Università di Pisa and Scuola Superiore Sant'Anna

Department of Computer Science,
Department of Information Engineering
and
Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

Master Thesis

Energy models in data parallel CPU/GPU computations

Candidate
Alessandro Lenzi

Supervisor	Referee
Prof. Marco Danelutto	Dott. Antonio Cisternino

Academic Year 2014/2015

Contents

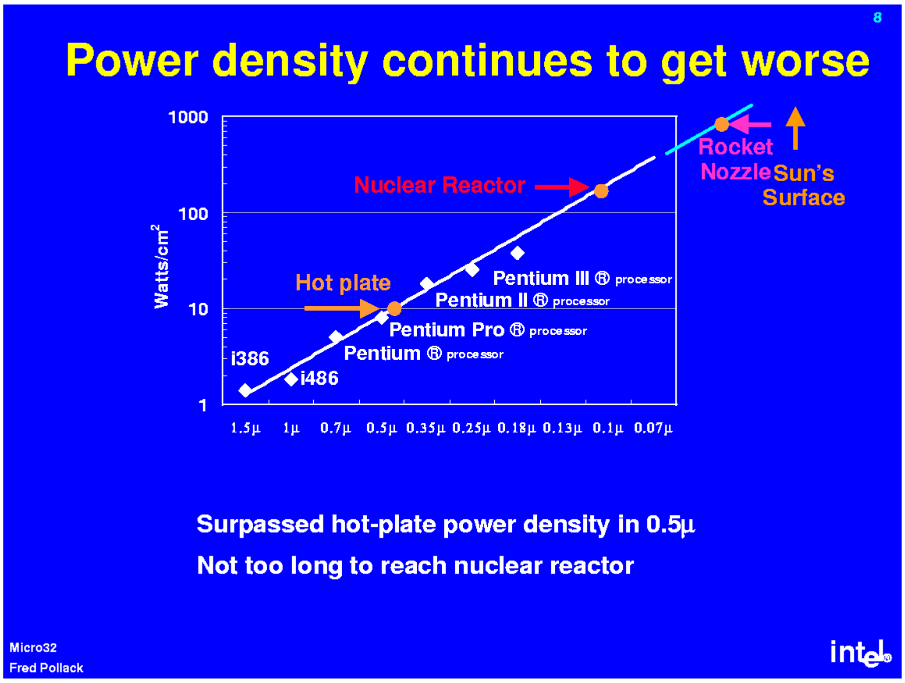
Introduction

Even though it has been exacerbated in the current days by the widespread of mobile computers, the energy problem is not a modern issue for computing. As reported in [Wei55], one of the first modern computer, the ENIAC - a 27 tons heavy and 167 m^2 wide monster - used to consume 173kW. This amount of power was so much for the time that, according to a common story, lights in Philadelphia used to dim whenever it was on. Different computing technologies have succeeded to the vacuum tubes used for ENIAC, but they all reached a point where their power dissipation was too much to go on.

In the 70s, moving from the vacuum tubes to bipolar transistors allowed to reach computing capabilities comparable to ENIAC's ones just dissipating a handful of watts (INTEL 4004, 1971). The 80s technologies brought new challenges in power dissipation and delivery. Often, super computers of the time needed liquid cooling to maintain high performances.

In the 90s, the CMOS technology started to become appealing also thanks to its interesting power behaviour: in fact, in such systems, power is dissipated mainly at switching transitions, thanks to *complementary gate design*. In other words, (theorically) energy is consumed only when the value of a gate changes from 0 to 1 or viceversa.

Even though at the beginning this technology seemed to be too slow to be adopted everywhere, advancements in technology and the interesting "green" properties led to the universal adoption that we witness today. This notwithstanding, eventually continuous CMOS scaling arrived to an end, together with the continuous uniprocessor performance scaling foreseen by *Moore's Law*. Decreasing the feature size and the the threshold voltage (the voltage over which current flows through a gate) brought to an increase in *leakage current*, that is the amount of power that flows through a transistor even though it is supposed to be off. As noted in a speech at Micro Keynote in 1999, this was making the power per unit area (expressed in $Watt/cm^2$) grow too much: the power density of a nuclear reactor was not long to be reached (see Figure 1). In fact, CMOS needs cooling mechanisms: while $200W$ spread over many square centimetres are quite easy to cool, in case of microchips (with relatively small area), cooling was going to become impossible. Heat is one of the biggest problems for computing, as it affects durability (it has been estimated that a $10°C$ increase in operational temperature reduces the lifetime of a chip by half) and makes power provisioning an issue.



Power density continues to get worse

Surpassed hot-plate power density in 0.5μ

Not too long to reach nuclear reactor

Figure 1: Power density trends as of 1999. Image by Fred Pollack, Intel.

From 2005 this triggered, as a response from CPUs manufacturers, a switch to multi-core as an attempt to maintain *Moore's Law* gain in terms of performance while maintaining manageable heat and power.

Another solution provided by architects has been progressive specialization of units; with the rise of *heterogeneous computing*, almost every device has specialized units, like accelerators and GPUs. Currently, even mobile phones are equipped with multiple cores and GPUs. From the mobile perspective, batteries growth in capacity does not keep pace with the increase in terms of power requirements of the devices. Moreover, such devices often have to deal with very power-hungry antennas, which are responsible (together with the now-trending huge screens) for most of the battery drain.

But also in areas of IT where energy bills have been neglected for a long time, power aware computing is becoming an area of importance, as testified by initiatives like Green500[1]. Environmental concerns are not a minor problem: in the 2006 the data centers placed in the US consumed an amount of energy of 61.4 Billions of KWhs [ENE07] [LSH10], equivalent to the one consumed by the whole manifacturing industry. This corresponds to about 2% of the worldwide greenhouse gas emission [BLO08]. However, with the world's ever-growing need for computational power and communication, it has been estimated that by 2020 the footprint will grow to reach the 3%.

[1] `http://www.green500.org/`

Reaching the target of *exascale computing* (10^{18} floating point operations per second) using current supercomputing technology will require, according to [Mil10], several gigawatts of power, thus imposing the introduction of power management techniques.

In this work we address the problem of *modelling energy consumption* in heterogeneous architectures. CPU and GPU architectures have been developed with different kinds of application in mind; this means that they are optimized for different tasks and will have, depending on the computation, different energy footprint. Deciding how a computation should be split among devices with different capabilities is crucial to minimize both energy consumption and completion time. To perform this task, a model providing an estimation of the consumption of different possible configurations is needed. This work represents a contribution in the field of energy-aware parallel computing, with which we want to provide a method to allow optimization of energy consumption in parallel application developed according to a structured methodology, namely using a parallel design pattern, a skeleton or an implementation following a well-known parallel exploitation pattern.

The main contributions of this work are:

- the proposal of an iterative process for developing energy models
- the individuation and characterization of the impact on power of different, high-level explanatory variables for GPU architectures
- a model for CPU and GPU architectures (as well as an heuristic for the latter), providing accurate energy predictions (with very high probability), together with a practical example demonstrating how this model can be used to save energy.

Outline

The work is organized as follows.

In **Chapter 1**, we give an overview of the background concepts needed to better understand this work: we introduce heterogeneous system, with a particular focus on the structure of GPU Architectures (and how they can be programmed). We then move to introduce some basic concepts of structured parallel programming and to explain how energy is dissipated in a computing environment.

In **Chapter 2**, we provide a quick primer on methods for energy saving used in IT systems.

We move, in **Chapter 3**, to present an iterative process that can be used to achieve a proper model for energy estimation.

The methodology proposed in Chapter 3 is then used in **Chapter 4**, were a model and an heuristic for estimating energy consumption on GPU architectures is presented. Both the heuristic and the model are validated and their accuracy in estimating energy consumption is assessed by means of proper computations.

Eventually, in **Chapter 5** we present and validate a model for CPU energy consumption estimation for data-parallel map computations.

In **Chapter 6**, we merge the two models and give an overview of potential practical usages of the proposed models. The potential advantages of using the presented model is demonstrated by co-scheduling an application on CPU/GPU core mixes, with the aim of minimizing different energy measurement functions.

Chapter 1

Background

The aim of this chapter is to provide the basic concepts that will be later used in the rest of this work. In the first part, we will describe the systems available today, both in simple, mobile environments and in more complex, potentially distributed systems. We will characterize the different kinds of devices with which we interact everyday, specially focusing on the now pervasive *Graphic Processing Units*, stressing their architecture and thereafter briefly exposing how they can be used for general purpose computation. A quick primer on structured parallel programming concepts, algorithmic skeletons/parallel design patterns and behavioural skeletons is given for the convenience of the reader unfamiliar with such concepts. Finally the sources of energy consumption in nowadays systems will be analysed in depth in the last section of this chapter.

1.1 Heterogeneous Computing

Currently, almost every computing system is characterized by an high level of parallelism: usually, every device is equipped with multiple cores and with some highly specialized coprocessors. As performance of a single core are becoming less and less satisfying, *parallel programming* is becoming ubiquitous. The introduction of parallelism in commodity computing calls for urgent actions also in terms of parallel applications, needed to exploit at their best currently available devices. A programmer developing an application should also consider the different capabilities of the computing devices that can be targeted.

According to *Flynn's Taxonomy* [Fly72], architectures can be characterized depending on the way in which data and instructions flow through the processing units, as it can be seen in Figure 1.1.

Single Instruction Stream Single Data Stream (SISD): is a classical CPU architecture, in which in any given moment a single instruction stream is in execution on a single data stream.

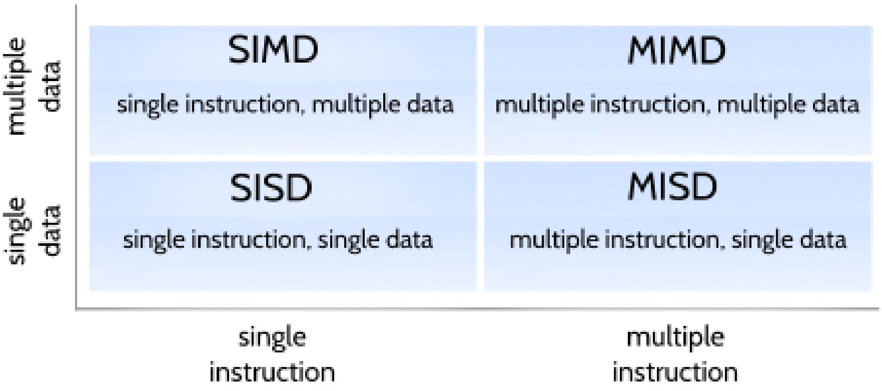

SIMD
single instruction, multiple data
MIMD
multiple instruction, multiple data
SISD
single instruction, single data
MISD
multiple instruction, single data
multiple data
single data
single instruction
multiple instruction

Figure 1.1: A depiction of Flynn's taxonomy.

Single Instruction Stream Multiple Data Stream (SIMD): is the model exploited by Graphic Processing Units - higly specialized components tuned for executing a fixed pipeline[1] of instructions, initially with graphics purposes - by vector units commonly embedded in modern CPUs and by array of processors. It implements a *data parallel* form of parallelism at the firmware and assembler level [Van14]. Data parallelism arises when data collections can be split and the resulting partisions processed independently (see Section 1.3). In this case it requires replicated workers to operate in the same manner (applying the same function) on different chunks of data.

The most pervasive and evident advantage of such architectures is exposed by vector units (VU), which allow programmers to achieve better completion time in their computations while writing sequential code, being usually *vectorization* a task carried on by the compiler. Consider the case of Intel's KNC, one of the most performing vector architectures available today. An operation in the form $for\ i = 1\ to\ N\ do\ c[i] = a[i] + b[i]$, since there are no dependencies between different iterations, can in principle be executed as:

$$c[0] = a[0] + b[0] \mid c[1] = a[1] + b[1] \mid \dots \mid c[N] = a[N] + b[N]$$

where | denotes the parallel composition operator. In case of a vector processing unit with KNC, with replicated workers accepting at every clock cycle either 8 `double` or 16 `float`, it will be rewritten by the compiler as

$$c[0:7] = a[0:7] + b[0:7];\ c[8:15] = a[8:15] + b[8:15];\ \dots$$

reducing completion time by applying the operator (addition, in the example) to a number of operands in parallel. This requires to extend the processor's ISA with special *SIMD instructions*, normally targeting arithmetic operations. These instructions will be executed in parallel by a set of identical data parallel workers, as it can be seen in Figure 1.2. In this case the data parallel principle is applied at a *fine-grain* level.

[1] a pipeline is the parallel analogous of a function composition

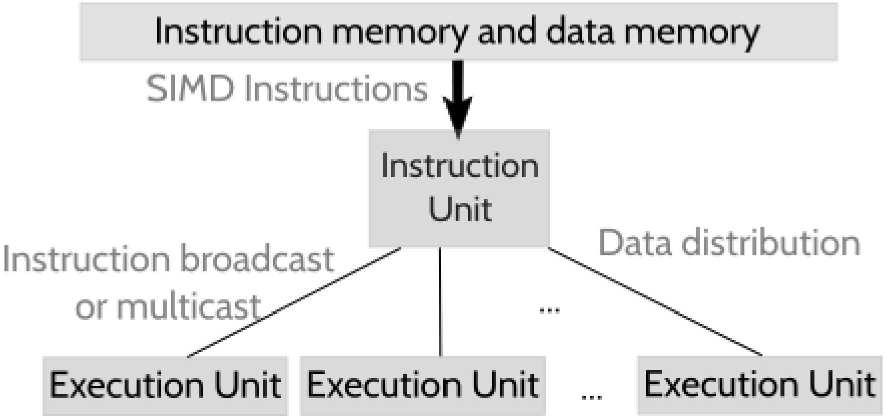


Figure 1.2: A SIMD architecture. Instruction Unit is responsible for fetching instructions (among which SIMD ones) and broadcasting/multicasting them to the execution units. Data distribution might be centralized as well as managed directly from execution units.

The same basic principle is applied for *Graphics Processing Units* (GPUs), which exploit the advantages in terms of performance and energy of SIMD processors with some trade-offs to increase programmability and alleviate performance problems. GPUs have been classically used to implement a pipeline of graphic operations, each stage being a data parallel implementation in the form of a *map*. In a map the same function is applied to different portions of data. Only recently, with the advent of GPGPU programming,these accelerators have become fully programmable.

Multiple Instruction Stream, Single data stream (MISD): this quite uncommon paradigm prescribes different cores operating on a unique stream of data, using different instructions.

Multiple Instruction Stream, Multiple data stream (MIMD): this is the parallel architecture paradigm. It comprises shared memory multiprocessors as well as distributed memory multicomputers. Because of their inherently general purpose nature they allow to implement any form of parallel computation, without intrinsic limitations. In general, MIMD architectures can be seen as composed by a set of processing units, connected through an *interconnection subsystem*, through which communication, synchronization and data exchanges are carried on. We defer to [Van14] for further reading.

Usually different kinds of parallel architectures are embedded in a heterogeneous system: together with multiple cores, we usually find a GPU. In such systems, tasks are offloaded to computing units with different capabilities, achieving advantages in terms of energy consumption as well as in terms of bare performance. This can be achieved only at the cost of increased complexity for the programmer, despite the introduction of high-level tools, like NVIDIA *Compute Unified Device Architecture*[2] (CUDA). It is possible to visualize a common implementation of an heterogeneous architecture in Figure 1.3, drawn from [CGM14].

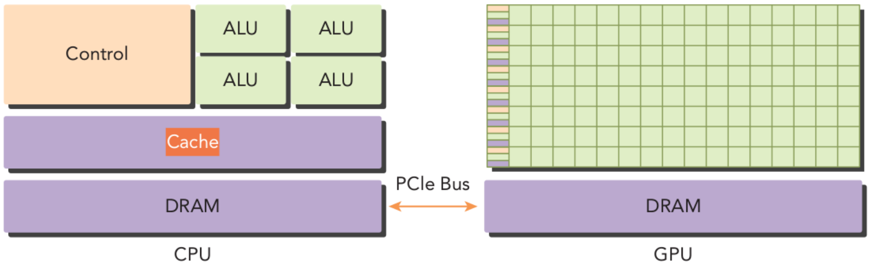


Figure 1.3: Common, widespread heterogeneous architecture schema. CPU and GPU subsystems are connected through a PCIexpress bus.

Following the approach commonly used in the world of GPU programming, we will refer to the - possibly multiple - CPUs as the *host(s)* and to the GPU as the *device*. Similarly, we consider a parallel application as composed of a *host part* and *device part*, depending on where it is intended to run; each part can be composed of different concurrent activities, all carrying on different part of the computation and organized accordingly to the target device architecture.

GPUs and CPUs have been designed for very different kinds of computations: while CPUs are overly optimized in terms of *latency*, GPUs are *throughput*-oriented devices. For this reasons, GPU are made to execute in parallel computational intensive programs. While CPUs are composed of few cores, with hardware support only for few execution flows, GPUs provide hardware parallelism exploitation, but with a less complex control flow management: complex mechanisms such as *branch prediction* do not exist in GPUs.

In a parallel, heterogeneous application, CPUs should be preferred for executing the tasks with unpredictable or complex control flows, sequential portions or in case the data to be processes is not big enough to justify the activation of the device. On the other hand, significant improvements can be achieved by exploiting GPU on data parallel, compute intensive tasks. The reason will be clear in Section 1.2, where an overview of GPU architectures, together with the CUDA programming model and its features, will be described.

1.2 GPU Architectures and CUDA

Graphic Processing Units follow the approach described in the SIMD paragraph of Section 1.1, with the introduction of some additional complexity to achieve better results in terms of programmability and flexibility.

As said before, first GPUs where at first dedicated parallel processors, meant to execute a special kind of computation: they were highly optimized and tuned

[2] `http://www.nvidia.com/object/cuda_home_new.html`

to execute primitives to elaborate 2D and 3D images to be displayed to the users. Their basic model was a pipeline of data parallel (map) functions, operating on multiple dimensional arrays. As the main market for such devices was videogaming, the main concerns where:

- render complex images, with complex optical effects
- provide seamless motions
- and do it fast, to provide better user experience

This required a special class of hardware accelerators to be developed, giving the rise to the industry of GPU production. In the beginning [Lue08], the hardware was built so that it resembled the graphics pipeline, shown in Figure 1.4.

The stages of the pipeline were implemented in hardware, and the programmer was only allowed to specify parameters. With the growth of graphics requirements, more stages have been added and *programmability capabilities* introduced in some of the stages. Since the early 2000's [McC10] programmers received the capability to send, along with the data, small programs (*shaders*) that operate on the data while in the pipeline. In principle, shaders where meant to use a screen position (x, y) and some additional information [SK10] in order to calculate the colour of the pixel in said position. Early adopter noticed immediately that, since the arithmetic was completely programmable, input and output colours could actually represent any kind of data.

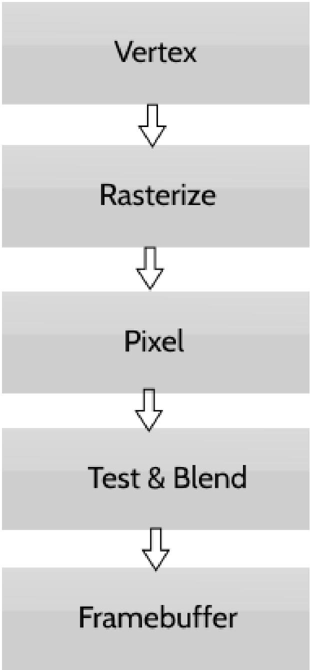


Figure 1.4: Logical depiction of the *graphics pipeline*

In the beginning of the era of *General Purpose GPU Programming* (GPGPU), the only way to interact with such devices was using OpenGL[3] or DirectX[4], requiring data to be transformed (unnaturally) in textures and programs to be rewritten in *shading languages*. GPUs harness both *task parallelism* and *data parallelism*. In a old GPU, the graphics pipeline was used for task parallelism, while the high number of parallel workers was used to process in parallel different portion of the images. However, as shaders started becoming more and more complex, maintaining the balance between different stages[5] became increasingly complex, leading architects to design *unified shader architectures* [OHL+08], available since the issue of NVIDIA GeForce 8800. In this model, *unified*, programmable shaders

[3] https://en.wikipedia.org/wiki/OpenGL

[4] https://en.wikipedia.org/wiki/DirectX

[5] in any pipeline, the throughput depends on the performance of the slowest stage

where provided: the hardware was equipped with fully programmable *Streaming Multiprocessors*, transforming the hardware graphics pipeline in a fully software abstraction. This was the first card supporting NVIDIA *CUDA*'s architecture, meant to harness the general purpose computational power of GPUs. Since CUDA and NVIDIA devices have been the main object off this work, in the following we will mainly use CUDA jargon and refer, without loss of generality, to NVIDIA GPUs.

1.2.1 GPU Architectures

Graphics Processing Units, since introduction of GeForce 8800 in 2007, started following a trend allowing general purpose programmability. These co-processors follow the *Single Instruction, Multiple Threads* model, a fashion of SIMD architectures with some trade-offs to enhance performances.

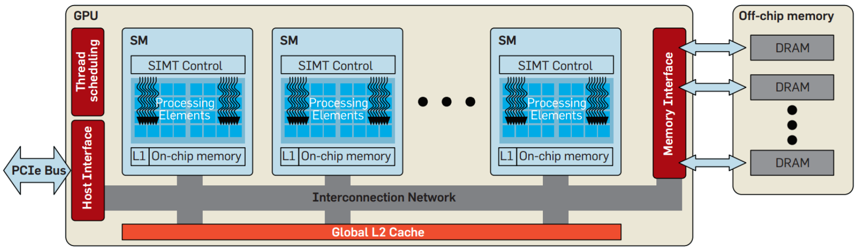


Figure 1.5: Depiction of an NVIDIA GPU architecture, taken from [GK10]

In normal SIMD, vector based machines, the ISA is extended with special vector instructions, that can be generated either by the compiler through *auto-vectorization* or by exploiting high-level, close to machine level special calls, called *intrinsics* [GK10]. In the case of SIMT, instead, each thread is constituted of a flow of *scalar instructions*, that are executed on simple processors in a SIMD fashion. Several threads' private scalar instructions are combined together to build a SIMT instruction. This technique allows for higher programmability, as the management of the vector instruction is automatically done in hardware and does not impact on the way the programmer writes the code: threads may cooperate in a data parallel computation or might operate differentially in a task-parallel fashion [ND10]. Complex mechanisms, such as synchronization barriers, atomic instructions and shared memory, are provided to increase programmability of the machine; not only *map* computations, but also more complex parallel structures (e.g. *stencil*, *reduce* etc.) can be implemented using the provided capabilities. Latency due to the communication with the host memory can be hidden overlapping it with computation, thanks to a dedicated coprocessor used to transfer data from the host memory [CGM14].

Current GPU Architectures supporting CUDA are built around an array of multiprocessors, called *Streaming Multiprocessors* (SMs), as it can be seen in

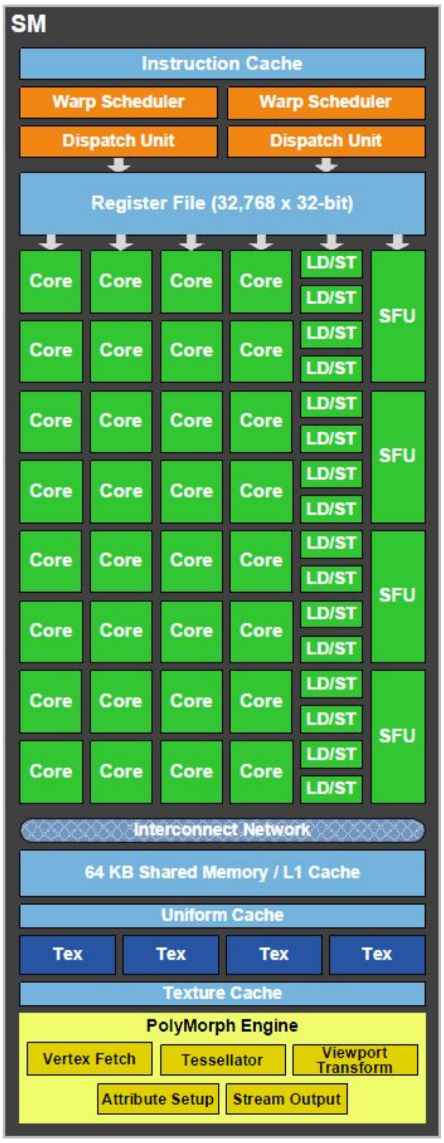


Figure 1.6: Depiction of the structure of a Streaming Multiprocessor, from [NVIa]

Figure 1.5. Horizontal scalability (and different capabilities for different market segments) can be brought by changing the number of Streaming Multiprocessor and the size and offered bandwidth of the memory.

Every SM is hardware multithreaded, supporting an elevated number (order of thousands) of threads. Threads are scheduled in groups of 32, called *warps*. In GPUs different warps are interleaved, to hide the latency of the long pipeline (10-20 clock cycles for arithmetic operations, 400-800 for global memory access) required by the operations. Switching threads has a negligible cost, as their context is entirely hardware managed [LNOM08]. Every thread has its own instruction address counter and register state, and are thus free to advance on its own and possibly to diverge even within a warp [NVI15].

As it can be seen in Figure 1.6, an SM is made of several components:

- Instruction Cache;
- *Warp schedulers*, responsible for the selection and execution of warps. A new warp is selected every 2-4 clock cycles. It operates at half of the processors' clock rate [LNOM08];
- one or more *Dispatch units*, responsible of retrieving the selected threads instructions. In Kepler architectures, more dispatch units are provided for each warp scheduler, allowing more (independent) instructions to be dispatched in a single cycle [NVI12];
- a *Register File*;
- load and store units (LD/ST);
- special function units (SFU);
- *CUDA cores*, corresponding to execution units;
- an *interconnection network*;
- *Shared Memory & L1 Cache*, allowing communication between threads operating in the same block;
- a *read-only* texture cache, that can be used to improve performances.

Active Warps (i.e. set of threads assigned to a SM) can be either in state *stalled*, *eligible* or *selected*. When less than 32 CUDA cores are available for execution or arguments of the current instruction of the warp are not yet available, a warp is stalled; otherwise, it is eligible. Warps to be run are chosen according to scheduling policies between the eligible ones, and in this case they pass to selected state. Instructions of 32 scalar threads are combined in a single SIMT instruction, executed concurrently on different workers. While warps can be executed in any order, within a warp we have in-order processing of the instructions.

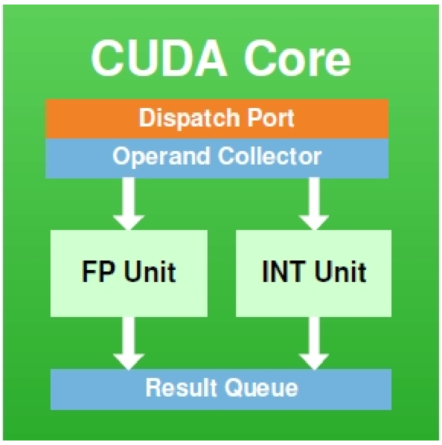


Figure 1.7: Structure of a CUDA Core, drawn from [NVI09]

The workers executing the threads are called *CUDA cores* (visible in Figure 1.7). They have a fully-pipelined arithmetical-logical unit and a floating point unit, accepting one instruction per clock cycle. Since a warp executes one common instruction at a time, as it happens for SIMD architectures, also threads in GPUs are affected by the *divergent branch problem*, arising in case of *data dependent branches*. Consider the following computation:

```
for i = 1 to N:
  if a[i] < b[i] then c[i] = a[i] + b[i]
  else c[i] = a[i] - b[i]
```

Listing 1.1:

Once compiled, the generated assembler will look like

```
LOOP:
  LOAD a[i]
  LOAD   b[i]
  IF (a[i] < b[i]) THEN
  SUB a[i], b[i], c[i]
  STORE c[i]
  GOTO CONT
THEN:
  ADD a[i], b[i], c[i]
  STORE c[i]
CONT:
  INCR i
  IF (i < N) LOOP
```

Listing 1.2:

since we have a unified controller for the flow of the program, only a fraction of the workers will be enabled to execute in any given moment. As an example, thread 0 could need to execute the *then* branch, while thread 1's flow goes through the *else* one. Since they belong to the same warp, and the CUDA cores on which they are scheduled will always receive the same instruction, one of the alternative branches will be selected and the instructions contained in it executed by the enabled cores, while the others will remain idle, waiting for the correct branch to be selected. The final condition (i.e. the loop guard) is not *data dependent*, and does not pose similar issues.

The condition in which the execution path of threads in the same warp diverges is called *warp divergence*, and is illustrated in Figure 1.8: in this condition, threads that should execute the *else* branch stall until the *then* branch terminates (and viceversa), hence degrading performances proportionally to the number of conditional paths executed by the warp. However, being the warps narrower, the performance penalty is not as high as with previous GPU architectures. Being warp divergence completely hardware managed using a *branch synchronization stack*[LNOM08], it does not impact on programmability: this issue could be completely neglected by the programmer from the mere functional point of view.

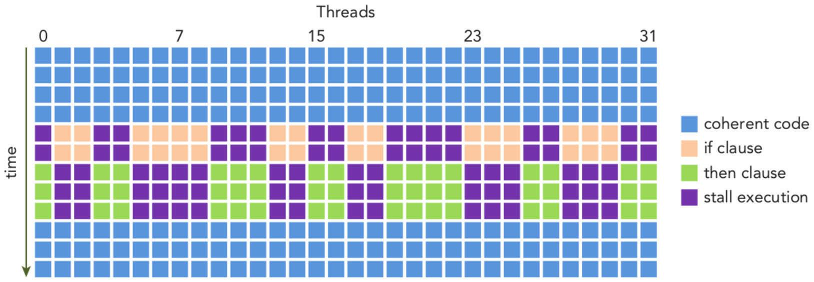


Figure 1.8: Illustration depicting the execution path taken by different threads a warp in case of warp divergence. Drawn from [CGM14]

Finally, some differences exist between different *versions* of architectures provided by NVIDIA:

Fermi

In the Fermi architecture, each of the SM is equiped with 32 CUDA Cores. It has 6 384-bit GDDR5 DRAM interfaces, supporting an high bandwidth for accessing a total of 6GB of memory provided on board. It has an L2 (coherent) cache, shared between all SMs, with a total size of 768KB. Every SM has 16 load and store units, hence allowing source and destination address to be calculated for a half warp every clock cycle.

Kepler

In this micro architecture every SM is equipped with 4 warp schedulers. For each warp scheduler, two instruction fetcher are provided, allowing more instructions to be fetched for each thread. Further, additional hardware parallelism is provided: the number of CUDA cores is increased to 192 per SM, 64 double precision units are provided and special function units become 32. The register file doubles in size, allowing more threads to be executed concurrently. A SM can support up to 2048 hardware threads in parallel. Also L1 and L2 cache sizes are increased. Features supporting, *dynamic parallelism* are introduced, allowing GPU to dynamically launch new kernels. Finally, a technology named *Hyper-Q* has been introduced; this technology adds more simultaneous hardware connections between the CPU and GPU, enabling CPU cores to simultaneously run more tasks on the GPU, increasing its utilization and reducing CPU idle time. In this work we used two different models (K20C and K40M) of this architecture.

1.2.2 CUDA

Compute Unified Device Architecture, commonly referred as CUDA, is a term used to describe both a parallel computing platform and a set of APIs created by NVIDIA to harness the computational power of General Purpose Graphics Processing Units. It mainly allows to express data parallel computations, while a limited support for task parallelism is also provided.

Using CUDA and the provided proprietary compiler, `nvcc`, the programmer is allowed to specify the way a thread executes by using plain C (C++ and Fortran are also supported), together with some proprietary functions for synchronization and atomic operations. Computations are expressed in terms of *kernels*; a kernel is instantiated in multiple threads and different behaviours can be defined using the thread identifier, which is provided by CUDA C, the version of ANSI C used to write kernels. Threads instantiation, scheduling and termination is entirely managed by the underlying system, alleviating the burden of programming GPUs.

A computation is typically structured in three phases:

1. Data transfer from Host memory to Device memory, using `cudaMemcpy` [6]

2. Kernel call, triggering a computation on the GPU which operates on the data previously transfer

3. Data collection, that is a transfer of the result from the Device global memory back to the host memory, using again `cudaMemcpy`

A computation running on the GPU is called *grid*. A grid is structured in terms of *blocks*, which are composed of several *threads*, the main execution component behaving as explained before. This is a two-level hierarchy, that should be used carefully to optimize computations. A depiction of the hierarchy can be seen in Figure 1.9

Threads

A thread belongs to a single block. They are assigned to a warp at creation time, using their id (sequentially). Threads execute the kernel function, and

[6] or variations

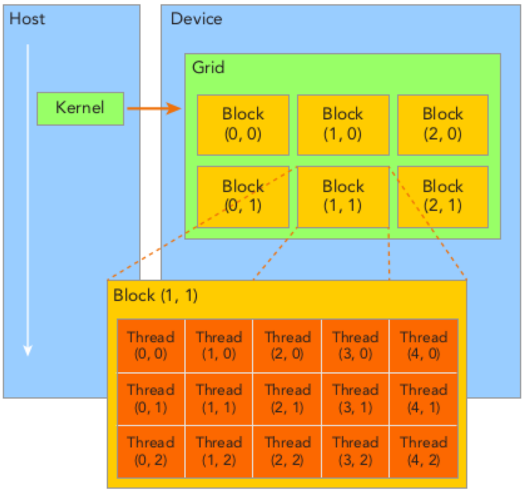


Figure 1.9: Grid's organization in the CUDA programming model, from [CGM14]. A kernel can be launched and executed asynchronously from the host, specifying the number of threads in a block and blocks in the grid.

can cooperate with other threads in the same block using global synchronization barriers (`__synchthreads()`) and *shared memory*. Shared memory is analogous to cache memory in CPUs from the performance viewpoint, except it can be managed explicitly and allows communication within a block. Every thread in the same grid shares the same global memory space. Threads can be optimized in terms of performance by using shared memory, texture and constant memory appropriately.

Blocks

Blocks represent a group of cooperating threads, organized in a 3-dimensional space. Blocks are physically mapped into a *SM*; once assigned, they will reside on the same SM for their whole lifetime.

Grids

A grid is a CUDA computation running on the GPU, with a unique global memory space. It is composed by a 2-dimensional array of blocks, that do not cooperate. A grid runs on a unique device, even in case several GPUs are present in the same heterogeneous architecture. In principle their execution is asynchronous with respect to host code execution, unless a `cudaDeviceSynchronize` is executed or the data collection is started: in this case the host process blocks, waiting for the result to be collected. Several grids can be in execution in parallel on the same device.

From a structured parallel programming point of view, CUDA is a *classical* parallel programming paradigm. As such, it provides basic mechanisms allowing to write parallel application, hence spanning concurrent activities, synchronization, mapping and scheduling. However, CUDA does not comprises facilities for exploiting known and efficient forms of parallelism, nor any abstraction to clearly separate business code from non-functional code.

We suggest the reader interested in CUDA to consult [CGM14, SK10, NVI15].

1.3 Structured Parallel Programming

Parallel computing is relative to the usage of two or more processing elements in combination to solve a single problem [Dan14]. The task of parallelizing a sequential application involves several *NP-hard* [Van14] problems, but during the years researchers studying efficient implementation noticed that the ways to implement efficient parallel computation tend to follow common patterns, called *parallelism forms* or *parallel paradigms.*

Coding parts common to such patterns so that parallelism is efficiently exploited is a very difficult task, requiring a kind of knowledge well discerned from the one necessary to code the application per se.

The need for separating *business code* from *parallel exploitation code*, lead to the emergence of *algorithmic skeletons* and, later, of *parallel design patterns*, all exploiting the same principle but produced by different research communities.

In both these approaches, the common parallel exploitation parallel is provided as a building block to the application programmer, which is consequently freed from the burden of using low level tools to express his target parallel computation. In the following, we will refer mainly to the world of algorithmic skeletons.

1.3.1 Algorithmic Skeletons

According to a well established definition, an **algorithmic skeleton** is a *parametric, reusable and portable programming abstraction modelling a known, common and efficient parallel pattern.* They are provided as parallel building blocks, available to the application programmer. He/she only needs to instantiate them with the proper business logic to efficiently exploit machine parallelism. Algorithmic skeletons are provided to the application programmer as a "framework", either as part of the language or as a library written in a host language. Adopting algorithmic skeleton brings several advantages:

- simplified parallel application development and rapid prototyping;
- guaranteed correctness of the parallel application;
- framework-guaranteed portability;
- thanks to the reduced number of parallel exploitation patterns, the framework is able to implement static and dynamic optimizations to the computation.

The main disadvantages of this approach of parallel exploitation resides in **functional and performance portability across different architecture** (from a system programmer's perspective) and in the imposed **limits in the form of parallelism to exploit**, meaning that in case there's no suitable skeleton in the framework to implement a particular form of parallelism, the application programmer is forced to use a classical (e.g. MPI/CUDA) framework.

Being skeleton structures designed to execute in parallel a different functions, they can be seen as *higher order function*, modelling parallel exploitation patterns. The function taken as argument represents the business code of the application. We can divide sources of parallelism exploitation in two main categories:

- **stream parallelism**, arising from computations relative to different, independent data items appearing in input
- **data parallelism**, arising from the decomposition of a single task in several subtasks

Let us define, abstractly, a stream as the following data type [7]:

```
type 'a stream = EmptyStream | Stream of 'a * 'a stream;;
```

Listing 1.3:

A *pipeline*, operating on the stream, is defined as:

```
let rec pipeline f g =
  function
     EmptyStream -> EmptyStream
  |  Stream(x, y) -> Stream((g(f(x)), (pipeline f g y));;
```

Listing 1.4:

It basically represents a parallel composition: the parallel semantics of the pipeline states that the two different functions `f` and `g` will be computed in parallel on x_{i+1} and $f(x_i)$ for any i.

Another stream-parallel skeleton is the *farm*, modelled after the following higher order function:

```
let rec farm f n:int=
  function
       EmptyStream -> EmptyStream
    |  Stream(x,y) -> Stream((f x), (farm f y));;
```

Listing 1.5:

representing the case in which n replicated parallel agents apply the same function over at most n different elements of the stream.

Data parallel skeletons exploit parallelism thanks to the fact that, it may be the case that different portions of compound data structures (like arrays) can be computed independently. A classical data parallel computation is the map, defined as follows:

[7] Using Ocaml notation

```
let map f x =
  let len = Array.length x in
  let res = Array.create(f x.(0)) in
    for i = 0 to len -1 do
      res.(i) <- (f x.(i))
    done;;
  res;;
```

Listing 1.6:

This map function receives an array `x` in input and returns an array `res` such that `res.(i)` is equal to `x.(i)` for every *i*. The parallel semantics is such that each element of the `res` vector is computed in parallel.

Other known data parallel skeletons are *reduce* and *parallel prefix*, for whose description - and further informations about basic concepts and implementation of algorithmic skeletons - we defer the reader to [Dan14].

From the implementation point of view, algorithmic skeletons can be provided using either *templates* or *macro data flow* graphs. In the template approach, we have a set of *concurrent activities* representing the nodes of a graph, while arcs represent communication or data movements between the nodes. In the macro data flow approach, instead, skeletons are compiled in terms of graphs, whose instances (one per input data set) are evaluated using a distributed interpreter.

Different skeletons are associated with performance models, whose presence is crucial for achieving good performances, as they i) allow to predict performances before developing the application; ii) to evaluate them once deployed and iii) to optimize the computation.

1.3.2 Behavioural skeletons

A behavioural skeleton [ACD+08] is the result of a co-design of a parallelism exploitation pattern together with an *autonomic manager*, whose concerns are granting non-functional features related to the parallelism exploitation pattern. Since the autonomic manager is aware of the structure of the parallel computation, several adaptation schemas can be implemented. Such schemas may respond to variation on the system, related either to *endogenous* or *exogenous* causes.

A behavioural skeleton responds to concerns like energy consumption, resource under/over utilization by implementing an *adaptation schema*, that might change the structure of the computation. It is implemented using the classic *Monitor, Analyze, Plan and Execute* (MAPE) loop, to, respectively

- collect information through sensors/probes (*monitor phase*);
- take decisions regarding whether adaptation mechanisms should be adopted are taken (*analyse phase*) exploiting some kind of abstract performance model;
- devise strategies to implement decisions (*plan phase*);

- implement strategy (*execute phase*)

The above loop is executed continuously to respond to changes in the computation or in the system onto which it runs. Obviously, energy consumption and power management concerns belong to the realm of problems targeted by behavioural skeletons.

1.3.3 Algorithmic skeleton frameworks in heterogeneous systems

With the advent of heterogeneous architecture, it is even more crucial to have implementations of algorithmic skeleton-based frameworks allowing to run on different kinds of architecture. Skeleton libraries, together with tools like CUDA, may allow to deliver good performances in heterogeneous architecture hiding the complexity of targeting different systems. Here, we give a quick primer on the existing (to the best of our knowledge) frameworks supporting structured parallel programming on heterogeneous CPU/GPU architectures.

SkePu

SkePu [8] [DLK13] [EK10] is a skeleton programming framework, allowing to execute parallel application on both multicore and heterogeneous systems (comprising one or more GPUs). It is provided as a library of C++ templates. The latest version as of today (1.2) provides (among others) map, reduce, mapreduce, scan and farm skeletons, most of which support hybrid execution on CPU and GPU.

Muesli

Muesli [9] [EK12] [MPAM13] is a template based skeleton programming framework, based on OpenMP and CUDA. It can run on clusters as well as in multi-core and in heterogeneous settings.

FastFlow

FastFlow [10] [ACD$^+$13] [GGGV12] [ADKT11] is a high-level, pattern based parallel programming environment, specially targeting stream parallelism and allowing development in heterogeneous platforms. It is developed in collaboration by the parallel computing groups of the Departments of Computer Science of the University of Pisa and University of Turin. The framework, that has been used to develop part of the code used in this work, is organized in layers:

- the lowest level layer provides lock-free synchronization mechanisms;

[8] `http://www.ida.liu.se/~chrke55/skepu/`

[9] `http://www.wi1.uni-muenster.de/pi/forschung/Skeletons/`

[10] `http://calvados.di.unipi.it/`

- the middle layer provides communication mechanisms, including single producer single consumer queues (SPSC) and multiple producer (MPSC) single consumer queues;
- the top layer provides implementation in term of concurrent activity graphs of streaming parallel patterns.

FastFlow is particularly efficient in case of fine-grained parallelism, mainly thanks to the efficient queues implemented in the middle layer. The queues are lock-free and wait-free, and their memory footprint is quite reduced. Communication is carried on, in shared memory environments, within nanoseconds. According to the website, a SPSC queue with asynchrony degree k requires only $144 + 64 \times k$ bytes, hence being very memory efficient.

The main constructs made available by the environment are *farm*, *pipeline* and *farm with feedback*. Through these basic constructs it is possible to implement also data-parallel computations without too much effort.

1.4 Energy consumption and metrics

Energy consumption (measured in Joules) is the fundamental metric that we seek to minimize, as it relates to expenditure in data center and to battery life in mobile environment. From a low level perspective, the energy consumption of a program is the sum of the energy consumed by all the operations performed. As it happens with the completion time, the energetic cost of performing an operation depends on [CA12]:

- the complexity of the operation and of the processing unit on which it is executed;
- the level of the memory hierarchy accessed;
- the cost of communication eventually involved in the performed operations.

In general, different operations will activate different resources and cause different switches in values in the CMOS circuits exploited, thus the instantaneous power $P(t)$ (expressed in *Watt*, Joules/s) will change along the time line. Given a certain computational environment and a computation, the total consumed energy will be:

$$E = \int_{t_{start}}^{t_{end}} P(t)dt \tag{1.1}$$

where t_{start} represents the starting time of the computation and t_{end} the time at which it terminates. A chip power consumption changes depending on several factors: the computation being executed as well as design choices of the manufacturer, as explained in the following section.

1.4.1 Power consumption in CMOS technology

In current chips, the consumed power $P(t)$, is classically divided (as in [KM08]) in 2 main categories: *static power*, arising from the mere fact that the die is powered, and *dynamic power*, which depends on the computation being executed.

Dynamic Power

Dynamic power is the main source of energy consumption; it can be calculated from the following equation:

$$P_{dynamic}(t) = CV^2Af \tag{1.2}$$

where:

- **C**, the *aggregate load capacitance*, mainly depends by the wire lengths used by on-chip structures. This value can be reduced in several manners, as an example by building smaller memories or multiple, less powerful processors rather than a larger one on the same chip.

- **V**, the *supply voltage* is the most important component used to minimize energy consumption. Architects managed to make this figure drop steadily in the past years, because of its quadratic impact on overall power. This value can be reduced further when the frequency is lowered [KGC99]. In fact, this is the only manner to reduce V, as the propagation delay of a gate is inversely proportional to V. This means that V cannot be decreased if f isn't, as it would lead to incorrect results.

- **A**, the *activity factor* is a fraction $\in [0, 1]$ referring to how often wires switch from 0 to 1 or from 1 to 0 in a computational environment. This activity factor relates both to circuit design and to the program currently being executed. In most wires, we often observe that $A < 1$, as they do not continuously switch. A can be further reduced (for idle hardware units) by using *clock gating* - a technique that places the clock signal in AND with a control signal [WPW00].

- **f**, the *clock frequency*, has a linear impact on power. However, in principle, decreasing only the frequency bears to *increases* in energy consumption, as every operation will obviously require more time to reach completion. Nonetheless this is an important leverage as it allows to reduce also V, impacting cubically on the instantaneous power consumed by a CMOS chip.

Dynamic power is the most studied component in literature, as it largely exceeds static power consumption [KM08] [BAM98] [JM01] [KG97] [SD95].

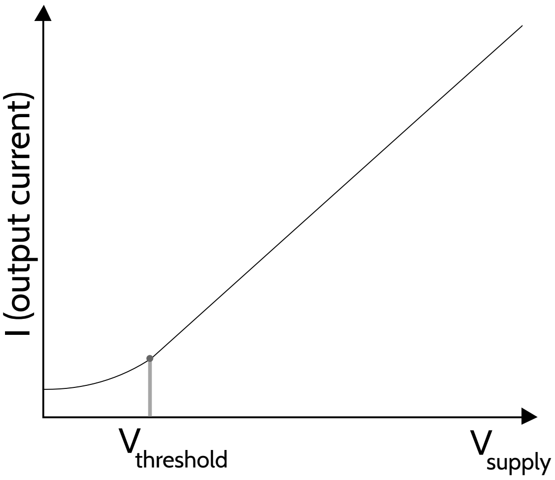


Figure 1.10: Example of an I-V curve for a semiconductor diode.

Static Power

Static power arises even when no active switching is present. In older technology, it was a major problem, which was partially solved in early CMOS chips. However, it is nowadays coming back in form of *leakage power* consumption. It has been estimated [KM08] that it currently represents roughly the 20-40% of power dissipation in modern chip designs, a figure that is expected to increase [BS00] as technology feature scales down.

CMOS static power arises mainly because of *leakage currents* (I_{leak}):

$$P_{static} = VI_{leak} \tag{1.3}$$

Leakage current is mainly due to the analog nature of transistors; in fact, even below the voltage threshold V_T above which we consider the transistor to be in *on* state, leakage currents still flow, as it can be seen in 1.10. The current flowing through a transistor even though it is off is called *sub-treshold* leakage; however, there are 5 more types of leakage consumption, for whose description we defer the interested reader to [KM08].

In CMOS, the supply voltage V_{dd} of the chip is usually scaled down so that the dynamic power consumption decreases. Since scaling down only V_{dd} would lead to an increase in transistor's delay, as it can be seen in 1.4, the only way to maintain acceptable delays is diminishing V_T accordingly.

$$Delay \propto \frac{V_{dd}}{(V_{dd} - V_T)^{\alpha}} \tag{1.4}$$

where α is a technology dependent factor: for current technologies it varies between 1.2 and 1.6 [S^{+}90]. However, since sub-threshold leakage increases exponentially with lower threshold voltage, it is becoming an always growing concern for circuit designers.

Other sources of power consumption

There are other sources of power dissipation (e.g. *glitching power*) in CMOS technology. Using a common approach, we will neglect them and consider the total power consumption as the sum of dynamic and static power, as shown in [KA11]

$$P(t) = P_{dynamic}(t) + P_{static} \tag{1.5}$$

1.4.2 Energy consumption of parallel applications

In the following, we will focus mainly on the energetic cost of the *computing* part and related communications, thus neglecting other (possibly important) components that sum up to the energetic bill, like disk accesses. Moreover, we will describe primarily the costs incurred while operating on a parallel, but not distributed environment. The case of distributed systems (e.g. data centers), are more difficult to model not just because of the surrounding infrastructure, but also because such structures can seldom be dedicated to single computations. In such scenarios, most of the energy optimization relies on *work consolidation* in order to maximize performance per Watt [SKZ].

Modern days architectures are made of several computing devices, possibly with different capabilities. Even on low-end computers, it is common to find multiple CPUs and possibly a GPU. Such devices can be exploited concurrently to carry on a single computation. We generalize the power cost of all these different devices as a set of resources, and we model the overall power consumed by a computing infrastructure as:

$$P_{total}(t) = P_{infr} + \sum_{i \in resources} P_i(t) \tag{1.6}$$

where P_{infr} represents the amount of power dissipated by everything constantly turned on during all the computation, and consequently which is not distinguishable from the cost of turning on the computing infrastructure, as shown in [MCC15].

