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

This is the first section.

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 exe-

cuting 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, indeed 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 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 did not arrive until 2003. For the gaming industry this meant more realistic game-play. For the scientific community, the addition of floating point, meant that overflow associated with fixed-point arithmetic was no longer a problem. Other computational advances would not be possible if it weren't for the programmable shaders, that broke the rigidity of the fixed graphics pipeline. LU factorization with partial pivoting on a GPU was one of the first common computational 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: archaic 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 that parallel platform, we'll show some further informations about CUDA in Section 1.4.

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 were data parallel.

In particular some of latest researches are moving towards *Task parallel* applications (sometimes also known as *Irregular-Workloads parallel patterns*). For example

- The backtracking paradigm: used in constraint satisfaction in AI, frequent itemset mining in data mining, maximal clique enumeration in graph mining, k-d tree traversal for ray tracing in graphics. Backtracking is oftentimes at the core of the problems that are combinatorial by nature and, therefore, compute-and-memory-intensive.

Recent advancements in parallel computing architectures have opened up possibilities for more computationally- and energy-efficient algorithms. In particular, graphics processing units (GPUs) have been maturing not only for graphics ap- plications, but also for general-purpose computations 1 [14]. Some computational motifs perform effectively on a GPU, while the effectiveness of others is still an open issue. For instance, Lee et al. note an average speedup of 2.5x of various algorithms on the GPU vs. optimized Nehalem implementations, and both Lee et al. and Vuduc et al. highlight memory-bound algorithms on the GPU that perform at the same level or worse than the corresponding CPU implementation [12,18].

Despite some of the successes of recent computational dwarfs on GPUs, the mapping of the backtracking paradigm onto the GPU architecture has been recognized as a notoriously difficult problem for a number of reasons. Table 1 names a number of difficulties that a mapping of a backtracking problem to the GPU could encounter, leading to a vastly inefficient use of the GPU memory hierarchy and SIMD-optimized GPU multi-processors.

There have, however, been algorithms successfully mapped onto the GPU, though with major departures from the general case of backtracking. The most visible example is in ray tracing, where k-d tree acceleration structures are used to compute ray intersections by traversing the tree in a depth-first fashion [5,10].

Our goal, therefore, is to investigate the parallelization of the backtracking paradigm on the GPU. To do this, we analyze the components of difficult back- tracking problems and propose tree-level and node-level parallelizations of search space traversal, as well as buffer-based output. At best, given the performance of other computational motifs and the nature of the backtracking problem, we cannot expect an order of magnitude increase in performance. Rather, a more re- alistic performance goal is to perform at one to two times the CPU performance, which opens up the possibility of building future backtracking algorithms on heterogeneous hardware (such as CPU-GPU clusters) and performing workload- based optimizations [3]

A variety of scenarios demand task-parallelism • We will discuss three – Reyes Rendering – Deferred Lighting – Video Encoding

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 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

$$x_m, ..., x_1$$

the farm with function f computes the output stream

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

[4] 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 and output data type:

- data structure 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 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* (we'll insist on occupancy topic in Chapter 3).

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 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;
- 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. In particular we worked with those 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 the 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 all tests have 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 Chapter 2 will be shown all features and details about the aforementioned tools.

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.
After that, 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 behind the followed approach and solutions.
Following those guidelines, in Chapter 4 will be presented and explained main implementation features. Here will be reported 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, connecting them 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 GPUs, it's really important to list all main specifics of devices in remote servers, see 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 that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems, in certain cases even in a more efficient way than on a CPU.
CUDA comes with a software environment that allows developers to use C as a high-level programming language.

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The CUDA challenge is to develop application software that scales its parallelism to leverage the increasing number of processor cores, while maintaining a low learning curve for programmers familiar with standard programming languages

	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 machines employed in this project.

such as C.

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;
- 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 –.

