



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	2
1.1.1	GPU Architecture and Data Parallel	2
1.1.2	Other Applications	2
1.1.3	GP-GPUs and Stream Parallel	4
1.2	Expectations	5
1.3	Results	6
1.4	Tools	6
2	Tools	9
2.1	CUDA	10
2.2	Profilers	13
2.2.1	nvprof	13
2.2.2	NVIDIA Visual Profiler	14
2.3	CUDA C/C++	15
2.3.1	Kernels	15

2.3.2	Thread Hierarchy	16
2.3.3	CUDA Streams	16
2.3.4	nvcc compiler	18
2.3.5	cuda-gdb debugger	19
2.4	Visual Studio Code	20
2.5	Tests, Result gathering, Plots	20
2.5.1	Bash scripts	21
2.5.2	Python scripts	21
3	Project Logic	22
3.1	Streaming Parallelism: Farm pattern	22
3.2	CPU-GPGPU: heterogeneous architecture	24
3.2.1	Overlapping: Data Transfer hiding	25
3.2.2	Occupancy of GPU cores	28
3.2.3	Occupancy drawbacks	30
3.3	Overall Logic	32
3.4	Tunings	39
3.4.1	Tuning on block and grid dimensions	41
3.5	CPU/GPU Scheduling	42
4	Implementation	44
4.1	Kernels	44
4.1.1	Repeated cosine	45
4.1.2	Matrix multiplication	45
4.1.3	Blur Box filter	47
4.2	Parallel Patterns implementation on GPU	48
4.2.1	Stream Parallel on GPU	48
4.2.2	Data Parallel un GPU	52
4.3	CPU and GPU Mix	54

5	Experiments	55
5.1	Expectations	55
5.1.1	Measures: What and How	56
5.1.2	Tests setup	61
5.1.3	Speedup	61
5.1.4	Results: gathering and evaluation	63
5.2	Simple-computation kernel	67
5.2.1	Results	71
5.3	Matrix Multiplication	78
5.3.1	Results	82
5.4	Image processing	94
5.4.1	Results	96
5.5	Results Summary	98
5.5.1	Stream parallel compared to Data parallel	99
6	Conclusions	106
6.0.2	Evaluation of the problem	107
6.0.3	Implementation and tests	109
6.0.4	Results and considerations	110
6.0.5	Final remarks and further works	112

CHAPTER 1

Introduction

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

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

Despite this, enthusiasm has been slowed down when scientific community had to deal with problems that seemed to be unsuitable for GPGPUs. But when the going gets tough, the tough get going and several studies and researches showed some good results and possibilities.

This gives oxygen to keep trying mapping to GPU apparently unsuitable problems and that's the core of this work too.

1.1 Goals

The main goal of this thesis is to study GPU's behavior when used for different purposes with respect to the common ones. In particular, we wanted to use a GPU to perform a code that comes closer to a *stream parallel pattern*. Then we observed ongoing, in terms of *completion time* and *speed up*. We now see in detail the concepts we've just introduced.

1.1.1 GPU Architecture and Data Parallel

GPU (*Graphics Processing Unit*) is a co-processor, generally known as a highly parallel multiprocessor optimized for visual 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, so more of the per-chip transistor budget is devoted to computation [1].

In most of situations, visual processing can be associated to a *data parallel pattern*. In general, we can roughly think to an image as a given and known amount of *independent* data upon which we want to do some computations. In most of cases, once the proper granularity of the problem has been chosen, this 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

However in recent years we're moving to GPGPUs (*General-purpose computing on graphics processing units*). In other words, lately GPUs have been used for

other applications than graphics processing.

One of the first attempts of 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 shaders, that broke the rigidity of the fixed graphics pipeline (for example LU factorization with partial pivoting on a GPU was one of the first common kernels, that ran faster than an optimized CPU implementation).

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

In our work we took advantage of CUDA features ¹.

One thing we should point out from GPGPUs birth: initially scientific applications on GPGPUs 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*).

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

Some computational motifs perform effectively on a GPU, while the effectiveness of

¹We'll show some further informations about CUDA in Section 1.4.

others is still an open issue. 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.

In particular a task-parallel system should:

- Handle divergent workflows;
- Handle irregular parallelism;
- Respect dependencies between tasks;
- Load balance all of this.

Those requirements can lead to inefficient use of the GPU memory hierarchy and SIMD-optimized 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; another is *H.264 Intra Prediction* video compression encoding; *Reyes Rendering*; Deferred Lighting.

But, 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 of an input stream, of which we don't know a priori the length or the emission rate.

Once the stream elements are available, parallel workers will make independent computations over them and, finally, the manipulated elements will become the output stream.

We recall as main stream parallel patterns ***Farm*** and *Pipeline*, the former is the object of this work.

The **Farm parallel pattern** is used to model embarrassingly parallel computations. The only functional parameter of a farm is the function f needed to compute the single task. Given a stream of input tasks [4]

$$x_m, \dots, x_1$$

the farm with function f computes the output stream as

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

It's not difficult to see that Farm pattern is really similar to a data parallel problem (in this case a *Map Pattern*). The key difference resides in the input/output data type:

- *data structures* for Data parallel patterns;
- *streams of items* for Farm.

This reveal the main problem of this work, that is the *Data Transfer times* between **host memory** (CPU side) and **device memory** (GPU side), and vice versa.

We'll show in detail all aspects of this and other minor 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. In other words we wanted to see that, running on GPU our streaming parallel code, it could take an advantage near the order of the number

of SMs (*Streaming Multiprocessors*).

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

- Data transfer time had in some way to be hidden behind Computation time;
- The GPU had to achieve a full *Occupancy* ².

Once we could gain these two 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 gaining some advantages with respect to CPU processing:

- We can delegate our streaming problems to the GPU while the CPU can compute other things, this allow the CPU to not being saturated (especially when stream has high throughput or each element require 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 ³;
- We hopefully want to see a GPU speedup with respect to the CPU, or see the same performances at worst.

1.3 Results

Put a summary on results here.

1.4 Tools

As mentioned in Subsection 1.1.2 we exploited NVIDIA's CUDA Toolkit. ⁴ In particular:

²We'll insist on occupancy topic in Chapter 3.

³See Section 3.5

⁴In Chapter 2 will be shown all features and details about the aforementioned tools.

- 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 Nsight**;
- 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:

- 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 notions presented in this introduction will be seen in depth. In Chapter 2 there will be an accurate description of all employed tools and how they were used.

We'll enter in the core of this work in Chapter 3, where we'll see the logic of the project, with both written and graphical illustrations. In other words here we point out the main ideas and concerns behind our approach and solutions.

In Chapter 4 will be presented and explained main implementation choices and there will be listed some fundamental part of the code.

Then in Chapter 5 will be shown either how experiments and test were set, obtained

results and some respective plots.

And we'll end up with Conclusions and some final remarks.

CHAPTER 2

Tools

In this project all tools and elaborations were made in GNU/Linux environment. We had two available remote computers, to which we connect by `ssh` command on terminal. 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)
- 80 CPUs Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
- 4 GPUs Tesla P100-PCIE-16GB

3. M40 remote server

- Ubuntu 16.04.6 LTS (4.4.0-154-generic x86_64)
- 48 CPUs Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
- 4 GPUs NVIDIA Tesla M40

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

In this work we mainly made use of many tools 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 CUDA

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model provided for compute engine in NVIDIA GPUs, to solve many complex computational problems (sometimes in a more efficient way than on a CPU).

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems and their parallelism continues to scale with Moore's law.

We used CUDA with C++ support. At its core are three key abstractions that are exposed to the programmer as a minimal set of language extensions:

- A hierarchy of thread groups;
- Shared memories;

	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.

- 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 –.

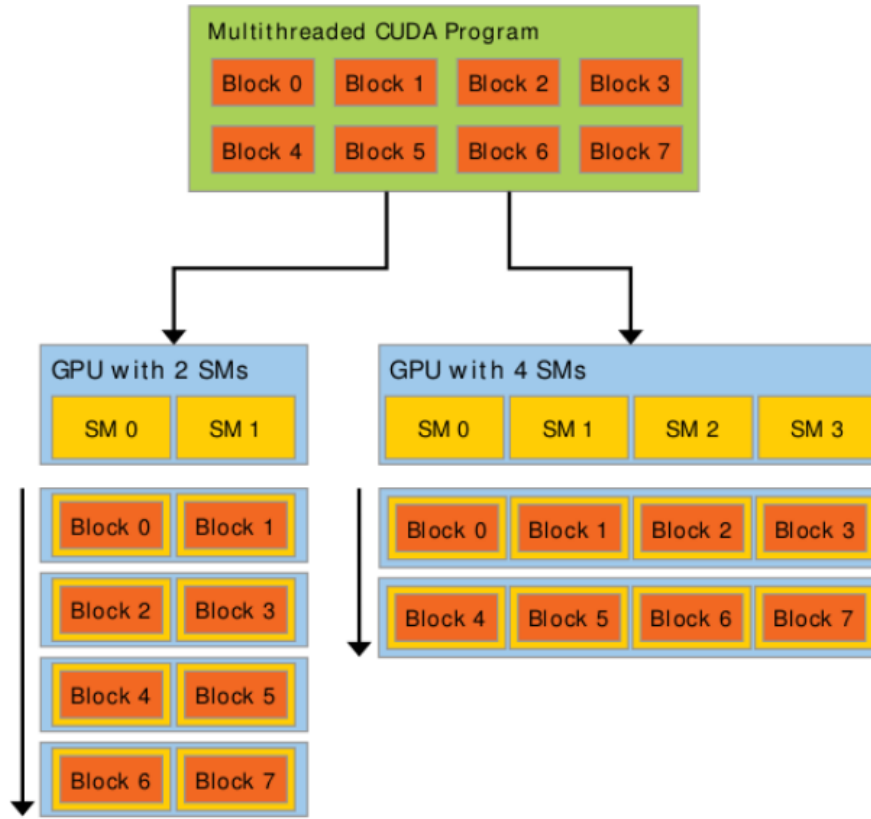


Figure 2.1: GPU scalability.

Indeed, *each block of threads can be scheduled on any of the available SMs within a GPU, in any order, concurrently or sequentially*, so that a compiled CUDA program can execute on any number of multiprocessors 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 [5].

2.2 Profilers

NVIDIA profiling tools were useful to optimize in performance our CUDA applications, we used three different versions: the **nvprof** profiling tool enables you to collect and view profiling data from the command-line.

2.2.1 nvprof

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 console ¹ or can be saved in a log file for later viewing ² [6, 7].

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

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 number of calls and total, max, min and average time) [6].

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 ...
```

¹The textual output of the profiler is redirected to `stderr` by default.

²Or for later import into either `nvprof` or the NVIDIA Visual Profiler).

Although when we launched our tests, we wanted to consult profiling results after running or whenever was necessary, it was useful especially when we had to compare them with time probes inside the code. So 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.

2.2.2 NVIDIA Visual Profiler

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

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 [6].

We used the standalone version of the Visual Profiler, `nvvp`. Furthermore we used NVIDIA Nsight Systems, the advanced version of Visual Profiler, this is a low overhead performance analysis tool that provides insights to optimize software, to investigate for bottlenecks. It also identifies issues, such as GPU starvation, unnecessary GPU synchronization, insufficient CPU parallelizing, unexpectedly expensive algorithms across the CPUs and GPUs of their target platform.

In particular in this project `Nsight Systems` was used in developing phase, to check if code was properly written to hide as much as possible data transfers ³, even though sometimes it may happens that profilers introduce some sampling synchronization, giving not fully reliable visual results.

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

2.3 CUDA C/C++

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

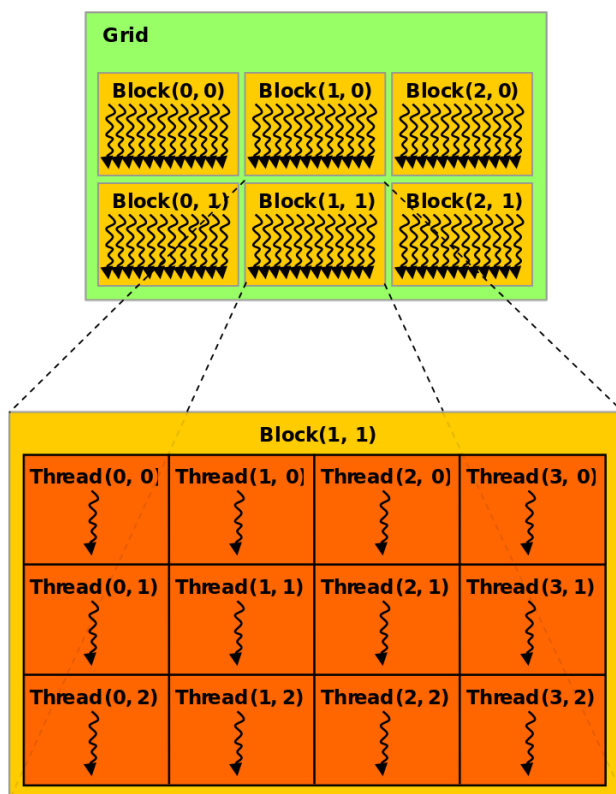


Figure 2.2: Above: a Grid formed by Blocks.

Below: a Block formed by Threads.

2.3.1 Kernels

CUDA C allows to define particular C functions, called *kernels*, when called, these are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel 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 [5].

2.3.2 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 threads ⁴.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that
$$Totalnumberofthreads = \#threadsPerBlock \cdot \#blocks$$

Blocks in turn are organized into either one, or two, or three dimensional ***grid of thread blocks*** as illustrated by Figure 2.2. 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`. [5].

2.3.3 CUDA Streams

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

⁴see Table 2.1 for limits in the machines we used).

`asyncEngineCount` device property ⁵, which is greater than zero for devices that support it. If host memory is involved in the copy, it must be *page-locked*.

Streams

Applications manage the concurrent operations described above through **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 their 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) [5].

A brief code example:

Listing 2.1: CUDA Strams creation

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Each of these streams is defined, by the following code sample, as a sequence of 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(inputDevPtr + i * size, hostPtr + i * size, size,
        cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr +
        i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
        cudaMemcpyDeviceToHost, stream[i]);
}
```

Each stream copies its portion of input array `hostPtr` to array `inputDevPtr` in device memory, processes `inputDevPtr` on the device by calling `MyKernel()`, and copies

⁵ See *Device Enumeration* in Table 2.1, For both of our machines, from the *deviceQuery*, we get 2 copy engines.

the result `outputDevPtr` back to the same portion of `hostPtr`. Note that `hostPtr` must point to page-locked host memory for any overlap to occur.

