



UNIVERSITÀ DI PISA

Computer Science Department

Thesis for Master Degree in Computer Science

**GP-GPU:
from Data Parallelism
to Stream Parallelism**

Candidato: **Maria Chiara Cecconi**

Relatore: **Marco Danelutto**

Contents

1	Introduction	1
1.1	Goals	1
1.1.1	GPU Architecture and Data Parallelism	2
1.1.2	Other Applications	3
1.1.3	GP-GPUs and Stream Parallel	6
1.2	Expectations	9
1.3	Results	10
1.4	Tools	11
2	Tools	13
2.1	NVIDIA Architecture and CUDA	14
2.1.1	Copy engines and Memory organization	19
2.2	CUDA C/C++	23
2.2.1	Kernels	23
2.2.2	Thread Hierarchy	24
2.2.3	CUDA Streams	25

2.2.4	<code>nvcc</code> compiler	27
2.2.5	<code>cuda-gdb</code> debugger	28
2.3	Profilers	29
2.3.1	<code>nvprof</code>	29
2.3.2	NVIDIA Visual Profiler	30
2.4	Visual Studio Code	31
2.5	Tests, Result gathering, Plots	31
2.5.1	Bash scripts	32
2.5.2	Python scripts	33
3	Project Logic	37
3.1	Stream Parallelism: Farm pattern	37
3.1.1	Farm performance model	39
3.2	CPU-GPGPU: heterogeneous architecture	43
3.2.1	Overlapping: Data Transfer hiding	44
3.2.2	Occupancy of GPU cores	47
3.2.3	Occupancy drawbacks	50
3.3	Overall Logic	52
3.4	Tunings	59
3.4.1	Tuning on block and grid dimensions	61
4	Implementation	63
4.1	Kernels	63
4.1.1	Simple-computational kernel	64
4.1.2	Matrix multiplication	65
4.1.3	Blur Box filter	67
4.2	Parallel Patterns implementation on GPU	69
4.2.1	Stream Parallel on GPU	69
4.2.2	Data Parallel un GPU	74

5	Experiments	77
5.1	Expectations	77
5.1.1	Measures: What and How	78
5.1.2	Tests setup	83
5.1.3	Speedup	84
5.1.4	Results: gathering and evaluation	85
5.1.5	Computation-bound and memory-bound	89
5.2	Simple-computation kernel	93
5.2.1	Results	97
5.3	Matrix Multiplication	105
5.3.1	Results	108
5.4	Image processing	121
5.4.1	Results	123
5.5	Results Summary	125
5.5.1	Stream parallel compared to Data parallel	126
6	Conclusions	133
6.1.2	Evaluation of the problem	134
6.1.3	Implementation and tests	137
6.1.4	Results and considerations	138
6.1.5	Final remarks and further works	140

CHAPTER 1

Introduction

In a scenario where image processing needed to get more and more sophisticated, we saw *graphic processors* follow the change getting increasingly powerful, not only in computation speed but also in flexibility.

The new elasticity provided by **GPUs** made possible to exploit their benefits for a wide range of non-image-processing problems. This is the beginning of **GP-GPUs** era.

Despite this, enthusiasm slowed down when scientific community had to deal with problems that seemed to be unsuitable for GP-GPUs. However, several studies and researches showed some good results and possibilities, giving oxygen to keep trying mapping to GPU apparently unsuitable problems. This is the core of this work too.

1.1 Goals

The main goal of this thesis is to study the behavior of GPUs when used for different purposes, with respect to the usual ones. In particular, we want to use a GPU to run

code that models after a *stream parallel pattern*, to understand if we can exploit *Streaming Multiprocessors* (SMs) in parallel and what is the relative efficiency. This means we investigated the possibility to perform stream parallelism among "small" data parallel tasks, exploiting the different Streaming Multiprocessors, available on the GP-GPU, as workers.

Then we observed ongoing, in terms of *completion time* and *speedup*.

The latter considered as sequential version the case without using CUDA Streams, while the parallel version using them. From these speedups we expected to see an improvement slightly below the number of used CUDA Streams, giving us an assessment on the number of employed Streaming Multiprocessors.

The different experiments¹, gave us the results that we expected. In particular we observed that, using CUDA streams, is possible to achieve a gain proportional to the number of SMs available on a GPU. We could see this behavior especially in arithmetic-intensive applications².

In next sections, we're going to show some preliminary details about the concepts we've just introduced.

1.1.1 GPU Architecture and Data Parallelism

GPU (*Graphics Processing Unit*) is a co-processor, generally known as a highly parallel multiprocessor optimized for parallel graphics computing.

Compared with multicore CPUs, manycore GPUs have different architectural design points, one focused on executing many parallel threads efficiently on many cores. This is achieved using simpler cores and optimizing for data parallel behavior among groups of threads[1].

In most of situations, visual processing can be associated to a *data parallel pat-*

¹Experiments will be presented in detail in Chapter 5.

²We'll show in Chapter 4 the different considered and implemented applications in this work. Chapter 5 will show all results and speedups relative to the different study cases.

tern.

In general, we can roughly think to an image as a given and known amount of *independent* data upon which we want to do the same computations, over the whole collection. In other words, generally, once the proper granularity of the problem has been chosen, a certain work should be done for each portion of the image.

Considering the above scenario and given that generally a GPU should have to process huge amount of data, we wish to have a lot of threads (lot of cores consequently) doing "the same things" on all data portions.

And that's why GPUs performs their best on data parallel problems.

1.1.2 Other Applications

As we've just mentioned, GPUs have been always known to perform at their best on data parallel problems.

However, in recent years, we're moving to **GP-GPUs** (***General-Purpose computing on Graphics Processing Units***). In other words, lately GPUs have been used for applications other than graphics processing.

One of the first attempts of executing non-graphical computations on a GPU was a matrix-matrix multiply. In 2001, low-end graphics cards had no floating-point support; floating-point color buffers arrived in 2003.

For the scientific community the addition of floating point meant no more problems on fixed-point arithmetic overflow.

Other computational advances were possible thanks to programmable shaders³, that broke the rigidity of the *fixed graphics pipeline*.

A ***Graphic Pipeline*** (or rendering pipeline) is a conceptual model that describes what steps a graphics system needs to perform to render a 3D scene to a 2D screen[4].

³For example LU factorization with partial pivoting on a GPU was one of the first common kernels, that ran faster than an optimized CPU implementation.

Pipeline is generally defined on two levels, i.e. software API level and hardware implementation level. However, it's possible to conceptually define three main stages⁴:

- user input, developer controls what the software should execute, passing geometry primitives to next stage;
- geometry processing, it executes per-primitive operations, in this phase we've dense computing;
- scan conversion phases, it handles per-pixel operations.

Only the first stage was programmable in early GPUs, that's why they're often referred to as *fixed-function graphics pipelines*.

Modern graphics hardware, instead, provides developers the ability to customize features in the two last stages⁵, with this growth in flexibility started the *programmable pipeline* era[5].

We can find pipeline concepts in NVIDIA GPUs too. In fact, each SM of a CUDA device, provides numerous hardware units that are specialized in performing specific task⁶. At the chip level those units provide execution pipelines to which the warp schedulers dispatch instructions[6].

Now we'll focus on the NVIDIA parallel computing platform used in this thesis, i.e. CUDA⁷.

The introduction of **NVIDIA's CUDA** (***Compute Unified Device Architecture***)

⁴So, given that a graphic pipeline consists of several consequential stages, the service time is given by the slowest stage.

⁵Generally this is achieved with two techniques: *vertex shader* and *fragment shader*

⁶For example, texture units provide the ability to execute texture fetches and perform texture filtering, load/store units fetch and save data to memory etc.

⁷We'll show some further informations about CUDA in Section 1.4.

in 2007, ushered a new era of improved performance for many applications as programming GPUs became simpler: terms such as texels, fragments, and pixels were superseded with *threads*, *vector processing*, *data caches* and *shared memory* [2].

From the very beginning of GP-GPUs, scientific general purpose applications on GPUs started from matrix (or vector) computations, that mainly could be referred to as **Data parallel** problems. But over time scientific community felt the need to cover other applications, that not necessarily fit data parallel model. In particular some of latest researches are moving towards **Task parallel** applications (sometimes also known as *Irregular-Workloads parallel patterns*)[3].

An example of non-data parallel problem is the *backtracking paradigm*. It's often at the core of compute-and-memory-intensive problems and we can find its application in different cases, such as: constraint satisfaction in AI, maximal clique enumeration in graph mining and k-d tree traversal for ray tracing in graphics.

Some computational parallel patterns perform effectively on a GPU, while the effectiveness of others is still an open issue.

For example, in several studies it was highlighted that memory-bound algorithms on the GPU perform at the same level or worse than the corresponding CPU implementation.

Task-parallel systems must deal with different situations, with respect to those present in data parallelism, e.g.:

- Handle divergent workflows;
- Handle irregular parallelism;
- Respect dependencies between tasks;
- Implementing efficient load balancing.

Those requirements can lead to inefficient use of the GPU memory hierarchy and

SIMD-optimized⁸ GPU multi-processors.

However, there have been backtracking-based or other task-parallel algorithms successfully mapped onto the GPU: the most visible example is in *ray tracing* rendering technique; other examples are *H.264 Intra Prediction* video compression encoding, *Reyes Rendering* and Deferred Lighting.

However, in general we cannot expect an order of magnitude increase in performance. Rather, a more realistic goal is to perform at one-two times the CPU performance, which opens up the possibility of building future non-data-parallel algorithms on heterogeneous hardware (such as CPU-GPU clusters) and performing workload-based optimizations [3].

1.1.3 GP-GPUs and Stream Parallel

In this work we were interested to a particular type of task parallelism:

Stream parallelism.

This means that our tasks are elements from an input stream, of which we don't know a priori the length or the interval times between tasks.

Once the stream elements are available, parallel workers will make independent computations over them and, finally, the manipulated elements should be delivered to some output stream.

We recall as main stream parallel patterns the *Farm* and the *Pipeline*, the former being the main subject in this thesis.

The **Farm parallel pattern**⁹ is used to model embarrassingly parallel computations. This pattern computes in parallel the same function $f : \alpha \rightarrow \beta$ over all the items appearing in the input stream of type α **stream** delivering the results on the output stream of type β **stream**.

⁸We'll see SIMD architecture, CUDA memory hierarchy and other architectural details in Section 2.1.

⁹The Farm parallel pattern will be seen in detail in Chapter 3.

The model of computation of the task-farm pattern consists of three logical entities: the *Emitter*, that is in charge of accepting input data streams and to assign the data to the Workers; a pool of *Workers* which compute the function f in parallel over different stream elements; the *Collector* that non-deterministically gathers Workers' partial results and eventually produces the final result.

The Emitter, the set of Workers and the Collector interact in a pipeline way using a data-flow model which can be implemented in several different ways depending on the target platform. For example, the Emitter and Collector, could be implemented in a centralized way using a single thread, or in a partially or fully distributed way.

Farm's workers can be formed by any other pattern[7]. In the general case, input/output data ordering may be altered due to the different relative speeds of the workers executing the distinct stream items. If ordering is important, it can be enforced by the Collector or by the scheduling/gathering policies of the farm pattern.

Master-Worker is a specialization of the task-farm pattern where the Emitter and Collector are collapsed in a single entity (called *master*).

The Workers deliver computed results back to the master. The master schedules received input tasks toward the pool of workers trying to balance their workload[19].

In the case of this thesis, we exploited *Streaming Multiprocessors as Farm Workers*, each computing one or more kernel executions, so the function f , in our case, is given by the kernel code. Furthermore we assumed that Emitter and Collector were both managed by CPU-side.

This means that we considered *GPU as Worker* and *CPU as Master*, in particular our implementation is roughly organized as follows:

- **host-side** code (code executed by CPU) manages input stream and forwards items to GPU according to a *Round-Robin* scheduling policy, furthermore host manages results arriving from the GPU;
- **device-side** code (code executed by GPU) mainly executes the worker function

f , here called *kernel*¹⁰.

Since several kernel calls are executed in parallel by a certain SM, we assumed Streaming Multiprocessors to be the workers.

For completeness, we'll also briefly introduce *data parallelism*.

It refers to those problems where more workers perform the same task on different portion of data. Generally this is achieved having different parallel entities (e.g. threads), such that they execute the same code on different parts of the input data structure[19].

One of the most important data parallel pattern, is *Map*. It computes a given function $f : \alpha \rightarrow \beta$ over all the data items of an input collection whose elements have type α . The produced output is a collection of items of type β . Given the input collection x_1, x_2, \dots, x_N , the output collection is y_1, y_2, \dots, y_N where $y_i = f(x_i)$ for $i = 1, 2, \dots, N$. Here each data item in the input collection is independent from the other items, so all the elements can be computed in parallel [19, 7].

The model of computation of the map pattern is very similar to the one described for the farm pattern.

The key difference is in the input/output data type:

- *data structures* for Map;
- *streams of items* for Farm.

So, the farm pattern works on streams of independent data (a stream may be unbounded), while the map pattern receives a data collection, of a fixed number of items, that is partitioned among the available computing resources[7].

In other words, the general difference between data parallel and stream parallel pat-

¹⁰In Chapter 2, the concept of kernels will be explained, together with other main features from CUDA C/C++ language.

terns, is that in the latter neither the full sequence nor the number of items in the sequence are known in advance[9].

The above difference points out one of the main problems of this work: the *Data Transfer times* between **host memory** (CPU side) and **device memory** (GPU side), and vice versa.

In particular, host/device memory copies overhead is a problem because:

- it represents an amount of time spent in memory operations, instead of necessary computations;
- it introduces host/device synchronizations¹¹, for example GPU waits for input data copy to end and CPU waits for output data to be fully copied back¹².

So data transfers, may represent a bottleneck, especially with respect to the arrival rate of input stream items.

We'll show in detail all aspects of this and other problems, together with respective solutions, in Chapter 3.

1.2 Expectations

The main expectation was to show that a not suitable problem, such as Farm parallel pattern, could fit in a GPU architecture.

It's important to point out that, in particular, we're modeling a Streaming parallel pattern having small data parallel portions as tasks.

In other words, running on GPU our stream parallel code, it calls a stream of light kernels, each computing one of the small data parallel task from the Farm input stream.

Then, we wanted to see that in this way we could take an advantage in the order of the **SMs'** number, available in the target GPU.

¹¹This holds for the synchronous version of the command for host/device data transfers `cudaMemcpy`.

¹²We'll see in Chapters 2-3 that we'll try to hide this overload by using asynchronous calls.

Looking closer at that this expected results, it means that:

- Data transfer time has to be hidden, in some way, by computation time;
- Kernel executions should take enough time in computations, in order to have chances to achieve overlapping between different kernel executions;
- The GPU needs to achieve a good *Occupancy* ¹³.

Once we could achieve these factors, no matter what kind of feature GPU has, we expected to get a $Speedup \approx number\ of\ SMs$. The reason why we wanted to see such a speedup is all about possibly gaining some advantages, with respect to CPU processing:

- We can delegate stream parallel problems to the GPU while the CPU can compute other things, this allows to not saturate the CPU (especially when stream has high throughput or each element requires high computation intensity);
- We can split the amount of work between CPU and GPU, the best would be to give respective quantities based on completion time[8];
- We hopefully want to see a GPU speedup with respect to the CPU, or see the same performances at worst.

1.3 Results

At this point, it was useful to experiment different applications, i.e. different kernels codes. More precisely we implemented three types:

- **Compute-bound**, where the amount of computations, performed by the kernel, is greater than the amount of memory operations;
- **Memory-bound**, dominated by memory operations;

¹³We'll insist on occupancy topic in Chapter 3.

- **Divergent flows** (and memory bound too), this means we have a kernel with a lot of branching code¹⁴.

In the case of our kernels, memory-bound operations are mainly given by load/store from the Global Memory¹⁵ to registers and vice versa[18].

We observed that compute-bound kernel gave us the expected results for GPU Farm, that is gaining a good overlap between kernel executions and, so, a speedup proportional to SMs number. While memory-bound gave a really low amount of overlap and speedup and for the divergent flow case we've even worse performances, given that we had no speedup at all.

Anyway, the above results represent what we expected and we'll give all relative details in Chapter 5.

1.4 Tools

As mentioned in Subsection 1.1.2 we mainly exploited NVIDIA's CUDA Toolkit¹⁶. In particular:

- The code was implemented in CUDA C++ language, so the compiler was `nvcc`;
- The profiling of GPU code performances was supported by `nvprof` and by its advanced visual version `NVIDIA Visual Profiler`;
- The debugging was made by using `cuda-gdb`;
- Studies on GPU Occupancy have been done with *CUDA Occupancy Calculator spreadsheet* and *Occupancy APIs*.

Tests on the code were implemented as bash scripts and they've been run on two machines:

¹⁴Chapter 5 explains details about compute-bound, memory-bound and divergent flows in kernels.

¹⁵In Chapter 2 we'll see an overview on the memory organization in GPUs.

¹⁶Chapter 2 will show features and details about CUDA Toolkit and all other tools that have been used.

- The first with four NVIDIA GPUs **Tesla P100-PCIE-16GB**;
- The second with four NVIDIA GPUs **Tesla M40**.

The code was developed with the following environments:

- *Visual Studio Code* for CUDA C++, Makefile, bash scripts;
- *Gedit* for Python scripts.

In next chapters all the concepts briefly outlined in this introduction will be discussed in depth.

Chapter 2 introduces an accurate description of all employed tools and how they were used.

Then Chapter 3 explains the logic of the project, with both text and graphical illustrations. In other words here we point out the main ideas and concerns, together with relative solutions, behind our approach.

Chapter 4 presents and explains main implementation choices and there will be listed some fundamental part of the code.

Then Chapter 5 shows how both experiments and tests are set, then the obtained results and some respective plots will be presented.

Finally, conclusions give an overall view of the thesis, some final remarks and future works suggestions.

CHAPTER 2

Tools

Everything in this project has been developed using a GNU/Linux environment. We had two available remote computers, to which we connect via `ssh` command.

In particular we worked on the following machines:

1. **Local host**

- Ubuntu 14.04 LTS, (4.4.0-148-generic x86_64)
- 1 CPU Intel® Core™ i5 CPU M 450 @ 2.40GHz x 4

2. **P100 remote server**

- Ubuntu 18.04.2 LTS (4.15.0-43-generic x86_64)
- 40 CPUs Intel® Xeon® CPU E5-2698 v4 @ 2.20GHz, 2 way hyperthreading
- 4 GPUs Tesla P100-PCIE-16GB, 56 SMs with 64 CUDA cores each, 16281 MBytes total Global Memory, 1.33GHz

3. M40 remote server

- Ubuntu 16.04.6 LTS (4.4.0-154-generic x86_64)
- 24 CPUs Intel® Xeon® CPU E5-2670 v3 @ 2.30GHz, 2 way hyperthreading
- 4 GPUs NVIDIA Tesla M40, 24 SMs with 128 CUDA cores each, 11449 MBytes total Global Memory, 1.11GHz

Given that this work is focused on the use of the remote GPUs, their main specifics are listed in Table 2.1, where all the contained informations have been obtained by executing `cudaDeviceQuery` application (located inside samples of CUDA Toolkit).

In this work we mainly used the different tools available in the CUDA Toolkit. In the following section will be presented all of employed stuff, with some specifications and how they've been exploited during this project.

2.1 NVIDIA Architecture and CUDA

The NVIDIA GPU architecture is built around a scalable array of multithreaded **Streaming Multiprocessors** (SMs).

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one SM, and multiple thread blocks can execute concurrently on one multiprocessor.

As thread blocks terminate, new blocks are launched on the vacated multiprocessors[10].

A multiprocessor is designed to execute hundreds of threads concurrently. To manage them, it employs a unique architecture called **SIMT** (***Single-Instruction, Multiple-Thread***) that is described in SIMT Architecture. SIMT Architecture and Hardware Multithreading describe the architecture features of the streaming multiprocessor that are common to all devices.

	Tesla P100	Tesla M40
Driver/Runtime Version	10.1	10.1
CUDA Capability	6.0	5.2
Tot. global memory amount	16281 MBytes	11449 MBytes
Multiprocessors	56	24
CUDA Cores/MP (Tot. CUDA cores)	64 (3584)	128 (3072)
GPU Max Clock rate	1329 MHz (1.33 GHz)	1112 MHz (1.11 GHz)
Tot. amount constant memory	65536 bytes	65536 bytes
Tot. amount shared memory/block	49152 bytes	49152 bytes
Tot. #registers available/block	65536	65536
Warp size	32	32
Maximum #threads/multiprocessor	2048	2048
Max #threads/block	1024	1024
Max thread block dimensions (x,y,z)	(1024, 1024, 64)	(1024, 1024, 64)
Max grid size dimensions (x,y,z)	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)
Concurrent copy & kernel exec	Yes with 2 copy engine(s)	Yes with 2 copy engine(s)

Table 2.1: GPUs specifics for the two remote machines employed in this project.

The multiprocessor creates, manages, schedules, and executes *threads in groups of 32 parallel threads* called *warps*. Individual threads, composing a warp, start together at the same program address, but they have their own instruction address counter and register state and are, therefore, free to branch and execute independently.

The term warp originates from weaving, the very first parallel thread technology. A half-warp is either the first or second half of a warp[1, 10].

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a **warp scheduler** for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive and increasing thread IDs, with the first warp containing thread 0[10]. **Thread Hierarchy** describes how thread IDs relate to thread indexes in the block, it will be discussed in the Subsection 2.2.2.

A warp executes one common instruction at a time, so *full efficiency is realized when all 32 threads of a warp agree on their execution path*. If threads of a warp diverge via a *data-dependent conditional branch*, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths[10]. This is the situation that we wanted to observe in one of our applications, i.e. *kernel with divergent flows*, as we mentioned in Chapter 1.

The SIMT architecture is akin to **SIMD** (***Single Instruction, Multiple Data***) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can

essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge[28, 14].

The threads of a warp that are participating in the current instruction are called the **active threads**, whereas threads not on the current instruction are *inactive* (disabled). Threads can be inactive for a variety of reasons including having exited earlier than other threads of their warp, having taken a different branch path than the branch path currently executed by the warp, or being the last threads of a block whose number of threads is not a multiple of the warp size[10].

Hardware Multithreading

The execution context (program counters, registers, etc.) for each warp, processed by a SM, is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution context to another has no cost, and at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction (the active threads of the warp) and issues the instruction to those threads.

In particular, each multiprocessor has a set of 32-bit **registers** that are partitioned among the warps, and a parallel data cache or **shared memory**¹ that is partitioned among the thread blocks.

The number of blocks and warps that can reside and be processed together on the multiprocessor, for a given kernel, depends on:

- the amount of registers and shared memory used by the kernel;
- the amount of registers and shared memory available on the SM;
- maximum number of resident blocks per SM;
- maximum number of resident warps per SM.

¹In next section the GPU memory hierarchy will be explained.

These limits are a function of the **compute capability**² of the device.

If there are not enough registers or shared memory available per SM to process at least one block, the kernel will fail to launch.

The total number of warps in a block is as follows:

$$\text{ceil}(\frac{Th}{W_{size}}, 1)$$

Where Th is the number of threads per block, W_{size} is the warp size (which is equal to 32), $\text{ceil}(x, y)$ is equal to x rounded up to the nearest multiple of y [10].

CUDA overview

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model provided for compute engine in NVIDIA GPUs.

Three key abstractions are supported, they are exposed to the programmer using a minimal set of language extensions:

- A hierarchy of thread groups;
- Shared memories;
- Barrier synchronization.

These abstractions provide fine-grained data parallelism and *thread parallelism*, nested within coarse-grained data parallelism and *task parallelism*.

This makes possible to partition the problem into coarse sub-problems –solved independently in parallel by *blocks* of threads –, and each sub-problem into finer pieces –solved cooperatively in parallel by all *threads* within the block –.

²Compute Capability is a classifications of features and technical specifications associated to each compute device.

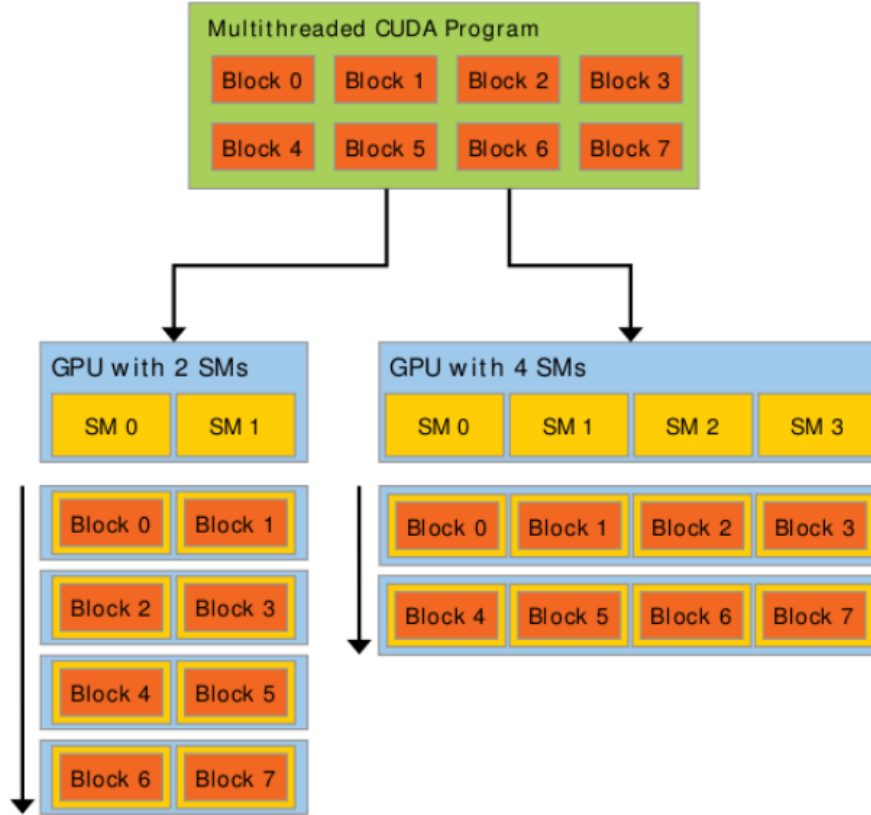


Figure 2.1: GPU scalability.

Indeed, *each block of threads can be scheduled on any of the available Streaming Multiprocessors within a GPU, in any order, concurrently or sequentially*, such that a compiled CUDA program can execute on any number of SM, as illustrated by Figure 2.1, and only the runtime system needs to know the physical multiprocessor count. This programming model scales on the number of multiprocessors and memory partitions[10].

We'll see in next section further details on CUDA programming model, especially the extensions for C/C++ since, in this thesis, we used CUDA C++.

2.1.1 Copy engines and Memory organization

On all operating systems that run CUDA, host memory is virtualized. The operating system component that manages virtual memory is the *Virtual Memory Manager*

(VMM), it provides services to hardware drivers, in order to facilitate direct access of host memory by hardware.

In modern computers, many peripheral, as GPUs, can read or write host memory using a facility known as ***Direct Memory Access*** (DMA). It avoids a data copy and enables the hardware to operate concurrently with the CPU³.

To facilitate DMA, operating system VMMs provide a service named *page-locking*. Page-locked memory is ineligible for eviction and its physical addresses cannot change. Once memory is page-locked, drivers can program their DMA hardware to reference the physical addresses of the memory. Memory that isn't page-locked is known as *pageable*, otherwise it's defined as ***pinned*** memory⁴.

We'll see that ***host pinned memory***⁵, with respect to the GPUs, is generally coupled with CUDA streams and it represents a page-locked portion of host memory, set up for DMA by the current CUDA context[24].

We'll see in Subsection 2.2.3 CUDA Streams and we'll see that they are linked to DMA for another reason. In particular, an important DMA mechanism is the so called ***copy engine***.

DMA allows the transfer of data between host and device, while eventually a kernel is executing on the GPU. In older GPU architectures there was a single copy engine, while in newer there are usually two. The benefit of dual copy engines, coupled with the fact that PCIe⁶ is a full duplex interconnection, allows to perform more operations simultaneously, for example:

- do a memory copy from device to host;
- Run kernel that operates on available data in memory;

³Furthermore hardware may achieve better bus performances over DMA.

⁴It's called *pinned* since its physical addresses cannot be changed by the OS.

⁵To be precise page-locked memory and CUDA's pinned aren't really the same thing: pinned memory, registered by CUDA, is mapped for direct access by the GPU; ordinary page-locked memory is not.

⁶*Peripheral Component Interconnect Express*, abbreviated as PCIe, is a high-speed serial computer expansion bus standard, designed to replace the older PCI.

- do another memory copy from host to device.