As energy consumption of an application depends both on the resources used to execute it and the completion time, we face a delicate trade-off from the energetic standpoint. In the parallel application world, in particular, there exists a delicate balance between the amount of used resources and completion time [CGM+10]: executing the computation on n CPUs will bear an increase in dynamic power (and thus energy) which will be proportional to n. However, in the ideal case it will also bring execution time to $\frac{1}{n}$-th of the sequential time, with consequent potential energetic advantages, also due to the amortization of the static component of power. Hence, potential energetic advantages in parallelizing an application depend on the *speed-up* of the application taken into consideration, which is defined as:

$$sp(n) = \frac{T_{seq}}{T(n)} \qquad \in (0, n] \tag{1.7}$$

where $T(n)$ is the time required to execute the application on n processors and T_{seq} is the time of the sequential execution. In the ideal case, we will have $sp(n) = n$.

However, speed-up has a upper bound given by the well-known Amdahl's Law [Amd67]:

$$sp(n) \leq \frac{T_{seq}}{F \times T_{seq} + (1 - F) \times T_{seq}} \tag{1.8}$$

where F is the *fraction* of work which is inherently sequential. This means that the parallel execution time $T(n)$ of the application on n processors is bounded as follows:

$$T(n) \geq F \times T_{seq} + (1 - F) \times \frac{T_{seq}}{n}$$

If by $E(n)$ we denote the energy spent in the system during the execution of the parallel application with n processing units belonging to $used\_res$, the set of used computational resources, we will have

$$E(n) = P_{infr} \times T(n) + \sum_{i \in used\_res} \left(\int_{t_{start}}^{t_{start}+T(n)} P_i(t)dt \right) + T(n) \times \sum_{i \in res \setminus used\_res} P^i_{static} \tag{1.9}$$

Notice that equation 1.9, is still valid in machines where unused processors can be turned off, as we can consider $P^i_{static} = 0$ for every $i \in turned\_off\_processors$.

Assuming that i) the average power consumed by a processing unit is not dependant on the starting time t_{start}; ii) that all used processing units require on average the same amount of power $\overline{P}$ and that iii) the static power P_{static} is the same for all $N = |res|$ available computing resources, we can rewrite equation 1.9 as follows:

$$\begin{aligned} E(n) &= P_{infr} \times T(n) + n \times \overline{P} \times T(n) + (N - n) \times P_{static} \times T(n) \\ &= P_{infr} \times \frac{T_{seq}}{sp(n)} + n \times \overline{P} \times \frac{T_{seq}}{sp(n)} + (N - n) \times P_{static} \times \frac{T_{seq}}{sp(n)} \end{aligned} \tag{1.10}$$

Hence, the minimum energy for a fixed n and a given computation is spent when the speed up is n; a lower bound for the energy spent in a parallel application is given in Equation 1.11:

$$\begin{aligned} E(n) &= \frac{T_{seq}}{sp(n)} \left(P_{infr} + n \times \overline{P} + (N - n) \times P_{static} \right) \\ &\geq \left[F \times T_{seq} + (1 - F) \times \frac{T_{seq}}{n} \right] \left(P_{infr} + n \times \overline{P} + (N - n) \times P_{static} \right) \end{aligned} \tag{1.11}$$

showing how important is achieving the maximum speedup (or, alternatively, efficiency) for having good energetic performances. This maps intuitively to the fact that low efficiency (formally defined as $\epsilon(n) = \frac{sp(n)}{n}$) basically means "resource underutilization", and thus energy waste.

Another important issue to be taken into consideration for energy efficiency is the impact of *communications*. Memory hierarchies, ubiquitous in current computing environments, often including remote, over the internet communication, do constitute an important burden from the energetic standpoint. The expected energy cost of data movement is directly proportional to the distance of communication [CA12]: single chip communication is order of magnitudes less costly than off-chip, external memory and network communication, as it can be seen in Table 1.1.

Operation	**Energy Cost (J)**
Floating Point operation	$10.6 \times 10^{-12} J$
Register File operands supply [11]	$5.5 \times 10^{-12} J$
L1 Instruction cache access	$3.6 \times 10^{-12} J$
L1 Data cache access	$3.6 \times 10^{-12} J$
L2 cache access	$18.5 \times 10^{-12} J$
L3 cache access	$39.5 \times 10^{-12} J$
Memory access	$168.5 \times 10^{-12} J$
Network	$311.5 \times 10^{-12} J$

Table 1.1: Energetic cost of different operations, adapted from [KBB$^+$08].

Let us now consider the case of a common, universally spread type of *heterogeneous systems*.

A system is called *heterogeneous* when it integrates two or more kinds of processors, with different performance and power. Heterogeneous systems may be implemented on the same chip or through different dies connected through some kind of interconnection network (usually a bus). The availability of different types of devices provides further space for energy optimization: since they will have different power/performance trade-offs, scheduling a computation in the adequate amount on the adequate set of processing units is crucial to minimize energy. In the following, we will consider a relatively simple heterogeneous architecture, with a set of n cores and 1 Graphic Processing Unit (GPU).

Despite consuming an elevated amount of power, GPUs are often used for their ability, in highly parallel and computing intensive scenarios, to minimize energy consumption because of their elevated *performance/watt* ratio.

Let us consider a computation, of which a fraction $g \in [0, 1]$ is offloaded to the GPU, using a total of m processing units, while $1 - g$ is processed in parallel on n of the N available cores. The total time to process the data will be the maximum between the data transfer time plus the GPU computation time and the time required to execute on n CPUs. Assuming the communication time T_{com} is dependent on the fraction of work offloaded to the GPU, we will have:

$$T_{CPU-GPU}(n, m, g) = max\{T_{CPU}(n, m, 1 - g), T_{GPU}(m, g) + 2T_{send}(g)\} \quad (1.12)$$

and, extending eq. 1.9 the total energy spent will be:

$$
\begin{aligned}
E(n,m,g) =& P_{infr} \times T_{CPU-GPU}(n,m,g) + \sum_{i=1}^{N} \left(\int_{t_{start}}^{t_{start}+T_{CPU-GPU}(n,m,g)} P^{i}_{CPU}(t)dt \right) \\
&+ T_{CPU-GPU}(n,m,g) \int_{t_{start}}^{t_{start}+T_{CPU-GPU}(n,m,g)} P_{GPU}(t)dt
\end{aligned}
\tag{1.13}
$$

Applying again the assumptions used to achieve eq. 1.10 from eq. 1.9, considering that the task of moving data from the main memory to the GPU on-chip resident memory is offloaded to a dedicated communication co-processor[12] and assuming that i) the GPU communication processor consumes on average a fixed amount of power $\overline{P}_{GPU\_coprocessor}$ during transfers and that ii) the GPU consumes on average a constant power during the computation, we can obtain the following energy consumption terms, which together sum up to $E(n,m,g)$ as of 1.13:

$$
\begin{aligned}
E_{infr}(n,m,g) &= P_{infr} \times T_{CPU-GPU}(n,m,g) \\
E_{CPU\_active}(n,g) &= \overline{P}_{CPU} \times n \times T_{CPU}(n,1-g) \\
E_{GPU\_active}(m,g) &= \overline{P}_{GPU} \times T_{GPU}(m,g) \\
E_{GPU\_transfer}(g) &= \overline{P}_{GPU\_coprocessor} \times 2T_{com}(g) \\
E_{CPU\_idle}(n,m,g) &= U(2T_{com}(g) + T_{GPU}(m,g) - T_{CPU}(n,1-g)) \\
&\quad \times (2T_{com}(g) + T_{GPU}(m,g) - T_{CPU}(n,1-g)) \times N \times P^{CPU}_{static} \\
&\quad + (N-n) \times P^{CPU}_{static} \times T_{CPU}(n,1-g) \\
E_{GPU\_idle}(n,m,g) &= U(T_{CPU}(n,1-g) - 2T_{com}(g) - T_{GPU}(m,g)) \\
&\quad \times (T_{CPU}(n,1-g) - 2T_{com}(g) - T_{GPU}(m,g)) P^{GPU}_{static}
\end{aligned}
\tag{1.14}
$$

In 1.14, we used the following terminology:

- E_{infr} is the cost of maintaining active the infrastructure, and depends only on the completion time.
- E_{CPU_active} and E_{GPU_active} are the energy costs of performing the computation respectively on the CPUs and on the GPU.
- $E_{GPU\_transfer}$ is the cost of data movement from main memory to the GPU dedicated memory.
- E_{CPU_idle} is the cost of maintaining $(N-n)$ CPUs idle plus the cost of a possible unbalance between the set of CPUs and GPU (paid in case GPU

[12] Hence GPUs communications and computations can be performed asynchronously

computation lasts more than CPUs one). To define it we used the *unitary step function* $U(x)$, whose value is 1 for positive values.

- E_{GPU_idle} accounts for the energetic consumption paid when the GPU is inactive.

E_{CPU_idle} and E_{GPU_idle} energy costs are due to unbalance between the computations executed, respectively, on the set of cores and on the GPU. It can be minimized by calculating a proper g such that the completion time is almost the same. We understand immediately that for $g = 0$ and $g = 1$ a very high amount of energy will be dissipated without performing any operation, hence suggesting that at least a part of the computation (depending on its features and the hardware's one) should be carried on on each component. A better characterization of $T_{CPU-GPU}(n, g)$ and a method to calculate g for some classes of computations can be found in [SDK13], to which we defer the interested reader.

Notice that the results involving the Amdahl's law still hold in this kind of system; in fact, given a computation with a serial fraction of F, we will have

$$T_{CPU}(n, g) \geq F \times T_{seq} + \frac{(1 - F) \times (1 - g) \times T_{seq}}{n}$$

and

$$T_{GPU}(m, g) \geq \frac{g \times (1 - F) \times T_{GPU}(1)}{m}$$

Obviously, the serial fraction impacts also on energy, as, unless it can be completely overlapped by communication with GPU, it would proportionally increment E_{GPU_idle}, regardless of g, m and n.

From an high-level perspective, with a given heterogeneous system, minimizing energy consumption means:

1. Reducing the completion time (and thus the performed operations), by carefully optimizing the computation so that the processor's ISA is used at its best [KGC99]. Obviously, the choice of the right algorithm is of the uttermost importance.

2. Dividing the computation carefully between devices with different capabilities, minimizing completion time and/or optimizing energy.

3. Maximizing the speed-up of a parallel application, amortizing better the fixed cost of the computing infrastructure and using the resources efficiently.

4. Increasing data locality, as accessing off-chip memory has a huge cost (see Table 1.1); even worst, remote communications require activating antennas and/or using a network infrastructure, increasing latency and energy consequently.

5. Reducing the power needed to carry on a computation, by scaling frequency and, consequently, allowing the supply voltage to be decreased.

Reducing the completion time relates to the realm of optimizations and of compilers. Of course carefully written assembly, as shown in [KGC99], may brought better performances; however, not all operations have the same energetic footprint, and thus more sophisticated transformations should be defined in the compilers, allowing to substitute operations with less power hungry, equivalent ones.

As shown previously in inequality 1.11, minimizing the sequential fraction of an application, and trying to achieve the best possible speed-up (hence also minimizing overheads) also bears significant advantages in terms of energy, as overall the machine is used for less time.

The third point, related to communication, imposes yet another trade-off, as sometimes it could be convenient to pay an additional energetic cost for communication (i.e.: to offload a part of the computation to a different processing unit) to decrease completion time or to use a more energy-aware device, thus possibly obtaining an energetic gain.

Most of the current methods for reducing energy consumption in modern architectures rely on the latest of the above points. The cubic impact of *frequency scaling* and *voltage scaling* is an enormously appealing leverage to achieve energy efficiency: while decreasing the frequency bears (normally) a linear disadvantage in terms of completion time, the instantaneous power of the computing devices used decreases more than linearly. However, specially in the HPC world where performance in term of time still has an higher consideration with respect to energy, frequency scaling is mainly used in situations where resources are not used at their maximum efficiency. Several *DVFS* (*Dynamc Voltage and Frequency Scaling* techniques exist, exploiting *slack time* in order to reduce power consumption without degrading performance proportionally. We will discuss further DVFS techniques in Section 2.1.

1.4.3 Metrics for combined energy and performance evaluation

Which metric should be used to evaluate performance and power trade off depends on the area of interest and the type of platform object of the study. In environments like mobile platform (e.g. smartphones, laptops), pure **energy**, (expressed in Joules) is considered to be the most important metric, as it relates to battery lifetime and, consequently, to the device availability: in this case low energy expenditure is a functional requirement.

More interesting are metrics combining performance and energy. The most simple case is *energy-per-instruction* (EPI). It is a measure of the amount of energy consumed by a micro-architecture for every instruction executed. It is expressed in $Joules/Instruction$. It is ideal to assess power-efficiency in environment where *throughput performance* [GA06] is the main target. When operating within a power budget, EPI must be as low as possible in order to deliver high performances.

Moving to an higher level of abstraction, one of the most commonly used metrics is *energy-delay product* (EDP) [GH96]. As the name itself states, it is the product of energy and execution time, and is thus expressed in $Joule \times seconds$, to combine the requirements of low energy and fast runtime. This metric can be used to compare different architectures given an instruction mix or to evaluate different (possibly parallel) applications. It improves (*is lower*) for approaches holding energy constant but with a shorter runtime or maintaining the same completion time while reducing energy usage.

EDP metrics has a catch: when comparing different systems, it fails in case *voltage scaling* is allowed. Consider the case of two different systems, A and B such that the energy consumed by A, E_A is twice the energy consumed by B for carrying on a computation with completion time $T_A = \frac{T_B}{2}$. If A's supply voltage can drop by half, we will have $E'_A = \frac{E_A}{4} = \frac{E_B}{2}$ and $T'_A = 2T_A = T_B$; this means that A is better, but EDP fails to capture this condition. Moreover, since EDP weights equally energy and power, it is sometimes replaced with *energy-delay-squared* (ET^2) [MNP02]; in this case the delay has a square impact, accounting for a more elevated concern about performance.

1.4.4 Measuring energy consumption

As with other kinds of monitoring, probes introduce differences in the behaviour of the computation that is monitored. This is a well-known problem, often named *intrusion problem* [Dan14] inserting probes to measure any given quantity requires extra system or library calls, that spend time (and hence energy), require memory and may in general affect the results of the measurement itself.

In the parallel applications world, the intrusion problem is even bigger, due to the fact that the delay introduced by probes may make some bottlenecks disappear.

Tools for energy consumption monitoring can be categorized in three classes [CAAB15b]:

I. **External devices monitoring:** with this approach, a monitoring device is placed *outside* of the monitored node. Measurements do not interfere with the experiment, but at the cost of less precision

II. **Intranode monitoring:** in this case monitoring is carried on with customized tools on restricted platforms, as it requires to develop different tools for different hardware. It can be carried on by using hardware performance counters.

III. **Other approaches to monitoring:** this class of tools focuses on providing API to access information provided by the hardware. The information is elaborated starting from hardware performance counters, as well as from application of analytical applied to workload metrics.

EML: Energy Measurement Library

EML [13], acronym of Energy Measurement Library[CAB13], has been the elected tool used to carry on the experiments described later in this work. Developed by the HPC Group of Universidad de La Laguna (Tenerife), it is a portable library, allowing to abstract the user from the tools used to measure the energy. Currently, it allows to perform measurements on the following devices:

- Intel CPUs, from Sandy Bridge on, exploiting Intel's Running Average Power Limit interface;
- Recent NVIDIA Tesla and Quadro GPUs, using NVIDIA Management Library (NVML);
- Intel Xeon Phi Many Integrated Core co-processors;
- Schleifenbauer Power Distribution Units, through a socket API; it can be used also in distributed environments.

It is provided under a GPLv2 licence, and is completely open source. The development of this library is still in process, hence it is not yet mature and installing it requires some notable efforts. It requires to have read privileges on some system files, namely `msr` files under `/dev/cpu/*/` on Linux Systems. Being the granularity of the measurements almost completely user defined (with some limitations due to the amount of samples needed to read significant values), it allows to *auto-tune* computations depending on their energetic behaviour. Energy consumption and time metrics are provided to the user using a unified interface, regardless of the monitored devices or of the way in which the samples are collected.

Measurements can be collected in a *instant* fashion, that returns the instantaneous energy consumption as collected by the hardware, or in a *interval* manner. This case, which was the kind of measurements exploited in this work, is useful in order to retrieve energy aggregated metrics in a way that is totally similar to time aggregated metrics. Hardware measurement of energy consumption can be provided either instantaneously or depending on the interval. However, the library abstracts from the way values are read and seamless derives the kind of values required from the user.

Further, it provides a standard manner to read energy consumption, unifying different approaches (notably, RAPL provides information about consumed energy, while NVML only reports instant power consumption of the monitored board). EML performs device discovery at runtime, and the user can perform measurements on all or on a part of them using appropriate calls. The library, according to [CAAB15b], works as follows:

1. at first, a discovery phase is executed, using `emlInit()`, in which available devices are individuated, the memory needed to measure is allocated and informations about the environment are made available to the user;

[13] `http://hpc-ull.github.io/eml/`

2. measurements are started using a call (`emlStart()`) to be placed before the section of code to be monitored;

3. the sampling process is terminated by a call to `emlStop()`;

4. data can be collected in a instantaneous manner (getting a dump of all samples in JSON format) as well as in aggregated form using appropriate calls, triggering an elaboration of the samples gathered by the library;

5. the used resources are freed through a call to `emlShutdown()`.

According to [CAAB15a], the library has a small overhead, accounting for about *2-4%* - depending on the data size - when monitoring with NVML and *1.54%* when monitoring with RAPL.

Chapter 2

Survey

In this chapter we will present some methods used in both CPU and GPU architectures to cope with the trade-offs between parallelism, energy and performance achievements. We will present the widely adopted technique of *Dynamic Frequency and Voltage Scaling* in 2.1. Even though this technique has not been used in this work, its wide usage in energy-aware system mandates at least to present a brief introduction to the technique and the main results achieved using it. In Section 2.2 we will present some methods used to reduce energy consumption in heterogeneous systems. Finally, we will move to briefly present some energy and power models.

2.1 Dynamic Frequency and Voltage Scaling

Dynamic Frequency and Voltage Scaling (DVFS) is a well known technique allowing to spare energy in computing systems. Starting from Equation 1.2, we see that the dynamic power P consumed in a CMOS device is $P \propto V^2 f$. Even with a small reduction in V, we gain the square in terms of power. Reducing the voltage requires the frequency to be scaled accordingly to ensure the correctness of the calculation, so performances are degraded linearly. We have a set - either discrete or continuous - of pairs (V, f) that can be changed at different granularity, affecting power and time. While the best results can be achieved when the voltage can be selected arbitrarily, interestingly, [HQ03] shows that using 3 or 4 different DVFS levels is enough to reach close-to-optimal energy consumption.

The main idea behind DVFS is that we will have a slowdown in the process execution, but with a disproportional impact on the required power, thus saving energy. This can be done with the highest advantage in presence of *slack time*. Slack time can be either due to bottlenecks (e.g. memory accesses), early termination of a task due within a deadline or underutilization of the computing resources (e.g. a system without jobs running on it).

We show the principle in Figure 2.1, where we show the impact of different configurations of (V, f) on time and energy. The red square shows the energy when the computation is executing using a voltage V and a frequency f. Halving the frequency will halve the power, but double the completion time: the energy of this configuration is the orange rectangle, whose area is the same of the red square. Finally, the purple square shows the real benefits of DVFS: in case the voltage is halved (and this is allowed only if the frequency is reduced), the power is proportional to the fourth of the previous one and the energy consumption is way smaller, as shown by the purple rectangle.

Dynamic voltage and frequency scaling grants energy saving by greatly reducing dynamic power consumption; the advantages of the technique terminate, as reported in [ZBSF04], when the voltage is diminished below a certain threshold. In this case the energy dissipated by leakage power *increases*, hence reducing the awarded benefit. Another limitation of this technique [HVC+06] is how often the (V, f) couple can be changed: usually, switching to a different frequency requires several microseconds, as the digital circuits are stopped during the transition.

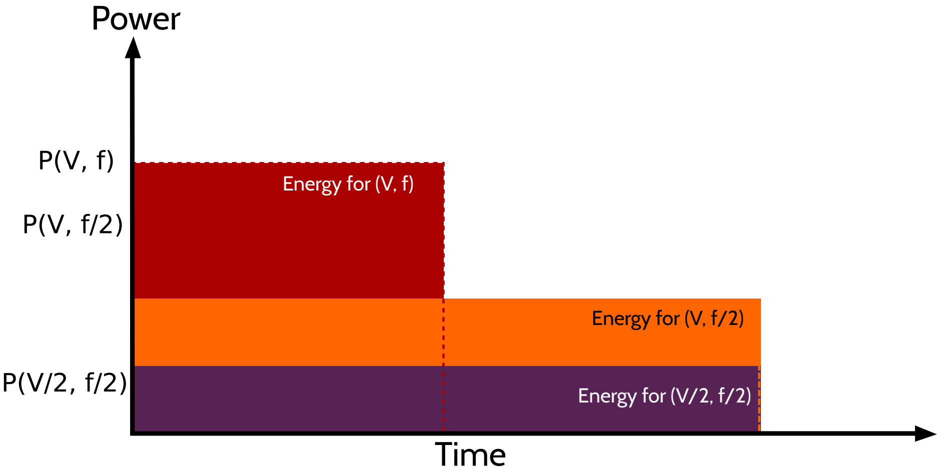


Figure 2.1: Energy consumption and power consumption using different voltages and frequencies.

The DVFS technique can be applied at several levels [KM08]:

- *Hardware level*, where DVFS is used at hardware level, using low utilization factors of different components to reduce their consumption;
- *System level*, where the "idleness" of the whole system is used to drive DVFS decisions;
- *Program level*, where single threaded programs could exploit instructions with long latencies (e.g. memory accesses) to reduce the frequency until their completion and parallel program can be tuned depending on the needs of every node of the concurrent activity graph.

Another important way to distinguish different approaches to DVFS is the moment in which the decision about scaling is taken: there exists static, compiler based approaches to this problem as well as reactive ones, based on the instantaneous behaviour of the system/program/component taken into consideration.

In the last years, gains in energy consumption due to DVFS have been steadily decreasing: as explained in [LSH10], the progressive decrease in transistor feature size and voltage required by today CPUs, severely impairs the potential to save energy using DVFS.

2.1.1 Hardware level DVFS

In Razor [EKD$^+$03], the fact that the so-called *critical voltage* is an upper estimation of the one effectively required for correctness, is exploited to reduce power consumption by lowering voltage below this threshold. In this device, the voltage is lowered until timing faults start arising; this approach has been tested on modified ARM cores, showing a gain of about 64% in power demand.

While most of the conventional multiprocessors run on a single clock domain, in *Globally Asynchronous Locally Synchronous* systems (GALS) [MVF00], contain several independent synchronous blocks, each one operating with its local clock and communicating asynchronously with each other. A particular design in GALS is the one called Multiple Clock Dynamic Voltage, in which every clock domain is supplied with a different voltage. The voltage can be changed [IM02] in a dynamic or application-dependant manner: since different application will require to use different resources of the processor, an intelligent selection of the frequency of different parts of the chip can give significant advantages without increasing the delay. This can be done easily in such systems as different parts can be controlled independently. As reported in [IM02], significant energy gains can be achieved in such systems, since the increased delay (due to the overhead of synchronizing different clock domains for communication) is alleviated by the greater reduction in terms of power.

In [SMB$^+$02], the authors present an architecture in which multiple clock domains are individuated by exploiting boundaries already well defined in terms of queues or between units with relatively small inter-function communication. In the proposed architecture, visible in Figure 2.2 *four subdomains* are individuated: the first frequency domain (F1) comprises the instruction cache, branch prediction, rename and dispatch; F2 and F3 domains separate execution units of different types (integer and floating point respectively); F4 comprises the load and store unit, L2 cache and L1 data cache. The memory is in a separate, external, frequency domain, named F0.

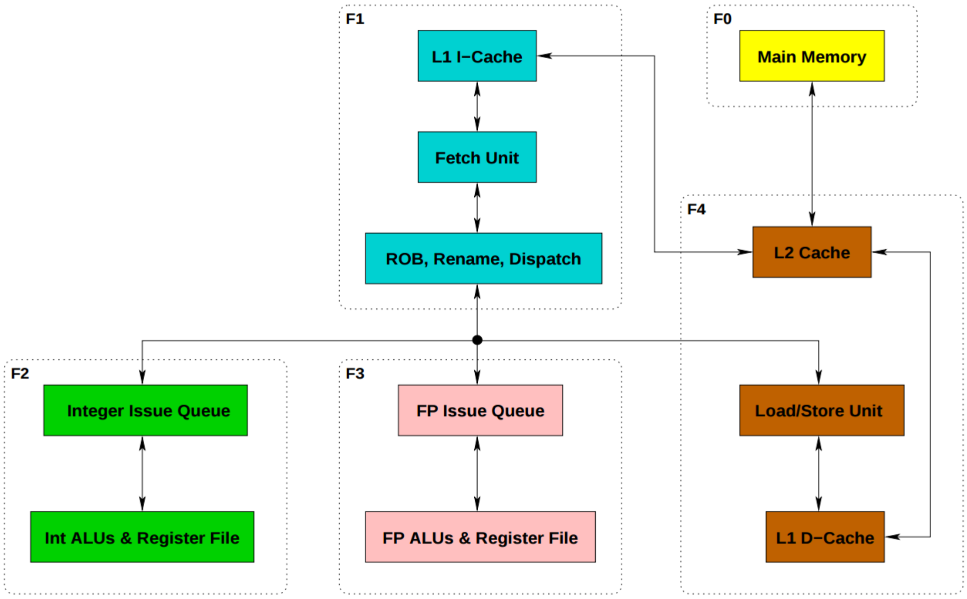


Figure 2.2: A multiple clock domain processor block diagram, drawn from [SMB+02]

As stated by the authors, choosing queues as the inter-domain synchronization points has the advantage of hiding the synchronization cost when the queue is not saturated or empty. Without voltage or frequency scaling the synchronization mechanism causes a performance degradation of less than 4% in average, and an increase of 1.5% in energy usage, using a baseline configuration with all domains running at the same frequency. However, by selecting a voltage scaling causing a 5% degradation in terms of performance in all domains but F1, the authors show that it is possible to achieve a 27% gain in terms of energy, while EDP improves of 20% with respect to baseline.

In [SAD+02], the authors propose an adaptive, on-line algorithm named *Attack-Decay* for the aforementioned architecture, achieving significant improvements from the point of view of energy and EDP. The algorithm uses informations about the queue utilization factor to scale frequency aggressively in a domain (*attack phase*) or to diminish it slowly (*decay phase*), in case no significant change in the utilization factor has been detected. The reported energy per instruction saving is 19.0%, while EDP gains the 16.7% with respect to the baseline processor.

[Mar00] proposes a microarchitecture-driven dynamic voltage scaling mechanism. The typical computation exposes different phases: computational intensive parts are followed by stalls while data is awaited. The author proposes to use different voltages for the different phases, detected directly at the hardware level. Simulations show a potential energy saving of 20%.

2.1.2 System level DVFS

System level DVFS starts from the consideration that *idle time* basically represents an energy waste: hence it should be eliminated as much as possible (from a system perspective), in order to maximize the performance per Watt. DVFS can be applied at system level provided that the operating system has some mechanisms to individuate idle time and to change the frequency accordingly. This mechanism can be applied, depending on the hardware, by exploiting the operating system's knowledge about the tasks currently executing: while the processor cannot detect whether an instruction currently in execution belongs to a compute intensive task and an application is not aware of the whole system utilization, the operating system has a global view of how resources are used. Consider the case in which a computation terminates before its scheduling quantum is elapsed: in this case, during all the remaining time, the processor would consume static power without elaborating, therefore wasting energy, even if in a small amount. If the voltage and frequency can be scaled on the used processor, we could achieve significant performance benefits by simply tuning (V, f) such that the computation spans over all the quantum and, overall, energy is saved. The approach, historically proposed first in [WWDS96], starts from the consideration that in case frequency scaling causes the computation to terminate within the quantum, no performance penalties are paid and energy is reduced. In the paper, they analysed several trace data taken from a number of workstations running on Unix. Three scheduling algorithms are proposed:

- **OPT** starts by elaborating the entire trace of the UNIX scheduler, stretching the computations one by one until all idle times were filled. This algorithm requires perfect future knowledge about the behaviour of the computation; also, while energy is saved, the completion times are stretched arbitrarily, without any consideration for the effective needs of the user.

- **FUTURE** needs only a small window into the future to predict the idle time, and optimizes energy consumption only within the considered window. The completion time of a computation is never stretched after the window: the longer the window, the better the energy savings.

- **PAST** is a practical implementation of FUTURE, where a fixed window into the past is analysed and, by assuming that the subsequent time period of the same length will have a similar behaviour, changes the completion time and (V, f) of the computation accordingly.

Using PAST on real environments, the authors show energy consumption reduction in the order of 50% for conservative settings and 70% scaling voltage aggressively. Approaches such as the one described before, however, fail to meet the needs in terms of delay required by interactive workloads, as described in [FRM01], where the authors deal with deadlines mainly due to users' perception. By understanding the interactions of the applications with the operating system kernel, it is possible to classify computations, hence assigning different priorities depending on their class: as an example, user interactive tasks would be assigned a deadline according to a normal user perception, choosing (V, f) accordingly. The authors report energy savings up to 75% with respect to baseline execution without their DVFS scheduler, not introducing perceivable delays for the user.

In [SKK11], a method for energy consumption reduction at the system level exploiting memory access slack instead of system level slack is proposed. The method is applied on high-end architecture, using Intel Core i7 and AMD Phenom II processors in the testing phase. Spiliopoulos et al. exploit the non-overlapping misses (last level cache misses) to individuate *steady state intervals* (time portions during which the processor operates without having to wait for data) and *miss intervals.* A governor is called periodically and uses the data collected through performance counters to select the most appropriate (V, f) pair for the next interval, according to different metrics. The decision is based on a correlation between the ratio of executed and retired instructions against the dynamic power consumption detected. EDP can be reduced of 10% in CPU-bound applications and up to 36% in memory-bound ones. One of the domain in which DVFS is more used at system level is *real-time* systems: in this case, since a deadline is already defined for each of the tasks, an appropriate operating frequency can be selected by the scheduler, thanks to the a-priori knowledge of the workload. In [YCK05], an energy-aware scheduler is proposed for real-time tasks on multiprocessor architectures. All tasks are assumed to be ready to execute at the beginning of the considered time frame; an appropriate assignment divides the task in n (number of cores) subsets. The assignment is optimal when it consumes the minimum energy between all tasks that are feasible (meaning that no task misses its deadline). The authors show that the problem is NP-hard and propose an approximated algorithm scheduling bigger (in terms of clock cycles) tasks first. Their algorithm is able to schedule tasks achieving only a 36% degradation over the optimal scheduling. For more information about real-time DVFS scheduling algorithms, we defer the interested reader to [SR12], where fourteen different RT-DVFS algorithms are compared from the performance and energetic viewpoints.

In cloud computing infrastructures, consuming cities-worth power, often jobs are scheduled using Virtual Machines (VM). A Virtual Machine is an abstraction of a computing system, which is able to interpret or execute a certain language using constructs offered by a different language. It allows to decouple from the hardware; such mechanisms are often used in cloud computing where the job can be scheduled on different machines, as they provide a desired view of the machine to the user. For this reason, approaches aiming at scheduling virtual machines are proposed in [VLWYH09]. For every scheduling loop in the cluster, the physical processing units are at first set at the lowest available (V, f). VMs are sorted by the requested operating frequency, so that those with higher operating frequency can be scheduled first to processing units operating at a matching or higher frequency with respect to the requested one. In case there's no compute node able to satisfy the VM requirement, one is elected and its frequency raised. The authors observe that this mechanism operates with the minimal power consumption at each scheduling loop, hence energy is minimized.

2.1.3 Program level DVFS

Memory and CPU are different subsystems, operating in different clock domains. When a computation is dominated by memory accesses (i.e. the processor often stalls while waiting for data), the processing unit can be slowed down without affecting performances.

In [CSP05], the authors propose an *intra-process* technique, exploiting memory access operations to understand whether the computation is memory bound, hence slowing down the processor. Authors separate instruction latencies in two categories: *on-chip* and *off-chip*. The completion time is divided depending on which of the instruction causes it (assuming no out of ordering is available) in two components: $T_{on-chip}$ and $T_{off-chip}$. While the first component is affected by CPU frequency scaling, the latter remains unchanged. Whenever $T_{off-chip} \gg T_{on-chip}$, the frequency can be scaled down without affecting the performances significantly. The decision is performed at runtime, since estimating the effective contributions to time of the two classes of instruction is quite complicated. The authors propose an estimation of the optimal frequency to be used in a time slot by using a regression model based on the number of executed instructions and the number of memory accesses. The parameters of the models are updated dynamically during the execution of the process. The achieved gain in terms of energy is 70% for memory-bound operations (with a degradation in terms of time of the 12%) and of $15-60\%$ for CPU bound computations, degrading less than 20% completion time.

Kimura [KSH$^+$06] presents a mechanism for energy saving in load unbalanced computations, represented as direct acyclic graphs. Their approach, operating at run-time, visits the graph and selects for each of the node in the concurrent activity graph an appropriate voltage and frequency. A parallel program encounters slack time when there's a synchronization between task. In case one of the task must wait for another, the shorter one's frequency can be modified so that no overhead is perceived from the outside, while power is saved. Suppose a certain task t_1 needs to synchronize with task t_2, and that the time for executing the former is greater than the latter's one ($T_1 > T_2$). The total slack time is $T_{slack} = T_1 - T_2$. While t_1 is on a critical path, t_2 is not, hence its frequency can be diminished without impairing performances. Specifically, the execution unit responsible for carrying on t_2 can be scaled down to a frequency f_2 such that $f_2 \geq f_{max} \times \frac{T_2}{T_2+T_{slack}}$. The combination with lowest voltage and frequency such that the above inequality holds is the one bearing the maximum advantages from an energetic standpoint. The slack time is calculated continuously on different nodes of the graph. When a different gear is selected, the graph is updated, until no changes are performed. The authors show that this method can diminish energy consumption of 16.8% in master-worker computations and up of 25% in tree-based parallel programs.

A method fully exploiting the slack time due to memory access is presented in [KBSSK13]. In the approach proposed by the authors, the program is written so that the access phase and the computation phase are completely decoupled, allowing to select for each one an appropriate (V, f) pair.

The program is expressed as a series of asynchronous tasks (a C/C++ function), each one divided in two fine grain phases: the former is the access one, in which data is manually prefetched from the memory to the L1 cache, while the latter performs the computation on the available data. In this way, most of the cache misses will be transformed into cache hits in the execute phase, hence reducing the slack time in the latter. The authors show that by tuning f at its minimum in the access phase and at its maximum on the execute one, they are able to decrease EDP by $25 - 30\%$, without impairing performances.

2.2 Optimizing energy consumption in heterogeneous architectures

While GPUs have some energetic advantages over CPUs, as shown in [HXF09], where the EDP of computations is compared with CPU executions, scheduling compute-intensive tasks only to GPUs will not necessarily be the most convenient approach to minimize energy.

While in homogeneous systems the optimal energy consumption can be achieved by equally distributing workload between the processing units [Li08], in CPU-GPU architecture finding the optimal energy consumption requires establishing an amount of data to be processed by each set of device and a parallelism degree.

In [WR10], a power efficient work-distribution algorithm taking into consideration all possible running levels (i.e. parallelism degree, frequency) of CPUs and GPUs available in the system is proposed. The algorithm decides the grain and schedules the computation to the different devices according to profiling informations on the computations. The energy is minimized as long as a deadline is not trespassed in terms of time.

An approach for holistic energy management of CPU-GPU computations is presented in [MLC+12] by the name of *GreenGPU*. Depending on the workload characteristics, the workload is divided between CPU and GPU in a first phase, so that the unbalance in terms of time between the two is minimal. Once this first split has been performed, the frequency of the GPU memory and cores are dynamically throttled depending on their utilization, as it is done with the frequency and voltage of the CPU. The authors claim to be able to save 20% on average.

2.3 Energy consumption models

Normally, to schedule a computation and to analyse its behaviour in terms of time and energy, we need some way to understand its energy consumption. To do so, several models have been developed to estimate power consumption. Here we provide a quick primer on some notable ones.

A first approach based on the access rate of different components is proposed in [IM03]. The authors propose an approach in which performance counters are sampled in order to estimate the power of a computation in CPU architectures.

In [CY12], the problem of how to schedule efficiently computations on heterogeneous, multi-core architectures is targeted. The authors propose an energetic model, in which through regression over four random benchmarks the energetic footprint of a computation is calculated using as parameters fifteen different hardware performance counters. The proposed model is in the form

$$\begin{aligned} Energy =& \beta_0 + \beta_1 \times Cycles + \beta_2 \times RetiredInstructions + \\ & \beta_3 \times L1DCacheAccess + \beta_4 \times L2CacheAccess \end{aligned}$$

and the parameters are achieved by using ordinary least squares. The authors show how the model is reliable in two different CPU architectures, achieving an average error below 3%. The authors propose a static analysis and use the developed model to schedule optimally a computation depending on the required energy.

In [HK10], an integrated power and performance model for GPU architectures is proposed. The authors address the problem by modelling the GPU power consumption with a low-level model, in which the dynamic power is divided between the memory and the streaming multiprocessor contribution. The two terms are then calculated using the access rate to different units within the architecture and for a certain computation. The achieved error in estimating the power is 9.18% on average.

An energy model based on micro-benchmarking is proposed in [PLS10]. Through careful monitoring of studied applications, the authors calculate the energy footprint of different computations and different types of memory accesses. The evaluation is then performed using graphical applications, showing an average error around 11% on average.

Chapter 3

Methodology

The aim of this chapter is to briefly present the motivation and the problem that we wish to solve. An informal setting will be provided for stating the need for a model capable of predicting energy consumption of structured, parallel computations on heterogeneous system, before explaining why an accurate, high level model is needed to provide automatic means to optimize for energy consumptions. In the final part of this chapter, we present the iterative process followed to develop the models used in this work.

3.1 Problem

An approach to minimize energy consumption is to use an appropriate behavioural skeleton. From the knowledge of the computation structure, and of performance and energy models, an autonomic manager can be used to minimize a given energy metric, hence optimizing behind the curtains the overall cost of the computation. This allows the programmer (following the usual approach of structured parallel programming) to write efficient code performance-wise while minimizing energy consumption. Without such a structure, writing code that is energy efficient, performant and capable of running on heterogeneous architectures would require:

- careful code optimization, exploiting the machines' ISA at their best;
- definition of a concurrent activity graph, where activities (previously carefully monitored by the developer) are split into different devices depending on execution time and power consumption;
- testing of the overall defined parallel application;
- verifying the energy consumption of the computation.

This activity should be executed in a loop, as it can be seen in Figure 3.1, until a satisfying result is observed in the verification phase. This loop can be very costly, and it could be difficult to assess the energetic performance of the proposed solution without a baseline model.

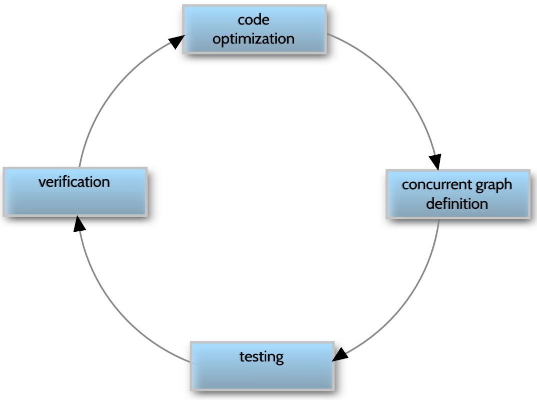


Figure 3.1: Manual process for energy optimization of a parallel application.

Even neglecting the cost of performing such a procedure, without a precise idea of the "ideal" behaviour of the computation, the programmer would have no precise idea of the "goodness" of the achieved result. Using this kind of static optimization it would be impossible to respond to degradations caused by *exogenous causes*. Consider the case of a multiprogrammed machine, with a GPU and some NUMA nodes. If an external process starts its execution while the computation is performed (for example occupying the same NUMA node of the executed computation) this could lead to an increase in completion time and to additional energy expenditure, while other resources (GPU, other NUMA nodes) are left idle, wasting energy dissipating leakage currents. The computation will experience a degradation in terms of performance, while more energy than needed will be dissipated both by the monitored application and by others racing with it for resources. An autonomic manager could be able to recognize the situation in which using a resource is more costly in terms of energy and time and perform changes in the concurrent activity graph so that the computations' and overall system's consumption is minimized.

An autonomic manager operates in terms of a loop, the well known *MAPE* loop. This loop requires to insert probes in order to monitor the computation. Then the behaviour is analysed in terms of a cost model before a suitable plan is devised and executed.

Several low level models exist for predicting the energetic behaviour of computations. However, to the best of our knowledge there are no high-level models able to predict how a computation implemented in terms of a well-known parallel programming pattern will behave on a heterogeneous system.

The aims of this work are:

- to develop and validate a methodology that can be used to predict power (and consequently, energy) in structured parallel programming, heterogeneous environments;
- to show how, with such methodology, accurate enough models can be developed;

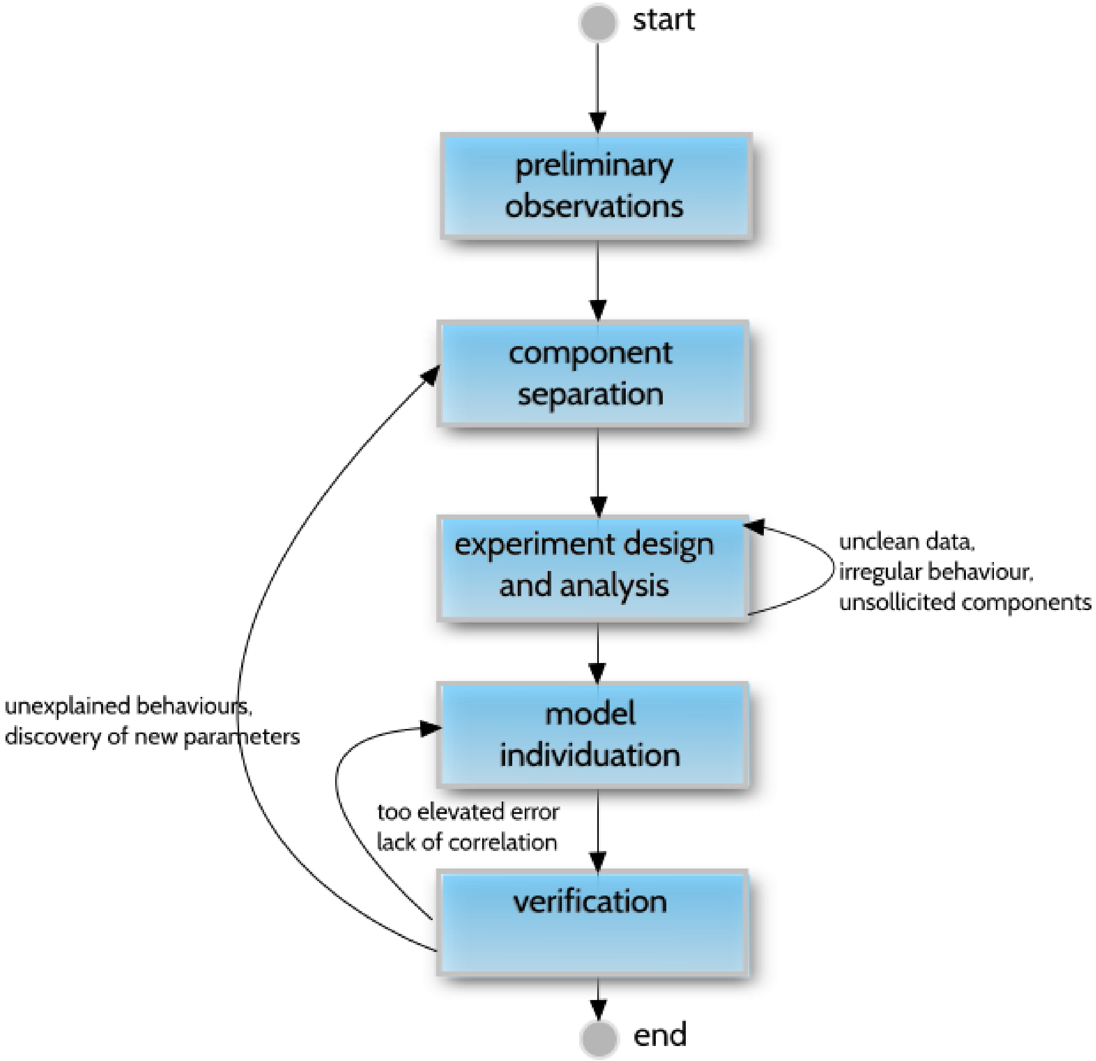

start
preliminary observations
component separation
experiment design and analysis
unclean data, irregular behaviour, unsollicited components
model individuation
unexplained behaviours, discovery of new parameters
too elevated error
lack of correlation
verification
end

Figure 3.2: Depiction of the procedure followed to develop the energy models used in this work.

- to present some models for well-known parallelism exploitation patterns, and show how they can be used to minimize energy consumption in heterogeneous systems.

In the following we will present the methodology used to develop the models presented, respectively, in Chapter 4 and 5. From an high-level perspective, the iterative process followed can be summarized with the block-diagrams of Figure 3.2. In the rest of this chapter, we will explain in detail the single steps.

3.2 Model development

3.2.1 Preliminary observations

Before starting studying the model, the behaviour of the system as a whole should be observed. This preliminary step is useful in order to understand whether there are patterns that can be exploited to develop a model capable of predicting the energy requests of a computation.

In this phase, the domain of interest should be reduced to a restricted part, so that stronger assumptions can be done. We present here the main ones made for developing our model: we will justify their validity later in Chapters 4 and 5, where the energetic models for CPU and GPU computations will be presented.

The first assumption is about the repeatability of the behaviour. We can state the assumption as follows:

The instantaneous power required during the execution of a certain computation C, with the same used resources and the same parameters, does not depend, from a probabilistic point of view, from its starting time.

We also assume that computations implemented using the same parallel pattern would exhibit similar curves from an energetic point of view, as they do with time. We can state it as follows:

Given a parallel exploitation pattern, different computations implemented with the pattern exhibit similar behaviours in terms of average power requirements when the amount and type of resources assigned for the calculation change.

This assumption can be verified experimentally.

The last observation needed to develop our model was about the meaningfulness of an *average case analysis.* In fact, while average could fail to capture the essence of the system, we show that with the tools we used and for the architecture analysed, the error introduced by average case studying is negligible for large enough computations.

This can be proved as for our domain of interest (map computations), usually the instantaneous power is constant at steady state, while transient states is constant and gets amortized for large enough computations.

3.2.2 Component separation

Starting from the knowledge of the system on which we are operating, the components that are part of it - and that supposedly contribute to the energy cost of the computation - should be individuated. This phase determines the *abstraction level* of the model. For example, in the specific case of an heterogeneous system, we could decide to consider the different devices as a whole as the single components to be analysed (high level). In the opposite approach, we could decide to divide the system in terms of the single units composing it. A lower abstraction level could bear to more precise results. However, the problem could become too complex to be solved, or, worst, could be difficult to use as too many parameters have to be taken into consideration when predicting.

Before individuating the components of interest, we should also consider the *precision* of the measurement infrastructure. As an example, an instruction-based analysis would be quite complicated using EML, given the operating sampling frequency: while an average case study could be performed, it would necessarily be affected by initialization costs and would require a complex setup. Even with more elevated sampling frequency, the impact of measurements should be taken into account: for microscopic variations, the impact of inserting probes might be too high to achieve meaningful results.

In case the decomposition is unsatisfying (depending on the outcome of the subsequent steps), a different one should be individuated.

In this work, we decided a decomposition based on the *parallelism degree* and the features of the analysed workload; this means that the model is expressed in terms of number of cores exploited (for the CPU part) and in number of blocks and warps per block for the GPU part, plus a computation-dependent rescaling.

3.2.3 Experiment design and analysis

In our case, data cannot be merely observed: we don't have a previously existing set of energy measurements for the target system, designed with a certain parallel exploitation pattern. Hence we should carefully design appropriate experiments, exercising all the individuated components in planned conditions. Variations between outcomes should be analysed to understand whether different parameters should be taken into consideration.

When monitoring an application, since a computing system is a complex environment, we are used to have variation in time (and also in energy) due to interference of non-predictable phenomenons. For this reason, the measurement must be robust and take into account possible variation in the environment from which the data is gathered.

During this work, we developed several experiments, that can be categorized summarily as follows:

- architecture level experiments, targeting the energetic cost of communication, leakage power or the cost of activating different resources;
- high-level experiments, targeting the difference in power caused by different parallelism degrees on different computations.

After the measurement of the experiments, we perform a sub-step, in which the outcome is analysed. Outcomes from different execution of the same experiment with same parameters are aggregated, cleaned (e.g. invalid behaviours are eliminated) and summarized so that they can be treated better.

3.2.4 Model individuation

Given the analysis of the impact of the components previously individuated on the measured quantity, in this phase we seek to find a "law" able to predict the behaviour of a system on the basis of the said parameters. We want this model to be an analytical one, so that changes in the system behaviour can be represented in a closed form. The usefulness of a model lies in the following three points:

- achieving a simple description/explanation of the data analysed, understanding the impact of each parameter on the measurements;
- given a functional relationship between a set of parameters and the measure of interest, we can infer the energy consumption for parameters' values that were not studied directly;
- the behaviour of the measured quantity can be predicted.

The way to identify such models changes; we can proceed either by *regression* or *interpolation.*

Interpolation

Interpolation [BM11] targets the problem of approximating an unknown function $f(x)$ starting from the knowledge of the value assumed by such function in a discrete set of n samples $x_1, x_2, \dots, x_n$. We seek to find a function $g(x)$ such that $g(x_i) = f(x_i), i = 1, 2, \dots, n$. Interpolation is particularly useful to understand which polynomial function approximates better the behaviour of the studied function and in order to understand the behaviour of f outside of the observed interval. From the definition of $g(x)$, we understand immediately that it could be that the function is *overfitting* the data we seek to model, hence providing a less trustworthy model than the one achievable through regression.

