] Jonathan Rose and Zvonko G Vranesic. *Field programmable gate arrays*, volume 180. Springer, 1992.

[51] Xilinx. Virtex 6. http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/, 2013.

[52] Xilinx. Virtex 7. http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/, 2013.

[53] Altera. Stratix 5. http://www.altera.co.uk/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp, 2013.

[54] Altera. Stratix 10. http://www.altera.co.uk/devices/fpga/stratix-fpgas/stratix10/stx10-index.jsp, 2013.

[55] Robert K Brayton, Gary D Hachtel, and Alberto L Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[56] Robert K Brayton. The decomposition and factorization of boolean expressions. In *Proc. Int. Symp. Circ. Sys.(ISCAS 82) Rome*, 1982.

[57] Jason Cong and Yuzheng Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(1):1–12, 1994.

[58] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117. ACM, 1995.

[59] Jon Frankle. Iterative and adaptive slack allocation for performance-driven layout and fpga routing. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 536–542. IEEE Computer Society Press, 1992.

[60] Maxeler Technologies. Multiscale dataflow programming. http://www.maxeler.com, 2013.

[61] Janet L Sawin. Charting a new energy future. *State of the World*, pages 85–109, 2003.

[62] U.S. Department of Energy. Basic research needs for solar energy utilization. Technical report, Washington, DC, USA, 2005.

[63] Ramteen Sioshansi and Paul Denholm. The value of concentrating solar power and thermal energy storage. *Sustainable Energy, IEEE Transactions on*, 1(3):173–183, 2010.

[64] Volker Quaschning. Technical and economical system comparison of photovoltaic and concentrating solar thermal power systems depending on annual global irradiation. *Solar Energy*, 77(2):171–178, 2004.

[65] Robert Pitz-Paal, Nicolas Bayer Botero, and Aldo Steinfeld. Heliostat field layout optimization for high-temperature solar thermochemical processing. *Solar energy*, 85(2):334–343, 2011.

[66] PL Leary and JD Hankins. User's guide for mirval: a computer code for comparing designs of heliostat-receiver optics for central receiver solar power plants. Technical report, Sandia Labs., Livermore, CA (USA), 1979.

[67] Tim Wendelin. Soltrace: a new optical modeling tool for concentrating solar optics. ASME, 2003.

[68] CL Laurence, FW Lipps, and LL Vanthull. User's manual for the university of houston individual heliostat layout and performance code. *NASA STI/Recon Technical Report N*, 85:21808, 1984.

[69] Bruce L Kistler. A user's manual for delsol3: A computer code for calculating the optical performance and optimal system design for solar thermal central receiver plants. Technical report, Sandia National Labs., Livermore, CA (USA), 1986.

[70] Pierre Garcia, Alain Ferriere, and Jean-Jacques Bezian. Codes for solar flux calculation dedicated to central receiver system applications: a comparative review. *Solar Energy*, 82(3):189–197, 2008.

[71] Patricia Kuntz Falcone. A handbook for solar central receiver design. Technical report, Sandia National Labs., Livermore, CA (USA), 1986.

[72] William B Stine and Michael Geyer. *Power from the Sun*. Power from the sun. net, 2001.

[73] Xiudong Wei, Zhenwu Lu, Zhifeng Wang, Weixing Yu, Hongxing Zhang, and Zhihao Yao. A new method for the design of the heliostat field layout for solar tower power plant. *Renewable Energy*, 35(9):1970–1975, 2010.

[74] Allen Nussbaum and Richard A Phillips. Contemporary optics for scientists and engineers. *Contemporary Optics for Scientists and Engineers by Allen Nussbaum, Richard A. Phillips New Jersey: Prentice Hall, INC, 1976*, 1, 1976.

[75] Paul Bendt and Ari Rabl. Optical analysis of point focus parabolic radiation concentrators. *Applied Optics*, 20(4):674–683, 1981.

[76] European Commission Institute for Energy and Transport. Photovoltaic geographical information system (pvgis). http://re.jrc.ec.europa.eu/pvgis/.

[77] National Renewable Energy Laboratory (NREL). National solar radiation data base. http://rredc.nrel.gov/solar/old_data/nsrdb/1991-2005/tmy3/.