Streams are released by calling `cudaStreamDestroy()`.

Listing 2.3: CUDA Strams destroy

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

2.3.4 nvcc compiler

To compile the CUDA C++ code was 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 ⁶.

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. `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 Kernels ⁷ 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

⁶NVCC documentation: <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#introduction>

⁷See subsection 2.3.1

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.

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

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

Here we present compiler versions installed in our two machines:

- **Tesla P100**

nvcc: NVIDIA (R) Cuda compiler driver, release 10.1, V10.1.168

- **Tesla M40**

nvcc: NVIDIA (R) Cuda compiler driver, release 10.1, V10.1.105

2.3.5 `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.

Cuda-gdb main features are:

- it gives 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.

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 exactly what code was doing inside our kernels.

2.4 Visual Studio Code

Visual Studio Code is an open-source code editor by Microsoft for Linux Operative Systems too. It includes support for debugging, embedded Git control and GitHub, syntax highlighting, intelligent code completion, snippets, and code refactoring⁸.

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 peripheral tools were involved in this project. In particular we needed:

- a way to serially run executions of our applications, possibly varying input dataset on interest values;
- put all results on text files;
- implement a script to compute averages, speedups and other interest metrics, from results on text files;
- a generator of graphs on important values from the obtained calculations.

⁸Other infos: https://en.wikipedia.org/wiki/Visual_Studio_Code

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

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

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 shell⁹.

`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.

2.5.2 Python scripts

As Python ¹⁰ 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` ¹¹

⁹<https://www.gnu.org/software/bash/manual/>

¹⁰The Python version used to compile, on local host, our scripts is: Python 2.7.6.

See documentation: <https://docs.python.org/2/>

¹¹<https://matplotlib.org/>

CHAPTER 3

Project Logic

The reasoning in this work started by considering the characters of a problem that is recognizable in a Farm parallel pattern. Then the study moved to consider how a GPU works, 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 performances ¹, 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 Streaming Parallelism: Farm pattern

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

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

output stream.

We focused on farm pattern, modeling embarrassingly parallel stream parallelism. For example, consider the correspondent skeleton:

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

whose type is

$$farm :: (\alpha \rightarrow \beta) \rightarrow \alpha stream \rightarrow \beta stream$$

The parallel semantics, associated to the higher order functions, states that the computation of any items appearing on the input stream is performed in parallel

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. A stream is not a vector. Items of the stream may 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 [4].

We recall that the only functional parameter of a farm is the function f needed to compute the single task. Given a stream of input tasks

$$x_m, \dots, x_1$$

the farm with function f computes the output stream

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

Its parallel semantics ensures it will process the single task in a time close to the time needed to compute f sequentially [4].

3.2 CPU-GPGPU: heterogeneous architecture

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

So we asked ourselves what could happen if we wanted to manage such computations on 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 cores apply specific computations on all items of the stream (as soon as they 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. Indeed these phases introduce an overhead per se, but especially in Farm parallel pattern they can be a not negligible bottleneck.

We should not forget that our input is a stream: even if elements are available with a high throughput, they come "one by one". In reality, as we mentioned in the previous section, we have only a part of input available at one point, so it's not realistic to have one worker per stream element.

In our case we can suppose Streaming Multiprocessors to be our Farm workers, so if we don't give enough work (items) to each core we would have a resources waste. Furthermore, we surely don't want to transfer from/to GPU one element at time but, at the same time, in some way we have to keep as much as possible the nature of a Stream parallel patterns.

Thus at this point has become necessary for first to find some stratagems:

- Hide as much as we can data transfer times, 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. Since the machines we worked on both have concurrency on data copy and kernel execution and they have 2 copy engines, we exploited this capability combined with streams.

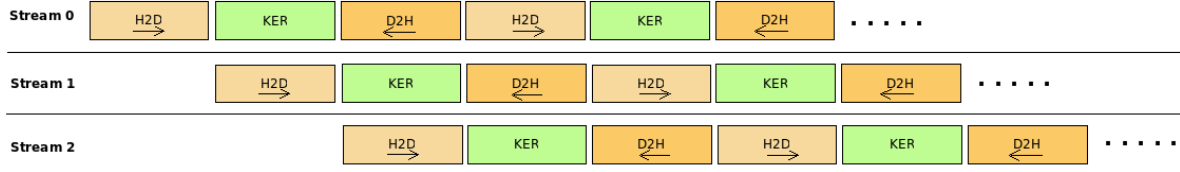


Figure 3.1: Ideal behavior for 3 CUDA Streams.

In this way we wanted 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 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, finding 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, the balancing between data transfers time and kernel executions time.

Given that our GPUs supported all kind of above mentioned concurrencies, among the above factors the only that can influence in our case are the order, in which commands are issued to each stream, and the kernel/data transfer time balancing.

We followed some useful guidelines to improve the potential for concurrent kernel execution:

- All independent operations should be issued before dependent operations;
- Synchronization of any kind should be delayed as long as possible.

For the former, we have that all operations are independent, given the Farm nature. Indeed all stream items and their computation given by workers, are independent.

For the latter, we were careful to avoid *Implicit synchronization*, this happens when are introduced host issued operations in-between different streams commands.

Moreover we avoided all possible *Explicit synchronizations* ².

Another important face of overlapping, is that we should try 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 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 heavy computations or "*irregular kernels*". By irregular we mean that any flow control instruction (`if`, `switch`, `do`, `for`, `while`) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, 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. So we should avoid different execution paths within the same warp [5].

Whilst the latter case can happen when we move an amount of data at each transfer such that it takes more time than calculations. So in this case the dominant factor will be the data transfer.

²In CUDA there are several command to force synchronization either between host and device, or between streams etc.

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 arrive near the full *Occupancy*.

Clearly, when we start, we'll have a portion of time, a sort of "warm up" phase, where we'll have first data transfers and kernels. So we'll have a narrowed number of running kernels. But as soon as we could have enough data transfers and so a lot of kernels, we would reach a workload peak on GPU.

In practice, when we just said *lot of kernels*, we meant a lot of small groups of items on which apply our computations, in other words this small items groups will be assigned each to a thread block.

Let's spend a bit to explain better what Occupancy means.

To *maximize utilization* the application should be structured in 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.

Here main ways to maximize utilization:

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;
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.*

From the above, it's clear that occupancy is directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction 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".*

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.

In the case of a back-to-back register dependency (i.e., some input operand is written by the previous instruction), 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.

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 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 [5].

Register and shared memory are others important Occupancy variables, but we didn't focused much on them as on execution configuration.

At this point, we had to reason about how to maximize Occupancy in our Farm parallel pattern.

For 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 ³.

So we mainly had to put our attention on kernel Execution configuration and number of kernels launched, 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.

That's why in this work we had to take into account of both sides of occupancy, this is another reason that led us to experiment and measure various settings and implementations.

³We'll see what kind of kernels we used to test the farm parallel pattern, with some code slices in Chapter 4.

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 only 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. 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 for these times, but they can be hidden by overlapping with other operations.

```
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, than occupancy, to hide latency is *Instruction Level Parallelism*.

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

Another 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 4 or even 8 floats per thread, instead of copying one and run more blocks and allocate shared memory to control occupancy.

For example some common concepts⁴ on CUDA says:

- "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";
- "In fact, 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.."

For the former there are studies that shows 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 close 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)[8].

3.3 Overall Logic

We have an input stream of items, in our case we chose `floats`, we don't know how much they are and their arrival frequency. What our project's doing, will be summarized in

⁴From CUDA Programming Guide.

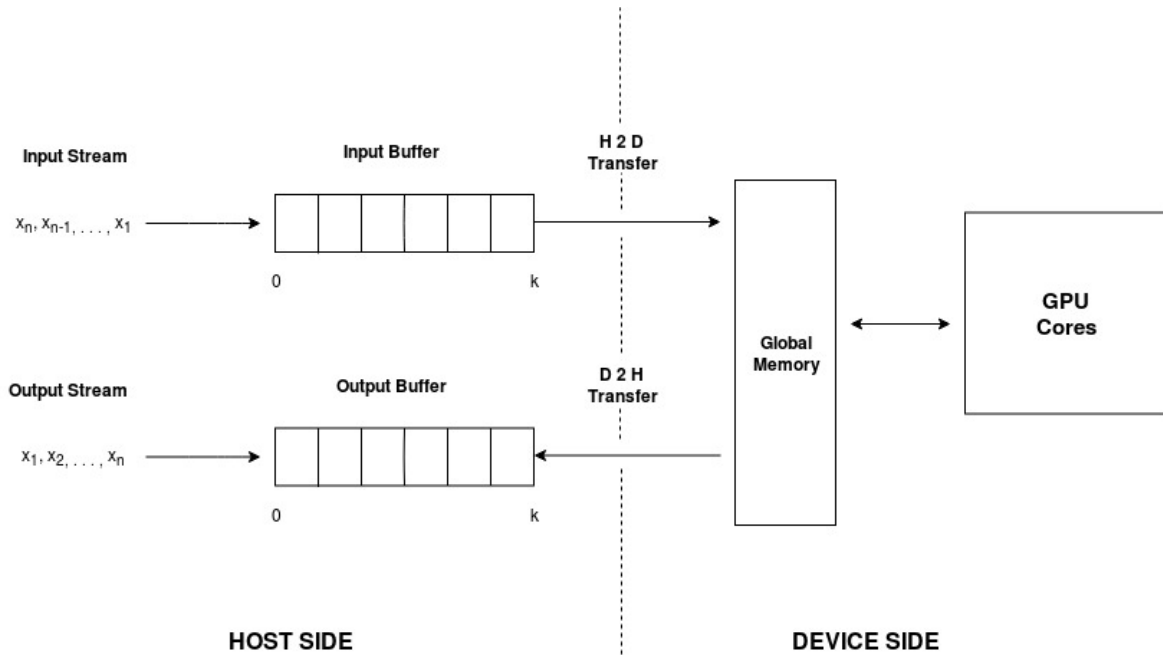


Figure 3.2: Here we have a general and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

the following steps:

1. As items start to arrive, we accumulate say k items at time in a buffer;
2. As the buffer is full, on a certain stream say `streams[k]`, we send out that chunk of data to the device (GPU Global memory);
3. Immediately after the data transfer call, we launch the kernel execution, with a certain Execution configuration. The kernel call will be placed to `streams[k]` as well;
4. Once the kernel ends its computations, we copy back to host, on `streams[k]`, the chunk of manipulated data as output buffer;
5. From the output buffer we'll send each item as output stream.

This behavior is illustrated graphically in Figure 3.2. Here we can see our input stream and every k items we transfer them to Global memory of GPU.

For reasons we showed in the previous sections, it would be unfeasible to work on single items but, at the same time, we should maintain a pattern as close as possible to Stream parallel. That's why we chose to work on chunks⁵ of k items, where k is empirically determined to be a relatively small number of items and strictly related to execution configuration on kernel, in particular to block size.

Items will be spread all over warps in a way such that for each item will be applied a set calculations, specified inside kernel code, that will be performed by one of the active threads in the warp.

From that figure it may seem we're sending only k items at time to/from GPU, assuming k items in a buffer as a single item, 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 Streams⁶ come into play and we used them relying on the following ideas:

1. We have as many streams as Streaming Multiprocessors⁷ and, at any given time, each of them hopefully issues a data transfer or a kernel execution;
2. We should arrive at the point where each stream have issued at least one kernel launch, ideally we expect that each kernel execution is taken over by a certain

⁵We represented chunks as arrays, but they can be either small matrices or tiny images, as we'll see in Chapter 4.

⁶Don't confuse input/output stream in Farm parallel pattern with CUDA Streams.

These are two completely different notions: the first refers to the parallel pattern behavior of input/output data, the last refers to special CUDA commands (shown in Section 2.3.3).

⁷Again CUDA Streams are a different concept with respect to Streaming Multiprocessors. The first are a set of commands, the last are physical processing units.

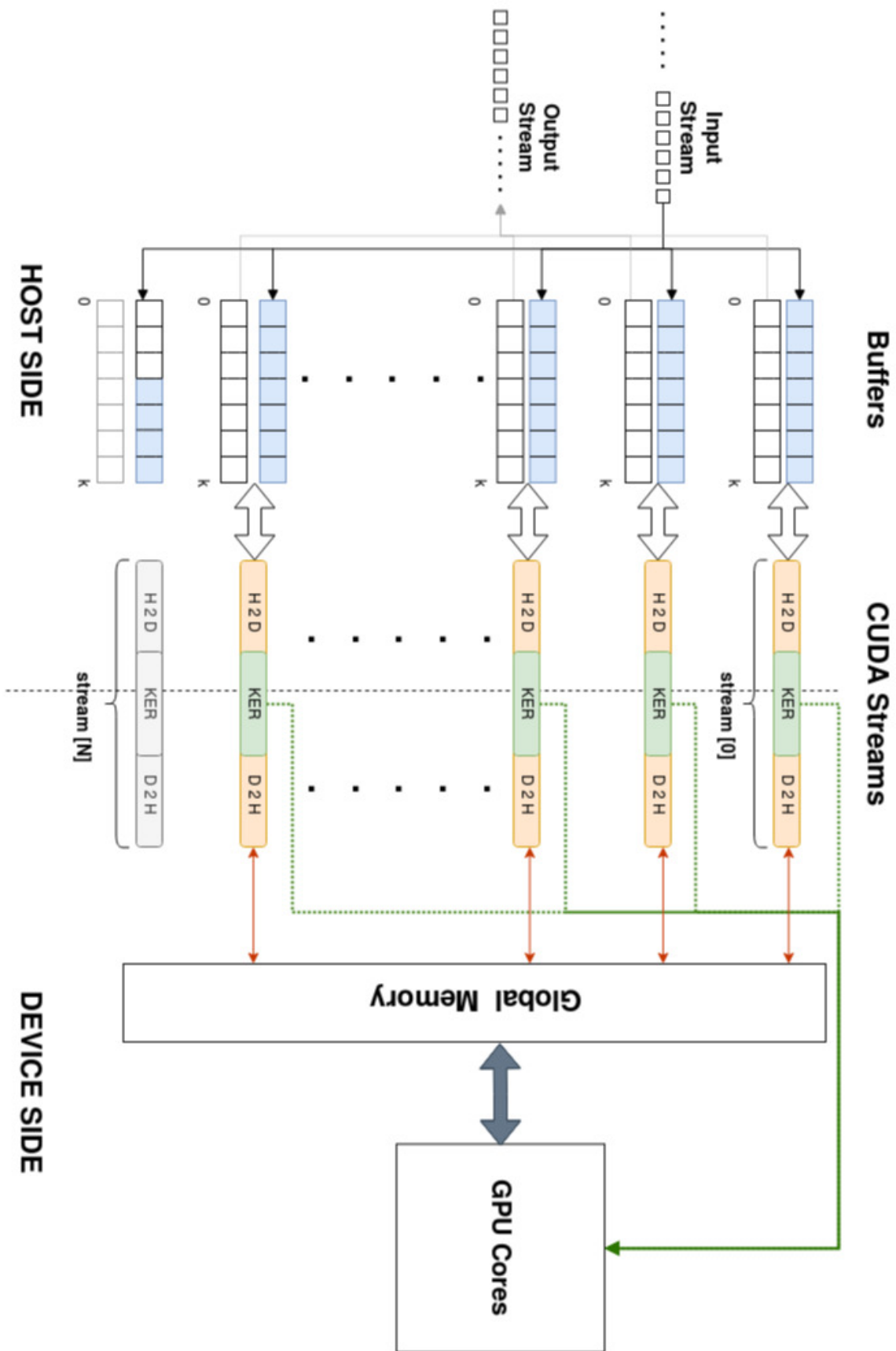


Figure 3.3: Here we have a general and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

multiprocessor. So we want to arrive at a moment in which we reach a work peak, where almost all SMs are busy;

3. Obviously each kernel execution configuration should be well tuned, in order to take advantage of the maximum of resources in a multiprocessor.

All of those parameters have been established at first with some reasoning and assumptions on NVIDIA GPUs nature, later we moved on experimental proves ⁸. Measures and other estimation lead us to consider specific values for those variable parameters.

So from the above facts is clear that we're trying to exploit 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 **SMs-workers** apply a **function-kernel** to all **tasks-chunks**. Bringing all pieces together we can summarize all project logic in Figure 3.3.

Putting down in words that schema:

- We have N CUDA streams, where N is the number of Streaming Multiprocessors in the machine where code is running;
- As input stream items arrive, we let them fill buffers;
- In a Round-Robin fashion we spread ready buffers all over the CUDA streams as follows:
 1. As soon as the i^{th} buffer is full, it's asynchronously sent on `stream [i]` to the GPU, with the command