Regression

With this kind of analysis we seek to find a mathematical description of a process in terms of a set of associated variables. We want to achieve a mathematical description of an observed phenomena between a *response variable* y and a set of p *explanatory variables* $x_1, x_2, \dots, x_p$. For this reason, the phenomena is observed under a set of n p-dimensional vectors of parameters, achieving a set of points y_i (i.e. values that we observe and want to explain) [AL06]. Through this process, we model every y_i as $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots \beta_p x_p + \epsilon_i = \hat{y}_i + \epsilon_i$, where ϵ represents the difference between the predicted value ($\hat{y}_i$) and the measured one y_i.

A regression model is *linear* when the derivative of y with respect to the β_i parameters do not depend on β_i; for example, a model in the form $y = \beta_1 \sqrt{x} + \beta_0$ will be considered linear. There exists different procedures to individuate the set of parameters $\{\beta_i\}$ from the given data. We used *ordinary least squares* (OLS) which returns a set of parameters $\{\beta_i\}$ such that [BM11]:

$$\min_{\{\beta_i\}} \sqrt{\sum_i \epsilon_i^2}$$

A more complex type of regression is the *polynomial regression* [Wik15]. It is a form of linear regression in which the relationship between the response variable and at least one explanatory variable is expressed in terms of a n-th degree polynomial function. It can be used to fit a non-linear relationship between the value of the explanatory variable. Since the regression parameters only intervene linearly in the function, it is still a linear regression.

The assumptions required to perform linear regressions are the following:

- errors for different cases are assumed to be independent;
- the random component ϵ is a random variable, with zero mean and variance σ^2; it is normally distributed.

In this work, we used mainly *simple linear regression modelling* techniques in order to explain the energetic behaviour of computations.

3.2.5 Verification

The model is compared with actual data, hence analysing its effectiveness when tested in a real environment. In this phase the error between the model and the measurements (the aforementioned ϵ_i) should be analysed, possibly searching for common patterns indicating missing parameters. Notice that, since our model takes the computation as a parameter, we need to verify the behaviour for:

- the average case of a single computation, pre-determined;
- a single execution of a computation;
- a different computation.

Statistical tests, such as R^2, the coefficient of determination, should be calculated to understand the *fitness* of the data. Let $\hat{y}_i$ be the i-th element in the estimation vector; if $\overline{y} = \frac{1}{n}\sum_i \hat{y}_i$, corresponding to the average of the samples $\{y_i\}$ for OLS, we will have:

$$R^2 = \frac{\sum_i(\hat{y}_i - \overline{y})^2}{\sum_i(y_i - \overline{y})^2} = 1 - \frac{\sum_i(y_i - \overline{y})^2}{\sum_i(y_i - \hat{y}_i)^2} = 1 - \frac{SS_{res}}{SS_{tot}} \qquad \in [0,1]$$

Where SS_{res} is the *residual sum of squares*, that is the sum of the squares of the difference between the sample value and the predicted value and SS_{tot} is the *total sum of squares*, the sum of the squares of the difference of the sampled value from the sampled average.

The closer R^2 is to one, the better the function approximates the given data.

In case the error is too high, or the found model does not fit the data, either the parameters used to characterize the system or the regression function should be changed.

As an example, by trying to separate only dynamic power from the static one, the model would not be complex enough to explain the behaviour of any computation; as such, a more detailed decomposition of the system should be selected to try to find a more accurate model. Another error could be trying to apply the same parameters to any computation, with different amount of memory requests and different instructions, and hence energy cost.

Chapter 4

Energy model for map computations on GPU

In this Chapter we apply the previously explained methodology to provide a model and an heuristic for energy consumption prediction in GPU architectures. We start in the first Section by carefully studying different components' footprint from the energetic standpoint. We than move to individuate the components for our high-level model and to explain the experiments designed to exercise them. In Section 4.4, we used regression to understand the nature of the impact of the explanatory variables, as well as to effectively estimate the power consumption. We used these insights to develop an heuristic based on a metric computation. We finally move to validate the process as well as the model and the heuristic by showing the accuracy of the predictors with different computations.

4.1 Preliminary observations

Behind the development of every model there are some assumptions about the measure that we want to predict. In this subsection we will explain the assumptions made when developing the model for GPU computations. The analysis has been carried on mainly in terms of power: this is because robust and accurate time models already exist for parallel patterns; also, studying the power gave us the possibility to analyse more stable behaviours. We used two different NVIDIA Kepler boards. The first one is a K20C board, with 13 streaming multi-processors, with 192 CUDA cores each. The other one is a K40M board, with 15 streaming multi-processors and the same number of CUDA cores as the previous one. Both devices are equipped with two DMA copy engines, supporting concurrent data movements and execution. In both cases, the version of CUDA runtime used is 7.0. Let us start by the first observation that we made in Chapter 3.

Observation 1 *The instantaneous power required during the execution of a certain computation C, with the same used resources and the same parameters, does not depend, from a probabilistic point of view, from its starting time.*

Observation 1 can be verified very easily in case the machine is load-free, with the exception of the executed computation. We can see that, for a given architecture, the behaviour almost totally overlaps between different executions, started in different moments. For GPU architectures, we show in Figures 4.1, 4.2 the behaviour of the computation as of Listing 4.1, performed on two different boards, a K20c and a K40m, both by NVIDIA. The x-axis reports the time, as $t - t_{start}$, while in the y-axis we show the instantaneous power as detected by EML.

```
template<typename T, unsigned int REPETITIONS>
__global__ void kernel(T* in_a, T* in_b, T* out_c, size_t size) {
  unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
  while(tid < size) {
    out_c[tid] = 0;
    for(int i = 0; i < REPETITIONS; i++) {
      out_c[tid] += (in_a[tid] + in_b[tid]*i);
    }
    tid += blockDim.x * gridDim.x;
  }
}
```

Listing 4.1: A simple CUDA C kernel; `tid` represents the thread identifier and is used to decide which values of the output vectors should be calculated by every thread.

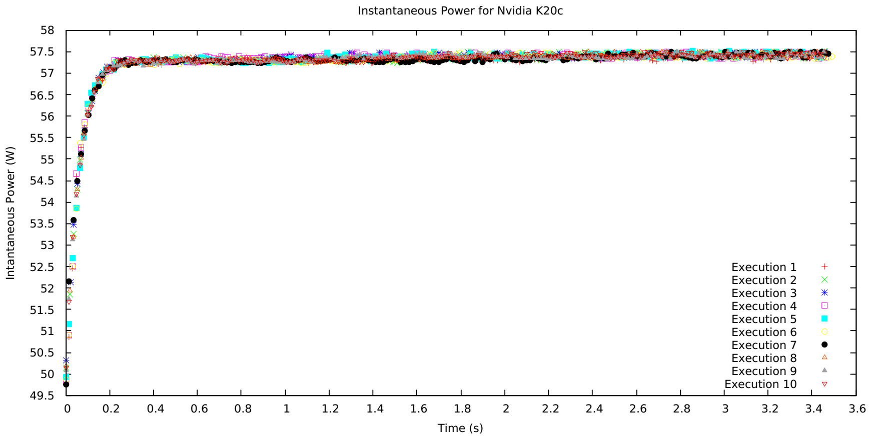


Figure 4.1: Detected power along the timeline of ten different executions of a map computation, Nvidia K20

The behaviour almost completely overlaps in both cases, and exhibits a very regular curve. This is because no other processes were running on the GPU while the computation was in execution. Notice that the high stability can be explained with the fact that several instructions are executed in the (wide) time between samples, $0.016s$. The power sampled in steady state varies at most of $\pm 0, 1W$ for the K40m board, hence having a negligible impact.

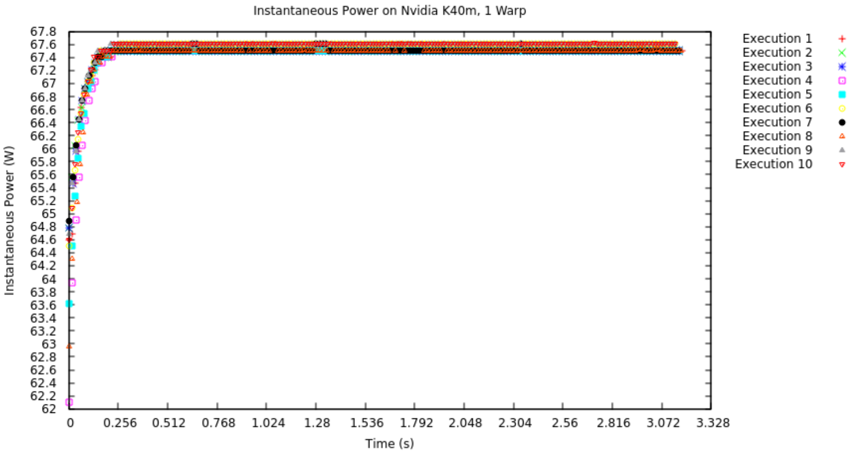


Figure 4.2: Detected power along the timeline of ten different executions of a map computation, Nvidia K40

The condition holds also in case of more irregular computations and for different levels of parallelism, as we show - respectively - in Figures 4.3 and 4.4. The first is the instantaneous power for a combination of two functions: in the former we compute a vector addition $c_i = a_i + b_i$, while in the latter we apply to vector c several trigonometric functions. The computation is executed on a single warp and on a single GPU. The second plot shows the kernel of Listing 4.1 in execution with 32 warps and 13 streaming multiprocessors. In the parallel case, we can notice a slightly higher difference between the instantaneous power of different computations.

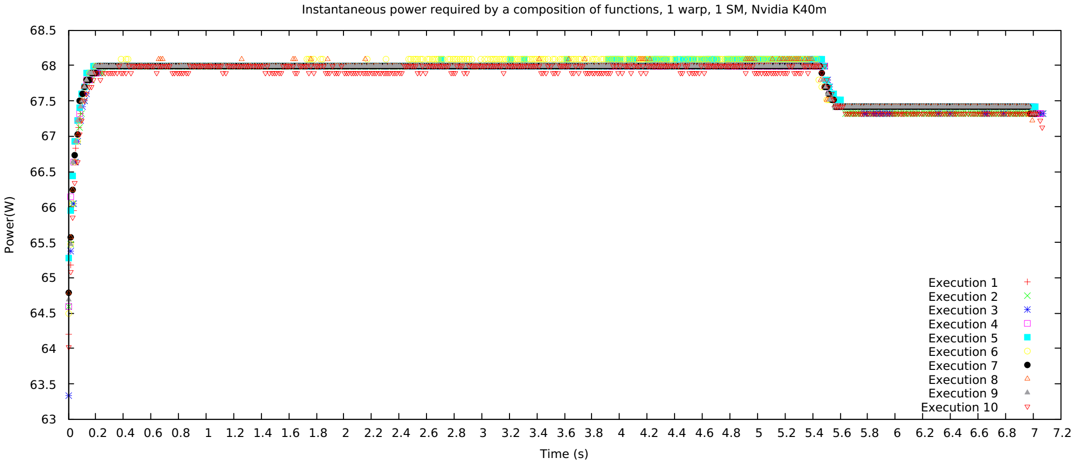


Figure 4.3: Instantaneous power of an irregular computation, executed on a Nvidia K40m board using 32 threads on a single streaming multiprocessor. Ten executions, launched at 100 seconds of distance are shown.

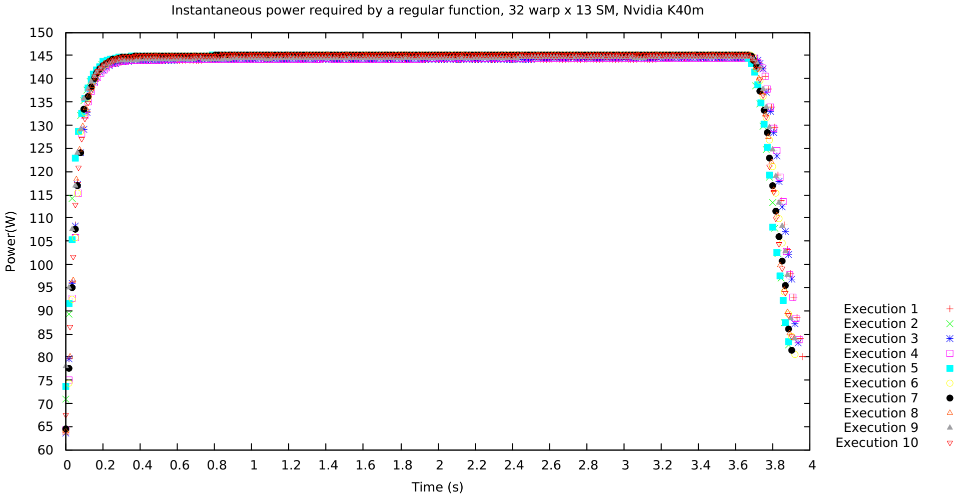


Figure 4.4: Instantaneous power of a regular computation executed over 32 warps for each of the 13 streaming multiprocessors available.

For the parallel case, the steady state power has the distribution visible in Figure 4.5. The histogram has on the x-axis the instantaneous power sampled at steady state on ten different computations (with more than 2100 samples), from the minimum detected value to the maximum one. Values are aggregated in 12 buckets of the same size. The plot shows the probability that a steady state power lies in a certain bucket. The detected average is $144.668W$, while the standard deviation is 0.59. We can see that the 96% of the values lie within $1W$ from the average.

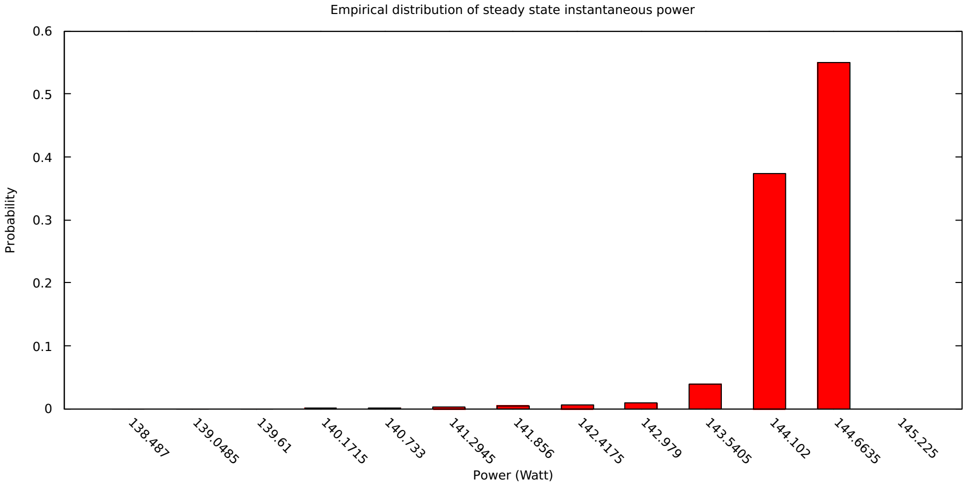


Figure 4.5: Distribution of steady state power for the computation of Listing 4.1 executed on 32 warps and 13 streaming multiprocessors.

For more realistic computation, as matrix addition, we can see that this condition holds from Figures 4.6 and 4.7, where in the first case the version executing with only one warp on a block is visible, while in the latter we see an execution spawned on 32 warps over each of 13 streaming multiprocessors.

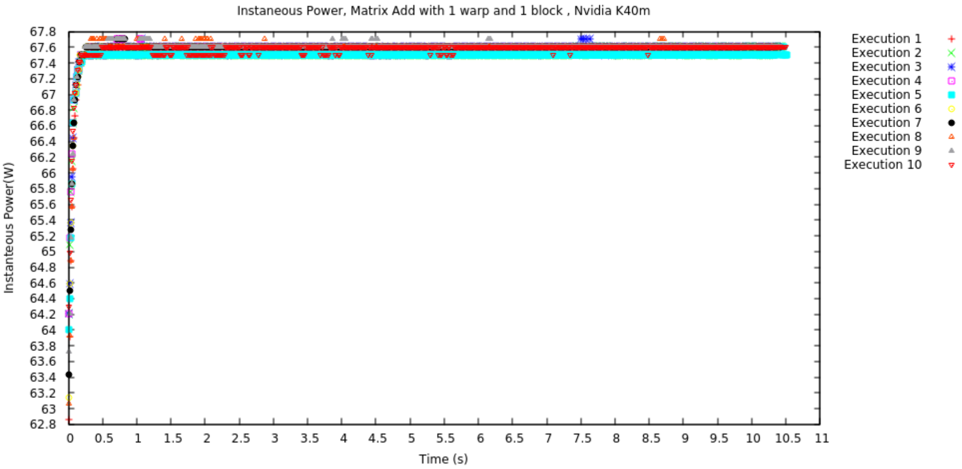


Figure 4.6: Instantaneous power of 10 different executions of a Matrix Addition (10240 × 10240) performed as a *map* computation on a single warp on a Nvidia K40m board

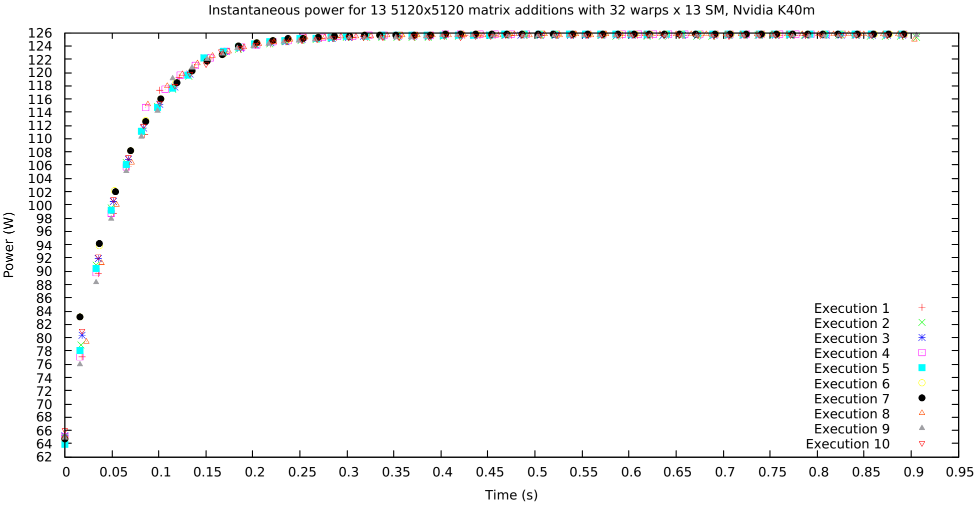


Figure 4.7: Instantaneous power of 10 different executions of a Matrix Addition (5120 × 5120) performed as a *map* computation on 32 warps for each of 13 allocated SM

The same holds for the case of matrix multiplication, visible in Figures 4.8 and 4.9 respectively, using the same resources of the matrix addition case previously explained. We see that for real computations the instantaneous power oscillates more, but has overall the same behaviour. In both cases, we observe the existence of an initialization phase, in which power requirement grows as resources are recruited and execution launched on them. Another - rather obvious - observation is that the power grows by increasing the parallelism degree.

Finally, a consideration on the two transient states that are visible in the plot. The first one visible in the beginning, can be explained with the initializations of the used resources. In all tested computations it does not exceed 0.2s: for computations large enough it has a negligible cost.

The other transient state is observable in 4.9. It can be caused by the progressive deactivation of resources: it can be seen in Figures 4.9 and 4.4. We observe how the duration of this phase depends on the grain of the work assigned to each of the parallel execution units: the coarser the grain, the more noticeable the unbalance. As a matter of fact, when we operate with fine-grained parallelism (see Figure 4.7), the deactivation cost is not observed as all resources are deactivated almost simultaneously.

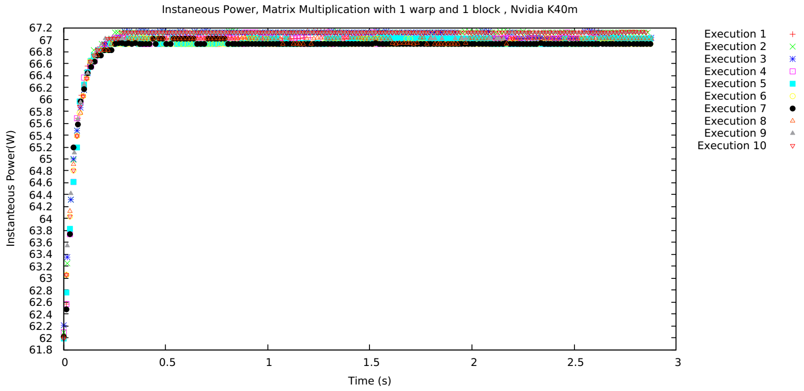


Figure 4.8: Instantaneous power of 10 different executions of a Matrix Multiplication (1024 × 1024) performed as a *map* computation on a single warp on a Nvidia K40m board

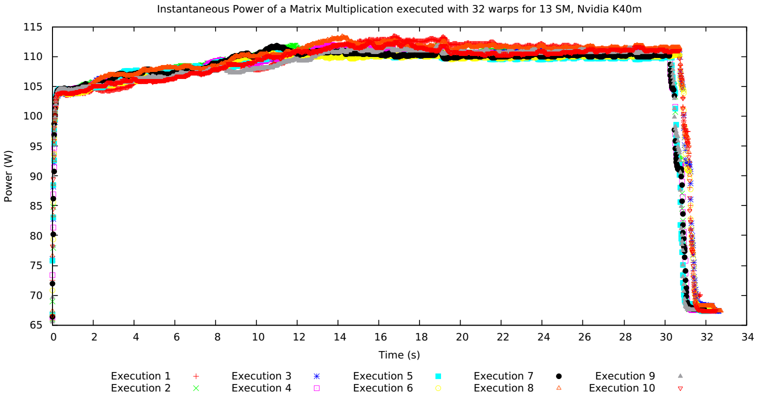


Figure 4.9: Instantaneous power of 10 different executions of a Matrix Multiplication, 32 warps for 13 SM, Nvidia K40M

Observation 2 *Given a parallel exploitation pattern, different computations implemented with the pattern exhibit similar behaviours in terms of average power requirements when the amount and type of resources assigned for the calculation change.*

To show how Observation 2 holds, let us start by considering the analysis, at the average case, of different map computation as computed on the GPU. We have different kinds of resources, namely blocks and warps, that can be allocated to the computation. We will consider them as a linear string of processing units.

Consider the curves depicted in Figure 4.10, representing the average power (in time) achieved using two blocks and a different amount of warps to perform different map computations.

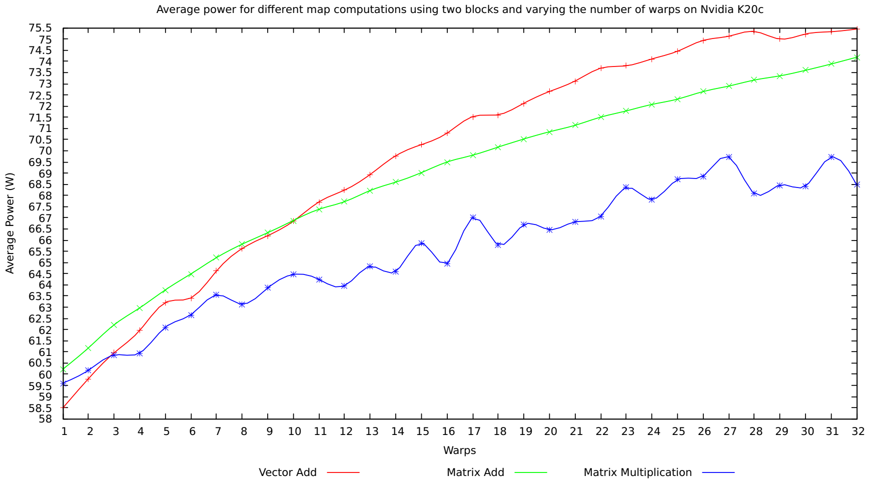


Figure 4.10: Consumed power with different map computations on 2 streaming multiprocessors, varying the number of warps.

It is possible to see that the behaviour is quite similar between the three considered computations. The same holds for different number of blocks: in Figures 4.11 and 4.12 we show the average required power for the same three computations considered before, varying the number of warps in each of the 4 and 8 blocks respectively.

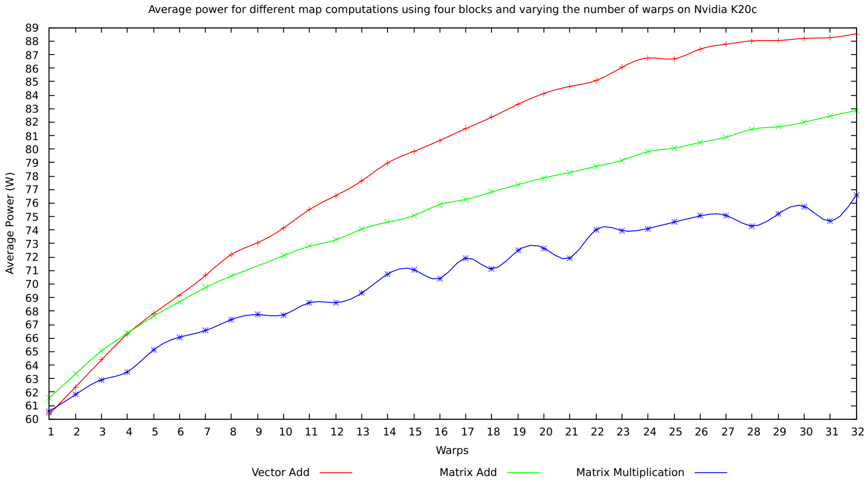


Figure 4.11: Consumed power with different map computations on 4 blocks, varying the number of warps.

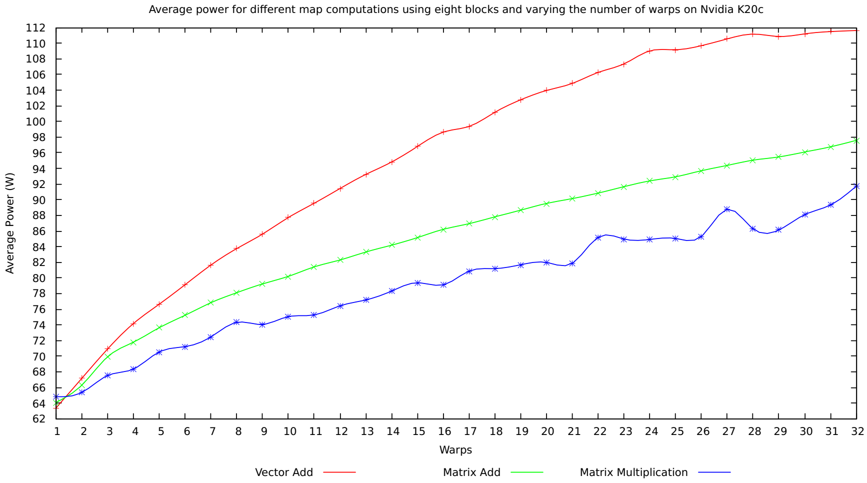


Figure 4.12: Consumed power with different map computations on 8 blocks, varying the number of warps.

The more irregular behaviour of the matrix multiplication computation is easily explained by the fixed partitioning, done in terms of rows.

Notice that the curve is always similar and always exhibits a sub-linear behaviour. Since the maximum number of CUDA cores available for a block will be no more than 192 on the analysed architecture, introducing more than 6 warps, while reducing completion time, will not cause a proportional increase in power consumption: from this point on, less resources are activated by increasing the parallelism degree. Hence, also in case of computations exhibiting a perfect scalability, we would expect power to follow a sub-linear growth.

For what regards the relationship between the different computations, we see that the required power for vector addition is (almost) always higher than in the other two considered computations. Also, the power requirements of matrix multiplication are always smaller with respect to the other two. This can be due to different ratio of memory operations against calculation operations, causing different requirements in power consumption.

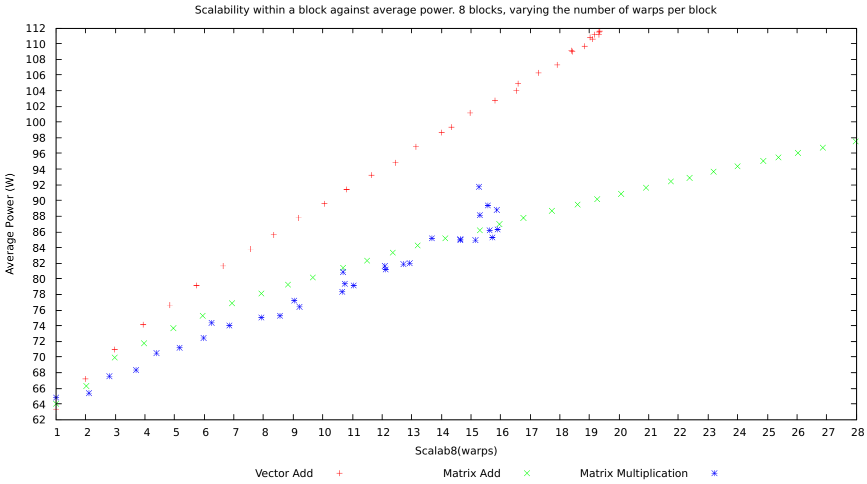


Figure 4.13: $scalab_8(w)$ against average consumed power for map computations.

To confirm the latest observation, we show how this behaviour is not related to the computation's scalability. In Figure 4.13 we show the average power for executing the computation with w warps on the y axis, while on the x axis we have $scalab_b(w) = \frac{T_{GPU}(b,1)}{T_{GPU}(b,w)}$, defined as the ratio between the execution time using b blocks with 1 warp each and the execution time using the same number of blocks with w warps each.

Finally, to validate further this observation, we wish to show how the same profile in term of power holds also for the steady state instantaneous power and on a different GPU board with respect to the one used to calculate the figures of the above plots. We show it for a single computation, spawned on two blocks: in Figure 4.16 we can see the instantaneous power in a portion of the steady state, as detected during the execution of a vector addition procedure on a different number of warps; a smaller number of warps obviously requires less power. However, the growth in power given by the activation of an additional warp decreases when the number of already activated warp increases. The sub-linear behaviour is confirmed also by Figure 4.15. In this last Figure, we depict the average (in time) of the power required to compute a vector addition using one block and different number of warps, against the instantaneous power as detected in the same time instant at steady state, in different executions of the same computation launched with a different number of warps. In Figure 4.15 it is also possible to observe that the average power and the steady state power approximately follow the same function. As said before, if the computation lasts long enough the transient state(s) get amortized.

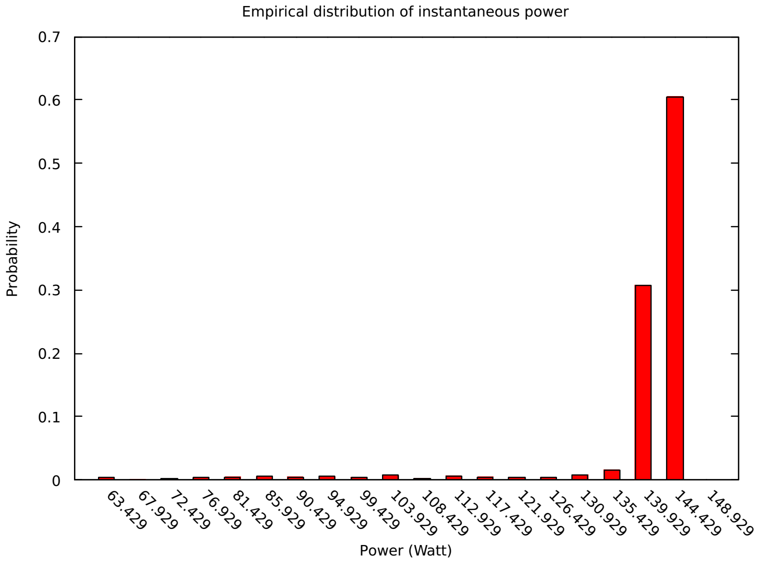


Figure 4.14: Distribution of total power for the computation of Listing 4.1 executed on 32 warps and 13 streaming multiprocessors.

Consider again Figure 4.5 in comparison with Figure 4.14, where we show instead the empirical distribution for the whole computation. The average becomes $142.29W$ with a standard deviation of 11.49. By considering a computation that lasts 4 times more, we achieve an average power of $143.93W$, way closer to the steady state average. This is because of the impact of the transient phases: while considering it will allow to have a more accurate information about energy consumption, changing the size of the data processed this information would become less and less reliable. Considering only the average in the instantaneous phase provides estimations that are independent of the data (and will hold for bigger computations), but the energy consumption will be overestimated.

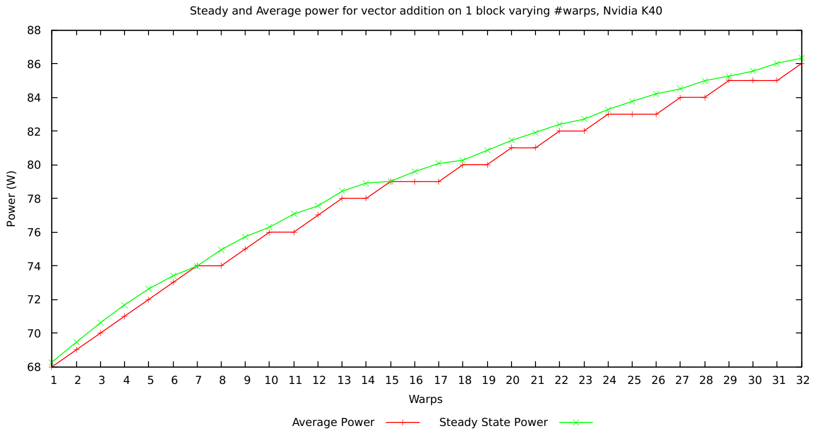


Figure 4.15: Instantaneous (green) and Average (red) power consumed by a vector addition on a Nvidia K40m, varying the number of warps within one block.

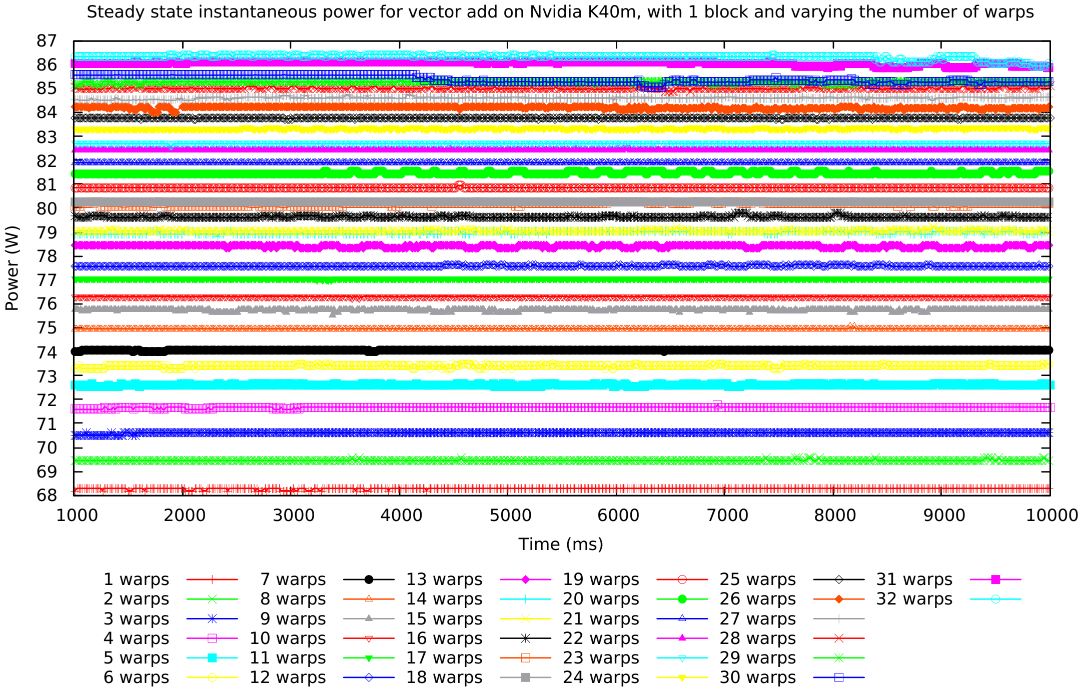


Figure 4.16: Instantaneous power requirements (at steady state) for a (huge) vector addition map, performed with different number of warps on a single block on a NvidiaK40m architecture.

If the map function is a sequential composition of functions with widely different instructions and hence energetic behaviour (as of the one depicted in 4.3), the average power will not converge for longer computations to the steady state one, making its study more complex.

As a final consideration, we consider the case of the reduce parallel pattern. *Reduce* is a well known second order function, taking in input an associative operator $\otimes$ and a collection A of size N. It calculates $\otimes$ over all the elements, until a single result is collected. Hence:

$$reduce(A, \otimes) = A[0] \otimes A[1] \otimes \cdots \otimes A[N]$$

Usually this operation is performed in parallel using a *tree* structure: the collection is, at first, split between n workers in equal parts. The operator is applied sequentially to each one of the elements, and the final result is calculated in $log_2(n)$ steps by collection of the temporary values elaborated by each of the units. In each of the step the number of processing units effectively used is halved, until the root of such tree returns the result, eventually broadcasting it to all the workers interested in it.

Consider the computation implemented in Listing 4.2. In this case the reduce step is implemented in term of warps: at step i of the tree descend, every thread $j \times 32 + x$ belonging to $warp_j$ applies the operator over its partial sum and the one by thread $(j + i) \times 32 + x$.

```
template<typename T>
__global__ void reduce_sum(T* input, size_t size, T* out) {
  unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
  __shared__ T partialsum[maxWarps][warpSize]; T tmp = 0;
  while (tid < size) {
    tmp += input[tid]
    tid += blockDim.x * gridDim.x;
  }
  partialsum[threadIdx.x/warpSize][threadIdx.x%warpSize] = tmp;
  __syncthreads();
  int i = (blockDim.x/warpSize)/2; //number of warp / 2
  /** Reduce between warps: thread x%32 in warp z takes the value of thread x%32 in warp z + i */
  while (i != 0) {
    if (threadIdx.x/32 < i) {
      partialsum[threadIdx.x/warpSize][threadIdx.x%warpSize] = partialsum[threadIdx.x/warpSize][threadIdx.x%warpSize] + partialsum[threadIdx.x/warpSize +i][threadIdx.x%warpSize];
    }
    __syncthreads();
    i/=2;
  }
  /** In this last step, a single warp exists. We reduce all partial sums within it. No synchronization is needed. */
  i=warpSize/2;
  while(i != 0 && threadIdx.x/warpSize == 0) {
    if(threadIdx.x%warpSize < i)
      partialsum[threadIdx.x/warpSize][threadIdx.x%warpSize] += partialsum[threadIdx.x/warpSize][threadIdx.x%warpSize + i];
    i/=2;
  } if(threadIdx.x == 0) atomicAdd(out, partialsum[0][0]);
}
```

Listing 4.2: $reduce(+, input)$ implemented on the GPU. It is logarithmic in the number of warps used within each block.

In Figure 4.17, we can see that the first step, in which the operator is applied to all of the data assigned to the thread, is the most costly, making the cost of computing the rest of the tree unobservable.

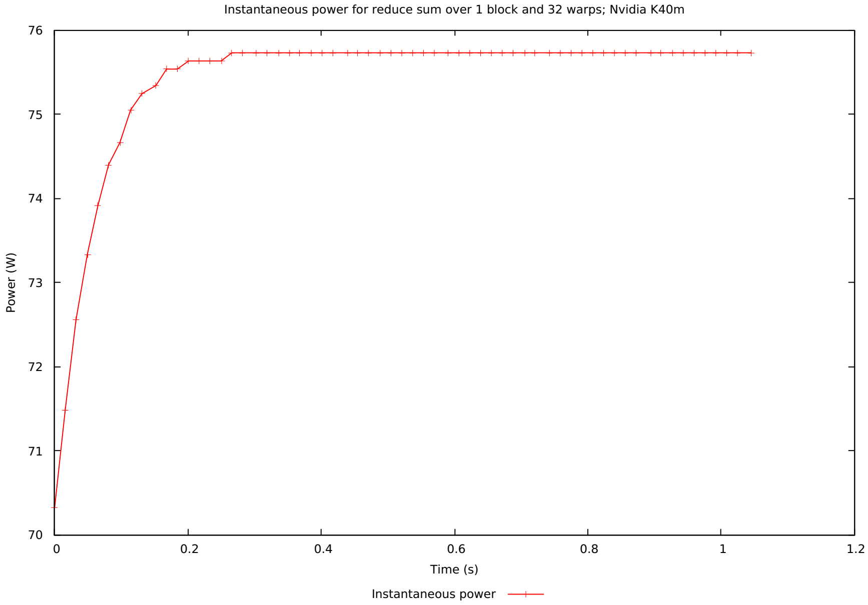


Figure 4.17: Instantaneous power of a reduce computation with operator +, executed using 32 warps and 1 block, performed on a Nvidia K40m board.

In this case, the overall power behaviour of the reduce computation will be totally similar to the case of the map, as shown by the red curve in Figure 4.18. However, in case a more costly operator ⊗ is given to the reduce, the instantaneous power exhibits the logarithmic steps, as depicted in 4.19 and the behaviour at the average case changes accordingly: using the green curve in Figure 4.18 we show the average power in time for performing a *costly* reduce computation varying the number of warps.

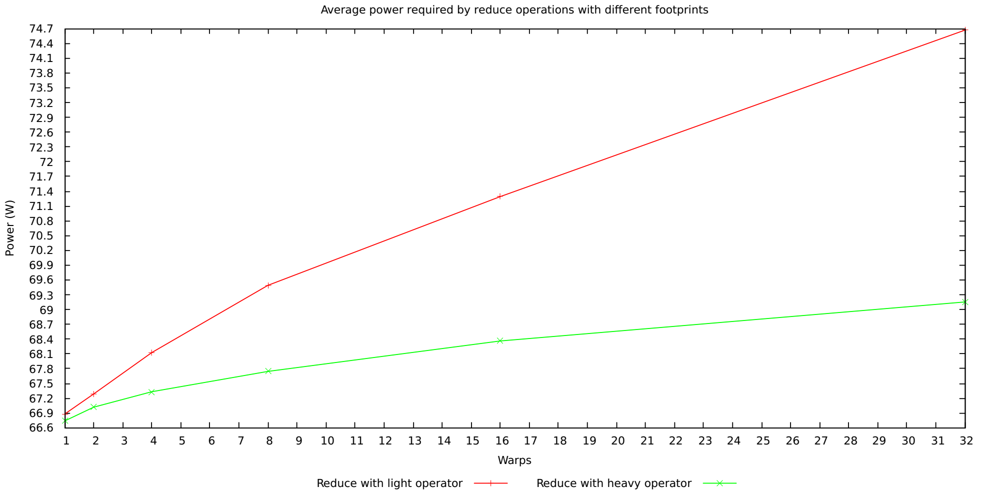


Figure 4.18: Average power required by two different reduce computations, executed on a NVIDIA K40m board.

The curves are quite different, meaning that in this case a more complex characterization (taking into account the weight of the used operator) should be performed.

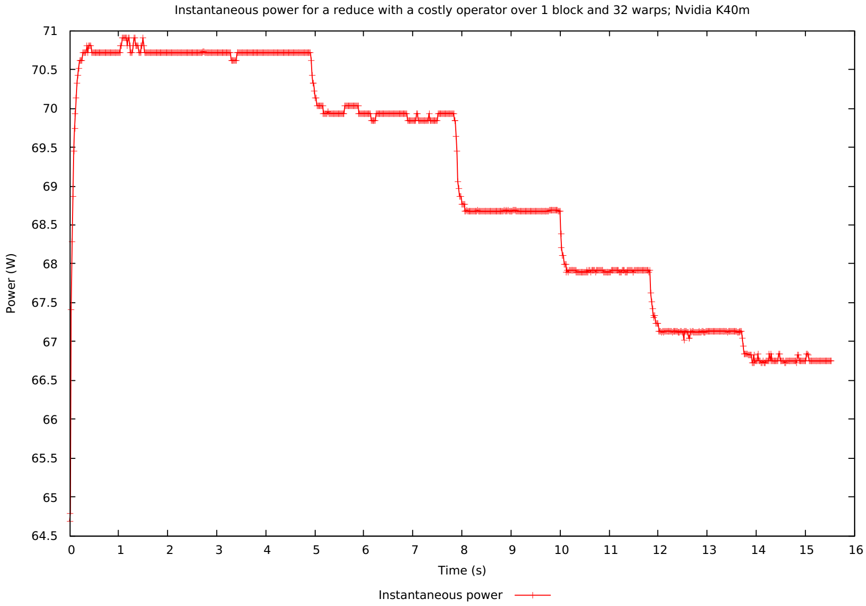


Figure 4.19: Reduce performed with 32 warps on a single block with a more costly (in terms of time) operator $\otimes$. We can see that the number of steps are $log_2(32) + 1 = 6$.

In this case the average in time will not converge to the steady state instantaneous power, hence making the study of such parallel pattern more complex. However, we advocate that, by knowing the behaviour of the overall computation, an approach similar to the one followed in this work could be applied to other parallel patterns.

Summary

In conclusion, we have seen that the same computation with the same parallelism degree has the same power at steady state, independently of the time in which it is executed and of the size of the problem. This allows to neglect the amount of data fetched to the device for the computation and consider only the computation for the sake of estimating steady state power. We have also seen that computations implemented according to similar parallel exploitation patterns behave similarly in terms of power as well as they do in terms of time, hence validating a study based on structured parallel programming with the aim of saving energy in heterogeneous architectures. Finally, we have seen that the average case study for map computation is a valid approach, as the average power converges to the steady state one.

4.2 Component Separation

As previously stated in Chapter 3, the proposed model is based on the parallelism degree and on some informations about the workload. In the following, we will give a justification for this choice.

GPUs are organized in two levels. The first level is the number of *streaming multiprocessors*; the real execution is carried on by *threads*, scheduled on CUDA cores. A streaming multiprocessor might execute more than one block. However, a block is always resident in a unique SM. Since threads are hardware managed and have negligible switching cost, the huge latency of the CUDA cores long pipeline is usually amortized by operating on more threads than available CUDA cores on a streaming multiprocessor. Threads are always scheduled, on the studied architectures, in batches of 32 (called *warps*), with consecutive identifiers.

For this reason, we have a sort of "two-level" parallelism degrees: we can establish a parallelism degree and decide where execution units should be used. CUDA allows to address this two level organization by defining programs in terms of threads and blocks. Hence it is natural to define a model that exploits this low-level knowledge given to the programmer in order to study the energy consumption of the system. Hence we seek to estimate power consumption in terms of *blocks* and *warps* scheduled in each block.

4.2.1 Warps

We show in Figure 4.20 that the impact in terms of power consumption of scheduling threads rather than warps is negligible. In the Figure 4.20 it is possible to see the instantaneous power consumption for the computation as of Listing 4.1. Different data lines correspond, respectively, to executions with 8, 16 and 32 threads, over the same amount of data.

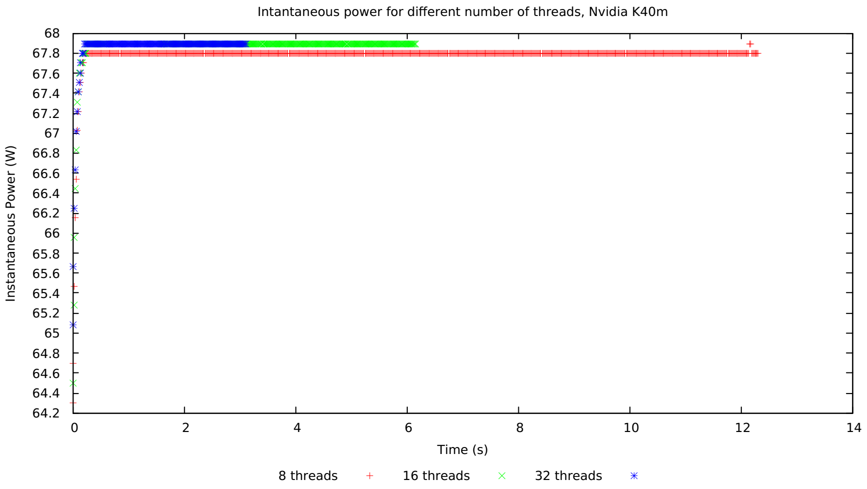


Figure 4.20: Instantaneous power for executing a computation using 8, 16 or 32 threads within a warp.

The area below each of the lines represents the consumed energy: we show in Table 4.1, the behaviour in terms of energy, time and average power for the above configurations. We can see from the data that operating in terms of single threads rather than in terms of warps does not affect significantly power consumption, while a considerable amount of energy is wasted.

Threads	**Energy(J)**	**Time (s)**	**Avg. Power(W)**
1	6631.48	98.49	67.33
8	833.92	12.31	67.75
16	418.51	6.17	67.83
32	211.69	3.12	67.85

Table 4.1: Energy, Time and Power for using different amounts of threads within a warp

We can appreciate better the small difference in power caused by the activation of threads within a warp if we compare it with the difference caused by activation of different warps. In Figure 4.21 we show the instantaneous power for the computation of Listing 4.3, starting with 2, 4, 6 and 8 warps within the same block. The kernel is designed to cause a progressive deactivation of the warps. We can see a number of steps, corresponding to the moments in which warps get deactivated. We can also observe (starting from the bottom) how the same amount of warps causes the computation to require approximately the same instantaneous power.

```
template<typename T, unsigned int r>
__global__ void kernel(T *in_a,T *in_b, T *out_c, size_t size) {
  unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
  unsigned int warpId = tid/warpSize;
  while(tid < size) {
    out_c[tid] = 0;
    for(int i = 0; i < (1+warpId)*r; i++) {
    out_c[tid] += (in_a[tid] + in_b[tid]*i);
  }
  tid += blockDim.x * gridDim.x;
  }
}
```

Listing 4.3: Computation causing progressive warp deactivation.

