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

CHAPTER 1

Introduction

In a scenario were 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 **GPU**s made possible to exploit their benefits for a wide range of non-image-processing problems. This is the beginning of **GPGPU**s.

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 ongoings, 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 oftentimes 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

[1] We'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]

$$x_m, ..., x_1$$

the farm with function f computes the output stream as

$$f(x_m), ..., f(x_1)$$

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:

[2] We'll insist on occupancy topic in Chapter 3.
[3] See Section 3.5
[4] In 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 –.

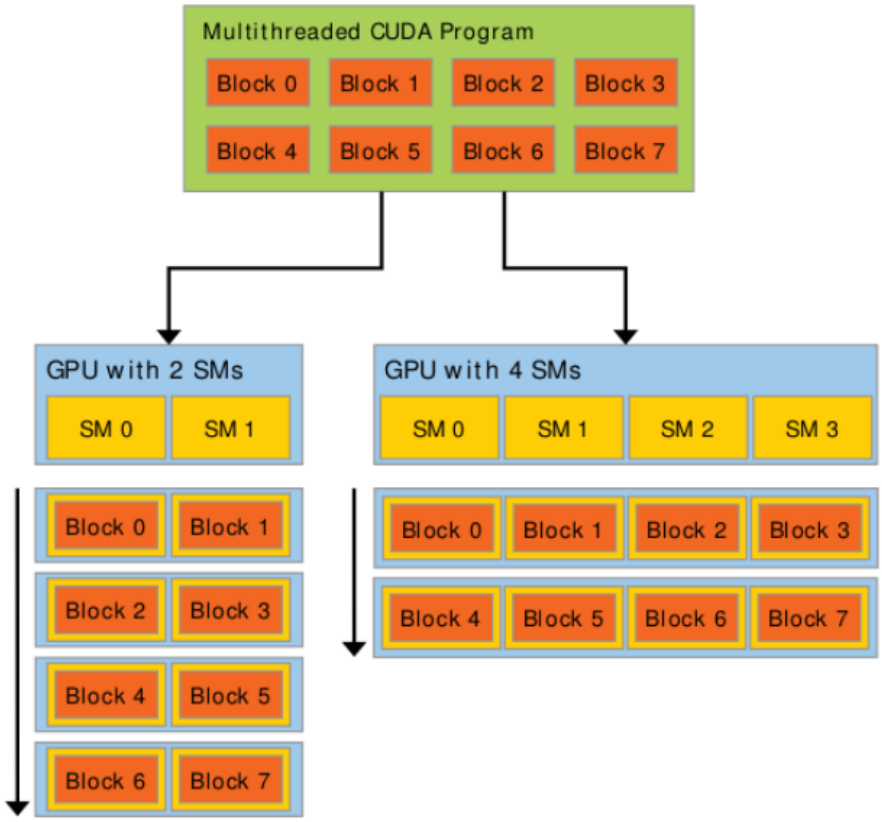

Figure 2.1: GPU scalability.

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 ...
```

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

[2] Or 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.

[3] We'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.

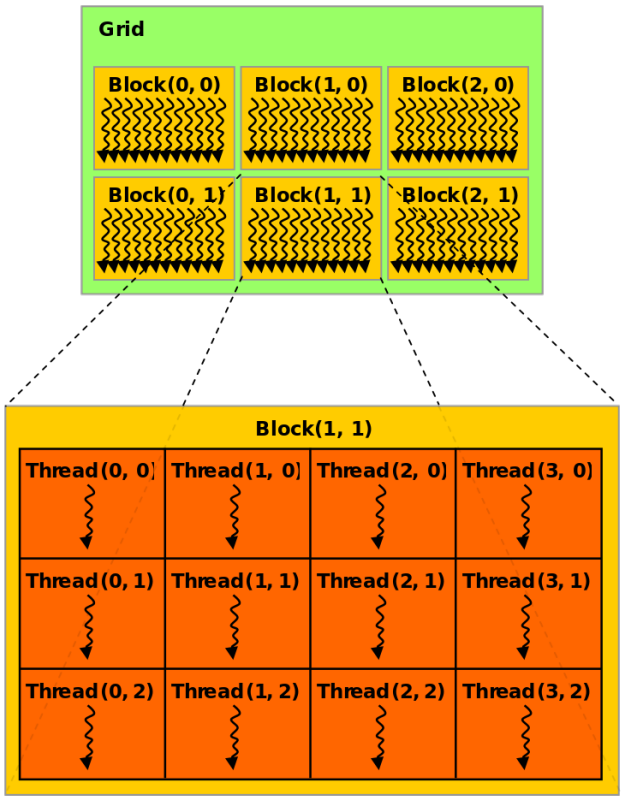

Figure 2.2: Above: a Grid formed by Blocks. Below: a Block formed by Threads.

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

[4] see 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:

```
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*:

```
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 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()`.

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

```
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 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:

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

[7] See subsection 2.3.1

```
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 refactoring [8].
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.

[8] Other 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 shell[9].

`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/

[10] The 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

[1] About 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

$$x_m, ..., x_1$$

the farm with function f computes the output stream

$$f(x_m), ..., f(x_1)$$

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.

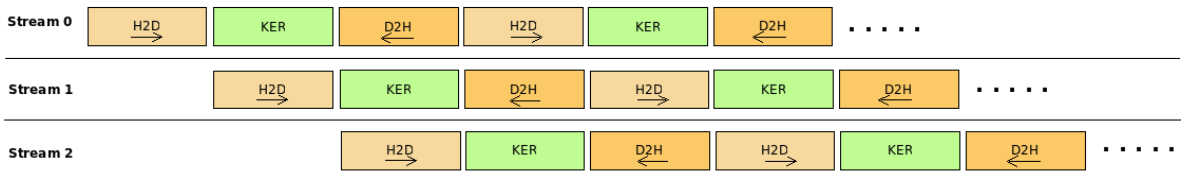

Figure 3.1: Ideal behavior for 3 CUDA Streams.

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.

[2] In 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.

[3] We'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 concepts[4] 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

[4] From CUDA Programming Guide.

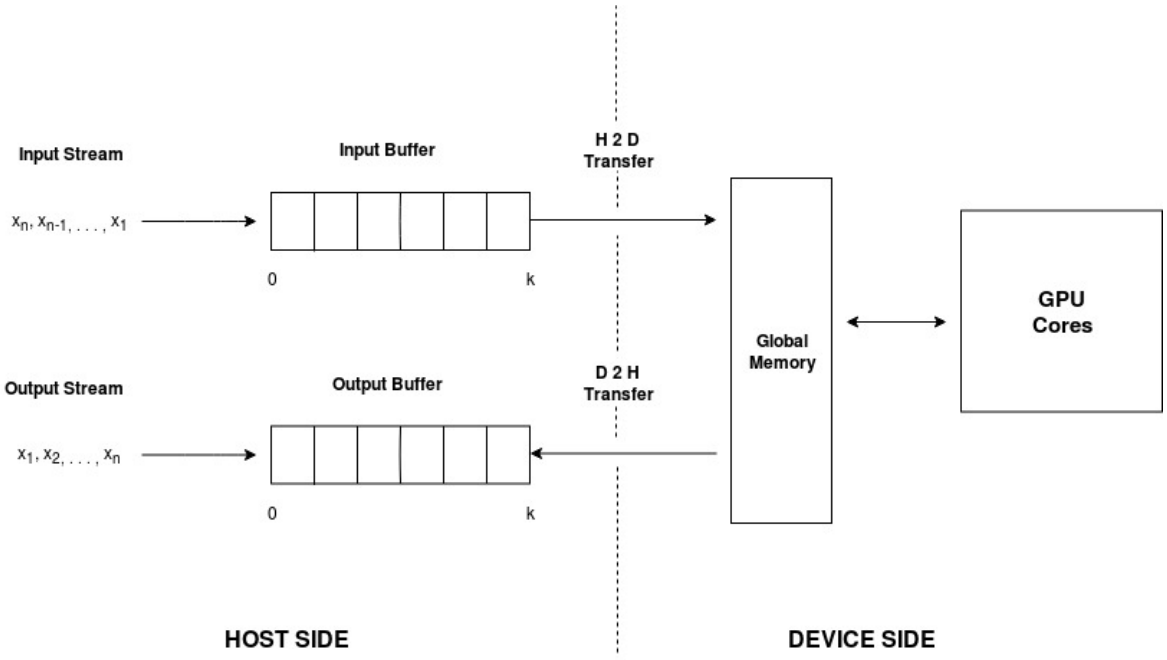

Figure 3.2: Here we have a general and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

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 chunks[5] 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 seems 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 Streams [6] come into play and we used them relying on the following ideas:

1. We have as many streams as Streaming Multiprocessors [7] and, at any given time, each of them hopefully issues a data transfer or a kernel executions;

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

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

[6] Don'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).

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

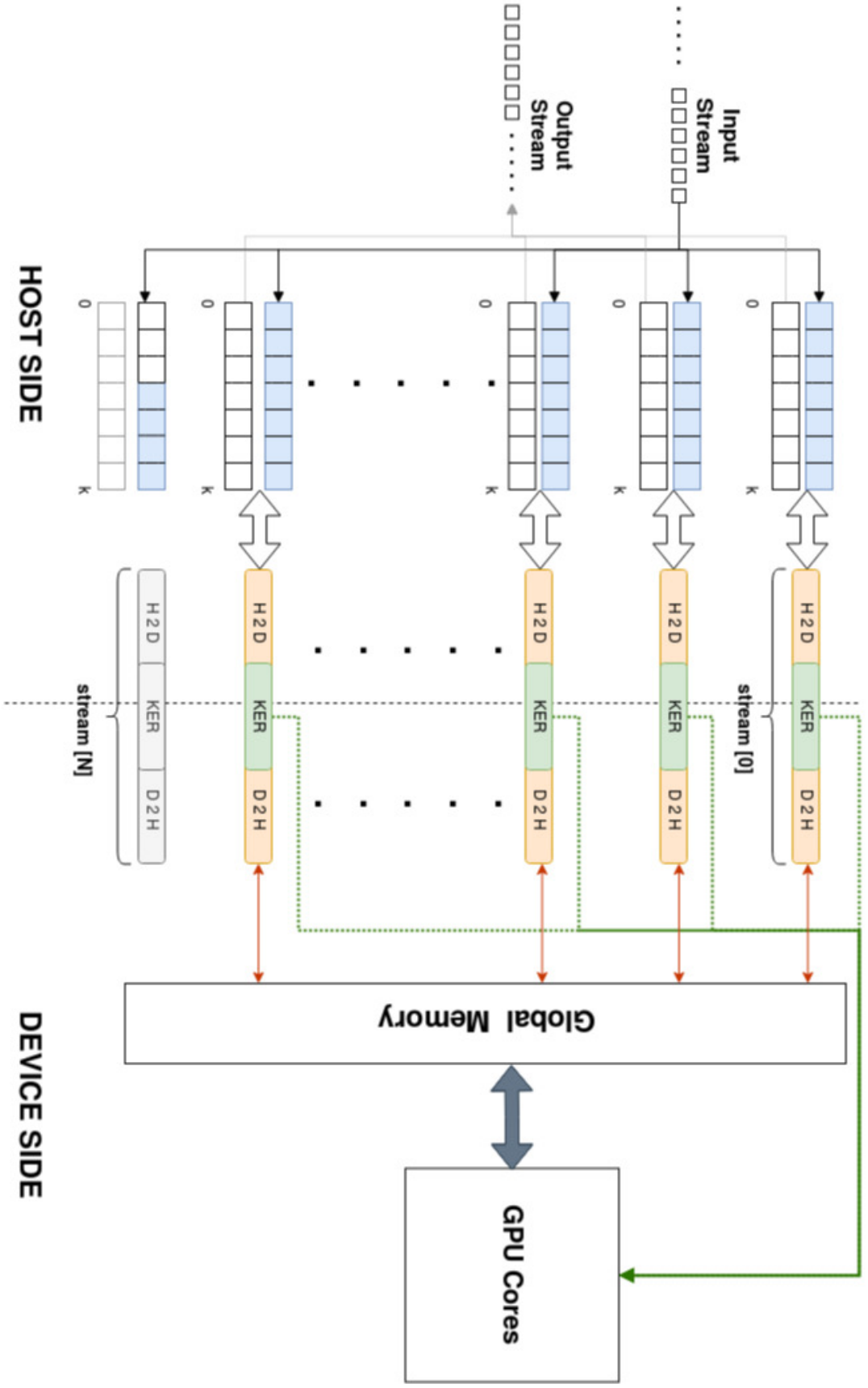

Figure 3.3: Here we have a general and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