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 scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions. [5]

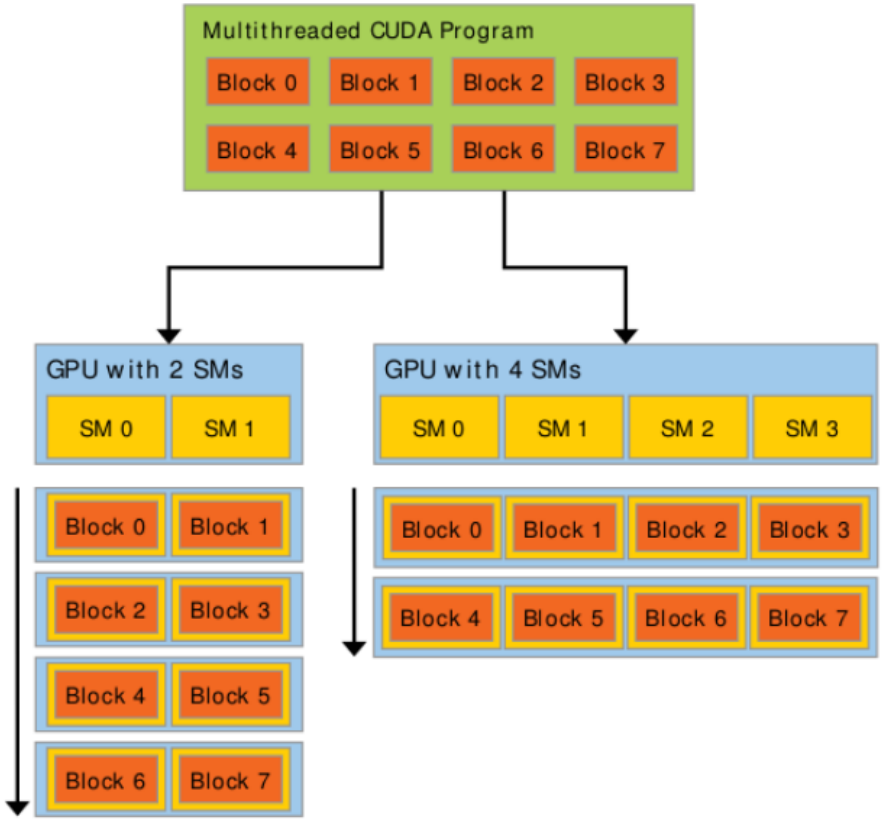

Figure 2.1: GPU scalability.

2.2 Profilers

NVIDIA profiling tools enable the understanding and optimization in performance of CUDA applications.

The Visual Profiler is a graphical profiling tool that displays a timeline of your application's CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities. 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.

There are many profiling options, provided through command-line. Profiling results are displayed in the console after the profiling data is collected, and may also be saved for later viewing by either `nvprof`. The textual output of the profiler is redirected to `stderr` by default [6, 7].

`nvprof` operates in one of the modes listed below:

1. *Summary Mode* is the default operating mode, here 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 we have number of calls and total, max, min and average time);

2. *GPU-Trace and API-Trace Modes* provides a timeline of all activities taking place on the GPU in chronological order with some detailed information such as kernel parameters, shared memory usage and memory transfer throughput (enabled individually or together);

3. *Event/metric Summary Mode* gives the list of all available events and/or all available metrics on a particular GPU (nvprof is able to collect multiple events/metrics at the same time);

4. *Event/metric Trace Mode* event and metric values are shown for each kernel execution.

[6]

We used it as a quick check, for example to see if the application isn't running kernels on the GPU at all, or it's performing an unexpected number of memory copies. To this aim it's enough to run the application with
`nvprof ./myApp arg0 arg1 ...`
we can quickly see a summary of all the kernels and memory copies that it used.

Although when we needed to launch our tests, we wanted to consult profiling results in a second moment and whenever was necessary. So we used `--log-file` to redirect the output to files available for deferred examination (or for later import into either nvprof or the NVIDIA Visual Profiler).