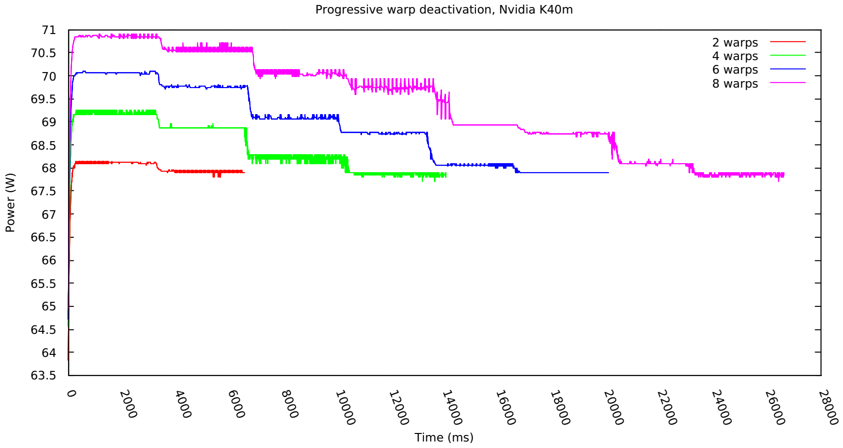


Figure 4.21: Progressive warp deactivation starting with 8, 6, 4 and 2 warps within the same block.

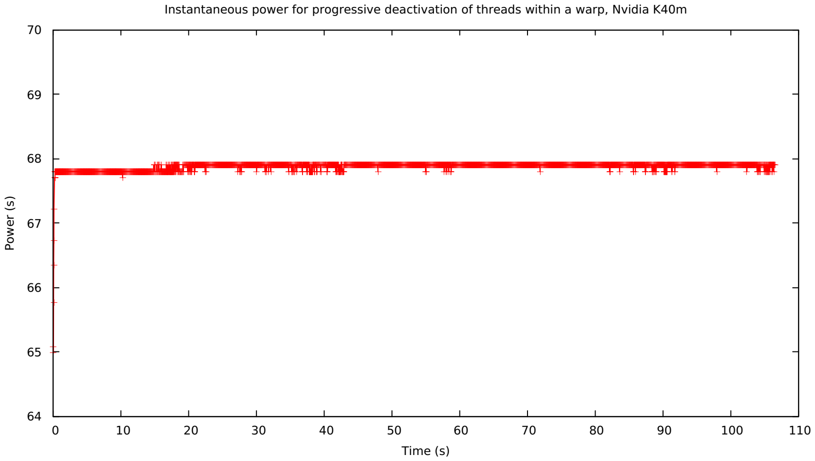


Figure 4.22: Progressive thread deactivation within a warp in execution on a single block.

We can compare Figure 4.21 with Figure 4.22, where the 32 threads within a warp are deactivated progressively using the same computation of Listing 4.3, with a number of iterations proportional to the thread identifier. We can see that there's no noticeable difference in power.

For this reason we decided to use the number of warps as parameter rather than the number of threads. Despite the fact that CUDA does not provide an abstraction to program directly in terms of warps, their number can be easily calculated as $\lceil threads/warpSize \rceil$ and given the small difference in power consumption, the energy estimation will remain reliable.

We also wish to show that making this consideration in terms of scheduled warps, rather than used CUDA cores makes sense. On the GPU used to generate Figure 4.21 every SM has only 192 CUDA Cores: this means that at most instructions belonging to 6 warps are served in a given clock cycle. However, we see that the power changes also in case of the purple curve, where 8 warps are activated.

We can see that this holds also for the maximum number of warps allowed as of today by CUDA (32) within a single block, in Figure 4.23.

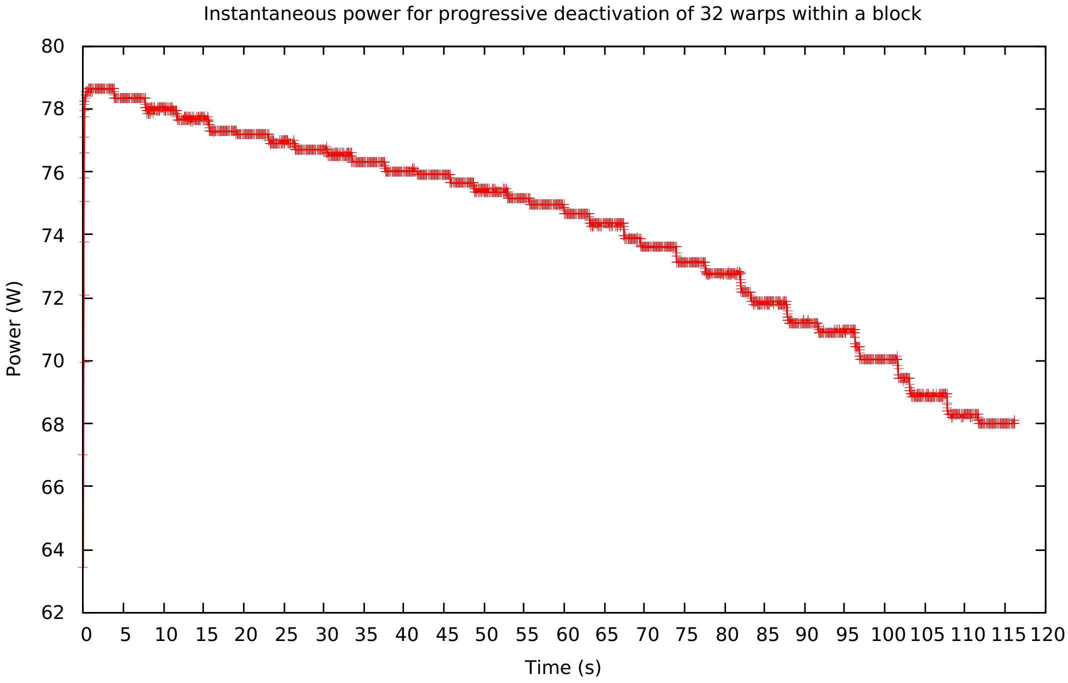


Figure 4.23: Power profile for the deactivation of 32 warps

4.2.2 Blocks

While not properly computing nodes, streaming multiprocessors are equipped with the part that manages the control flow of the computation and dedicated memories. These parts, responsible of data movement and of instruction control, according to [Kec11] pose about 20-40x energy overhead. This cost dwarves the one of calculation, that for current CPUs is about $1.2nJ$.

For this reason, we decided to consider the blocks as parameter of our model for GPU architectures. In Figure 4.24, we see the instantaneous power for 8 warps, executing (red line) on a single SM or across 8 (1 warp per block, green line). We see that using different blocks impacts both on power (that will be higher) and on the completion time, that will be slightly smaller when using more blocks.

CUDA does not guarantee that different blocks will be spanned over different SMs: we are not able to know in advance (unless we schedule the maximum number of threads manageable by a single SM) where will a single block be executed. However thE larger difference in power that we can observe in Figure 4.24, sug-

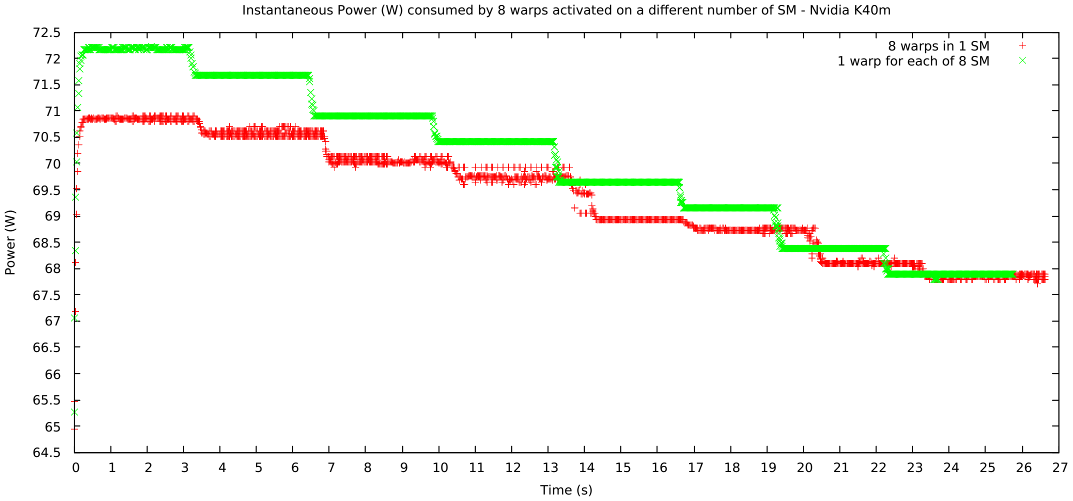


Figure 4.24: Instantaneous power for warps (activated on different SMs) progressively deactivated

gests that different SM are allocated to different blocks. While we are not able to demonstrate exactly the way in which blocks are allocated, the huge difference in power between the green and red first steps (even though the operation is the same and on the same amount of data) suggests that more resources are allocated. Being the number of CUDA cores effectively used exactly the same, the difference in power can be safely attributed to the usage of 7 more SMs.

4.2.3 Computation

However, knowing which processing units are used to carry on a computation is not enough to predict their behaviour in terms of power. First of all, different operations will require different amount of energy; as reported in [PLS10], different operations will have widely different energetic requirements and latencies. In Figure 4.25 we see the difference (both in terms of power and of completion time) between different operations, operating on the same amount of data of the same data type, carried on with the same amount of resources.

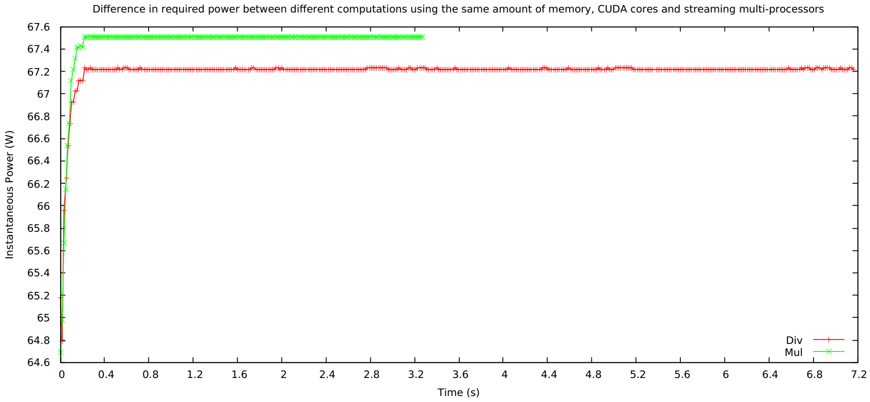


Figure 4.25: Instantaneous power for 1 warp performing two different operations on the same amount of data and exploiting the same data access pattern.

The green line (*mul*) represents the instantaneous power for executing Listing 4.3, while the red line shows the instantaneous power when we replace `in_b[tid]*i` with `in_b[tid]/(i+1)` (*div*). We see that the latter operation is less costly in terms of power consumption, hence variation in power requirements due to different operations should be taken into account when developing the model. To understand better the nature of our model, we also show the case in which more warps are used to perform the said computations in Figure 4.26. We can see that not only the power is different at the beginning, but the difference in power caused by warps deactivation is also different for the two computations, indicating that the function performed has a multiplicative effect in terms of the required power for a given parallelism degree.

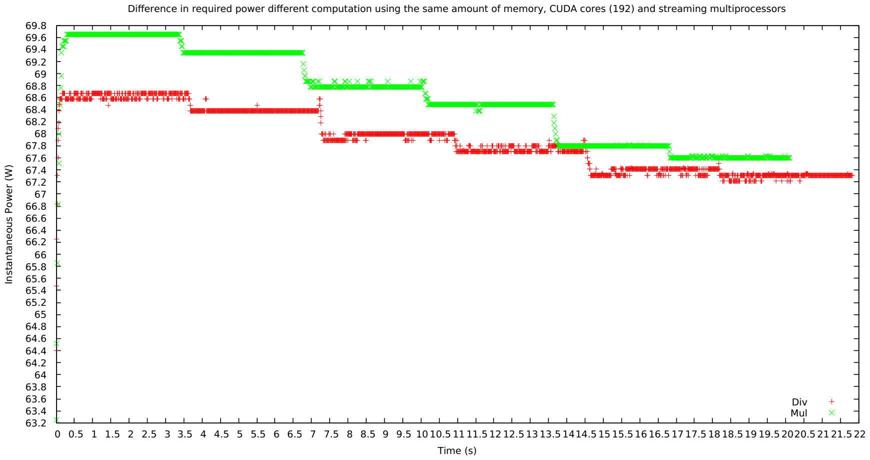


Figure 4.26: Instantaneous power for progressively deactivating warps (6 in the beginning) performing different operations. *Div* line is time normalized (factor of 0.5) to better show the difference in instantaneous power.

4.3 Experiment design and analysis

We divide the experiments designed in this phase in two categories: architectural experiments and high level experiments. As said before, in the first part we target a specific part/usage scenario of the targeted system, in order to collect meaningful information about energy consumption in these cases. We choose architectural experiments to target core parts, like *communication* or *leakage power* that are useful for better understanding high level behaviour. The second case addresses the behaviour of the system as a whole, as it would be used in a normal situation.

We perform the experiment design and analysis in a loop. The main aim of the analysis, in this case, is to check whether the achieved results are reliable and no "odd" behaviours were detected. After this first phase, we *summarize* the data, by using the average between different experiment and their variance to characterize the features of a specific analysed component.

4.3.1 Architectural experiments

Leakage Power estimation

The first experiment targets the leakage power. This is the amount of power consumed by a device even though no operation is currently performed on it. In order to perform this experiment, the analysed component should be *completely load free*. With the used monitoring library, it is not possible to ensure this condition: in fact, the library reads a specific register available on the GPU with a specific sampling frequency; this will always cause an overhead in the measurement. However, since it is not possible for us to exclude this component from the behaviour of the system, we characterize leakage power as the consumed power when no computation is going on and the monitor is active.

To detect the leakage power of a GPU device, a continuous sample on it is executed using a thread spawned on the CPU. The thread performs continuous samples of the device power consumption at fixed intervals. The process terminates when the detected power converges. The CPU code for the experiment can be seen in Listing 4.4.

```
do {
     previousPower = power;
     emlStart();
     sleep(sleep_seconds);
     emlStop(data);
     emlDataGetConsumed(data[0], &consumed);
     emlDataGetElapsed(data[0], &time);
     power = consumed / time;
  } while (fabs(power - previousPower) > 0.001);
```

Listing 4.4: Leakage power monitor for GPU; code executed on the host part. `sleep_seconds` is the duration of the sample

The value required for convergence is equal to the maximum variation that can be detected using the library, which has a precision of $10^{-3}W$. As it can be seen in Figure 4.27, the detected power in the interval, while starting with a very high value, converges rapidly to a common value, independent of the sampling interval. We consider this value the static power consumed by the device.

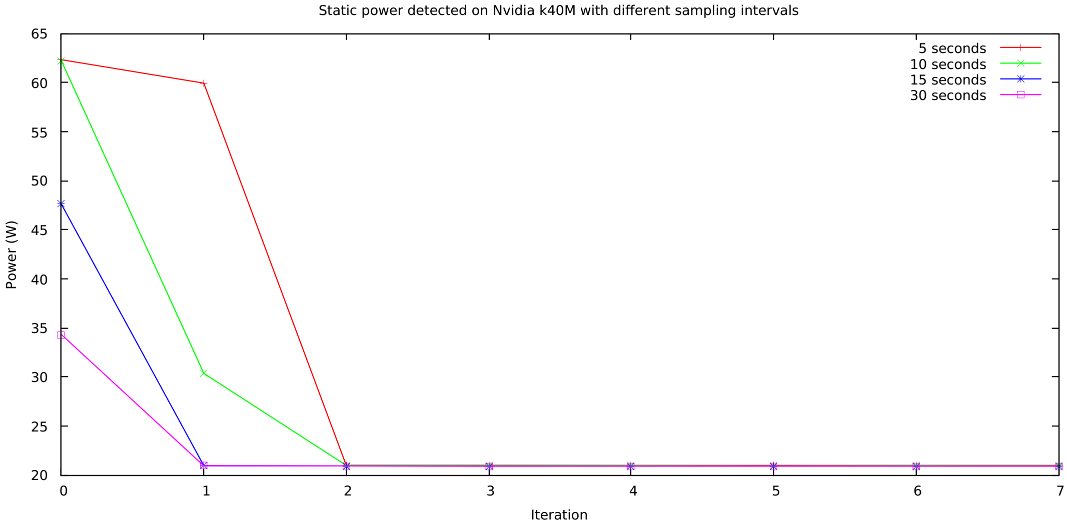


Figure 4.27: Power consumption detected at different iterations of Listing 4.4, changing the interval length

Difference in resource activation

Through the kernel of Listing 4.3, we address the difference in power and time caused by the activation of different resources. In this section we do not depict the results, as they have been used and explained before in this Chapter when the choice of the model parameters has been justified and in the preliminary observations part.

"Empty" computation energy consumption

This kind of experiment is a sort of extension of the previous one. However, while in the other we were targeting mainly the instantaneous power behaviour, through this experiment we wish to have a depiction of the behaviour at the average case. To do so, we defined two different computation, that will be later used as baseline for the model. The aim of this experiment is finding a computation with the "minimal" energy consumption, given a certain parallel exploitation pattern. The first case that we address is a sort of implementation of the *sleep* function on a GPU architecture. The code is visible in Listing 4.5.

```
template<typename T>
__global__ void sleepComputation(T *A, size_t N) {
 unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
 clock_t start_clock = clock64();
 clock_t clock_offset = 0;
 while (clock_offset < N)
  clock_offset = clock64() - start_clock;
 A[tid] = clock_offset;
}
```

Listing 4.5: Sleep kernel. The clock offset is saved in global memory so that the compiler does not optimize away the kernel code.

Since we expect any map computation to have a similar form (e.g. iterating over a set of data parallel tasks and perform something on them until completion), this is the computation with the minimal energy footprint that we were able to obtain reflecting the effective behaviour of the architecture. We analyse the average required power from the outside, that is the time average.

In Figure 4.28, we show the average power required for executing the kernel, varying the number of warps activated within a single block. The same data is plotted using $\sqrt{\#warps}$ in the x-axis in 4.29. The curve is very similar varying the number of blocks between 2 and 13.

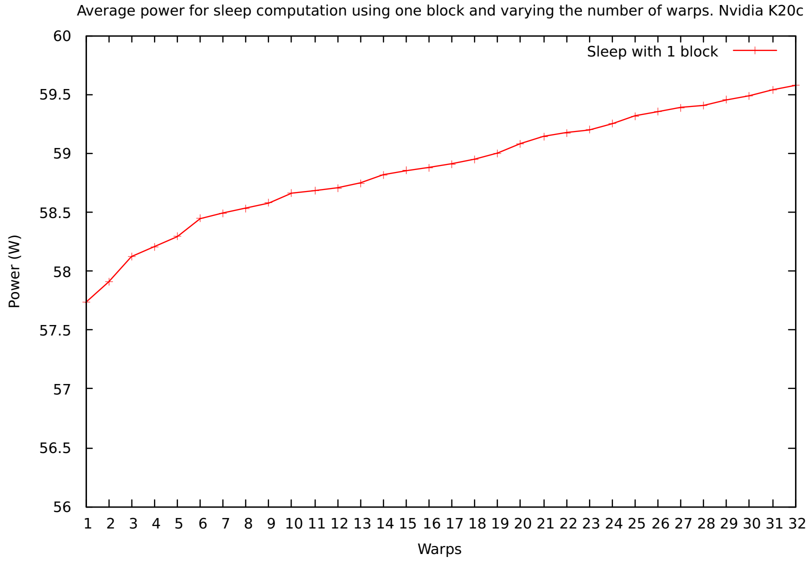


Figure 4.28: Average power for sleep computation (Listing 4.5), against the number of warps.

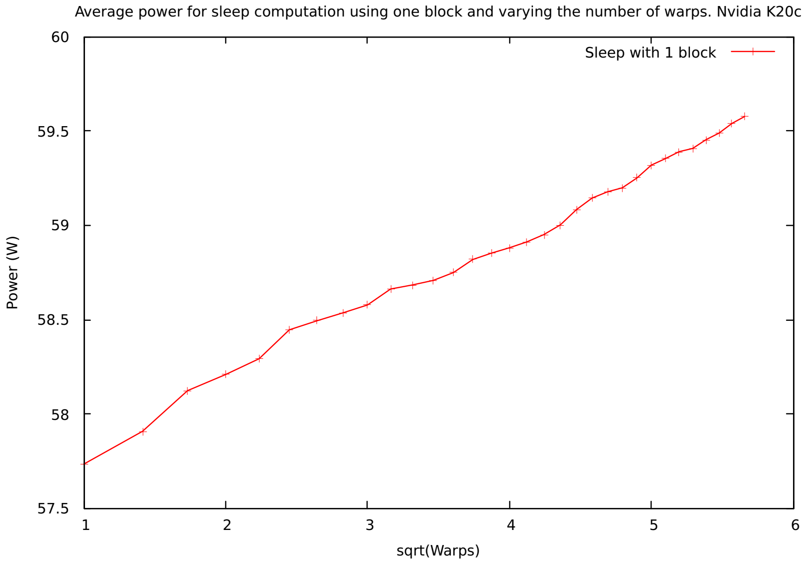


Figure 4.29: Average power for sleep computation, against the square root of the number of warps

The second computation that we defined for analysing the minimal footprint of the computation is a sort of empty map, in which every worker performs a fixed number of `nop` operations. The average power consumption for this computation can be seen in Figure 4.30 against the number of warps and in 4.31 against the square root of the number of blocks. The behaviour is slightly more irregular with respect to the previous case.

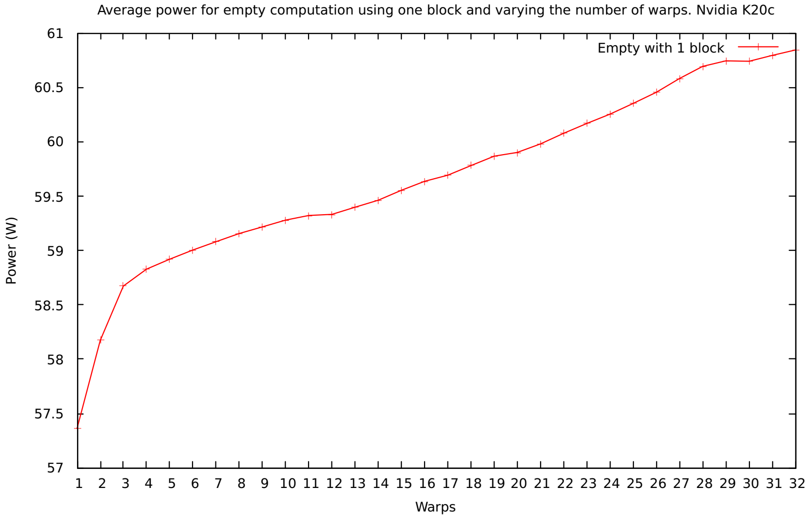


Figure 4.30: Average power for empty computation, against the number of warps.

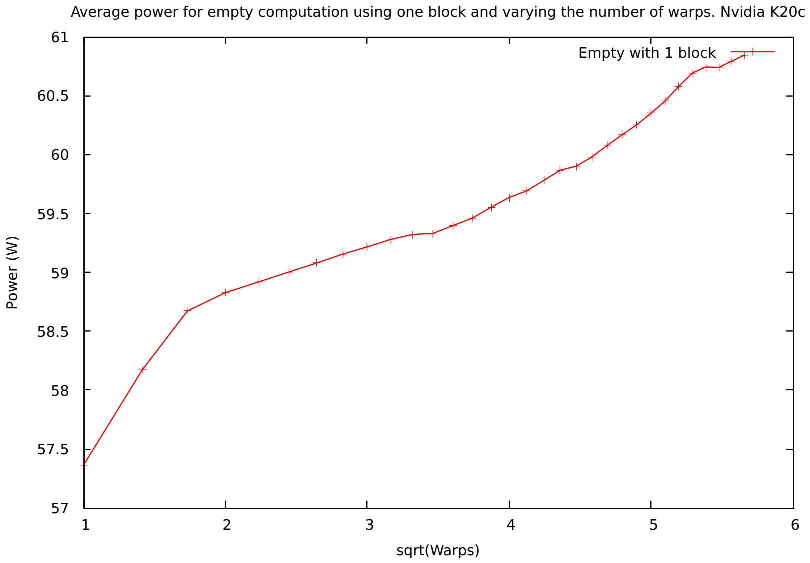


Figure 4.31: Average power for empty computation against $\sqrt{warps}$

We will discuss further the outcome of this experiment in Section 4.4.

4.3.2 Communication costs

This experiment addresses the costs of data motion, from an energetic and performance standpoint. What we want to understand is in what form we can model the time to transfer data from the host memory and the device memory, and what is the cost of activating the GPU structures (copy engine and memory) designated to perform said operation. We individuated two experiments, to understand the impact of using *pinned* memory and *unpinned* memory. For the first case the allocation must be performed using a special instruction provided by the CUDA runtime, namely `cudaMallocHost`. The memory allocated through this call is *page locked*, meaning that the page cannot be switched to virtual memory. To perform transfers in case the page used is not pinned, the runtime [CGM14]:

- copies the data from host non-pinned memory to host pinned memory;
- starts transferring the data from pinned host memory to the GPU local DRAM.

In case the data to be transferred already resides in pinned memory, the transfer is performed at much higher bandwidth; the amount of page-locked memory is limited, as an exceeding amount of it would cause performance degradation.

To understand the energetic cost of data motion, we need to address i) the *time* used to send data, depending on the granularity of the transfer, depending on the type of host memory used and ii) the power of the copy engine and DRAM activation.

The code of the experiment is shown in Listing 4.6 for the case of non-page locked memory. A chunk of 1GB (`TOTALSIZE`) is transferred; the chunk size (`grain`) is given as input parameter. The program is compiled with `-O0` flag, avoiding optimizations. The experiment is repeated several times, to have better estimations.

```
emlStart();
cudaEventRecord(start, 0);
for(unsigned long i = 0; i < TOTALSIZE;  i+=grain) {
  cudaMemcpy(&dev_input[i], &input[i], sizeof(char) * grain,
      cudaMemcpyHostToDevice);
}
/*Force synchronization, avoid optimizations*/
input[rand()%TOTALSIZE] = 'c';
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
emlStop(HostToDevice);
//empty kernel, returns immediately.
kernel<<<1, 1>>>(dev_input, dev_output);
emlStart();
cudaEventRecord(start, 0);
for(unsigned long i = 0; i < TOTALSIZE; i+=grain) {
  cudaMemcpy(&output[i], &dev_output[i], sizeof(char)*grain,
      cudaMemcpyDeviceToHost);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
emlStop(DevicetoHost);
```

Listing 4.6: Code used to monitor the energy (power × time) of data motion between host and device memory, and viceversa.

Host to device transfer, non-pinned memory

At high level, we see that the cost of moving data depends on the number of calls to the `cudaMemcpy` runtime function, as we can see in Figure 4.32. The Figure shows the average time required to perform the movement of 1GB of data, depending on the size of the transferred chunk. We see that for smaller chunks, the time will increase, as will do energy, that can be seen in 4.34.

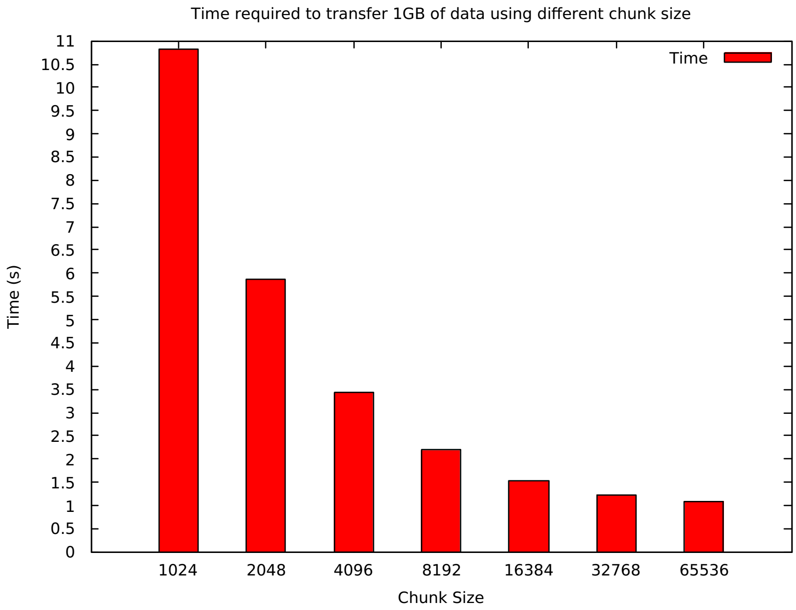


Figure 4.32: Average time for transferring 1GB of data from non-pinned memory from host to device.

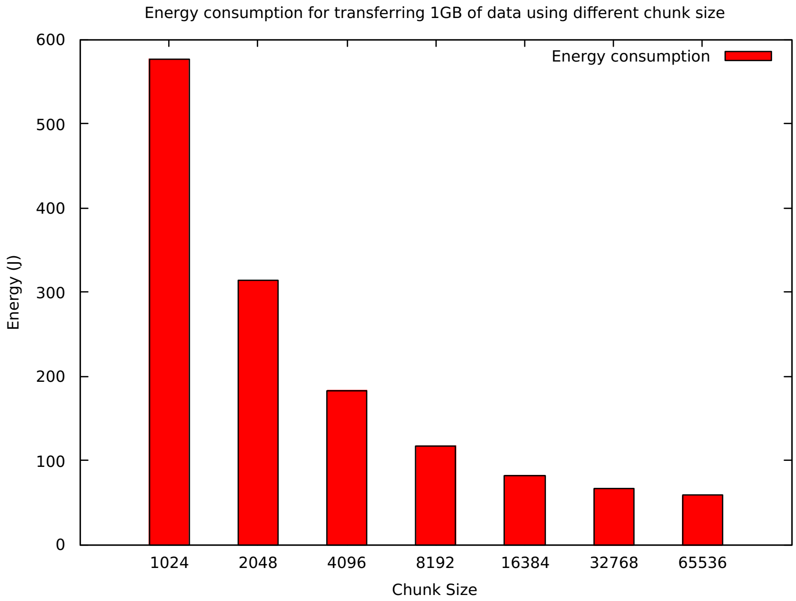


Figure 4.33: Average energy time for transferring 1GB of data from non-pinned memory from host to device.

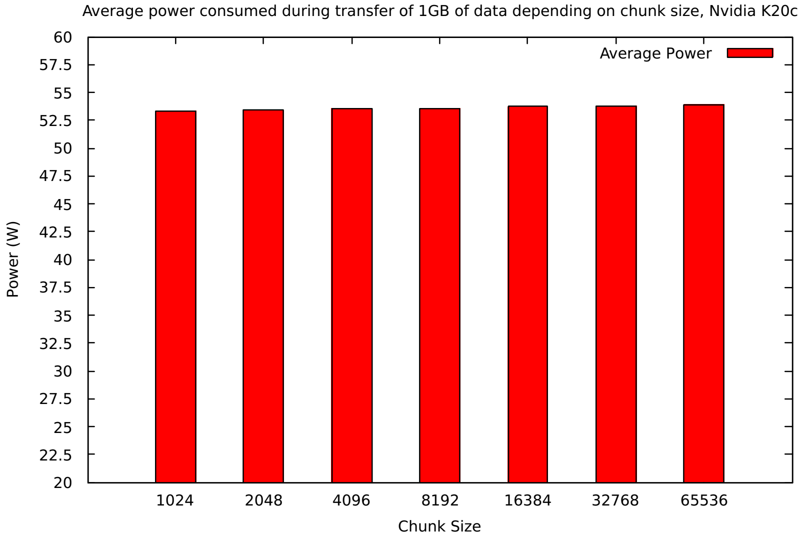


Figure 4.34: Average power time for transferring 1GB of data from non-pinned memory from host to device.

The profile of the instantaneous power for the case of transfers of 1024, 2048 and 4096 bytes can be seen in Figure 4.35. Once again we can observe an initialization phase, followed by a steady state characterized by frequent oscillations in power requirement. Intuitively, the amount of variation does not depend on the chunk size, hence it does not depend on the number of calls and could be caused by the behaviour of the bus. However the maximum instantaneous power consumption grows. When the chunk size grows, we can always observe a growth in the peak power consumption. As an example, when transferring 10GB (the maximum available memory on the device) in a unique chunk, we observe a peak power of $70.22W$, which is the maximum power consumption of data transfers.

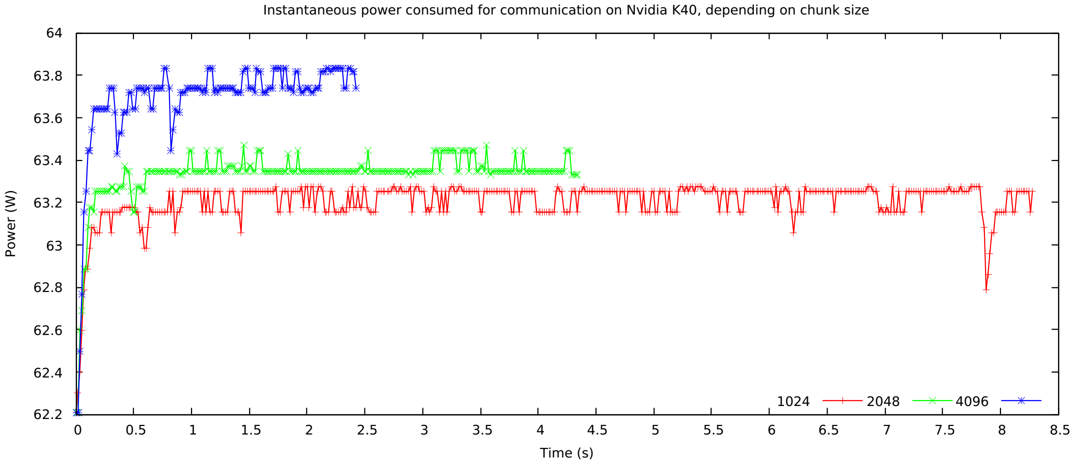


Figure 4.35: Instantaneous power consumption for transferring 1GB using different chunk sizes.

In our opinion, the reason for this behaviour lies in the fact that resources are exploited for longer, and consequently enter their steady state. On the other hand, when we have a sequence of consecutive transfer, resources will be deactivated and re-activated very rapidly, making almost impossible to reach the steady state. However, the average power does not grow as much as the instantaneous power maximum, as for bigger chunks the transient state will have an higher impact.

Host to device transfer, pinned memory

When we consider the case of pinned memory, as we would expect the time and energy requirements are smaller. Consider 4.37 and 4.36. We can see that in both cases the required time and energy is smaller. This is because less movements of data are performed in the host memory, hence allowing to perform faster copies. In fact, the difference in average power consumption with respect to non pinned memory is negligible, as it can be seen in Figure 4.38.

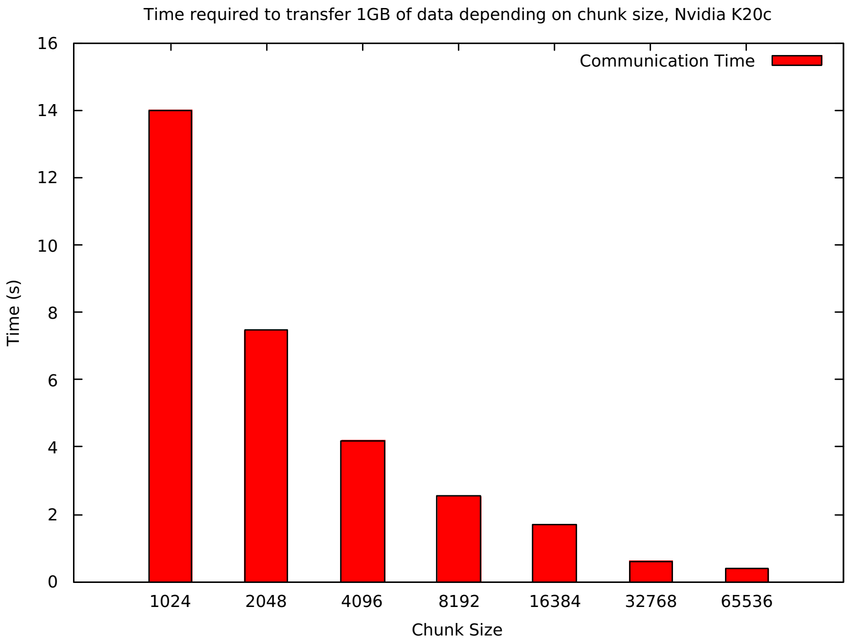


Figure 4.36: Average time required to transfer from host to device, with pinned memory

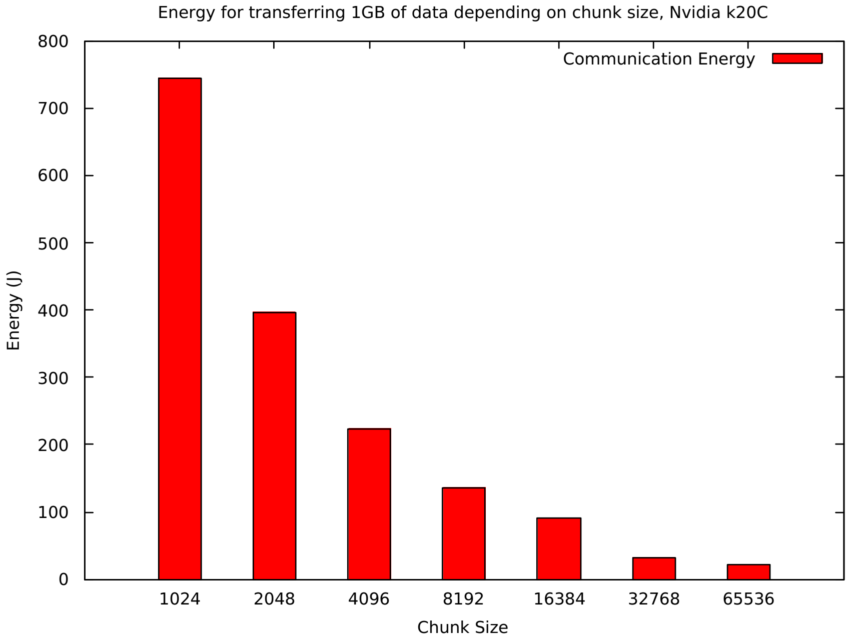


Figure 4.37: Average energy cost of transfer from host to device, with pinned memory

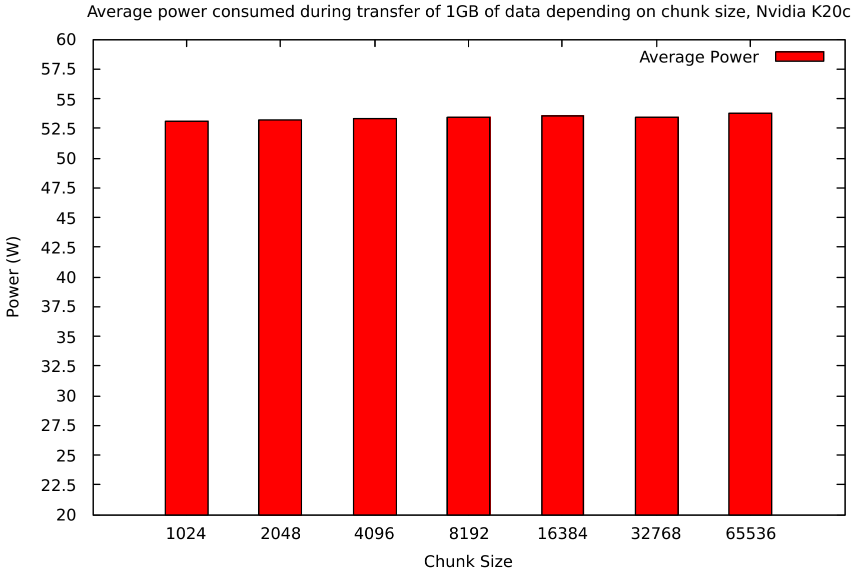


Figure 4.38: Average power during data transfers from host to device, with pinned memory.

For what regards the instantaneous power, we see a more regular behaviour in 4.39, with less oscillations in all the cases. Also in this case, however, the maximum power grows proportional to the chunk size.

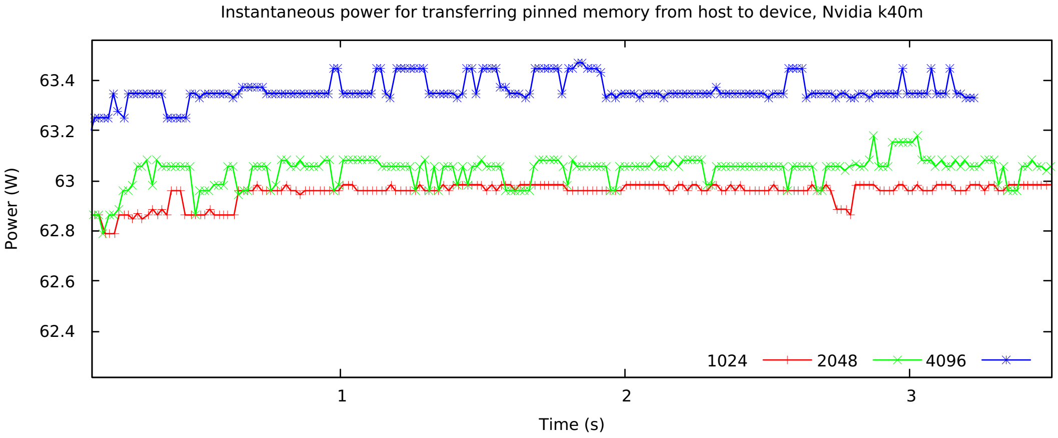


Figure 4.39: Intantaneous power consumption for transferring 1GB of pinned memory from host to device, using different chunk sizes

Device to host transfer, non-pinned memory

In this case we experience a degradation with respect to the communication performed in the opposite direction. In Figure 4.40 we can see the difference in time between transfers from device to host memory (in red) and in the opposite direction (green). In this case and also in the case of Figure 4.41, we see that moving memory from the device to the host is much more costly from time and energy standpoint. For what regards the average power, we see that it is comparable from figure 4.42.

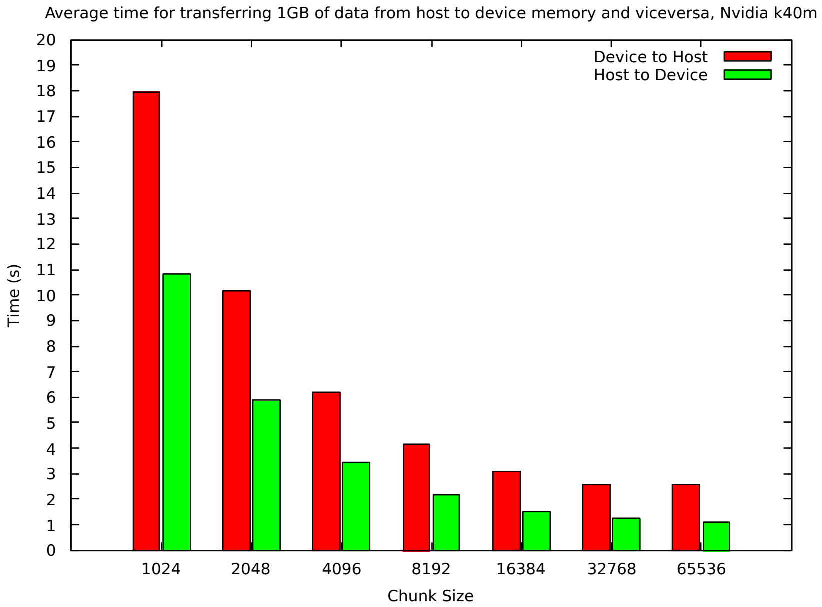


Figure 4.40: Comparison of the time required to transfer 1GB of data (with different chunks) from device to host and viceversa.

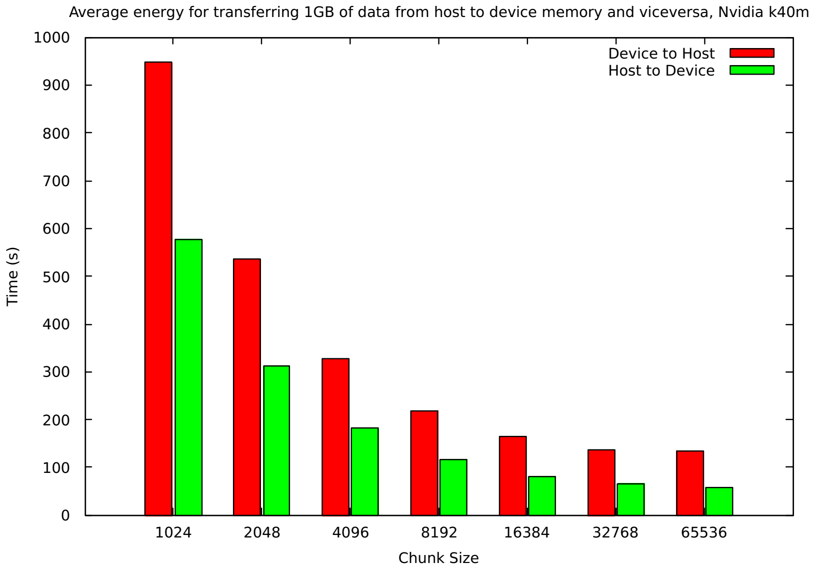


Figure 4.41: Comparison of the energy required to transfer 1GB of data (with different chunks) from device to host and viceversa.

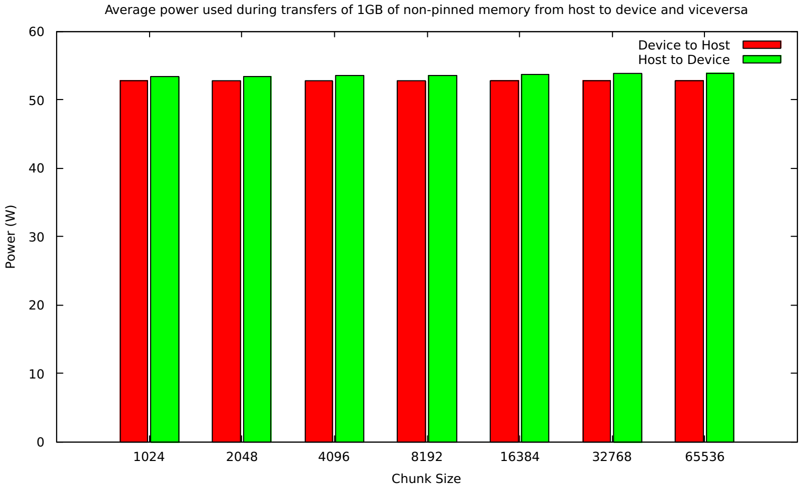


Figure 4.42: Average power consumption during transfers of data from host to device memory and viceversa

Device to host transfer, pinned memory

In case of pinned memory transfers, the difference between host and device transfers are quite smaller. In fact, they are so small they could even be attributed to small variations in the system. In Figures 4.43, 4.44 and 4.45 we can see that in this case there's a small difference in resources utilization with respect to the case of non-pinned memory.

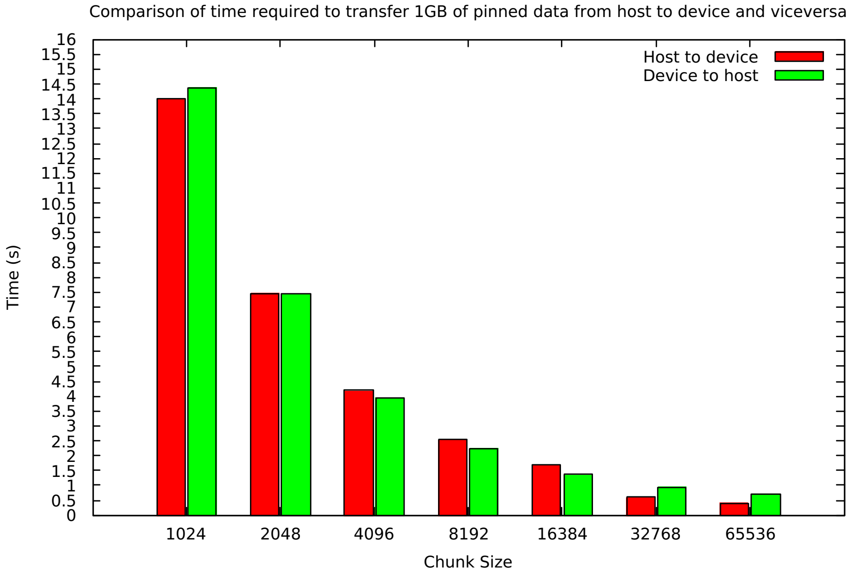


Figure 4.43: Comparison of time required to transfer 1GB of pinned data from host to device and viceversa, depending on chunk size.