[78] Lorenzo Pancotti. Optical simulation model for flat mirror concentrators. *Solar energy materials and solar cells*, 91(7):551–559, 2007.

[79] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 31. IEEE Press, 2008.

[80] FMF Siala and ME Elayeb. Mathematical formulation of a graphical method for a no-blocking heliostat field layout. *Renewable Energy*, 23(1):77–92, 2001.

[81] D. Valerio Fernndez. a 11.0-mwe solar tower power plant with saturated steam receiver. Technical report, Solucar, 2004.

[82] eSolar. Sierra suntower. http://www.esolar.com/sierra_fact_sheet.pdf, 2012.

[83] Steffen Ulmer, Tobias März, Christoph Prahl, Wolfgang Reinalter, and Boris Belhomme. Automated high resolution measurement of heliostat slope errors. *Solar Energy*, 85(4):681–687, 2011.

[84] Alanod Solar. Miro90 technical specifications. http://alanod-solar.com/opencms/opencms/Reflexion/Technische_Info.html, 2012.

[85] KK Chong, FL Siaw, CW Wong, and GS Wong. Design and construction of non-imaging planar concentrator for concentrator photovoltaic system. *Renewable Energy*, 34(5):1364–1370, 2009.

[86] Michael Showerman, Jeremy Enos, Avneesh Pant, Volodymyr Kindratenko, Craig Steffen, Robert Pennington, and Wen-mei Hwu. Qp: a heterogeneous multi-accelerator cluster. In *Proc. 10th LCI Int. Conf. on High-Performance Clustered Computing*, 2009.

[87] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008.

[88] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W Keller. Ship: A scalable hierarchical power control architecture for large-scale data centers. *Parallel Distrib. Syst., IEEE Trans. on*, 23(1):168–176, 2012.

[89] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *Parallel Distrib. Syst., IEEE Trans. on*, 21(5):658–671, 2010.

[90] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *Proc. of the 2006 ACM/IEEE Conf. on Supercomputing*, pages 14–14. IEEE, 2006.

[91] Nandini Kappiah, Vincent W Freeh, and David K Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Proc. of the 2005 ACM/IEEE Conf. on Supercomputing*, page 33. IEEE Computer Society, 2005.

[92] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proc. of the 2005 ACM/IEEE Conf. on Supercomputing*, page 1. IEEE Computer Society, 2005.

[93] Kyong Hoon Kim, Rajkumar Buyya, and Jong Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *Proc. of the seventh IEEE Int. Symp. on cluster computing and the grid*, pages 541–548, 2007.

[94] Xiaorui Wang and Ming Chen. Cluster-level feedback power control for performance optimization. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th Int. Symp. on*, pages 101–110. IEEE, 2008.

[95] Maja Etinski, Julita Corbalan, Jesús Labarta, and Mateo Valero. Parallel job scheduling for power constrained hpc systems. *Parallel Computing*, 2012.

[96] Bin Lin, Arindam Mallik, Peter Dinda, Gokhan Memik, and Robert Dick. User-and process-driven dynamic voltage and frequency scaling. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE Int. Symp. on*, pages 11–22. IEEE, 2009.

[97] Li Tang and Yiji Zhang. Low-power task scheduling for gpu energy reduction.

[98] Nouveau Wiki. Accelerated open source driver for nvidia cards, December 2012.

[99] Krhonos group. Opencl, April 2013.

[100] AMD. Gpu open-source drivers., April 2013.

[101] Cong Liu, Jian Li, Wei Huang, Juan Rubio, Evan Speight, and Xiaozhu Lin. Power-efficient time-sensitive mapping in heterogeneous systems. In *Proc. of the 21st Int. Conf. on Parallel architectures and compilation techniques*, pages 23–32. ACM, 2012.

[102] William Lloyd Bircher and Lizy Kurian John. Core-level activity prediction for multicore power management. *Emerging and Selected Topics in Circuits and Systems, IEEE J. on*, 1(3):218–227, 2011.

[103] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency. *Computer Science-Research and Development*, 27(4):277–287, 2012.

[104] Edward Anderson. *LAPACK Users' guide*, volume 9. Siam, 1999.