`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 cross-platform performance profiling tool that delivers developers visual feedback for optimizing CUDA C/C++ applications.

The Visual Profiler displays a timeline of your application's activity on both the CPU and GPU so that you can identify opportunities for performance improvement. In addition, the Visual Profiler will analyze your application to detect potential performance bottlenecks, using highly configurable tables and 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, this is a low overhead performance analysis tool that provides insights to optimize software, and it allows tuning to scale efficiently across any quantity or size of CPUs and GPUs.

Unbiased activity data is visualized to investigate bottlenecks, avoid inferring false-positives, and pursue optimizations with higher probability of performance gains. It 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 for data transfers and kernel executions overlapping. This allowed to check if code was properly written to hide data transfers, even though often it happens that profilers introduce some sampling synchronization, giving not fully reliable visual results. By the way we'll see in detail this aspect and how it was managed in Chapter 3.

2.3 CUDA 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.

2.3.1 Kernels

CUDA C allows the programmer to define particular C functions, called *kernels*, these, when called, 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 thi 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.
The index of a thread and its thread ID relate to each other in a straightforward way:

- For a one-dimensional block, they are the same;
- for a two-dimensional block of size $(D_x, D_y) \rightarrow$ a thread of index (x, y) has $threadID = (x + y \cdot D_x)$;
- for a three-dimensional block of size $(D_x, D_y, D_z) \rightarrow$ a thread of index (x, y, z) has $threadID = (x + y \cdot D_x + z \cdot D_x \cdot D_y)$;

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, a thread block may contain up to 1024 threads (see Table 2.1 for limits in our used machines).
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.

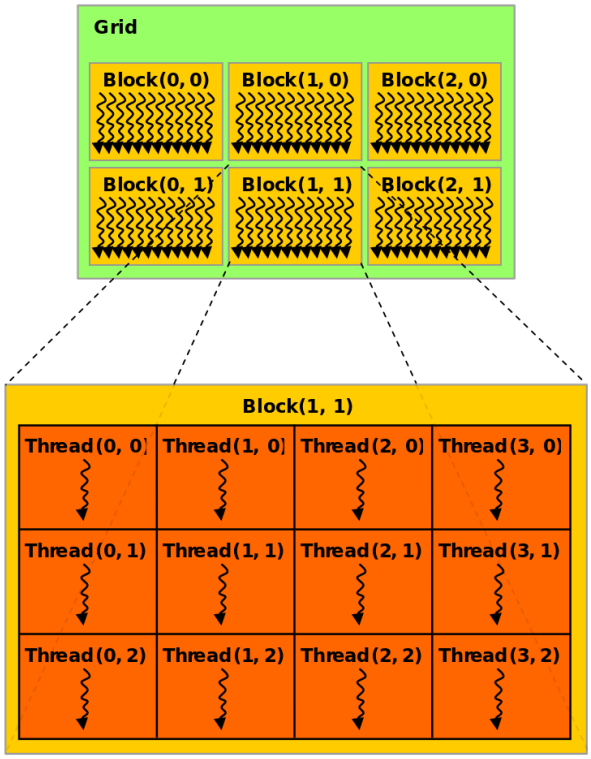

Figure 2.2: Above: a Grid formed by Blocks. Below: a Block formed by Threads.

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 is accessible within the kernel through the built-in `blockDim` variable. [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. Applications may query this capability by checking the `asyncEngineCount` device property (see Device Enumeration), which is greater than

zero for devices that support it. If host memory is involved in the copy, it must be page-locked.

In particular some devices, of compute capability 2.x and higher, can overlap copies to and from the device. Applications may query this capability by checking the `asyncEngineCount` device property (see Device Enumeration), which is equal to 2 for devices that support it [1].

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. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore 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 from host to device, one kernel launch, and one memory copy from device to host:

```
for (int i = 0; i < 2; ++i) {
        cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
        size, cudaMemcpyHostToDevice, stream[i]);
        MyKernel <<<100, 512, 0, stream[i]>>>
```

[1] This is the case for both of our machines.