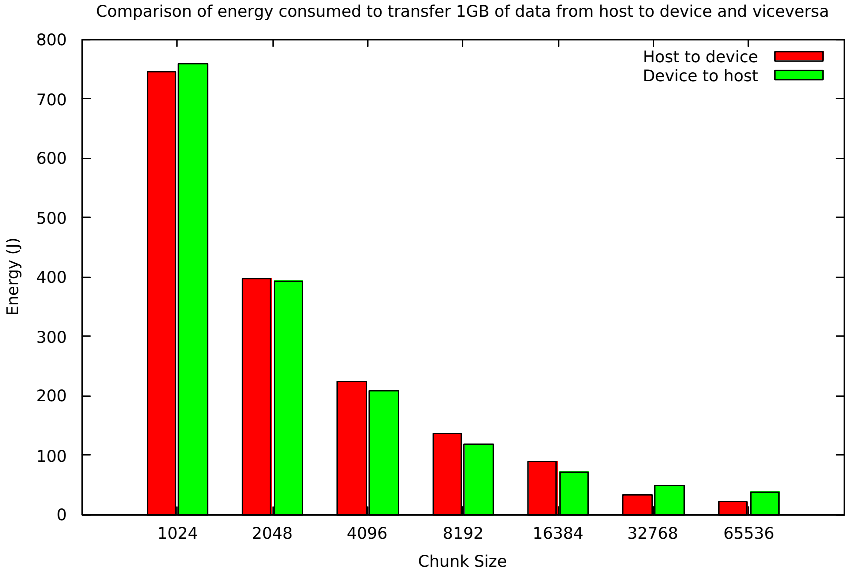


Figure 4.44: Comparison of energy consumed to transfer 1GB of pinned data from host to device and viceversa, depending on chunk size.

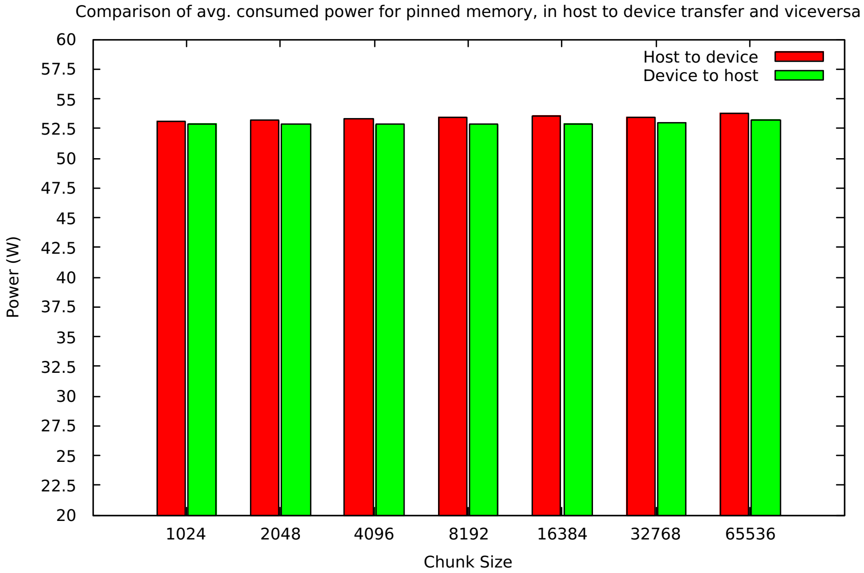


Figure 4.45: Comparison of average power consumed to transfer 1GB of pinned data from host to device and viceversa, depending on chunk size.

Other communication mechanisms

While the cost of data motion can be partially alleviated by using *cudaStreams*, that allows, in architecture with at least two copy engines, to execute kernels in parallel with data movement, we will neglect this mechanism for the moment being, as its analysis is more complex. Another mechanism to avoid explicit copying is *zero-copy memory*. Through a specific call - `cudaHostAlloc` [CGM14]- the memory can be allocated in host memory, but is also placed in the device address space, thanks to Nvidia *Unified Virtual Addressing*. This allows to have identical pointers for memory allocated with `cudaHostAlloc`. The data used by the threads is still moved through the PCI express, but this happens behind the curtains. With this mechanism, the kernel performance might suffer a great deal of degradation, as data is resident in host memory and is migrated to the device using data locality principles.

Another mechanism, easing further the process of accelerating applications with CUDA is *Unified Memory* [Har13]. Unified memory is a mechanism built on top of Unified Virtual Addressing. In this case, the run-time automatically migrates data from the host to the device resident DRAM and viceversa, without explicit transfers coded by the programmer. The vision proposed by Nvidia with Unified Memory is depicted in Figure 4.46.

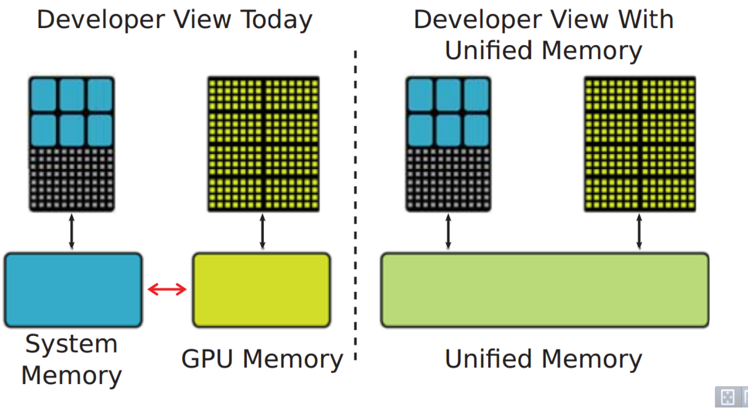


Figure 4.46: Programmer's vision of the new CUDA Unified Memory. Image from [LZCH14]

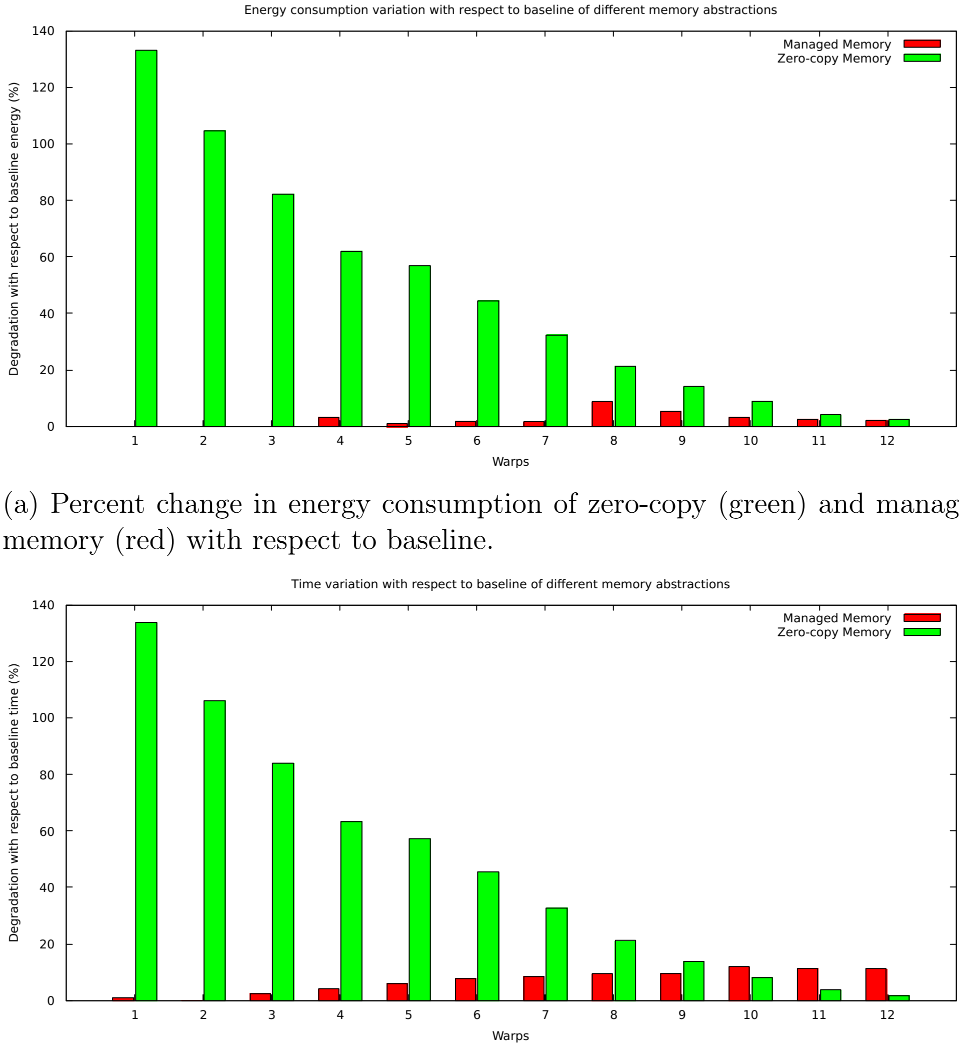
We performed experiments using zero-copy and managed memory, to understand the performance and energy degradation of these mechanisms. To do so, Listing 4.7 has been taken as baseline. In the case of zero-copy memory, instead, we only monitored an appropriately synchronized kernel execution, where data was allocated previously using the appropriate call. The same has been done for the case of managed memory.

```
template<typename TYPE>
__global__ void kernel(TYPE *in_a,TYPE *in_b, TYPE *out_c, size_t
    size) {
    unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while(tid < size) {
      out_c[tid] += in_a[tid] + (in_b[tid]);
      tid += blockDim.x * gridDim.x;
    }
}


int main() {
...
 //allocate pinned host memory
 cudaMallocHost(&a, size*sizeof(float));
 cudaMallocHost(&b, size*sizeof(float));
 cudaMallocHost(&c, size*sizeof(float));
 //allocate device memory
 cudaMalloc(&dev_a, size*sizeof(float));
 cudaMalloc(&dev_b, size*sizeof(float));
 cudaMalloc(&dev_c, size*sizeof(float));
 //initialization
 ...
 emlStart(); //start monitoring
  cudaMemcpy(dev_a,a,size*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b,b,size*sizeof(float),cudaMemcpyHostToDevice);
  kernel<float><<<blocks,threads>>>(dev_a, dev_b, dev_c, size);
  cudaMemcpy(c,dev_c,size*sizeof(float),cudaMemcpyDeviceToHost);
  emlStop(computation); //end monitoring
...
}
```

Listing 4.7: Kernel and monitoring code for the baseline computation, with explicitly managed communication

From the plot in 4.47, we can see the difference from the baseline for different number of warps within a single block, using zero-copy and managed memory.



(a) Percent change in energy consumption of zero-copy (green) and managed memory (red) with respect to baseline.

(b) Percent change in time of zero-copy (green) and managed memory (red) with respect to baseline.

Figure 4.47: Variation with respect to baseline of different memory abstractions.

Conclusions

The cost of communication can be quite high and provides a huge optimization space: sending more data all together is essential to save energy and time. We have seen how different memory mechanisms and communication directions mainly have an impact in terms of the time, while power remains almost constant. This provides an insight on how the communication is performed: the resources used are probably the same, and with similar utilization factors. The main difference would thus be in how efficiently such resources are used.

Finally, we tested different, more high-level communication mechanisms. We can see that the degradation in terms of energy consumption and time using managed memory mechanisms is small enough to consider the feasibility of this approach when programming parallel heterogeneous architectures. On the other hand, using zero-copy memory could be quite advantageous in terms of time for high parallelism degree and elevated (spatial) data locality.

The availability of this kind of mechanism would be of great advantage for real implementation of an energy-aware behavioural skeleton operating on a heterogeneous system.

4.3.3 High-level experiments

In the high-level experiments, rather than targeting a single component of the system, we want to target its behaviour as a whole. Since we are mainly interested in how map computations behave, we should test if there's a common pattern considering:

1. different data types;
2. different data structures;
3. different map functions.

The experiments are performed by calling a kernel, surrounded by proper calls to the monitoring library. We ensure that the kernel is finished by using either `cudaDeviceSynchronize()` or surrounding the kernel with proper CUDA events calls. Every execution of the program is preceded by a period of about 60 seconds, during which the GPU is left idle. This allows to avoid effects of previous computations to affect new measurements. In fact, we can see experimentally that by calling kernels continuously, the average consumed energy grows.

For practical purposes, most of the results shown have been performed making the data size on which the kernel operates parametric in the number of workers. This is because with huge problems, testing with a small parallelism degree would require a huge, impractical time. On the other hand, if we decrease the size, it could be that using the device at its full power would return less trustworthy results, as the monitoring interval would be too small.

4.4 Model Individuation

We describe the cost of offloading a map computation from the host to a GPU coprocessor as made of two parts: the former is the communication cost (involving both energy and time), while the latter is the cost of performing the actual computation. The model implicitly assumes that the computation can be carried on on the device, hence we don't need high precision and the problem is small enough for its data to fit completely in the device memory. Another implicit assumption of our model is that CUDA cores and streaming multiprocessors can be used as a 2D array of resources. We neglect the three-dimensional abstraction proposed in CUDA and just see the streaming multi-processors (addressed in terms of blocks) as a linear array of processors, each one comprising another linear string of CUDA processors. For the CUDA cores, we also consider the fact that threads are hardware managed, and can hence be scheduled in a higher number with respect to the effective number of physical resources without too much overhead.

4.4.1 Estimating power consumption using regression

Regression using samples taken over all values

To understand power consumption, we start by modelling a kernel with very regular behaviour, both in terms of power, energy and time. This is the *sleep* kernel, explained in Section 4.3.1. Such computation will constitute the baseline over which we build our model. Once again, fixed a certain number of blocks, the power grows sub-linearly in the number of warps assigned for each one.

If we want to understand the appropriate function to model the behaviour, we can use polynomial regression, fixing the degree of the polynomial function to two. We will achieve a regression function in the form $y = \beta_2 x^2 + \beta_1 x^1 + \beta_0$ in case β_2, coefficient of the 2nd-degree term of the model, is small enough, we can move back to a simple linear regression.

What we seek to model is the power of a *given* computation C, depending on the number of blocks and warps. The number of blocks b and of warps for each of them $w \in [1, 32]$ are hence the explanatory variables, while the response variable is $P_C(b, w)$. We try to find an estimator in the form $\hat{P}_C(b, w)$ that gives a good approximation of the consumed power given a certain parallelism degree.

Using:

- least squares to find a polynomial regression function for the sleep kernel,
- a fixed number of blocks and
- using the square root of the number of warps as explanatory variable

we obtain $\hat{P}_{sleep}(1, w) = -3.1696 \times 10^{-4} \times w + 0.3738 \times \sqrt{w} + 57.4374$. With this predictor, RSS (residual sum of squares) is 5.1243×10^{-2}, while the total sum of squares is $\sim 11,0918 \times 10^4$.

The coefficient of determination R^2 is hence 0.999999538, thus the model fits the data almost completely. In case we force β_2 to be zero, R^2 becomes 0.9999995273. Neglecting the second degree component of the polynomial regression function does not affect the precision of the estimation appreciably. The same applies for different number of blocks, varying from 1 to 13 (the number of physical streaming multi-processors).

If instead of the square root we use the logarithm, the residual sum of squares almost doubles with respect to the previous case. While R^2 will remain close to one, we will have a bigger error, so we excluded the logarithm. Finally, by using warps linearly, we achieve a residual sum of squares equal to $1.3579 \times 10^{-}1$, which grows in the number of used blocks. This justifies, together with the intuition, that the growth in average power consumption depends on the square root of the number of warps used within a block. Using regression, we estimate power consumption of a computation C as follows:

$$\hat{P}_C(b, w) = \beta_0^C(b) + \beta_1^C(b) \times \sqrt{w}$$

In Figure 4.48, we show the error in estimating power consumption with regression for the sleep computation. The error will be defined as:

$$\epsilon_C(b, w) = \frac{\hat{P}_C(b, w) - P_C(b, w)}{P_C(b, w)}$$

this error formulation will be used for the rest of this work. When we show an empirical frequency, we use the method proposed in [Gal02]: we plot an histogram of the values assumed by the random variable on the sampled values. To do so, we divide the space between the minimum and maximum value into a fixed number k of disjunct intervals $[b_0, b_1), [b_1, b_2), \ldots, [b_{k-1}, b_k)$. For each interval we count the number of elements of the sample within it and divide it by the total number of samples.

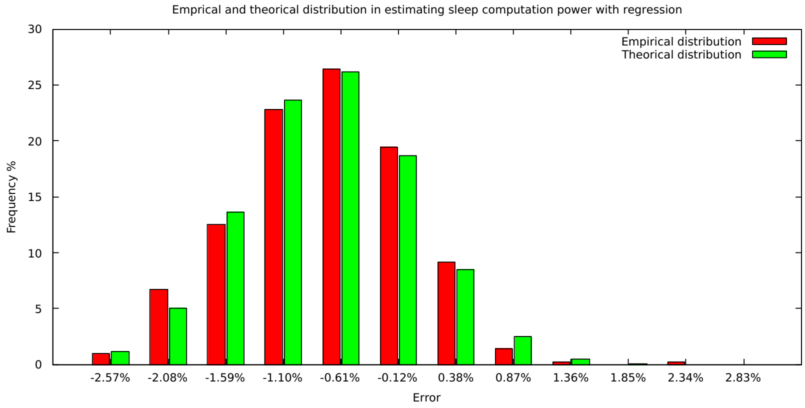


Figure 4.48: Empirical frequency (red) of error when estimating power using $\beta_0 + \beta_1 \times \sqrt{w}$, against the $N(0, \sigma)$ (green) distribution required to perform regression modelling. The average error in the empirical case is -0.51%, the standard deviation is ~ 0.0072.

We want to show that regression can be used to model successfully also more complex computations: consider the case of a vector addition, executed entirely on the GPU. The vector addition is written so that every thread accesses *coalesced* (i.e. with consecutive locations within a warp) memory.

In Figure 4.49 we show the effective average power (calculated over 5 executions) varying the number of warps within 4 blocks, against the regression model; in Figure 4.50, instead, we show the distribution of the error for linear regression on said computation.

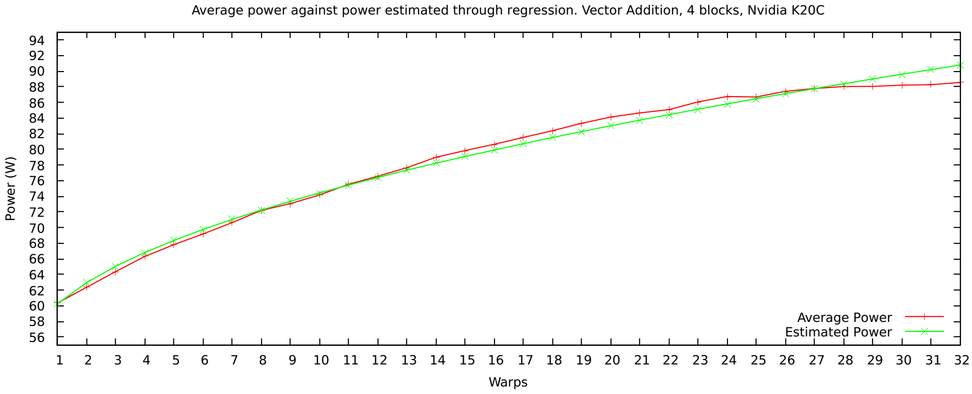


Figure 4.49: Average power for vector addition on the GPU against the power estimated through regression. 4 blocks, variable number of warps. Nvidia K20C

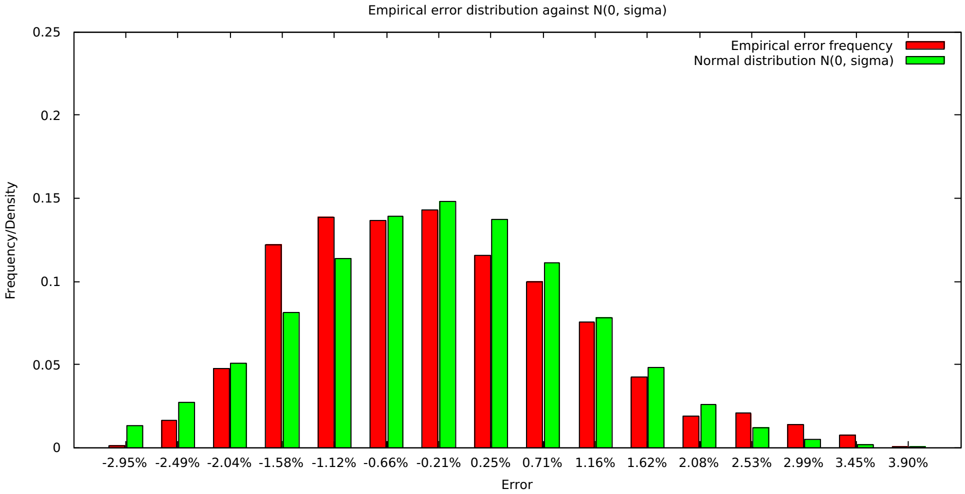


Figure 4.50: Distribution of the error with linear regression against $N(0, \sigma)$.

The low error achieved with this approach demonstrates that:

- the explanatory variables chosen do relate to the power
- this approach can be used with success across different computations

Regresssion with sub-sampling

It is not necessary to have a precise estimation to test the average power of the computation for any value of w and any value of b.

In fact, the regression parameters can be estimated even with less samples. If we sub-sample the power considering only the case for $w = 1, 6, 12, 18, 24, 30$, we achieve the error profile visible in Figure 4.51. The difference between estimated power and effective does not exceed 2% in absolute value in above the 91% of the cases.

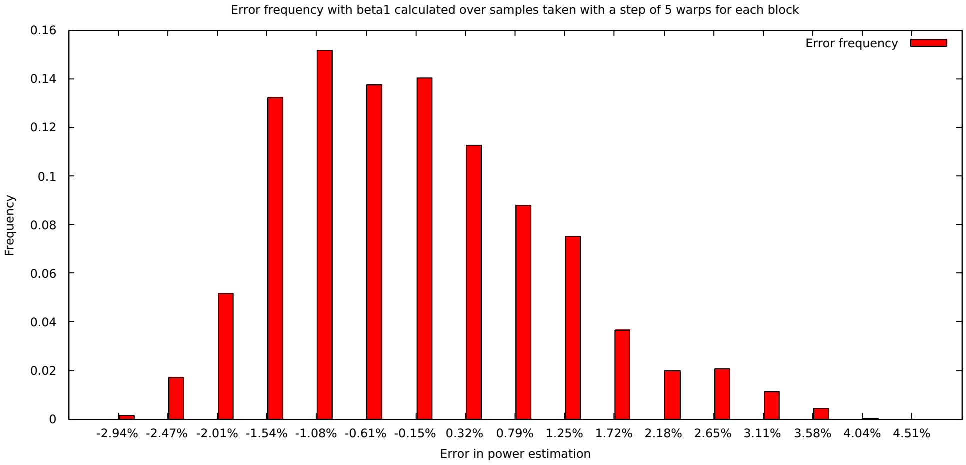


Figure 4.51: Error profile sampling the power only for some values of w for each block.

In the case in which we only consider 3 samples per block ($w = 1, 18, 30$, we achieve a maximum error of 3.38%, a minimum of -3.21% and an average of -0.25%. The probability that the error is less than 2% in absolute value is above 0.85.

Estimating the regression parameters

Until now the regression parameters used where function of the number of blocks used in the computation. Power was predicted as $\hat{P}_C(b, w) = \beta_0(b) + \beta_1(b)\sqrt{w}$, where the values of β where dependant on the block. We want to find a way to estimate the regression parameters without need to calculate them for any possible value of b. More formally, we wish to find a function $\tilde{\beta}(b)$ allowing to estimate the parameters used to estimate power. To do so, it is possible to perform a sort of "second order" regression.

In Table 4.2 we show the values of $\beta_1(b)$ obtained for different number of blocks in the sleep computation. The values of $\beta_1(b)$ (sampled over all values of w) for said computation are visible in Figure 4.52 (in red), against the regression function (in green). We see that they almost completely overlap, hence allowing to estimate the value of the regression parameter without having to analyse all of the blocks.

Blocks b	$\beta_1(b)$
1	0.3715
2	0.6870
3	1.0340
4	1.3725
5	1.7195
6	2.0568
7	2.4074
8	2.7522
9	3.1070
10	3.4508
11	3.7878
12	4.1219
13	4.4831

Table 4.2: Regression parameters for estimating the power depending on the square root of the warps, depending on the number of blocks.

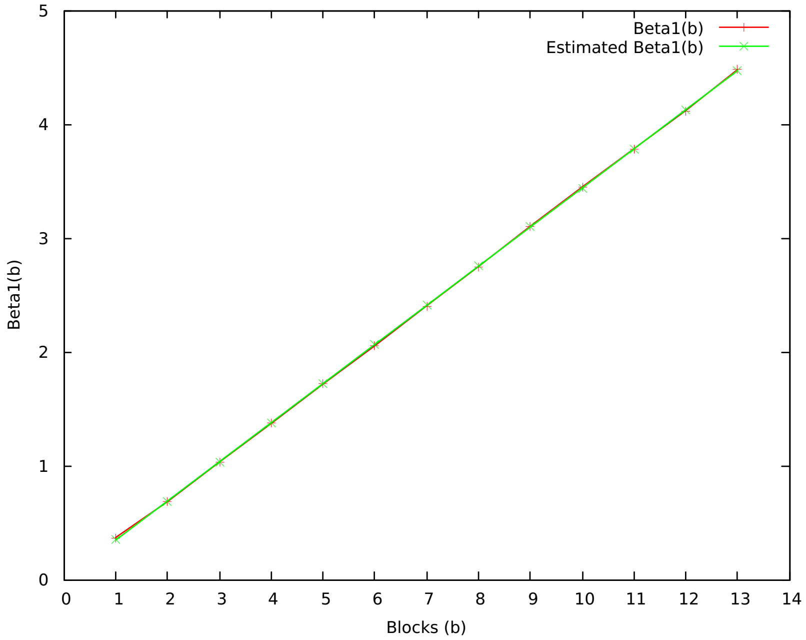


Figure 4.52: $\beta_1(b)$ varying the number of blocks. The green line is the linear interpolation performed over $\beta_1(b)$ using b as explanatory variable. $\beta_1(b) \approx 0.005 + 0.34 \times b$

Consider again the vector addition computation. Also in this case $\beta_1(b)$ has a linear behaviour. By interpolation, we can estimate $\tilde{\beta}_1^{vector\_add}(b) = 1.15854b + 1.72193$. By using this approximation, we achieve the error profile visible in Figure 4.53. We can see that the probability that the error in absolute value is less than 2% exceeds the 85%.

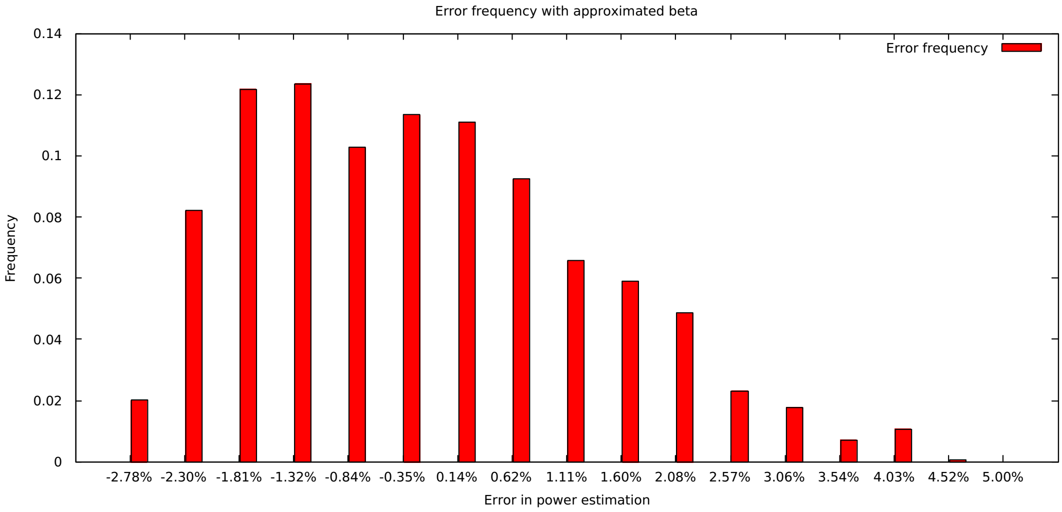


Figure 4.53: Error frequency for vector addition with approximated $\beta_1^{vector\_add}(b)$.

If instead we use $\tilde{\beta}_1^{vector\_add}(b)$, achieved in the case where we consider only 3 samples for block, we achieve an error below 4% in absolute value, with an average of -0.28%.

Similar error profiles are achievable by sampling less values of $\beta_1^{vector\_add}$. As an example, by calculating the regression only for $b \in 1, 2, 6, 10, 13$ we achieve a maximum error of 4.51%, a minimum of -3.37% and an average of -0.43%. The probability that the error is between -2% and $+2\%$ exceeds the 77%, while we reach 90% for error whose absolute value is less than 2.5%.

If instead we perform the regression only on b equal to 1 and to 13, we have a maximum error in power estimation of the 3.89% and a minimum of -4.53%, with the 61% of errors not exceeding 2% in absolute value, and more than the 73% not exceeding 2.5%.

For calculating the previous figures, we were still assuming to be able to calculate $\beta_0^{vector\_add}$ precisely. Despite being the behaviour of this regression parameter less regular than in the previous case, we can use a similar approach on it. We show it for both the considered computation in Figures 4.54 and 4.55.

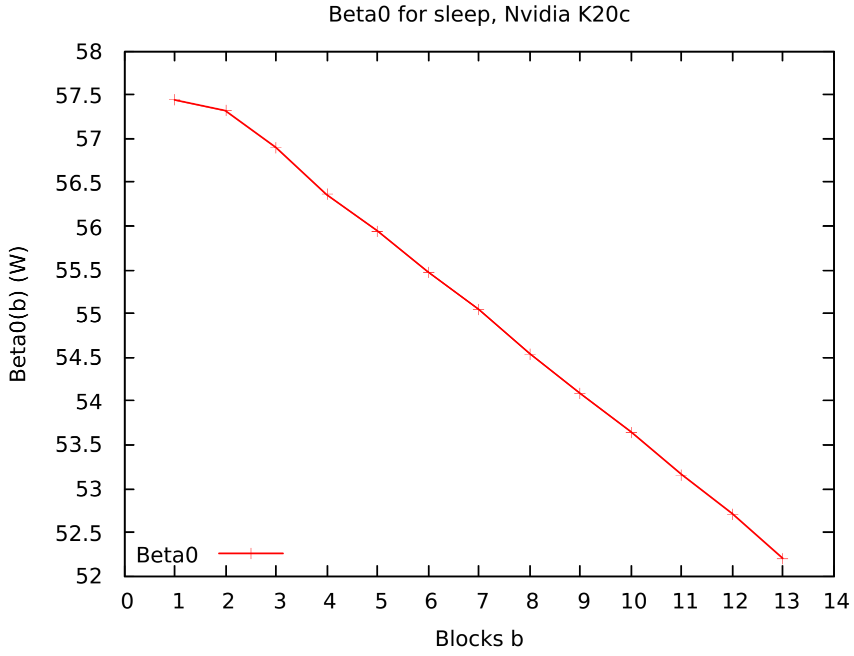


Figure 4.54: Regression parameter $\beta_0(b)$ varying b for sleep.

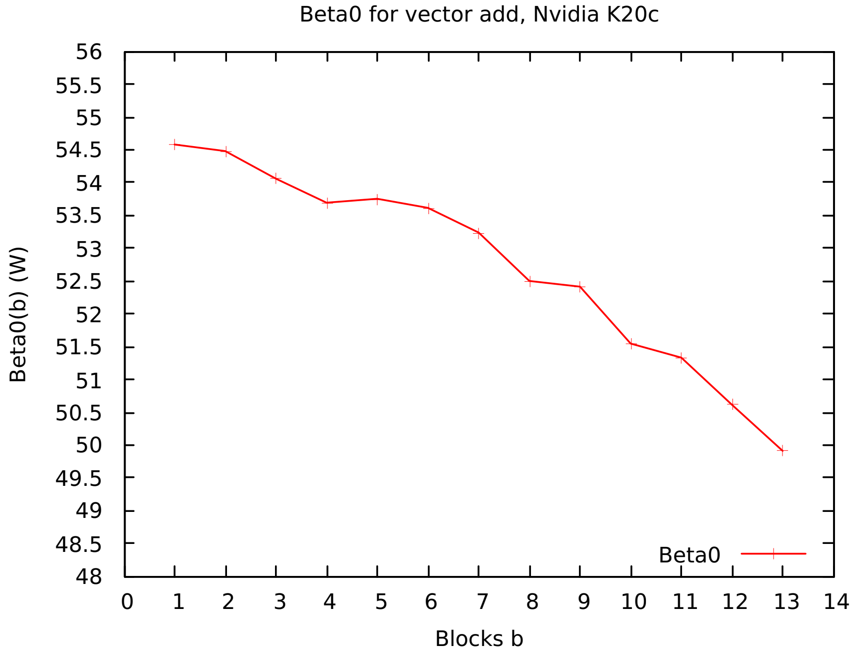


Figure 4.55: Regression parameter $\beta_0(b)$ varying b for vector add

If we estimate also $\beta_0(b)$ by regression on vector addition computation, the error increases notably: we have only the 70% of the error in $[-2.5\%, 2.5\%]$. The maximum error is 4.32% and the minimum is -4.61%; the average is -1.29%. In this case, however, estimating with less values of b affects the error less: for b in $\{1, 2, 6, 10, 13\}$ we have a very similar behaviour, with 68% of estimations with less than 2.5% difference from the sample. If, instead, we model $\beta_0(b)$ through regression sampling the values only with b equal to 1 and 13, we reach 59% of errors within the absolute value of 2.5%, a maximum error of 3.89% and a minimum of -5.16%. We see that in this case the degradation due to estimation of β_0 with less knowledge is less impairing on the precision of the model.

Finally, if we estimate $\beta_0(b)$ and $\beta_1(b)$ only using this second-order regression over $b = 1, 13$ and $w = 1, 18, 30$, we get the error profile visible in Figure 4.56.

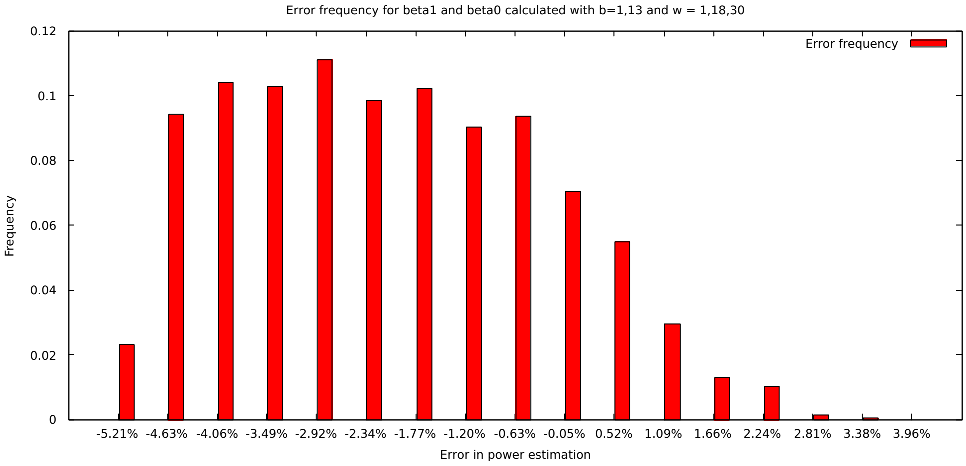


Figure 4.56: Error profile for regression model calculated sampling only 6 values.

While this error profile is in general worst than in more sampled cases, the maximum error (calculated over more than 2200 samples of power) barely exceeds 5% in absolute value.

Conclusions

The regression approach can be used safely whenever the number of calculations is enough to amortize the samples needed to estimate power consumption. The high precision achieved, and the elevated values of R^2 suggest that this way of modelling power consumption as a function of the parallelism degree is robust and can be applied safely for the targeted architecture and for map computations executed on it, regardless of the function applied to the elements.

The cost for preparing this model is not extremely high, as we only need to execute the computation with 6 different parameters (b, w) to estimate power with less than 5% error with high probability. More precise models can be achieved by measuring the power for more values of (b, w) on the computation. Possibly, rather than picking at random the execution parameters, we could decide to use complex algorithms (like *Simulated Annealing* [RN05]) to find a global optimum, building a power model in the meanwhile.

4.4.2 Heuristic model for estimating power consumption

Since we saw from the preliminary observations that, the power requirements of computations designed according to the same parallel pattern exhibit overall a similar power profile, we could think of using one of such computation as a metric, in order to "guess" how another one will behave.

The idea is to take a very stable, regular computation and monitor it carefully. Then, from its behaviour and the ratio between the figures achieved with other computation, we can try to estimate the values for other computations.

As a meter, we decided to use the sleep computation. The features of this computation are interesting, as it is very regular and we were not able to achieve similar results with other, more meaningful ones.

For modelling the power of this computation, we use an estimation drawn from the regression model, explained in the previous section. The power of the sleep computation can be expressed as follows:

$$P_{sleep}(b, w) = P_{base} + P_b \times b + (P_w + P'_w \times b) \times \sqrt{w} \tag{4.1}$$

The P parameters are expressed in Watt and depend on the architecture on which the computation is executed. The estimation of P_w and P'_w is provided through linear regression. In particular, we will have $P_w = 0.005W$ and $P'_w = 0.3438W$. These two parameters (in the regression part they summed up to $\beta_1^{sleep}(b)$) represent the cost in terms of power caused by the activation of additional warps. In particular, P'_w is multiplied by b, so that it reflects the incremented cost of spawning $w \times b$ warps.

The P_{base} represents a "minimal" power required to carry on a computation. It is comprehensive of the leakage power (estimated as explained in Section 4.3) and of the cost of actively using the board. P_b is meant to represents the cost of activating a streaming multiprocessor. P_{base} and P_b are calculated with regression over the detected values $P_{sleep}(b, 1)$, using b as explanatory variable. Hence they do not sum up to b, and give an upper estimation on the cost of the sleep function. We justify this choice with:

- for real computations C, $P_C(b, 1) > P_{sleep}(b, 1)$, because the warps will be used more;
- the curve described by $P_{sleep}(b, 1)$ is more regular;
- we verified experimentally that these two parameters were more robust for evaluating the heuristics explained in the following.

A first heuristic

The first idea could be to "measure" any computation C against the baseline, by using a metric in the form $m(C) = \frac{P_C(b,w)}{P_{sleep}(b,w)}$. However, this metric is not reliable as it is not "stable" across different (b, w) computations: hence, despite

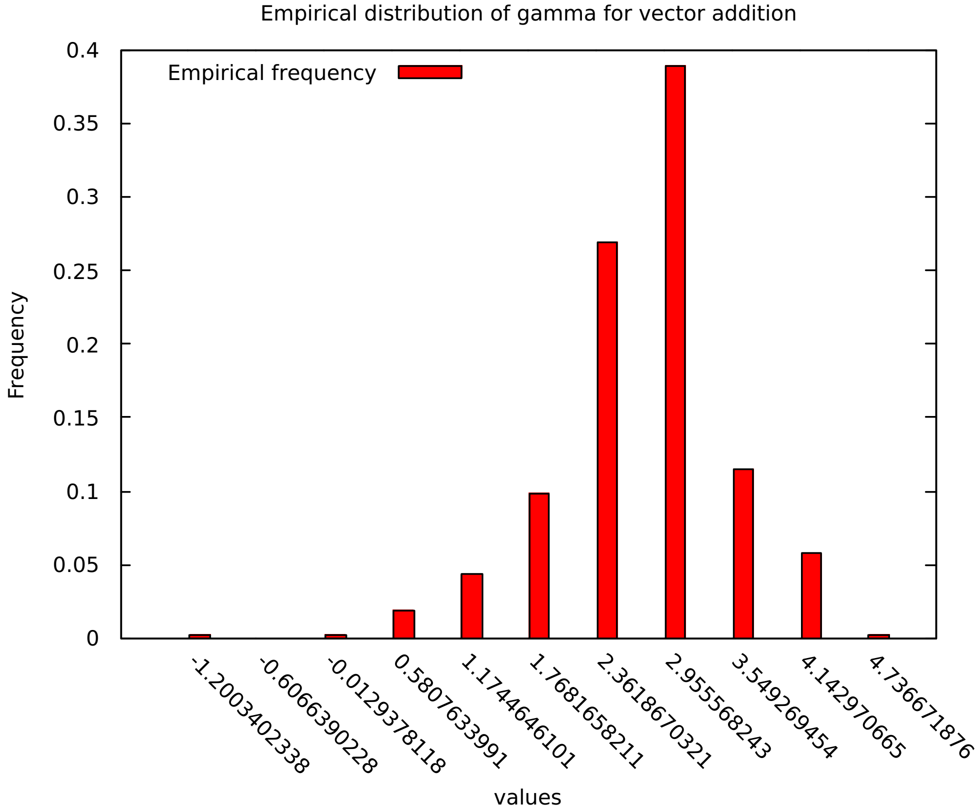


Figure 4.57: Distribution of $\gamma_{vector\_add}(b, w)$ for vector addition. Nvidia K20

the similar curve, a simple model estimating $P_C(b, w)$ as $P_{sleep}(b, w) \times m(C)$ would be unreliable.

Another approach could be to evaluate the ratio between the increment with respect to baseline of the two computations. The ratio of this approach is that the base cost will be almost the same for any computation and being the increment parametric we would experience difference in the ratio between the metre and the measure because of a different "ammortization" of the base cost. For this reason, we defined γ as follows:

$$\gamma_C(b, w) = \frac{P_C(b, w) - P_{base}}{P_{sleep}(b, w) - P_{base}}$$

For γ_{vector_add} we have an average of 2.98 over the whole computation. The variance is 0.58. We can see the distribution of the values of $\gamma_{vector\_add}(b, w)$ in 4.57. The average value of the metric will be referred as $\overline{\gamma}_{vector\_add}$ and is be defined as follows:

$$\overline{\gamma}_C = \mathbb{E}[\gamma_C(b, w)] = \frac{1}{\#blocks \times \#warps} \times \sum_{b=1}^{\#blocks} \sum_{w=1}^{\#warps} \gamma_C(b, w)$$

Despite the high concentration of values around the average, that would made us expect this metric to be quite precise in a high number of cases, in the tail the high difference with respect to the average calculated value will lead to elevated errors. In fact, if we try to estimate $P_{vector\_add}(b, w)$ as $P_{sleep}(b, w) \times \overline{\gamma_{vector\_add}}$, we achieve an average error of 8.05%, a maximum of 30.02% and a minimum of −3.40%, meaning that we almost always overestimate (significantly) the power. Since the value of $\gamma_{vector\_add}(b, w)$ changes significantly in w with fixed b, using a "local" measure for each b we verified experimentally that calculating $\overline{\gamma}_C$ locally to a block does not increase the precision of the predictor.

A more precise heuristic

Let us now evaluate a different metric used to predict the power. We define $\alpha_C(b, w)$ for a certain computation C as:

$$\alpha_C(b, w) = \frac{P_C(b, w) - P_{sleep}(b, 0)}{P_{sleep}(b, w) - P_{sleep}(b, 0)} \tag{4.2}$$

$\alpha_C(b, w)$ represents the ratio of the increment in power, due to an increase in the number of warps and blocks, between the analysed computation and the computation taken as a metre. It can be rewritten as follows:

$$\alpha_C(b, w) = \frac{P_C(b, w) - (P_{base} + P_b \times b)}{P_w + P'_w \times b \times \sqrt{w}}$$

We show the empirical distribution of values of $\alpha_{vector\_add}(b, w)$ in Figure 4.58.

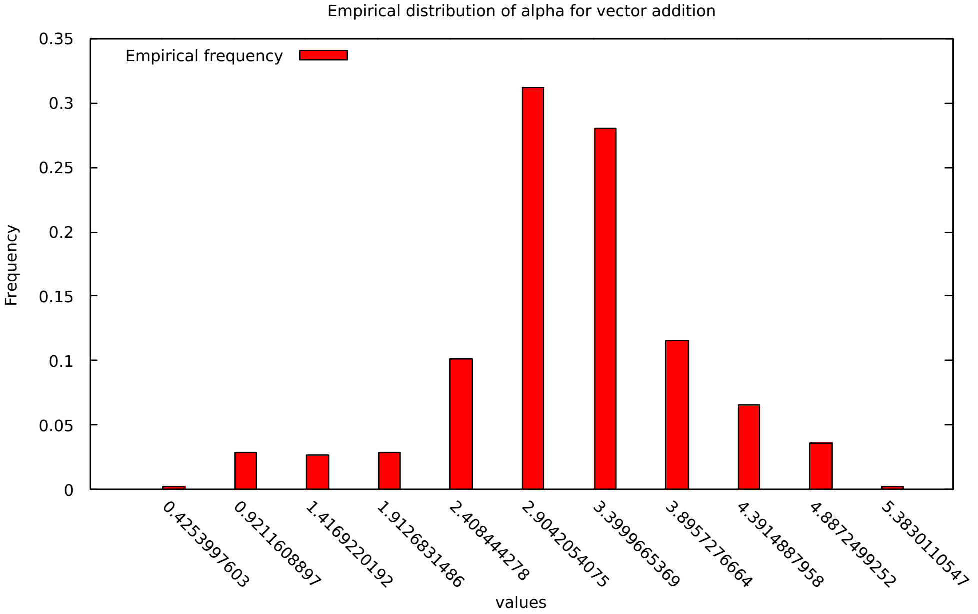


Figure 4.58: Distribution of the values of $\alpha_{vector\_add}(b, w)$.

Once again the values are in great part (more than the 87%) concentrated within ±1 from the average (~ 3.39). The average, indicated by $\overline{\alpha}_C$ is defined similarly to the case of γ_C.

Because of this concentration, we expect that:

- $\beta_0(b) + \beta_1^C(b)$ to be similar to $\beta_1^{sleep} \times \overline{\alpha}_C$;
- even in case the exact value of $\overline{\alpha}_C$ is not known, we can just select a random combination of (b, c) , sample the power for C and the value will be a good approximation of the real average;
- when the number of samples grow, the precision of the estimation will grow.

We can hence estimate the power of a certain computation C as follows:

$$\hat{P}_C(b, w) = P_{base} + (P_w + P'_w \times b) \times \overline{\alpha}_C \times \sqrt{w} \tag{4.3}$$

Even though it would be more logical to model $\hat{P}_C(b, w)$ taking into consideration the additional term $P_b \times b$, we see experimentally that this brings to an overestimation of the power, overall leading to worst results. We thought of two possible explanations for this behaviour:

- being the control logic very simple in all tested computations, the cost of activating an additional streaming multi processor *per se* (i.e. without considering the CUDA cores) is not as high as we would expect, or
- part of the constant cost of activating an additional streaming multiprocessor gets captured by α_C.

This issue should be investigated further. We use the heuristic proposed above as we verified experimentally that it bears to better results in the majority of cases.

By using Equation for estimating the power consumption of vector add computation, we achieve a maximum error (defined as $\frac{\hat{P}_{vector\_add}(b,w) - P_{vector\_add}(b,w)}{P_{vector\_add}(b,w)}$) of the 9.36%, a minimum of the −7.43% and an average of −1.67%. If we consider the absolute value of the error, the minimum is 0.02% and the average is 3.43%. The distribution (calculated over 5 samples for each valid value of b and w) of the error can be seen in Figure 4.59.

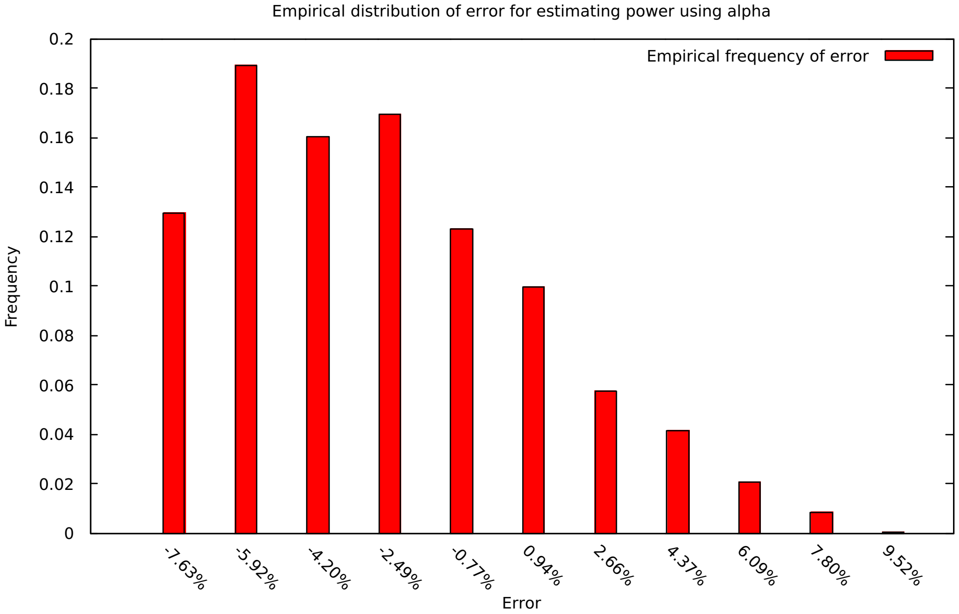


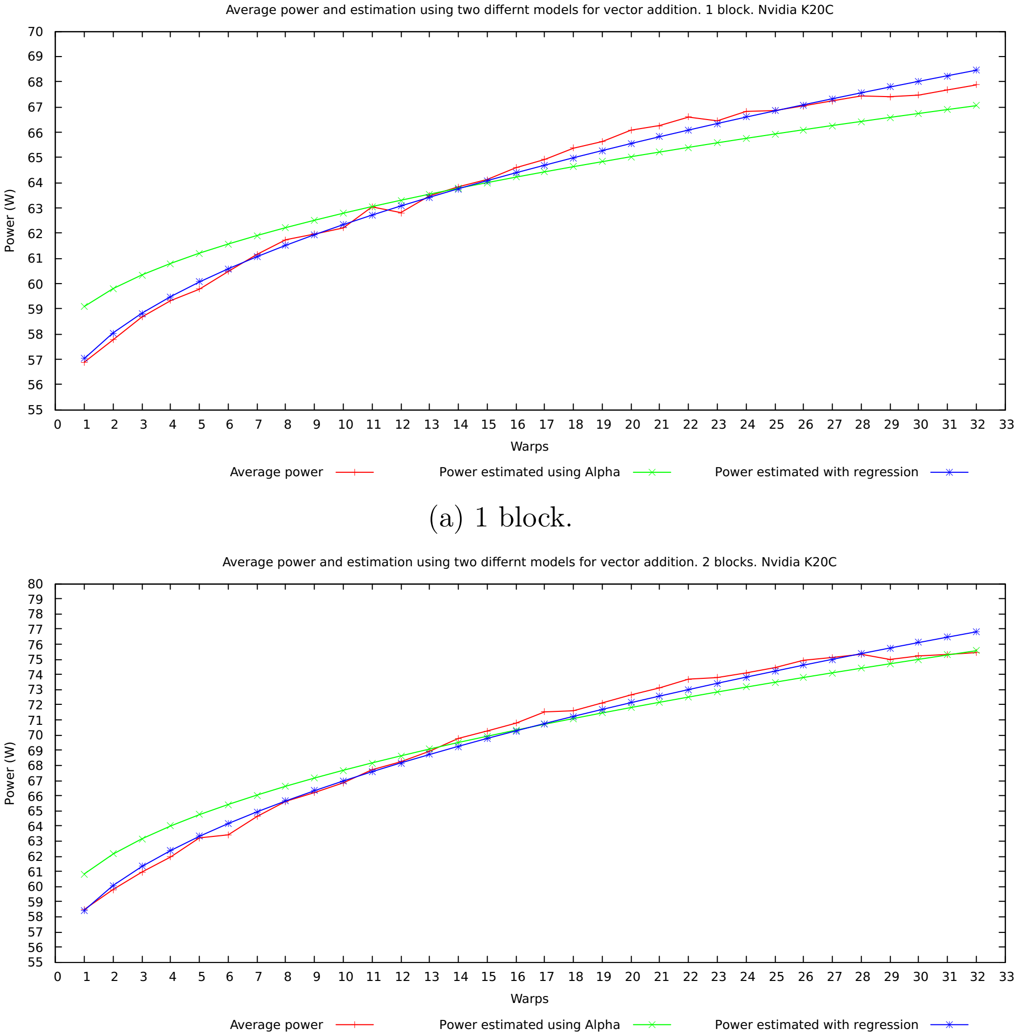
Figure 4.59: Distribution of error using $\overline{\alpha}_{vector\_add}$ to estimate power

In Figure 4.60a we show (in red) the detected average power for vector addittion, varying the number of warps in a single block. The blue line is the "ideal" model, using the parameters estimated using ordinary least squares. The green line represents the estimation of power achieved with the approximation using α_C within a single block.