```
cudaMemcpyAsync( devBuffer, hostBuffer, bytes, cudaMemcpyHostToDevice,  
stream [i]);
```

⁸We'll see in next section more informations about Tunings.

2. Soon after we put kernel call, on the `stream [i]`, to make desired computations on that buffer;

3. Then, asynchronously again, we bring back results to host side, using the instruction

```
cudaMemcpyAsync( devBuffer, hostBuffer, bytes, cudaMemcpyHostToDevice,
                stream[i]);
```

4. hopefully this should make a Streaming Multiprocessor, or a part, busy.

Initially only few cores will be really busy, but as soon as the buffers and streams get full, the pressure 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 means that we tried to have the maximum number possible of active threads inside each SM, having to do some work.

Now let's put a magnifying glass upon Figure 3.3, we want take a closer look to what we just said in Figure 3.5.

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

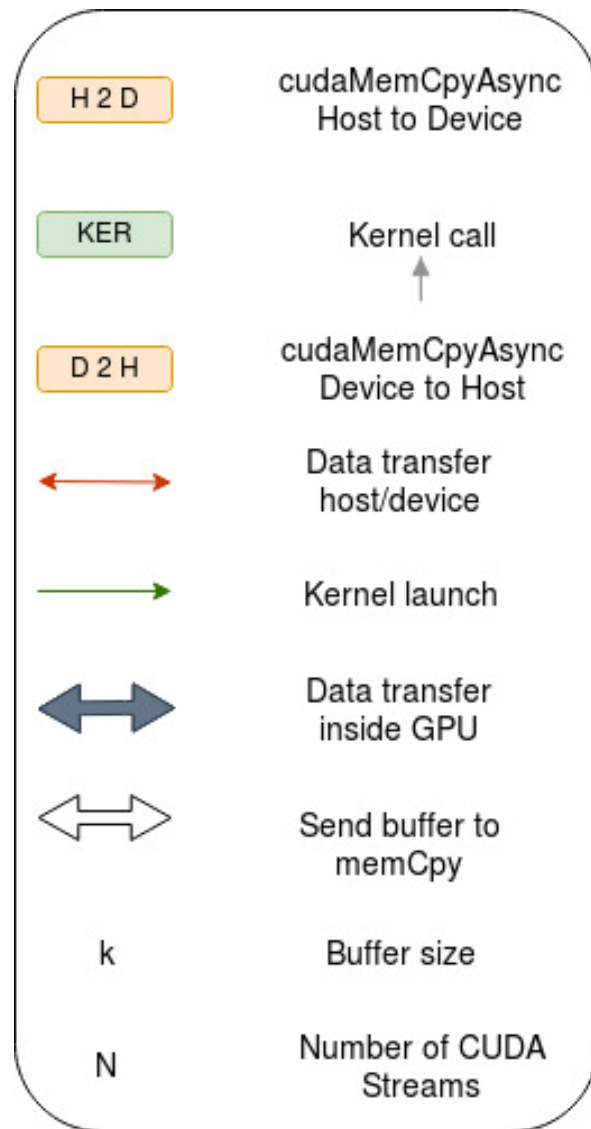


Figure 3.4: Legend about Figure 3.3.

stream.

The behavior of overlapping between different streams, isn't predictable. 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 [k]` (for some $k \in [0, N - 1]$, for $N = \#SMs$). Obviously, when the number of streams is greater than 3, we can have only 2 data transfer operations issued at the same time (by two distinct streams) ⁹.

Note that in the Figure 3.5, we represented a single kernel execution as fully occupying an entire SM; in reality it's not exactly how it goes. We'll see how we practically tried out Streaming Multiprocessors *occupancy* in Chapter 5.

So 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 *classical approach*.

By "classical approach" we mean to transfer data and execute kernels without any type of overlapping.

This is equivalent to send input to device, wait for data transfer completion on host, call kernel, send data back to host, only when all computations ended up, and finally results are transferred back to the host, waiting for data transfer ending. This similarly means that if, for example, we have 3 CUDA Stream we would expect to take an advantage on only at most 3 SMs (at peak work flow), so this should give us an improvement, in completion time, of at most 3 times compared to classical approach.

⁹As mentioned in Chapter 2, for concurrent memory copy between host and device, we have 2 copy engines.

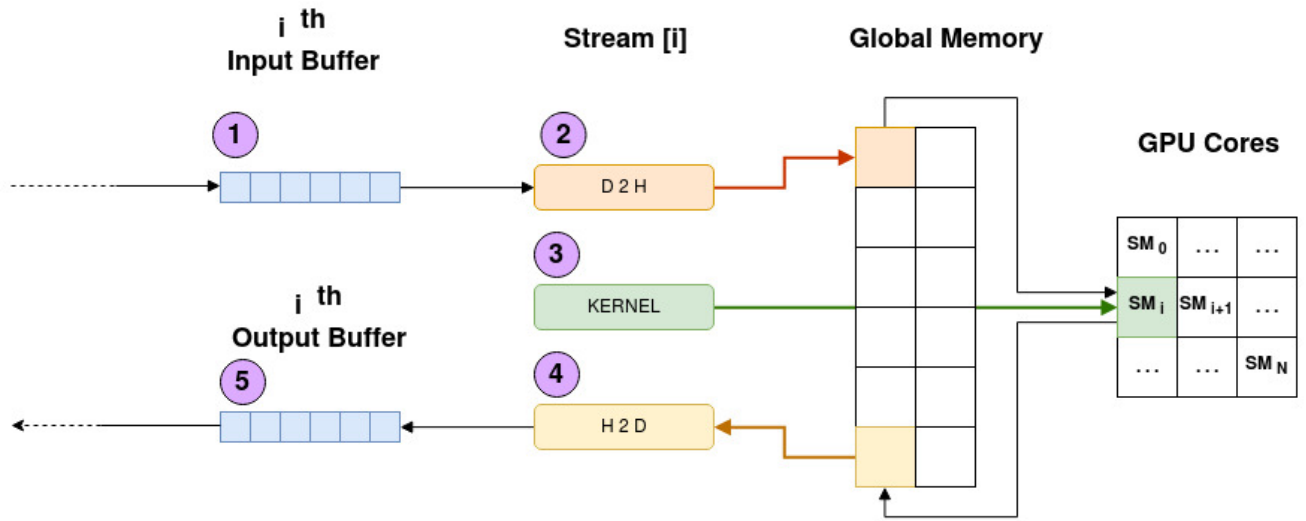


Figure 3.5: Here we can see what exactly happens in a certain CUDA Stream. Light violet numbered labels shows the order in which commands are issued by host to a certain stream.

3.4 Tunings

We showed a lot of peculiar behavior and architecture characteristics, because all of the above mentioned were taken into account for different implementation, 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 give a huge work slice to experiments and empirical evaluations. This has many reasons why:

- It was important to think about a general logic, that wasn't architecture-dependent¹⁰;
- To validate our total idea we had to make a lot of experiments, time measures, examples and even counterexamples;
- Clearly experiments required to get a little deeper on NVIDIA GPUs architecture,

¹⁰At least we can say that the complexive view, showed in Fig. 3.3, can be plausible with almost all NVIDIA architecture having ≥ 2 copy engines and allowing concurrent kernel execution.

having a generic idea on good practices, not necessarily bounded to a specific model;

- Finally, we had to consider some feature totally model-bounded to launch tests and to give a sense to obtained results.

So after the logical phase, has followed a *tuning phase*, that for first has been done facing general NVIDIA GPUs behavior and structure ¹¹. We followed for first some important best practices to try some good tunings:

- The effect of execution configuration on performance for a given kernel call generally depends on the kernel code, so experimentation is recommended 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 warps as much as possible ¹²;
- We exploited *Occupancy Calculator*¹³ both in spreadsheet and API functions¹⁴ formats , [\[5\]](#).

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

¹¹In Chapter 5 we'll mainly see tunings based on the GPUs used to run tests –**P100** and **M40**.

¹²That's because kernels issue instructions in warps (groups of 32 threads). For 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.

¹³Those tools are included in CUDA Toolkit, they assist programmers in choosing thread block size based on kernel behavior, register and shared memory requirements.

¹⁴These are special function to call inside code, we'll see in Chapter 4 a code example on how and where they are used.

- The number of Streams;
- The number of blocks (grid size);
- The number of threads per block (block size) and as a consequence
- The buffer dimension.

After a lot of attempts and experiments, for each kernel type, we extrapolated best suitable values ¹⁵.

3.4.1 Tuning on block and grid dimensions

It's important to understand some main concepts, that are the basis for the logic of this project.

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 study of this project. There's a tight bond in between *Occupancy*, *kernel execution configuration* and kernel code nature. For 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 an SM, it will become an *active block*.

In an SM we can have multiple blocks running independently, each of which grabbing its portion of resources, ie we can have multiple active blocks on a SM until they don't hit the maximum allowed. Here may emerge some particular cases:

1. Maximum block dimension (aka lot of threads per block), means that a smaller number of blocks, even only one, can fit in a certain SM;
2. Small block dimension, means a higher number of blocks, even the maximum blocks number per SM supported, running on the same SM;

¹⁵Check Chapter 5 to see all main values and their respective performances

In the first scenario, we can have cases of really good performances, but it may not be true when we have unbalanced workload for different blocks. This means that if a block has lot of work to do, it will monopolize resources of the SM in which it's running, making all other 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 (for example for $blocksize = 32$), we can have really poor performances due to poor resources exploitation, but for some kernels it may give a gain, especially in cases as the above mentioned of unbalanced workload.

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

So our tests had to face with the above explained behavior, without forgetting the nature of the various implemented kernels and the relative latencies.

3.5 CPU/GPU Scheduling

In addition to the main logic of our project we introduced another branch of study.

In particular, we extended the Farm parallel pattern on GPGPU introducing a sort of ***CPU/GPU Scheduler***.

This consist in an implementation that, given an initial work percentage, it gradually and experimentally tunes those percentages to balance jobs. In particular, the scheduler adjusts the dimensions of data chunks directed to CPU or GPU on the basis of previous measured completion times of both processors.

Clearly, starting from a user provided percentage, measured times are used to recompute percentages (and thus chunks dimension). So, in a finite number of algorithm steps, the two portion size will stabilize around two values.

This allow us to let host and device cooperate to apply same computations, but with different workloads. Clearly the main idea is to let GPU have a greater workload with respect to the one for CPU, so that latter can be lighten from doing the entire compu-

tations; at the same time, having a good occupancy on GPU, we can gain a speedup compared to letting only one of the two processors doing all the work.

CHAPTER 4

Implementation

Given the logical schema of the previous chapter, we had to build different types of code to observe their behaviors in performances.

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

- Repeated cosine application;
- Matrix multiplication;
- Blur Box filter.

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 structure.

4.1.1 Repeated cosine

This is a very simple kernel, in which given as inputs an array of floating point and a number M , it computes, for each input float, the cosine applied M times. The output will be again a floats array (given by cosines).

Note that, since here we're working on one-dimensional data structures, 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 almost equal workload for each thread that could execute that code.

It was a very useful kernel, because changing a single parameter we could then test situations either of low or high iterations amount, ie different workloads. Moreover this is clearly a computation-bounded kernel.

4.1.2 Matrix multiplication

Here's the most classical version of matrix multiplication in CUDA. An important difference from the application above is that now we're working with matrices, both in input and output. So, for convenience, we'll use two-dimensional thread blocks.

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 $i = 0, \dots, N$ and `sum` will be `C[ROW, COL]`, ie one of the items result matrix.

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;
    }
    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 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 three times ¹. This means we have a low arithmetic intensity w.r.t. memory accesses, so cores won't 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*.

4.1.3 Blur Box filter

The last type of application implemented in this project is an image processing kernel to apply blur filter. Here input and output pictures are represented as char buffer, items are **RGB** values in [0, 255] that represent pixels such as "RGB RGB RGB...". 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 writes it to the output image. This filter is known as a Box blur ².

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){
        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) {

```

¹We'll have two pull from memory for A[i, k] and B[k, j] and a push for C[i, j].