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 were 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]);
    ```

[8] We'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

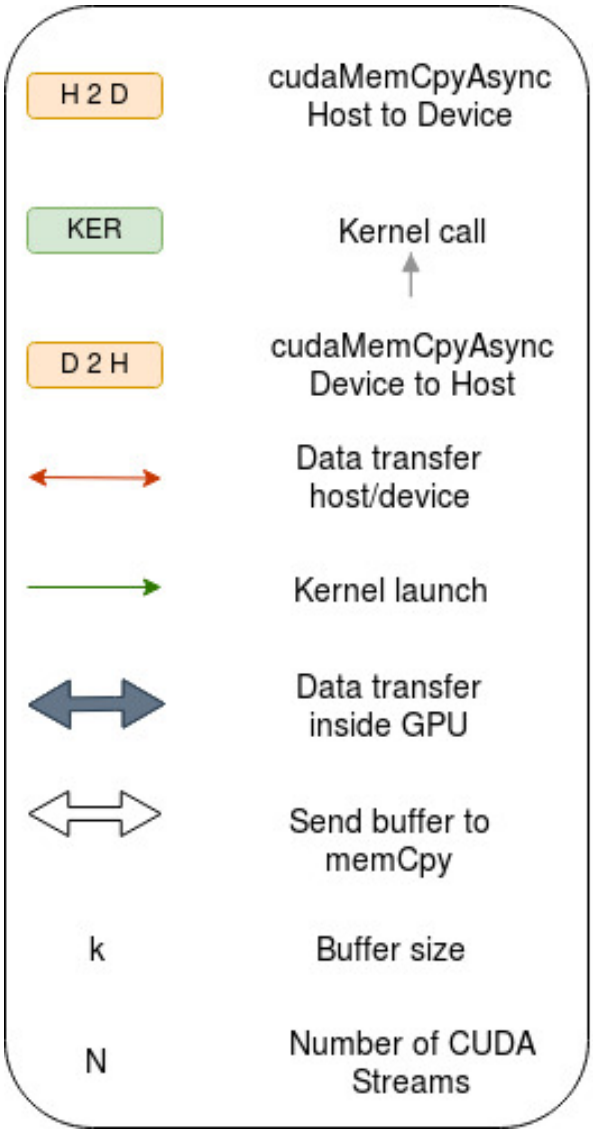

Figure 3.4: Legenda about Figure 3.3.

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.

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

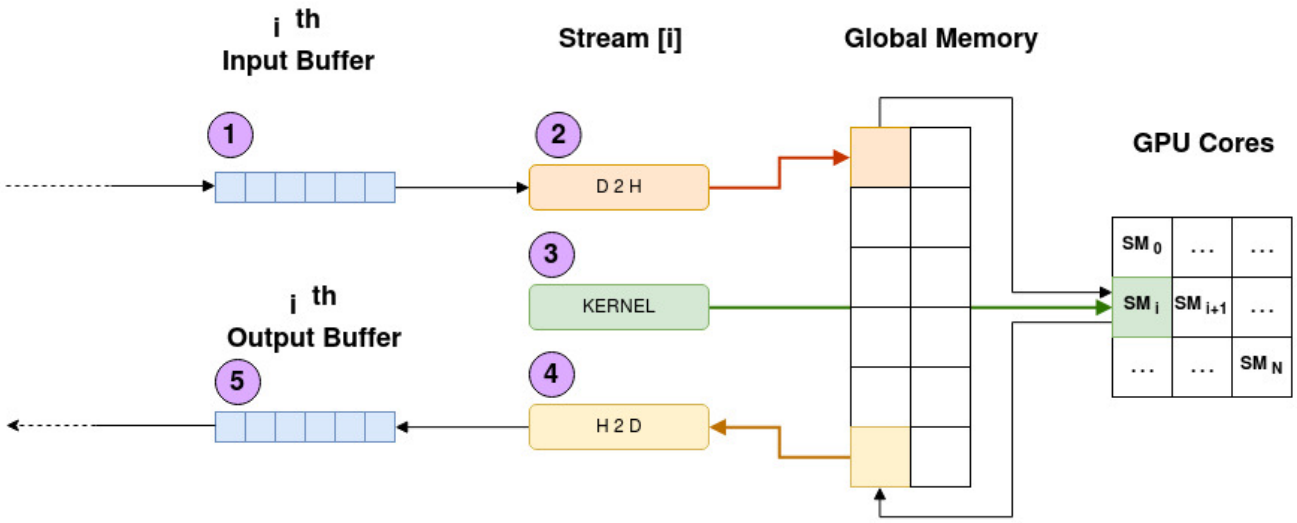

Figure 3.5: Here we can see what exactly happens in a certain CUDA Stream. Light violet numbered labels shows the order in which commands are issued by host to a certain stream.

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-dependent [10];
- 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,

[10] At 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 functions[14] 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:

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

[12] That'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.

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

[14] These 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;

[15] Check 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.