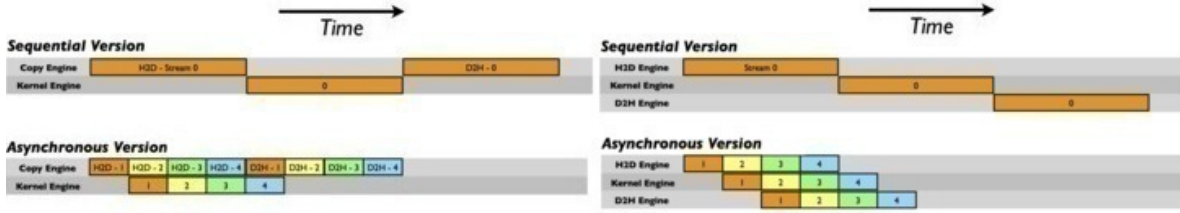


Figure 2.2: Comparison between one and two copy engines, using default (above) and non-default CUDA streams (below).

In the ideal case, with two copy engines the data copies should be completely overlapped by kernel execution, i.e. the transfers overheads are hidden by the kernel time⁷. With a single copy engine, instead, first and last steps above cannot run concurrently⁸[24, 20].

In Figure 2.2, starting from the top, we have a comparison between the cases of: memory copies and kernel executions completely serial; one memory copy and kernel execution overlapping (one copy engine); two memory copies and kernel execution overlapping (two copy engines).

To maximize performance, CUDA uses different types of memory, depending on the expected usage.

As mentioned before, host memory refers to the memory attached to the CPU(s) in the system. Device memory is attached to the GPU and accessed by a dedicated memory controller and data should be copied explicitly⁹ between host and device memory in

⁷We'll see what overlapping really means and how it's implemented in subsection 2.2.3

⁸Note that the simultaneous upload/download of large amounts of data across PCIe can use up a significant portion of available system memory bandwidth.

⁹A new CUDA functionality, called *Unified Memory* allows to abstract both device and host memories as if they are a unique memory, this avoids the use of explicit copies between host and device memory. However this feature wasn't used in this work, as it wasn't suitable to allow tests and measures we needed to perform in experiments.

order to be processed by the GPU.

Device memory can be allocated and accessed in different of ways:

- **Global memory** may be allocated statically or dynamically and accessed via pointers in CUDA kernels, which translate to global load/store instructions;
- **Constant memory** is read-only memory accessed via different instructions that cause the read requests to be serviced by a cache hierarchy, optimized for broadcast to multiple threads;
- **Local memory** contains the stack, that is local variables that cannot be held in registers, parameters, and return addresses for subroutines;
- **Texture memory** (in the form of CUDA arrays) is accessed via texture and surface load/store instructions; like constant memory, read requests from texture memory are serviced by a separate cache that is optimized for read-only access[24].

Shared memory is an important type of memory in CUDA that is not supported by device memory. Instead, it is an abstraction for an on-chip memory that can be used for fast data interchange between threads within a block, has much higher bandwidth and lower latency than local and global memory[18]. Physically, shared memory comes in the form of built-in memory on the SM.

In Figure 2.3 In this work we mainly exploited Global memory, it is the main abstraction by which CUDA kernels read or write device memory.

The device pointer base resides in the device address space, separate from the CPU address space used by the host code in the CUDA program. As a result, host code in the CUDA program can perform pointer arithmetic on device pointers, but they may not dereference them. CUDA kernels can read or write global memory using standard C semantics[24].

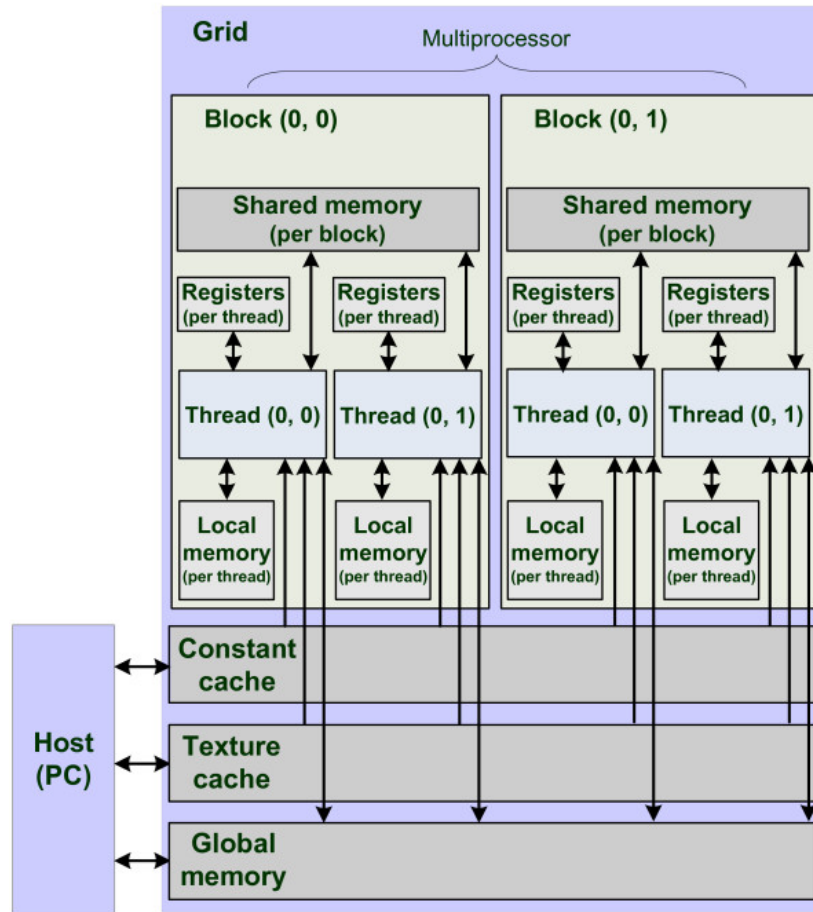


Figure 2.3: CUDA memory hierarchy and SM organization.

2.2 CUDA C/C++

Here we'll briefly introduce main concepts behind the CUDA programming model, by outlining how they are exposed in C/C++. Especially we'll show important notions about features involved in this project and how/why these were included.

2.2.1 Kernels

CUDA C allows to define particular C functions, called *kernels*, when called, the code is executed N times in parallel by N different CUDA threads, instead of executing only once, like it happens in regular C functions[10].

A kernel code is defined using the `__global__` declaration specifier. The number of

CUDA threads that will execute the kernel for a given call is specified using this special *execution configuration* syntax: `<<<...>>>`.

Each thread executing the kernel is given a unique thread ID, accessible within the kernel through the built-in `threadIdx` variable^[10].

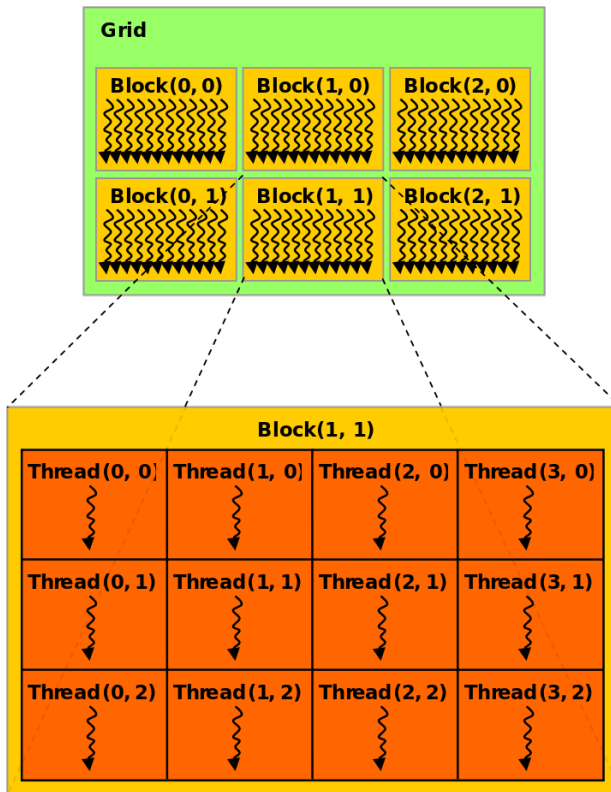


Figure 2.4: Grid/Block organization: above a Grid formed by Blocks; below a Block formed by Threads.

2.2.2 Thread Hierarchy

In practice `threadIdx` is a 3-component vector, so that threads can be identified using either one, or two, or three dimensional thread index. In turn these threads will form either one, or two, or three dimensional block of threads, called a *thread block*. This provides a way to invoke computation across the elements in domains such as a vectors, matrices, or volumes.

There is a limit to the number of threads per block, since *all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core*. On current

GPUs, and on the two we worked on, a thread block may contain up to 1024 threads¹⁰. However, a kernel can be executed by multiple equally-shaped thread blocks, so that

¹⁰see Table 2.1 for limits in the machines we used.

Total number of threads = #threadsPerBlock · #blocks

Blocks in turn are organized into either one, or two, or three dimensional ***grid of thread blocks*** as illustrated by Figure 2.4. So, the number of blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system. The number of *threads per block* and the number of *blocks per grid* specified in the <<<...>>> syntax can be of type `int` or `dim3`.

The dimension of the thread block, block index and thread index are accessible within the kernel through the respective built-in variables: `blockDim`, `blockIdx`, `threadIdx`[10].

2.2.3 CUDA Streams

CUDA enables fine-grained concurrency, with hardware facilities that enable threads to closely collaborate within blocks, using a combination of shared memory and thread synchronization.

But it also has hardware and software facilities that enable more *coarse-grained concurrency*[10]:

- *CPU/GPU concurrency.* Since they are separate devices, the CPU and GPU can operate independently of each other;
- *Memcpy/kernel processing concurrency.* For GPUs having one or more copy engines, *host* ↔ *device* `memcpy` can be performed while the SMs are processing kernels;
- *Kernel concurrency.* SM, in 2.x compute capability and later hardware, can run up to 4 kernels in parallel;
- *Multi-GPU concurrency.* For problems with enough computational density, multiple GPUs can operate in parallel.

CUDA streams enable these types of concurrency, but we're interested in the first three types listed above.

Within a given stream, operations are performed in sequential order, but operations in different streams may be performed in parallel.

CUDA events complement CUDA streams by providing the synchronization mechanisms needed to coordinate the parallel execution enabled by streams. CUDA events may be used for CPU/GPU synchronization, for synchronization between the engines on the GPU, and for synchronization between GPUs.

They also provide a GPU-based timing mechanism that cannot be perturbed by system events such as page faults or interrupts from disk or network controllers [24, 10].

Overlap of Data Transfer and Kernel Execution

Some devices can perform an asynchronous memory copy to or from the GPU concurrently with kernel execution. It's possible to query this capability by checking the `asyncEngineCount` device property¹¹, which is greater than zero for devices that support it. When host memory is involved in the asynchronous copy, it must be **page-locked** to possibly achieve a kernel-copy overlap[27]. Applications manage the concurrent operations described above through **CUDA streams**.

A stream is a sequence of commands (possibly issued by different host threads) that execute in order. On the other hand, a stream may execute its commands out of order or concurrently with respect to another; this behavior is not guaranteed and should not be relied upon for correctness (e.g. inter-kernel communication is undefined)[10, 20].

A brief code example:

Listing 2.1: CUDA Strams creation

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
```

¹¹ See *Device Enumeration* in Table 2.1, for both of our machines, from the *deviceQuery*, we get 2 copy engines.


```
float* host;
cudaMallocHost(&host, 2 * size); //host mem allocation must be pinned
```

On each of these CUDA streams will be issued a sequence of commands. In the following code sample we have one memory copy *host* \rightarrow *device*, one kernel launch, and one memory copy *host* \leftarrow *device*:

Listing 2.2: CUDA Strams and Async example

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev+i*size, host+i*size, size, cudaMemcpyHostToDevice,
        stream[i]);
    MyKernel<<<BLOCK,GRID,0,stream[i]>>>(outputDev+i*size, inputDev+i*size, size);
    cudaMemcpyAsync(host+i*size, outputDev+i*size, size, cudaMemcpyDeviceToHost,
        stream[i]);
}
...
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Each stream copies its portion of input array *host* to array *inputDev* in device memory, calls *MyKernel()* to process computations on *inputDev* and copies the result *outputDev* back to the same memory portion, i.e. *host*. Note that *host* must point to page-locked host memory for any possible overlap to occur. Streams are finally released by calling *cudaStreamDestroy()*.

2.2.4 nvcc compiler

To compile the CUDA C++ code it is necessary to use a special compiler, included in CUDA Toolkit, that is *nvcc*.

The CUDA compiler driver hides the details of CUDA compilation from developers. It accepts a range of conventional compiler options, for example for the project we could define macros or include library paths. All non-CUDA compilation steps are forwarded to a C++ host compiler that is supported by *nvcc*. Source files compiled with *nvcc* can include a mix of host code and device code[21].

`nvcc`'s basic workflow consists in separating device from host code and then:

- compiling the device code into an assembly form (PTX code) and/or binary form (cubin object),
- and modifying the host code by replacing the `<<<...>>>` syntax introduced in Kernels¹² by the necessary CUDA C runtime function calls, to load and launch each compiled kernel from the PTX code and/or cubin object.

The modified host code is output either as C code that is left to be compiled using another tool, or as object code directly by letting `nvcc` invoke the host compiler during the last compilation stage. Applications can then either link to the compiled host code (most common case), or ignore the modified host code (if any) and use the CUDA driver API to load and execute the PTX code or cubin object[21].

This compiler can be used almost the same way as a classic `gcc`, for example:

```
nvcc -std=c++14 -g -G -o exec source.cu
```

This is an example that compiles the file `source.cu` in the executable file `exec`, keeping debug information with `-g -G` flags. Here we present compiler versions installed in our two machines:

- **Tesla P100**

`nvcc`: NVIDIA® Cuda compiler driver, release 10.1, V10.1.168

- **Tesla M40**

`nvcc`: NVIDIA® Cuda compiler driver, release 10.1, V10.1.105

2.2.5 `cuda-gdb` debugger

CUDA-GDB is the NVIDIA tool for debugging CUDA applications (available on Linux). It is an extension to the x86-64 port of GDB, the GNU Project debugger[22].

CUDA-GDB main features are:

¹²See subsection 2.3.1

- it provides an environment that allows simultaneous debugging of both GPU and CPU code within the same application;
- as programming in CUDA C is an extension to C programming, debugging with CUDA-GDB is an extension to debugging with GDB, so the existing GDB debugging features are present for debugging the host code (additional features have been provided to support CUDA device code);
- it allows to set breakpoints, to single-step CUDA applications, and also to inspect and modify the memory and variables of any given thread running on the hardware;
- it supports debugging all CUDA applications, whether they use the CUDA driver API, the CUDA runtime API, or both.
- it supports debugging kernels that have been compiled for specific CUDA architectures, but also supports debugging kernels compiled at runtime, referred to as just-in-time (JIT) compilation[\[22\]](#).

CUDA-GDB was used to debug all compiled source files, both `.cu` and `.cpp`. Mainly it was really helpful in this project to step device code, inspect runtime errors thrown by CUDA API calls and check for errors in the code inside our kernels.

2.3 Profilers

NVIDIA profiling tools were useful to optimize performances of our CUDA applications. We used two different versions, that will be showed in the following sections.

2.3.1 nvprof

Then `nvprof` provides a tool to collect and view profiling data from the command-line. `nvprof` was added to CUDA Toolkit with CUDA 5. It is a command-line profiler, so

it's a GUI-less version of the graphical profiling features available in the NVIDIA Visual Profiler.

The `nvprof` profiler enables the collection of a timeline of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, memory set and CUDA API calls and events or metrics for CUDA kernels.

After all data is collected, profiling results are displayed in the console¹³ or can be saved in a log file for later viewing ¹⁴.

`nvprof` operates in different modes: *Summary Mode*, *GPU-Trace and API-Trace Modes*, *Event/metric Summary Mode* and *Event/metric Trace Mode*[11, 12].

For our purposes we used only *Summary Mode*, this is the default operating mode, where we have a single result line for each kernel function and each type of CUDA memory copy performed by the application (for each operation type are shown total, maximum, minimum, average times and number of calls)[11].

We used it, in some situations, as a quick check, for example we exploited it to see if the application wasn't running kernels on the GPU at all, or it was performing an unexpected number of memory copies, etc. To this aim it's enough to run the application with

```
nvprof ./myApp arg0 arg1 ...
```

Given that we even wanted to consult profiling results whenever it was necessary, we used `--log-file` option to redirect the output to files for deferred examination.

`nvprof` revealed peculiarly suitable for remote profiling. That's because of the fact command line is faster to check and save an application profiling.

2.3.2 NVIDIA Visual Profiler

The NVIDIA Visual Profiler, introduced in 2008, is a performance profiling tool providing visual feedback for optimizing CUDA C/C++ applications.

¹³The textual output of the profiler is redirected to `stderr` by default.

¹⁴Or for later import into either `nvprof` or the NVIDIA Visual Profiler.

The Visual Profiler displays a timeline of an application’s activity on both the CPU and GPU to make performance improvement, it analyzes the application to detect potential bottlenecks, using graphical views, that allow to check memory transfers, kernel launches, and other API functions on the same timeline [11].

Visual Profiler was also used in developing phase, to check if the application gave overlapping, or to see if we could hide as much as possible data transfers¹⁵, even though sometimes it may happen that profilers introduce some sampling synchronization, giving some little imprecision in visual results.

2.4 Visual Studio Code

Visual Studio Code is an open-source code editor by Microsoft that also runs on Linux Operating Systems. It includes support for debugging, embedded Git control and GitHub, syntax highlighting, intelligent code completion, snippets, and code refactoring¹⁶.

Since it’s customizable we could add C++ and CUDA editor extensions, furthermore an SFTP extension allowed us to quickly upload/download files from remote machines.

2.5 Tests, Result gathering, Plots

Some other additional tools have been used in this project. In particular we needed:

- to serially run executions of our applications, possibly varying input dataset on interest values;
- to group all results on text files;

¹⁵We’ll see in detail the Overlapping topic and how it was managed in Chapter 3.

¹⁶Documentation: <https://docs.microsoft.com/it-it/dotnet/core/tutorials/with-visual-studio-code>

Website: <https://code.visualstudio.com/>

- to implement a script that, from results on text files, computed average time measures, speedups, other interest metrics and generate plots on important values.

2.5.1 Bash scripts

Bash (*Bourne-Again SHell*) is the shell, or command language interpreter, for the GNU operating system; the latter provides other shells, but Bash is the default one¹⁷.

Bash scripts were needed to implement tests, that run more executions of a certain CUDA application, also varying input dataset. We programmed bash scripts (`.sh`) tests to contain several command as compile a certain CUDA application, run many times the related executable, profile it with `nvprof` and so on.

Below we give a brief example on how we used bash scripts to implement tests:

Listing 2.3: Tests on bash scripts example

```
#!/bin/bash

...
let GPU=3
let B=1024
let nTests=4

Mitters=(10000 400000 800000)
Ns=(57344 114688 229376 458752 917504 1835008)

#parameters: deviceID, BLOCK, N_elements, M_iterations, cuStr[bool], strNum
make cleandpflow
echo -e "${BLUE}compiling stream...${NC}"
make streamlow

echo ***STREAM_LOW_NOSTREAM_test***
for N in "${Ns[@]}"
do
    for m in "${Mitters[@]}"
    do
        echo sizeN = $N elements, kerM = $m , CUstreams = 0
        echo -n execIndex =
```

¹⁷<https://www.gnu.org/software/bash/manual/>

```

for((j=0; j<nTests; j+=1));
do
    echo -n "$j , "
    ./bin/streamlow.out $GPU $B $N $m 0 0 >> ./results/
        dev_cos_dp_stl.txt
done
echo -ne "$nTests \n"
nvprof --log-file ./profiling/dev_str_low$N-$m-$B-0.txt ./bin/
    streamlow.out $GPU $B $N $m 0 0 >> ./results/dev_cos_dp_stl.txt
done
done

```

Here, for example, the code is setting up all input data set, it compiles the code with the `streamlow` Makefile rule into an executable, then it loops over all data sets to run the executable in each configuration.

Here we can see that each setting is repeated more times (to allow outliers elimination) and one of those executions is performed via the `nvprof` profiler, in any case all executions outputs are redirected to `.txt` files for later consultation.

2.5.2 Python scripts

As Python ¹⁸ has dynamic typing, together with its interpreted nature, it's an ideal language for scripting and rapid application development.

In the case of this work was most useful to quickly implement a result filter: given text files with time measures, we computed averages and some speedups.

Furthermore we implemented a script to generate some plots on averages and speedups, exploiting the library `matplotlib`¹⁹.

Below we show a part of the code to compute speedups and generate some plots:

¹⁸The Python version used to compile, on local host, our scripts is: Python 2.7.6.

Documentation: <https://docs.python.org/2/>

¹⁹<https://matplotlib.org/>

Listing 2.4: Portion of speedup and plots Python script

```
def main():
    res_path = "./output/"+sys.argv[1]+"/"
    for file in os.listdir(res_path):
        if file.endswith("avgs.csv"):
            f=os.path.join(res_path, file)
            resType=file[0:3]
            if resType=="cos":
                dataPar,zeroStream,threeStream,coresStream =
                    getDividedData(dpCosTest,strCosTest,f)
            elif resType=="mat":
                ...
            ...
            zeroThree,zeroCore,coreDp = getSpeedup(zeroStream,threeStream,
                coresStream,dataPar)
            ##write speedup to CSV##
            csvPath=res_path+resType+"_sp.csv"
            with open(csvPath, "wb") as fcsv:
                writer = csv.writer(fcsv)
                printSpToCSV("Tseq/T3",zeroThree,resType,writer)
                printSpToCSV("Tseq/Tsm",zeroCore,resType,writer)
                printSpToCSV("Tsm/Tdata",coreDp,resType,writer)

            #####
            # Plots #
            #####
            # Completion Time
            launchPlots(resType,zeroStream,"Zero Streams")
            launchPlots(resType,threeStream,"Three Streams")
            launchPlots(resType,coresStream,"#SM Streams")
            # Speedup
            if sys.argv[1]=="M40":
                streamNums=[1,3,24]
            elif sys.argv[1]=="P100":
                streamNums=[1,3,56]

            if resType=="cos":
                plotSpeedup(streamNums, zeroStream, zeroThree,
                    zeroCore, resType, cosN, cosM)
            elif resType=="mat":
                plotSpeedup(streamNums, zeroStream, zeroThree,
                    zeroCore, resType, matNum, matSize)
            ....
```



```
#####
# Speedup plot function #
#####
def plotSpeedup(streamNums, zeroStream, zeroThree, zeroCore, resType, param1, param2):
    title="Speed Up"
    title=title.upper()
    plt.figure(figId)
    plt.plot([0]+streamNums, [0]+streamNums, marker='.', label='Ideal speed up',
             linestyle='--')
    .....

    if resType=="cos":
        indexes=[0, param2-1, len(zeroCore)-param2, len(zeroCore)-1]
        for i in indexes:
            tmp=[1.0, zeroThree[i][2], zeroCore[i][2]]
            lblLine=str(zeroThree[i][0]) + " - " + str(zeroCore[i][1])
            plt.plot(streamNums, tmp, '-', label = lblLine, marker=markers
                    [j], linestyle='-', linewidth=lineW, alpha=alphaVal)

            lineW+=0.5
            alphaVal-=0.2
            j+=1
        .....