The same figures are plotted for 2, 4, 6, 8, 12 blocks in Figures 4.60b, 4.61a, 4.61b, 4.62a, 4.62b, respectively.

Conclusions

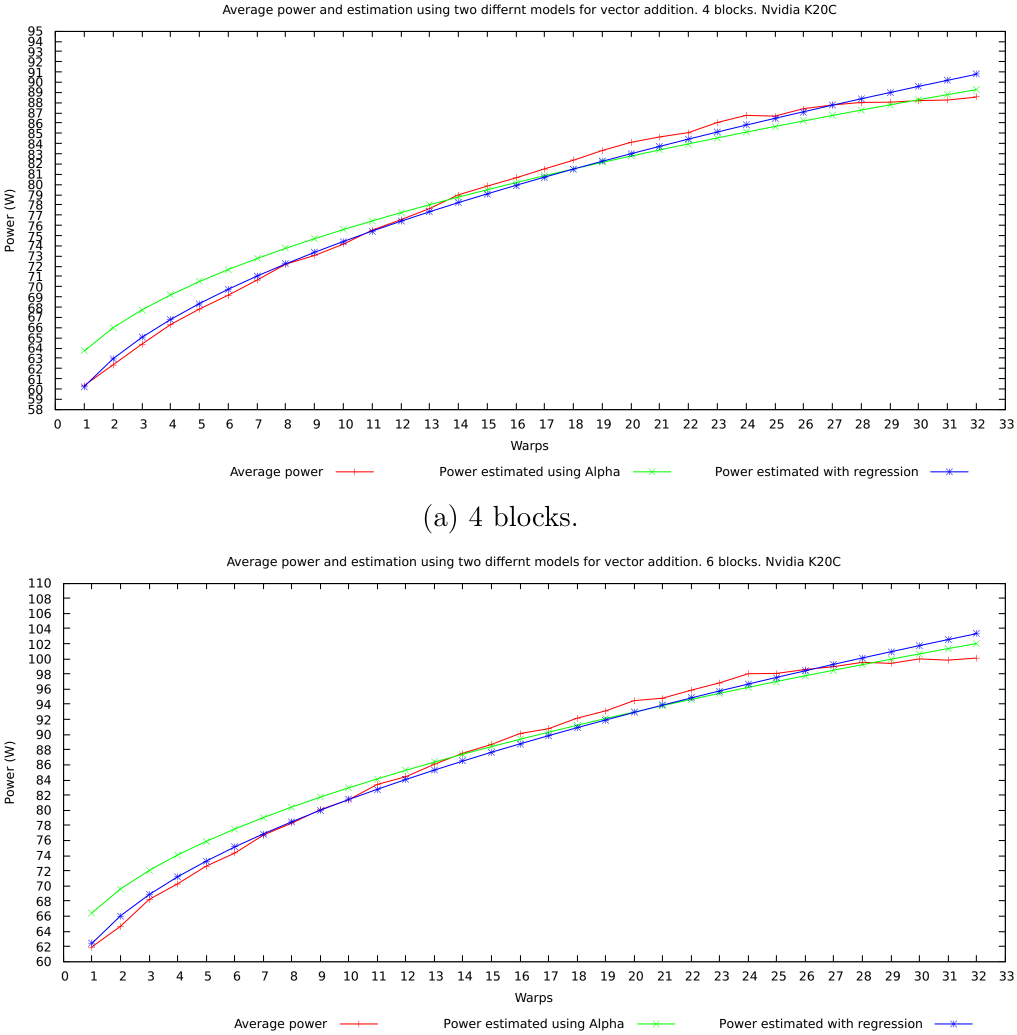
The heuristic proposed above and denoted by the greek letter α_C provides quite precise estimations in terms of power, even though it does not require as much samples as in the regression cases. While of course the error will be higher with respect to plain regression method, as explained in previous section, we will show in Section 4.5.2 that the intuition according to which a random sample is enough to provide a good estimation holds. This validates this approach in case of heterogeneous systems, where a rapid estimation of the consumed power for a computation should be provided in order to decide the parallelism degree and the grain to be scheduled to the GPU coprocessor.



(a) 1 block.

(b) 2 blocks.

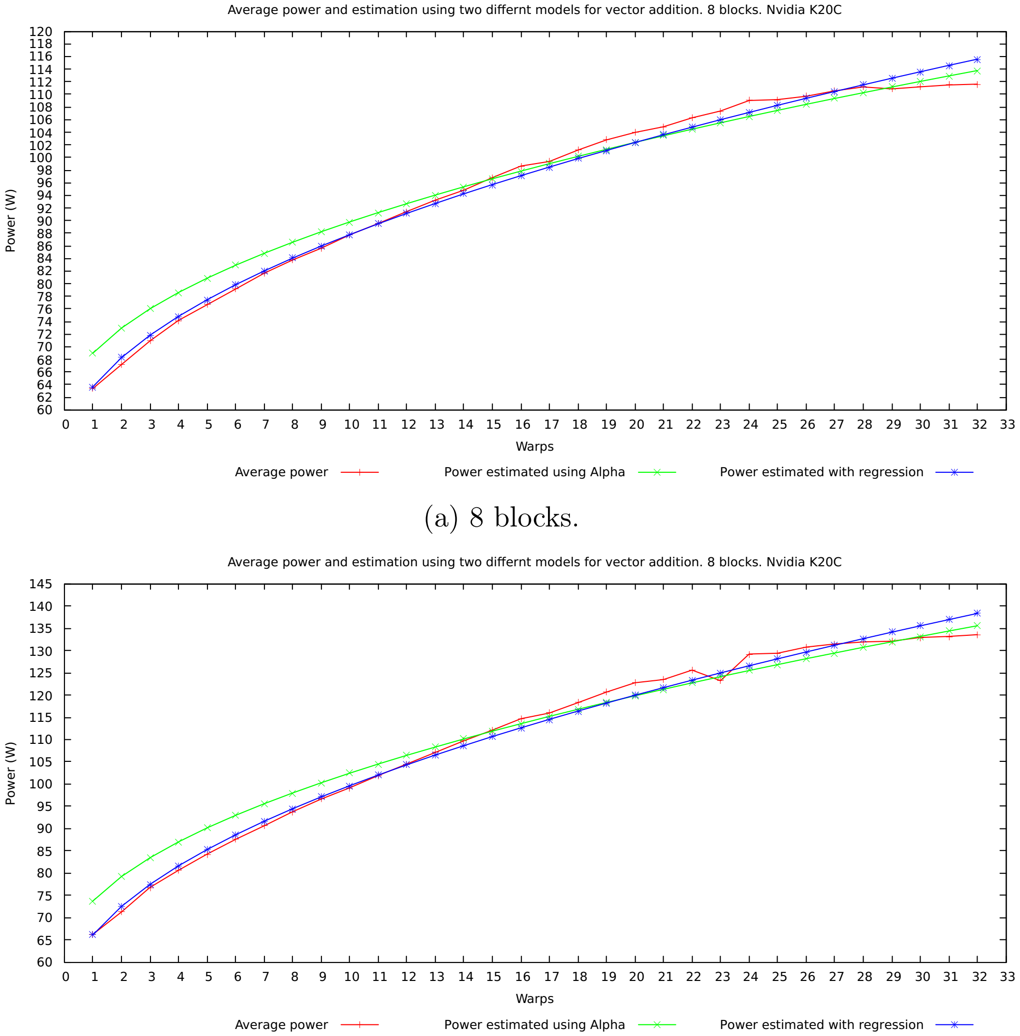
Figure 4.60: Effective power against regression against heuristic



(a) 4 blocks.

(b) 6 blocks.

Figure 4.61: Effective power against regression against heuristic



(a) 8 blocks.

(b) 12 blocks.

Figure 4.62: Effective power against regression against heuristic

4.4.3 Computation energetic model

As we know, the energy expended for a computation can be expressed as $E = T \times \overline{P}$, where T is the execution time and P is the average power consumed during the execution time by the device(s) on which the computation is carried on.

For this reason, we need a precise way to estimate the time to carry on a certain computation. Given a parallel exploitation patter, there are already stable cost model providing good approximations of the ideal completion time.

Consider the case in which the function f taken as parameter by the higher-order *map* function has an execution time equal , in average, to T_f. If each worker is given a partition of g elements to work on, the completion time for a map computation is [Van14]:

$$T_{map} = g \times T_f$$

to which the *scatter* (a concurrent activity responsible of dividing the data structure between the various workers) and *gather* (a concurrent activity responsible of recomposing the calculated results into a single data structure) time should be summed. In case we are in a shared memory environment, however, the cost of such components could be neglected.

The grain assigned to each worker will depend on the size of the data structure on which the map is applied. Suppose that the size is N, and that the number of execution units is n. In this case, the amount of work assigned to each worker will be $\sim N/n$; hence the map computation will have an execution time of:

$$T_{map}(n) = \left\lceil \frac{N}{n} \right\rceil T_f$$

In practice this will hold only if T_f has a very low variance. Assuming the size of the data structure is big enough and that the variance is low, we will have

$$T_{map}(n) = \frac{T_{map}(1)}{n}$$

Since on a GPU device the scheduling unit is a *warp*, up to 32 applications of f (the warp size for current day GPU architectures) will be executed in parallel by the workers, without additional latency with respect to using a single thread. Scheduling threads in batches of 32 is hence the most convenient alternative. We will have, within a single block, an ideal cost model in the form:

$$T_{map}(1, w) = \left\lceil \frac{N}{w \times warp\_size} \right\rceil \times T_f$$

However, the threads that can be scheduled do not map $1:1$ to execution units in all possible considerations. If we are using a single streaming multi-processor on a Kepler architecture, for example, only up to 6 warps will be executed in parallel, as there are 192 CUDA Cores. Even though scheduling more warps is in general advantageous, thanks to the long latency of the execution units pipeline, the map completion time in terms of warps depends, after more than 6 warps are executed, on the specific computation. In practice, we can see that, empirically, until 8 warps scalability is almost perfect. This holds as long as the function f taken into consideration does not uses `double` data types; in this case, in fact, the amount of units available to perform such operations is way smaller than the number of CUDA cores. For these aforementioned scenarios, we will have:

$$T_{map}(1, w) = \left\lceil \frac{N}{w \times warp\_size} \right\rceil \times T_f \qquad w = 1, 2 \ldots 8$$

For w greater than 8, the specific application should be studied: however the time almost always diminishes, even though not proportionally to the number of warps activated. We show the scalability for 3 different computations in Figure 4.63. Here scalability is defined in terms of the number of warps. In the case of matrix multiplication, the scalability does not perfectly follow the ideal case, since there's unbalancing.

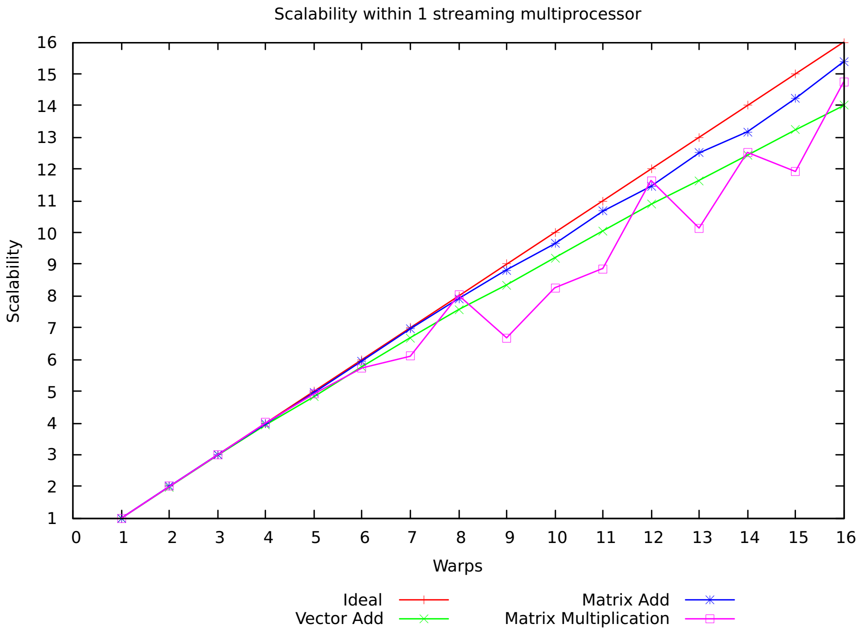


Figure 4.63: Scalability for three different computation, against the ideal.

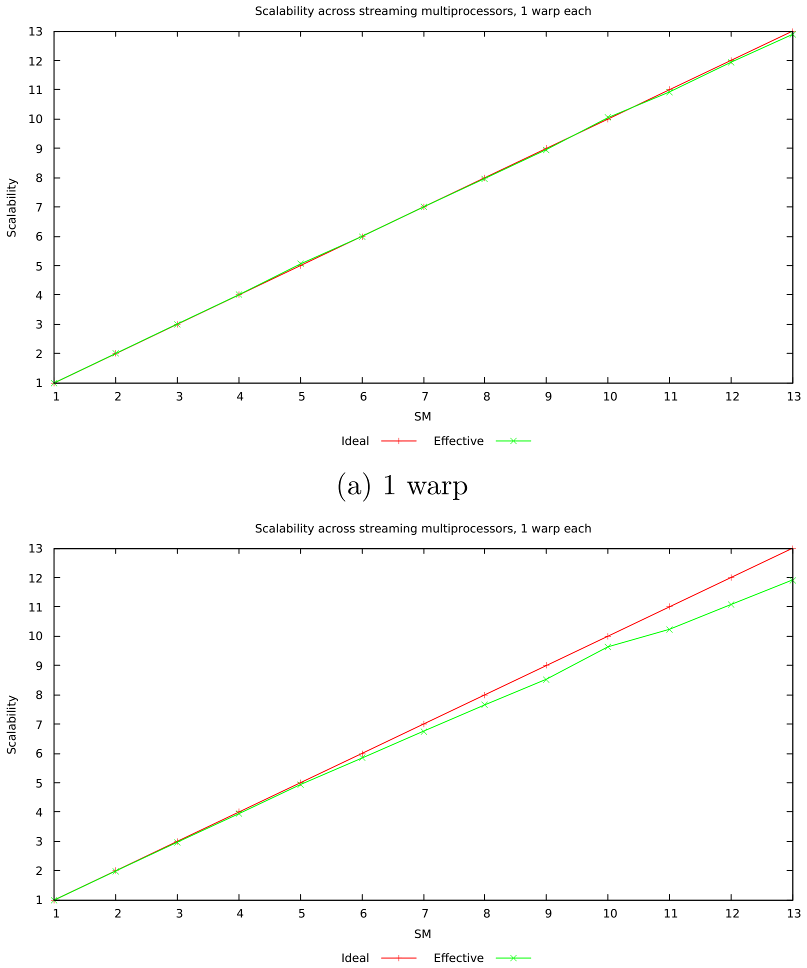
Let us now consider the case in which multiple blocks (and streaming multiprocessors) are exploited to carry on the computation. In this case, resources are duplicated, hence:

$$T_{map}(b,w) = \left\lceil \frac{N}{b \times w \times warp\_size} \right\rceil \times T_f \qquad\qquad w = 1, 2...8$$

Since the behaviour within a block is replicated, in case the introduction of more blocks does not causes further unbalance, we can also write

$$T_{map}(b,w) = \frac{T_{map}(1,w)}{b}$$

We can see experimental that this model holds as long as the blocks are mapped one to one on streaming multiprocessors. In Figure 4.64a we see on the y-axis $T_{map}(1,1)/(T_{map}(b,1)$, depending on b, while in Figure 4.64b we see the same figures but calculated in the case of 6 warps. The machine from which the numbers have been drawn is an Nvidia K20C, with 13 streaming multiprocessors.



(a) 1 warp

(b) 6 warps

Figure 4.64: Scalability varying the number of blocks and maintaining a fixed number of warps. The execution with 1 blocks and a fixed amount of warps is taken as baseline.

From this model of the execution time for a map computation, we can now estimate power for a given parallelism degree and a certain computation C applying f over N elements as:

$$E_C(b, w) = \hat{P}_C(b, w) \times T_{map}(b, w)$$

4.4.4 Communication model

Before offloading a computation to the GPU, another metric that should be considered is the cost of transferring the data from the host memory to the device one. This comprises a cost in terms of time and of energy.

From Section 4.3.2, we see that the average power used to perform a transfer is almost constant and is independent - in the device - of the type of memory used and of the direction of the communications. For this reason, we can estimate the power consumed during communication as a constant value, P_{comm}. This figure is architecture dependent and can be estimated by performing appropriate measurements on the targeted architecture. In case of Nvidia K20c can be estimated as $54.3W$.

For what regards the time spent in communication, we see that its value depends on the number of effective transfers (calls to `cudaMemcpy` as well as on the amount of data moved. For this reason, we propose a model in the form:

$$T_{send}(k) = T_{setup} + k \times T_{trasm}$$

where k is the amount of bytes sent to the device. From the outcomes of the experiments of Section 4.3.2, it is clear that we need to calculate T_{send}, T_{setup} and T_{trasm} at least for three scenarios: the case with non-pinned host memory, dividing between transmission from host to device and viceversa and the case with pinned memory, that can be safely modelled using the same parameters in both directions, as their difference is negligible.

We will denote the different cost in terms of sending the data as $T^{HD}_{send}(k)$, $T^{DH}_{send}(k)$, T^{P}_{send} for the transmission of data in case, respectively, of non-pinned memory from host to device, from device to host and of movement of pinned data.

If we fix the amount of data transferred (1GB in the carried on experiment) and take as explanatory variable for linear regression the number of calls to `cudaMemcpy` performed, we achieve a regression function in the form $\beta_0 + \beta_1 \times \#calls$. Modelling the function linear can be validated by intuition by observing 4.65. By dividing β_0 for the amount of moved data, we can find out the cost in time to transfer a single byte or a PCIExpress TLP (Transaction Layer Packet), in case its size is known in advance.

In case of *host to device* transfer with *non-pinned* memory, we achieve $\beta_0 = 0.87s$ (a small overestimation of the effective cost for transferring 1GB, $0.77s$) and $\beta_1 = 9.65 \times 10^{-6}s$. In the case the transfer is performed in the opposite direction (*device to host*), these figures increase: in the first case, β_0 will become $2.21s$ and $\beta_1 = 1.50 \times 10^{-5}$. Finally, for the case of pinned memory we will have $\beta_1 = 1.31 \times 10^{-5}$ and $\beta_0 = 0.49s$. The final values are summarized in Table 4.3.

type of memory	$T_{setup}(s)$	$T_{trasm}(s/bytes)$
host to device, non-pinned	9.65×10^{-6}	8.16×10^{-9}
device to host, non-pinned	1.50×10^{-5}	2.05×10^{-9}
pinned	1.31×10^{-5}	$4,52 \times 10^{-10}$

Table 4.3: Parameters for estimating communication costs in Nvidia K20c, depending on the type of memory used.

Overall, the maximum error in percentage for host, non-pinned memory is 7.46%, which corresponds to an error in estimation of $0.07s$, hence a negligible

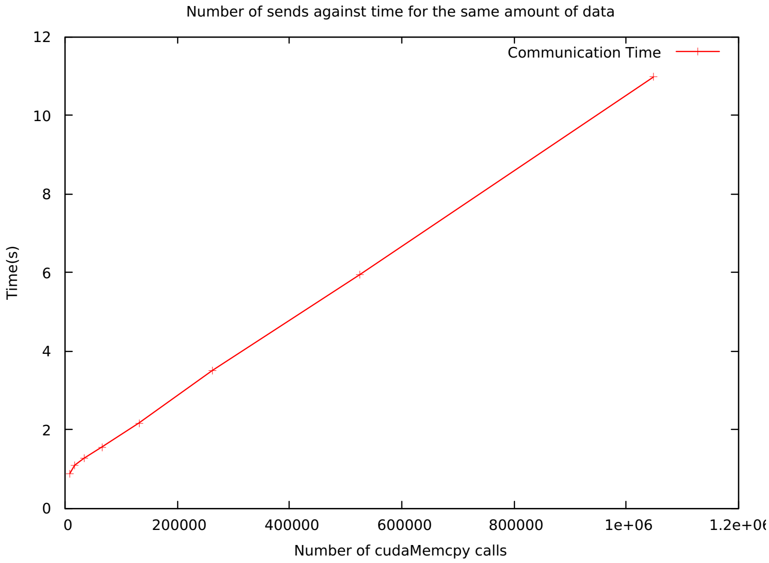


Figure 4.65: Communication time against number of sends. The amount of data transferred is constant (1GB)

amount. Consider that, if we estimate with this model the cost of transferring 1 GB in a single send, we have an error of $\sim 30\%$, which however represents an upper-estimation of barely $0.2s$.

4.4.5 Comprehensive map energetic cost model

From the host perspective, offloading a map computation is composed of two communication phases: the former is used to send data from the host memory to the device, while the latter is used to retrieve the result. Notice that the amount of data moved in the first case and in the second case might be different: as an example, if we do not change the input parameters copying them back to host memory is useless.

We will denote the amount of data to be sent from the host to the device with g, while the data retrieved after the calculation will be denoted with h. As usual, the parallelism degree is expressed in terms of blocks b and warps w.

The complete energetic model will be:

$$E_{map}(b, w, g, h) = T_{send}(g) \times P_{comm} + T_{map}(b, w) \times \hat{P}_C(b, w) + T_{send}(h) \times P_{comm}$$

The impact in terms of energy and time of the case in which the communication is overlapped with the calculation (achievable using *CUDA Streams*) has not been investigated.

4.5 Validation

4.5.1 Regression model validation

To prove the reliability of the regression approach, we will only consider the overall energy estimation for three target computations. All of the computations have been executed 5 times for every combination of parameters (b, w). The estimation has been performed using only six values, as described in Section 4.4.1. All experiments have been executed on a Nvidia K20c GPU, except the matrix multiplication one which was performed on a Nvidia K40m.

Vector Add

For this computation, we already described the error in estimating power through regression. We consider the case in which the "second-order" regression is performed only over $b = 1, 13$ and $w = 1, 18, 30$. We saw that the error is minimal when we consider only power (the error distribution for this case is visible in 4.56). However, when we multiply it by th estimated time to achieve an energy prediction, the error is much bigger. In Figure 4.66, we see for every warp and every block (represented as different lines) the detected error.

We estimate energy with $\hat{E}_{vector\_add}(b, w)$, defined as follows:

$$\hat{E}_{vector\_add}(b, w) = T_{map}(1, 1)/(b \times w) \times \hat{P}_C(b, w)$$

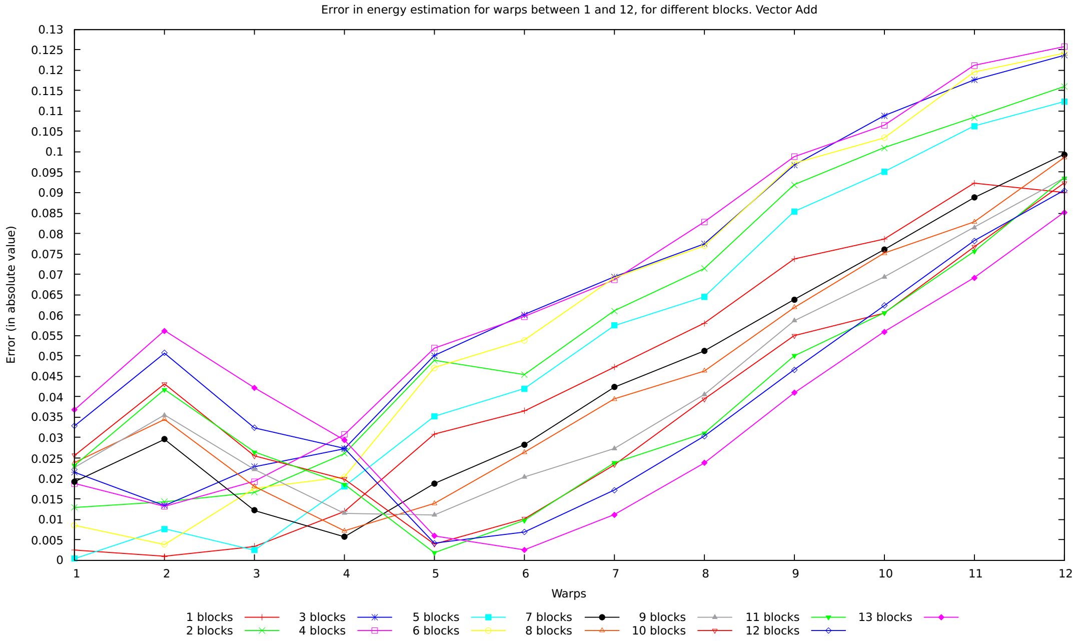


Figure 4.66: Error (in absolute value) for energy estimation. Warps between 1 and 12, different blocks. Vector add.

Notice that the lines showing the highest error (3, 4, 5, 6 blocks) have lower energy footprints, hence the overestimation in terms of Joules is not dramatic. For example, for $b = 4, w = 12$ (the highest error), we underestimated energy consumption of barely $1.68J$.

Since we consider the model in terms of time to be reliable only until 8 warps, this result is better than expected.

After 12 warps, even though the power estimation remains consistent, we see a drop in the precision of energy estimation. This is due to the fact that the computation does not scale perfectly once the number of physical resources is surpassed.

If instead we consider the case in which $T_{map}(1, w)$ is known, the scalability varying the number of blocks is almost perfect. In this case energy can be estimated as:

$$\hat{E}_{map}(b, w) = \frac{T_{map}(1, w)}{b} \times \hat{P}_C(b, w)$$

and we achieve the error profile visible in Figure 4.67.

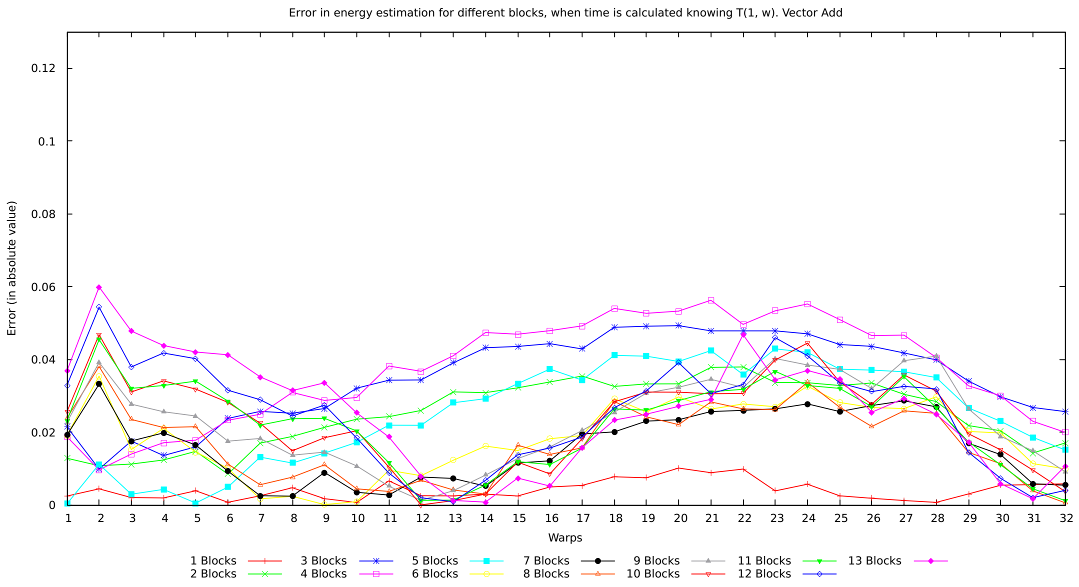


Figure 4.67: Error (in absolute value) for vector add computation, with different blocks.

The most important information that can be drawn from the above Figures is the fact that, in case time is estimated correctly (that happens, with our model, when the partitions assigned to each worker are balanced or the time for a given number of warps is already known for a certain parameter b), the error barely exceeds 6% in absolute value.

Matrix Add

We used the usual model described above for vector addition to estimate energy consumption in the case of a matrix addition computation. In Figure 4.68 we can see the error for executions using different number of warps and different number of blocks. Power is calculated using the regression predictor, while time is achieved by dividing $T(1,1)$ for $b \times w$.

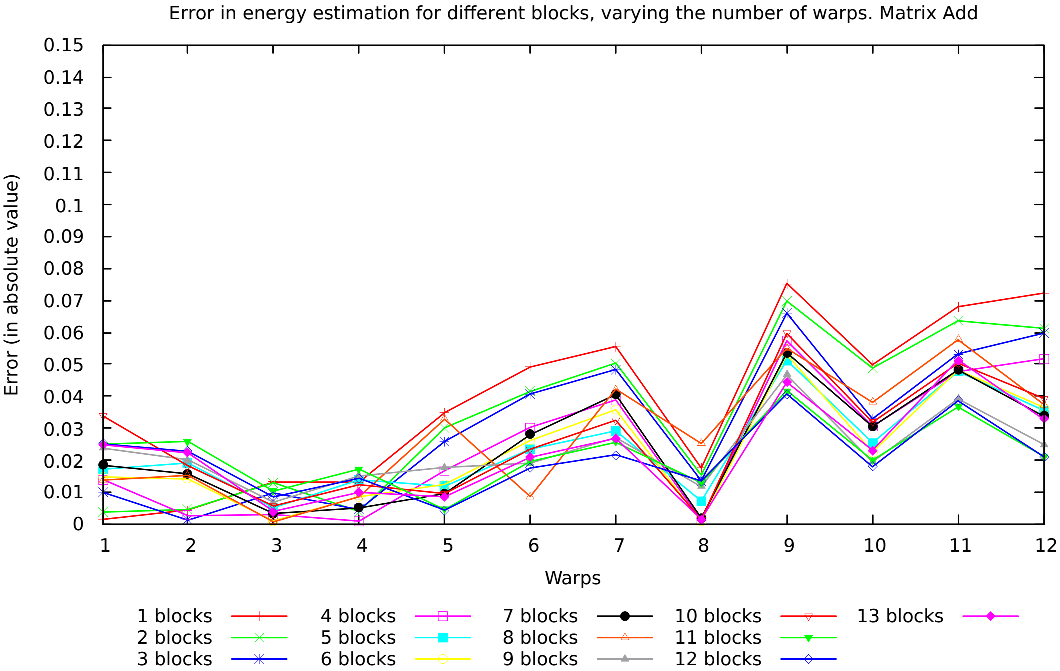


Figure 4.68: Error (in absolute value) for energy estimation, using different blocks. Warps between 1 and 12. Matrix add.

The error is not high because the scalability of the computation is pretty good even if the mapping between threads and CUDA cores is not one to one. We can see that in case the number of worker divides the size of the matrix, we have a lower error, meaning an increased precision in time estimation due to better partitioning.

If we estimate $T(b, w)$ assuming that we know the value for $T(1, w)$ and predict energy consumption accordingly, we achieve the error profile visible in Figure 4.69.

We can see that the maximum error never exceeds the 15%. If we limit the number of warps to be less than 12, we have a maximum error which barely reaches 5%, against the 8% of the case in which we were estimating time starting only from $T(1, 1)$. The error is very low also for higher number of threads.

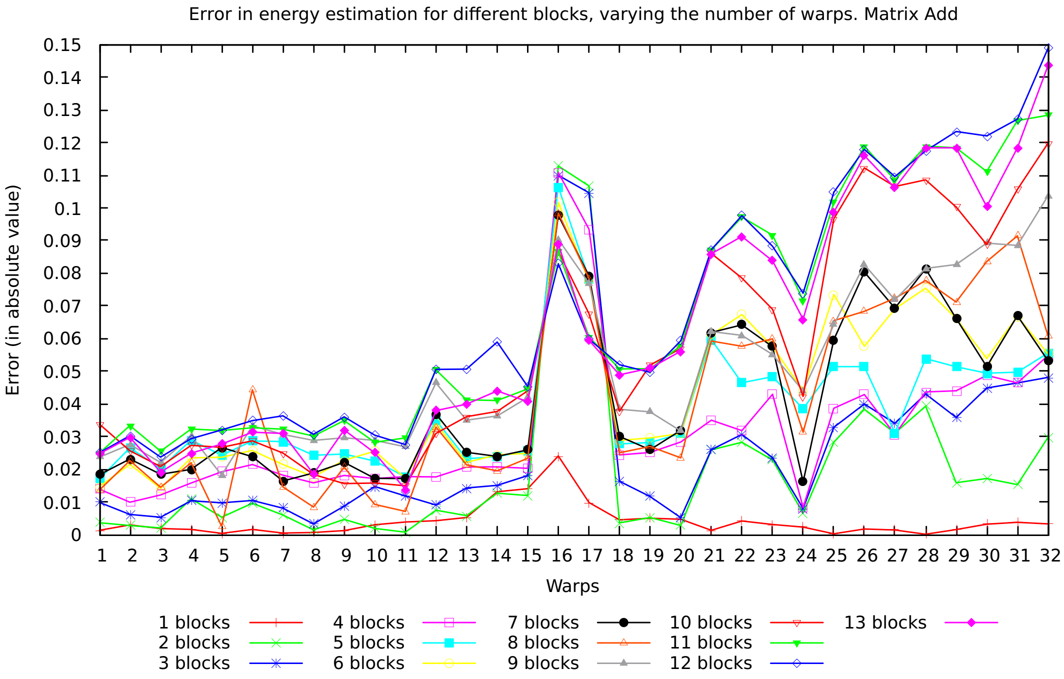


Figure 4.69: Error (in absolute value) for energy estimation, using different blocks. Matrix add.

Matrix Multiplication

The matrix multiplication operation (whose code is presented in Listing 4.8) is implemented in terms of a map. The matrix given in input is partitioned in terms of rows. The threads within a warp operate in parallel on a tile with fixed size, given as a template parameter.

In this case, it was not possible to give an amount of data proportional to the number of workers: for this reason, we have unbalance in the computation and this case is more difficult to treat.

```
template<typename T, unsigned int TILE>
__global__ void matrix_mul(T *a, T *b, T *c, size_t N) {
 /** Initialization of destination matrix */
 int rowJump = (blockDim.x * gridDim.x)/32; //rowJump also represents the total number of warps spawned in the grid.
 int tid = threadIdx.x + blockIdx.x * blockDim.x;
 int rowId = tid/32; //identifier of the warp --> of the initial row.
 for(int jext = tid%32; jext < N; jext+=TILE) {
  for (int kext = 0; kext < N; kext+=TILE) {
   rowId = tid/32;
   while(rowId < N) {
    register T cij=0;register T cij1=0;register T cij2=0;
    register T cij3=0;register T cij4=0;register T cij5=0;
    register T cij6=0;register T cij7=0;
    __shared__ T aValues[32][TILE];
    for(int f=0; f < TILE; f+=32) { //preload data in l1
     aValues[threadIdx.x/32][threadIdx.x%32+f]=a[rowId*N+f];}
    //no synch needed, since we access data only from our warp.
    for(int kk = 0; kk < TILE; kk++) {
     int k = kk+kext;
     int aik = aValues[threadIdx.x/32][k%TILE];
     /** Corresponds to a nested subloop on j */
     cij += aik*b[k*N+jj];
     cij1 += aik*b[k*N+jj+32];
     cij2 += aik*b[k*N+jj+64];
     cij3 += aik*b[k*N+jj+96];
     cij4 += aik*b[k*N+jj+128];
     cij5 += aik*b[k*N+jj+160];
     cij6 += aik*b[k*N+jj+192];
     cij7 += aik*b[k*N+jj+224];
    }
    c[rowId*N+jj]+=cij; c[rowId*N+jj+32]+=cij1;
    c[rowId*N+jj+64]+=cij2; c[rowId*N+jj+96]+=cij3;
    c[rowId*N+jj+128]+=cij4; c[rowId*N+jj+160]+=cij5;
    c[rowId*N+jj+192]+=cij6; c[rowId*N+jj+224]+=cij7;
    rowId += rowJump;
   }
  }
 }
}
```

Listing 4.8: A simple map matrix multiplication, with additional management of L1 memory (`__shared__`)

The tests were performed by calling the kernel using a huge matrix 16384 × 16384. However, the results are not satisfactory, since the fixed partitioning in terms of rows is too subject to unbalance. The highest the number of blocks, the worst the result, as it becomes more and more difficult to schedule an amount of warps proportional to the size of the matrix. In Figure 4.70, we see the error in energy estimation for 1 block, varying the number of warps between 1 and 12.

Despite the unsatisfying results in energy prediction, the estimation in terms of power is always reliable, as it can be seen in Figure 4.71, where we plot the error in estimating the average power against $N(0, \sigma^2)$. We see that the error profile closely remembers the gaussian, hence validating the regression-based approach for estimating power. The main problem with this computation is the fact that the partitioning in terms of rows is too subject to unbalance. We need a high-dimensional matrix to be able to measure reliably times on the GPU. On the other hand, even one row more in a partition will cause a huge difference in terms of time, making the timing difficult to be estimated empirically.

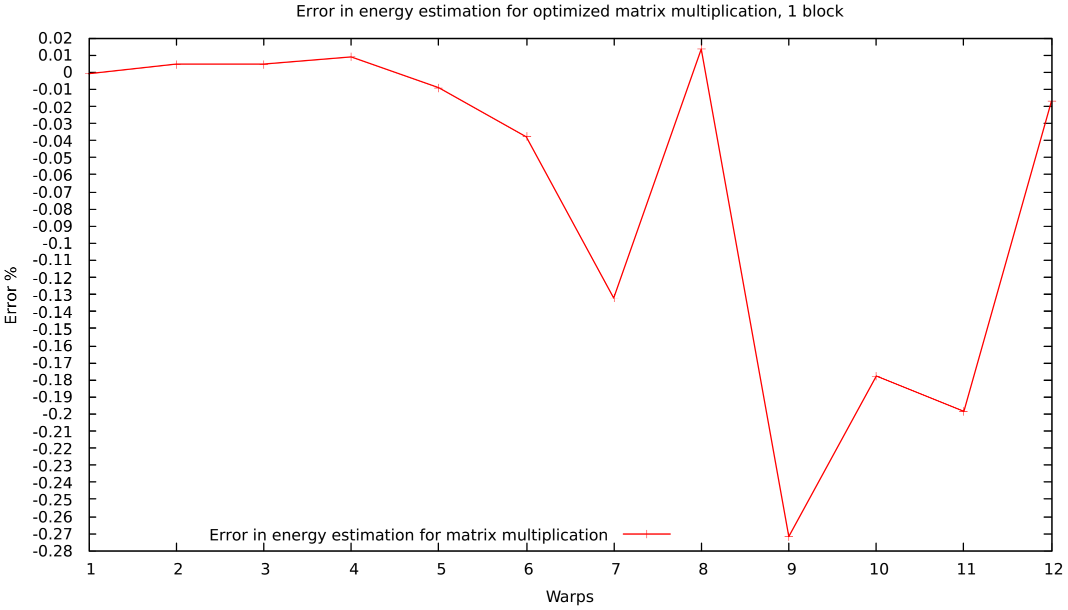


Figure 4.70: Error in energy estimation for optimized matrix multiplication using 1 block and warps between 1 and 12.

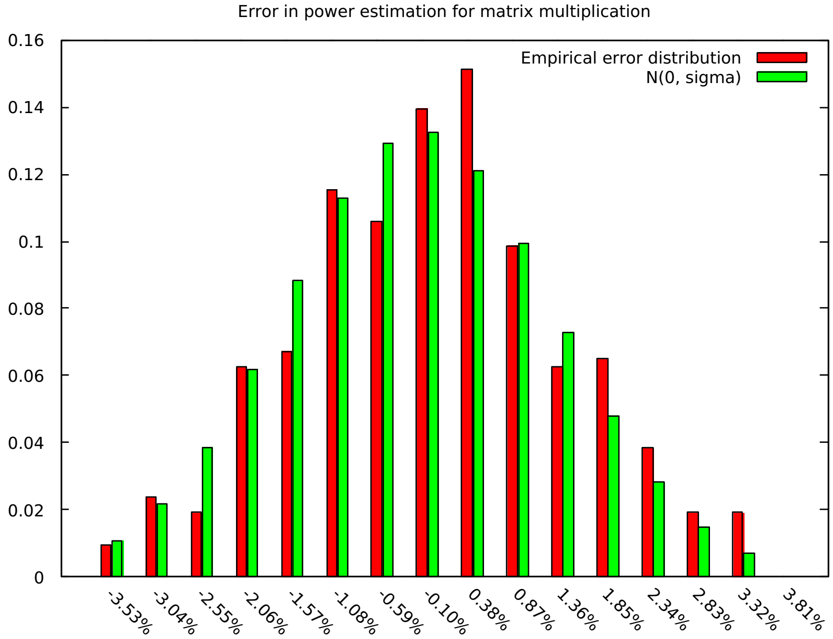


Figure 4.71: Error distribution in power estimation for matrix multiplication against $N(0, \sigma^2)$. On the x-axis we have the value of the error, while on the y-axis its probability.

If we consider a different version (less optimized) of matrix multiplication, in which data is partitioned in terms of threads (using an approach similar to virtual processor) as the one of Listing 4.9, we see a better behaviour in terms of scalability, which reflects on a overall more precise energy prediction.

```
template<typename T>
__global__ void matrix_mul_simple(T*a, T*b, T*c, size_t N) {
  unsigned long threadId = threadIdx.x + blockIdx.x * blockDim.x;
  while(threadId < N*N) {
    unsigned int i = threadId / N;
    unsigned int j = threadId % N;
    register T cij = 0;
    for(int k = 0; k < N; k++) {
      cij += a[i*N+k]*b[k*N+j];
    }
    c[i*N+j] = cij;
    threadId += blockDim.x * gridDim.x;
  }
}
```

Listing 4.9: Simple, less optimized kernel for matrix multiplication

In this case the partitioning is performed in terms of single elements, rather than on rows. Threads within the same warp access the same location of matrix A, while their access is coalesced with respect to B. Threads with the same id within different warps reuse the elements of B for their calculation. Every thread calculates a (set of) distinct elements of C. We will have less unbalance in the computation and we will have more precise timings. We tested the computation by executing the matrix multiplication kernel 10 times with different parallelism degree on an Nvidia K40m.

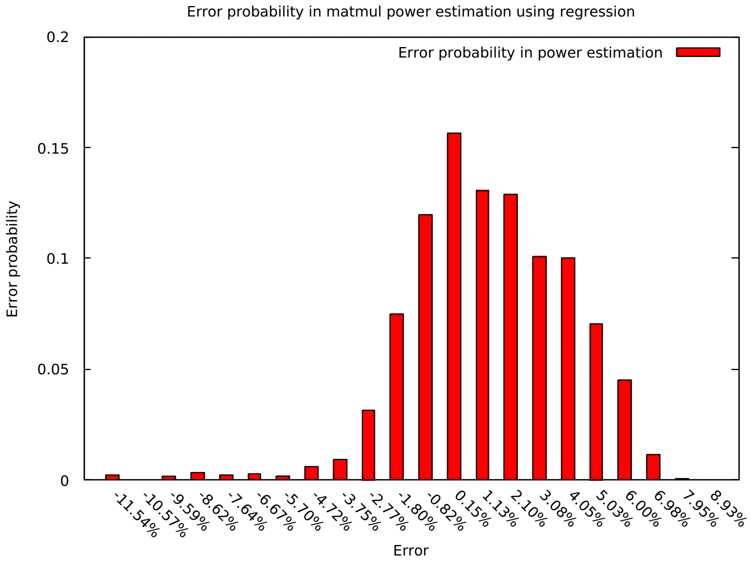


Figure 4.72: Error in estimating power for matrix multiplication using regression

The error in power estimation for this computation, on more than 2000 samples, is distributed as visible in Figure 4.72. The estimation is achieved by using only the usual 6 samples.

The error is only 1.85% in average, while the minimum is -11.85% and the maximum the 7.95%. More than the 85% of the estimations have below 5% error. If we consider the energy (whose error profile is visible in Figure 4.73), we maintain a very high precision. We can see that the model starts increasing the difference between the estimation and the effective values for elevated number of warps.

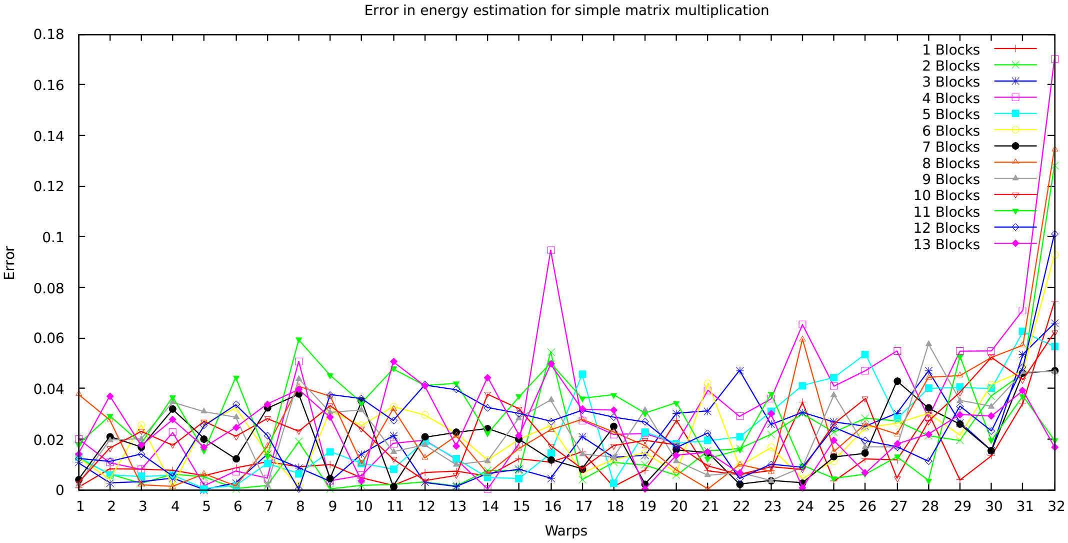


Figure 4.73: Error in estimating energy for matrix multiplication using regression

This validates the approach for the Kepler architectures, as we used a different board with overall similar features.

4.5.2 Heuristic validation

For evaluating the heuristic, we consider the probability of "guessing" the right power starting from a single sample of a computation. To evaluate this situation, we select uniformly at random a pair (b, w), we sample $P_C(b, w)$ and calculate $\alpha_C(b, w)$ for this pair. Then we try to predict the consumed power for another random pair (b', w') as $\hat{P}_C(b, w)$. This process has been performed selecting uniformly at random 500 quadruples of parameters (b, w, b', w').