```
__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, ..., N` and `sum` will be `C[ROW, COL]`, ie one of the items result matrix.

```
/**** 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

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

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].

```
/**** 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) {
                                        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];
```

[2] Another 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.

```
                                        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 = maxThreads/BLOCK;`

```
#ifdef 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.

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

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

        cosKernel<<<GRID, BLOCK>>>(m, chunk, x_d);
        #ifndef MEASURES
```

[3] Note 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.

```
                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.

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

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

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

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

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:

```
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.

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]

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

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

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

```
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);

        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 experiments on code and, to this aim, what kind of comparisons will be made.
The second section will explain how to implement tests, ie chosen datasets for each type of code and scripts main features. After that, the other section will move on results, in particular will be shown time measures and plots with some final remarks and comparisons with respect to what we expected.

5.1 Expectations

As previously mentioned, what we want to see is that our model and implementation of a 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. This is strictly related to the fact that we can't expect Streaming parallel problems, that we implemented, to perform better than the equivalent Data Parallel version. [1] The second point above means we expect the 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 computations;
- When the kernel execution takes an amount of time near the one for a data transfer.

When one, or more, of the above conditions isn't meet 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*

[1] As we mentioned in previous Chapters, the GPU is specifically designed to be efficient and to perform at its best on Data Parallel problems.

execution. Clearly, in the case were CUDA streams are used, we have an additional time cost to create and destroy streams, especially when lot of streams are spawned. For completeness some measures on CUDA Stream creation/destruction [2] will be reported, but we won't sum up them with measures on data transfer and kernel execution. This is because, even if the streams overhead can be notable [3], it's a one-time cost to pay.

This means that it won't weigh on performances, given that in the beginning we create CUDA streams, then we'll run kernels on a indefinitely long input stream 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 kernel, 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 positioned:

```
/**** 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, ...);

                . . . .
```

[2] Measures on CUDA Streams spawn/deletion are collected from *nvprof* log file, where all CUDA APIs time is precisely measured.

[3] We'll see that CUDA streams creation/destruction can take from few to hundred milliseconds, depending on how many they are.

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