²Another 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.

```

        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;
}

```

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 executions 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 Fig. 3.3. Let's start by explaining the setting phase:

- The block dimension is provided as command line parameter;
- Then grid dimensions are initially determined;
- Here we can have two different settings

- The chunk has the same dimension of the block, given the capabilities in our machines means having a size < 1024 . The grid will have size one;
- The chunk takes as dimension the maximum number of threads active in a SM, for both our machines is 2048. Here the grid will be adjusted to cover the chunk dimension w.r.t. the block dimension, ie $GRID = \text{maxThreads}/BLOCK$;

Listing 4.4: Kernel Launch configuration, ie Grid and Block dimensions setting

```
#ifndef LOWPAR
    GRID = 1;
    chunkSize = BLOCK*GRID;
#else
    GRID = maxThreads/BLOCK;
    chunkSize = BLOCK*GRID;
#endif
```

Now we arrive at the core of the implementation. The number of CUDA streams to spawn, as we previously told, is equal to the number of Streaming multiprocessor in the target machine. So, we start by distinguish two different cases:

- Number of streams equal to zero, this means we won't use CUDA stream
 1. We allocate space for our device chunk, with a simple `cudaMalloc` (because here we'll not use streams);
 2. As input stream items arrive, we store them on the host chunk buffer;
 3. We send it to the device as soon as it's full, with a simple `cudaMemcpy`³;
 4. We launch the Kernel, that will execute as soon as data is fully copied;
 5. We call another `cudaMemcpy` to bring back results to host.

Listing 4.5: Data transfer host/device and kernel call, NO-CUDA Streams version

```
void cosKer(int m, int chunk, float *x, float *cosx, float *x_d)
```

³Note that `cudaMemcpy` is a blocking operation w.r.t. the host, this means that other CUDA calls from the host will be issued after data is fully copied to device.

```

{
    int xBytes = chunk*sizeof(float);

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

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

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

```

- Number of streams greater than zero, this means we will use CUDA stream, so steps are analogous to the previous except for some tricks (See Code Listing)
 1. We allocate space for our device chunk, with a `cudaMallocHost` (in order to use CUDA streams and gain best overlapping possible, host should allocate memory as *Pinned*);
 2. In a Round-Robin fashion, we send full chunks in an asynchronous way, using `cudaMemcpyAsync` ⁴;
 3. We launch the Kernel in the same CUDA stream of the previous copy;
 4. We call another `cudaMemcpyAsync` to bring back results to host, on the same stream as before.

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) );
}

```

⁴Note that `cudaMemcpyAsync` is non-blocking for the host, in parameters we'll have to specify which stream will issue the copy.

```

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

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

```

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

In particular we want to show that a Stream Parallel Pattern would be unfeasible in terms of device completion time, because of overhead introduced by data transfers. Instead, we wanted to show that, with CUDA Streams and some other precautions and experiments, a Stream Parallel Pattern could work near to the Data Parallel performances.

It may be interesting to see how the Round Robin scheduler sends buffers to the function `cosKerStream` via CUDA streams, we present a pseudo-code version that shows only main features:

Listing 4.7: Host side pseudo-code: input stream + kernel launcher function

```

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 project ⁵.

As we can see we presented most important code parts relative to the Cosine Kernel, but the structure and the implementation to execute Matrix multiplication and Blur Box are totally analogous.

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. In the reality we know that we cannot have such informations on input/output streams. So we make the assumptions to know input stream length only for a time measuring purpose.

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

Furthermore, this allow us to compare our Farm model, having an input stream of `N_size` items length, with a Data Parallel model sending all `N_size` items in once, computing them all in a classic configuration kernel and send back all again ⁶

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

```
x = (float *) malloc(N_size*sizeof(float));
cosx = (float *) malloc(N_size*sizeof(float));
gpuErrchk( cudaMalloc((void**)&x_d, N_size*sizeof(float)) );

generate N_size items and put in the "x" data structure

createAndStartEvent(&startEvent, &stopEvent);

float msKer = optimalCosKer(M_iter, N_size, x, cosx, x_d, clocks, clocks_d);

/** Kernel launcher */
float optimalCosKer( int m, int n, float *x, float *cosx, float *x_d){
    int gridSize;      // The actual grid size needed, based on input size
    int minGridSize;   // The min grid size needed to achieve the maximum occupancy
                       // for a full device launch
    cudaEvent_t startEvent, stopEvent;

    cudaOccupancyMaxPotentialBlockSize( &minGridSize, &BLOCK, cosKernel, 0, 0);
    GRID = (n + BLOCK - 1) / BLOCK; // Round up according to array size

    gpuErrchk( cudaMalloc((void**)&clocks_d, GRID*sizeof(int)) );
    createAndStartEvent(&startEvent, &stopEvent);

    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) );

    gpuErrchk( cudaPeekAtLastError() );
    cudaDeviceSynchronize();
    float ms = endEvent(&startEvent, &stopEvent);
}
```

⁶And this is how generally the GPU is meant to be used and this is the kind of problem a GPU is designed for.

```
        return ms;
    }
```

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.

This was one example of the use of 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 Cosine kernel, but the implementation structure is analogous to the one for Matrix multiplication and Blur Box. For example, in Matrix multiplication, we'll have a stream of small matrices for Farm parallel and a single huge matrix for Data parallel.

4.3 CPU and GPU Mix

CHAPTER 5

Experiments

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

The second section, for each kernel implementation, will explain how tests are set up, ie chosen datasets for each type of code and scripts main features and what results we get. In particular, will be shown time measures and plots with some remarks. The last section gives a brief summary and some final consideration. Furthermore we'll show 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**, ie the total time needed to perform all 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 for 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 these concepts 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 and this may generally lead to bad occupancy;
- 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 expect to have a considerably lower speedup.

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 are relative to *data transfers* and *kernel execution* 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 a lot of streams are spawned.

If we want to have as many CUDA Streams as many SMs in device, then creation costs in the order of hundreds of milliseconds and destruction 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, given that initially we create CUDA streams, then we'll run kernels on an 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 be 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:

```
/**** Code with events time probes *****/
streamCreate(streams, nStreams); // Create CUDA streams

createAndStartEvent(&startEvent, &stopEvent); // Create "start" and "stop" events,
start recording

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

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

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

```

        . . . .

        ++k;

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

msTot = endEvent(&startEvent, &stopEvent);
cudaEventDestroy();

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

/**** Events end and time measure collection ****/
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 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
    ernel<<<GRID, BLOCK, 0, streams[i]>>>(input_device, output_device, ...);
#ifdef 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*. 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 5.1.1.

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*”). 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 [9]. 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.

It’s important to point out why we used events on the default streams. Given the asynchronous nature of CUDA calls, that we perform in non-default stream, the

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

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.

This is one of the reasons why we chose to put timers outside the loop over input stream. We could insert events inside the loop, instead, 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, apart from synchronization time.

For first problem, we recall that `cudaEventSynchronize()` blocks CPU execution until the specified event is recorded, but 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, 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 some special cases and their relative technical details and performance analysis.

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

5.1.2 Tests setup

Once we determined the time measure 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

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 a 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 compares the latency for solving a certain computational problem on one hardware unit, generally referred to as *worker*, versus solving the same problem on P hardware units, as below

$$speedup = S_P = \frac{T_1}{T_P}$$

where T_1 is the latency of the program with one worker and T_P is the latency on P workers. If T_1 is the latency of the parallel program running with a single worker, the above equation for speedups, is sometimes called *relative speedup*, because it shows relative improvement from using P workers. This uses a serialization of the parallel algorithm as the baseline. Sometimes there is a better serial algorithm that does not parallelize well.

If so, it is fairer to use that algorithm for T_1 , and report *absolute speedup*, as long as both algorithms are solving an identical computational problem.

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**.

However, as exceptions, an occasional program will exhibit **superlinear speedup**⁴. However, in general, **sublinear speedup** is the norm[11].

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

Amdahl identified in the execution time T_1 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:

$$\begin{aligned} T_1 &= T_{ser} + T_{par} \\ T_P &\geq T_{ser} + \frac{T_{par}}{P} \end{aligned}$$

⁴In 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.

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

Plugging these relations into the definition of speedup, we gave before, yields **Amdahl's Law**:

$$S_P \leq \frac{T_{ser} + T_{par}}{T_{ser} + T_{par}/P}$$

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

$$\begin{aligned} T_{ser} &= f \cdot T_1 \\ T_{par} &= (1 - f) \cdot T_1 \end{aligned}$$

Substituting these into speedup equation:

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

Speedup is *limited by the fraction of the work that is not parallelizable*, even using an infinite number of processors[11].

5.1.4 Results: gathering and evaluation

From all `.txt` files, containing time measures for all the execution that have been run from tests, we have to work on some results and calculations.

In particular, implemented Python scripts ⁵ provides a tool to:

- First of all, output all necessary `.csv` containing all averages, of the times got by the multiple runs for a certain input ⁶;
- Then from all those average Completion times, in `.csv` format, another script computes all **Speedups**;

⁵See Chapter 2.

⁶Remember that outliers values are rejected, and mean values are computed on remaining values.

- Finally, the same script that computes speedups, outputs plots on most significant results.

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

Remembering what we introduced in Section 5.1.3, for a certain program, the speedup, in brief, is the ratio between the time spent in sequential (or single worker) version and the time spent on parallel version, having P workers:

$$speedup = S_P = \frac{T_1}{T_P}$$

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

- T_1 , ie the sequential version, is the case in which CUDA Streams aren't used⁷. 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 item as input, this means sending only a small chunk at time to the device, thus using only a small amount of computational resources at time;

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

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, ie the target numbers of SMs we want to make busy at computation peak time.

So the speedup, obtained from those two implementations, will also give us an indicator

⁷More precisely only default stream is used, instead non-default streams aren't used.

on how many SMs will be 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 to see if those channels will successfully overlap, hiding data transfer from/to device, executing multiple kernels at the sametime and other possible latencies. So, the computed speedups are:

1. $speedup = S_3 = \frac{T_1}{T_3}$ as we mentioned before, this is a base case for CUDA streams use;
2. $speedup = S_{\#SM} = \frac{T_1}{T_{\#SM}}$ this is the special case that aims to make 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 are completion times, upon which we base analysis. Suppose we focus on a portion of input/output stream made of n elements:

$$T_{Tot} = n \cdot (T_{InStream} + T_{H2D} + T_{Kernel} + T_{D2H} + T_{OutStream})$$

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, ie latency from the first available item on the input stream, until the last result item is sent to the output stream (this involves both host and device elapsed times);
2. $T_{Instream}$ this is the time it takes to get a chunk from input stream of size $k \ll n$ ⁸ (host measure);
3. T_{H2D} is the time spent in transferring a chunk of k items from host to device (device time);

⁸It's a buffer of k floats in Simple-computational kernel, while it's a matrix of k items in Matrix Multiplication kernel, or an image with k pizels

4. T_{Kernel} is the time needed by the GPU to compute a certain kernel on k elements (device time);
5. T_{D2H} is the time spent in transferring back k -sized results from device to host (device time);
6. $T_{Outstream}$ the time it takes to send results to the output stream (host measure).

Obviously measures and all considerations are based only on device completion times, ie points 3,4 and 5 of the above list.

We can reduce our test to an important assumption: we don't know how much it is n (the real length of the input stream), that's why we focus our measures on a reasonably long portion of the input stream, ie $(l \cdot k) < n$ elements will be the stream limit.

So we'll focus our attention exclusively to:

$$T_{Device} = l \cdot T_{Comp}$$

where

$$T_{Comp} = k \cdot (T_{H2D} + T_{Kernel} + T_{D2H})$$

In other words T_{Comp} is the time needed to send a chunk of k) items to GPU, make computations on them and then send back to host k) results.

As we told before, host elapsed times aren't in the study domain of this thesis. So, what we want to parallelize is T_{Device} , in particular we want all l chunks to run in parallel. Clearly, we're also assuming that $l > \#SM$, ie our tests will run on a limited input stream, but still ensuring that we have several tasks to send for each CUDA streams, 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 law, showed in previous section. Recall that according to Amdahl, serial code can be identified in two categories: serial fraction and parallelizable fraction.

Since we're only focusing on T_{Device} and, since we wish to parallelize all of it, ideally

all operations included in this completion time may be considered parallelizable. This leads trivially to: $f = 0$ and $(1 - f) = 1$.

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

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

And since we have a limited amount of resources, given by the Streaming Multi-processors number, we can conclude that the maximum speedup we can achieve is: $Sp_{\#SM} \leq \#SM$. This formally proves our expectations.

In the next sections will be reported 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).

5.2 Simple-computation kernel

For the computational-bound kernel, for each type of input dataset, we identified three different values of kernel iterations number, 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 Cosine).

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, ie **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, computational-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.

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

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 length for both devices.

1. Classic data parallel approach

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

To this aim, we should assume that, instead of having as input a stream of items, we have a quite long array of data, in particular consisting of floating point numbers.

In Table 5.1 we show length that was used, in this case is the number of items grouped in a data structure.

Note that, 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.

2. **Streaming parallel with smaller buffers** Here we're facing the Farm parallel pattern for GPU, but with smaller buffers. 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 that each called kernel could fill fully or partially a SM, we had to take into account of:

- How many items we push for each kernel execution;

- Consequently, how many thread blocks our kernel will issue.

These choices followed from our devices features, 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 thread blocks active, and so running, on a certain Streaming Multiprocessor. However we can hit this limit only when we have a poor amount of threads in each block or a consistent quantity of thread blocks.

These aren't our case, since we decided to use the maximum number of threads in a block, ie for x block dimension and the lowest possible size in grid.

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.

The execution configuration for smaller buffers is such that we want to have 1024 buffers and kernel configuration such as $\langle\langle\langle 1, 1024 \rangle\rangle\rangle$. 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 launch a lot of buffers to device, ie enough to hopefully fill all SMs. We recall that the number of chunks will be 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 any non-default CUDA stream, so we'll have serial and synchronous data transfers. Kernel

are still an asynchronous call, but immediately after we want to have data back from device and this means have a `cudaMemcpy`, ie a synchronous call (w.r.t. the host);

- **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 stream is the classic configuration 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 CUDA Streams. But in our case it's necessary to try to achieve the expected speedup with respect to the version without non-default streams (we'll sometimes refer to as "zero version").