    plt.xticks(streamNums)
    plt.yticks(streamNums)
    plt.legend(loc="upper left")
    plt.grid(axis='both')
    plt.ylabel("Speedup")
    plt.xlabel("#CUDA Streams")
    plt.show()
    figId+=1
```

The code above starts in function `main()`, it initially reads from a `.csv` file the average time measures for all different executions, given by one of the three applications we tested.

Then in the main we split times of all execution types (e.g. zero, three or number of SM amount of CUDA streams used). At this point code has what it needs to get speedups²⁰

²⁰We'll see in Chapter 5 all types of speedup we compute, how and why.

in `getSpeedup`, this function computes speedup (from input parameters) then it saves results in a `.csv` file and output them to the main function too.

Finally the main calls the functions that provide completion time and speedup plots, respectively called `launchPlots` and `plotSpeedup`. The latter plots all different speedups, given by the different input data sets of executions, compared with the line of ideal speedup.

Note that the mean values of time measures (used as input of the speedup/plot script above) are in turn generated by another python script.

CHAPTER 3

Project Logic

In this work we started considering the features and problems recognizable as a Farm parallel pattern. Then the study moved to consider how a GPU works, its main architectural characteristics and facing its data parallel nature. Next we had to think how to "merge" two such different behaviors, in order to reach reasonable performances¹, i.e. almost competitive with a classic data parallel problem. Finally, once the main idea behind the development was clear, we had to make some tunings. All of these steps will be shown in detail in next sections.

3.1 Stream Parallelism: Farm pattern

Stream parallel patterns describe problems exploiting parallelism among computations relative to different independent data items, appearing on the program input stream at different times. Each independent computation ends with the delivery of one single

¹About expected performances see Section 3.3 and Section 3.4 for more clarifications.

item on the program output stream.

We focused on **Farm parallel pattern**, modeling embarrassingly parallel stream parallelism.

The only functional parameter of a farm is the function f needed to compute the single task[7]. Given a stream of input tasks

$$x_m, \dots, x_1$$

the farm with function f computes the output stream as

$$f(x_m), \dots, f(x_1)$$

Its parallel semantics ensures it will process the single task in a time close to the time needed to compute f sequentially. The time between the delivery of two different task results, instead, can be made close to the time spent to compute f sequentially divided by the number of parallel agents used to execute the farm, i.e. its parallelism degree[7, 19].

The correspondent task farm skeleton, with a parallelism degree parameter, may therefore be defined with the high order function description:

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

whose type is

$$farm :: (\alpha \rightarrow \beta) \rightarrow \alpha \text{ stream} \rightarrow \beta \text{ stream}$$

The parallel semantics, associated to the higher order functions, states that the computation of any item appearing on the input stream may be performed in parallel, according to the number of available workers[7].

In the farm case, according to the parallel semantics a number of parallel agents, computing function f onto input data items, equal to the number of items appearing onto the input stream could be used. This is not realistic, however, for two different reasons:

1. items in the stream do not exist all at the same time, since a stream is not a vector. Items of the stream do appear at different times. Actually, when we talk of consecutive items x_i and x_{i+1} of the stream we refer to items appearing onto the stream at times t_i and t_{i+1} with $t_i < t_{i+1}$. As a consequence, it makes no sense to have a distinct parallel agent for all the items of the input stream, as at any given time only a fraction of the input stream will be available.
2. if we use an agent to compute item x_k , presumably the computation will end at some time t_k . If item x_j appears onto the input stream at a time $t_j > t_k$ this same agent can be used to compute item x_j rather than picking up a new agent.

This is why the parallelism degree of a task farm is a critical parameter: a small parallelism degree doesn't exploit all the parallelism available (thus limiting the speedup), while a large parallelism degree may lead to inefficiencies as part of the parallel agents will be probably idle most of time[7].

3.1.1 Farm performance model

It's important to consider which kind of performance indicators are useful. When dealing with performances of parallel applications, we are in general interested in two distinct kind of measures:

- those measuring the absolute (wall clock) time spent in the execution of a given (part of) parallel application. Here we can include measures such as

1. **Latency** (L). The time spent between the moment a given activity receives input data and the moment the same activity delivers the output data corresponding to the input.
 2. **Completion time** (T_c). The overall latency of an application computed on a given input data set, that is the time spent from application start to application completion.
- those measuring the throughput of the application, that is the rate achieved in the delivering of the results. Here In this case we can include measures such as
 1. **Service time** (T_s). The time intercurring between the delivery of two successive output items (or alternatively, the time between the acceptance of two consecutive input items), and
 2. **Bandwidth** (B). The inverse of the service time.

These are the basic performance measures of interest in parallel/distributed computing. Applications with a very small service time (a high bandwidth) have a good throughput but not necessarily small latencies[7].

As an example, if we are interested in completing a computation within a deadline, we will be interested in the application completion time rather than in its service time.

Each performance measure has an associated “semantics”:

- L, T_c the latency and the completion time represent “wall clock time” spent in computing a given parallel activity. We are interested in minimizing latency when we want to complete a computation as soon as possible;
- T_s, B service time represents (average) intervals of time incurring between the delivery (or acceptance) of two consecutive results (or tasks). We are interested in minimizing service time when we want to have results output as frequently as possible, but we don’t care of latency in the computation of a single result.

As bandwidth is defined as the inverse of the service time, we are interested in bandwidth maximization in the very same cases[7].

Given those measures of interest, we can describe an *approximate performance model* for Farm parallel pattern.

First we start considering the service time strictly for workers activity

$$T_s = \frac{T_w}{n_w} \quad (3.1)$$

So in task farm, the service time is given by the service time of the workers (T_w) in the farm, divided by the number of workers (n_w), as hopefully n_w results will be output by the workers every T_w units of time[7].

We can add to this model the times spent by emitter and collector, but this will depend strictly on how the Farm is implemented. For example, suppose we have an emitter/collector single process, it gathers input tasks from the input stream and schedules these tasks for execution on one of the available workers. Workers, in turn, receive tasks and once workers compute tasks, they send back to the emitter/collector process the results². This template is often referred to as *master-worker pattern*.

The three activities –task scheduling, task computation and results gathering and dispatching –happen to be executed in pipeline³. Therefore the service time of the master worker is approximated by the maximum of the service times of the three stages. However, first and third stages are executed on the same processing element (the one hosting the emitter/collector concurrent activity), so the model approximation will be

$$T_s(n) = \max\left\{ \frac{T_w}{n_w}, (T_e + T_c) \right\} \quad (3.2)$$

²The emitter/collector process –concurrently to its task scheduling activity–collects results, possibly re-order them to respect the input task order in the output sequence too.

³A performance model of the pipeline service time states that its service time is the maximum of the service times of the pipeline stages, as it is known that the slowest pipeline stage, sooner or later, will behave as the computation bottleneck.

Another example could be to have two distinct processing elements, one for the emitter and one for collector, so we could consider the performance model exactly as a three stages pipeline[7]. This will change the service time approximation in

$$T_s(n) = \max\left\{ \frac{T_w}{n_w}, T_e, T_c \right\} \quad (3.3)$$

In our case in particular, we can assume that T_e is given by host to device memory copy and T_c is device to host transfer time.

In this work we mainly wanted to observe behaviors and performances in terms of completion time. We focused on decreasing as much as possible the time needed to compute a set of Stream parallel tasks on GPU.

Furthermore, when modeling performance of parallel applications, we may also be interested in some derived performance measures. In this case we wanted to derive, from completion times, the speedup measures.

Speedup is the ratio between the best known sequential execution time (on the used target architecture) and the parallel execution time. Speedup is a function of n , the parallelism degree of the parallel execution⁴.

According to what we want to measure in Farm parallel pattern on GPU, we can give an approximation as completion time performance model. Assuming that sequential version is given by

$$T_{seq} \approx T_{H2D} + T_{ker} + T_{D2H} \quad (3.4)$$

where T_{ker} is the overall time spent in computations (kernel execution), T_{H2D} the time it takes for memory copy from host to device and T_{D2H} the time to transfer results back from device to host.

Then we can approximate our Farm completion time as follows

$$T_{comp} \approx \frac{T_{seq}}{n_w} + T_{ov} \quad (3.5)$$

Here n_w is the number of workers, T_{seq} is the time needed in sequential version given by equation 3.4, while T_{ov} is the overhead given by all those times where we can't achieve

⁴We'll see details on speedup, how it's calculated and how we introduced it in this thesis, in Chapter 5

perfect overlapping (for both transfer/kernel and kernel/kernel cases).

We recall that we don't know a priori the input/output stream length, it can be indefinitely long. But, the above formula means that, no matter how many tasks arrive from the input stream, the Farm should ideally divide the sequential time among all the workers, apart from (a hopefully negligible) overhead.

A more detailed analysis will be given in experiments, Chapter 5.

3.2 CPU-GPGPU: heterogeneous architecture

The target of this project was to exploit GP-GPUs high parallelism to lighten the CPU from computation intensive problems, in particular associated to the above explained Farm parallel pattern.

So we had to think what could happen if we wanted to manage such computations on input/output streams, in a way such that:

1. Input stream arrives from host side, being directly generated by CPU or acquired from a remote source;
2. Items are sent from host (main memory) to the GPU (global memory);
3. GPU multiprocessors perform specific computations on all items of the stream (as soon as they're available in global memory);
4. Finally, computed elements will be copied back to host side and will become the output stream.

In this list our main concern is about data transfer, i.e. step 2 and 4 (as we mentioned in Section 1.1.3). Indeed these phases introduce an overhead per se, but especially in Farm parallel pattern they can represent a not negligible bottleneck.

We should not forget that the input we're handling, is a stream of items: even if elements are available with a high throughput, they come "one by one". As we mentioned

in the previous section, we only have a part of input available in a certain point, so it wouldn't be realistic to have one worker per stream element.

Furthermore, in our case, the items in the input/output streams are data parallel tasks, i.e. each Farm worker will have to process an embarrassingly parallel function on the input task.

This means that the tasks are small collections, made up by simple elements, such as float numbers for example. Here "small collection" means that a single data parallel task has much smaller dimensions, than classical data parallel computations for GPUs⁵.

In our case we can suppose Streaming Multiprocessors to be the Farm workers so, if we don't give enough work (tasks) to each of them, we would have a resources under-utilization.

Furthermore, we surely don't want to transfer from/to GPU one simple element (e.g. a single float) at time but, in some way we have to keep as much as possible the nature of a Stream parallel pattern. That's why our Farm model transfers and computes small data parallel tasks, so it's important to observe the behavior for different task sizes too(i.e. how much small should be data parallel tasks from the input stream).

As a consequence, we had to figure out some mechanisms to:

- Hide data transfer times as much as we can, both in *Host* \rightarrow *Device* and *Device* \rightarrow *Host* direction;
- Exploit almost completely our workers resources, in other words try to make Streaming Multiprocessors as busy as possible.

3.2.1 Overlapping: Data Transfer hiding

In Section 2.3.3, we introduced CUDA Streams and we recall that they can perform asynchronous memory copies to or from the GPU concurrently with kernel execution.

⁵In Chapter 5 we'll see applications and management of Farm streams and tasks.

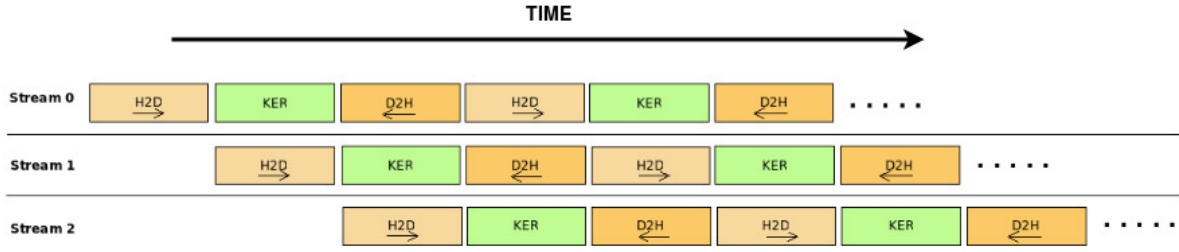


Figure 3.1: Ideal behavior for 3 CUDA Streams.

Since the machines we worked on, both have concurrency on data copy and kernel execution and both include 2 copy engines⁶, we exploited these capabilities combined with CUDA streams.

In this way we aimed to achieve a situation in which we could overlap, as much as it was possible, the time it took for the GPU to execute a kernel (possibly overlapping kernels between them too) and the time it took to transfer data back and forth.

As an example see Figure 3.1 to understand a simple case with 3 CUDA Streams. In that diagram we can see the expected behavior of three streams, but we can extend our expectations to more than three streams, without forgetting that we can have at most 2 data transfer at the same time (given that we've two copy engines).

It's important to point out that not always we can have an ideal behavior in Streams, leading to a performance improvement lower than the amount we expected.

Overlap amount depends on several factors: on the order in which the commands are issued to each stream, whether or not the device supports overlap of data transfer and kernel execution, concurrent kernel execution, and/or concurrent data transfers and finally the relative weight of data transfers time and kernel executions time[18, 10].

Given that our GPUs supported all kind of above mentioned mechanisms, among the above factors the ones that can have some impact in our case are the order in which commands are issued to each stream, and the kernel/data transfer time ratio.

⁶Chapter 2 explains copy engines utility and functioning.

To improve the potential for concurrent kernel execution, synchronization of any kind should be delayed as long as possible[18]. So, we were careful to avoid *Implicit synchronizations*⁷ and all unnecessary *Explicit synchronizations*⁸.

Another important face of overlapping, is that it requires to balance Kernels work in such a way it's sufficient to hide the time spent in data transfers, as we quoted just above. This said we can have two possibly unfair scenarios:

- Data transfers take a small amount of time, while kernels are doing lot of computations;
- Data transfers take a big amount of time, with respect to time spent in kernel execution.

The former case may arise when we have a computation-intensive application or an *irregular kernel*. By "irregular" we mean that we're facing an inefficient kernel, due to any flow control instruction (`if`, `switch`, `do`, `for`, `while`), that can significantly affect the instruction throughput by causing threads of the same warp to diverge, i.e. to follow different execution paths.

If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path[10]. So we should avoid different execution paths within the same warp.

However, sometimes having short memory copy and long kernels can be profitable. As an example we'll see, in Chapter 5, that having more long kernels makes them to overlap more likely both in between them and with data transfers.

The latter case in the above list, may happen when we move an amount of data

⁷Implicit synchronization automatically happens when certain host operations are issued in-between commands given from different streams. E.g. this happens in case of: pinned memory allocations, device memory allocations, non-asynchronous memory operations etc.[27].

⁸In CUDA there are several command to force synchronization either between host and device, or between streams.

such that it takes a huge amount time at each transfer, w.r.t. the amount it takes in calculations. So in this case the dominant overhead factor could be the data transfer.

3.2.2 Occupancy of GPU cores

Once we carried out the stream logic, we had to understand how to try to exploit almost every Streaming Multiprocessor at any given time. This means that we wanted to launch as many kernels as needed to exploit SMs at their best, sometimes this means to arrive near the full ***Occupancy*** of an SM for each kernel execution, in other situations it's better to decrease this resource exploitation⁹.

Clearly, when we start an execution, we'll have a portion of time, a sort of "*warm up*" *phase*, where we'll have first data transfers and kernels launches. So we'll have a narrowed number of running kernels. But as soon as we could have enough data transfers, and therefore enough kernels to execute, we hope to reach a workload peak on GPU.

In practice, when we just said *lot of kernels*, we meant a lot of small groups of (data parallel) items on which (data parallel) computations have to be applied. So, each of these items will be assigned to one or more (few) thread blocks.

Now we'll better explain what Occupancy means.

To ***maximize utilization*** the application should be structured in such a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the system to keep them busy most of the time[10].

The main ways to maximize utilization can be classified as follows:

1. **Application Level** At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using *asynchronous functions* calls and streams;

⁹Chapter 5 will also discuss occupancy and the role it played in our applications and experiments.

2. **Device Level** At a lower level, the application should maximize parallel execution between the multiprocessors of a device. Multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently;
3. **Multiprocessor Level** At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor. In particular, *a GPU multiprocessor relies on thread-level parallelism to maximize utilization of its functional units*[10].

From the above, we can deduce that occupancy is directly linked to the number of resident warps¹⁰.

At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the relative instructions to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called the **latency**, *and full utilization is achieved when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when latency is completely "hidden"*[28, 14].

The most common reason a warp is not ready, to execute its next instruction, is that the instruction's input operands are not available yet.

If all input operands are registers, latency is caused by register dependencies, i.e. some of the input operands are written by some previous instruction(s) whose execution has not completed yet.

So, in this case, the latency is equal to the execution time of the previous instruction and the warp schedulers must schedule instructions for different warps during that time[10, 14].

¹⁰We recall that warps are groups of 32 threads running in parallel on a set of instructions. Resident warps are those warps that, at a given time, are active on a certain thread block. In Chapter 2 we gave a detailed explanation of these concepts.

Another reason a warp is not ready to execute its next instruction, is that it is waiting at some *memory fence* (*Memory Fence Functions*) or synchronization point. A synchronization point can force the multiprocessor to stay idle as more and more warps wait for other warps in the same block to complete execution of instructions. So, having multiple resident blocks per multiprocessor can help reduce idling in this case, as warps from different blocks do not need to wait for each other at synchronization points.

The *number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call* (grid and block dimensions), the memory resources of the multiprocessor, and the resource requirements of the kernel[10]. Register and shared memory are others important Occupancy variables, but we didn't focused much on them as on execution configuration. This is because, in our applications, those factors had a negligible impact on eventual occupancy.

At this point, we had to reason about how and when to maximize Occupancy in our Farm parallel pattern. We shouldn't forget that after looking for full occupancy, experiments can give a prove that a lower one could be better.

First, we have to make some assumptions:

- no shared memory was used;
- we took a really poor amount of registers, given the really simple nature of our example Kernels ¹¹.

Anyway, our chosen kernels represent some *important application categories*, as we mentioned in previous chapters.

Then we mainly put our attention on kernel *execution configuration* and number of

¹¹We'll see what kind of kernels we used to test the farm parallel pattern, with some code listings in Chapter 4.

kernels launched (by different CUDA streams), in order to try to maximize the number of active warps inside each Streaming Multiprocessor.

3.2.3 Occupancy drawbacks

Occupancy is a very important factor to take into account, but it's more important to be aware that *occupancy isn't the only factor to take care of*.

In other words, not always trying to achieve maximum occupancy is the best idea, in some cases lower occupancy gives even better performances.

It is common to recommend running more threads per Streaming Multiprocessor and/or running more threads per thread block; the motivation is that this is the main way to hide *latencies*.

Indeed, common beliefs are: multithreading is the only way to hide latency on GPU; shared memory is as fast as registers[\[10\]](#). Those facts aren't always true.

Some studies demonstrated how was possible to hide arithmetic latency or to hide memory latency using fewer threads, leading to code that runs faster. The *Latency* is the time required to perform an operation, for arithmetic operations it takes ≈ 20 cycles; for memory we have $\approx 400+$ cycles instead.

This, in particular, means that we can't start a dependent operation during these times, but they can be hidden by overlapping with other (independent) operations[\[13\]](#).

```
x= a + b; // takes about 20 cycles to execute
y = a + c; // independent, can start anytime(stall)
z = x + d; // dependent, must wait for completion
```

So *latency hiding* means to do other operations when waiting for latency, this will make code run faster (not faster than the peak). For example another way to hide latency is *Instruction Level Parallelism*¹².

¹²In general ILP in a kernel is intended as assigning more instructions (possibly equal between them) to each single thread, instead of having a lot of threads executing a lower amount of instructions each. Furthermore ILP can be used to execute independent instructions between two dependent ones[\[28\]](#).

Furthermore another common belief is that occupancy is a metric of utilization, but, as we anticipate, it's only one of the contributing factors[13].

Another type of latency is memory-bounded, let's take an example:

```
--global__ void memcpy( float *dst, float *src){
    int block = blockIdx.x+ blockIdx.y* gridDim.x;
    int index = threadIdx.x+ block * blockDim.x;
    float a0 = src[index];
    dst[index] = a0;
}
```

To hide memory latency, using even fewer threads, we can do more parallel work per thread:

```
--global__ void memcpy( float*dst, float*src){
    int iblock= blockIdx.x+ blockIdx.y* gridDim.x;
    int index = threadIdx.x+ 2 * iblock* blockDim.x;
    float a0 = src[index];
    //no latency stall
    float a1 = src[index+blockDim.x];
    //stall
    dst[index] = a0;
    dst[index+blockDim.x] = a1;
}
```

Note: threads don't stall on memory access, they stall on data dependency instead.

Performances improve copying more floats per thread, instead of copying one and run more blocks and allocate shared memory to control occupancy[13].

For example some common concepts¹³ on CUDA state that:

- "In general, more warps are required if the ratio of the number of instructions with no off-chip memory operands to the number of instructions with off-chip memory operands is low";
- "For all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads" [10].

¹³From CUDA Programming Guide.

For the former there are studies that show how a reduced quantity of warps gives good performances on memory intensive kernels.

For the latter, in reality shared memory bandwidth is lower than register bandwidth, in fact we should use registers to run closer to the peak. But requiring more registers can result in having a low occupancy.

So, in many cases, this can be accomplished by computing multiple outputs per thread (see above example on multiple floats copy)[13, 14].

3.3 Overall Logic

We have an input stream of items, in particular they are small data parallel tasks, we don't know how much they are and their arrival frequency. The logic of this work can be summarized in the following steps:

1. In the beginning, as items arrive, we start to spread them among different CUDA streams, according to a Round-Robin policy;
2. On a certain stream, say `streams[k]`, we send out a task (e.g. the k -th item that has arrived from the input stream) to the device (GPU Global memory);
3. Immediately after the data transfer call, we launch the kernel execution, with a certain *execution configuration*¹⁴. The kernel call will be placed in `streams[k]` as well;
4. Once the kernel ends its computations, we copy back to host, on `streams[k]`, the result data as output item;
5. We'll send each result element onto the output stream.

¹⁴We recall that this is given by grid and block sizes that we set for a certain kernel call with the `<<< grid, block >>>` syntax.

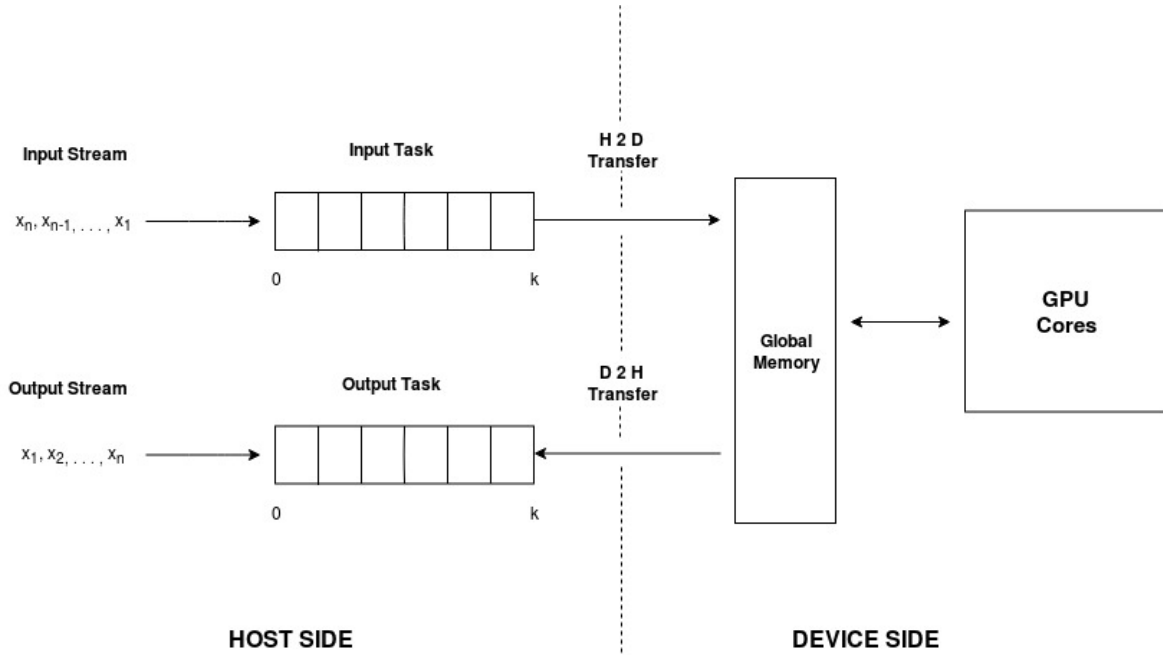


Figure 3.2: In this picture we show the schema of Farm on GPU, here we have only one task to/from the GPU.

This behavior is illustrated graphically in Figure 3.2. Here we can see our input stream, from that at a given time we get a certain item, i.e. a task. In the diagram we simplified the concept of (data parallel) tasks, from the input stream, as it was a buffer of size k ; clearly this is only a graphical simplification. Then, we transfer the various items to Global memory of GPU.

For reasons we showed in the previous sections, it would be unfeasible to work on single simple elements (e.g. float numbers) but, at the same time, we should maintain the pattern as close as possible to Stream parallel. That's why we're working on small data parallel tasks¹⁵ of k items, where k is tested for several values but to remain a relatively small number of simple elements, possibly related to execution configuration on kernel, in particular to block size.

¹⁵We represented tasks as arrays, but they can be either small matrices or tiny images, as we'll see in Chapter 4.

Once we start to have several items available on GPU memory, they will be spread all over the SMs that will have enough available resources. Each task will be splitted in warps, so each active thread in a warp will compute one instruction per time, in parallel with other threads in the warp. The instruction that will be executed are essentially those specified inside kernel code.

This means that we're executing in parallel the same operations over all the content in a task; in fact, as we mentioned before, each task is a small data parallel collection of items, upon which we're performing data parallel computations.

Since Figure 3.2 is a simplification, it may seems we're sending only one item per time to/from the GPU, this would correspond almost to a farm with one worker, processing one item per time. And this isn't completely what we wanted.

So, here's where CUDA Streams ¹⁶ are needed and we used them relying on the following ideas:

1. We have as many streams as Streaming Multiprocessors ¹⁷ and, at any given time, each of them hopefully issues a data transfer or a kernel executions;
2. We should come to the point where each stream has issued at least one kernel launch, ideally we expect that each kernel execution is taken over by a certain multiprocessor. So, at a given time, we want to reach a work peak, where almost all SMs are busy;
3. Obviously each kernel execution configuration should be well tuned, in order to

¹⁶Don't confuse input/output stream in Farm parallel pattern with CUDA Streams.

These are two completely different notions: the first is the parallel pattern input/output data type, the last are a CUDA feature (shown in Section 2.3.3).

¹⁷Again CUDA Streams are a different concepts with respect to Streaming Multiprocessors. The first are a set of commands, the last are physical processing units inside the GPU.

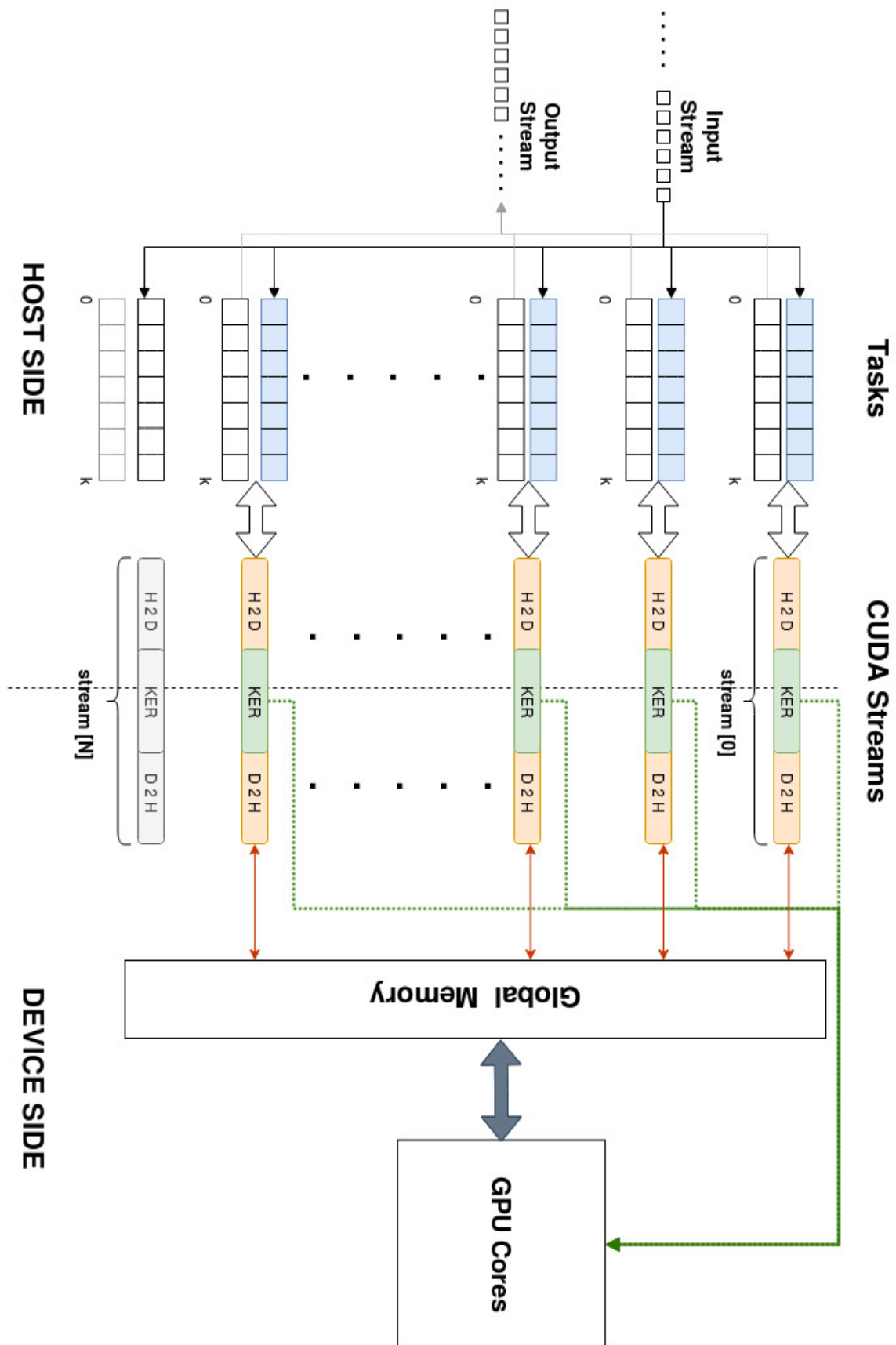


Figure 3.3: Here we have an overall and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

take advantage of the maximum of resources in a multiprocessor (according to a certain kernel nature).

All of those parameters have been initially established taking into account of the NVIDIA GPUs' nature, then experimental proves have been performed¹⁸. Measures and other estimation lead us to consider specific values for those variable parameters. So from the above facts is clear that we're exploiting each SM as a Farm parallel worker, furthermore, in such a way that all of those workers are as busy as possible. Note that our n_w **SMs-workers** apply a **kernel-function** f to all (data parallel) **tasks**.

Bringing all pieces together we can summarize all project logic in Figure 3.3.

The schema may be further detailed as follows:

- We have N CUDA streams, where N ($= n_w$) is the number of Streaming Multiprocessors on the target machine;
- As input stream items arrive, in a Round-Robin way, we spread them all over the CUDA streams as follows:
 1. As soon as we get the i^{th} task, it's asynchronously sent on **stream** [i] to the GPU, with the command


```
cudaMemcpyAsync( devTask, hostTask, bytes, cudaMemcpyHostToDevice,
                  stream [i]);
```
 2. Immediately after we put kernel call, again on the **stream** [i], to schedule the desired computations on that input item;
 3. Then, asynchronously again, we bring back results to host side, using the instruction