[105] UsersGuide PLASMA. Parallel linear algebra software for multicore architectures. *Version*, 2(4):5, 2011.

[106] Charles Lively, Xingfu Wu, Valerie Taylor, Shirley Moore, Hung-Ching Chang, Chun-Yi Su, and Kirk Cameron. Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems. *Computer Science-Research and Development*, 27(4):245–253, 2012.

[107] Yoshihiko Hotta, Mitsuhisa Sato, Hideaki Kimura, Satoshi Matsuoka, Taisuke Boku, and Daisuke Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a pc cluster. In *Parallel and Distributed Processing Symp., 2006. IPDPS 2006. 20th Int.*, pages 8–pp. IEEE, 2006.

[108] Jin Heo, Praveen Jayachandran, Insik Shin, Dong Wang, Tarek Abdelzaher, and Xue Liu. Optituner: On performance composition and server farm energy minimization application. *Parallel Distrib. Syst., IEEE Trans. on*, 22(11):1871–1878, 2011.

[109] Wu Ye, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proc. of the 37th Annual Design Automation Conf.*, pages 340–345. ACM, 2000.

[110] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News*, 28(2):83–94, 2000.

[111] Reiji Suda et al. Accurate measurements and precise modeling of power dissipation of cuda kernels toward power optimized high performance cpu-gpu computing. In *Parallel and Distributed Computing, Applications and Technologies, 2009 Int. Conf. on*, pages 432–438. IEEE, 2009.

[112] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proc. of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[113] Pat Bohrer, Elmootazbellah N Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case for power management in web servers. In *Power aware computing*, pages 261–289. Springer, 2002.

[114] Olli Mämmelä, Mikko Majanen, Robert Basmadjian, Hermann De Meer, André Giesler, and Willi Homberg. Energy-aware job

scheduler for high-performance computing. *Computer Science-Research and Development*, 27(4):265–275, 2012.

[115] Florian Ries, Tommaso De Marco, and Roberto Guerrieri. Triangular matrix inversion on heterogeneous multicore systems. *Parallel Distrib. Syst., IEEE Trans. on*, 23(1):177–184, 2012.

[116] Anne Krampe, Joachim Lepping, and Wiebke Sieben. A hybrid markov chain model for workload on parallel computers. In *Proc. of the 19th ACM Int. Symp. on High Performance Distributed Computing*, pages 589–596. ACM, 2010.

[117] Nima Sharifimehr and Samira Sadaoul. Markovian workload modeling for enterprise application servers. In *Proc. of the 2nd Canadian Conf. on Computer Science and Software Engineering*, pages 161–168. ACM, 2009.

[118] NVIDIA CORPORATION. Gpu gtx 590 specifications, December 2012.

[119] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.

[120] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[121] Liangjun Zhang, Young J Kim, and Dinesh Manocha. A simple path non-existence algorithm using c-obstacle query. In *Algorithmic Foundation of Robotics VII*, pages 269–284. Springer, 2008.

[122] Avneesh Sud, Erik Andersen, Sean Curtis, Ming Lin, and Dinesh Manocha. Real-time path planning for virtual agents in dynamic environments. In *ACM SIGGRAPH 2008 classes*, page 55. ACM, 2008.

[123] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[124] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[125] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 25–25. IEEE, 2005.

[126] PR Venkata Subramaniam and Kam-Hoi Cheng. A fast graph search multiprocessor algorithm. In *Aerospace and Electronics Conference, 1997. NAECON 1997., Proceedings of the IEEE 1997 National*, volume 1, pages 247–254. IEEE, 1997.

[127] Michael Delorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomás E Uribe, Thomas F Knight, and André DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 143–151. IEEE, 2006.

[128] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Efficient breadth-first search on the cell/be processor. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1381–1395, 2008.

[129] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15. IEEE, 2012.

[130] David A Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 523–530. IEEE, 2006.

[131] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[132] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.

[133] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. *Computer Science Department*, page 541, 2004.

[134] NVIDIA. Cuda community showcase. http://www.nvidia.com/object/cuda_showcase_html.html, 2013.

[135] Maxeler. Max-up publications. http://www.maxeler.com/, 2013.

[136] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[137] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 162–170. ACM, 2004.