```
}
```

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

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

A problem with using host-device synchronization points, such as `cudaDeviceSynchronize()`, is that they stall the GPU pipeline. Events, instead, provide a relatively light-weight 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 make use of the concept of CUDA streams. 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 in the first argument the number of milliseconds time 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 hundreds milliseconds.

It's important to point out why we used events on the default stream. If one of the events were last recorded in a non-NULL stream, the resulting time may be greater than expected (even if both used the same stream handle). This happens because the `cudaEventRecord()` operation takes place asynchronously and there is no guarantee

that the measured latency is actually just between the two events.
Any number of other different stream operations could execute in between the two measured events, thus altering the timing in a significant way [10]. So given the asynchronous nature of CUDA calls we do in non-default stream, the behavior and order in between different streams is unpredictable. This means that a call from a different non-null stream can actually be issued in between two events we're trying to recording, even if they were issued from the same non-default stream.

Another significant fact on events, is why we chose to put timers outside the loop over input stream. We could insert events again on default stream, but inside the loop, so that we measured singularly each iteration[4] and sum up all those elapsed times. There would have been three problems:

- 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 overhead, apart from synchronization time.

In 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 host-device synchronizations as we can: given that we're working on input/output streams of items from host and, thus, "stopping" this flow on host side at each iteration would invalidate the gain of our model, increasing the overall completion time (of 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.

[4] And so measure each single memory copy H2D, Kernel execution and memory copy D2H.

For completeness, we'll show some performances case of interest measured by profilers, in addition to those from timers. This will allow us not only to observe the correctness of some measurements, but also to check some special cases or technical details.

5.2 Tests setup

Once it is determined the time measure criterion, we have to decide what behaviors we want to observe from our code and its performances. Note that for each type of input dataset, we run multiple times (say N_{test}) the executable so that, for a certain input setup, we can collect more 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 subsections we'll show, for each type of kernel, what type of tests have been made.

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.

Tesla P100	Tesla M40
114 688	—-
458 752	—-
1 835 008	—-
3 670 016	—-

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

5.2.1 Simple-computation kernel

For the computational-bound kernel, for each type of input dataset, we identified three different values for kernel iterations number, call it M: $10000, 500000, 1million$.
These values will identify how many times our 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 dimension for both of the GPUs we used to run tests, ie **P100** and **M40**.
The choice for 1024 was made because CUDA Occupancy APIs, suggested this as best block size for our 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.
Then we performed the following executions:

1. **Classic data parallel approach**

   Here we launch the execution of our simple-computation kernel, as it would be classically used, that is as data parallel application. To this aim, we should assume that, instead of having as input a stream of items, we'll have a quite long array of data, in particular made 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 in this case, 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 are in the case of 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 kernel launched could fill an entire SM or at least a part, 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 block active, and so running, on a certain Streaming Multiprocessor. However we should hit this limit only when we have a poor amount of threads in each block and a consistent quantity of thread blocks. This isn't our case, since we decided to use the maximum number of threads in a block, ie for x dimension.

   The first limit, instead, is our main goal here. Having at most 2048 active threads in a SM means that 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 `<<<1, 1024>>>`. So here each launched kernel will have one block containing 1024 threads and this theoretically should correspond to half the occupancy of a SM.
Clearly the code will launch a lot of buffers to device, ie enough to hopefully fill all SMs. The number of chunks will be limited by the limits we gave to the input stream length (see 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`, that is 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;
- $\mathbf{N_{SM}}$ CUDA Streams, with $N_{SM} = \#Streaming\ Multiprocessors$. This is our 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 (our zero case).

  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, will be spread in SMs as soon as will be available requested resources.