```
cudaMemcpyAsync( devTask, hostTask, bytes, cudaMemcpyHostToDevice,
                  stream[i]).
```

¹⁸We'll see in next section more informations about Tunings.

Hopefully this approach should make each Streaming Multiprocessor (or at least a part) busy. Initially only few cores will be really busy, but as soon as CUDA streams get full, the pressure¹⁹ on the GPU should increase, so we expected that workload should be enough to almost fill all of Streaming Multiprocessors.

In particular, for us this translates in trying to have, at any given time, the maximum number possible of active threads, having to execute instructions, inside each SM.

Figure 3.5 gives a closer look to what we just said.

From that schema, looking at violet numbered labels, we can see the order in which we issue commands in a generic stream, and this will be the order in which commands will be issued to device side too, for that stream.

The behavior of overlapping between different streams, isn't predictable[10]. Anyway we should take advantage of the fact that, considering two different CUDA streams, we can overlap data transfer and/or kernel execution in a `stream [i]` with the ones in a `stream [j]` (for some $j \in [0, N - 1]$, for $N = \#SMs$). Obviously, when the number of CUDA streams is greater than 3, we can have only 2 data transfer operations executing at the same time (by two distinct

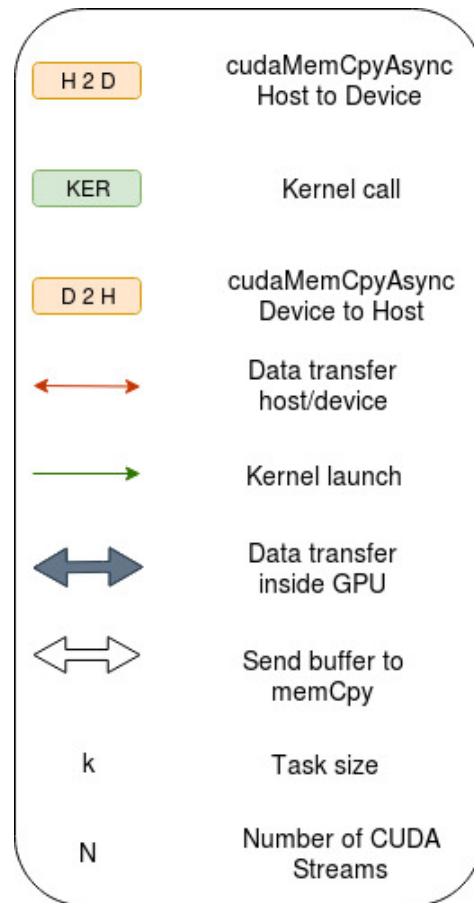


Figure 3.4: Legenda about Figure 3.3.

¹⁹In other words the amount of tasks assigned to each CUDA stream and so to each SM.

streams)²⁰.

Note that in the Figure 3.5 we represented a single kernel execution as fully occupying an entire SM; in reality not always we'll have this behavior, sometimes it's even convenient to not fully occupy an entire SM with a single kernel launch²¹.

Essentially if all of our reasoning and theories are right, we would expect that we can have an improvement, on completion time, roughly in the order of SMs number with respect to the *serial approach*.

This similarly means that if, for example, we have 3 CUDA Stream we would expect to take an advantage on approximatively 3 SMs (at peak work flow), so this should give us an improvement, in completion time, of at most 3 times compared to serial approach. The case of ***serial approach***, instead, processes input stream items without any type of overlapping, it is equivalent to:

- send an input task to device and wait on host for data transfer completion;
- call the kernel;
- host calls the copy back, from device to host, and keeps waiting until the end of data transfer;
- in the meanwhile kernel is running on GPU;
- finally, only when all computations have ended up, results are transferred back to the host, that at this point finishes to wait for output and, so, it can continue with next task.

²⁰As mentioned in Chapter 2, for concurrent memory copy between host and device, we have 2 copy engines.

²¹We'll see how we practically tried out Streaming Multiprocessors *occupancy* in Chapter 5.

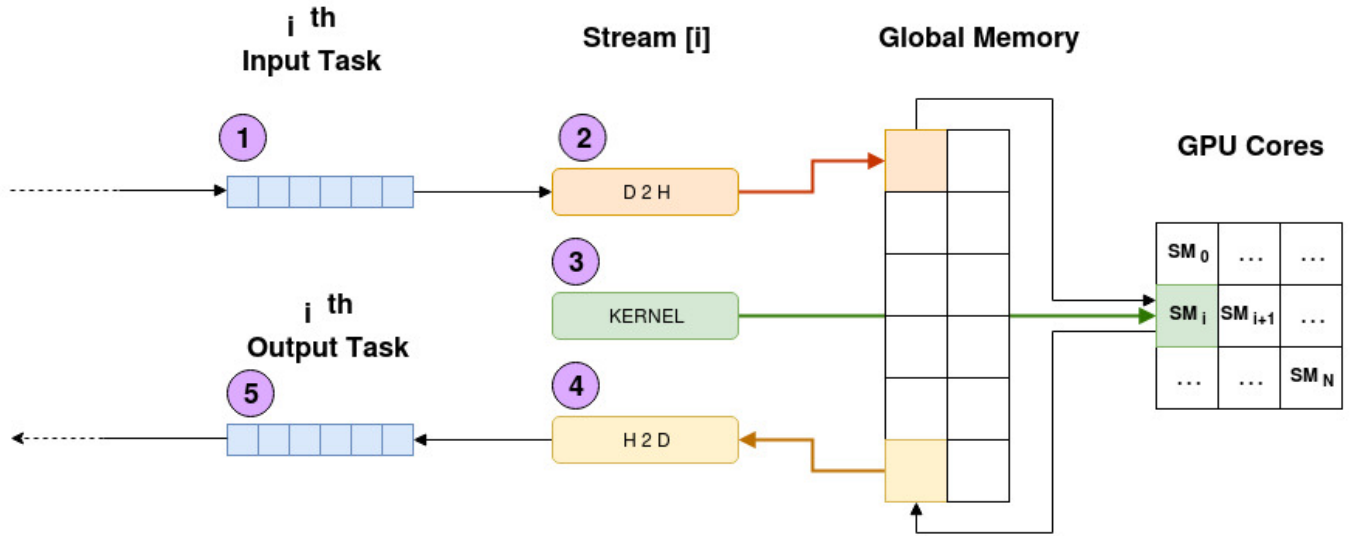


Figure 3.5: In this picture we can see what exactly happens in a certain CUDA Stream. Light violet numbered labels show the order in which commands are issued by host to the GPU by a generic stream.

3.4 Tunings

We showed a lot of peculiar behavior and architectural characteristics, as they were taken into account for different implementations, tests datasets and results analysis. It's clear that, once we decided how to organize our Farm parallel pattern for the GPU, we had to perform experiments and empirical evaluations. This is because:

- It was important to think about a general logic, that wasn't architecture-dependent²²;
- To validate our idea we had to make a lot of experiments, time measures, examples and counterexamples too;
- Clearly experiments required to get a little deeper about NVIDIA GPU's architecture, considering the good practices, apart from the considered application;

²²At least we can say that the overall view, showed in Fig. 3.3, can be plausible with almost all NVIDIA's architecture having more than one copy engine and allowing concurrent kernel execution.

- Finally, we had to consider some feature totally application-bounded to launch tests and obtain results of interest.

So after the logical phase, we went through a *tuning phase*, that first had to face general NVIDIA GPUs behavior and structure²³. Some of the important best practices we introduced in tuning phase were:

- The effect of execution configuration on performance for a given kernel call generally depends on the kernel code, so experimentation is recommended[18] and in fact we followed that approach;
- The number of threads per block should be chosen as a multiple of the warp size (generally equal to 32 threads) to avoid wasting computing resources with under-populated warps²⁴[18];
- We exploited *Occupancy Calculator*²⁵ both in spreadsheet and API functions²⁶ formats[10].

Given those initial guidelines, it's important to highlight what are main variable parameters, in Figure 3.3, on which the tuning was made:

- The number of CUDA streams;
- The number of threads per block (block size);
- The number of blocks (grid size).

²³In Chapter 5 we'll mainly see tunings based on the GPUs used to run tests –**P100** and **M40**.

²⁴For example, if we have a block size of 50 threads, the GPU will still issue commands to 64 threads, so we would waste 14 of them idling.

²⁵Those tools are included in CUDA Toolkit, they provide mechanisms to choose thread block size based on kernel behavior, register and shared memory requirements.

²⁶These are special functions to be called inside the code, we'll see in Chapter 4 a code example on how and where they are used.

The first parameter changes mainly as the target machine changes²⁷.

The second parameter, in our case, generally depends both on input tasks dimensions and CUDA limits on thread blocks dimensions. But the threads per block choice may be also influenced by the kernel nature and a tuning according to performance measures. The third is generally roughly determined dividing the size of the input data by the block size. However Farm is a particular case and sometimes may be useful to determine the grid size empirically.

The Chapter 5 shows all main parameters tested and their respective performances.

3.4.1 Tuning on block and grid dimensions

As we mentioned above, the variation on *thread block size* and *grid size*, can affect heavily performances, especially in an extreme scenario as the case studies of this thesis²⁸. There's a tight correlation in between *Occupancy*, *kernel execution configuration* and kernel code nature. First note that a certain block, whatever its dimension is, will be run on a single Streaming Multiprocessor and once it's assigned it will never be moved. When resources are allocated for a thread block in a SM, it will become an *active block*. In a SM we can have multiple blocks running independently, each of which grabbing its portion of resources, i.e. we can have multiple active blocks on a SM until they don't hit the maximum allowed[28, 10]. So, inside a certain SM, some particular cases may happen:

1. Maximum block dimension (meaning a lot of threads per block), can lead to have

²⁷We'll see in Chapter 5 that the experiments are performed on zero and three CUDA streams. Furthermore we test the case in which we have as many CUDA streams as SMs in the target GPU, so this value will change between P100 and M40 too.

²⁸Here we report some useful discussions on occupancy, block and grid dimensions from Stack Overflow:
<https://stackoverflow.com/questions/9985912/>
<https://stackoverflow.com/questions/5643178/>
<https://stackoverflow.com/questions/54715373/>

a smaller number of blocks²⁹;

2. Small block dimension, can give a higher number of blocks running on the same SM³⁰.

In the first scenario, we can have cases of good performances. For example, if few blocks have lot of work to do, it's more likely that they will monopolize resources of the SM in which they're active, making all other waiting blocks, scheduled in the same SM, idle for too long (for example in our GPUs it may happens for *blocksize* = 1024). In the second scenario we can have cases of really poor performances due to low resources exploitation, but for some kinds of kernel we may have a gain, especially in cases as the above mentioned monopolization. So smaller blocks size, means more active blocks on a SM, at any given time. This may allow to have smaller and, so, faster blocks that allow new ones to start executing. Smaller block sizes eventually means to have more active blocks per SM[18].

In general, there is a performance *sweet spot* for middle values on thread block size (for example usually identified in *blocksize* = 512).

So our tests had to face the above explained behaviors too, without forgetting the nature of the various implemented kernels.

²⁹Even only one, as it happens for one of the applications we tested. Anyway, from best practices guide and from profiling this should be a inefficient configuration, so we should be very careful on performances from these situations.

³⁰We recall that GPUs have also a limit on the number of active blocks per SM.

CHAPTER 4

Implementation

Given the logical schema of the previous chapter, then we had to build different types of code to test its correctness and efficiency.

In particular, we wanted to distinguish some kernels of interest and adapt them to the Farm parallel pattern on GPU. So, in this project we've implemented the following applications:

- Simple-computational kernel;
- Matrix multiplication;
- Image processing.

4.1 Kernels

As we anticipated in the previous section, our kernels mainly doesn't use shared memory. Each kernel clearly is designed such that each thread executes a different element from the input data parallel task, i.e. one of the input stream's items.

4.1.1 Simple-computational kernel

This is a very synthetic kernel, where, given as inputs a data parallel task and a number M , it computes the cosine applied M times, for each element in the task. In this case small data parallel tasks are implemented as an array, of size N , formed by floating point numbers. So, the output will be again a floats array (given by cosines of the inputs).

This particular kernel has been chosen because it allows:

- to vary the load of computation T_{comp} , changing the amount of iterations (cosines) to compute inside the kernel (M);
- to vary the communication weight T_{comm} , changing the size of the task (N).

The first parameter models the arithmetic intensity, while the second the amount of memory operations. The second parameter, in this case, may also include all those mechanisms that perform host/device transfers, but also the time spent in transfer between global memory and registers inside the GPU. However, in Farm performance model, the main cost of time we focus on is host/device memory copy, neglecting memory operations internal to GPU.

Note that, since here we're working on one-dimensional data structures for this application, we'll use one-dimensional thread blocks for convenience.

Listing 4.1: Implementation for Simple-Computation Kernel

```
__global__ void cosKernel(int M, int N, float *x_d){
    int idx = offset+blockIdx.x*blockDim.x + threadIdx.x;
    if(idx<N){
        for(int j=0; j<M; ++j)
            x_d[idx]=cosf(x_d[idx]);
    }
    return ;
}
```

This is almost a regular kernel, with no branching and an equal workload for each thread that executes that code.

It is a very useful kernel, because changing a single parameter we could then test situations either of low or high iterations amount, i.e. different workloads.

Moreover this is a computation-bound application¹. In particular it models all those kernels that perform a consistent amount of work, on a restrained quantity of data (low amount of memory operations) and that don't diverge during their execution.

4.1.2 Matrix multiplication

Here we have the most classical and simple version of matrix multiplication in CUDA. We're always working on input/output streams of small data parallel tasks. However, there's an important difference from the application above, i.e. now we've matrices as tasks, both in input and output. So, for convenience, we'll use two-dimensional blocks and grid.

Each couple of threads perform the computation of a single element in result matrix. For example, assume we have `thread[ROW]` and `thread[COL]`, then they will perform `sum += A[ROW, i] * B[i, COL];`, where `i = 0, ..., N` (having `N` as square matrices order) and `sum` will be `C[ROW, COL]`, i.e. one of the items in result matrix[10, 33].

Listing 4.2: Implementation for Matrix Multiplication Kernel, both non-square and square

```

/**** MATMUL ****/
__global__ void matMulKernel(float* A, float* B, float* C, int m, int k, int n) {
    int ROW = blockIdx.x*blockDim.x+threadIdx.x;
    int COL = blockIdx.y*blockDim.y+threadIdx.y;

    if (ROW<m && COL<n) {
        float tmpSum = 0.0f;

        for (int i = 0; i < k; ++i) {
            tmpSum += A[(ROW*k)+i] * B[(i*n)+COL];
        }
        C[(ROW*n)+COL] = tmpSum;
    }
}
```

¹Chapter 5 explains the differences between computation-bound and memory-bound.

```

        return ;
    }

    /**** SQUARE MATMUL ****/
    __global__ void squareMatMulKernel(float* A, float* B, float* C, int N) {

        int COL = blockIdx.x*blockDim.x+threadIdx.x;
        int ROW = blockIdx.y*blockDim.y+threadIdx.y;

        if (ROW<N && COL<N) {
            float tmpSum=0.0f;

            for (int i = 0; i < N; ++i) {
                tmpSum += A[(ROW*N)+i] * B[(i*N)+COL];
            }
            C[(ROW*N)+COL] = tmpSum;
        }
        return ;
    }
}

```

Matrix multiplication is one of the most widespread applications in GPU computing, this is the basis of other applications too.

It is well known that this kind of very trivial matrix multiplication implementation is quite inefficient. In fact, each for loop iteration will have to perform a multiplication and a sum but, at the same time, the GPU will have to access global memory two times².

This means we have a low arithmetic intensity w.r.t. memory accesses, so threads generally will not properly hide memory access latency. That's why in general other optimized algorithms are used, the most known of them is the one decomposing matrices in tiles that will fit in *shared memory* and using a *coalesced* memory access pattern³[23, 18, 10, 33].

²We'll have two load from memory for A[i, k] and B[k, j] and, after the for loop, a store for C[i, j].

³Grouping of threads into warps is also relevant to global memory accesses. The device coalesces global memory loads/stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth.

Even if this implementation is poor in performances, to our testing purposes it is very useful, because it represent a counterexample w.r.t. the implementation presented in the previous section. In fact it's almost the opposite from Simple-computational kernel, as it's a kernel having a low arithmetic intensity with respect to the number of elements involved in computations.

As we mentioned before, just looking at each single iteration we've approximately many memory accesses as arithmetic operations. That's why this type of kernel is considered memory-bound.

4.1.3 Blur Box filter

The last type of application implemented in this project is an image processing kernel computing a blur filter.

Again we're working on input/output stream of data parallel tasks that, in this case, are given by small images.

Here input and output pictures are represented as char buffers, items are **RGB** values in $[0, 255]$ range that represent pixels as color tuples.

For each pixel, in the input image, we take the average of each of the pixels in neighborhood (inside the limits of filter size) and we write the average value to the correspondent pixel of the output image. This filter is known as a Box blur⁴[25].

Listing 4.3: Implementation for Image processing Kernel (Blur Box Algorithm)

```
/* ** BLURBOX ** */
__global__ void blurBoxFilterKer(unsigned char* input_image, unsigned char*
    output_image, int width, int height) {

    const unsigned int offset = blockIdx.x*blockDim.x+threadIdx.x;
    int dim = width*height*3;
    if(offset<dim){
```

⁴Another blur filter is the *Gaussian blur*, generally preferred as to be more accurate. In fact Box blurs are frequently used to approximate a Gaussian blur. By the central limit theorem, repeated application of a box blur will approximate a Gaussian blur.

```

int x = offset % width;
int y = (offset-x)/width;
int fsize = 5; // Filter size
if(offset < width*height) {
    float output_red = 0;
    float output_green = 0;
    float output_blue = 0;
    int hits = 0;
    for(int ox = -fsize; ox < fsize+1; ++ox) {
        for(int oy = -fsize; oy < fsize+1; ++oy) {
            if((x+ox) > -1 && (x+ox) < width && (y+oy) > -1
                && (y+oy) < height) {
                const int currentoffset = (offset+ox+oy*width)*3;
                output_red += input_image[currentoffset];
                output_green += input_image[currentoffset+1];
                output_blue += input_image[currentoffset+2];
                hits++;
            }
        }
    }
    output_image[offset*3] = output_red/hits;
    output_image[offset*3+1] = output_green/hits;
    output_image[offset*3+2] = output_blue/hits;
}
return;
}

```

This type of kernel is another counterexample with respect to the Simple-computational kernel.

In fact this application models all those kernels that have low arithmetic intensity, opposed to a consistent amount of memory operations and, furthermore, having a lot of branching instructions. So in this case we're considering not only a memory-bound kernel, but also an example of kernel with divergent flows.

4.2 Parallel Patterns implementation on GPU

What really makes the difference in the implementation is how we send data to the GPU and kernel execution configurations. This is what really determines a behavior associated to either a Stream Parallel pattern or a Data Parallel pattern.

4.2.1 Stream Parallel on GPU

The code translates the diagram in Figure 3.3. So, we start by explaining the setting phase:

- The block dimension is provided as command line parameter;
- Then grid dimensions are derived;
- Here we can have two different settings
 - The dimension of the task is equal to the block maximum size, given the capabilities in our machines means having a block size with ≤ 1024 threads. In this case the grid will have size one (half of a SM should be exploited);
 - The task has size equal to the maximum allowed number of active threads in a SM, for both our machines is 2048. Here the grid will be adjusted to cover the task dimension w.r.t. the maximum block dimension (1024 for both machines), i.e. $GRID = \text{maxThreads}/BLOCK$ and, in this case, will be equal to two. In this situation we expect that only one SM is fully occupied (or two are occupied by half).
 - The task size is greater than the maximum allowed number of active threads in a SM. So, having the maximum block dimension (1024) and obtaining grid as in the point above, we'll have more than two blocks (having 1024 threads each) and so they'll occupy (eventually completely) more than one SM.

Below we present an example of execution configuration, for Simple-computational kernel.

Listing 4.4: Simple-computational Kernel Launch configuration, i.e. Grid and Block dimensions setting

```
//Task size = maximum block size
    GRID = 1;
    chunkSize = BLOCK*GRID;
//Task size > maximum block size
    GRID = maxThreads/BLOCK;
    chunkSize = BLOCK*GRID;
```

Now we outline the core of the implementation. The number of CUDA streams to spawn in our applications, as we previously told, can be equal to the number of Streaming multiprocessors in the target machine⁵. So, we start by distinguish two different cases:

- Number of CUDA streams equal to zero, this means we won't use non-default streams, but default-stream only
 1. We allocate space on device for the single task, with a `cudaMalloc`;
 2. As input stream items arrive, we send them to the device with a simple `cudaMemcpy`⁶ (because here we're not using CUDA streams);
 3. We launch the Kernel, that will execute as soon as input data is fully copied;
 4. We call another `cudaMemcpy` to bring back results to host.

Below we present an example of code showing the steps described above, clearly this code will be executed for each of the tasks arriving from the farm input stream.

⁵It's one of our tested use case of CUDA streams. It's the most particular one, since in literature generally such a big number of non-default streams isn't used.

⁶Note that `cudaMemcpy` is a blocking operation w.r.t. the host, this means that successive CUDA calls from the host will be issued after data is fully copied to/from device.

Listing 4.5: Data transfer host/device and kernel call, synchronous version

```
void cosKer(int m, int chunk, float *x, float *cosx, float *x_d)
{
    int xBytes = chunk*sizeof(float);

    gpuErrchk( cudaMemcpy(x_d, x, xBytes, cudaMemcpyHostToDevice) );

    cosKernel<<<GRID, BLOCK>>>(m, chunk, x_d);
    #ifndef MEASURES
        gpuErrchk( cudaPeekAtLastError() );
    #endif

    gpuErrchk( cudaMemcpy( cosx, x_d, xBytes, cudaMemcpyDeviceToHost) );
}
```

- Number of CUDA streams greater than zero, this means we will use several non-default streams, so steps are analogous to the previous ones, except for some specific mechanism used, such as:
 1. We allocate on host the space for the tasks, with a `cudaMallocHost` (in order to use CUDA streams and possibly achieve overlapping, host have to allocate memory as *Pinned*);
 2. We allocate on device the space for the tasks, with a `cudaMalloc`;
 3. In a Round-Robin fashion, we send tasks in an asynchronous way, using `cudaMemcpyAsync`⁷;
 4. We launch the Kernel in the same CUDA stream we issued the copy for input data;
 5. We call another `cudaMemcpyAsync` to bring back results to host, on the same stream as before.

Note that in first two steps, in reality, we allocate space not for a single task, but for as many tasks as will be the number of CUDA streams. This allows us to

⁷Note that `cudaMemcpyAsync` is non-blocking for the host, in parameters we'll have to specify which stream will issue the copy.

avoid some explicit synchronizations and to have space reserved for a certain task for each CUDA stream.

Listing 4.6: Data transfer host/device and kernel call, CUDA Streams version

```
void cosKerStream(int m, int chunk, float *x, float *cosx, float *x_d,
                  cudaStream_t strm, int strBytes)
{
    gpuErrchk( cudaMemcpyAsync(x_d, x, strBytes, cudaMemcpyHostToDevice, strm) );

    cosKernel<<<GRID, BLOCK, 0, strm>>>(m, chunk, x_d);

    #ifndef MEASURES
        gpuErrchk( cudaPeekAtLastError() );
        gpuErrchk( cudaDeviceSynchronize() );
    #endif
    gpuErrchk( cudaMemcpyAsync(cosx, x_d, strBytes, cudaMemcpyDeviceToHost, strm) );
}
```

The reason why we implemented these two versions, for Farm Parallel Pattern on GPU, is that we want to show the gain obtained with CUDA Stream.

In particular we want to compare:

- The implementation without CUDA streams, that represents the serial version for each of our three applications;
- The implementation using CUDA streams, that correspond to the parallel version for each of our applications.

Given that we're working on a Stream parallel problem, suppose we have an input stream of items, that in our model are "small" data parallel tasks.

So, in the first case from the above list, we'll have a serial computation of the input tasks. While, in the second case, we consider SMs as *Farm workers*, that executes different tasks in parallel.

Therefore, we want to show that with CUDA Streams and some other adjustments⁸, a

⁸Chapter 5 shows how we set kernel execution configurations and other relevant assumptions.

Stream Parallel Pattern would have performances near to Data Parallel ones (comparing them for the same application and the same total amount of computed data).

It may be also interesting to see how the Round Robin scheduler sends tasks to the GPU, we present the implementation for the function `cosKerStream`, using CUDA streams. We present a pseudo-code version that shows only the main features:

Listing 4.7: Host side pseudo-code: input stream & kernel call function(scheduler).

```
const int streamBytes = chunkSize*sizeof(float) ;
int strSize = nStreams*chunkSize;
//host pinned mem
gpuErrchk( cudaMallocHost((void **)&x, strSize*sizeof(float)) );
gpuErrchk( cudaMallocHost((void **)&cosx, strSize*sizeof(float)) ); //pinned cosx
//device memory
gpuErrchk( cudaMalloc((void **)&x_d, strSize*sizeof(float)) );
//stream array and events creation
cudaStream_t streams[nStreams];
streamCreate(streams, nStreams);
createAndStartEvent(&startEvent, &stopEvent);

int k=0;
while (InputStream) {
    if (buffer x[i: i+chunkSize] is full)
    {
        int i = k%nStreams;
        int strOffs = i*chunkSize;

        cosKerStream( M_iterations, chunkSize, x[i: i+chunkSize], cosx[i: i+chunkSize],
            x_d[i: i+chunkSize], streams[i], streamBytes);

        send output buffer cosx[i: i+chunkSize] to output stream

        ++k;
    }
    else
    { add item to buffer x[i: i+chunkSize] }
}

msTot = endEvent(&startEvent, &stopEvent);
streamDestroy(streams, nStreams);
```

It's interesting to highlight the use of *CUDA Events*, they were useful to measure the completion time of memory copies and kernel executions. So they allowed us to make device side measures, that are the main concern in this project⁹.

As we can see we presented most important code listings relative to the Simple-computational kernel. The structure and the implementation to execute Matrix multiplication and Blur Box are similar to Simple-computational kernel, since they are really different applications but each of them is, however, modeling a Farm parallel pattern with small data parallel tasks.

4.2.2 Data Parallel un GPU

To prove that our Farm Pattern had acceptable performances, it was useful to compare it with its respective Data Parallel version.

Clearly, to have control on time probes and to compare such two different models, we set a maximum length on input stream for Stream parallel version. In the reality we know that we cannot have such informations on input/output streams¹⁰. So we make the assumption to know input stream length only for time measuring and comparison purposes.

So, supposing to have an input stream of length `N_size` tasks, this allows us to compare our Farm model with a Data Parallel one. The latter will send to the GPU a single data structure in once, having a suitable size such that in total it will allow to perform the same amount of work that is done for Stream parallel version.

So pure data parallel computes a single and big data structure, in a canonical configuration kernel¹¹ and than sends back a single big data structure as result to the output¹².

⁹We'll explain other details on measures on Chapter 5.

¹⁰We already pointed out this aspect in Chapter 3 in Farm parallel pattern description.

¹¹A classic configuration, in general, is obtained experimenting for block size and get grid size dividing the data dimension by the block size.

¹²This is how generally the GPU is meant to be used, this is the kind of problems a GPU is designed for.

Below we'll show a brief code example of pure data parallel implementation for Simple-computational kernel.

Listing 4.8: Optimal Kernel caller for Simple-Computation kernel, uses Occupancy APIs to get best Block configuration

```

/** main function */
//data allocation
x = (float *) malloc(N_size*sizeof(float));
cosx = (float *) malloc(N_size*sizeof(float));
gpuErrchk( cudaMalloc((void*)&x_d, N_size*sizeof(float)) );
//init input data
Acquire Or Generate input Data Structure in "x"
//call kernel launcher
float msKer = optimalCosKer(M_iter, N_size, x, cosx, x_d, clocks, clocks_d);

/** Kernel launcher function */
float optimalCosKer( int m, int n, float *x, float *cosx, float *x_d){
    //Actual grid size needed, based on input size
    int gridSize;
    //Min grid size needed to achieve max occupancy in a full device launch
    int minGridSize;

    //Occupancy API and block/grid setting
    cudaOccupancyMaxPotentialBlockSize( &minGridSize, &BLOCK, cosKernel, 0, 0);
    GRID = (n + BLOCK - 1) / BLOCK; // Round up according to array size

    //data transfer and execution
    Event Measures Start

    gpuErrchk( cudaMemcpy(x_d, x, n*sizeof(float), cudaMemcpyHostToDevice) );
    cosKernel<<<gridSize, blockSize>>>(m, n, x_d);
    gpuErrchk( cudaMemcpy( cosx, x_d, n*sizeof(float), cudaMemcpyDeviceToHost) );

    Event Measures End

    gpuErrchk( cudaPeekAtLastError() );
    cudaDeviceSynchronize();
    return timeMeasure;
}

```

The peculiarity of this Kernel is that we exploited CUDA Occupancy APIs. The

occupancy-based launch configurator APIs, `cudaOccupancyMaxPotentialBlockSize`, heuristically calculate an execution configuration (thread block and grid sizes) that achieves the maximum multiprocessor-level occupancy[10].

This was one example of use for CUDA Occupancy calculator tools, in this case we used it to achieve the best block and grid configuration possible for our Kernel.

Again the code presented above is relative to Simple-computational kernel, but the implementation structure is analogous to the one for Matrix multiplication and Blur Box.

CHAPTER 5

Experiments

In this chapter will be shown all the performed experiments and their results. The first section starts from what we expect to get from different code tests and, to this aim, what kind of comparisons will be made and how tests were set up.

Then in second, third and fourth sections, for each kernel implementation, we will show the chosen datasets for each type of execution to test and results we get; in particular, here we'll show time measures and plots with some remarks. The last section gives a brief summary and some final consideration, furthermore it gives comparisons between stream parallel and data parallel version, for each kernel type.

5.1 Expectations

As previously mentioned, what we want to see is that our model and implementation for Farm parallel pattern can fit in a GPU. To this aim is necessary to gain a speedup in the order of the number of Streaming multiprocessors of the GPU we're running code. Let's clarify some concepts in the sentence above:

- The speedup will be estimated in terms of **GPU completion time**, i.e. the total time needed to perform all host/device data transfers and kernel executions for a certain application;
- We expect to have the best speedup only when we have certain conditions;
- The best speedup would be in the order of multiprocessors number.

The last point means we can't expect to reach greater gain than the available amount of hardware resources, further and specific definitions about this concept will be given in Speedup subsection.

The second point above means we can expect best performances in the following cases:

- When we have a regular kernel, that is a kernel with the lowest possible amount of branching and, thus, very low (or absent) threads divergence;
- When the kernel is more computational-bound than memory-bound, the less access to Global memory the less data transfer latency will slow down execution. Memory-bound applications, in fact, may generally lead to bad occupancy and/or kernels that could not overlap (or they do it rarely);
- When the kernel execution takes an amount of time near the one for data transfer and/or other kernel calls.

When one, or more, of the above conditions isn't met, we're aware to have a considerably lower speedup than what we expected.

5.1.1 Measures: What and How

Before the test setup and writing, it's important to understand what we should measure, in order to get significant comparisons.

First we recall that the measures of interest include *data transfers* and *kernel executions* and all completion times are reported in **milliseconds**.

In the case where CUDA streams are used, we have an additional time cost to create and destroy streams, especially when lot of streams are spawned.

If we want to have as many CUDA Streams as many SMs in device, then creation cost is in the order of hundreds of milliseconds and destruction is in the order of hundred of microseconds¹. However, we won't sum them up with measures on data transfer and kernel execution. This is because, even if the streams overhead can be notable, it's a one-time cost to pay.

This means that it won't weigh on performances of a Stream parallel application, given that initially we create CUDA streams, then we'll run kernels on a indefinitely long input stream (theoretically) and, only when input is totally consumed out, CUDA streams will be destroyed. So on a reasonably long input stream, the CUDA streams APIs cost should become negligible.

So focusing on data transfers and kernels, we put two time probes, one before the start of the input stream loop and one at the end. The time probes are implemented using **CUDA Events**.

Below will be reported a pseudo-code to clarify how the probes are placed inside the code:

Listing 5.1: Example of code for events utilization as time probes.