Clearly, at a certain time say t_i , we can have at most two data transfer but there's no limit on kernel calls, clearly they will be effectively executed as long as there are available resources on the device. So this is the key point why all of kernel launches, at peak CUDA stream filling, should be spread in SMs, as soon as requested resources will be available.

3. **Streaming parallel with bigger buffers** This tests setup has similar premises to the one for smaller buffers, here the only thing is changing is buffer length set to 2048.

Thus, having always `blockSize=(1024, 1, 1)`, the code will set `gridSize=(2, 1, 1)`, this is because we'll have two blocks, each covering calculations on one half of the buffer. This can sound as having better performances, with respect to smaller buffers, 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 buffers.

Clearly we repeated the above mentioned amount of CUDA streams, so for this

configuration we executed 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 buffers

All collected elapsed times for 1024-long buffers 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 (eg 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 clear sign that this type of kernel is computation-bound.

Furthermore, input stream lengths grows by a factor of 2.

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

Again this confirms that, no matter how many elements 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.

This leads to compute speedups, following the approach explained in Section 5.1.4.

	Tesla P100 (zero stream)		Tesla M40 (zero stream)	
M iterations	Event Times	N elements	Event Times	N elements
10000	4622.86	57344	693.747	24576
400000	181465.333		27453.933	
800000	361199.666		54888.933	
10000	9294.18	114688	1382.363	49152
400000	281507		54901.6333	
800000	407750.666		109783.666	
10000	10217.933	229376	2765.323	98304
400000	407779.666		109799.333	
800000	815513.666		219553	
10000	20433.633	458752	5528.96	196608
400000	815561		219589	
800000	1631013.333		439097.666	
10000	40865.4	917504	11058.433	393216
400000	1631096.666		439192.333	
800000	3261986.666		878195	
10000	81731.733	1835008	22112.6	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	N elements	Event Times	N elements
10000	104.772	57344	30.6074	24576
400000	3968.913		1178.72	24576
800000	7818.843		2355.413	24576
10000	205.729	114688	60.208	49152
400000	7828.193		2358.656	49152
800000	15691.833		4714.36	49152
10000	407.712	229376	119.3446	98304
400000	15687.966		4715.123	98304
800000	31396		9425.92	98304
10000	803.223	458752	238.249	196608
400000	31422.033		9429.89	196608
800000	62818.7		18853.966	196608
10000	1619.586	917504	475.590	393216
400000	62793.4		18856.8	393216
800000	125575		37705.666	393216
10000	3229.063	1835008	949.497	786432
400000	125547.666		37711.9	786432
800000	251503		75445.266	786432

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).

	Tesla P100 (56 Streams)			Tesla M40 (24 Streams)		
M iterations	N elements	Sp(3)	Sp(56)	N elements	Sp(3)	Sp(24)
10000	57344	5.337	44.122	24576	2.976	22.665
400000		5.246	45.721		2.963	23.291
800000		5.221	46.196		2.963	23.303
10000	114688	5.366	45.176	49152	2.967	22.959
400000		4.069	35.960		2.962	23.276
800000		2.947	25.984		2.963	23.287
10000	229376	2.988	25.061	98304	2.970	23.170
400000		2.986	25.993		2.963	23.286
800000		2.986	25.975		2.962	23.292
10000	458752	2.988	25.439	196608	2.967	23.206
400000		2.986	25.955		2.963	23.286
800000		1.940	25.963		2.963	23.289
10000	917504	1.635	25.231	393216	2.967	23.251
400000		1.689	25.975		2.963	23.290
800000		2.861	25.976		2.963	23.290
10000	1835008	2.998	25.311	786432	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: Here are showed speedups for all data sets of simple-computation kernel. Results are reported for both devices.

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:

$$Sp(3) = \frac{T_1}{T_3} \text{ and } Sp(SM) = \frac{T_1}{T_{SM}}.$$

To have an overall view on speedup we also present plots for both P100, in Figure 5.1, and M40, in Figure 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 input stream length 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 streams⁹.

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*", ie the time interval it takes for the program to fill buffers 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 and, furthermore, they don't have a bad impact in this particular application.

A good signal 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 have the maximum work rate.

⁹More precisely this gain is bounded to the number of Streaming Multiprocessors the GPU exploits.

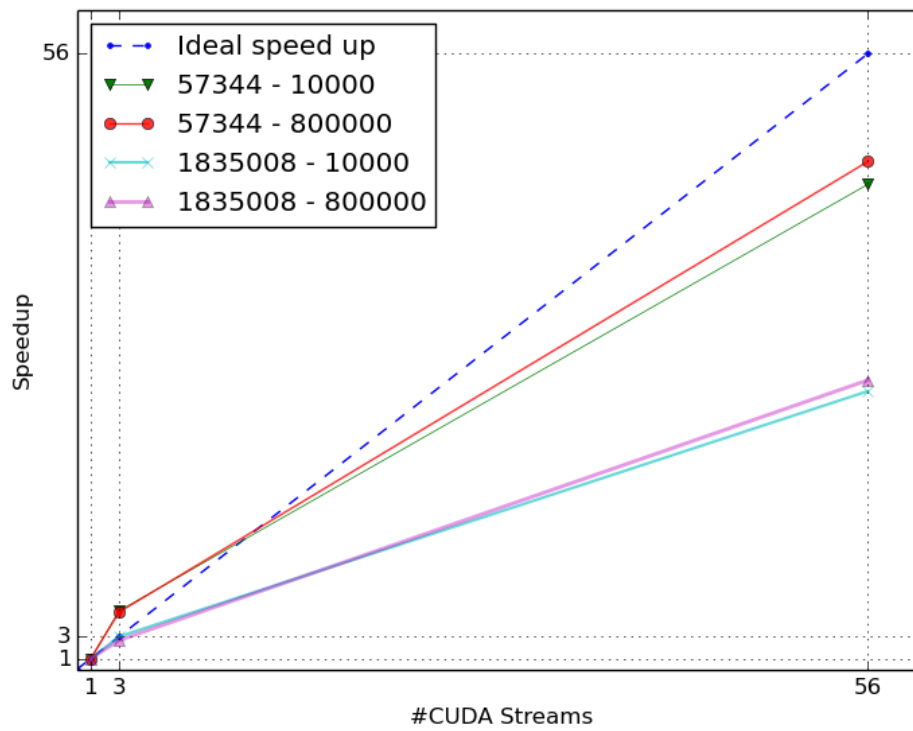


Figure 5.1: Speedup for 3 and 56 CUDA streams on P100 device.

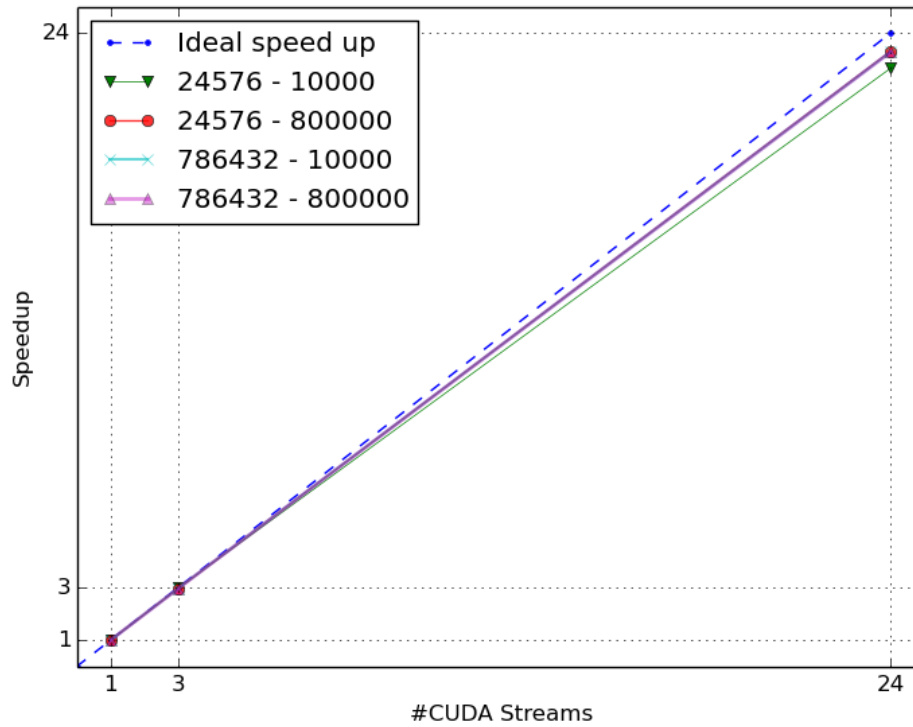


Figure 5.2: Speedup for 3 and 56 CUDA streams on M40 device.

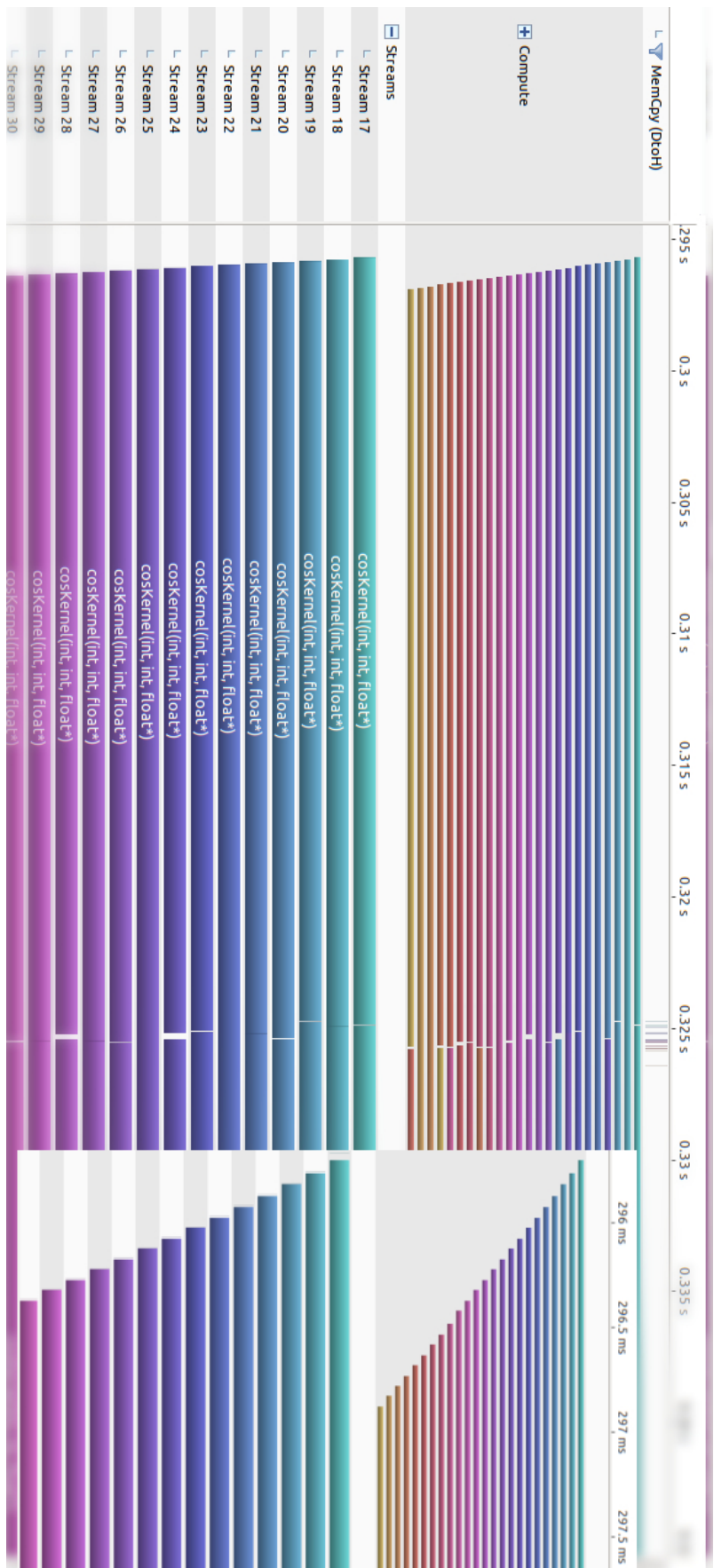


Figure 5.3: Profiling for an example execution: limit for input stream 786432, kernel iterations 10 000, 24 CUDA streams, on M40 device.

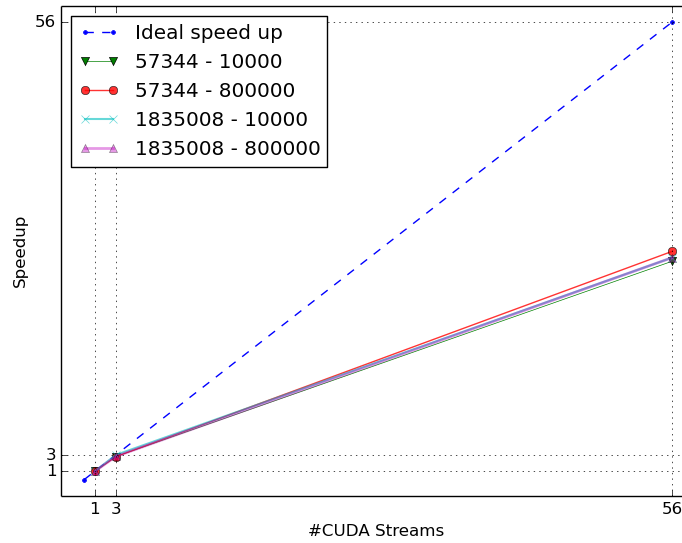


Figure 5.4: Sublinear speedup in bigger buffers execution, the performances drop by a factor of 2.

Bigger buffers

As expected, we get about half the ideal speedup.

This is because chunks of 2048 should take up the maximum possible for 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 buffers, 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

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 floats that were accumulated in buffers, here we have an *input stream of matrices*. In particular as soon as we have two available input matrices (say A and B), 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 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 $\text{BLOCK} = 32$ and $\text{GRID} = (N/\text{BLOCK}) + \text{BLOCK} - 1$, so we have:

```
blockSize = (BLOCK, BLOCK, 1)
gridSize=(GRID, GRID, 1)
```

We performed the following executions:

1. **Classic data parallel approach** Again, we have to put a limit on input stream, so that we can measure completion times. Note that the length limit for input, doesn't straightly correspond to the effective number of incoming matrices, but to the number of matrix multiplication that will be performed.