See the last line of `deviceQuery` in Table 2.1

```
        (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 . Overlapping Behavior describes how the streams overlap in this example depending on the capability of the device. Note that hostPtr must point to page-locked host memory for any overlap to occur. Streams are released by calling cudaStreamDestroy().

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

2.3.4 Visual Studio Code

Visual Studio Code is an open-source code editor developed 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 [2].

It is customizable and so it was exploited to add C++ and CUDA editor extensions. Furthermore an SFTP extension allowed to quickly upload/download files from remote machines.

Visual Studio Code is based on Electron, a framework which is used to deploy

[2]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/

Node.js applications for the desktop running on the Blink layout engine. Although it uses the Electron framework,[10] the software does not use Atom and instead employs the same editor component (codenamed "Monaco") used in Azure DevOps (formerly called Visual Studio Online and Visual Studio Team Services).[11]

In the Stack Overflow 2019 Developer Survey, Visual Studio Code was ranked the most popular developer environment tool, with 50.7% of 87,317 respondents claiming to use it

2.3.5 nvcc compiler

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

The compilation here involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file. 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 used options for defining macros and include/library paths [3].

All non-CUDA compilation steps are forwarded to a C++ host compiler that is supported by nvcc. This compiler can be used almost the same way as a classic `gcc`, for example:

```
nvcc -std=c++14 -g -G nvcc -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

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

- **Tesla M40**

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

2.3.6 `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.
The tool provides developers with a mechanism for debugging CUDA applications running on actual hardware. This enables to debug applications without the potential variations introduced by simulation and emulation environments. 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 especially to support debugging 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 compilation, or JIT compilation for short.

Cuda-gdb was used to debug all compiled `.cu` source file. 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 did inside kernels.

2.4 Tests, Result gathering, Plots

2.4.1 Bash scripts

2.4.2 Python scripts

Python 2.7.6

CHAPTER 3

Project Logic

The reasoning in this work started by considering the characters of a problem that is recognizable as Farm parallel pattern. Then the study moved to considering how a GPU works, main architectural characteristics and 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 that see sect ****** 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].

However it has no effect on the "function" computed by the skeleton. It has only effect on the "parallel semantics" metadata.

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. The time between the delivery of two different task results, instead, can be made close to the time spent to compute f sequentially divided by the number of parallel agents used to execute the farm, i.e. its parallelism degree. [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 happens if we wanted to manage such computations on stream's elements 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 (already available in global memory);

4. finally, computed elements will be copied back to host side.

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". So, as we mentioned in the previous section, it's not realistic to have one worker per stream element. Furthermore, we surely don't want to transfer from/to GPU one element at time, but 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 a stratagem, to hide as much as we can data transfer times (both in $Host \rightarrow Device$ and $Device \rightarrow Host$ direction).

3.2.1 Overlapping: Data Transfer hiding

We introduced in Section 2.3.3 CUDA Streams, they can perform an asynchronous memory copy to or from the GPU concurrently with kernel execution. Since the machines we worked on have both concurrency on data copy and kernel execution and they have 2 copy engines, we exploited this capability combined with streams.
In this way we wanted to achieve a situation in which we could hide, 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.

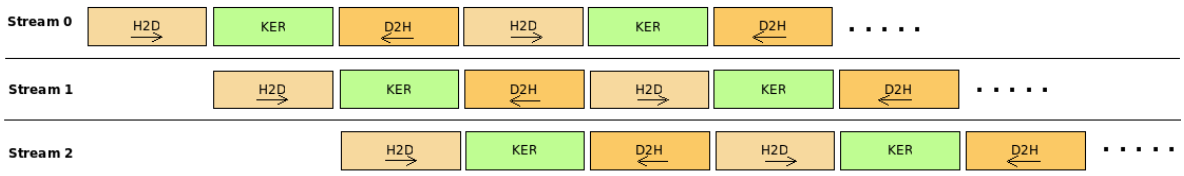

Figure 3.1: Ideal behavior for 3 CUDA Streams.