```
/* main function */
// Create CUDA streams
streamCreate(streams, nStreams);
// Create "start"/"stop" events and start measuring
createAndStartEvent(&startEvent, &stopEvent);
//input stream starts
int k = 0;
while (InputStream) {
    if (buffer x[i: i+chunkSize] is full)
    {
        int i = k%nStreams;
```

¹Measures on CUDA Streams spawn/deletion were collected with *nvprof* log file, where all CUDA APIs time are precisely measured.

```

        kernelCaller(input_host, output_host, input_device, output_device, streams[i],
                     streamBytes, ...);

        . . . .

        ++k;
    }
    else
    {
        add item to buffer x[i: i+chunkSize]
    }
}
// Stop time measuring (input stream has ended)
msTot = endEvent(&startEvent, &stopEvent);
// Destroy events
cudaEventDestroy();

/**** Events Creation and start function ****/
void createAndStartEvent(cudaEvent_t *startEvent, cudaEvent_t *stopEvent)
{
    gpuErrchk( cudaEventCreate(startEvent) );
    gpuErrchk( cudaEventCreate(stopEvent) );
    gpuErrchk( cudaEventRecord(*startEvent, 0) );
}

/**** Events end and time collection function ****/
float endEvent(cudaEvent_t *startEvent, cudaEvent_t *stopEvent)
{
    float ms = 0.0f;
    gpuErrchk( cudaEventRecord(*stopEvent, 0) );
    gpuErrchk( cudaEventSynchronize(*stopEvent) );
    gpuErrchk( cudaEventElapsedTime(&ms, *startEvent, *stopEvent) );
    return ms;
}

/**** Kernel caller function example ****/
void kernelCaller(input_host, output_host, input_device, output_device, streams[i],
                 streamBytes, ...)
{
    // H2D mem copy
    gpuErrchk( cudaMemcpyAsync(input_device, input_host, streamBytes,
                              cudaMemcpyHostToDevice, streams[i]) );
    // Kernel call
    kernel<<<GRID, BLOCK, 0, streams[i]>>>(input_device, output_device, ...);
}

```

```

    #ifndef MEASURES
        gpuErrchk( cudaPeekAtLastError() );
    #endif
    // D2H mem copy
    gpuErrchk( cudaMemcpyAsync( output_host, output_device, streamBytes,
        cudaMemcpyDeviceToHost, streams[i]) );
}

```

CUDA event APIs are a device-bound tool and they were chosen as inside-code measurement for several reasons. Another approach could be to use any CPU timer provided for C++ in a way such as:

```

t1 = myCPUTimer();
Kernel<<<GRID, BLOCK>>>(param0. param1, ...);
cudaDeviceSynchronize();
t2 = myCPUTimer();

```

A problem with using host-device synchronization points, such as `cudaDeviceSynchronize()`, is that they stall the GPU pipeline. Events, instead, provide a relatively light-weight² alternative to CPU timers via the *CUDA event API* [15]. This API includes calls to create and destroy events, record events, and compute the elapsed time in milliseconds between two recorded events, exactly as it's shown in code Listing 5.1.

CUDA events are of type `cudaEvent_t` and are created and destroyed with `cudaEventCreate()` and `cudaEventDestroy()`. In the above code `cudaEventRecord()` places the start and stop events into the default stream, or `stream 0` (also called the “*Null Stream*”) [10]. This holds for all device timers we introduced in our code.

The `cudaEventRecord()` will record a time stamp in device for the event, but only when that event is reached in the specified stream. The function `cudaEventSynchronize()` blocks CPU execution until the specified event is recorded.

The `cudaEventElapsedTime()` function returns the number of milliseconds elapsed between the recording of *start* and *stop*. This value has a resolution of approximately 0.5 microseconds [15, 10]. So those timers will be enough accurate for our purpose, since we'll see that almost all elapsed times will be from tens to thousands milliseconds.

²CUDA events make use of the concept of CUDA streams.

It's important to point out why we used events on the default stream. Given the asynchronous nature of CUDA calls, that we perform in non-default stream, the behavior and order in between different streams is unpredictable. This means that a call from a different non-null stream can actually be issued in between two events we're trying to recording, even if they were issued from the same non-default stream[10]. This is one of the reasons why we chose to put timers on default stream and, furthermore, outside the loop over input stream. We could have inserted events inside the loop instead, but in that way we'd have measured singularly each iteration³ and sum up all those elapsed times. There would have been three problems with that approach:

- Each "end" event, must be sure to measure everything until the ending event, that's why it's necessary to introduce `cudaEventSynchronize()`;
- Given that the input stream should be quite long, all those timers in each loop iteration would have introduced an amount of undesired sampling overhead, coupled with useless synchronization time.

For first problem, we recall that `cudaEventSynchronize()` blocks CPU execution until the specified event is recorded and we really want to avoid that. We should avoid as much (explicit) host-device synchronizations as we can: given that we're working on input/output streams of items from host, "stopping" this flow on host side at each iteration would invalidate the gain of our model (especially in terms of overlapping), increasing the overall completion time (probably for a non negligible amount).

The second problem is related to the first. Even if events are a light-weight solution for device activities timing, it doesn't mean they don't introduce a bit of overhead (in addition to the synchronization one) in both host and device side.

For completeness, we'll show some performances case of interest measured by profilers, in addition to those from timers. In designing and implementation phase, this allowed us, not only to observe the correctness of some measurements, but also to check

³And so measure each single memory copy H2D, Kernel execution and memory copy D2H.

some special cases and their relative technical details and performance analysis.

5.1.2 Tests setup

First we must point out that the target machines, where we have ran tests on, *weren't available in exclusive access*. So, all collected time measures, could be affected by a perturbation introduced from other processes, that aren't under our control.

Anyway, during the tests execution we set an automatic monitor for the resources occupancy of target machines. In particular, we made log files where there was printed the informations obtained by `top` and `nvidia-smi` commands⁴.

Once we determined the time measurement criterion, we had to decide what behaviors we wanted to observe from our code.

Note that for each type of input dataset, we run multiple times (more precisely 5 times) the executable so that, for a certain input setup, we can collect several time measures. This allows us to delete some *outliers* completion times, as they may distort the result, and then we take the mean value among the remaining measures.

Moreover, as we mentioned in Subsection 2.5.1, we implemented our tests as bash scripts. These scripts will cover the task of:

- Compiling a certain executable, exploiting the rules available in our Makefile;
- Run that executable $N_{test} - 1$ times and then redirect the output, of the running application, to a specific `.txt` file;
- Run for the N_{test}^{th} time the executable via `nvprof`, redirecting the profiler output to a folder of `.txt` log files

⁴`top` gives us informations on the machine state on host side, while `nvidia-smi` allows to monitor NVIDIA GPUs state. By state we mean the main running processes, utilization percentage, used memory etc.

In next sections we'll show, for each type of kernel, what type of tests have been made and relative results.

It's important to recall that *input stream length shouldn't be known a priori*, but in tests we'll see that we have to give an input limit. This is for time measuring purpose only, because we need to have a knowledge on what and how much data we are measuring.

5.1.3 Speedup

An important metric related to performance and parallelism is **speedup**, it's a derived measure from completion time measures. Speedup compares the latency for solving a certain computational problem with its *best known* sequential implementation (on the target architecture available), versus solving the same problem on P hardware units, generally called *workers*, as below

$$speedup = S_P = \frac{T_{seq}}{T_P} \quad (5.1)$$

where T_{seq} is the sequential execution time and T_P is the parallel completion time, using P parallel resources[7].

An algorithm that runs P times faster on P processors is said to exhibit **linear speedup**. It is rare in practice, since there is extra work, involved in distributing work to processors and coordinating them. This extra work clearly introduces extra time, also known as **overhead**.

While **sublinear speedup** is the norm, there may be exceptions, i.e. occasional programs exhibiting **superlinear speedup**⁵[17].

Amdahl's Law gives an important limit on speedup: it considers speedup as P varies and the problem size remains fixed.

⁵In general, some causes of superlinear speedup may be: restructuring a program for parallel execution can cause it to use memory better (cache in CPU implementations), even with a single worker; or the parallel algorithm may be able to avoid work that its serialization would be forced to do.

Amdahl identified in the execution time T_{seq} of a program, two categories: time spent doing *non-parallelizable serial work* and time spent doing *parallelizable work*. Call these T_{ser} and T_{par} , respectively.

Given P workers available to do the parallelizable work, the times for sequential and parallel execution are:

$$T_{seq} = T_{ser} + T_{par} \quad (5.2)$$

$$T_P \geq T_{ser} + \frac{T_{par}}{P} \quad (5.3)$$

The bound on T_P assumes no superlinear speedup, and is an exact equality only if the parallelizable work can be perfectly parallelized.

Plugging relations in 5.2 and 5.3 into the definition of speedup in Eq.5.1, we gave before, yields **Amdahl's Law**:

$$S_P \leq \frac{T_{ser} + T_{par}}{T_{ser} + T_{par}/P} \quad (5.4)$$

Let f be the non-parallelizable serial fraction of the total work. Then the following equalities hold:

$$\begin{aligned} T_{ser} &= f \cdot T_{seq} \\ T_{par} &= (1 - f) \cdot T_{seq} \end{aligned} \quad (5.5)$$

Substituting Equations 5.5 into speedup equation 5.1:

$$S_P \leq \frac{1}{f + (1 - f)/P} \Rightarrow S_\infty \leq \frac{1}{f} \quad (5.6)$$

Speedup is *limited by the fraction of the work that is not parallelizable*, even using an infinite number of processors[17, 7].

5.1.4 Results: gathering and evaluation

From all `.txt` files, containing time measures for all the execution run from tests, we have to manipulate results and do some calculations.

In particular, implemented Python scripts⁶ provide a tool to:

- First of all, output all necessary `.csv` containing all averages on the measures obtained by the multiple runs for a certain input set⁷;
- Then from all those average Completion times, in `.csv` format, another script computes all **Speedups**;
- Finally, the same script that computes speedups and then it outputs plots on most significant measures and results.

It's important to point out what kind of speedups will be computed, so that in next sections we can present numeric and graphic results.

Remembering what we introduced in Section 5.1.3, the speedup, in brief, is the ratio of the time spent in sequential version to the time spent on parallel version, having P workers (see Equation 5.1).

Here we have to define what those times correspond in our implementation:

- T_{seq} , the sequential version, is the case in which CUDA Streams aren't used⁸. In a sense, this corresponds to serialize all data transfers and kernel execution as

$$H2D_0, Ker_0, D2H_0, H2D_1, Ker_1, D2H_1, \dots$$

So, even if we have a stream of items as input, this means sending only a small task per time to the device, thus using only a small amount of computational resources⁹ at time;

- T_P , the parallel version, is the case in which CUDA Streams are used, where P will be the number of non-default streams spawned.

⁶See Chapter 2.

⁷Remember that outliers values are rejected, and mean measures are computed on remaining values.

⁸More precisely only default stream is used, instead non-default streams aren't.

⁹E.g. this can be seen as using a single SM for all the duration of computations on farm input stream.

In the particular scenario of a Farm for GPU, the number of workers has a more complex meaning. Those workers, more precisely, corresponds to how many Streaming Multiprocessors we're going to use in the GPU, i.e. the target number of SMs we want to make busy at computation peak time.

So the speedup, obtained from those two implementations, should also give us an indicator on how many SMs are effectively used. In other words, here we can see a CUDA stream as a sort of channel in which we put tasks and we want those channels to successfully overlap, hiding data transfer time from/to device by executing multiple kernels at the same time. So, the computed speedups are:

1. $S_3 = \frac{T_{seq}}{T_3}$ as we mentioned before, this is a base case for CUDA streams usage;
2. $S_{\#SM} = \frac{T_{seq}}{T_{\#SM}}$ this is the special case that aims to show the Farm parallel pattern fitting in GPUs architecture.

The way we defined speedup lead us to ask what is the best we can achieve and this is where Amdahl's law can be applied.

First we should define what completion times, upon which we based the analysis, are. We start by focusing on the total completion time for Farm in GPU, that we introduced in Chapter 3. We can extend the Equation 3.5 to have an approximation of all possible times in our Farm for GPU

$$T_{Tot} = T_{InStream} + T_{Comp} + T_{OutStream} \quad (5.7)$$

Below we'll explain this formula's components:

1. T_{Tot} is the overall time it takes to compute n elements from an input stream, i.e. latency from the first available item on the input stream, until the last result item is sent to the output stream (this can involve both host and device elapsed times);

2. $T_{Instream}$ this is the time it takes to obtain all –data parallel small– tasks from input stream (host measure);
3. T_{Comp} is the time given by the Equation 3.5 (device measure);
4. $T_{Outstream}$ the time it takes to send all results to the output stream (host measure).

Obviously the measures we performed and all next considerations will be based only on device completion time, i.e. point 3 from the above list.

We recall that our tests rely on an important assumption: we don't know how much the real length of the input stream is (call it n), that's why we focus our measures on a reasonably long portion of the input stream, e.g. take as limit an l such that ($l \leq n$ elements will be the stream limit).

So, focusing our attention exclusively to device time, from Equation 5.7 we'll only consider T_{comp} , plugging it with Equation 3.4, then we get

$$T_{comp} \approx \frac{T_{seq}}{n_w} + T_{ov} \approx \frac{(T_{H2D} + T_{ker} + T_{D2H})}{n_w} + T_{ov} \quad (5.8)$$

In other words T_{Comp} is approximatively the time needed to send items of the input stream to the GPU, make computations on them and then send back the results item to host¹⁰. In the specific case of our measures in tests this will be the time employed for all of the l items.

As we told before, host elapsed times aren't in the study domain of this thesis. So, from Equation 5.7 what we wanted to parallelize is T_{Comp} , in particular we want all l tasks to run in parallel on n_w workers. Clearly, we're also assuming that $l \gg n_w$ where $n_w = \#SM$, i.e. our tests will run on a limited input stream, but still ensuring that we have many tasks to send for each CUDA stream, consequently more tasks per SM.

Following all these assumptions, it's normal to ask what would it be the maximum reachable speedup and to define this we exploit Amdahl's law, recall Equation 5.6.

¹⁰As we mentioned in 3.1.1 the overhead is given by all intervals where we can't get perfect overlapping.

Recall that according to Amdahl, serial code can be identified in two categories: serial fraction and parallelizable fraction.

Considering our sequential code T_{Seq} , theoretically we should be able to parallelize all of it using our Farm model and CUDA streams. So, ideally, we can assume that it's all parallelizable. This leads trivially to: $f = 0$ and $(1 - f) = 1$.

Merging this with the Amdahl's upper bound we obtain:

$$S_P \leq \frac{1}{f + (1 - f)/P} = \frac{1}{1/P} = P \quad (5.9)$$

Since we have a limited amount of resources, i.e. Streaming Multiprocessors number, exploiting Equation 5.9 we can conclude that the maximum speedup we can achieve is: $Sp_{\#SM} \leq \#SM$. This formally proves our expectations.

In the next sections we will report all completion times and speedups, that are mostly representative. For each type of kernel that was implemented, we'll show inputs, tests and results (with some graphics and plots).

5.1.5 Computation-bound and memory-bound

In Section 3.1.1 we presented a performance model for Farm parallel pattern, then we coupled that with speedup and Amdahl's law in the previous section. These performance models are the most used to predict (approximately) performances.

Anyway, in this study we're interested also in another performance model: ***Roofline Model***.

Often off-chip memory bandwidth is a constraining resource in system performance. Hence, we want a model that relates processor performance to off-chip memory traffic[\[29\]](#). To this aim we define some important concepts:

- **work W** denotes the number of operations performed by a given kernel or application. This metric may refer to any type of operation according to the study necessity (e.g. the number of integer operations, the number of floating point operations (FLOPs), etc.). In the majority of the cases however, W is expressed

as FLOPs.

Note that the work is a property of the given kernel or application and thus only partially depend on the platform characteristics.

- **memory traffic** Q denotes the number of bytes of memory transfers incurred during the execution of the kernel or application. In contrast to W , Q is heavily dependent on the properties of the target machine[30].
- **arithmetic intensity** I , also referred to as operational intensity, is the ratio of the work W to the memory traffic Q :

$$I = \frac{W}{Q} \tag{5.10}$$

and denotes the number of operations per byte of memory traffic. When the work W is expressed as FLOPs, the resulting arithmetic intensity I will be the ratio of floating point operations to total data movement (FLOPs/byte)[32, 30].

In particular, the term “operational intensity” generally means operations per byte of DRAM traffic. That is, we measure traffic between the caches and memory rather than between the processor and the caches. Thus, operational intensity predicts the DRAM (global memory) bandwidth needed by a kernel on a particular computer[29, 30].

The *naive Roofline* is obtained by applying simple bound and bottleneck analysis. In this formulation of the Roofline model, there are only two parameters, the *peak performance* and the *peak bandwidth* of the specific architecture, and one variable, the *arithmetic intensity*.

The peak performance, in general expressed as GFLOPs¹¹, usually it’s derived from architectural manuals, while the peak bandwidth, is generally obtained via benchmarking[29] (in our case we found it from the features manual of our two target machines).

¹¹Floating point operations per second are called FLOPs, it is a measure of computer performance, useful when we’ve floating-point calculations. GFLOPs, are simply giga FLOPS.

The Roofline model gives a plot, that is derived by the following formula:

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases} \quad (5.11)$$

where P is the attainable performance, π is the peak performance, β is the peak bandwidth and I is the arithmetic intensity[31].

The point at which the performance saturates at the peak performance level π (that is where the diagonal and horizontal roof meet), is defined as ridge point.

The ridge point offers insight on the machine's overall performance, by providing the minimum arithmetic intensity required to achieve peak performance.

A given kernel or application is then characterized by a point given by its arithmetic intensity I (on the x-axis). The attainable performance P is then computed by drawing a vertical line that hits the Roofline curve.

So, the kernel or application is said to be **memory-bound** if $I \leq \pi/\beta$.

Conversely, if $I \geq \pi/\beta$, the computation is said to be **compute-bound**[29, 31].

We'll give an example on two of our kernel applications, mainly to prove whether they're memory or compute bound.

To simplify this example, consider the amount of computations and memory operations needed to produce a single floating point number as output. Moreover, we consider memory traffic as simple count of loads/stores, instead of single bytes (as we're considering an example on individual floats, that is always 4 bytes).

First, consider the Simple-computational kernel and assume to set it to perform M iterations (e.g. take 10000¹²) to produce a single float as output. At each iteration we perform a single-precision cosine (`cosf`) that is internally computed in approximately

¹²We'll see in next sections that this is the minimum amount of computations that kernel had to perform in our tests.

10 FLOP¹³. Furthermore, to get a single output, we perform 2 memory operations, one load and one store between registers and global memory.

So we get as computational intensity: $I \approx \frac{10}{2} \cdot M = 5 \cdot M$, where M is the number of times we repeat arithmetic operations. For example for $M = 10\,000$ we get $I = 50\,000$.

Now we make an example on matrix-multiplication kernel. Call k the amount of floating-point numbers upon which we have to perform computations to get each floating-point results (i.e. a generic element $C[i, j]$).

So, for each output, the kernel performs $2 \cdot k$ FLOP (one sum, one multiplication for each iteration). Furthermore, we have $2 \cdot k$ loads (the couples of elements from input matrices that will be multiplied) and 1 store (one element of the result matrix), between global memory and registers.

So computational intensity will be given by: $I = \frac{2 \cdot k}{(2 \cdot k) + 1}$.

Summarizing the results of the above examples, we can observe that in first kernel, computation intensity simply will proportionally grow, as the number of iterations will increase.

While the second kernel shows a different behavior, i.e. computation intensity doesn't depend on the number of iterations of the kernel, since it will give always a result < 1 (we have more memory than compute operations).

According to the Naive Roofline model we can determine if an application is memory or compute bound, by check respectively if $I \leq \pi/\beta$ or $I \geq \pi/\beta$.

From architecture manuals[26] and specifications¹⁴, we obtained

$$\frac{\pi}{\beta} \approx \frac{10600 \text{ GFLOP/s}}{732 \text{ GB/s}} = 14,48 \text{ FLOP/B in P100 device and}$$

$$\frac{\pi}{\beta} \approx \frac{6840 \text{ GFLOP/s}}{288 \text{ GB/s}} = 23,75 \text{ FLOP/B in M40 GPU.}$$

¹³Here we can see a discussion on amount of FLOP per mathematical operation:

<https://devtalk.nvidia.com/default/topic/983983/>

¹⁴We checked the CUDA Device Query too.

Merging it with the above kernel examples of different arithmetic intensity, we get¹⁵:

- simple-computational kernel $\implies I \approx \frac{10}{4(\text{bytes})} \cdot M = 2.5 \cdot M$
- matrix multiplication $\implies I = \frac{8 \cdot k}{(12 \cdot k) + 1} \approx 1$

So we can conclude that for matrix multiplication computation intensity will be $I \leq \pi/\beta$ for both target machines.

While in simple-computational kernel, we have a computation bound behavior yet for a small amount of M (e.g. for $M=10$ the kernel is already considerable as compute-bound for both target machines, we recall that the minimum amount we used in tests is $M = 10000$).

5.2 Simple-computation kernel

For this computation-bound kernel, for each type of input dataset, we identified three different kernel iterations amount, call it M , it takes the following values: 10000, 400000, 800000. These values identify how many times the kernel will have to repeat a certain mathematical operation (in our case the single-precision Cosine). This is what makes the computation load variable.

Another important parameter is the Block size that we set to $BLOCK = (1024, 1, 1)$. We recall that 1024 is the maximum we can give to x and y block dimension, this holds for both of the GPUs we used to run tests, i.e. **P100** and **M40**.

The choice for 1024 was made according to CUDA Occupancy APIs, that suggested this as best block size for the considered application. In general, but it's not a strict rule, computation-bound kernels perform at their best on higher block size, because this should allow us to use the maximum number of threads possible and, thus, to use as much computational resources as possible.

¹⁵Now we'll consider memory operations in terms of bytes, instead of load/stores number, to uniform them with the model.

Tesla P100	Tesla M40
57 344	24 576
114 688	49 152
229 376	98 304
458 752	196 608
917 504	393 216
1 835 008	786 432

Table 5.1: Input dataset for Simple-Computation kernel, these are the input stream limits for both devices.

All types of tests performed on Simple-computational kernel are:

1. **Classic data parallel approach**

Here we launch the execution of our simple-computation kernel, as it would be classically used: as fully data parallel application.

This means that we have a single big data structure, that we'll send entirely to the GPU for data parallel computations.

Clearly this collection of data will be computed by the same kernel and performing the same amount of work w.r.t. the Stream parallel version.

So, essentially, we're trying to model a problem from the perspectives of two different parallel pattern. It's important to point out that stream parallel will model input/output streams, of unknown length, made by small data parallel tasks, while data parallel is modeling a single and known-sized big data structure¹⁶.

In Table 5.1 we show length that was used. In data parallel those values are the known sizes of the data structure upon which data parallel computations will be performed.

¹⁶E.g. Simple-computation's data parallel version could be used to make data parallel computations over a given big and unique array of floats.

Instead, in stream parallel we'll have different input streams, formed by small data parallel tasks. Anyway, to compare two such different parallel patterns, we ensured that stream of tasks will give the same total amount of work, w.r.t. the one given by data parallel version.

Note that it's wrong to think that data parallel version is the equivalent of bringing together tasks from input stream of stream parallel problem¹⁷.

2. **Stream parallel with smaller tasks** Here we're facing the Farm parallel pattern for GPU, but with smaller tasks size. As we mentioned in Chapter 3, we're trying to get maximal occupancy, especially in a computational-bound kernel.

So, given that the goal of our code is fully or partially filling a SM with a kernel call, we had to take into account of:

- How big should have been each task for each kernel execution;
- Consequently, how many thread blocks our kernel would have to issue.

These choices followed from our devices features. Even though the two GPUs are located in different Compute Capabilities (P100 is c.c. 6.0, M40 is c.c. 5.2), they have the same limits for

- Resident **threads** per SM = 2048 (equivalent to 64 resident warps per SM);
- Resident **thread blocks** per SM = 32.

The second limit means that we can have at most 32 active thread blocks, i.e. running on a certain Streaming Multiprocessor. The first limit, instead, is our main goal here. Having at most 2048 active threads in a SM and having configuration of `blocks = (1024, 1, 1)`, we will have at most two resident blocks in a SM.

¹⁷Here, as in all data parallel versions we implemented, we don't make use of CUDA Streams, they'd be useless since we're launching a single kernel on a single huge data structure

The execution configuration for smaller tasks version is such that we want to have 1024 as task size and, so, a kernel configuration such as `<<<1, 1024>>>`. So here each launched kernel will have one block containing 1024 threads and this theoretically should correspond to half the occupancy of a SM.

Clearly the code will send a lot of tasks to device, i.e. enough to hopefully fill all SMs (at peak work load). We recall that all of the tasks should give a total amount of work limited according to values in Table 5.1.

All of the above mentioned configurations will be tested for the following CUDA streams cases:

- **Zero** CUDA Streams. This is the scenario where we use only default CUDA stream, so we'll have serial and synchronous data transfers (w.r.t. the host). Kernel are still an asynchronous call, but immediately after we want to have data back from device and this means to have a `cudaMemcpy`, i.e. a synchronous call;
- **Three** CUDA Streams. Here we'll use 3 non-default streams, because we want to observe the behavior of our code in a sort of base case. In general, using three CUDA streams is the classic approach for devices with two copy engines. This means it's the minimum to expect an overlap such as a kernel and at most two simultaneous data transfers;
- **N_{SM}** CUDA Streams, with $N_{SM} = \#Streaming\ Multiprocessors$. This is the special case because, in general, applications don't use such a high number of non-default Streams. But in our case it's necessary to try to achieve the expected speedup, with respect to the version without CUDA streams (we'll sometimes refer to as "zero-stream" version).

Clearly, at a certain time, say t_i , we can have at most two data transfers but there's no limit on kernel calls, clearly they will be effectively executed as long as there are available resources on the device.

3. **Streaming parallel with bigger tasks** This tests setup has similar premises to the one for smaller (data parallel) tasks, clearly the only thing is changing is the task size, that will be set to 2048.

Thus, having always `blockSize=(1024, 1, 1)`, the code will set `gridSize=(2, 1, 1)`, so we'll have two blocks, each covering calculations on one half of the task. So having such configurations, we should have ideally a fully occupied SM. This can sound as having better performances, with respect to smaller tasks, but, as we said before, it's not a strict rule to have better performances on maximum occupancy. We'll see from results that instead this approach behaves worse than smaller tasks.

Clearly we reused the above mentioned amount of CUDA streams, so for this type of execution we ran code using: **Zero**, **Three** and **N_{SM}** CUDA Streams.

5.2.1 Results

All the above tests on Simple-Computation Kernel give us the measures of device times, on which most of observations will rely on. We can group the measures into two parts: Smaller tasks and Bigger tasks.

Smaller tasks

All collected elapsed times for 1024-sized tasks are reported in Table 5.2, for the zero-streams version, and Table 5.3, for the SM-streams version.

From measures in Table 5.2(zero-streams) we can see that Completion Time, fixed an input stream length, increases proportionally with iterations number (e.g. completion times for 40 000 iterations kernel compared to the one for 1000, is almost $40\times$ bigger). This holds on both of devices.

This is a further sign that this type of kernel is computation-bound.

	Tesla P100 (zero stream)			Tesla M40 (zero stream)		
M iterations	Event Times	Tasks Num	Tot.Work Amount	Event Times	Tasks Num	Tot.Work Amount
10000	4622.86	56	57344	693.747	24	24576
400000	181465.333			27453.933		
800000	361199.666			54888.933		
10000	9294.18	112	114688	1382.363	48	49152
400000	281507			54901.6333		
800000	407750.666			109783.666		
10000	10217.933	224	229376	2765.323	96	98304
400000	407779.666			109799.333		
800000	815513.666			219553		
10000	20433.633	448	458752	5528.96	192	196608
400000	815561			219589		
800000	1631013.333			439097.666		
10000	40865.4	896	917504	11058.433	384	393216
400000	1631096.666			439192.333		
800000	3261986.666			878195		
10000	81731.733	1792	1835008	22112.6	768	786432
400000	3262250			878433		
800000	6617950			1756373.333		

Table 5.2: Device completion times for Simple-computation kernel, without using CUDA Streams, results are reported for both machines (P100 and M40).

	Tesla P100 (56 Streams)			Tesla M40 (24 Streams)		
M iterations	Event Times	Tasks Num	Tot.Work Amount	Event Times	Tasks Num	Tot.Work Amount
10000	104.772	56	57344	30.6074	24	24576
400000	3968.913			1178.72		
800000	7818.843			2355.413		
10000	205.729	112	114688	60.208	48	49152
400000	7828.193			2358.656		
800000	15691.833			4714.36		
10000	407.712	224	229376	119.3446	96	98304
400000	15687.966			4715.123		
800000	31396			9425.92		
10000	803.223	448	196608	238.249	192	196608
400000	31422.033			9429.89		
800000	62818.7			18853.966		
10000	1619.586	896	917504	475.590	384	393216
400000	62793.4			18856.8		
800000	125575			37705.666		
10000	3229.063	1792	1835008	949.497	768	786432
400000	125547.666			37711.9		
800000	251503			75445.266		

Table 5.3: Device completion times for Simple-computation kernel, using as many CUDA Streams as SM number, results are reported for both machines (P100 and M40).

Furthermore, in the input data set for tests on Simple-computational kernel, we set the tasks number to increase, for each different test, by a factor of 2¹⁸.

Always looking at Table 5.2, fixed a number of iterations, we can see that even completion times grows by a factor of 2, as the tasks number grows.

This confirms that, no matter how many tasks the input stream sends, no matter how many iterations the kernel does, *we'll have a completion time directly proportional to the computations amount*, performed by the simple-computation kernel.