Vector Addition

For the case of vector addition, the heuristic is quite efficient. The profile in terms of error in the power estimation can be seen in Figure 4.74. We see that the error is tightly concentrated between -10 and $+10\%$. In fact, more than the 85% of the

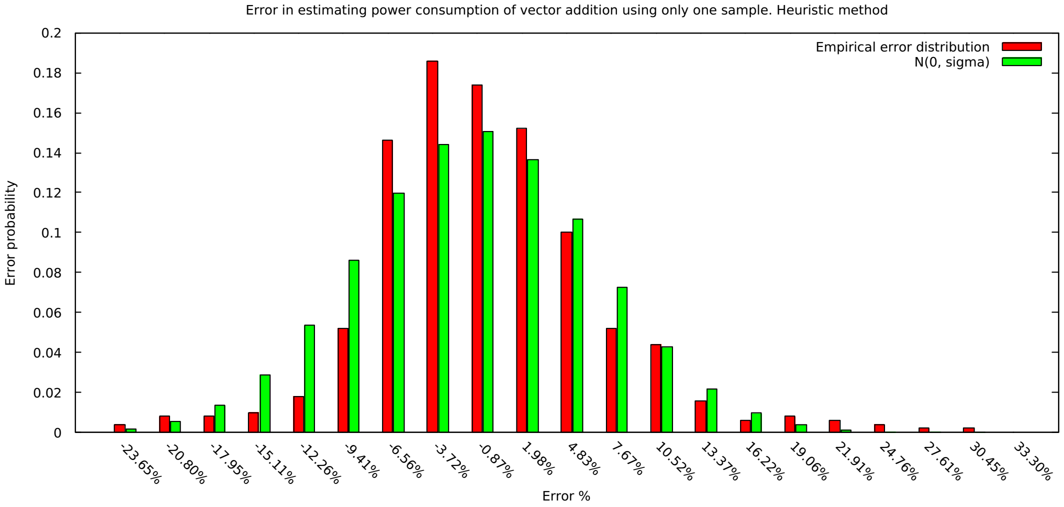


Figure 4.74: Empiric error distribution in estimating power for vector addition, against $N(0, \sigma^2)$

cases return an error of less than 10%: this means that, with very high probability, we are able to estimate power consumption of a computation using only a single sample. The error variance σ^2 is ~ 0.0056. In the aforementioned Figure, we see the empiric error distribution against a normal in the form $N(0, \sigma^2)$; the values for our estimation are more concentrated near 0 than in the case of the Gaussian.

Let us now move to consider how this precision in terms of power prediction maps into estimation of energy consumption. In this case, we estimate the time for an execution with parameters (b', w') starting from the time sampled with (b, w) as follows:

$$\hat{T}_{map}(b', w') = \frac{T_{map}(b, w) \times b \times w}{b' \times w'}$$

meaning that we are considering perfect scalability in both cases. Of course this condition will not hold for elevated number of warps.

Energy is estimated as:

$$\hat{E}_{map}(b', w') = \hat{T}_{map}(b', w') \times \hat{P}(b', w')$$

For this estimation, we have a very elevated error, that reaches (in very unfortunate cases) up to 85% overestimation, even though this happens with a very small probability. The error profile can be seen in 4.75. Even in this case we see that, however, we estimate energy consumption in above the 45% of the cases with less than 10% error.

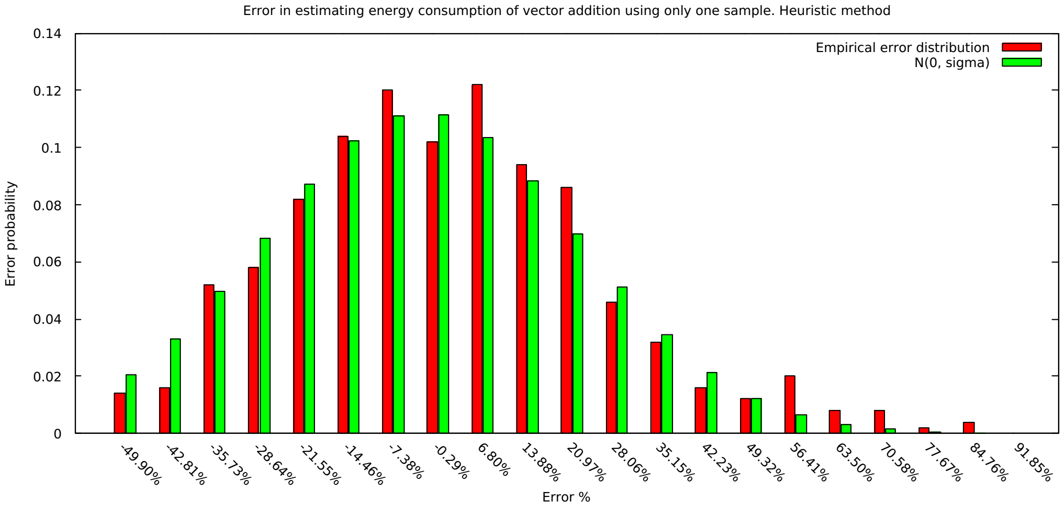


Figure 4.75: Empiric error distribution in estimating energy for vector addition, against $N(0, \sigma^2)$

By comparison of Figures 4.74 and 4.75, it is clear that the error lies in a wrong estimation of required time. If we limit the number of warps scheduled to be at most 8, however, the estimation precision widely improves. Over 500 random estimations, we achieve the error distribution in energy estimation visible in Figure 4.76. Again we have the 85% of the estimations giving less than 10% of error in absolute value, and an error in $[-5\%, 5\%]$ in above of the 52% of the cases.

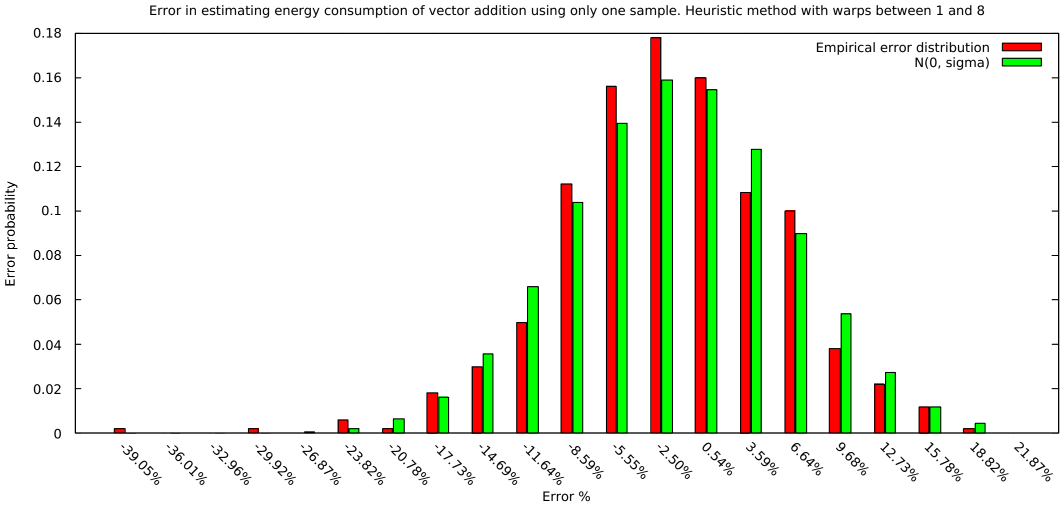


Figure 4.76: Empiric error distribution in estimating energy for vector addition, against $N(0, \sigma^2)$. The number of warps for this experiment has been limited between 1 and 8

Matrix Addition

We use the same modelling in terms of time and energy used for vector addition. Of course $\hat{P}_{matrix\_add}(b, w)$ will be different, as it will be calculated with a different value of α.

Also in this case the heuristic estimation of power is reliable: we can see in Figure 4.77 the error profile in power estimation. With respect to the vector addition case, the maximum error is higher (45.65% against 33.3%). However, values are more concentrated near 0, as we have the 88.8% of the samples with an error in $[-0.1, 0.1]$. In this case the distribution of the error resembles more a gamma distribution than a normal one. However we mainly care about the fact that, with very high probability, the power will be estimated correctly.

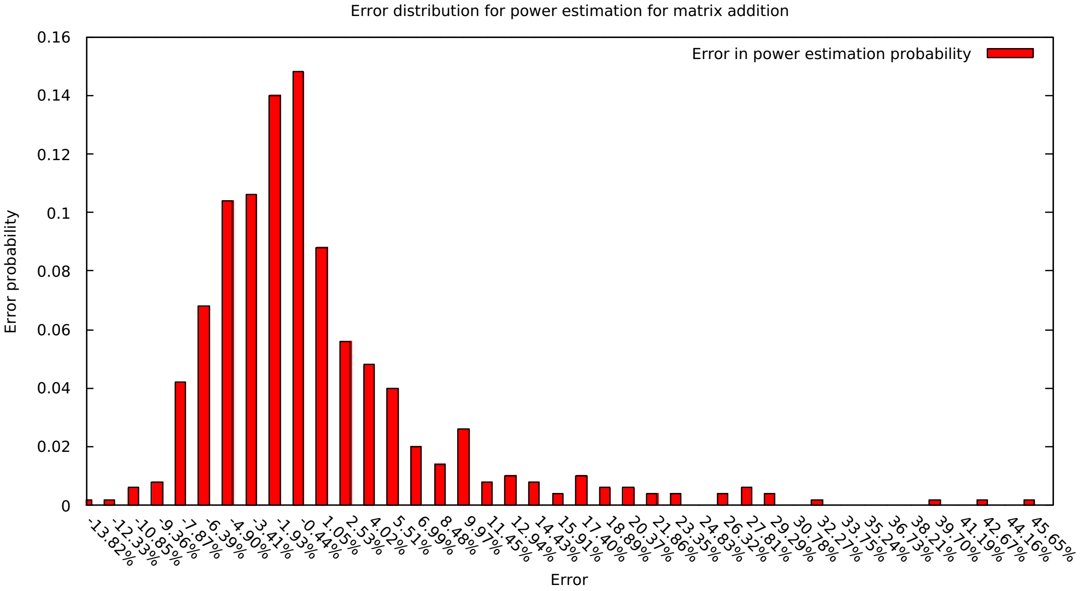


Figure 4.77: Empiric error distribution in estimating power for matrix addition

Let us now move to consider the energy estimation. As usual, we will consider at first the distribution of the error in energy estimation for all possible warps (from 1 to 32, in Figure 4.75) and then limiting the number of warps to at most 8 (Figure 4.79). Thanks to the increased scalability (in the number of warps), with this computation (as in the regression case) we achieve a better error profile with respect to the vector addition case. Considering all available warps within each block, the maximum error is 79%, against the error above 90% detected in the previous case. We have 50% of samples having less than 10% error with respect to the real value, while 71% of the values have less than 15% error.

In the case in which we reduce the number of warps, we enhance the energy estimation as time can be predicted more precisely.

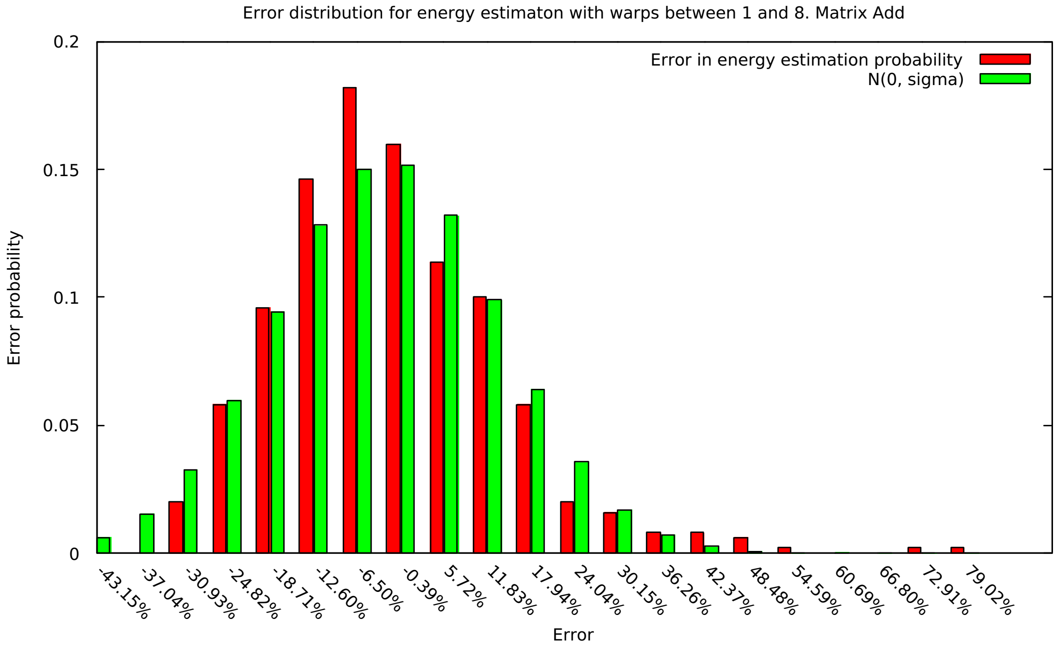


Figure 4.78: Empiric error distribution in estimating energy consumption for matrix addition, against $N(0, \sigma^2)$

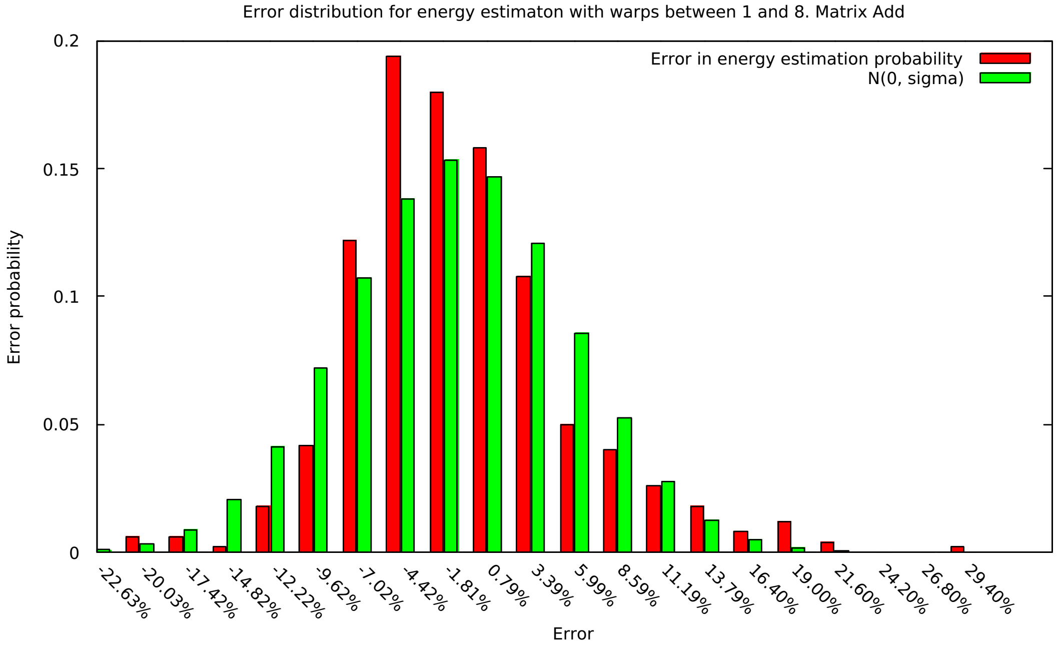


Figure 4.79: Empiric error distribution in estimating energy consumption for matrix addition, against $N(0, \sigma^2)$. Improved with warps between 1 and 8.

Matrix Multiplication

In this case the heuristic has been validated on a Kepler architecture, but on a different model. The experiments below refer to an Nvidia Kepler K40m, with 15 Streaming Multiprocessor with 192 CUDA cores each.

The parameters used for the heuristic have been estimated through regression on the sleep computation, as it was done for the aforementioned K20c board in Section . We did not considered the case of all the 15 SM active together, but we show that even though this case was neglected, the heuristic is reliable even if we consider this case. When the parameters are estimated, we can see similar behaviour for the K20C but, of course, with different constants:

- P_{base}, the base consumed power is in this case $66.5436W$;
- P_b is $0.1930W$;
- P'_w is $0.2679W$;
- P_w is $0.001W$ and we will consider it negligible.

By applying the usual heuristic process to the simple matrix multiplication kernel, whose code is visible in Listing 4.9, we achieve the error profile visible in Figure 4.80.

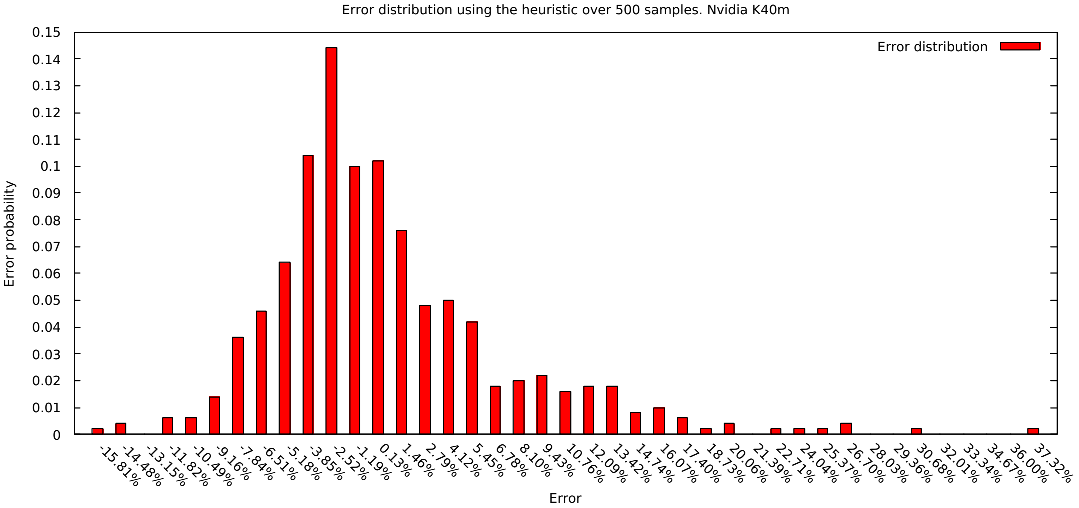


Figure 4.80: Error distribution for the heuristic and matrix multiplication computation. Nvidia K40M

The error is less than 10% in absolute value with a probability of over the 87%, hence validating the usage of this heuristic on other Kepler architectures. When we consider errors below 15%, we see that we have over 95% of the samples in this range, hence validating that the heuristic is correct, works on different models and can be reused across different devices and different map computations.

We also provide the error distribution of the energy estimation done accordingly to the heuristic and the map time model. It is visible in Figure 4.81. The energy estimation is more precise than the power estimation, as it happened also in other cases previously mentioned. The reason for this more accurate precision can be found on the nature of the heuristic: in fact, it tends to *overestimate* the power consumption; the time model - instead - *underestimates* the completion time of the map.

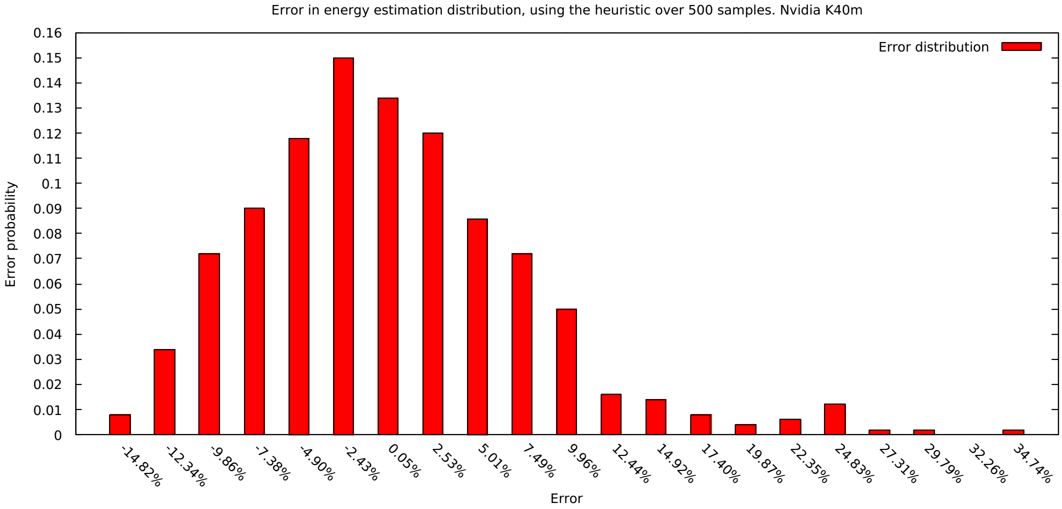


Figure 4.81: Error distribution for the estimation of energy consumption on matrix multiplication, using the heuristic.

Once again, the probability that the energy estimation is wrong of at most the 15% is above 95%. This demonstrates that not only the heuristic and the time model are accurate, but also that they work across different architectures. The error profile for energy was achieved in this case by selecting at random between all of the scheduling parameters available.

Conclusion

As a final consideration, we see experimentally that the error in energy estimation depends on the "distance" between the parameters (b, w) and (b', w'). For this reason, we suppose that by increasing the number of samples, we can increase the precision of the heuristic in estimating energy consumption. A weighted average implementation, taking into account the distance between computations for estimating both power and time could provide a better estimation and is left for future works.

Observe that we have seen computations operating on different data structures, with different partitioning methods and different computational requirements. Also, we used two different devices of the same family to assess the precision of the heuristic and of the model. We have seen that in all considered cases the power estimation is quite precise; a small loss of precision can be seen whenever the partitioning is not precise: this corresponds to the idea that, in case a different number of elements is given to a certain worker, it will deactivate early and hence the consideration underlying this model do not hold anymore. However, we can estimate power also in this unfortunate situations with acceptable loss of precision.

The estimation in terms of energy can be considered precise as long as the number of warps utilized maps into the number of available physical CUDA cores. After this point is reached, the accuracy of the prediction of time is lowered considerably, hence making the energy predictor unreliable. For cases (like simple matrix multiplication) in which the computation scales almost linearly in the number of warps, the model predicts accurately for all the available parameters.

Chapter 5

Energy model for map computations on CPU

In this Chapter we will briefly outile a very simple, approximated energy model for data-parallel map computations on CPU architectures. In the first part, we introduce some observations on the environment, pointing out the regularity of the behaviour of the computation across the execution, the similarity between executions of different computations exploiting the same parallel pattern and the validity of the average case. We then move to give a short justification of the choice of the explanatory variables for the model, achieved through regression and presented in Section 5.4. We finally validate this approach for estimating energy consumption.

5.1 Preliminary observations

As we did in Chapter 4, we wish to perform some assumptions about the measure that we want to predict. In this section, we will (briefly) show how the assumptions done for GPUs also hold in case of CPU multi-core architectures. The tests have been performed on a Intel Xeon E5-2650 processor, with 2 NUMA nodes with 8 processing units each. We did not selected frequency and voltage manually (see Section 2.1), leaving this task to the default Linux governor. For this reason, we developed a frequency monitor to understand the impact on power due to different frequencies. We observed experimentally that the frequency is automatically scaled up to the maximum available f_{max} when a demanding computation is executed. Even though studying how a parallel computation behaves in case of different frequency selection is an interesting problem and could potentially enrich our model, this further development is left for future work.
Let us now start by considering the first observation made in Chapter 3:

Observation 1 *The instantaneous power required during the execution of a certain computation C, with the same used resources and the same parameters, does not depend, from a probabilistic point of view, from its starting time.*

As evident in Figure 5.1, were we show the intantaneous power for the execution on a single worker of ten differnet execution of the computation as of Listing 5.1, we don't have an initialization phase as in the GPU case. This can be explained with the fact that the core is already in use (even though not at peak performance) when the computation is launched. We can see an irregular behaviour, as there are spikes in power, due, in our opinion, to the execution of operating system processes in the same NUMA node: measurements are provided by EML in a socket-aggregated manner. Further, EML sampling is $0.001s$, hence small variations have a huge impact.

```
template<typename T, unsigned int repetitions>
inline void cpu_power(T *a, T * b, T * c) {
  c[0] = 0;
  for(int i = 0; i < repetitions; i++) {
    c[0] += (a[0] + b[0] *i);
  }
}
int main() {...
 emlStart();
  for(int i = 0; i < vectorSize; i++) {
    cpu_power<float, R>(&a[i], &b[i], &c[i]);
  }
 emlStop(computation); ...
}
```

Listing 5.1: code for the computation used to test the instantaneous power of a NUMA node

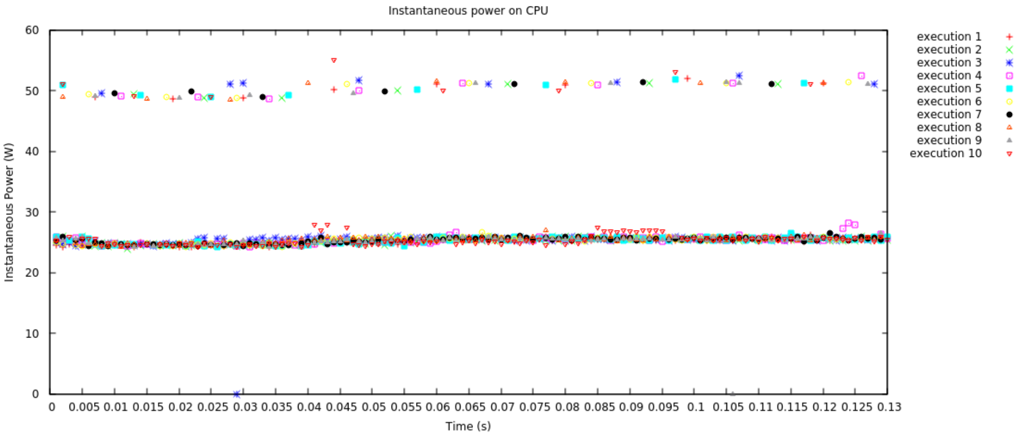


Figure 5.1: Instantaneous power detected on a single NUMA node while carrying on a sequential computation on it. 10 different executions are visible in the plot.

Since the spikes are distributed almost uniformly along the time line, and there are no apparent regular patterns explaining their occurrence, we can safely assume that they are caused by exogenous causes and hence can be neglected for the analysis of the power required by a computation. Notice that to verify this condition we need a *load-free* (or almost so) machine; otherwise, the power requirements of other applications will inevitably alter the readings. Despite this, we can assume safely ideal conditions, as it is often done in the HPC world.

If, instead, we sample a program comprehensive of the computation of Listing 5.1 followed by a more complex map using some trigonometric functions, we see a variation in power consumption, that can be seen in 5.2.

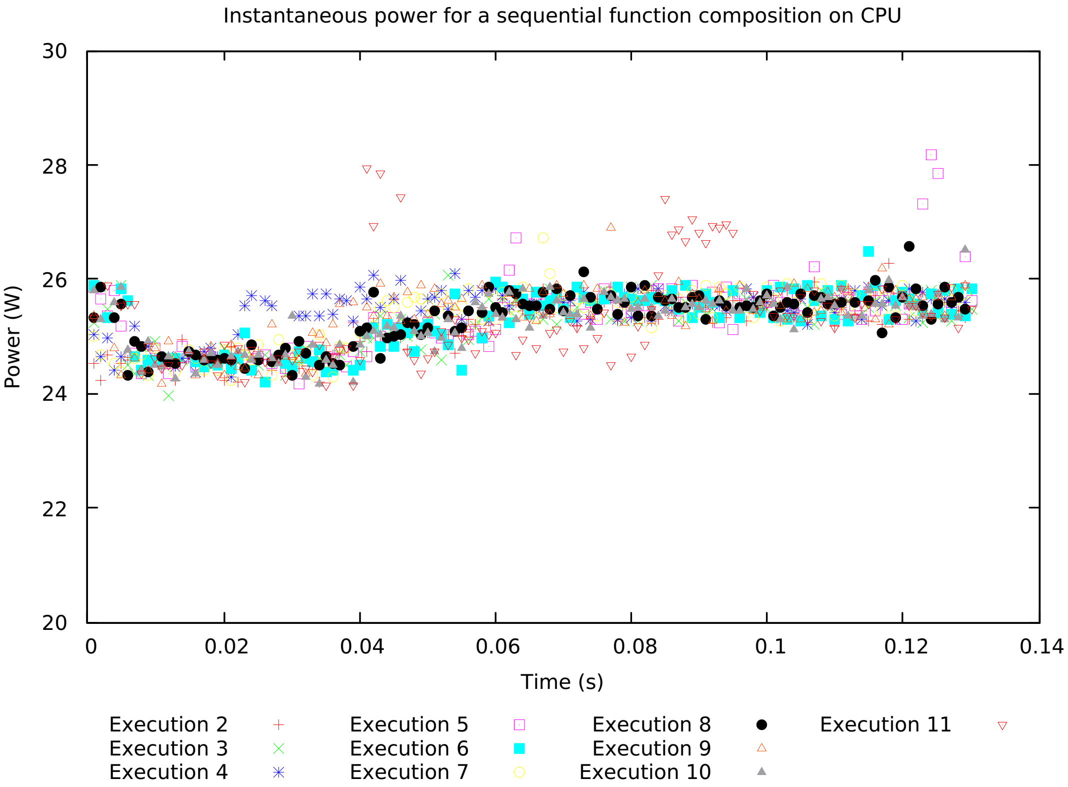


Figure 5.2: Instantaneous power of an irregular computation, executed sequentially on a Intel Xeon E5-2650 processor

We show the impact of parallelism on dynamic power in Figure 5.3, where the power profile of two different executions of a vector addition performed as a map computation on a CPU is visible. As in the GPU, we can see that the instantaneous power profile overlaps. The noise caused by activation of other processes is visible also here, but we did not show it. The experiment has been performed by pinning the workers on the CPUs with identifiers from 1 to 4; hence only the first NUMA node (0) has been used.

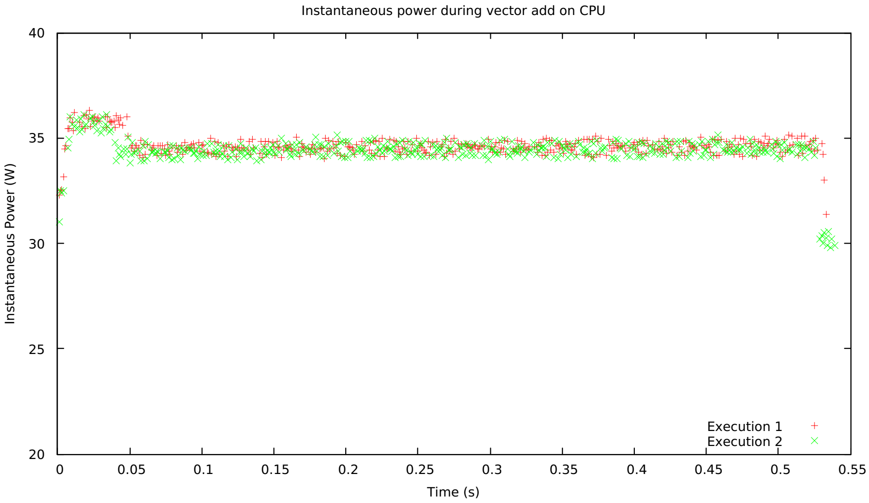


Figure 5.3: Instantaneous power of different executions of vector addition computation executed in parallel on 4 processing units , as detected on the NUMA node on which the computation is executing.

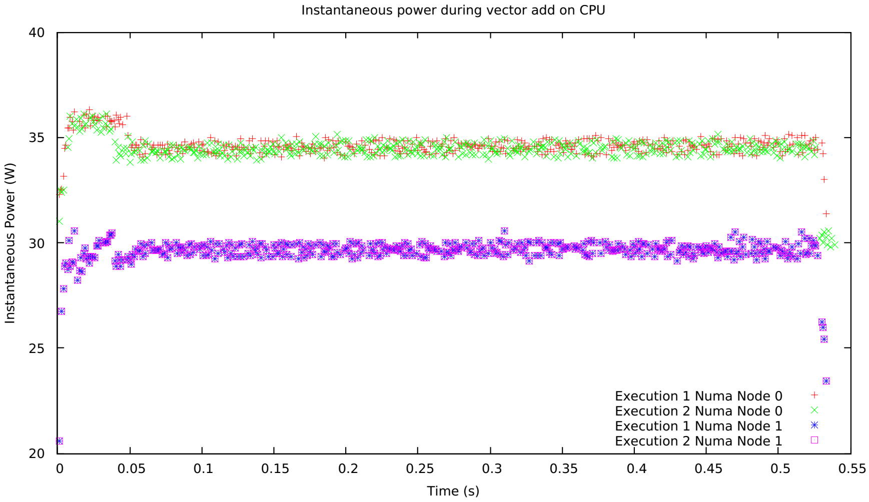


Figure 5.4: Instantaneous power of different executions of vector addition computation executed in parallel on 4 processing units , as detected both on the node on which the computation is executing and on the unused one.

For contrast, we show the profile of instantaneous power of different executions of vector add of both NUMA node 0 NUMA node 1 in Figure 5.4.

We also show the difference in power caused by different parallelism degrees in Figure 5.5, where a (part of) the instantaneous profile of the aforementioned computation is plotted when the execution is performed with 1, 2 and 4 cores.

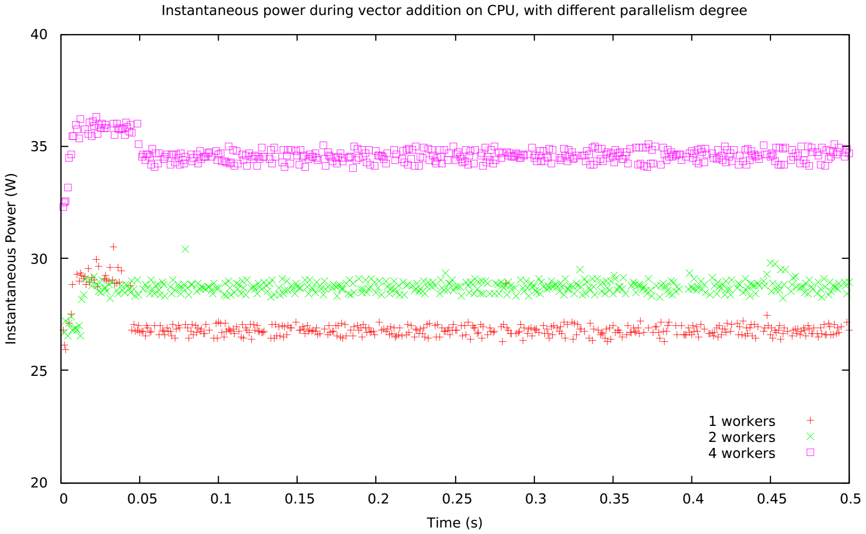


Figure 5.5: Instantaneous power for vector add execution with different parallelism degrees.

Observation 2 *Given a parallel exploitation pattern, different computations implemented with the pattern exhibit similar behaviours in terms of average power requirements when the amount and type of resources assigned for the calculation change.*

In the case of a map computation, we can see that the average power grows linearly in the number of processing units allocated to it. In Figure 5.6, we show the average power varying the number of workes; the first computation (in red) is a simple vector addition, the second one (green) calculates some trigonometric functions over the elements of two different arrays and stores the results on the result array, while the last one is a matrix addition in which the partitioning is performed in terms of rows.

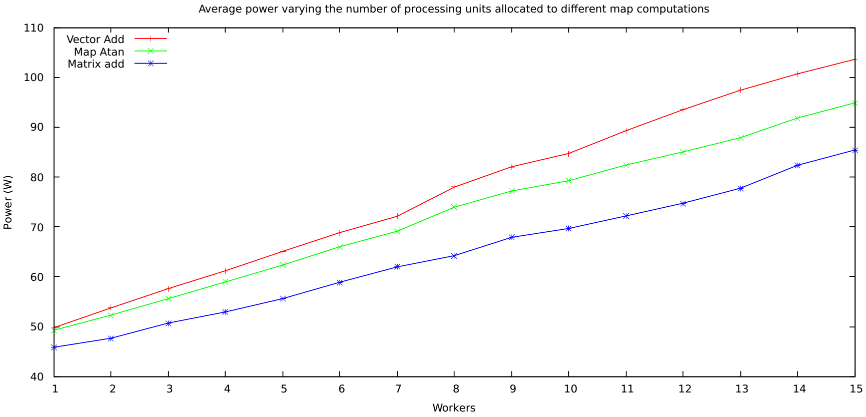


Figure 5.6: Average power on Intel Xeon E5-2650 for different map computations depending on the number of processing elements in use.

In case of multiprocessor architectures, the fact that the power would grow linearly in the number of units used is rather obvious: all units have exactly the same structure and perform (if the computation is balanced) exactly the same work. Hence we would expect the same resources to be exploited and the same power to be consumed. This differs from the case of the GPU, where we have, instead, different kinds of resources. As a final remark, a little step is visible between 7 and 8 workers in all the computations taken into consideration. This corresponds to the point in which the second NUMA node is activated.

Up to now we did not considered multi-threading. In Figure 5.7, we can see the detected average power using the same machine also in multithreaded mode. As it is possible to see, after the number of physical execution units is completely used, the power reaches a plateau and remains (almost) constant. However, the completion time will in general remain the same or decrease, hence making this approach not convenient as energy consumption will increase.

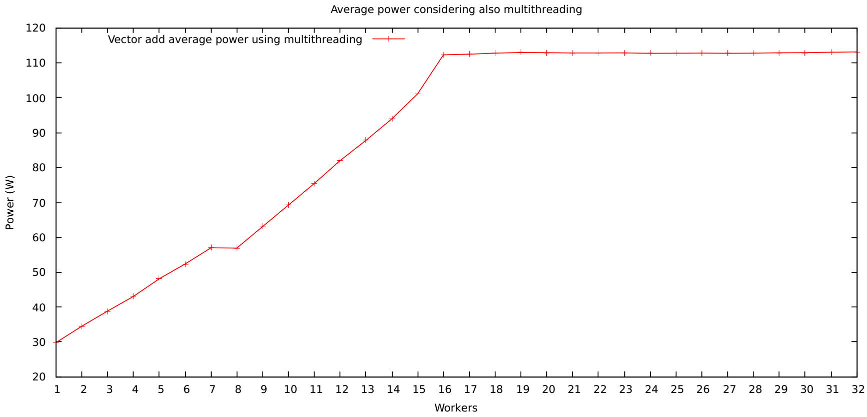


Figure 5.7: Average power for vector add using multithreading

As a final consideration, we wish to make an argument for the convergence of the average in time of power to the steady state instantaneous power also for this architecture. In fact:

- the instantaneous power behaviour is rather regular: the samples have almost a constant value;
- the spikes in power consumption caused by other processes can be considered an almost constantly distributed noise;
- the presence of (very small) initialization and de-initialization phases in which power consumption is different from the steady state gets amortized for long enough computations.

We also expect, for this architecture, that the average reflects better the steady state power, as initialization and de-initialization phase (if present at all) have a very little duration.

5.2 Component Separation

As we have seen previously, the power consumed in a computation can be expressed as a function of:

- the parallelism degree;
- the computation.

As we said before, the parallelism degree impacts on power in a linear manner. This is caused by the fact that a certain function f will have a certain average time T_f and a certain average footprint in terms of energy E_f. We consider for both cases the variance to be very small.

Consider the case in which this function is used sequentially on a certain number N of elements. We will have $E_{f\_seq} \simeq N \times E_f$, as the same set of instructions will be executed on N different elements [CA12]. Similarly, the time will be $T_{f\_seq} \simeq N \times T_f$.

When we move to consider a parallel implementation in terms of a *map*, we will have n identical workers operating over $\sim \frac{N}{n}$ elements each. This will lead to have:

$$E_{par\_f}(n) \simeq n \times \frac{N}{n} \times E_f$$

and

$$T_{par\_f}(n) \simeq \frac{N}{n} \times T_f$$

the average power for the parallel case will consequently be:

$$P_{par\_f}(n) = \frac{E_{par\_f}(n)}{T_{par\_f}(n)} \simeq n \times \frac{E_f}{T_f} = n \times P_f$$

As a consequence it is quite clear that the components to be taken into account are both the computation f, which determines the power, and the number of units n used to carry it on. In the above description we neglected the impact of parallelism on energy and time: in fact, in general moving to a parallel implementation will introduce additional overhead, with its own cost in terms of energy and time. However, for computations scaling well enough we can assume that what said above holds true. We also avoid to consider the case of different NUMA nodes: while the impact in power consumption of using one more can be noticed, it adds complexity to the model without giving much advantage.

5.3 Experiment design and analysis

5.3.1 Monitoring power requirements

The used measurement library (EML, see Section 1.4.4) works by spawning an independent thread, that continuously samples some hardware counters in order to detect current power/energy consumption on the monitored device. Obviously, this process has a footprint in terms of consumed energy.

We considered interesting for this work trying to understand the impact in terms of power consumption of the measurement library: this is an overhead that could theoretically be considered and, in case of practical implementation of an energy aware behavioural skeleton, should be taken into account when measuring the difference with respect to baseline.

In order to measure the footprint, we slightly modified the library source code, so that it was possible to "pin" the monitoring thread to a certain core. This allows to:

- monitor footprint of the library, by pinning the monitor to the same NUMA node on which it performs measures;
- monitor leakage power without accounting for this further overhead, by pinning the monitor on another NUMA node if available.

By the code of Listing 5.2, we estimated the measurement library energy consumption library to be of about $\sim 0.77W$.

5.3.2 Leakage power estimation

The logic for estimating leakage power consumption is the same as the one used for GPU architectures. However, while in the GPU case the thread spawned by the measurement library (EML) is resident on the CPU (hence the measured values will only be affected by the sampling process), in the case of CPU we need to avoid the overhead of the library. To do so, we measured consumption of load-free NUMA node 0 pinning the monitor to NUMA node 1, and viceversa, using the code of Listing 5.2.

```
//parameters: samplingInterval, cpuId, monitorCpuId
int main(...) {
 ...
 CPU_SET(cpuId, &cpuset);
 EMLSetMonitorAffinity(monitorCpuId);
 ...
 unsigned float currentPower = 0;
 unsigned float prevPower = 0;
 do {
  emlStart();
  sleep(samplingInterval);
  emlStop(leakage);
  emlDataGetElapsed(leakage[0], &elapsed);
  emlDataGetConsumed(leakage[0], &consumed);
    prevPower = currentPower;
    currentPower = consumed/elapsed;
  } while(fabs((prevPower - currentPower)) > 0.001);
  ...
}
```

Listing 5.2: Code used to estimate leakage power on CPU. The sampling is executed on `cpuId` with an interval of `samplingInterval`. The monitor is pinned to `monitorCpuId`

Notice that in this case the `samplingInterval` must be appropriately long, since the environment is more noisy. For very short sampling intervals, it could be that convergence is not reached because of the operating system processes activation. We estimated leakage power consumption for the considered architecture to be of about $18.87W$.

5.3.3 High level experiments

In this case we defined some map computations. To do so, we used **FastFlow**, a C++ parallel programming framework previously described in Chapter 1. The computations test different map functions, different data structures and different data types.

The execution of a map data-parallel computation, implemented using the `ff_farm` skeleton as base, is surrounded by proper calls to the monitoring library. All tests have been performed calling single computations, executed with a distance of 60 seconds one from the other, to give the possibility to the frequency governor to scale down frequency again and start in similar conditions all executions.

5.4 Model Individuation

5.4.1 Estimating power consumption through regression

Also in the case of CPU architectures, we wish to model the behaviour of the power by using regression. Given a certain computation C, we try to find an estimator $\hat{P}_C(n)$, which depends on the parallelism degree and on the specific power footprint of a computation to predict power expenditure.

As usual, we use ordinary least squares and try to find a law in the form:

$$\hat{P}_C(n) = \beta_0 + \beta_1 \times n$$

In the considered architecture, β_0 is comprehensive of:

- the leakage power of both NUMA nodes considered;
- the power consumed by the memory.

By considering the three different computations used also in Section 5.1, we have the coefficents of determination visible in Table 5.1. Since the values are very close to one, this demonstrates the validity of estimating power through regression.

Computation	R^2
Vector Add	0.998770994
Vector with atan	0.998882549
Matrix Add	0.998443599

Table 5.1: Coefficient of determination (R^2) for the linear regression estimation of three different estimations.

Let us consider also for this architecture what happens in case we use less samples. Consider the case in which only the power sampled for 1 and 15 is used as parameter to achieve the predictor. In this case the maximum error in absolute value, for vector addition, moves from 1.16% (achieved when all samples were available) to 1.88%, an error that is negligible. For the case of the function previously denominated as `atan`, we have a maximum error in absolute value of the 2.55% (against 2.10%). For matrix addition we have an increase in error of the 0.61% against the baseline of 2.10%.

In Figure 5.8, we show the distribution of errors for estimating power consumption using only two samples on the CPU. The empirical distribution has been calculated over only 150 samples (10 for parallelism degree), explaining the higher irregularity with respect to the GPU case.

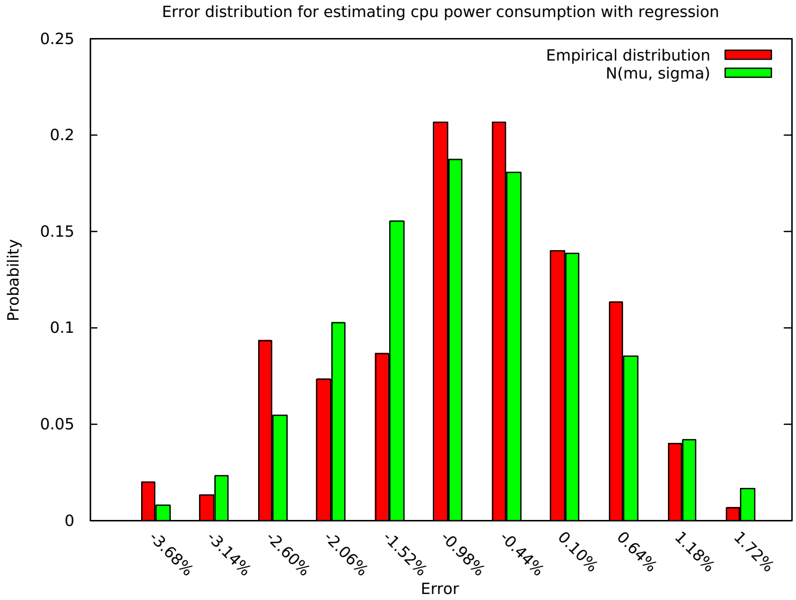


Figure 5.8: Error in estimating power using only two samples on the CPU. Plotted against a gaussian $N(\mu, \sigma)$.

Similar error profiles can be achieved with the other two considered computations.

In this case, we did not developed a suitable heuristic for estimating power consumption on the CPU. The reason is that, if the computation scalability is close to the ideal and DVFS is not used, on the computations taken into account the behaviour was easy to model for each computation and with a negligible footprint. In fact, the very high sampling rate used by the library to monitor this architecture, together with the little duration of the transient phases, suggests that it is enough to sample power for a small interval in time in order to develop an accurate power model.

The introduction of DVFS techniques, while in general giving another optimization space, would complicate the model and could bear to the necessity of introducing heuristics for estimating power also on this architecture.

5.4.2 Map energetic cost model

As we did in Section 4.4, we consider a simple, high-level model to estimate the duration of a map computation. Consider the case in which a function f requiring an average time of T_f is applied on a collection of N elements. We remember that in this work, we assume time and energy requirements to have a very low variance.

Since we assign (approximately) the same amount of work to each of the n workers, we will estimate power depending on the parallelism degree as:

$$T_{map}(n) = \left\lceil \frac{N}{n} \right\rceil \times T_f \simeq \frac{T_{map}(1)}{n}$$

Given the estimator $\hat{P}_f(n)$ calculated as explained before, we estimate energy for a map computation on CPU using n identical parallel workers as:

$$E_{map}(n) = \hat{P}_f(n) \times T_{map}(n)$$

5.5 Validation

Since we discussed before the errors and the coefficient of determination for the power consumption estimation, we wish to show now how the above models allow us to estimate energy.

We show the error profile in energy estimation for vector addition in Figure 5.9. Even though we do not see the regularity of the case of GPU and of the power estimation, it is possible to see that the error is very small. This is thanks to the fact that in this case not only the power estimation is precise, but also the model used to predict time is very reliable. We *always* have an underestimation of energy consumption. This is because we underestimate the completion time, being scalability not perfect and the estimation of power is very precise, hence not giving the margin that we had for the GPU.

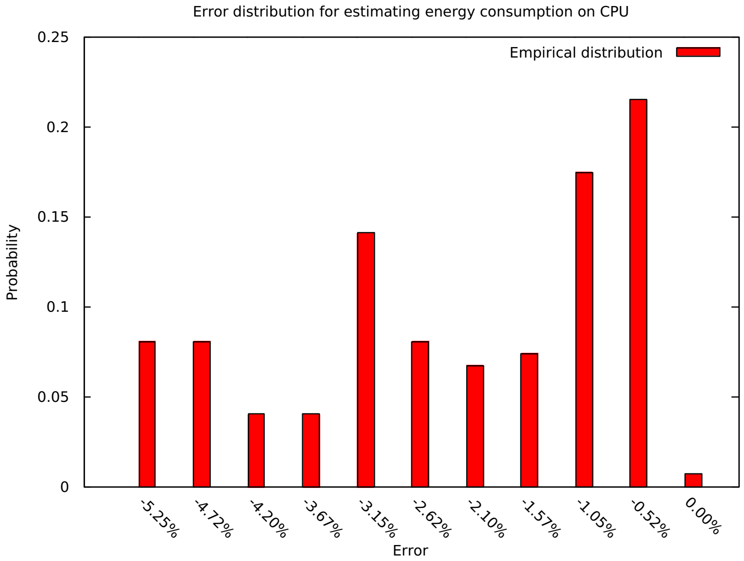


Figure 5.9: Error in energy estimation for vector addition.

In the case of the map computation computing the result by means of a number of trigonometric operations, we have a slightly worst result. This is probably due to the fact that the behaviour is more noisy. Similarly to the case of vector addition, we see a distribution skewed towards zero. The error profile is visible in Figure 5.10. We can observe that even though at worst case we will have an error higher than 10% in absolute value, we have an error between -5% and 0 with very high probability (above 88%).