So, say we have l as limit, it correspond to have $2 \cdot l$ matrices for input stream (A and B) and effectively l result matrices (C).

To test the data parallel version it suffices to treat input/output streams as huge matrices (but we'll have only one matrix both for A, B and C). This means that we'll do computations, no longer on stream of small data structures, but on a unique big data structure.

Tesla P100		Tesla M40	
Mat. Order	In stream limit	Mat. Order	In stream limit
128	225	64	100
256	441	128	196
512	900	256	400
1024	1764	512	784
2048		1024	

Data Parallel		Data Parallel	
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, below Data Parallel correspondent.

So, for example:

- We have two input streams, each has limit to 9 square matrices of order 2 (one input stream is for **A** matrices and one for **B**);
- So suppose that our stream parallel model, sends out one matrix A and one B at time;
- Then we launch the kernel, performing the matrix multiplication, this will give back a matrix result C;
- This means in total we will perform 9 multiplications between couples of matrices, giving 9 result matrices;
- If we consider those 9 small matrices as block matrices, we can combine them into a bigger one;
- This will be equivalent to pick A and B matrices each of order 6;
- note that, for simplicity, we'll choose as input stream limit a square number, so that the equivalent combined data structure will be again square;
- In our example 9 is a square number so that we can obtain as composed matrix dimension $[(2 \cdot 3) \times (2 \cdot 3)] = [6 \times 6]$

In Table 5.5, in the lower portion, are showed the matrices orders we used to test data parallel version for matrix multiplication. By giving these values as (square) matrix dimension and computing only one multiplication between matrices, we'll get the a balanced comparison to some Stream parallel versions.

2. **Streaming parallel** As we mentioned before, we have to put a limit on the stream length for both streams of input matrices.

In the upper portion of Table 5.5 are reported all input stream limits and, for each of them, we test all of matrices order.

For every combination given by the input dimensions, 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 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:

- input streams of matrices have lengths that grow by a factor of 2 and it's easy to see that this makes a proportional increase in completion times, ie even measures grows by factors of 2;
- input matrix sizes again grow of $2\times$ each, but in this case the completion times don't grow proportionally;
- for zero-streams we can see that, as the matrix order grows by a factor 2, the completion time can increase from $\approx 4\times$ to $\approx 7\times$;
- for three-streams we can see that, as the matrix order grows by $2\times$, the completion time can increase from $\approx 5\times$ to $\approx 8\times$;
- finally for SM-streams we can see that, the completion time can increase from $\approx 7\times$ to $\approx 8\times$.

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

Tesla P100 (zero Streams)			Tesla M40 (zero Streams)		
Event Times	Number of Mats	Mat. Order	Event Times	Number of Mats	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 streams), results are reported for both P100 and M40.

Tesla P100 (3 Streams)			Tesla M40 (3 Streams)		
Event Times	Number of Mats	Mat. Order	Event Times	Number of Mats	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	Number of Mats	Mat. Order	Event Times	Number of Mats	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 number, results are reported for both P100 and M40.

The first point tells us that: the elapsed time to send/receive to/from the device, grows linearly with the number of matrices, so this input parameter would not affect performances. Especially, no matter the CUDA streams amount we decide to use, the increase by 2x of matrices quantity, will always give a growth of 2x in 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 matrices order grows (respectively on M40 and P100. In Figures 5.7-5.8, instead, we have similar plots, but for the time changing as the number of matrices increments.

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

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.

This is because, the simpler implementation, at each iteration, spends more time in *global memory/registers* transfers than in effective calculations. So that's why this kind of matrix multiplication is considered memory-bound.

So, the more elements a matrices have, the more data transfers (internal to the device) the GPU will have to perform and 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 reflected on GPU Farm approach. All speedups are listed in Table 5.9.

From those results, we can mainly observe that:

- $Sp(3)$ gives results near to the expected value, ie ≈ 3 for the smaller matrix sizes (128-256 for the P100, 64 for the M40);
- $Sp(SM)$ gives a really poor gain w.r.t. $Sp(3)$;

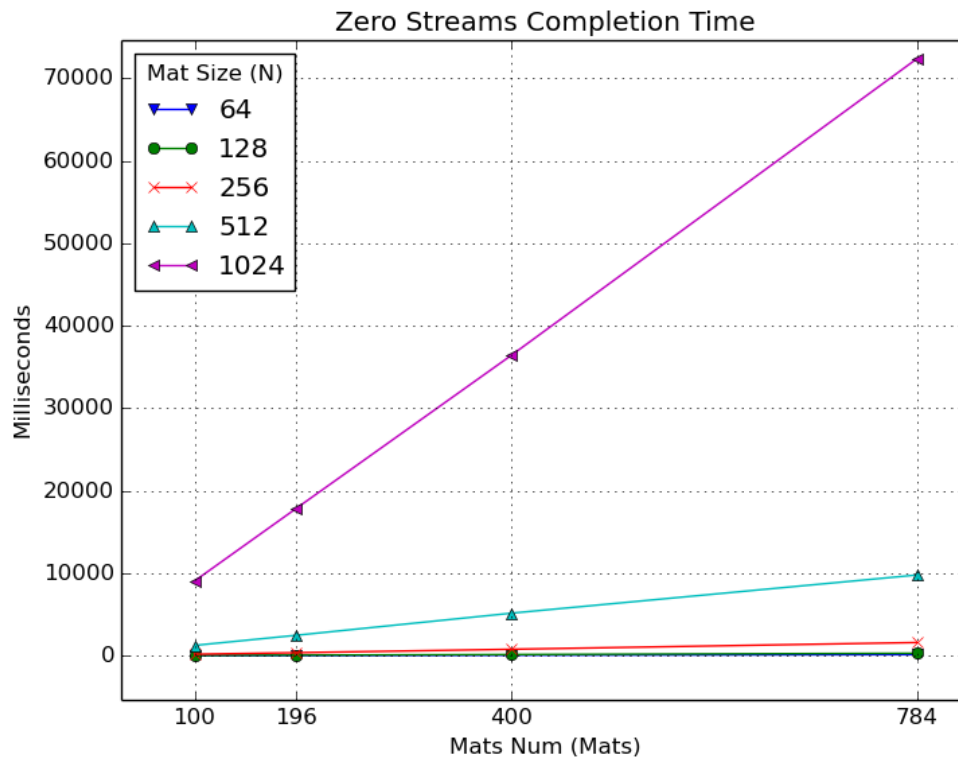


Figure 5.5: Completion Time as the matrix order changes on M40.

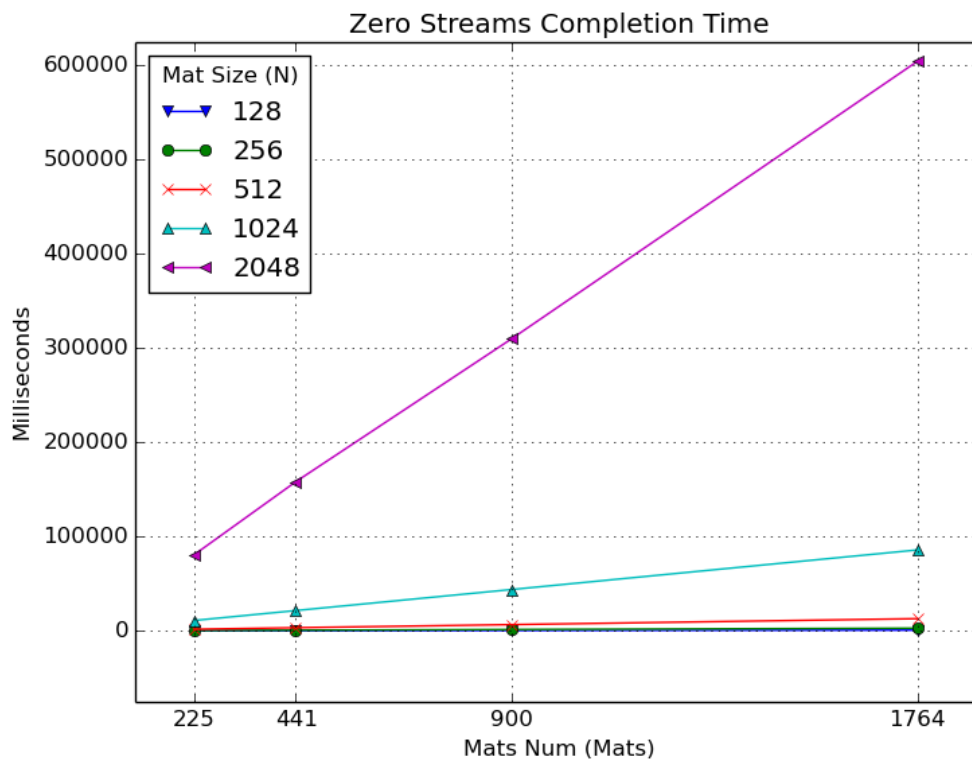


Figure 5.6: Completion Time as the matrix order changes on P100.

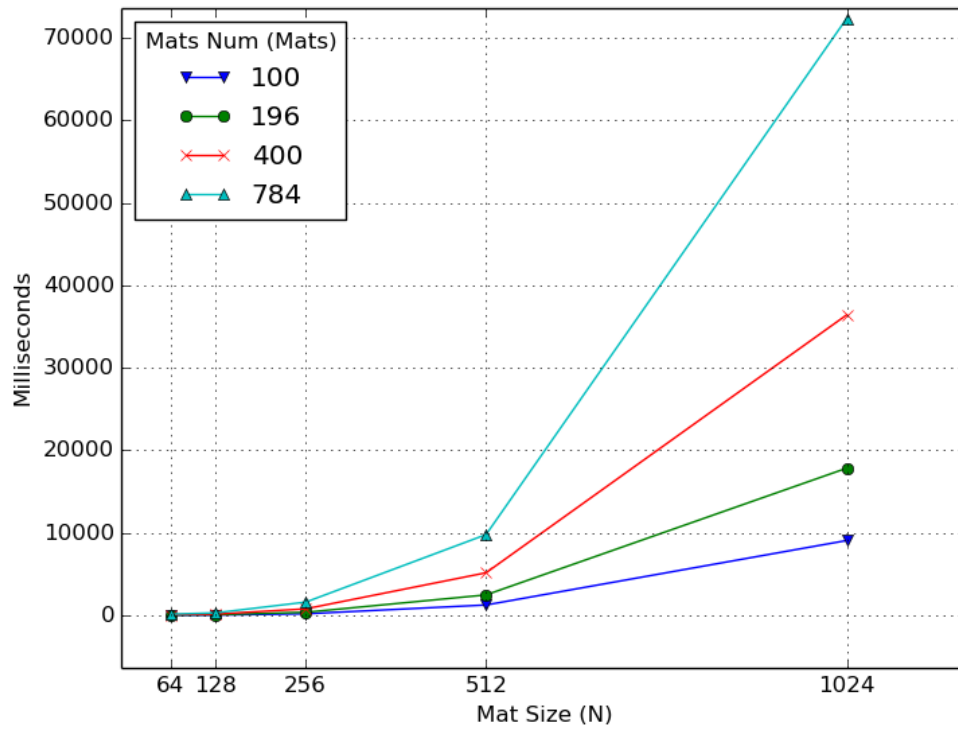


Figure 5.7: Completion Time as the number of matrices changes on M40.

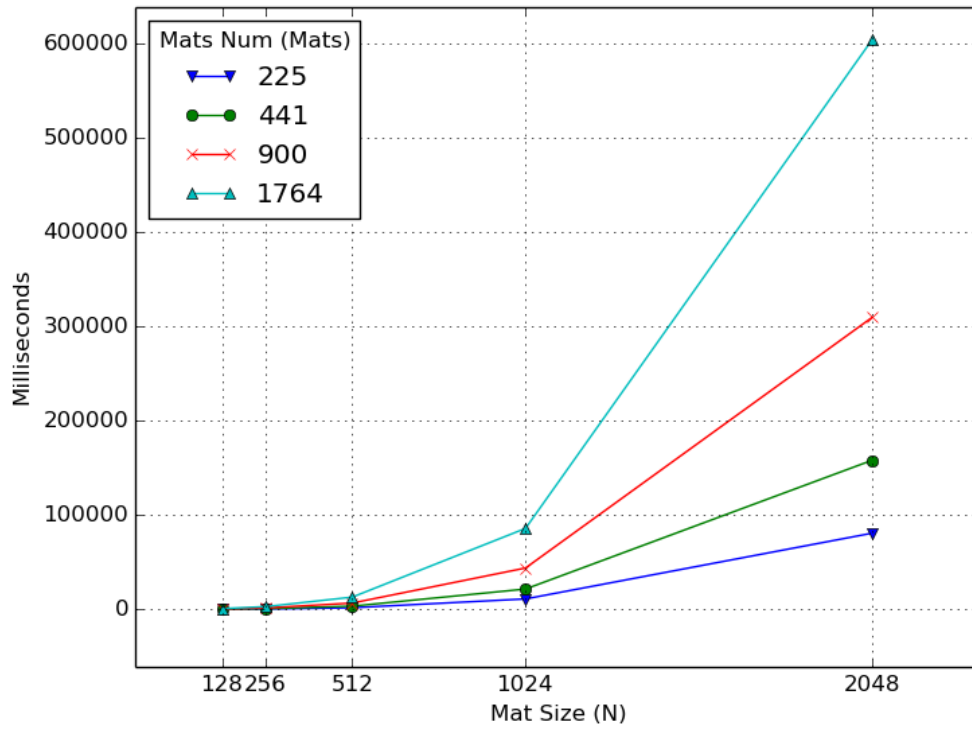


Figure 5.8: Completion Time as the number of matrices changes on P100.

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.

- all speedups degrade to ≈ 1 as the matrix order grows.

This behavior translates in the following: when the matrices get bigger, even if CUDA Streams push to have more simultaneous mat-mul, we'll have a lot of active threads (and so Multiprocessors) busy and probably mainly stalled on gathering floats from global memory.