Now turning on Table 5.3, we can see the same behavior just presented for zero-streams version. In SM-streams version too we can observe that measures grows as computations amounts grows. However it's immediate to see that zero-streams and SM-streams have completely different completion times.

Note that in those tables we reported both the tasks number given as input and the correspondent total amount of work they effectively produce.

This leads us to compute speedups, following the approach explained in Section 5.1.4. All of the speedups are shown in Table 5.4. In this table the most important columns are $Sp(3)$ $Sp(SM)$, those columns stands for:

$$Sp(3) = \frac{T_{seq}}{T_3} \text{ and } Sp(SM) = \frac{T_{seq}}{T_{SM}}.$$

To have an overall view on speedup we also present plots for both P100, in Figure 5.1, and M40, in 5.2.

The two plots show the speedups only for a part of the real input dataset; in particular, each plot shows the smaller and bigger tasks amount and for both it shows the smaller and bigger kernel iterations number.

From the two plots we can clearly see how performances increase proportionally with the number of CUDA streams¹⁹.

¹⁸This means we're increasing the pressure of tasks that each SM will have to compute in total

¹⁹More precisely this gain is bounded to the number of Streaming Multiprocessors the GPU exploits.

	Tesla P100 (56 Streams)			Tesla M40 (24 Streams)		
M iterations	Tasks Num	Sp(3)	Sp(56)	Tasks Num	Sp(3)	Sp(24)
10000	56	5.337	44.122	24	2.976	22.665
400000		5.246	45.721		2.963	23.291
800000		5.221	46.196		2.963	23.303
10000	112	5.366	45.176	48	2.967	22.959
400000		4.069	35.960		2.962	23.276
800000		2.947	25.984		2.963	23.287
10000	224	2.988	25.061	96	2.970	23.170
400000		2.986	25.993		2.963	23.286
800000		2.986	25.975		2.962	23.292
10000	448	2.988	25.439	192	2.967	23.206
400000		2.986	25.955		2.963	23.286
800000		1.940	25.963		2.963	23.289
10000	896	1.635	25.231	384	2.967	23.251
400000		1.689	25.975		2.963	23.290
800000		2.861	25.976		2.963	23.290
10000	1792	2.998	25.311	768	2.968	23.288
400000		2.996	25.984		2.964	23.293
800000		2.654	26.313		2.963	23.280

Table 5.4: Speedups for all data sets of simple-computation kernel. Results are reported for both devices.

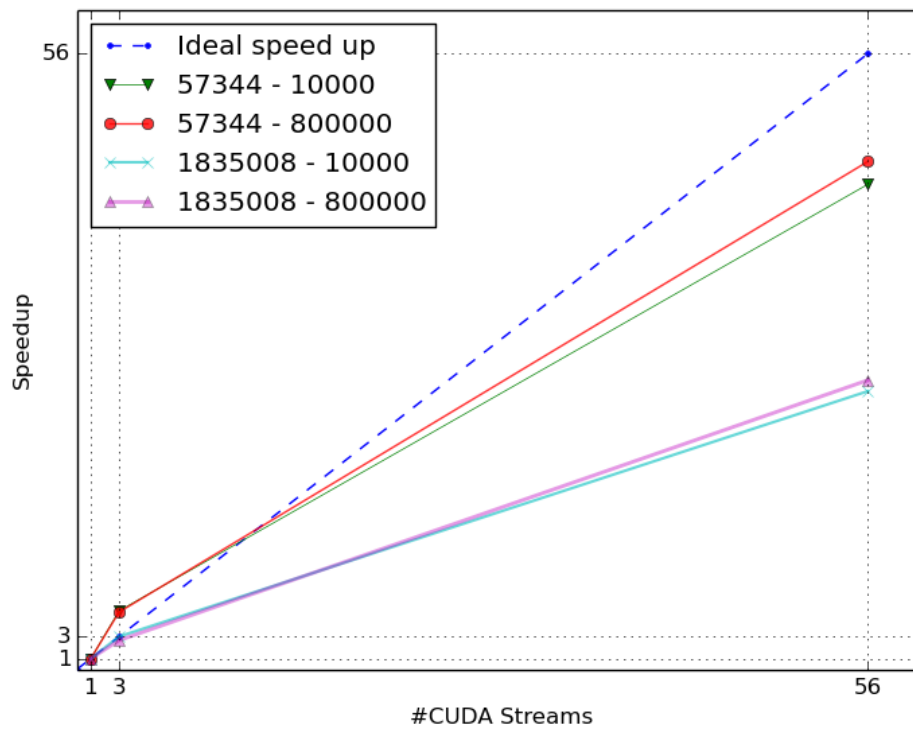


Figure 5.1: Speedup for 3 and 56 CUDA streams on P100 device.

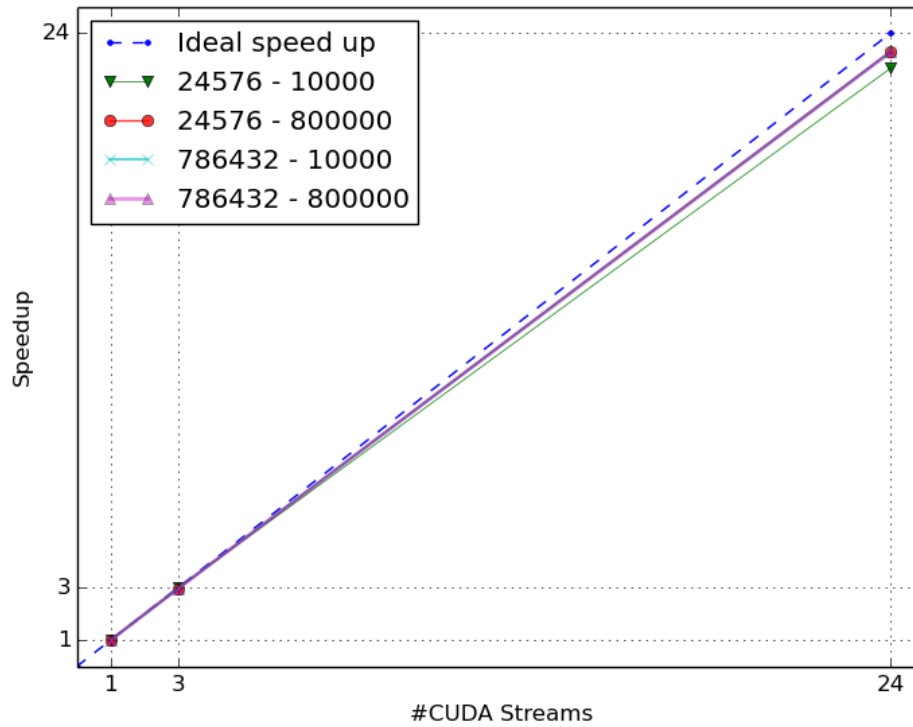


Figure 5.2: Speedup for 3 and 24 CUDA streams on M40 device.

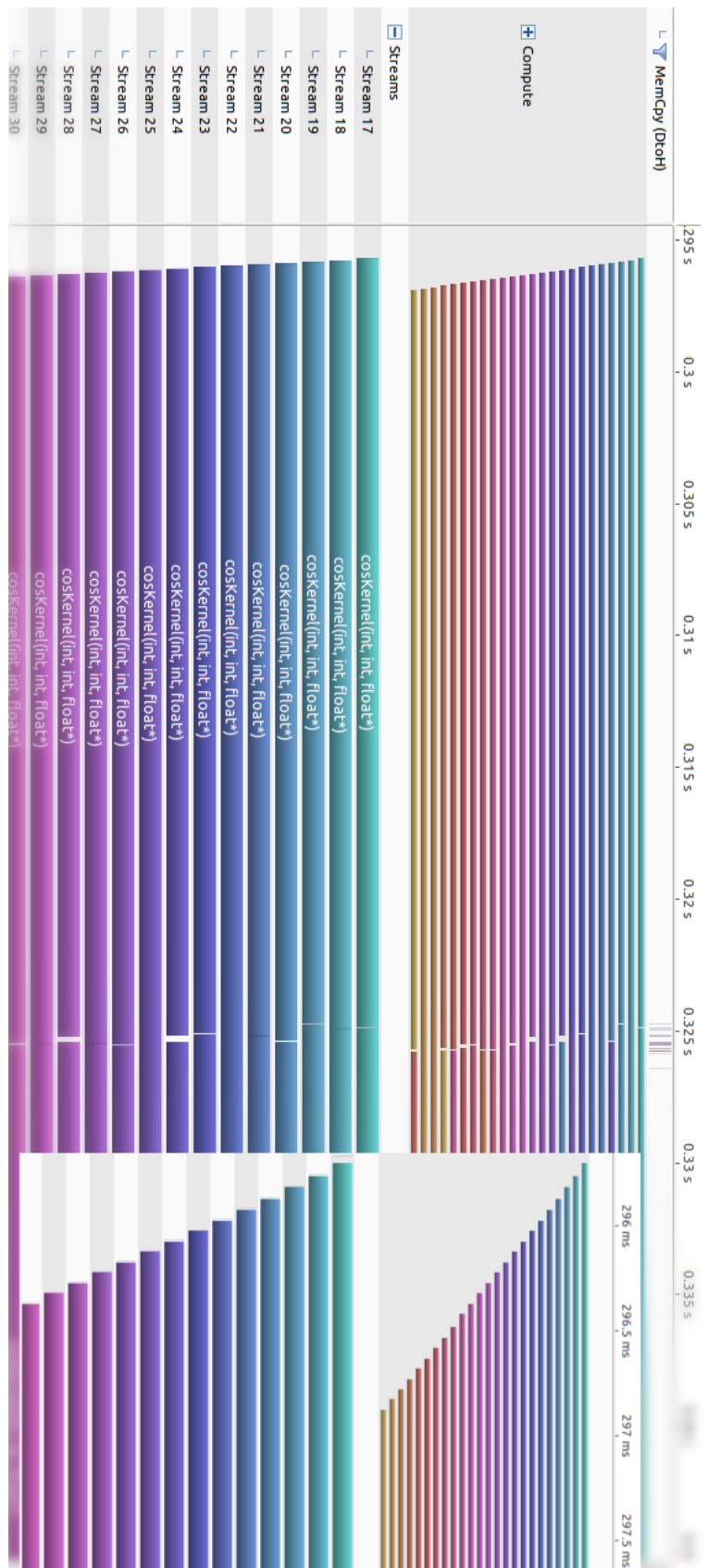


Figure 5.3: Profiling for an example of execution: number of tasks 768, total amount of work load 786432 floats, kernel iterations 10 000, 24 CUDA streams, on M40 device.

We can have a further proof of the achieved speedup, by running the **NVIDIA Visual Profiler**, we can see a portion of timeline representation in Figure 5.3. It's evident that we have a good overlapping between CUDA Streams, on the right side of the figure we can see a zoom in to the "*stabilization phase*", i.e. the time interval it takes for the program to fill CUDA streams with tasks and send them out to the device, until we reach a peak work rate and that "stairs" behavior vanishes.

However the profiler analysis reported some issues in our code, for example the profiling showed a too much low grid/block size and a poor memory copy overlapping (having 2 data transfers at the same time). But all these issues are really dependent on the "extreme" use case we're approaching in order to achieve stream parallel computations and, in any case, these issues don't have a bad impact in this particular application.

A good sign from the profiler is that the average usage of SMs, for a certain kernel call, is equal to one almost fully used SM. And this is exactly what we wanted to achieve, so this allows CUDA streams to spread kernel calls on all SMs, as soon as we reach the maximum work rate.

Bigger tasks

As expected, we get about half the ideal speedup.

This is because chunks of 2048 should take up the maximum possible active threads per SM. This means having, for each kernel call, a grid containing 2 blocks, each of size 1024.

So it seems that performances, in this kernel, are related to grid size. Recalling the smaller tasks, the profiler showed an almost full occupancy of one SM at $\lll 1, 1024 \ggg$ yet. So it makes sense to deduce that, $\lll 2, 1024 \ggg$ is a kernel configuration such that it occupies two SMs simultaneously.

This is the same of having about the half of available resources. We show a plot about this result, tested on P100, in Figure 5.4

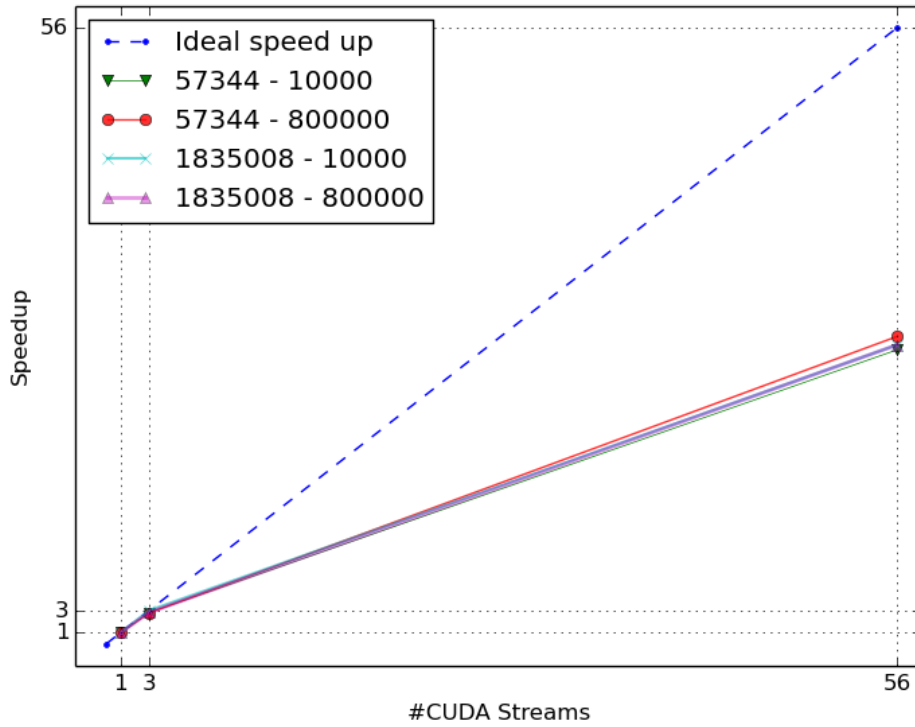


Figure 5.4: Sublinear speedup in bigger buffers execution, the performances are clearly halved.

5.3 Matrix Multiplication

With Matrix multiplication we're facing a memory-bound kernel, so we had to make some slightly different tests with respect to the ones in simple-computation kernel.

Note that, even if the Farm logic is the same as the previous application, there are some details to redefine.

First of all, before we were dealing with an input stream of small data parallel tasks formed by arrays of float numbers, here those small data parallel tasks are built by small *matrices* of float numbers. In particular as soon as we have two available input matrices (say A and B) from the input streams, they're sent out to device to apply the matrix multiplication kernel.

Finally we get back to host with the result matrix (say C), that will be one of the output stream components.

For simplicity, we're testing and measuring for square matrices case, even if code was

implemented for non-square case too.

Another assumption is that, for each Matrix Multiplication test, the kernel execution configuration was set up as follows:

assume N to be the matrices order, then $BLOCK = 32$ and $GRID = (N/BLOCK) + BLOCK - 1$, so we have:

`blockSize = (BLOCK, BLOCK, 1)`

`gridSize=(GRID, GRID, 1)`

So, we performed the executions with the following input data sets:

Tesla P100		Tesla M40	
Task Order Dim (Mat. Order)	Task Num	Task Order Dim (Mat. Order)	Task Num
128	225	64	100
256	441	128	196
512	900	256	400
1024	1764	512	784
2048		1024	
Tot. Work Load		Tot. Work Load	
1920		1280	
2816		1792	
3840		2560	
5632		3584	
7680		5120	
11264		7168	

Table 5.5: Input dataset for Matrix Multiplication kernel. Above Stream parallel configuration (task size and task amount), below the total work load for the overall execution.

1. **Classic data parallel approach** The fully data parallel version here, is totally analogous to the one explained for Simple-computational kernel.

Here we have again big data parallel data structures (in this case big matrices), upon which we're performing data parallel computations (i.e. a single matrix multiplication between two big matrices, having known size). In Table 5.5, in the lower portion, are showed the values that represent the overall work performed by all tasks in a stream, this amount is used as term of comparison with pure data parallel version of matrix multiplication. So we'll have big data structures having a known size, that will be assigned using the values in the lower part of Table 5.5, this is because we have to make a fair comparison between stream and data parallel.

Again we shouldn't confuse a totally data parallel matrix multiplication on single and big data structures, with a stream parallel version having as items small data parallel tasks.

So data parallel must not be intended as a grouping of tasks give by a stream parallel version. Instead, the values presented in Table 5.5, below the ones used for Data Parallel and above the ones for Stream parallel, should be considered as simple indicators to compare the two versions in a situation in which they're computing the same amount of overall work (and clearly they're modeling the same problem, but on totally different types of input).

2. **Stream parallel** As we mentioned before, we have to put a limit on the amount of tasks arriving from both streams²⁰.

In the upper portion of Table 5.5 are reported all relative values for tasks amount and, for each of them, we test different input tasks sizes (matrices order).

Note that values for tasks number from Table 5.5 means how many tasks will be

²⁰As in matrix multiplication we have to multiply 2 matrices per time obtaining one as result, we suppose to have two input streams, giving us two tasks per worker. Workers in turn will output one item per time on the output stream.

arriving from each of the two input streams, i.e. more precisely these values represent how many matrix multiplications will be performed (and, so, how many result items should be sent to the output stream). For every combination given by the input parameters, we'll test for different numbers of CUDA streams: **Zero**, **Three** and N_{SM} non-default Streams (with $N_{SM} = \#Streaming\ Multiprocessors$). The above test on different numbers of CUDA streams, is implemented in a totally analogous way to the one for Simple-computation Kernel.

5.3.1 Results

All the above tests, on Matrix Multiplication Kernel, give us the measures of device times.

Below we'll see that completion times and performance will notably be different, with respect to the previous computation-bound application.

All collected elapsed times are reported in Table 5.6, for the zero-streams version, in Table 5.7, for the three-streams one, and Table 5.8, for the SM-streams version.

From these tables we can highlight some behaviors:

- we made tests for several input stream's tasks total amount so, fixed a certain task size, we have the number of tasks growing by a factor of 2 (i.e. tests are set up in such a way that we execute an application for growing input parameters, in this case the tasks number per SM doubles) and we can see that this makes a proportional increase in completion times, i.e. even measures grows by factors of 2;
- each tested input task's size again grows of $2\times$ each, but in this case the completion times don't grow proportionally. Fixed a certain number of tasks arriving from the input stream:
 - for zero-streams we can see that, as each input parameter for tasks size grows

Tesla P100 (zero Streams)			Tesla M40 (zero Streams)		
Event Times	Task Num (#Mat.Mul)	Task Order Dim (Mat.Order)	Event Times	Task Num (#Mat.Mul)	Task Order Dim (Mat.Order)
86.8854	225	128	17.4869	100	64
175.189	441		34.8778	196	
359.9716	900		70.9718	400	
725.5573	1764		139.3896	784	
334.0376	225	256	36.2095	100	128
672.9463	441		74.2685	196	
1435	900		147.7336	400	
2828.5366	1764		299.138	784	
1673.2133	225	512	186.3913	100	256
3325.6533	441		368.6813	196	
6611.7566	900		786.7536	400	
12919.6666	1764		1603.4933	784	
10998.7666	225	1024	1256.4	100	512
21511.1666	441		2479.4333	196	
43828.7666	900		5162.6333	400	
85853.0333	1764		9791.98	784	
80764.8666	225	2048	9075.22	100	1024
158136.3333	441		17849.5666	196	
309724.6666	900		36441.3666	400	
604324	1764		72396.8666	784	

Table 5.6: Device completion times for Mat-Mul kernel, without using CUDA Streams (zero non-default streams), results are reported for both P100 and M40.

Tesla P100 (3 Streams)			Tesla M40 (3 Streams)		
Event Times	Task Num (#Mat.Mul)	Task Order Dim (Mat.Order)	Event Times	Task Num (#Mat.Mul)	Task Order Dim (Mat.Order)
25.5549	225	128	5.2046	100	64
53.2203	441		10.7422	196	
102.0575	900		23.9773	400	
193.2436	1764		47.1808	784	
148.1446	225	256	21.5294	100	128
289.6773	441		41.9395	196	
590.6776	900		74.5879	400	
1157	1764		143.3123	784	
1173.3466	225	512	129.985	100	256
2298.3866	441		254.2516	196	
4690.97	900		518.3303	400	
9205.5	1764		1016.28	784	
9371.7766	225	1024	1033.9166	100	512
18371.2666	441		2027.17	196	
37480.4	900		4136.54	400	
73258.6666	1764		8113.0933	784	
74966.4666	225	2048	8273.1066	100	1024
146156.3333	441		16194.1	196	
285788.3333	900		33041.1	400	
559469.6666	1764		64763.6666	784	

Table 5.7: Device completion times for Mat-Mul kernel, with three CUDA Streams, results are reported for both P100 and M40.

Tesla P100 (56 Streams)			Tesla M40 (24 Streams)		
Event Times	Task Num (#Mat.Mul)	Task Order Dim (Mat.Order)	Event Times	Task Num (#Mat.Mul)	Task Order Dim (Mat.Order)
20.8758	225	128	2.739	100	64
40.5783	441		5.0942	196	
74.6636	900		10.0277	400	
145.4766	1764		19.8252	784	
147.765	225	256	19.3538	100	128
288.5343	441		37.8560	196	
588.9643	900		65.6809	400	
1153.7333	1764		128.317	784	
1173.32	225	512	130.0533	100	256
2298.3966	441		254.281	196	
4690.9633	900		518.615	400	
9202.3333	1764		1016.55	784	
9371.0433	225	1024	1034.0666	100	512
18374.7	441		2027.2866	196	
37474.7	900		4136.7066	400	
73348.3333	1764		8110.9966	784	
74971.6666	225	2048	8262.7533	100	1024
146175.6666	441		16202.4666	196	
285955.6666	900		33059.2666	400	
559425	1764		64786.5333	784	

Table 5.8: Device completion times for Mat-Mul kernel, with as many CUDA Streams as SM amount, results are reported for both P100 and M40.

by a factor 2 (w.r.t. the previous one listed in the table), then the completion time can increase from $\approx 4\times$ to $\approx 7\times$;

- analogously for three-streams, having input tasks sizes each growing by 2, we get that the completion times can vary from $\approx 5\times$ to $\approx 8\times$;
- finally for SM-streams we can see that, the completion time can increase between them from $\approx 7\times$ to $\approx 8\times$.

Those evidences hold for both machines measures and they give some further hints on matrix multiplication nature.

The first point tells us that: the elapsed time to send/receive to/from the device, grows proportionally with the number of tasks, so this parameter would not really affect performances. Especially, for any CUDA streams amount we're using, the increase by 2x of each tasks quantity, will always give a growth of 2x in time measures.

To have a visual comparison, we show plots for completion times in Figures 5.5-5.6, where we can have a graphical view of the completion time variation, as the tasks size grows (respectively on M40 and P100). In Figures 5.7-5.8, instead, we have similar plots, but this time for completion times variations as the number of tasks increments.

The other points, emerging from completion times behavior, tell us that, as task size increases, we'll get worser and worser performances. Clearly this doesn't depend on CPU/GPU data transfers overhead, otherwise we'd have this same behavior when the number of tasks grows too.

So, the cause must reside on what happens inside the GPU. In reality, the classic matrix multiplication is a well known problem in GPUs paradigm and the classical implementation is known as a not efficient[10].

This is because, our trivial implementation, at each iteration, spends more in *global memory/registers* memory operations than in effective calculations. In fact, this kind of matrix multiplication is considered memory-bound²¹.

²¹We also showed that in Section 5.1.5

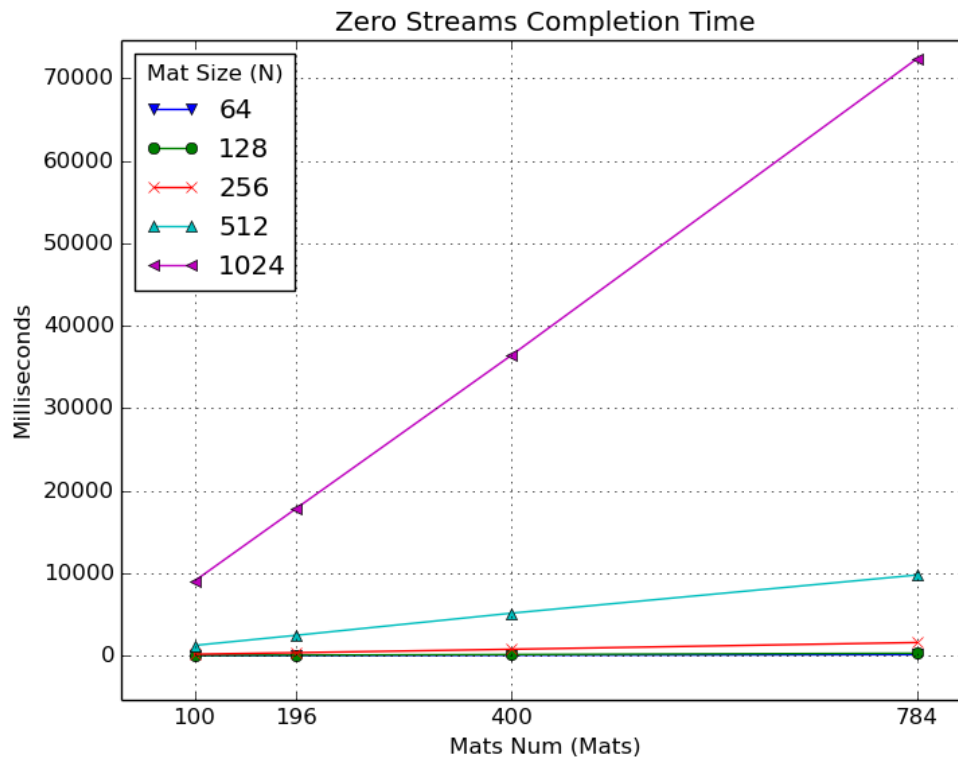


Figure 5.5: Completion Time as the task size (matrix order) changes on M40.

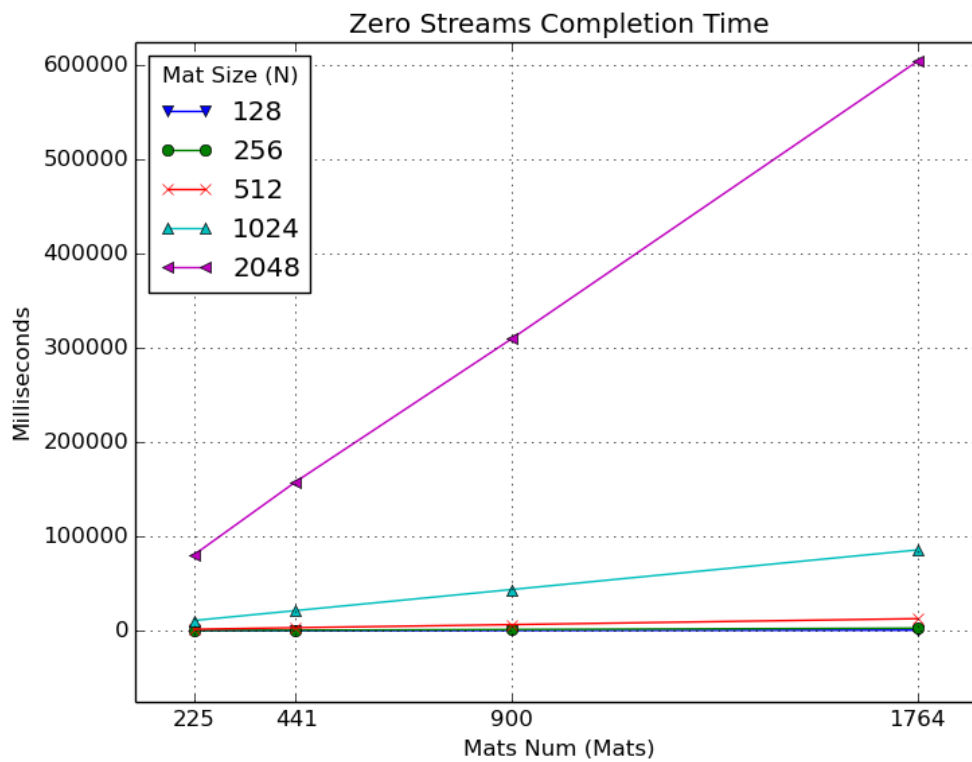


Figure 5.6: Completion Time as the task size (matrix order) changes on P100.

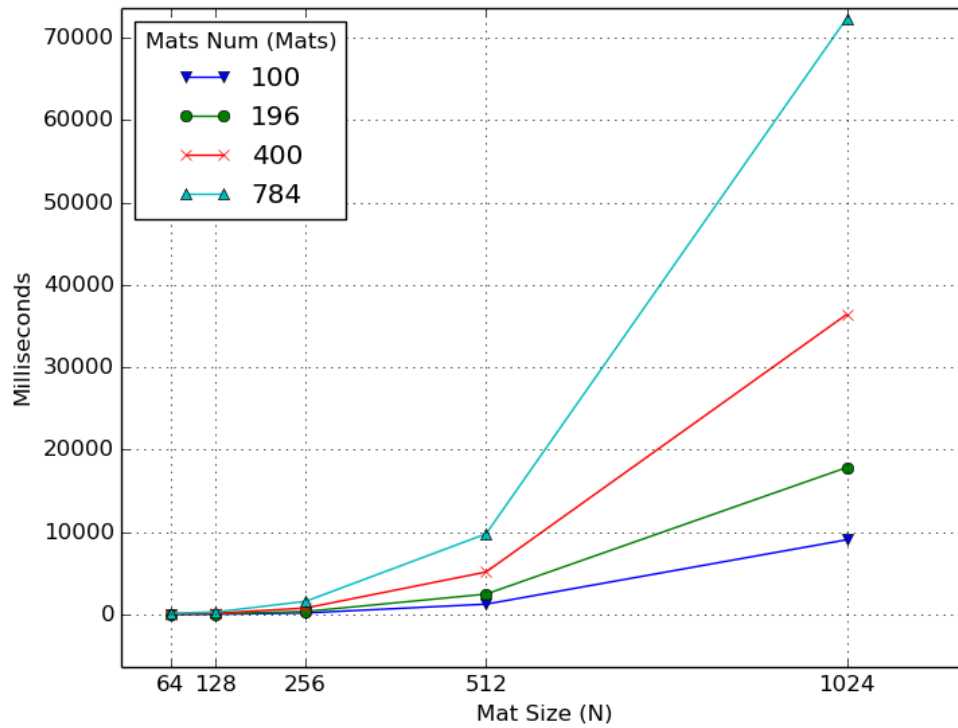


Figure 5.7: Completion Time as the number of tasks (number of matrix multiplications) changes on M40.

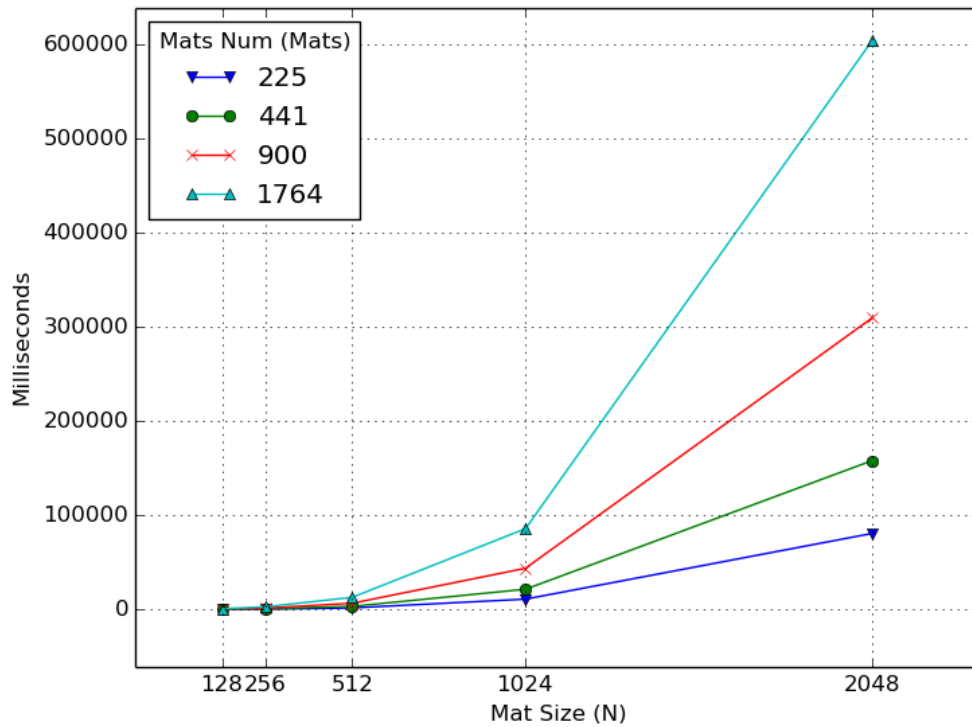


Figure 5.8: Completion Time as the number of tasks (number of matrix multiplications) changes on P100.

So, the more elements matrices have, the more data transfers (internal to the device) the GPU will have to perform and, so, the more active threads will stall waiting for data to be available for computations.

So we'll now focus on speedups, to see that this memory-bound behavior will be negatively reflected on GPU Farm parallel pattern. All speedups are listed in Table 5.9.

From those results, we can mainly observe that:

- $Sp(\mathcal{J})$ gives results near to the expected value, ie ≈ 3 for the smaller tasks sizes (128-256 for the P100, 64 for the M40);
- $Sp(SM)$ gives a really poor, or inexistent, gain especially w.r.t. $Sp(\mathcal{J})$;
- all speedups degrade to ≈ 1 as the task dimension grows.

This behavior translates in the following: when the tasks get bigger, even if CUDA Streams push to have more simultaneous matrix-multiplication, we'll have a lot of active threads (and so Multiprocessors) busy and probably waiting on gathering floats from global memory and instruction dependencies in memory operations.

This, in fact, limits amount performances per se, even when using a lot of CUDA streams. Furthermore, those results tell us that we will fit in Multiprocessor less simultaneous matrix multiplications (kernel executions) than we wish²².

In fact, the necessity of having multiple blocks in the grid²³, translates in a monopolization of SMs resources by a small amount of kernel calls (that moreover performs a poor amount of computations).

Profiling the application for some key data set we can inspect the above described facts and the relative causes. In Figure 5.9 we can have a visual cue on what is hap-

²²Clearly this strictly holds for this particular type of matrix multiplication that we implemented.

²³This is determined by the implementation we gave, see Chapter 4 and it also depends on the specific

Tesla P100 (56 Streams)				Tesla M40 (24 Streams)			
Mat. Number	Mat. Order	Sp(3)	Sp(56)	Mat. Number	Mat. Order	Sp(3)	Sp(24)
225	128	3.3999	4.1620	100	64	3.3598	6.3839
441		3.2917	4.3173	196		3.2467	6.8464
900		3.5271	4.8212	400		2.9599	7.0775
1764		3.7546	4.9874	784		2.9543	7.0309
225	256	2.2548	2.2606	100	128	1.6818	1.8709
441		2.3230	2.3322	196		1.7708	1.9618
900		2.4294	2.4364	400		1.9806	2.2492
1764		2.4447	2.4516	784		2.0873	2.3312
225	512	1.4260	1.4260	100	256	1.4339	1.4331
441		1.4469	1.4469	196		1.4500	1.4498
900		1.4094	1.4094	400		1.5178	1.5170
1764		1.4034	1.4039	784		1.5778	1.5773
225	1024	1.1736	1.1736	100	512	1.2151	1.2150
441		1.1709	1.1706	196		1.2231	1.2230
900		1.1693	1.1695	400		1.2480	1.2480
1764		1.1719	1.1704	784		1.2069	1.2072
225	2048	1.0773	1.0772	100	1024	1.0969	1.0983
441		1.0819	1.0818	196		1.1022	1.1016
900		1.0837	1.0831	400		1.1029	1.1023
1764		1.0801	1.0802	784		1.1178	1.1174

Table 5.9: Here are showed speedups for all data sets of matrix multiplication kernel. Results are reported for both devices.

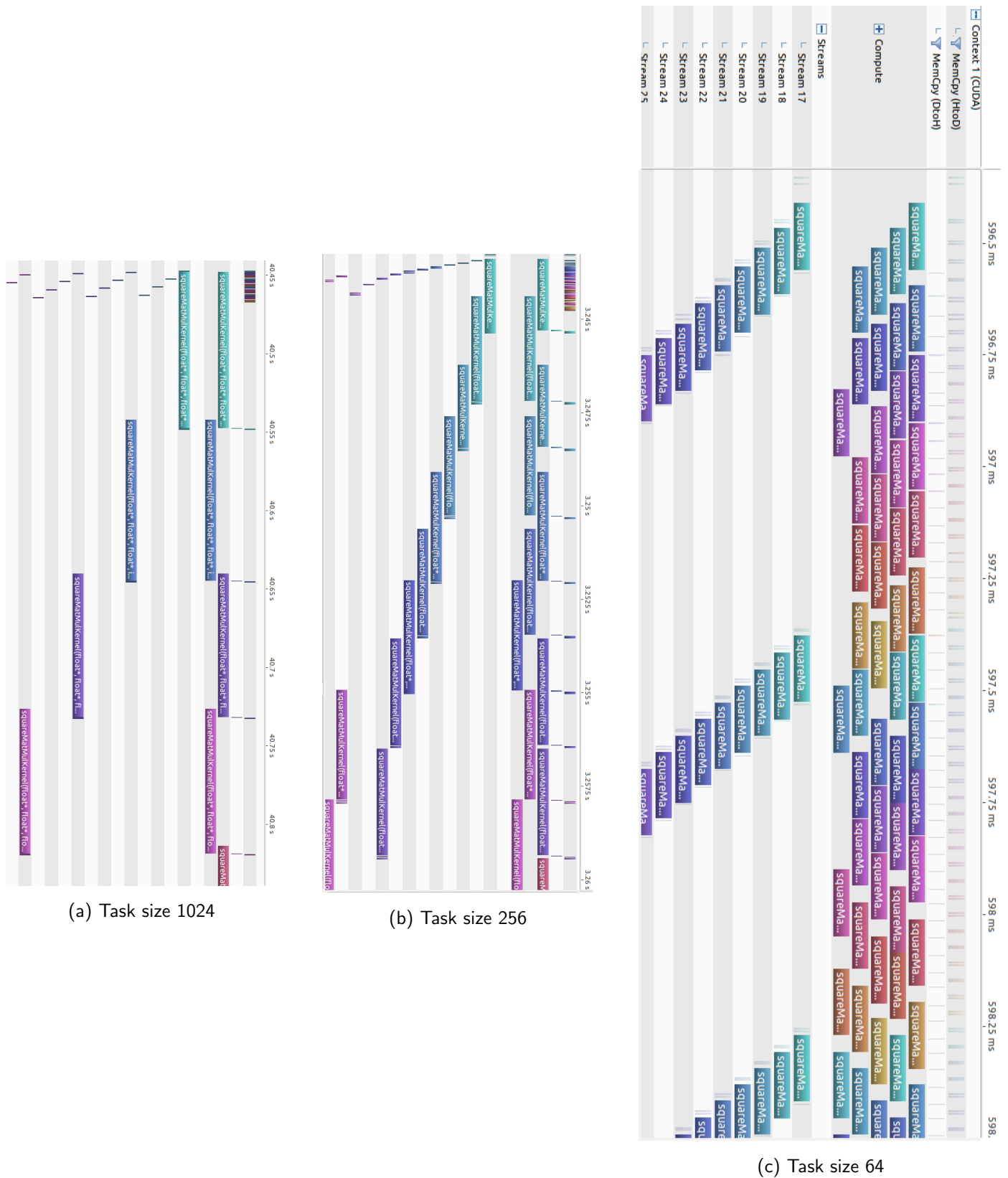
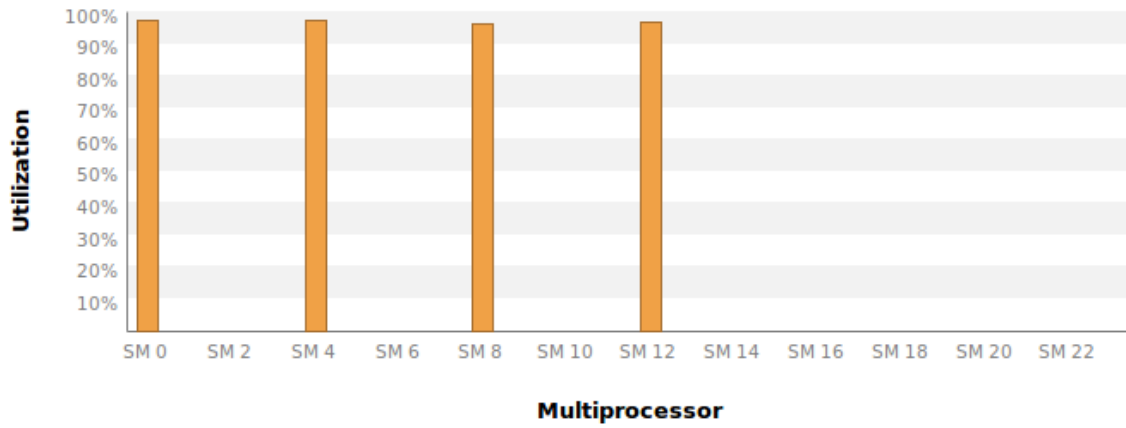


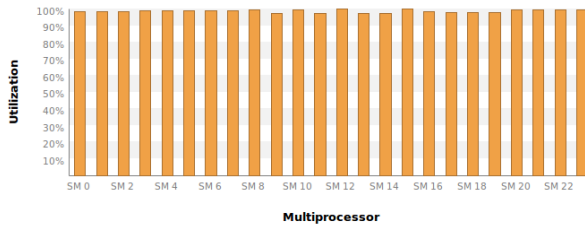
Figure 5.9: NVIDIA Visual profiler generated *timeline* on M40, using 24 CUDA streams, running code for 784 tasks (matrix multiplications) and the different Task sizes (matrix orders) reported in captions above.

i Multiprocessor Utilization

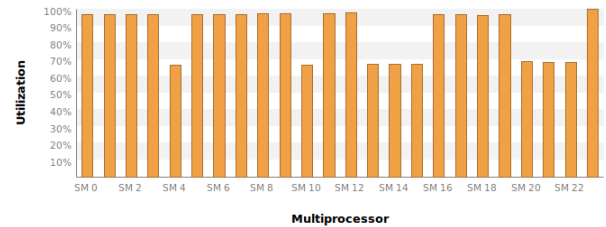
The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.



(a) Task size 64



(b) Task size 1024



(c) Task size 256

Figure 5.10: NVIDIA Visual profiler generated *timeline* on M40, using 24 CUDA streams, running code for 784 matrices and the different Task sizes (matrix orders) reported in captions above.

pening during an execution of matrix-multiplication on M40, with 24 CUDA streams, having an input stream of 784 tasks (i.e. multiplications to perform) of effective size: 64×64 (Figure 5.9c), 256×256 (Figure 5.9b), 1024×1024 (Figure 5.9a).

Analyzing the figure we can see that the amount of overlapping, between operations (kernel overlapping too) in different streams, is really limited in general, this just confirms what we saw from speedups.

From the above mentioned pictures, we can observe that in 64-sized case we're having a slightly better overlapping and a little more kernels running at the same time. In this application we're considering.

case this may happens because grid and block sizes are smaller for each kernel launch, so this allow to have multiple kernels fit in SMs.

Having each thread block such that it contains 32×32 threads, then a 64×64 result matrix (say C) is managed by a grid of 2×2 blocks, while a 256×256 C is managed by a grid of 8×8 blocks and, finally, a 1024×1024 C is managed by a grid of 32×32 blocks²⁴.

This means that, in the two latter cases, we don't even have all blocks, from a single kernel launch, fitting in all of the SMs. If this may theoretically give a full occupancy of the GPU by a certain kernel, from the other side it means saturate the cores without permitting other launches to fit, until the residing kernel ends or, at least, some resources are freed.

We can confirm this fact by looking at the occupancy graphs (generated by Visual Profiler), showed in Figure 5.10. In 64-sized case, we've only 4 SMs almost fully employed, while in the other cases all SMs are almost busy (and this holds on average in a single kernel launch).

The above cases exposes an example of the fact that *not always a high or full occupancy may give better performances*, it strongly depends on the kernel nature[13, 18].

On the other side as we decrease the tasks size we can't exploit enough resources. As we can see from Figure 5.9c we have a better overlapping, but it seems that kernels lasts too little, so the host can't push kernels quickly enough to fill SMs. In fact Visual Profiler suggests us that those kernels perform a really poor amount of computations, especially with respect to memory latencies, see figure Figure 5.11. This pie chart shows that, most of the time, kernels are idle on:

²⁴This depends on how we implemented kernel launch, that is the classical kernel launch approach setting blocks at the maximum size possible. Remember implementations in Chapter 4 and remember the kernel launch configuration explained above.

Sample distribution

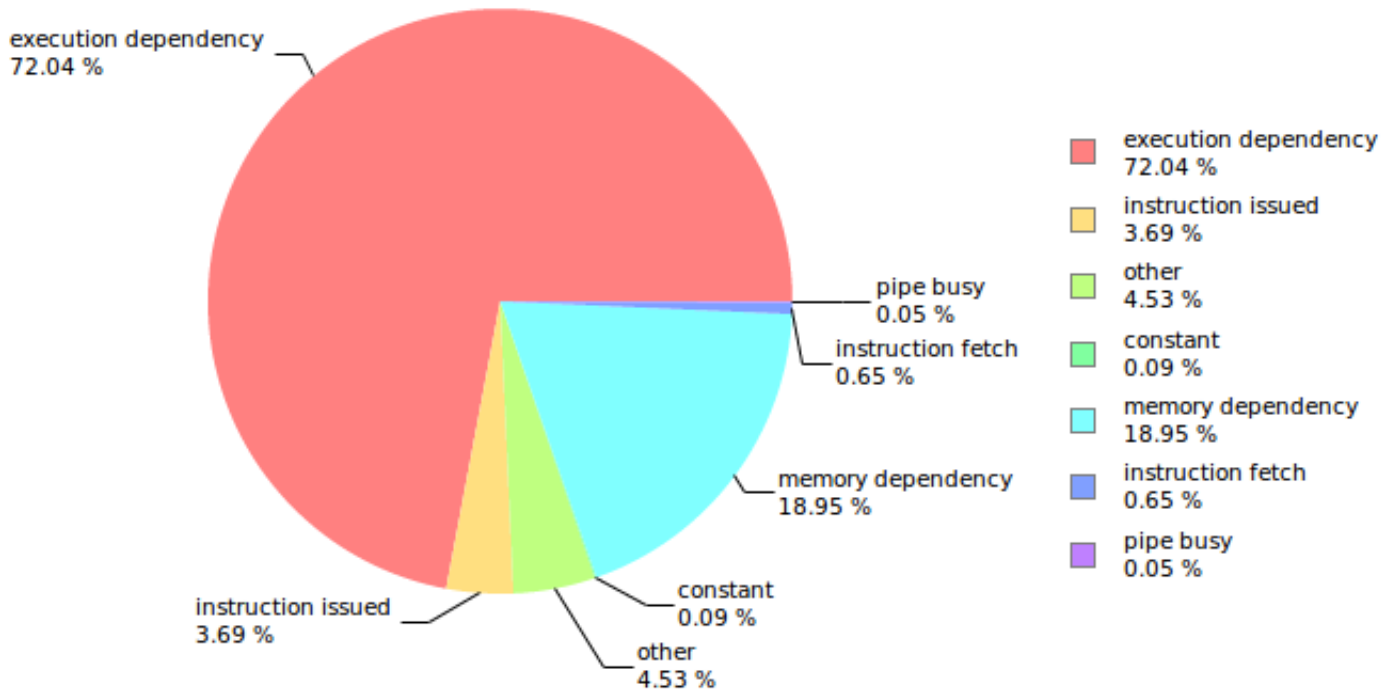


Figure 5.11: Nsight profiler execution on M40, 24 CUDA streams, 784 matrices of size 256. This pie chart gives the types and amounts of latencies inside mat-mul kernel execution.

- execution dependency²⁵, i.e. an input required by the issued instruction isn't yet available;
- memory dependency²⁶, a load/store cannot be made because the required resources aren't available or are fully utilized, or too many requests of a given type are outstanding.

²⁵Execution dependency stall can be potentially reduced by increasing instruction-level parallelism (ILP). We saw in Chapter 3, that we can improve parallelism by increasing the amount of instructions per thread. This can translate in less threads having a greater work-load each. Furthermore we can put independent operations between dependent ones, to cover their latencies.

²⁶Memory dependency stall can potentially be reduced by optimizing memory alignment and access patterns.

This further demonstrates the fact that matrix multiplication is a memory-bound problem in GPU and, so, poor in computation amount.

This is why we have too much short kernels for smaller input data size and heavy kernels for bigger input sizes.

5.4 Image processing

With image processing, i.e. Blur Box algorithm, we're facing a memory-bound kernel and especially rich of divergent execution flows, in fact we made similar tests to the ones for Matrix multiplication.

For each Image Processing we're working on input/output streams of small data parallel tasks, given in this case by small images. So, assume that task size is represented by N , i.e. the order for image resolution, where the latter is given by $N \times N$, then in tests, the kernel execution configuration was set up as follows:

$$\text{BLOCK} = 1024$$

and

$$\text{GRID} = (N/\text{BLOCK}) + \text{BLOCK} - 1$$

so we'll have `blockSize = (BLOCK, 1, 1)` and `gridSize=(GRID, 1, 1)`.

Then we performed the following executions:

1. Classic data parallel approach

If we think to a picture as a matrix of pixels, then it's easy to see the analogy to the previous application. In particular the Data Parallel approach will be tested giving a single image of large dimensions.

Again we're considering a single big image, such that it's comparable with some Stream Parallel versions. Similarly to matrix multiplication case, we remember that in data parallel version we have a data parallel big structure, it will be sent to the GPU, where data parallel computations will be performed all over the data

Tesla P100 & Tesla M40	
Task Size (Img.Order)	Task Num (Img.Number)
128	64
256	256
512	1024
Tot. work load	
1024	
2048	
4096	
8192	

Table 5.10: Input dataset for Image Processing kernel. Above Stream parallel configuration, below the relative Data Parallel.

structure elements.

This is totally different from stream parallel case, where we have a stream of small data parallel tasks, in this case given by small images. However we can find some interest sizes for data parallel structures, such that they allow to perform the same amount of work with respect to the stream parallel execution. In the lower part of Table 5.10 we show the values for some overall work loads computed from an input stream, we use these values only to make a fair comparison with the different data parallel version.

Clearly, again we've an input stream of small data parallel tasks (theoretically in unknown quantity) that is totally different from having a big data parallel data structure of known dimension. So, we're only comparing two different approaches for the same application and, in order to give a fair comparison, we execute the two versions for the same overall amount of work.

2. Stream parallel

As previous cases of Farm parallel pattern, we have to determine the tasks number arriving from the input stream (we recall that stream items are small data parallel tasks, that are made up by small images).

In the upper portion of Table 5.10 are reported different amount of tasks and, for each of them, we test the three types of tasks size (resolution of the small images). For every combination given by the variation on input parameters, we'll test for different numbers of CUDA streams: **Zero**, **Three** and N_{SM} CUDA Streams (with $N_{SM} = \#Streaming\ Multiprocessors$). The above test on different numbers of non-default streams, is implemented in a totally analogous way to the ones for the two applications showed above.

5.4.1 Results

As expected, this image processing kernel demonstrates a bad fitting for Farm parallel pattern in GPU. It gives even worse performances than matrix multiplication.

We report in Table 5.11 all completion times, for zero-streams, three-streams and SM-streams versions.

We can see that, fixed a task size, the input stream length (i.e. total number of tasks to compute) is a parameter tested for values each growing by a factor 4 (w.r.t. to the previous chosen one), in fact completion times follow this trend by increasing each $\approx 4\times$, i.e. measures grow proportionally with number of tasks arriving from the input stream. Instead, fixed a certain number of total tasks arriving from the input stream, we vary the task size to experiment dimensions that double from one to another. This makes the respective completion times growing by a slightly bigger factor, i.e. $\approx 3\times$.

But the really evident and important behavior is that we don't have much difference between the version not using CUDA Streams and the ones using them.

This is, in fact, confirmed by the speedups in Table 5.12, where we can see that almost everywhere the best speedup we can achieve is about 1, that means no speedup at all.

CUDAStreams	Tasks Num.	Tasks Size	Tesla M40	Tesla P100
0	64	128	1608.7367	1425.9833
	256		6153.5800	5670.5567
	1024		25422.7333	22774.7333
	64	256	5099.54	4648.4467
	256		20633.2	18077.1667
	1024		81256	73411.1333
	64	512	15652.6333	14281.8
	256		63566.5333	56694.1333
	1024		255928.3333	224584.6667
3	64	128	1547.5200	963.1717
	256		5477.0967	3851.38
	1024		21754.4667	15387
	64	256	4557.1400	3933.31
	256		17582.4	15724.8
	1024		71291.5	60029.0667
	64	512	14315.9333	11833.2
	256		56507.8667	49691.8
	1024		224832.3333	197940.3333
24 - 56	64	128	1482.0533	1043.4733
	256		5651.3833	4178.6333
	1024		21790.8333	16213.8667
	64	256	4461.85	3766.4833
	256		17751.7667	15682.6667
	1024		70421.1	63726.1667
	64	512	13900.0333	12902.0667
	256		54006.6667	51737.7333
	1024		226638	208308.3333

Table 5.11: Device completion times for Image processing kernel, all types of tested CUDA Streams number are reported, results are given for both P100 and M40.

		Tesla M40 (24 Streams)		Tesla P100 (56 Streams)	
Tasks Num.	Task Size	Sp(3)	Sp(24)	Sp(3)	Sp(56)
64	128	1.0396	1.0855	1.4805	1.3666
256		1.1235	1.0889	1.4723	1.3570
1024		1.1686	1.1667	1.4801	1.4046
64	256	1.1190	1.1429	1.1818	1.2342
256		1.1735	1.1623	1.1496	1.1527
1024		1.1398	1.1539	1.2229	1.1520
64	512	1.0934	1.1261	1.2069	1.1069
256		1.1249	1.1770	1.1409	1.0958
1024		1.1383	1.1292	1.1346	1.0781

Table 5.12: Here we showed speedups for all data sets of image processing kernel. Results are reported for both devices.

So, we almost have no overlapping, we expected a similar behavior though.

5.5 Results Summary

Merging all obtained results, we can state that, under specific assumptions and adjustments, *Farm parallel pattern can give a speedup really near to linear one.*

We can observe this behavior especially in computation-bound case, where we're below the ideal speedup for a small amount. Anyway, we can't achieve perfectly the ideal for more possible reasons:

- First we've to take into account that we have a "*stabilization phase*", meaning that when input stream starts to send first items we've a little interval where CUDA streams and SMs need to fill up, until we reach the peak occupation;
- We can have the rare situation where in a certain portion of time, say $[t_i, t_i + \Delta_t]$,

we have multiple requests (from different CUDA Streams) for memory copy, i.e. we've $> \text{number of Copy Engines}(= 2)$ simultaneous requests;

- We can have cases such that we can't reach the best possible overlapping²⁷

However, we only proved theoretically that $Sp_{\#SM} \leq \#SM$, just for completeness we could show some tests for a greater number of CUDA Streams, for example

$$\#CUDAStreams \geq 2 \cdot \#SM.$$

We briefly show the result for Simple-computational kernel, comparing the SM-version and the double-SM-version²⁸. So this means we're trying to use an amount of CUDA streams that is the double of SMs number, just to see that anyway we can't achieve a speedup such that $Sp_{\#SM} > \#SM$.

And, as we expected, the Table 5.13 confirms the theoretical upper bound determined with Amdahl's law. The table clearly shows that the speedup is still $Sp_{\#SM} \leq \#SM$. Furthermore, it seems we're having speedups even lower than the ones for SM-version and it may be due to the overhead caused by the management of such a huge quantity of CUDA streams.

5.5.1 Stream parallel compared to Data parallel

As introduced on the tests settings, we executed and collected completion time measures for data parallel version too. The latter clearly is setup in such a way that it computes the same workload that we're computing in stream version.

But we enforce again the concept that we're comparing the same application from two distinct perspectives of two totally different parallel patterns.

In fact, we have to remember that pure data parallel version will take a single data structure, having known size both theoretically and practically, and over all the content of this collection we perform data parallel computations, so the result will be again

²⁷Since the behavior of commands issued by different CUDA streams is undefined, we can't be sure to always have the best overlapping.

²⁸We only report results for M40 machine, but performances are analogous on P100.

N items	M iterations	Comp.Time (48 Streams)	Sp(48)
245	10000	35.6340	19.4686
	400000	1204.41	22.7945
	800000	2379.3233	23.0691
49152	10000	70.2144	19.6877
	400000	2380.3433	23.0645
	800000	4734.9566	23.1857
98304	10000	135.2063	20.4526
	400000	4741.3866	23.1576
	800000	9537.72	23.0194
196608	10000	282.214	19.5913
	400000	9893.6566	22.1949
	800000	19760.9333	22.2204
393216	10000	519.2003	21.2989
	400000	18951.0666	23.1750
	800000	38050.7333	23.0795
786432	10000	969.7603	22.8021
	400000	37473.3333	23.4415
	800000	74944.7333	23.4355

Table 5.13: Here are showed completion time and speedup for $48 = 2 \cdot \#SM$ CUDA Streams.

a big data structure having the same size as the input.

The Farm version, instead, takes a stream of small tasks that are sent to the GPU to perform data parallel computations on each of those small items and, as output, we'll have back the same amount of tasks, having the same size as input each.