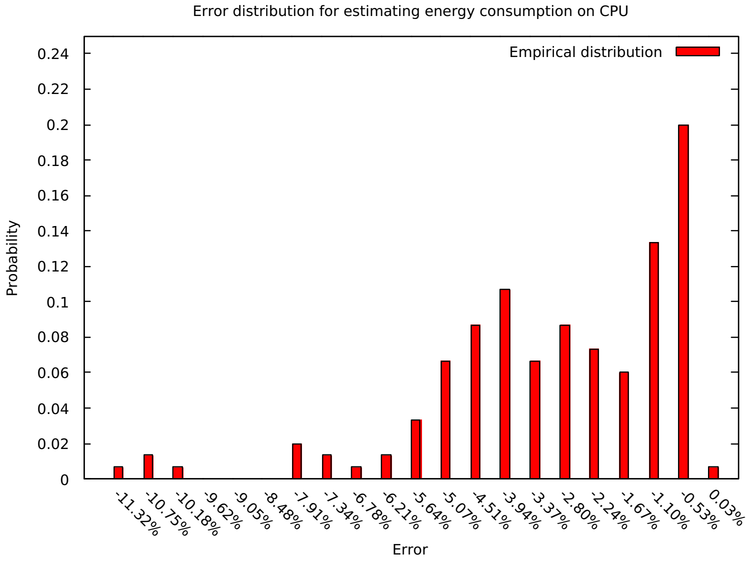


Figure 5.10: Error in energy estimation for map on vector with trigonometric functions.

The low value of the error in estimating energy on the CPU validates the approach followed in this work and makes this kind of model useful to estimate the power consumption of CPU map computations with low variance. While the power estimation is quite reliable, even in case of computations with scalability less than ideal, the model predicting time introduces additional error whenever the scalability is not nearly perfect.

This also provides an insight on the fact that low efficiency (in terms of time) have a correspondence on waste of energy.

Chapter 6

Using the models

In this Chapter we re-adapt the comprehensive energy model for heterogeneous architectures presented in Section 1.4 according to the observations and to the models or heuristic presented in this work. In the first section, we will address the problem of minimizing EDP. In the second one, we will provide some ideas on how to use the developed models.

6.1 Minimizing EDP

A system that uses both CPU cores and GPU devices to minimize energy consumption, as said before, needs a model to understand how to:

- divide the application between the different components
- execute the application within a single component

For this reason, we have developed in the previous Chapter proper analytical functions estimating energy, time and power of computations. We here present an aggregated energy model, taking into consideration both CPU and GPU. We start from the generic energy consumption for heterogeneous systems explained in Section 1.4, and in particular from Equations 1.14 and we rewrite it to take into consideration the difference between blocks and warps. We remember that the parameter n represents the number of CPU cores used to carry on the computation, m the number of workers on the GPU side and g the portion of work scheduled to the GPU: m should then be rewritten as (b, w), as we did before. Let us assume to apply a function f with very small variance to all of the M elements in the collection x over which the parallel map is executed. For the sake of simplicity, we will consider $M' = k \times M$ as the size of the data structure on which we are operating. Further, we will assume to give in input to the GPU $g \times M$ elements and to have in return $g \times M$ elements.

We will achieve a specialization of the terms of Eq. 1.14 as follows:

$$\begin{aligned}
E_{infr}(n,b,w,g) &= P_{infr} \times T_{CPU-GPU}(n,b,w,g) \\
E_{CPU\_active}(n,b,w,g) &= \hat{P}_{CPU}(n) \times n \times T_{CPU}(n,1-g) \\
E_{GPU\_active}(b,w,g) &= \hat{P}_{GPU}(b,w) \times T_{GPU}(b,w,g) \\
E_{GPU\_transfer}(g) &= \hat{P}_{GPU\_coprocessor} \times 2T_{send}(M' \times g) \\
E_{CPU\_idle}(n,b,w,g) &= U(2T_{send}(M' \times g) + T_{GPU}(b,w,g) - T_{CPU}(n,1-g)) \\
&\quad \times (2T_{send}(M' \times g) + T_{GPU}(b,w,g) - T_{CPU}(n,1-g)) \\
&\quad \times N \times P^{CPU}_{static} + (N-n) \times P^{CPU}_{static} \times T_{CPU}(n,1-g) \\
E_{GPU\_idle}(n,b,w,g) &= U(T_{CPU}(n,1-g) - 2T_{send}(M' \times g) - T_{GPU}(b,w,g)) \\
&\quad \times (T_{CPU}(n,1-g) - 2T_{send}(M' \times g) - T_{GPU}(b,w,g)) \\
&\quad \times P^{GPU}_{static}
\end{aligned} \tag{6.1}$$

Where:

$$T_{CPU}(n,k) = \frac{M \times k \times T_f^{CPU}}{n} \qquad n = 1 \dots N$$

and

$$T_{GPU}(b,w,g) \simeq \frac{M \times g \times T_f^{GPU}}{b \times w \times warp\_size} \qquad b = 1 \dots \#SM, w = 1 \dots 6$$

The total time for offloading the computation to the GPU is used will be:

$$T_{total\_GPU}(b,w,g) \simeq 2 \times T_{send}(M \times g) + T_{GPU}(b,w,g)$$

In order for T_{send} (see Section 4.3.2) to be equal both in transfers from host to device and from device to host, we need the memory to be pinned. From now on, we will assume to use only pinned memory. Remember that $T_{send}(k) = T_{setup} + k \times T_{trasm}$

Minimizing energy will hence require calculating the minimum of $E_C(n,b,w,g)$, composed by the sum of all terms defined above in 6.1, depending on 4 parameters.

When we consider as metric the widely used EDP (see Section 1.4.4), time is of the uttermost importance, impacting quadratically on the value of the metric. For this reason, we should *minimize* it. Since EDP corresponds to $Power \times Time^2$ and power grows linearly in the number of processors on the CPU side, and sublinearly on the GPU one, we can safely assume that this metric is minimized for the maximum parallelism degree on both devices, assuming perfect scalability.

If we denote by B the maximum number of streaming multiprocessors, and W as the number of CUDA cores available in the architecture divided by the warp size (i.e. 6 in Nvidia Kepler case), we will have a completion time of:

$$max\{T_{GPU}(B, W, g) + 2 \times T_{setup} + 2 \times M' \times g \times T_{trasm}), T_{CPU}(N)\}$$

which is minimized for:

$$T_{CPU}(N, 1-g) = 2 \times (T_{setup} + T_{trasm} \times M' \times g) + T_{GPU}(B, W, g)$$

The grain minimizing time consumption g for a certain parallelism degree will be:

$$g = \frac{M \times T_f^{CPU} \times P - 2 \times T_{setup} \times N \times P}{2 \times N \times P \times T_{trasm} \times M' + M \times T_f^{GPU} \times N + M \times T_f^{CPU} \times P} \tag{6.2}$$

with $P = B \times W \times warp\_size$.

By minimizing the difference between the CPU and GPU execution, we also minimize the terms of $E_C(n, b, w, g)$ due to idle resources: E_{CPU_idle} and E_{GPU_idle}.

Consider the case for a matrix multiplication executed on an Nvidia K40m and on a Xeon E5-2650, previously used. By using the above method, when we partition a 9000×9000 `float` matrix in terms of rows between the devices, we get $g = 0.9080$; if we schedule 828 rows to the 16 CPUs available and 8172 rows to the GPU using the maximum parallelism degree, we achieve $T_{CPU} = 24.86s$ and $T_{GPU} = 24.70$, hence with a very small difference. Moreover, the energy consumption as a whole is $4749.22J$, of which only $8.52J$ are due to idle resources (the host part). The EDP is consequently ~ 117775.

In case the same computation is scheduled only on the CPU, we have an energy consumption of $22225J$, a completion time of $283.62s$ and an EDP value of 630345. In the case only the GPU is used, we consume $\sim 5206J$ in $\sim 27.30s$ with an EDP of ~ 142127. We see that with respect to using only the CPU with this method we achieve an EDP 5.63 times higher, while the version using only the GPU has an EDP of 1.09 times the heterogeneous version.

6.2 Other considerations

In order to minimize energy we need to find an optimum depending on the values of g, b, w and n. However, it is not granted that we have a global minimum, as several configurations might return the same energy consumption for the same computation. In Figure 6.1, we show the average power over 5 samples for vector addition performed on GPU, changing the number of blocks and warps used to carry on the computation.

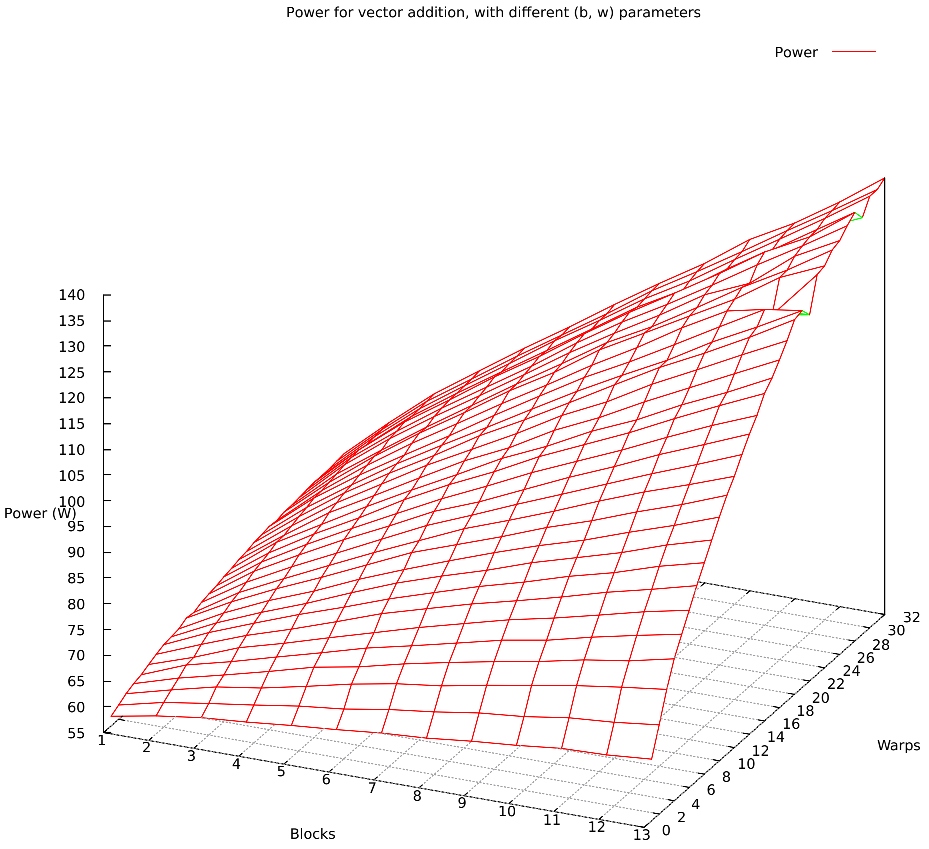


Figure 6.1: Power varying the number of blocks and of warps

The model can be simplified by removing some low interest cases:

- when a certain parallelism degree p on the GPU is desired, we can achieve it by different configurations of (b, w). In this case, if $w \times warp\_size$ is less than the number of available CUDA cores for a single streaming multiprocessor for the given architecture, the configuration with the minimal b should be preferred, as it provides the same completion time with less power consumption (see again 6.1).

- for some parallelism degrees, the efficiency is low and the energy consumption is hence too high: in these cases we consider both i) (too) unbalanced partitions and ii) parallelism degrees not mapping to processing units one to one

We will now provide considerations on the possible usage of the models in some scenarios.

6.2.1 Operating within a power budget

This can be done either to limit the temperature of the machine (affecting ddurability) or in the scenario in which the supply power is limited. In this case, we can use the model/the heuristics to select the configuration providing the best performances within the budget.

We can neglect the power (almost constant) consumed by the infrastructure (previously indicated with P_{infr}) and just subtract it from the threshold. We know that for both types of architectures the minimal power is expended for the minimal parallelism degree. So we can execute a proper portion g' of map tasks on the GPU architecture with $(1, 1)$ and by calculating α_C and using the heuristic $\hat{P}_C^{GPU}(b, w)$ achieve the maximum parallelism degree available on this architecture. We envision a similar heuristic to be developed also for CPU architectures; this task is left for future work. However, in case we have more than 1 NUMA node available on the architecture, we can launch a portion c' and $2c'$ of the map tasks in parallel with the GPU execution on the CPU, with parallelism degree respectively of 1 and 2. This would allow to estimate regression parameters, hence permitting to schedule the computation with the amount of workers giving the *faster* completion time while not overcoming the threshold.

Of course this method would work under the assumption that $P_{threshold} > P_{infr} + P_C^{GPU}(1, 1) + P_C^{CPU}(3)$. In fact, in order to carry on C on the CPU we need at least $P_{infr} + P_C^{CPU}(1)$, which is the cost of the infrastructure plus the minimum power required to operate using a single core. On the other hand, if we only use the GPU we need at least $P_{infr} + P_C^{GPU}(1, 1)$ watts in order to operate. Since the cost of infrastructure is assumed to be constant regardless of the computation that is carried on and we need to sample the power at least with two different parallelism degrees on the CPU in order to estimate power consumption, this is the minimum value of the threshold allowing to use the model.

6.2.2 Minimizing energy consumption

In case the function f of the map is expensive enough to be monitored when operating on a single task, or can be successfully averaged to be analysed in an aggregate manner it is possible to use the model to estimate the energy consumption of any configuration of the computation. We can use the energy estimation for different parallelism degrees on the CPU and different execution configurations (b, w) on the GPU. While performing this estimation could be quite expensive (there are more than 416 meaningful configurations on an Nvidia K20c), if we filter out the low interest cases above mentioned we can reduce the estimation to consider less than 1/4-th of the cases. This would allow to select the configuration (n, b, w) by using the two components configurations with the minimal energetic cost. The grain should be calculated according to Eq. 6.2, trying to reduce the difference between the two components parts as much as possible. Equation 6.2 can be used replacing B with b, W with w and N with n for the selected configuration.

For any triple (n, b, w), the equation returns a single value of g minimizing the difference between completion times. Hence, we will have a 3-dimensional table with parameters (n, b, w) in which $E[n][b][w] = (1-g)\times\hat{E}_{CPU}(n)+g\times\hat{E}_{GPU}(b, w)$. These values represent the expected energy cost of executing a single function with a certain parallelism degree on both device and host and with a certain grain. If we multiply the values of such table for the total number of data-parallel tasks, we can achieve as a result the estimation of energy consumption for the combined CPU/GPU execution with a certain parallelism degree and the consequent grain.

Practical example of energy minimization using the heuristic

We analysed the matrix multiplication computation used in Section 4.5.2. The implementation in the CPU divides the work in terms of rows (since the maximum parallelism degree is limited), while in the GPU we use the usual kernel of Listing 4.9. However, we give the capability to the computation to perform a split between the different devices in terms of rows, as it was done before in Section 6.1.

The following samples are performed on the architecture in order to instantiate the model for the considered computation:

1. The calculation of a row of the matrix on the CPU (NUMA Node 0) using a single core;

2. the calculation of the two following rows of the matrix on the CPU (NUMA Node 1) using two cores;

3. the calculation of the subsequent row of the matrix on the GPU (K40M) using 1 warp and 1 streaming multiprocessor.

By sampling such values, we can calculate:

1. β_0 and β_1 for the CPU part, using ordinary least square: this allows to find $\hat{P}_{CPU}(n) = \beta_0 + \beta_1 \times n$;

2. $T_{CPU}(n) = T_{CPU}(1)/n$ and by the product with $\hat{P}_{CPU}(n)$ the energy $\hat{E}_{CPU}(n)$ required to process (on average) a single row on the CPU with parallelism degree n;

3. $\tilde{\alpha}$ used for the heuristic as it was done in Section 4.5.2: this allows to calculate $\hat{P}_{GPU}(b, w)$ for the computation;

4. $T_{GPU}(b, w) = T_{GPU}(1, 1)/(b \times w)$ and by multiplying it for $\hat{P}_{GPU}(b, w)$ the predicted energy for processing a single row on the GPU with a certain number of blocks and warps

We now want an energy estimation: hence we perform it on the basis of the parameters (n, b, w), since the grain, given the model, depends only on these three parameters. We can reduce the size and the complexity of calculating the table by eliminating all the values for which $b \times w$ is the same, by simply removing the (more energetic costly) case in which b is higher.

The values of (n, b, w) for which the average energy for calculating a row is minimized is $(1, 14, 32)$, with an average estimated consumption of $0.4896J$ per row, followed by $(2, 14, 32)$ with $0.4942J$. For the former, the calculated grain will be 0.9931, hence in the case of a 9000×9000 computation, the GPU should calculate 8939 of the rows, while the single core used on the CPU is responsible for working over 61 rows.

By executing with these parameters, we achieve the values of Table 6.1.

Device	**Time**	**Energy**
CPU	$25.79s$	$1035.15J$
GPU	$27.01s$	$3064.32J$

Table 6.1: Energy and time for executing a 9000×9000 matrix multiplication on CPU/GPU cores mixes, using a single CPU core and 14 streaming multiprocessors with 32 warps each

The total energy is not the sum of the two, due to the unbalance between the two computations. In fact, we have $T_{CPU\_idle} = 1.22s$, hence $E_{CPU\_idle} = P_{waiting\_CPU} \times T_{CPU\_idle} \simeq 38.95J$. The total energy consumption for executing the computation on CPU/GPU core mixes taking advantage of the model is thus $\sim 4147.92J$.

We see that this result holds also if we use different parallelism degrees on the side of the CPU. If we maintain the same grain and we increase the parallelism degree on the CPU (so that the completion time decreases), we pay an additional amount of energy due to the CPU thread waiting for the GPU part of the computation to finish: this has been estimated to be a consumption of $\sim 40.1J/s$. In Table 6.2, it is possible to see the additional cost due to increase in the parallelism degree on the CPU. In practice, almost for every different value of n we observe an increased energy consumption.

CPU cores **n**	$\mathbf{T_{CPU}}$ (s)	$\mathbf{E_{CPU}}(J)$	$\mathbf{E_{tot}}(J)$
2	14.21	611.99	4231.946
4	6.86	333.066	4247.69
8	3.49	208.60	4280.06
16	1.92	147.06	4259.845

Table 6.2: Total energy for matrix multiplication using different parallelism degrees for the CPU offloaded part

Instead, if we use the value of g calculated in case of $(2, 14, 32)$ we can see that the energy is 4320.02, and the total time is 27.98, hence partially validating the method followed to search for a combination of (n, b, w, g) leading to decreased energy consumption. However, before being applied this method would need a more thorough evaluation with different parameters combination to evaluate its validity.

To demonstrate the energy efficiency of the achieved result, we show in Figure 6.2 the ratio between different, meaningful configuration and the best energy consumption result achieved ($4147.95J$).

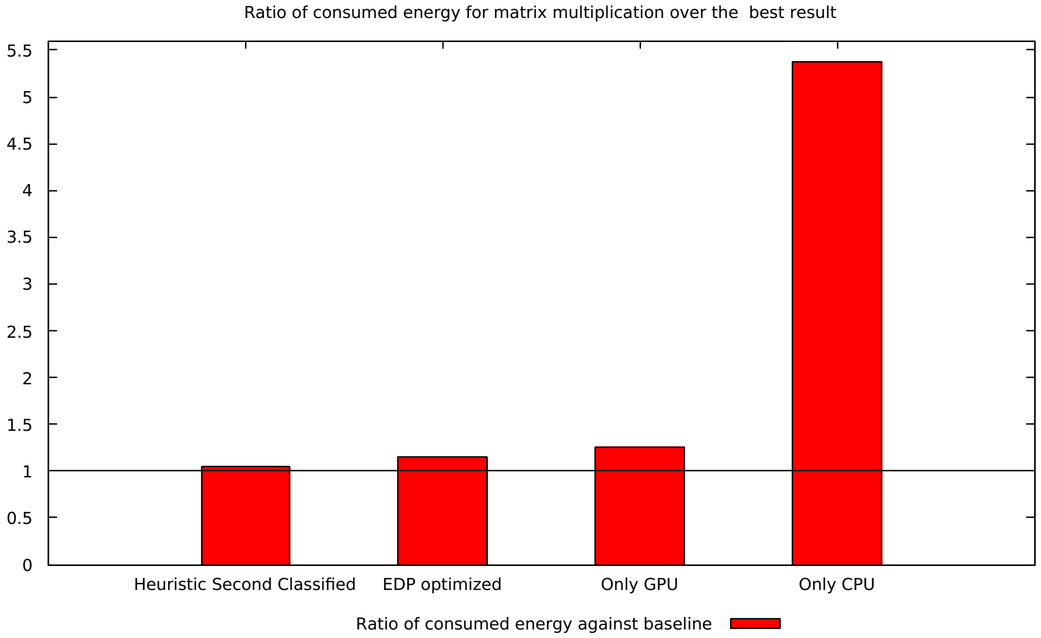


Figure 6.2: Ratio of energy consumed for meaningful configurations of matrix multiplication over the best result suggested by the model

In the case the computation is executed with $(2, 14, 32)$ (the second classified configuration according to the heuristic), the energy consumption is the 4.17% higher. Instead, if we use the EDP optimized case we consume 14% more Joules. The difference is even higher when we use only the GPU using all resources to carry on the computation (26% more energy) or only on the CPU with the maximum parallelism degree (537% more energy).

Conclusion and future work

In this thesis we addressed the problem of estimating energy and power consumption of data-parallel, map computations. The regular structure of this kind of computation is suitable to support the design of effective models relative to the energetic behaviour of a system. Given the presence of tested and well-established models predicting the completion time of data-parallel map computation, we only need to study the power requirements of a computation. However, since all resources deployed on the job are similar and execute similar instructions, the power requirements are regular.

With the achieved high-level map model, we demonstrated for a selected computation the feasibility of an approach using the model to minimize energy consumption. We were able to save the 14% of the energy consumed with the maximum parallelism degree on both devices, of the 26% on the version running only on the GPU with the maximum parallelism degree and of the 537% with respect to the CPU case. This shows that this model can be used successfully in order to minimize energy consumption, as well as the fact that co-scheduling map computations to CPU/GPU core mixes can bring significant advantages from the energetic standpoint.

We started by giving an introduction on the basic concepts useful for this thesis (Chapter 1) and by providing a small survey (Chapter 2) on the methods for reducing energy consumption currently adopted by the it community.

We then moved (Chapter 3) to define an iterative process allowing to individuate a model for energy consumption. This process has been validated by means of *actually modelling* the power and time (and consequently energy) requirements in different architectures. While the methodology developed resembles the *CROP model*, used by data analysts, the small differences make its definition of interest.

We applied the methodology to develop regression-based models for the GPU architecture (Chapter 4). Since energy is dissipated also when the computation is observed and when samples are taken to develop the regression model, we proceeded by reducing progressively the samples, finding a way to estimate the regression parameters without need to have too much information on the behaviour of the computation. Not only the regression model provided an useful way to estimate power consumption, but it allowed us to have an insight on the way different components affect power requirements. The model developed estimates the energy consumption of a computation with an error below the 15% using only six sampled values, hence with a very small computational and energetic footprint. By using the modelling information, we developed a heuristic that allows to predict GPU power consumption depending on the ratio between the sampled power and the power of a computation used as a meter. This method predicts energy consumption with less than 10% error with a probability exceeding 85% and by using a single, random sample of the effective power consumption. This heuristic is based only on the relationship of the computation with the metric and on the parallelism degree. To the best of our knowledge, there are currently no similar, high-level heuristics for GPU architectures.

We validated the possibility of using this heuristic on different devices of the same family (Nvidia Kepler) applying the same methodology to different devices, achieving similar error profiles both in energy and power prediction.

After validating with proper statistical analysis the reliability of the model and the heuristic in predicting energy requirements, in Chapter 5 we faced the problem of understanding the energetic requirements of CPU architectures. On this subjects, much more work has been performed and there are already models addressing accurate energy estimation in CPU architectures. We simply presented a technique based on regression that, at the cost of two samples, allows to understand how much energy will be consumed by a map computation on a multi-core environment.

Finally, Chapter 6 combines the results achieved in the previous part in order to provide some insight and to depict some useful scenarios in which the models could be fruitfully used.

Future works

Even though the provided models for GPU are very precise as far as power estimation is concerned, we lack precision when we estimate the timing of the computations. This is an issue, as in general it could be convenient to use more threads than effective execution units. A more sophisticated model should consider and face this issue, allowing to harness at its best the computational power of Graphics Processing Units. The case for more blocks than Streaming Multiprocessors should similarly be studied. Another possible development is the provisioning, starting from the same concepts used for GPUs, of a similarly precise heuristic for estimating power in the CPU case. This would reduce the energetic footprint used for the model.

We did not considered, in this work, the impact of other well-known parallel programming patterns, like reduce. Its study would be of particular interest as many computations require a combination of map and reduce higher order functions. Other parallel patterns (like *scan* or *parallel prefix*) should also be taken into consideration.

Finally, from a more practical perspective, an interesting work could be the development of a behavioural skeleton, supporting the co-scheduling of computations on both CPU and GPU and using the developed models and heuristics, minimizing energy consumption.

Bibliography

[ACD$^+$08] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons for component autonomic management on grids. In *Making Grids Work*, pages 3–15. Springer, 2008.

[ACD$^+$13] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *Euro-Par 2012: Parallel Processing Workshops*, pages 47–56. Springer, 2013.

[ADKT11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multicore.(a fastflow short tutorial). *Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing*, 2011.

[AL06] Bovas Abraham and Johannes Ledolter. *Introduction to regression modeling.* Thomson Brooks/Cole, 2006.

[Amd67] 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.

[BAM98] R Bahar, Gianluca Albera, and Srilatha Manne. Power and performance tradeoffs using various caching strategies. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 64–69. IEEE, 1998.

[BBL$^+$11] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, Albert Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82(2):47–111, 2011.

[BLO08] Giulio Boccaletti, Markus Löffler, and Jeremy M Oppenheim. How it can cut carbon emissions. *McKinsey Quarterly*, 37:37–41, 2008.

[BM11] Roberto Bevilacqua and Ornella Menchi. *Appunti di calcolo numerico.* 2011. http://www.di.unipi.it/ bevilacq/Dispensa11-12.pdf.

[BS00] J Adam Butts and Gurindar S Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201. ACM, 2000.

[CA12] Andrew S Cassidy and Andreas G Andreou. Beyond amdahl's law: An objective function that links multiprocessor performance gains to delay and energy. *Computers, IEEE Transactions on*, 61(8):1110–1126, 2012.

[CAAB15a] Alberto Cabrera, Francisco Almeida, Javier Arteaga, and Vicente Blanco. Energy measurement library (eml) usage and overhead analysis. 2015.

[CAAB15b] Alberto Cabrera, Francisco Almeida, Javier Arteaga, and Vicente Blanco. Measuring energy consumption using eml (energy measurement library). *Computer Science-Research and Development*, 30(2):135–143, 2015.

[CAB13] Alberto Cabrera, Francisco Almeida, and Vicente Blanco. Eml, an energy measurement library. In *IFIP WG 7.3 Performance 2013 31 st International Symposium on Computer Performance, Modeling, Measurements and Evaluation 2013 Student Poster Abstracts September 24-26, Vienna, Austria*, page 5, 2013.

[CFMC10] Antonio Cisternino, Paolo Ferragina, Davide Morelli, and Massimo Coppola. Information processing at work: On energy-aware algorithm design. In *Green Computing Conference, 2010 International*, pages 407–415. IEEE, 2010.

[CGM$^+$10] Alexandre Carissimi, Claudio FR Geyer, Nicolas Maillard, Philippe OA Navaux, Gerson GH Cavalheiro, Mauricio L Pilla, Adenauer C Yamin, Andréa S Charao, Benhur Stein, César AF De Rose, et al. Energy-aware scheduling of parallel programs. In *Conferência Latino Americana de Computação de Alto Rendimento (CLCAR). Gramado,:[sn]*, pages 95–101, 2010.

[CGM14] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[CSB92] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. Low-power cmos digital design. *IEICE Transactions on Electronics*, 75(4):371–382, 1992.

[CSP05] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):18–28, 2005.

[CY12] Jason Cong and Bo Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 345–350. ACM, 2012.

[Dan14] Marco Danelutto. *Distributed Systems: paradigms and models.* September 2014.

[DLK13] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the skepu skeleton programming library. In *Advanced Parallel Processing Technologies*, pages 170–183. Springer, 2013.

[DPR08] Gaurav Dhiman, Kishore Kumar Pusukuri, and Tajana Rosing. Analysis of dynamic voltage scaling for system level energy management. *USENIX HotPower*, 8, 2008.

[EK10] Johan Enmyren and Christoph W Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[EK12] Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, 2012.

[EKD$^+$03] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18. IEEE, 2003.

[ENE07] STAR ENERGY. Program: Report to congress on server and data center energy efficiency. *Public Law*, pages 109–431, 2007.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[FRM01] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 260–271. ACM, 2001.

[GA06] Ed Grochowski and Murali Annavaram. Energy per instruction trends in intel microprocessors. *Technology@ Intel Magazine*, 4(3):1–8, 2006.

[Gal02] G Gallo. Note di simulazione, 2002.

[GGGV12] Mehdi Goli, Michael T Garba, and Horacio Gonzalez Velez. Streaming dynamic coarse-grained cpu/gpu workloads with heterogeneous pipelines in fastflow. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 445–452. IEEE, 2012.

[GH96] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, 1996.

[GK10] Michael Garland and David B Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[Har13] Mark Harris. Unified memory in cuda 6, 2013.

[HK10] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[HQ03] Shaoxiong Hua and Gang Qu. Approaching the maximum energy saving on embedded systems with multiple voltages. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 26. IEEE Computer Society, 2003.

[HVC+06] Santa Concepcion Huerta, M Vasic, A de Castro, Pedro Alou, José Cobos, et al. Review of dvs techniques to reduce power consumption of digital circuits. In *Integrated Power Systems (CIPS), 2006 4th International Conference on*, pages 1–6. VDE, 2006.

[HXF09] Song Huang, Shucai Xiao, and Wu-chun Feng. On the energy efficiency of graphics processing units for scientific computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[IM02] Anoop Iyer and Diana Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 379–386. ACM, 2002.

[IM03] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[JM01] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140. ACM, 2001.

[KA11] Vijay Anand Korthikanti and Gul Agha. Energy-performance tradeoff analysis of parallel algorithms for shared memory architectures. *Sustainable Computing: Informatics and Systems*, 1(3):167–176, 2011.

[KBB$^+$08] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, W Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[KBSSK13] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. Towards power efficiency on task-based, decoupled access-execute models. In *PARMA 2013, 4th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures, Berlin, Germany, January 23, 2013*, 2013.

[Kec11] S Keckler. Life after dennard and how i learned to love the picojoule. In *Keynote at the he 44th Intl. Symposium on Microarchitecture*, 2011.

[KG97] Milind B Kamble and Kanad Ghose. Analytical energy dissipation models for low power caches. In *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, pages 143–148. IEEE, 1997.

[KGC99] Gangadhar Konduri, James Goodman, and Anantha Chandrakasan. Energy efficient software through dynamic voltage scheduling. In *IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS*, pages 358–361. Citeseer, 1999.

[KM08] Stefanos Kaxiras and Margaret Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[KSH$^+$06] Hideaki Kimura, Mitsuhisa Sato, Yoshihiko Hotta, Taisuke Boku, and Daisuke Takahashi. Emprical study on reducing energy of parallel programs using slack reclamation by dvfs in a power-scalable high performance cluster. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2006.

[Li08] Keqin Li. Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *Parallel and Distributed Systems, IEEE Transactions on*, 19(11):1484–1497, 2008.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, (2):39–55, 2008.

[LSH10] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8. USENIX Association, 2010.

[Lue08] David Luebke. Gpu architecture: Implications & trends. *ser. SIGGRAPH*, 2008, 2008.

[LZCH14] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. An investigation of unified memory access performance in cuda. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[Maj10] Shikharesh Majumdar. On the energy-performance tradeoff for parallel applications. In *Computer Performance Engineering*, pages 67–82. Springer, 2010.

[Mar00] Diana Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *Workshop on Complexity-Effective Design*, volume 42. Citeseer, 2000.

[McC10] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 2010.

[MCC15] Davide Morelli, Andrea Canciani, and Antonio Cisternino. A high-level and accurate energy model of parallel and concurrent workloads. *Concurrency and Computation: Practice and Experience*, 2015.

[Mil10] Rich Miller. Exascale computing = gigawatts of power, 2010.

[MLC$^+$12] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 48–57. IEEE, 2012.

[MNP02] Alain J Martin, Mika Nyström, and Paul I Pénzes. Et2: A metric for time and energy efficiency of computation. In *Power aware computing*, pages 293–315. Springer, 2002.

[MPAM13] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D Medeiros. Algorithmic skeleton framework for the orchestration of gpu computations. In *Euro-Par 2013 Parallel Processing*, pages 874–885. Springer, 2013.

[MVF00] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Advanced Research in Asynchronous Circuits and Systems, 2000.(ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 52–59. IEEE, 2000.

[ND10] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, (2):56–69, 2010.

[NVIa] NVIDIA. Geforce gtx580 architecture.

[nVib] nVidia. What is gpu accelerated computing?

[NVI09] NVIDIA. Fermi whitepaper v 1.1. Technical report, NVIDIA, 2009.

[NVI12] NVIDIA. Kepler gk110 whitepaper. Technical report, NVIDIA, 2012.

[NVI15] NVIDIA, https://docs.nvidia.com/cuda/pdf/ CUDA _C_Programming_Guide.pdf. *CUDA C Programming Guide, version 7.5*, September 2015.

[OHL$^+$08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[PLS10] Jeff Pool, Anselmo Lastra, and Montek Singh. An energy model for graphics processing units. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 409–416. IEEE, 2010.

[PS06] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006.

[RN05] Stuart J Russell and Peter Norvig. *Intelligenza artificiale. Un approccio moderno*, volume 1. Pearson Italia Spa, 2005.

[RSR$^+$08] Mahsan Rofouei, Thanos Stathopoulos, Sebi Ryffel, William Kaiser, and Majid Sarrafzadeh. Energy-aware high performance computing with graphic processing units. In *Proceedings of the 2008 conference on Power aware computing and systems*, pages 11–11. USENIX Association, 2008.

[S$^+$90] Takayasu Sakurai et al. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. *Solid-State Circuits, IEEE Journal of*, 25(2):584–594, 1990.

[SAD$^+$02] Greg Semeraro, David H Albonesi, Steven G Dropsho, Grigorios Magklis, Sandhya Dwarkadas, and Michael L Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 356–367. IEEE, 2002.

[SD95] Ching-Long Su and Alvin M Despain. Cache designs for energy efficiency. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 1, pages 306–315. IEEE, 1995.

[SDK13] Tudor Serban, Marco Danelutto, and Peter Kilpatrick. Autonomic scheduling of tasks from data parallel patterns to cpu/gpu core mixes. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 72–79. IEEE, 2013.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[SKK11] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[SKZ] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing.

[SMB$^+$02] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, and Michael L Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40. IEEE, 2002.

[SR12] Sonal Saha and Binoy Ravindran. An experimental evaluation of real-time dvfs scheduling algorithms. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 7. ACM, 2012.

[Van14] Marco Vanneschi. *High Performance Computing - parallel processing models and architectures*. Pisa University Press, 2014.

[VGD$^+$11] Jeffrey S Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford, et al. Keeneland: Bringing heterogeneous gpu computing to the computational science community. *Computing in Science and Engineering*, 13(5):90–95, 2011.

[VLWYH09] Gregor Von Laszewski, Lizhe Wang, Andrew J Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[Wei55] Martin H Weik. A survey of domestic electronic digital computing systems. Technical report, Ballistic Research Labs., Aberdeen Proving Ground, Md., 1955.

[Wik15] Wikipedia. Polynomial regression, 2015.

[WPW00] Qing Wu, Massoud Pedram, and Xunwei Wu. Clock-gating and its application to low power design of sequential circuits. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 47(3):415–420, 2000.

[WR10] Guibin Wang and Xiaoguang Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 122–129. IEEE, 2010.

[WWDS96] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1996.

[YCK05] Chuan-Yue Yang, Jian-Jia Chen, and Tei-Wei Kuo. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 468–473. IEEE Computer Society, 2005.

[ZBSF04] Bo Zhai, David Blaauw, Dennis Sylvester, and Krisztian Flautner. Theoretical and practical limits of dynamic voltage scaling. In *Proceedings of the 41st annual Design Automation Conference*, pages 868–873. ACM, 2004.

Ringraziamenti

Dopo aver scritto questo enorme papiro, non potevo esimermi dall'aggiungere una ulteriore sezione per i ringraziamenti. Chiunque abbia visto quanto mi piace parlare non poteva dubitarne. Nella prima sottosezione verranno adeguatamente elencati e ringraziati, in ordine sparso, vari concetti, teoremi e documenti che mi hanno causato gli incubi durante il corso di questi anni universitari. Analizzeremo poi nel dettaglio gli individui che in qualche maniera mi hanno aiutato durante questi anni, o che semplicemente ritengo di dover elencare per completezza scientifica.

Concetti, teoremi e documentazione

Il mio primo ringraziamento non può che andare ai draghi, fili d'erba, poligoni e vari virtuosismi in 3D di NVidia (Figura 1). Onnipresenti in qualsivoglia documento tecnico rilasciato dai suddetti, hanno reso lo studio delle architetture GPU molto *fantasy* e quasi altrettanto divertente. Ovviamente sono proprio l'ideale quando uno prova a decrittare le loro architetture: almeno ci si rassegna e si capisce che sono magiche.

Figure 1: Tipica immagine presente in un Whitepaper Nvidia. DRAGHI.

In questa sezione ovviamente non può mancare il simpaticissimo algoritmo di *Range Minimum Query*, grazie al quale si può fare una ricerca su un array (statico) in $O(log(log(N)))$. Ricordando che il logaritmo di 1 Tera è 40, quindi con circa un accesso al disco per elemento con una normale ricerca binaria andiamo nell'ordine dei 0.2 secondi, prima di ammortizzare il tempo che ho utilizzato per provare a capire la riduzione RMQ -> LCA -> RMQ usata ad Algorithms Engineering, serviranno almeno quei 10 anni di esecuzione.

Menzione d'onore per diversi mostri trattati nel corso di SPQT. Incontrarvi è stato un onore, dimenticarvi è stato automatico.

Tra le cose più interessanti studiate durante questi anni, non posso esimermi dal nominare il fantastico teorema di *Myhill-Nerode*. Risultato fondamentale usato dagli informatici di 5 continenti (ma grazie al teorema potrebbero essere anche di piú), rimane uno dei più grandi misteri della triennale in informatica.

Theorem 1. *Un linguaggio $\mathcal{L}$ è accettato da un ASFD se e solo se è l'unione di* ***alcune*** *classi di equivalenza di una relazione invariante destra di indice finito.*

Ovviamente il fatto che la sapessi dimostrare non ha nessuna correlazione col fatto che io non abbia capito a cosa servisse.

L'RFIX e il circostante `makefunrec` (Listing 1) ha cambiato completamente la mia vita. La totale non-comprensibilità del frammento di codice mostrato mi ha causato incubi per mesi. E comunque nessuno mi convincerà mai che funzioni.

```
type eval = |Int of int | Bool of bool | Unbound | Funval of efun
  and efun = expr * eval env
and makefunrec(i, el, (r:eval env)) =
  let functional (rr: eval env) =
    bind(r, i makefun(el, rr)) in
      let rec rfix = function x -> functional rfix x
      in makefun (el, rfix)
```

Listing 1: MakeFunRec: come trovare il punto fisso in pochi semplici passi e vivere felici.

En passant, volevo anche ricordare al prof. Abate, autore del libro di Algebra Lineare che romani (ed etruschi, mi dicono dalla regia) costruivano ponti senza autovalori ed autovettori... e che quella introduzione mi ha fatto veramente paura.

Un grazie anche alla mia prima lezione di fisica, dove dopo aver abbassato un secondo lo sguardo si è passati come per magia dalla nave che andava da Livorno all'Elba sad avere degli integrali grandi circa mezza lavagna. Circa 3 anni dopo ho capito anche a cosa servisse.

Dedicherei un momento speciale alla parte del mio cervello che mi gestisce i sogni e i deliri. In particolare vorrei ricordare il fantastico sogno di una ricerca interminabile della calcolatrice scientifica in un mondo completamente violaceo, gli alieni che mi visitano spesso mentre dormo (e che puntualmente al mio risveglio mi aspettano in qualche libro) e l'architettura degli elaboratori che mi ha fatto immaginare PC e PO durante i sogni di varie notti di mezz'estate 2011.

Persone, non-persone e piante

Il primo ringraziamento non può che andare al mio relatore, Marco Danelutto, per la pazienza nel leggere interminabili papiri sia durante la tesi che durante il progetto di SPM.

Ovviamente non posso che ringraziare i compagni d'armi Michele Carignani, Cesare Bassu e Luca Atzori, con i quali ho condiviso buona parte del mio percorso universitario, nonché questo ultimo periodo estremamente vario della mia vita e la festa che faremo sabato (in Figura 2 una illustrazione da due sabati fa). Poveri noi. In particolare vogliamo ricordare:

- Michele per il divertentissimo progetto di PAD: 3 mesi pieni per 6 CFU sembra proprio che sia stato il caso;
- Cesare per gli attraversamenti pericolosi alla striscia di Gaza, lo studio della summenzionata `rfix` e di tutto PR2, nonché per essere stato uno dei pochi con il coraggio di rivolgermi la parola durante Analisi. È solo grazie a lui se questo elenco puntato finisce così -> ;
- Luca per il progetto di SPM. Verrà ricordato principalmente in quanto in grado di produrre, al costo di sole due ore/uomo (alle 04.30 del mattino), la grammatica mostrata in Listing 2, aggiungendo "il Danelutto apprezzerà senz'altro". Se poi si sia rivelato vero o meno rimarrà per sempre un mistero.

Figure 2: Due sabati fa. Divertimento

```
A simple grammar that generates (some of) the string that could
   be used to compile!

S' -> make S
S -> all | mic-all | strassen-all | T | strassen-T
T -> mic-D | host-D
D -> double | float
```

Listing 2:

Seguendo l'esempio Atzoriano, si fornisce una sintetica quanto completamente innecessaria grammatica dei ringraziamenti in Listing 3.

```
ACK -> Ringrazio Formal_Acks | Ringrazio Other_ACKS;
Formal_ACKS -> PROF per suggerimenti a vario titolo durante la
   stesura della tesi;
PROF -> Giancarlo Bigi | Antonio Cisternino | Massimo Torquati;
Other_Acks -> Friends per REASONS | Family per REASONS;
REASONS -> l'aiuto o suggerimenti durante la stesura della tesi |
    la compagnia | il delirio notturno | avvermi offerto una
   birra quando ormai volevo tornare a casa | avermi salvato da
   un investimento (da parte di un'auto) | la pazienza incessante
    dimostrata | avermi mantenuto agli studi | aver sentito un
   sacco di cavolate | i viaggi (mentali e non);

Friends -> GIRLFRIENDS | Antonella C. | Antonella N. | Claudia |
   Bob | Raffo |  Sasa' | Giuseppe Doto | Francesco Tribenga
   Quaranta | MV | Zoba | Lallotta | Billy, Samba, Papa etc |
   Antonia Anna Rosa Maria etcetcetc | Stefania  | Luca | Cesare
   | Margherita | Daniele C. | Michele | Jacopo | Eleonora P. |
   Marco G. | Roberto L. | Marco P. ... ;
GIRLFRIENDS -> Laura;
```

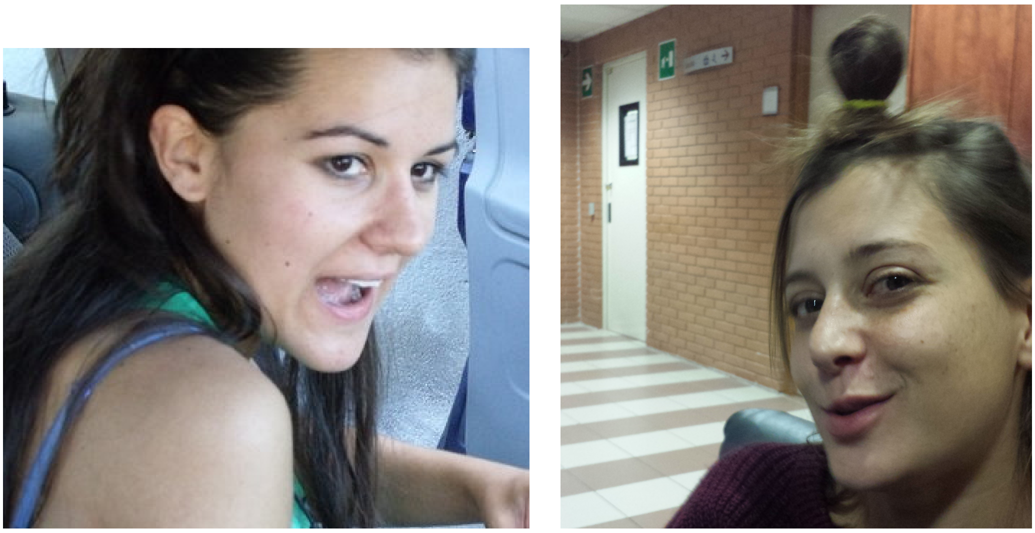
Listing 3: Grammatica dei ringraziamenti

Si noti che il linguaggio dei ringraziamenti generati dalla grammatica non risulta essere esaustivo e contemporaneamente contiene più ringraziamenti del dovuto, esattamente come la grammatica del Makefile. Lo sviluppo di una grammatica corretta è parte dei lavori futuri. Per tale ragione abbandoneremo l'approccio formale per ringraziare, invece, una persona alla volta.

Iniziamo quindi dai colleghi, ringraziando Marco Grandi (Figura 3) per essere stato un ineffabile compagno di studio da Marzo ad oggi e soprattutto per avermi ricordato in continuazione che 'sti terroni non lavorano abbastanza. Gratzias meda fintzas a Roberto Ladu, pro m'aer cussizadu de m'iscriere a custu cursu de Laurea.

Figure 3: A lavorare!

Thanks to all of my C.S. and Networking colleagues, with whom I shared most of my time at least in the first years of the degree and when I was able to get up on time.



(a) Eleonora impegnata in attività lecite. Non spreca alcol.

(b) La Sili all'apice della sua eleganza

Figure 4: Individue informatiche.

Le ragazze di informatica sono *troppe* per essere nominate tutte. Ma come dimenticare Margherita (visibile in tutta la sua grazia in Figura 4b), che ringrazio aver portato costantemente "allegria", anche quando poteva non farlo, in un dipartimento altrimenti triste e buio e Eleonora (Figura 4a), con la sua proverbiale contesa con Marco Grandi riguardo allo spreco di alcool, le sue apparizioni in perenne ritardo e il pigiama cucito sulla pelle.

Alle due viterbesi: tenete duro, il Piano Marshall 2 per la vostra provincia sta arrivando, serviva giusto il tempo di laurearci.

Un grazie particolare a Tribenga e Doto (Figura 5) per le multiple ... fregature (è pur sempre una tesi) raccattate in Spagna. Secondo me sarebbe il caso di iniziare a frequentare il Nord Europa. Purtroppo a Doto non sono bastate e ha ben deciso di fare un'erasmus a Madrid. Si ringrazia anticipatamente l'agenzia funebre che ci riporterà le sue spoglie truffate a morte. Tribenga invece ha saggiamente migrato verso il Nord Italia a seguire i dollaroni. Aspettiamo sue notizie da allora, ma voci di corridoio dicono che sia diventato il capo dei broker milanesi. Un grazie in anticipo perchè sento già che la prossima crisi sarà colpa tua.

Grazie ad Antonia Turi, venditrice seriale (una sera mi son distratto e mi son trovato sul groppone un mutuo per una casa in Tibet), psicopatica, zitella (si fa per dire) e convinta di essere la più bella di Pisa.

Come non menzionare le persone che hanno aiutato il mio fegato a soffrire durante questi anni. Oltre ai precedenti Stefania (desparecida con le lasagne), Antonella Nisi (e il suo superpotere), Antonella Catte (che non ha superpoteri, ma ha un bel salotto presso p.za Sant'Omobono), Bob (con Claudia) e Claudia (coi gatti e altro bestiame vario) , Sasà (mbriacone), Raffo (che conquista), MV

Figure 5: Due persone sperdute.

(che risulta sempre misteriosa) e *todos los otros*[1] hanno portato un contributo cruciale nel farmi passare dei bei week-end etilici. Siccome questo documento finirà su ETD e lì risiederà per tempo imperituro, la maggiore motivazione per fare questa tesi è stata la mia volontà di danneggiarvi la reputazione. Alcuni dei summenzionati individui sono visibili in Figura 6. Le altre persone che hanno attraversato serate, giornate e occupazioni pisane sono innumerevoli. Mi sembra che visto il numero enorme di persone ringraziate, vi potete accontentare della grammatica (se ci siete) oppure consideratevi citati in qualche anfratto del mio cervello.

Figure 6: Alcune persone molto serie

[1] spagnolo impeccabile.

Tra le non-persone (orwellianamente parlando, per carenza di ortodossia), un grazie a Lallotta (con la quale mi scuso, perché in questo momento mi attende al Fibonacci a suo rischio e pericolo), Zoba, Matteo, Carmen, Francesco, Lucheddu e Simona dell'oramai estinta Arbeschida.

Grazie alla piantina nell'angolo del Dipartimento, per essere una fonte di ossigeno fondamentale, mentre aspetto di capire come si possa fare la fotosintesi.

Un enorme grazie a Laura (Figura 7) per aver sopportato tutti i miei deliri per anni e anni. Non deve essere stato facile avere a che fare con tutti i miei scleri: gli ultimi mesi ancora di meno. Senza la sua presenza e pazienza sarebbe stato tutto molto più difficile. Grazie.

Figure 7: Una persona estremamente paziente. In primo piano una studentessa di Giurisprudenza.

Ultimi, ma non perciò meno importanti i miei soci finanziatori e i miei co-debitori: i miei genitori e i miei fratelli, i quali sono sempre stati presenti a livello di bonifico i primi e a livello di prestito gli altri.

Meta-ringraziamenti

Si ringrazia Cesare per il prezioso aiuto nello scrivere i ringraziamenti, Luca per essersi rotto ed essere diventato irrimediabilmente autistico ed infine i ringraziamenti stessi per esserci sempre stati dandomi la possibilità di sfogarmi liberamente.