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 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. Among those factors the only that can influence in our case is the order in which commands are issued to each stream, given that our GPUs supported all kind of above mentioned concurrencies.

Other 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 introducing host issued operations in-between different streams commands, this should have not introduced *Implicit synchronization*. Moreover we avoided all possible *Explicit synchronizations*.

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. 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. To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps. This is possible because the distribution of the warps across the block is deterministic [5].

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.

So about these two scenarios we had to make some assumptions and tunings, that we will see in ********.

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 they was necessary to reach almost the full ***Occupancy***.

Clearly, when we start, we'll have a portion of time, a sort of "warm up" where we'll have first data transfers and few kernels. But as soon as we have enough data transfers and so a lot of kernels.

In the reality when we just said lot of kernels we meant, lot of blocks. 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 as described in Asynchronous Concurrent Execution. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices.

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 as described in Asynchronous Concurrent Execution.

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 cleat 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.
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 as described in Hardware Multithreading.
Register and shared memory are other important Occupancy variables [5].

Therefore 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 [2].

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.

3.3 Overall Logic

We have a stream of items, in our case we chose floats, we don't know how much they are and their arrival frequency. What our project do will be summarized in the following steps:

1. as items start to arrive, we accumulate say k items at time in 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, we launch the kernel execution, with a certain Execution configuration. It will be sent to **streams[k]** as well;
4. once the kernel ends its computations,on **streams[k]**, we copy back to host the chunk of manipulated data;
5. from the output buffer we'll send each item as output stream.

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

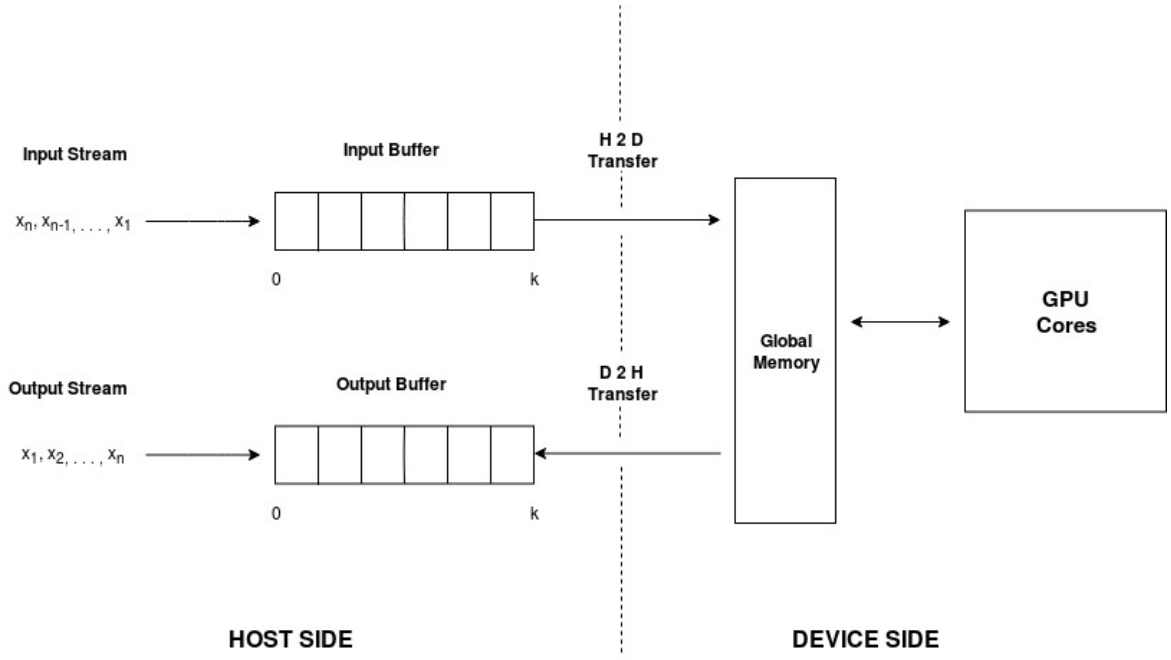

Figure 3.2: Here we have a general and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

This behavior is illustrated graphically in Figure 3.2. Here we can see our input stream, every k items we transfer them to Global memory of GPU; items will be spread all over warps in a way such that on every item will be applied all calculations, specified inside kernel code.