3. **Streaming parallel with bigger buffers** This tests 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 to (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 CUDA streams scenarios, so for this configuration we executed code using: **Zero**, **Three** and $\mathbf{N_{SM}}$ CUDA Streams. Motivations for those number of non-default streams are completely analogous to the one explained before.

4. **Streaming parallel using `std::future` approach (smaller buffers)** Here's a particular case, because we wanted to experiment a different approach for host side. Until now we have seen all cases in which CUDA calls were issued synchronously by a single host thread, ie the main thread.

   We tried, instead, to see what could happen if we used more threads calling in asynchronous way, at each iteration, a copy H2D, kernel call, and copy D2H.

   We tested only a single scenario, $\mathbf{N_{SM}}$ CUDA Streams, just to observe, in our CUDA stream special case, if this approach would have give even more advantage, than single thread host.

5.2.2 Matrix Multiplication

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

	Tesla P100	Tesla M40
Mat. Order	**In stream limit**	**In stream limit**
256	128	—-
484	256	—-
900	1024	—-
	Data Parallel	**Data Parallel**
	2048 $(= 128 \cdot 16)$	—-
	3840 $(= 128 \cdot 30)$	—-

Table 5.2: Input dataset for Matrix Multiplication kernel. Above Stream parallel configuration, below Data Parallel correspondent.

kernel.

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

First of all, before we were dealing with a stream of floats that were accumulated in buffers, here we have a stream of matrices. In particular as soon as we have two available input matrices, they're sent out to device to apply the matrix multiplication kernel. Finally we get back to host with the result matrix, that will be one of the output stream components.

Another assumption is that, for simplicity, we're measuring on square matrices case, even if code was implemented for non-square case too.

Thus we performed the following executions:

1. **Classic data parallel approach** As always we have to put a limit on input stream, so that we can measure completion times and compare different approaches.

   To test the data parallel version it suffices to treat input/output streams as huge matrices. This means that we'll do computations, no longer on stream of small

data structures, but on a unique big data structure.

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 5.2, 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 relative comparison to some Stream parallel versions [5].

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 5.2 are reported all input stream limit, for each of

[5] We'll see later what cases we match between data and stream parallel to make comparisons.

	Tesla P100 & Tesla M40	
Img. Order	**In stream limit**	**In stream limit**
128	256	—-
256	1024	—-
	Data Parallel	**Data Parallel**
	4096 $(= 128 \cdot 32)$	—-
	8192 $(= 256 \cdot 32)$	—-

Table 5.3: Input dataset for Matrix Multiplication kernel. Above Stream parallel configuration, below Data Parallel correspondent.

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 $\mathbf{N_{SM}}$ 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. And the reasons why we test for those numbers of streams are the same too.

5.2.3 Image processing

With image processing, ie Blur Box algorithm we're facing a kernel mostly memory-bound and especially rich of divergent execution flows, so we made similar tests to the ones for Matrix multiplication.

Thus we performed the following executions:

1. **Classic data parallel approach** If we think to a picture as a matrix of pixels, then it's easy to see the analogy to the previous application.

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

Again we can consider 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 in 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**

   - **Zero** CUDA Streams
   - **Three** CUDA Streams
   - **N** CUDA Streams, with $N = \#ofStreamingMultiprocessors$

5.3 Results

Results are collected, for each group of execution, in some `.txt` files. In those files we can find a list of applications outputs in `.csv` format.

Dire che smaller buffer meglio perché non monopolizza interamente un SM, lascia spazio ad un altro kernel launch di entrare.

5.4 Plots

Plots

CHAPTER 6

Conclusions

Conclusions

List of Tables

Bibliography

[1] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design: The Hardware and Software Interface*, V Edition. Appendix C by J. Nickolls, D.Kirk.

[2] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, Jack Dongarr, *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platformGPU Programming*

[3] John Jenkins, Isha Arkatkar, John D. Owens, Alok Choudhary, Nagiza F. Samatova, *Lessons Learned from Exploring the Backtracking Paradigm on the GPU*

[4] Marco Danelutto, *Distributed Systems: Paradigms and Models*, 2014

[5] CUDA Toolkit Documentation, *CUDA C PROGRAMMING GUIDE*, 2018

[6] CUDA Toolkit Documentation, *https://docs.nvidia.com/cuda/profiler-users-guide/index.html*

[7] Mark Harris, *CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler*, https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

[8] Vasily Volkov, *Better Performance at Lower Occupancy*, September 2010

[9] Mark Harris, *How to Implement Performance Metrics in CUDA C/C++*, article from CUDA Dev Blogs: https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/ , 2012

[10] AUTH, *NVIDIA Library Documentation- Event Management*, from site: http://horacio9573.no-ip.org/cuda/group__CUDART__EVENT_g14c387cc57ce2e328f6669854e6020a5.html, YEAR