This, in fact, inevitably leads to a very limited amount of gain, even when using a lot of CUDA streams. Furthermore, those results tell us that we will fit in Multiprocessor less matrix multiplication than we wish ¹⁰.

Furthermore, the necessity of having multiple blocks on grid, for this specific use case, translates in a monopolization of SMs resources by a small amount of kernel calls.

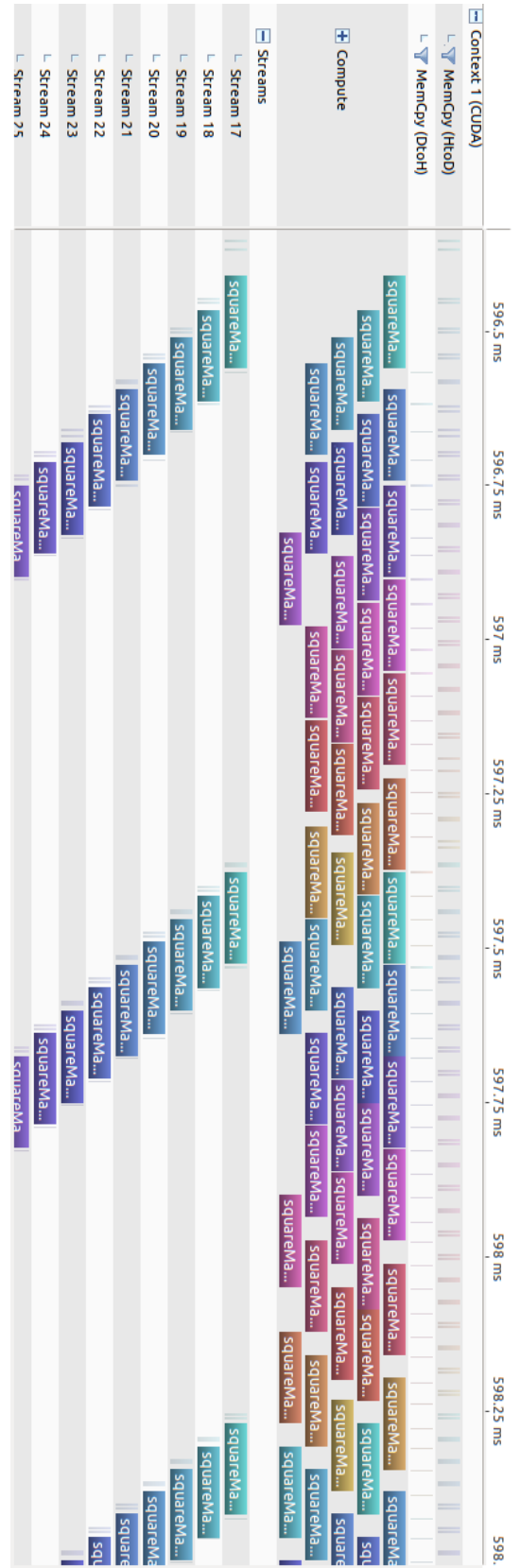
Profiling the application for some key data set we can inspect the above described facts and the relative reasons.

In Figure 5.9 we can have a visual cue on what is happening during an execution of mat-mul on M40, with 24 CUDA streams, having an input stream of 784 matrices of sizes: 64×64 (Figure 5.9 (c)), 256×256 (Figure 5.9 (b)), 1024×1024 (Figure 5.9 (a)). Analyzing the figure we can see that the amount of overlapping, between operations in different streams, is quite limited in general, this just confirms what we saw from speedups.

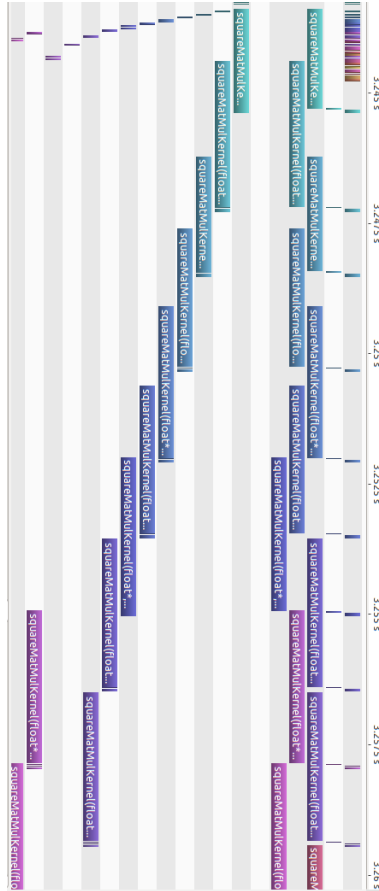
From the above mentioned pictures, we can observe that in 64-sized case we're having a slightly better overlapping and a little more kernels running at the same time. In this case this may happen because grid and block sizes are smaller for each kernel launch, so this allow to have multiple kernels fit 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

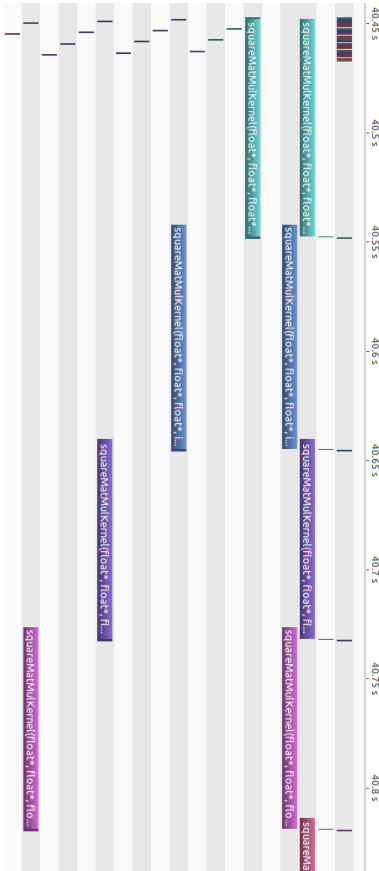
¹⁰Clearly this strictly holds for the type of matrix multiplication we implemented.



(c) Matrix size 64



(b) Matrix size 256

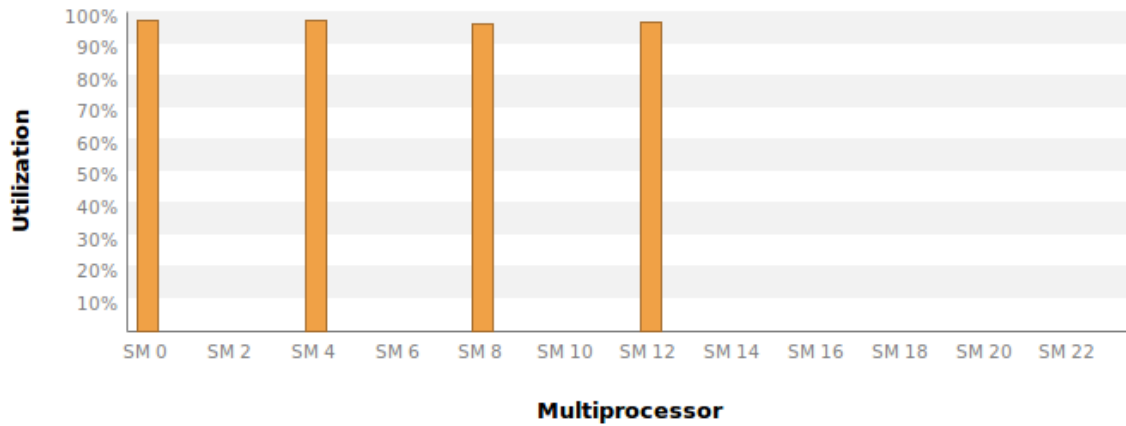


(a) Matrix size 1024

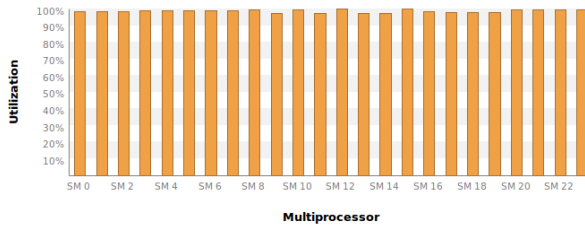
Figure 5.9: NVIDIA Visual profiler generated *timeline* on M40, using 24 CUDA streams and running code for 784 matrices.

i Multiprocessor Utilization

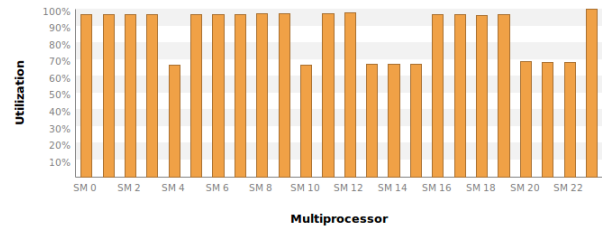
The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each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



(a) Matrix size 64



(b) Matrix size 1024



(c) Matrix size 256

Figure 5.10: NVIDIA Visual profiler generated *timeline* on M40, using 24 CUDA streams and running code for 784 matrices.

blocks¹¹.

This means that, in the two latter cases, we don't even have all blocks, from a single kernel launch, fitting in all the SMs. If this may theoretically give a full occupancy of the GPU by a certain kernel, from the other side means saturate the cores without permitting other launches to fit, until the residing kernels ends or, at least, some resources are freed.

We can confirm this fact by looking at the occupancy graphs (generated from Vi-

¹¹This 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

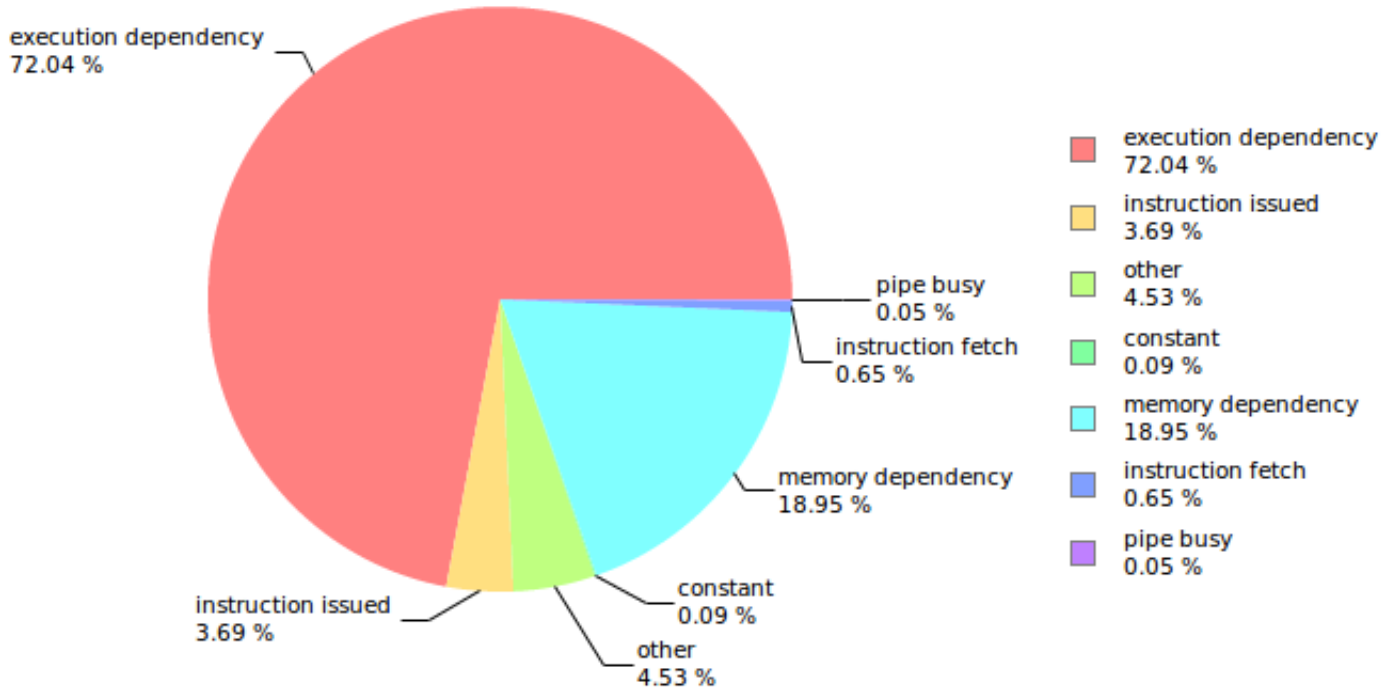


Figure 5.11: Nsight profiler execution on M40, 24 CUDA streams, 784 matrices of size 256. This graph gives the types and amounts of latencies inside mat-mul kernel execution.

sual 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 (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* [8, 12], it strongly depends on the kernel nature.

On the other side as we decrease the matrices size we can’t exploit enough resources. As we can see from Figure 5.9 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 is idle on:

- execution dependency¹², ie an input required by the issued instruction isn't yet available;
- memory dependency¹³, 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.