From that figure it may seems we're sending only k items at time to/from GPU;but, assuming k items as a single item, this would correspond almost on a farm with one worker, processing one item per time. And this isn't surely what we want.
And this is here where CUDA Streams [3] come into play.

[3]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)

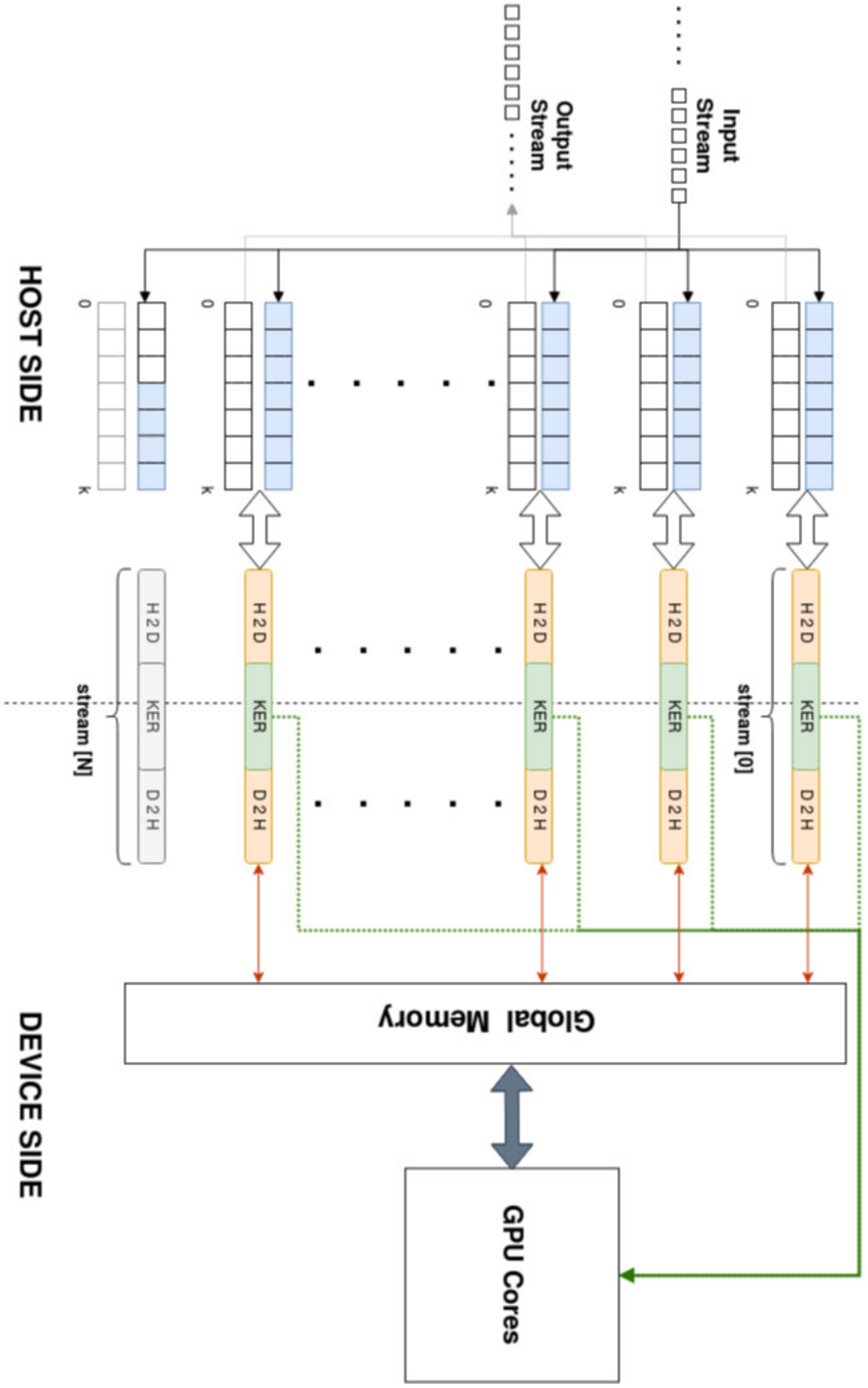

Figure 3.3: Here we have a general and broad graphical representation of our idea on how to fit a Farm parallel pattern on GPU architecture.

We've based the use of CUDA Streams on the following ideas:

1. we have as many streams as Streaming Multiprocessors [4] and, at any given time, each of them hopefully issues a data transfer or a kernel executions;

2. when we arrive at the point where all stream, sooner or later, issues a kernel launch (a maximum speed in a sense) ideally we expect that each kernel execution is taken over by a certain multiprocessor;