So we're not clearly comparing directly input types and dimensions, but we're comparing a certain application for the overall amount of work given by the two different versions.

Below we'll show results for each kernel type:

- **Simple-computation kernel**

From the Table 5.14 we can clearly see that data parallel has really similar performances with respect to stream parallel version (with as many CUDA streams as SMs number).

This is what we expected, since, the almost-linear speedup we obtained, means that we successfully overlapped different executions for small tasks. Formally, suppose that for any data/stream parallel comparison, we consider the same application and the same overall work load.

Call $T_{DataPar}$ the amount of time needed to: copy the whole single data structure from host to device, perform data parallel computations on the whole data structure, then copy back the all the result data structure. For stream parallel version we'll wish to have instead

$$T_{StreamPar} \approx \Delta_t + T_{DataPar} \quad (5.12)$$

where Δ_t is a not predictable overhead, given by the different behavior on GPU of stream parallel w.r.t. data parallel.

In this Δ_t we may also think to include those overheads due to imperfect streams overlapping, i.e. host/device copy or kernel execution that couldn't hide with other activities (partially or totally). So, in those cases where Δ_t can be negligible

		Tesla P100 (56 Streams)		Tesla M40 (24 Streams)	
N items	M iterations	56 Streams	Data Parallel	24 Streams	Data Parallel
57344	10000	104.7727	51.3901	30.6074	34.9234
	400000	3968.9133	1825.36	1178.72	1178.71
	800000	7818.8433	3648.6333	2355.4133	2352.63
114688	10000	205.7297	90.2111	60.2087	58.0608
	400000	7828.1933	3574.74	2358.6567	2303.43
	800000	15691.8333	7145.6633	4714.36	4601.0867
229376	10000	407.712	179.2643	119.3447	112.1287
	400000	15687.9667	7124.46	4715.1233	4456.6633
	800000	31396	24437.4333	9425.92	8910.3567
458752	10000	803.2237	669.5133	238.249	241.901
	400000	31422.0333	26757.7	9429.89	9627.5267
	800000	62818.7	52896.3667	18853.9667	19256.4333
917504	10000	1619.5867	1361.9133	475.5907	441.5377
	400000	62793.4	53258.9667	18856.8	17591.4667
	800000	125575	105657	37705.6667	35190.9
1835008	10000	3229.0633	2682.6767	949.497	884.6433
	400000	125547.6667	105891	37711.9	35013.4667
	800000	251503	209018.6667	75445.2667	70397.2

Table 5.14: Simple-computational kernel. Comparison between completion times for stream parallel (max stream -56 and 24 respectively-) and data parallel versions. Results are reported for both devices.

with respect to the overall completion time, we hope to achieve:

$$T_{DP} \approx T_{SP} \quad (5.13)$$

where in general we expect that it's more likely to have $T_{DP} \leq T_{SP}$, than the contrary.

- **Matrix Multiplication kernel**

In M40 Table 5.16 and P100 Table 5.15 we can see a really particular behavior, because on both devices we get consistently better performances from streaming parallel version, with respect to the data parallel version.

Tasks Num.	Tasks Size	56 Streams	Event Time	Data Par. size
225	128	20.8758	289.4027	1920
441		40.5783	898.0440	2688
900		74.6636	2256.0733	3840
1764		145.4767	7063.8233	5376
225	256	147.7650	2256.0733	3840
441		288.5343	7063.8233	5376
900		588.9643	17878.3667	7680
1764		1153.7333	56190.3667	10752
225	512	1173.3200	17878.3667	7680
441		2298.3967	56190.3667	10752

Table 5.15: Matrix multiplication kernel. Comparison between completion times for stream parallel (max stream -56) and data parallel versions (Partial dataset of stream version is considered). Results are reported for P100.

For Completeness in M40 Table 5.16 we also introduced a column to compare the data parallel version with serial version (streaming parallel with zero CUDA

Streams).

From that column we can see that Data parallel version also has worse performances than serial version. Furthermore we introduced a column to compute the ratio $\frac{T_{dataParallel}}{T_{\#SMstreams}}$, and as we can see from Stream parallel version, we have a gain of $\approx 20\times$.

Mat Number	Mat Order	Zero Streams	24 Streams	Data Par Time	Mat Order Data Par	(DataPar)/(24Str)
100	128	36.2095	19.3538	172.6997	1280	8.9233
196		74.2686	37.8561	456.3713	1792	12.0554
400		147.7337	65.6809	1315.6467	2560	20.0309
784		299.1380	128.3170	3590.4367	3584	27.9810
100	256	186.3913	130.0533	1315.6467	2560	10.1162
196		368.6813	254.2810	3590.4367	3584	14.1200
400		786.7537	518.6150	10437.7	5120	20.1261
784		1603.4933	1016.55	28711	7168	28.2436
100	512	1256.4	1034.0667	10437.7	5120	10.0938
196		2479.4333	2027.2867	28711	7168	14.1623

Table 5.16: Matrix multiplication kernel. Comparison between completion times for stream parallel (max stream -24-) and data parallel versions (Partial dataset of stream version is considered). Results are reported for M40.

This singular behavior again depends on the memory-bound nature of this implementation.

Having a huge single data structure, computing a poor amount of arithmetic on a huge amount of items residing in global memory, inevitably leads to a drop in performances, mainly due to latencies and thread stalls.

Recalling the implementation we studied for matrix multiplication, have a big data structure means: each thread couple computes a single element for result

		Tesla M40 (24 Streams)		Tesla P100 (56 Streams)		
Tasks Num.	Task Size	Streams 56	Data Par time	Streams 24	Data Par time	DataPar Size
64	128	1043.4733	1042.1233	1482.0533	1928.8767	1024
256		4178.6333	2522.0767	5651.3833	7369.49	2048
1024		16213.8667	25430.7667	21790.8333	29222.2	4096
64	256	3766.4833	2522.0767	4461.85	7369.49	2048
256		15682.6667	25430.7667	17751.7667	29222.2	4096
1024		63726.1667	46776.4667	70421.1	50971.8333	8192
64	512	12902.0667	25430.7667	13900.0333	29222.2	4096
256		51737.7333	46776.4667	54006.6667	50971.8333	8192

Table 5.17: Here is showed the data parallel and stream parallel comparison for image processing kernel. Results are reported for both devices.

matrix and since we're working on high order matrices, for each $C[i, j]$ we'll have a high number of sums of products but especially a lot of double loads (for $A[i, k]$ and $B[k, j]$). This inevitably will give heavy and slow blocks of threads, monopolizing SMs resources.

Furthermore, at any given time, we'll have only part of the total computations fitting in all SMs. So having such heavy blocks, in such a huge quantity, contributes to drop the performances, especially w.r.t. the stream parallel version.

- **Image Processing kernel**

In this case, looking at Table 5.17, we can observe a fluctuation in behavior.

Sometimes Data Parallel performs better than Stream Parallel, and sometimes the vice versa happens. But we recall that this type of kernel is peculiarly chosen to have divergent flows. Indeed, even the profiler reported issues and latencies due to diverging flows.

CHAPTER 6

Conclusions

The main goal of this thesis was to experiment if a Farm parallel pattern could fit in GPU architecture and, if this was the case, how.

Even though a Streaming parallel pattern may seem so far from the classic GPU use, we founded our attempt on the increasing and pervasive concept of *General-Purpose computing*. Nowadays it's a common practice to use high parallelism and huge computational power of GPUs as co-processors, even if it isn't strictly for graphical problems. Also the research moved, in last years, the focus on problems that generally are assigned to CPUs. Clearly, in General Purpose (GP) it's easy to spot applications that are clearly embarrassingly parallel; we recall that GPUs are known to be mostly suited for data parallel approaches.

However, there are many others problems that are really far from data parallel behavior and in some of those cases GP-GPUs demonstrate a fair behavior (generally with some proper adjustments).

So it makes perfectly sense to inspect for new non-data parallel applications to fit in

GPU model. The main goal, in general, is to exploit the high computation potential of GPUs.

6.1.2 Evaluation of the problem

The starting point of this study was to consider and understand some main features and the functioning of a graphic processor, in particular taking into account of the organization about parallelism, threads, cores, internal memory and so on. We showed main GPUs and NVIDIA CUDA characteristics, briefly introducing them in Chapter 1 and deepening on more specific concepts in Chapter 2 and Chapter 3.

In the latter we also showed how some best practices and considerations were exploited to evaluate and then design our model.

Once we had an overall view on tools and NVIDIA GPUs architecture, we had a base knowledge for the next step, i.e. to design a *Farm parallel pattern* for a graphic processor. Obviously some key problems have arisen:

- Handle the difference on input/output with respect to canonical data parallel problems, i.e. dealing with streams of data parallel tasks, instead of a single and purely data parallel structure;
- Manage the way we send data to device;
- Experiment several types of executions, each for different sizes of small data parallel tasks (from farm input stream);
- How to hide the overhead due to data transfers between host and device;
- How to execute many "small" kernels at the same time, instead of a single "big" one;
- How to exploit the resources of the GPU at their best, taking into account the studied application nature.

The first two points were accomplished by thinking to a *particular way* to use CUDA streams and asynchronous host/device memory copies.

In other words, we decided to use many non-default streams, up to an high amount compared to the usual way CUDA streams are used. This hopefully should have allowed us to hide data transfers and to be able to run, at the same time, a lot of "small" data parallel tasks on the several GPU multiprocessors.

Clearly we could expect to have different kernels running in parallel on the SMs, since tasks are arriving from a Farm stream, are supposed to be reasonably small to occupy a part of resources, without saturating too much SMs. So this should allow more small tasks to fit in multiprocessors together.

The third point was mainly linked to the concept of *occupancy* evaluations. By both *empirical approach* and a study on *best practices* for GPUs, as we showed in Chapter 3, we get interesting results and we could experiment where occupancy was a benefit. However, we also saw how occupancy may not be a relevant factor; a lot of performances bottlenecks may depend on the kernel nature. We have to face some **latencies** that happens inside the Streaming Multiprocessors, in our study we mainly pointed out two causes of bottleneck in kernels: **memory-bounded** code and **diverging flows**.

Those concepts are straightly linked to the problem of last point in the above list.

The fourth and fifth points are again strongly related to a powerful programming technique in CUDA: ***Asynchronous calls*** and ***CUDA Streams***.

We recall that here asynchronous is from a host side point of view, with respect to the device. That is, host can continue executing his code, after invoking a memory copy (or any other call that is generally synchronous). Any asynchronous call will be forwarded to GPU that will "silently" work, sending back eventual results to CPU, that hasn't to wait for device calls to end.

We pointed out that often asynchronous calls require to have some synchronization point. Sometimes CUDA codes with asynchronous calls are implemented introducing

explicit synchronizations, in order to have correct results and avoid memory overwriting. We also met that problem, having a lot of CUDA streams trying to write back results, possibly at the same time, this sometimes led to overwriting data among the non-default streams¹.

This problem mainly showed up in device memory: we needed sufficient GPU memory locations for all streams.

In device side, the possible solutions are, in general, to:

- Have a reduced amount of space in global memory and use some explicit synchronization and that's the most used approach;
- Reserve enough memory to support data transfers for each single CUDA streams and, so, avoid to use explicit synchronization.

The first approach can be used in problems that are more suitable for GPUs, as probably it introduces a negligible amount of overhead, but in a stream parallel context it can cause a slowdown due to synchronization, leading to a performance drop. That's why we decided to use the second approach. Synchronization, with such a particular use of CUDA streams, can give a bottleneck, slowing down the amount of tasks we're capable to send to SMs at any time.

The first impression could be to risk for a saturation in global memory, but in our case it didn't happen, since we were using relatively small tasks of data parallel data (even if the reserved space for tasks has to be as much as number of CUDA streams, i.e. SMs number).

This doesn't mean that it cannot exist any stream parallel application where synchronization may not be a disadvantage².

¹We recall that operations, issued inside a CUDA stream, are executed serially. For commands, belonging to different CUDA streams, instead, we can have an undefined behavior, i.e. we don't have any guarantee on the execution order.

²Maybe to hide some other computations that are happening at the same time. Again it's a matter of experimenting according to the type of problem we're facing.

Furthermore trying a *hybrid approach* could be a starting point for future works. Here hybrid means having less allocated global memory locations (than CUDA Streams number) and introduce only few explicit synchronizations.

6.1.3 Implementation and tests

Those ideas and designs were implemented as described in Chapter 4. We decided to implement different kind of kernels to experiment the behavior of Farm parallel pattern in different conditions.

We recall that we decided to implement three types of kernels: Simple-computational, Matrix Multiplication and Image processing.

The first would have been the one from which we expected good performances, while we expected worse completion times and speedups from the second kernel type and even worse from the third one.

The next step has been the tests building and results gathering, so that we could extract useful considerations.

Tests have been set up in such a way to observe the performances of our model in different situations, such as varying the tasks size and varying the pressure on CUDA Streams, i.e. the number of these tasks sent on a certain non-default stream and so the number of tasks that a certain SM has to execute.

What we wanted to mainly measure was:

- The global time spent to "consume" an input stream by transferring data one task per time, do all computations of the kernel and send back results. This is what we considered the *serial version* for our applications, in other words the approach without CUDA Streams;
- The global time spent to "consume" an input stream by overlapping more data transfers and/or more kernel executions. This is what we considered the *parallel*

version, in other words the approach using CUDA Streams (three was the base case, having as many CUDA streams as SMs was the special case);

- The completion time of the relative data parallel version, i.e. assuming that the input is given by a single data structure, it should anyway compute an amount of work equal to the overall work computed for the whole Farm tasks input stream.

The latter point means only that we're doing a fair comparison, but the nature of the two problems and input/output data is completely different³.

6.1.4 Results and considerations

The results, in Farm parallel pattern for GPU, using CUDA streams, gave just what we expected:

1. Simple-computational kernel showed a good overlapping, especially among kernels, clearly with some appropriate adjustments. We got the expected speedups and the version with the maximum number of CUDA Streams⁴ performs almost as the data parallel version;
2. Matrix multiplication kernel showed a low overlapping behavior, especially as matrices size increased. We got poor speedups, however the version with CUDA Streams often performed better with respect to the data parallel version;
3. Image processing kernel showed an almost inexistent overlapping behavior. We get no speedups and the version with the maximum number of CUDA Streams performed sometimes better sometimes worse than the data parallel version.

From those results we understood that we have the best gain when we have long computations on each single task.

³Farm has discussed and compared with Map mainly in Chapter 3

⁴That is the version with an amount of non-default streams equal to the number of Streaming Multi-processors.

In fact, as we showed in chapter 5, the first type of kernel has a high computation intensity, while the others two are memory bound and, so, provide a counterexample for Farm on GPU.

Even if host/device data transfers introduce a not-negligible overhead in Farm pattern, in any case, we're carrying small data parallel items.

Furthermore, these small tasks aren't available all at the same time, they arrive one at time, as they're generated from an input stream. This potentially leads to a low overlap among various data transfers, but it's just for a timing matter.

That's why we should mostly rely on overlapping kernels, as they should last longer than memory copy⁵ and so we've more chances to achieve an overlap.

About this consideration we recall Figure 5.3.

That's why most of the problems in performances appear in memory-bound kernels, because we have a lot of memory operations, merged with a really small amount of work per thread to do. This leads to inefficient kernels, that, in any case, last too short to afford a good overlap with other kernels or transfers.

Furthermore, we saw that this behavior in memory-bound can even degrade as the tasks, sent to the GPU, grow in size. It's like what happened in our Matrix Multiplication case, because we had a high number of thread blocks occupying all hardware resources, for a single matrix multiplication. Anyway, in this application, having longer kernels lead to completely monopolization of Multiprocessors, without permitting the fitting of other kernels in SMs.

⁵This isn't a rule, it just often happens, as in our applications. There may still be cases in which this statement is false.

6.1.5 Final remarks and further works

The results and considerations just discussed in previous section, expose the following necessities for Farm parallel pattern:

- It better performs in high-computation intensity scenarios;
- We get the best advantages from parallelizing executions, more than host/device memory copies;
- It relies on overlapping between CUDA streams, meaning it needs quite long kernels executions to hide the host latency deriving from the acquisition of items from an input stream;
- Kernel launches should be configured such that they don't monopolize many multiprocessors (the best would be at most one SM occupied by a single kernel call).

The second and third points, again underline the necessity of having a small amount of memory traffic with respect to computations.

Maybe it would be a good compromise even to have only a slightly higher amount of computations w.r.t. memory operations. This means that maybe we can observe acceptable performances, from GPU Farm, even in an average cases.

In this work, in fact, we considered application having an opposite behavior.

In terms of arithmetic intensity and Roofline model this concept would be expressed by $I \approx \pi/\beta$, meaning that we're working on a kernel having a behavior in the middle of computation and memory-bound.

So one of the future works, for this field of study, could be to do experiments for some "average-behavior" applications. For example, it could be interesting to study an optimized implementation for matrix multiplication, there are many proposed versions in CUDA guides too[18, 10].

Investigate in this direction could be a turning point to prove that Farm parallel pattern

for GPU behaves fairly not only in completely compute-bound applications.

However the above listed requirements may translate in a quite challenging effort, especially in evaluating problems, experimenting and profiling performances, more than in implementation difficulty.

As we said before, this especially happens in all those cases where we have memory-bound problems. Anyway, some future workarounds to improve performances may be tried, even in apparently not suitable applications, in particular we can get advantages from:

- Using efficient memory access patterns for GPU memory (especially needed for global);
- Assigning more work to each thread, for example giving more instructions to execute per kernel (Instruction Level Parallelism) and alternate dependent instruction with independent ones[18, 13];
- exploiting shared memory, it has smaller dimensions but it's much more faster than global memory[10];
- changing the scheduling policy of input tasks among the CUDA Streams.

These stratagems, in the future, may expand to a lot of further applications using Farm parallel pattern on GPUs.

As an example, numerous studies have been made on matrix multiplication to optimize device global memory latencies with shared memory[10, 33]. Other studies showed that we can give smaller kernel configurations, in order to make each thread performing several matrix multiplications[13, 33], instead of computing only one element of the result matrix for each couple of threads (as it happens in our simple and inefficient application).

So, merging those optimizations with Farm parallel pattern may give, in future, some interesting results.

Given that this thesis based all the hypothesis on equally sized tasks from an input stream, during a certain Farm execution, we worked only on balanced workloads for each kernel. However, another interesting further study could be done in those scenarios treating unbalanced tasks (possibly still data parallel tasks), leading again to a stream of data parallel tasks, but giving different workloads among the respective kernel launches.

Suppose, for example, a scenario where the input stream sends items at fluctuating speeds, so the task size may be established according to a time interval, instead of a predefined buffer size. This means to send portions of items of unknown size to the GPU.

Another assumption on which we based all this study was in having a certain type of scheduling.

In particular, we scheduled small data parallel task towards CUDA Streams (tasks arriving from the input stream) with a Round Robin policy.

In the problems treated in this thesis, or in other Farm applications where we're trying to achieve a performance improvement (memory-bound ones for example), maybe other scheduling techniques can be adopted.

This potentially may result in a more efficient spreading of tasks between CUDA streams. So, if this gives a better exploitation in Multiprocessors resources or a better work distribution, this may give a further performance improvement.

The above mentioned suggestions and improvements, coupled with the huge diffusion of Stream parallel applications, maybe a shift in GP-GPUs field expansion.

Listings

2.1	CUDA Strams creation	26
2.2	CUDA Strams and Async example	27
2.3	Tests on bash scripts example	32
2.4	Portion of speedup and plots Python script	33
4.1	Implementation for Simple-Computation Kernel	64
4.2	Implementation for Matrix Multiplication Kernel, both non-square and square	65
4.3	Implementation for Image processing Kernel (Blur Box Algorithm) . .	67
4.4	Simple-computational Kernel Launch configuration, i.e. Grid and Block dimensions setting	70
4.5	Data transfer host/device and kernel call, synchronous version	71
4.6	Data transfer host/device and kernel call, CUDA Streams version . . .	72
4.7	Host side pseudo-code: input stream & kernel call function(scheduler). .	73
4.8	Optimal Kernel caller for Simple-Computation kernel, uses Occupancy APIs to get best Block configuration	75
5.1	Example of code for events utilization as time probes.	79

List of Tables

2.1	GPUs specifics for the two remote machines employed in this project. .	15
5.1	Input dataset for Simple-Computation kernel, these are the input stream limits for both devices.	94
5.2	Device completion times for Simple-computation kernel, without using CUDA Streams, results are reported for both machines (P100 and M40).	98
5.3	Device completion times for Simple-computation kernel, using as many CUDA Streams as SM number, results are reported for both machines (P100 and M40).	99
5.4	Speedups for all data sets of simple-computation kernel. Results are reported for both devices.	101
5.5	Input dataset for Matrix Multiplication kernel. Above Stream parallel configuration (task size and task amount), below the total work load for the overall executiont.	106
5.6	Device completion times for Mat-Mul kernel, without using CUDA Streams (zero non-default streams), results are reported for both P100 and M40.	109

5.7	Device completion times for Mat-Mul kernel, with three CUDA Streams, results are reported for both P100 and M40.	110
5.8	Device completion times for Mat-Mul kernel, with as many CUDA Streams as SM amount, results are reported for both P100 and M40.	111
5.9	Here are showed speedups for all data sets of matrix multiplication kernel. Results are reported for both devices.	116
5.10	Input dataset for Image Processing kernel. Above Stream parallel configuration, below the relative Data Parallel.	122
5.11	Device completion times for Image processing kernel, all types of tested CUDA Streams number are reported, results are given for both P100 and M40.	124
5.12	Here we showed speedups for all data sets of image processing kernel. Results are reported for both devices.	125
5.13	Here are showed completion time and speedup for $48 = 2 \cdot \#SM$ CUDA Streams.	127
5.14	Simple-computational kernel. Comparison between completion times for stream parallel (max stream -56 and 24 respectively-) and data parallel versions. Results are reported for both devices.	129
5.15	Matrix multiplication kernel. Comparison between completion times for stream parallel (max stream -56) and data parallel versions (Partial dataset of stream version is considered). Results are reported for P100.	130
5.16	Matrix multiplication kernel. Comparison between completion times for stream parallel (max stream -24-) and data parallel versions (Partial dataset of stream version is considered). Results are reported for M40.	131
5.17	Here is showed the data parallel and stream parallel comparison for image processing kernel. Results are reported for both devices.	132

Bibliography

- [1] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design: The Hardware and Software Interface*, RISC-V Edition, Morgan Kaufmann, 2014
- [2] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, Jack Dongarr, *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming*, University of Tennessee Knoxville, University of Manchester,
<http://www.netlib.org/lapack/lawnspdf/lawn228.pdf> , 2010
- [3] John Jenkins, Isha Arkatkar, John D. Owens, Alok Choudhary, Nagiza F. Samatova, *Lessons Learned from Exploring the Backtracking Paradigm on the GPU*, 2011
- [4] Microsoft, *Graphics Pipeline*, Documentation guide:
<https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline?redirectedfrom=MSDN>, 2018
- [5] Hujun Bao, Wei Hua, *Real-Time Graphics Rendering Engine*, 2011

- [6] NVIDIA, *Pipe Utilization*, CUDA documentation:
<https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/pipeutilization.htm>, 2015
- [7] Marco Danelutto, *Distributed Systems: Paradigms and Models*, Course notes, University of Pisa, 2014
- [8] T. Serban , M. Danelutto , M. Coppola, *Data parallel patterns on CPU/GPU mix*, Dept. Computer Science Univ. of Pisa, ISTI C.N.R. Pisa, 2012
- [9] J. Daniel Garcia, D. del Rio, M.F. Dolz, J. Garcia-Blas, L.M. Sanchez, M. Danelutto, M. Torquati, *Stream parallelism patterns*, Computer Science Dept. University of Pisa, Computer Science and Engineering Dept. University Carlos III of Madrid, <http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2016/p0374r0.pdf> , 2016
- [10] NVIDIA, *CUDA C Programming Guide*, CUDA Toolkit Documentation:
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2019
- [11] NVIDIA, *NVIDIA Profilers Guide*, CUDA Toolkit Documentation:
<https://docs.nvidia.com/cuda/profiler-users-guide/index.html> , 2019
- [12] Mark Harris, *CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler*, NVIDIA Developer Blog, article from <https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>, 2013
- [13] Vasily Volkov, *Better Performance at Lower Occupancy*, slide show from:
https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf , 2010
- [14] Vasily Volkov, *Understanding Latency Hiding on GPUs*, Technical report:
<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.pdf> , 2016
- [15] Mark Harris, *How to Implement Performance Metrics in CUDA C/C++*, NVIDIA Developer Blogs:
<https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/> , 2012

- [16] NVIDIA, *NVIDIA Library Documentation- Event Management*, YEAR
- [17] M. McCool, A.D. Robinson, J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, 2012
- [18] NVIDIA, *CUDA C Best Practices Guide*, from CUDA Toolkit Documentation:
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html> , 2019
- [19] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, M. Torquati, *P3ARSEC: Towards Parallel Patterns Benchmarking*, paper available here:
http://pages.di.unipi.it/desensi/assets/pdf/2017_SAC.pdf , 2017
- [20] Mark Harris, *How to Overlap Data Transfers in CUDA C/C++*, NVIDIA Developer Blog:
<https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/> , 2012
- [21] NVIDIA, *CUDA Compiler Driver NVCC*, CUDA Toolkit Documentation:
<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html> , 2019
- [22] NVIDIA, *CUDA-GDB: CUDA debugger*, CUDA Toolkit Documentation:
<https://docs.nvidia.com/cuda/cuda-gdb/index.html> , 2019
- [23] Mark Harris, *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*, NVIDIA Developer Blog,
<https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/>
 , 2013
- [24] Nicholas Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*, Addison-Wesley, first edition, 2013
- [25] Mads Ravn, *Simple image processing with CUDA*,
<http://madsravn.dk/posts/simple-image-processing-with-cuda> , october 2013

- [26] NVIDIA, *NVIDIA Tesla P100*, Technical whitepaper:
<https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf> , 2016
- [27] Steve Rennich, NVIDIA, *CUDA C/C++ Streams and Concurrency*, CUDA Developers Download:
<https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf> , 2011
- [28] Paulius Micikevicius, NVIDIA, *Performance Optimization: Programming Guidelines and GPU Architecture Reasons Behind Them*, available here:
<http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf>, 2013
- [29] S. Williams, A. Waterman, D. Patterson, *Roofline: An insightful Visual Performance model for multicore Architectures*, paper here:
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.156.756&rep=rep1&type=pdf> , 2009
- [30] S. Williams, D. Patterson, *The Roofline Model: A pedagogical tool for program analysis and optimization*, slide show here:
https://crd.lbl.gov/assets/pubs_presos/parlab08-roofline-talk.pdf , 2008
- [31] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, M. Püschel, *Applying the roofline model*, available here:
http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/ispass-2013_177.pdf , 2014
- [32] NVIDIA corporation, *Optimizing CUDA*, slide show from CUDA developer download:
http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Optimization_April-2009.pdf , 2009

- [33] Robert Hochberg, *Matrix Multiplication with CUDA - A basic introduction to the CUDA programming model*, from:
[https://www.shodor.org/media/content/petascale/materials/UPModules/
matrixMultiplication/moduleDocument.pdf](https://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf) , August 2012

Pánta Rheî.