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, ie 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 test, the kernel execution configuration was set up as follows: assume N to be the image resolution (ie $N \times N$, then `BLOCK` = 1024 and `GRID` = $(N/BLOCK) + BLOCK - 1$, so we have:

```
blockSize = (BLOCK, 1, 1)
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.

¹²Execution dependency stall can be potentially reduced by increasing instruction-level parallelism Instruction-Level Parallelism (ILP), ie we can improve parallelism by increasing the amount of instructions for each thread. This can translate in less threads having a greater work-load each.

¹³ Memory dependency stall type can potentially be reduced by optimizing memory alignment and access patterns.

Tesla P100 & Tesla M40	
Img. Order	In stream limit
128	64
256	256
512	1024
Data Parallel Tesla P100 & Tesla M40	
	1024
	2048
	4096
	8192

Table 5.10: Input dataset for Image Processing kernel. Above Stream parallel configuration, below the relative Data Parallel.

In particular the Data Parallel approach will be tested giving a single image of large dimensions, instead of a stream of small images.

Again we're considering a single big image, as formed by merging smaller ones. Similarly to matrix multiplication case, we chose, as input stream limits, square numbers to make possible a balanced comparison to the correspondent square picture for data parallel.

In the lower part of Table 5.3 we show the order of dimension of the images used to compare to some streaming parallel cases.

2. Streaming parallel

As previous cases of Farm parallel pattern, we have to put a limit on the stream length for input stream of images.

In the upper portion of Table 5.10 are reported input stream limits and, for each of them, we test the three types of picture dimensions. Note that those images are

square, so, for example, a size of 128 stands for a picture of (128×128) resolution (ie 16 384 pixels).

For every combination given by the input dimensions, 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 worst 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 the input stream of images grows in length by a factor 4, in fact completion times follow this trend by increasing $\approx 4\times$ proportionally with input limits. Instead, fixed a certain number of images as input stream limit, the image size grows by a factor 2. This time this lead to an increasing of completion times by a slightly bigger factor, ie $\approx 3\times$.

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.

This means that we almost have no overlapping, we expected a similar behavior though.

CUDA Streams	Img Number	Img 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)	
Img Number	Img 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 are showed speedups for all data sets of image processing kernel. Results are reported for both devices.

5.5 Results Summary

Merging all results we obtained, we can state that, under specific assumptions and adjustments, *Farm parallel pattern can give a speedup really near to linear.*

We can observe this behavior especially in computational-bound case, where we're above the ideal speedup for a small quantity. 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 buffers need to fill up, until we reach the peak occupation for CUDA streams and so for SMs;
- 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, ie

we've $> \#CopyEngines(= 2)$ simultaneous requests.

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-version¹⁴:

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 $Sp_{\#SM} \leq \#SM$.

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.

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 (in SM-CUDA-streams setting).

This is what we expected, since, the almost-linear speedup we obtained, means that we successfully overlapped different executions for small chunks. Formally, suppose to consider the amount of input data already splitted in k parts and suppose data parallel version's device completion time is give by the following formula:

$$T_{DP} \approx k \cdot T_{items}$$

¹⁴We 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.

		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.

where T_{items} is the time needed to compute one of the k portions of data. For stream parallel version we'll have instead

$$T_{SP} \approx \Delta_t + \frac{k}{\#SM} \cdot T_{items}$$

where Δ_t is a not predictable overhead, given by the temporal difference $t_r - t_0$, t_0 is when initially the first item is sent for the first CUDA stream.

While t_r is the moment when the first element arrives for the $r - th$ stream.

In this Δ_t we may also think to include other overheads due to small amount of time in which we have an imperfect streams overlapping. So, assuming Δ_t to be negligible with respect to the time spent in computations, we can conclude:

$$T_{DP} \approx k \cdot T_{items} \leq \Delta_t + \frac{k}{\#SM} \cdot T_{items} \approx T_{SP} \Rightarrow T_{DP} \approx T_{SP}$$

- **Matrix Multiplication kernel**

In M40 Table and P100 Table we'll 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.

For Completeness in M40 Table 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 $T_{dataParallel} / T_{\#SMstreams}$, and as we can see that from Stream Parallel version we have a gain of $\approx 20\times$.

This singular behavior again depends on the memory-bound nature of this implementation.

Having a huge data structure, computing simple operations on a huge amount of items residing in global memory, inevitably leads to a over-occupancy in SMs (mainly due to latencies and thread stalls), dropping dramatically performances.

Mat Number	Mat Order	56 Streams	Event Time	Data Par Mat Order
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 is of stream version is considered). Results are reported for P100.

- **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 as an issue the latency due to diverging flows.

Mat Number	Mat Order	Zero Streams	24 Streams	Data Parallel	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 is of stream version is considered).

Results are reported for M40.

		Tesla M40 (24 Streams)		Tesla P100 (56 Streams)		
M40 Img Number	Img Size	Streams 56	Data Par	Streams 24	Data Par	Img 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 vs. stream parallel comparison for image processing kernel. Results are reported for both devices.

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 concept of normal GPU use, we founded our attempt on the increasing and pervasive concept of General-Purpose computing. Nowadays it's a common practice to use the high parallelism and huge computational power of GPUs as co-processors, even if it isn't strictly for graphical problems.

Also research moved, in last years, the focus on problems that generally are assigned CPUs. Clearly, in General Purpose (GP) it's easy to spot applications that are clearly embarrassingly parallel; we recall that GPUs are mostly well suited in data parallel approaches.

However, there are many others problems that are really far from data parallel. Again, GP-GPUs demonstrate a fair behavior (with some adjustments) in some of those cases. So it makes perfectly sense to inspect for new non-data parallel applications to fit in

GPU model, to exploit its good computation potential.

6.0.2 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 GPU 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, implement and then test our model.

Once we had an overall view on tools and NVIDIA GPUs architecture, we had the knowledge to the next step, ie to imagine a Farm parallel pattern in a graphic processor. Obviously some key problems have arisen:

- Handle the difference on input/output, ie streams of items instead of data structures;
- Handle how to group and send data to device;
- Define the dimension of data chunks;
- 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 capabilities of the GPU at their best.

The first two points were accomplished by thinking to a system of *accumulator buffers* that was sent to device as soon as they were filled by the input stream. This was mainly designed for the simple-computation kernel, but it applies to all those scenarios where

we have an input stream made of simple items (eg floats, integers, etc.).

Instead for the other two kernels we simply had to test the Farm parallel pattern on small matrices or small images, that straightly arrived from the input stream, so they were ready to be directly sent to device.

The third point is again bounded to all applications having simple items as input stream. The dimension of buffers was determined by both *empirical approach* and a study on *best practices* for GPUs, as we showed in Chapter 3, most of this reasoning relied on *occupancy* evaluations.

However, we also showed, 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 types of bottleneck in kernels: **memory-bounded** code and **diverging flows**.

Those concepts are straightly linked to the problem of last point.

The fourth and fifth points are strongly related to a powerful programming technique in CUDA: ***Asynchronous calls*** and ***CUDA Streams***.

We recall that here asynchronous is from a device side point of view, with respect to the host. That is, host can continue executing his code, after invoking a memory copy (or any other call that is generally blocking). Any asynchronous call will be forwarded to GPU that will "silently" work, sending back eventual results to CPU.

This means, in general, to have some synchronization at some point. Often CUDA codes with asynchronous calls are implemented to introduce *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 at the same time, that sometimes led to overwriting data from another non-default stream.

This problem mainly showed up in device memory: in host side, from the beginning, we had foresee the need of sufficient host memory locations for all streams. In device side, the possible cases were:

- Have a single chunk of space in global memory and use some explicit synchronization, that's the most used approach;
- Reserve several chunks, as many as created CUDA streams amount, and not use any kind of explicit synchronization.

The first approach can be used in data parallel approach without introducing a big amount of overhead, but in a stream parallel context it can cause a performance drop. So we decided to use the second approach. The first impression could be to risk for a saturation in global memory, but in our case this didn't happen, since we were using relatively small chunks of data (even if they were as many as number of CUDA streams).

This doesn't mean that it cannot exist any stream parallel application where synchronization may result in an advantage¹.

Furthermore trying a *hybrid approach* could be a starting point for future works, where hybrid means having less allocated global memory locations (than CUDA Streams number) and introduce only few explicit synchronizations.

6.0.3 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 a "good behavior" in terms of performances, while we expected worse completion times and speedups from the

¹Maybe to hide some other computations that are happening in the same time. Again it's a matter of experimenting according to the type of problem we're facing.

second kernel type and even from the third one.

The next step has been to build tests, gather results and make the following considerations.

Tests have been set up in such a way to observe the performances of our model in different situations, such as varying the chunks size and varying the pressure on CUDA Streams, ie the number of tasks globally sent on a certain non-default stream.

What we wanted to mainly measure was:

- the global time spent to "consume" an input stream by transferring data one chunk 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 chunks transfers and computations of the kernel. This is what we considered the *parallel version*, in other words the approach using CUDA Streams (three or equal to the number of SMs);
- the completion time of the relative data parallel version, ie assuming that all our input stream is grouped in a single data structure.

6.0.4 Results and considerations

The results we obtained are just as we expected:

1. Simple-computational kernel showed a great ability on overlapping, clearly with some appropriate adjustments. We get the expected speedups and the version with the maximum number of CUDA Streams² performs almost as the data parallel version;

²That is the version with an amount of non-default streams equal to the number of Streaming Multi-processors.

2. Matrix multiplication kernel showed a low ability on overlapping, especially as matrices size increased. We get poor speedups and the version with the maximum number of CUDA Streams performs quite bad with respect to the data parallel version;
3. Image processing kernel showed an almost inexistent ability on overlapping. We get no speedups and the version with the maximum number of CUDA Streams performs really far from the data parallel version.

From those results we understand that we have the best gain when we have long computations on each single chunks. Even if host/device data transfers introduce a not-negligible overhead, in Farm pattern, we're carrying small groups of items.

Furthermore, these small groups aren't available all at the same time, they arrive one at time, as they're generated from an input stream. This leads potentially to a low data transfers overlap, just for a timing matter.

That's why we should mostly rely on overlapping kernels, as they should lasts longer than memory copy³ and so we've more chances to achieve an overlap.

About this consideration we recall text.

In fact, most of the problems in performances appeared 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 this behavior can even degrade as the portions of data sent to the GPU grows in size, as in our Matrix Multiplication case, because we had a high number of thread blocks occupying all hardware resources, for a single matrix multiplication. This led to longer kernels, but completely monopolizing Multiprocessors.

³This isn't a rule, it just often happens, as in our applications. There may still be cases in which this statement is false.

6.0.5 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-computational intensity scenarios;
- We get the best advantages from parallelizing executions, more than 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 above requirements may translates in a quite challenging effort, especially in evaluating problems, experimenting and profiling performances, more than in implementation difficulty.

This especially holds in all those cases were we have memory-bound problems. But in this case we some chances of future workaround:

- 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)[[12](#), [8](#)];
- exploiting shared memory, it has smaller dimensions but it's much more faster than global memory.

These stratagems may expand to a lot of further applications using Farm parallel pattern on GPUs in the future.

For example numerous studies have been made in matrix multiplication to optimize device global memory latencies with shared memory [METTERE UNA FONTE CHE PARLA DI SHARED]. Other studies showed that we can give smaller kernel configurations, in order to make each threads perform several matrix multiplications[8], instead of computing only one element of the result matrix for each threads (as it happens in classical approach).

So by merging in future, those optimizations with Farm parallel pattern may give some interesting results.

Given that this thesis based all hypothesis on equal chunks of work, ie on balanced workloads for each kernel, an interesting further study could be done in those scenarios treating unbalanced chunks of works, leading to different workloads among kernel launches.

Suppose, for example, a scenario where the input stream sends items at fluctuating speeds, so the chunk size may be established according to a time interval instead of a predefined buffer size. This means send portions of items of unknown size to the GPU.

Another assumption on which we based all this study was in having the ready data chunks scheduled to CUDA Streams in a Round Robin fashion. But, according to the treated problem, other scheduling techniques can be adopted and may result in far more efficient spreading of work loads between streams and so giving a better exploitation in Multiprocessors resources.

Listings

2.1	CUDA Strams creation	17
2.2	CUDA Strams and Async example	17
2.3	CUDA Strams destroy	18
4.1	Implementation for Simple-Computation Kernel	45
4.2	Implementation for Matrix Multiplication Kernel, both non-square and square	46
4.3	Implementation for Image processing Kernel (Blur Box Algorithm) . .	47
4.4	Kernel Launch configuration, ie Grid and Block dimensions setting . . .	49
4.5	Data transfer host/device and kernel call, NO-CUDA Streams version .	49
4.6	Data transfer host/device and kernel call, CUDA Streams version . . .	50
4.7	Host side pseudo-code: input stream + kernel launcher function	51
4.8	Optimal Kernel launcher for Simple-Computation kernel, uses APIs to get best Block configuration	53

List of Tables

2.1	GPUs specifics for the two remote machines employed in this project. .	11
5.1	Input dataset for Simple-Computation kernel, these are the input stream length for both devices.	68
5.2	Device completion times for Simple-computation kernel, without using CUDA Streams, results are reported for both machines (P100 and M40).	72
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).	73
5.4	Here are showed speedups for all data sets of simple-computation kernel. Results are reported for both devices.	74
5.5	Input dataset for Matrix Multiplication kernel. Above Stream parallel configuration, below Data Parallel correspondent.	80
5.6	Device completion times for Mat-Mul kernel, without using CUDA Streams (zero streams), results are reported for both P100 and M40.	83
5.7	Device completion times for Mat-Mul kernel, with three CUDA Streams, results are reported for both P100 and M40.	84

5.8	Device completion times for Mat-Mul kernel, with as many CUDA Streams as SM number, results are reported for both P100 and M40.	85
5.9	Here are showed speedups for all data sets of matrix multiplication kernel. Results are reported for both devices.	89
5.10	Input dataset for Image Processing kernel. Above Stream parallel configuration, below the relative Data Parallel.	95
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.	97
5.12	Here are showed speedups for all data sets of image processing kernel. Results are reported for both devices.	98
5.13	Here are showed completion time and speedup for $48 = 2 \cdot \#SM$ CUDA Streams.	100
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.	101
5.15	Matrix multiplication kernel. Comparison between completion times for stream parallel (max stream -56) and data parallel versions (Partial dataset is of stream version is considered). Results are reported for P100.	103
5.16	Matrix multiplication kernel. Comparison between completion times for stream parallel (max stream -24-) and data parallel versions (Partial dataset is of stream version is considered). Results are reported for M40.	104
5.17	Here is showed the data parallel vs. stream parallel comparison for image processing kernel. Results are reported for both devices.	105

Bibliography

- [1] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design: The Hardware and Software Interface*, V Edition, 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*, 2012
- [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] Marco Danelutto, *Distributed Systems: Paradigms and Models*, 2014
- [5] NVIDIA, *CUDA C Programming Guide*, CUDA Toolkit Documentation, from <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2019
- [6] NVIDIA, *NVIDIA Profilers Guide*, CUDA Toolkit Documentation, from <https://docs.nvidia.com/cuda/profiler-users-guide/index.html> , 2019

- [7] 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
- [8] Vasily Volkov, *Better Performance at Lower Occupancy*, 2010
- [9] Mark Harris, *How to Implement Performance Metrics in CUDA C/C++*, NVIDIA Developer Blogs, article from <https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/> , 2012
- [10] NVIDIA, *NVIDIA Library Documentation- Event Management*, from site: http://horacio9573.no-ip.org/cuda/group__CUDART__EVENT_g14c387cc57ce2e328f6669854e6020a5
YEAR
- [11] M. McCool, A.D. Robinson, J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, 2012
- [12] NVIDIA, *CUDA C Best Practices Guide*, <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>, 2019