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

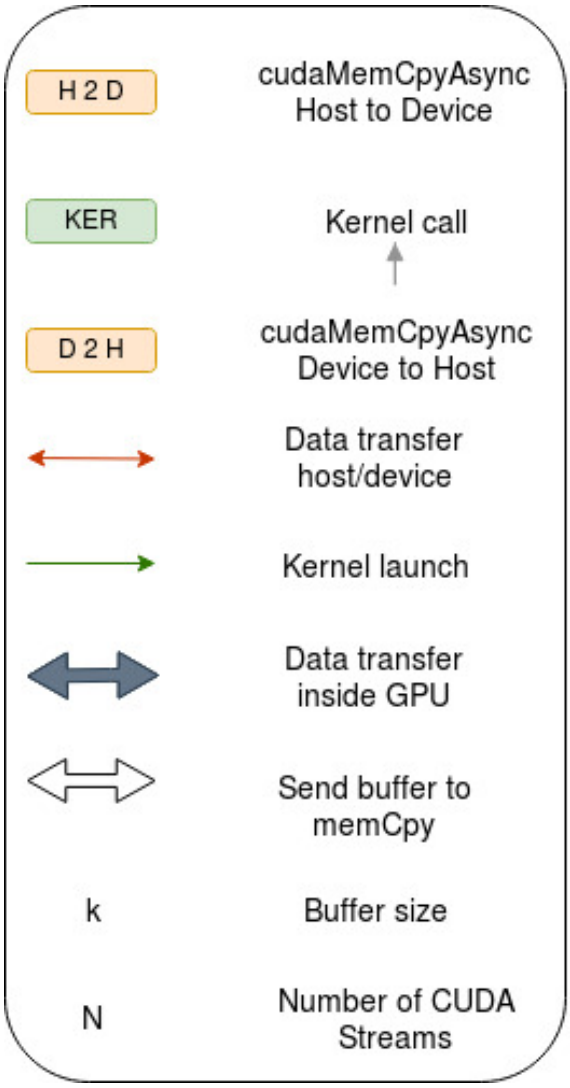

Figure 3.4: Legenda about Figure 3.3.

All of those parameters have been established at first with some reasoning and assumptions on NVIDIA GPUs nature, later we moved on experimental proves [5]. Measures and other estimation lead us to consider specific values for those variable parameters.

[4] 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.

[5] We'll see in next section more informations about Tunings.

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;
- as input stream items arrive, we let them fill buffers;
- in a Round-Robin fashion we spread 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
     ```
     cudaMemcpyAsync( devBuffer, hostBuffer, bytes,
                      cudaMemcpyHostToDevice, stream[i]);
     ```
  2. soon after we call the kernel, on the `stream [i]`, to make desired computations on that buffer;
  3. then, asynchronously again, we bring back results to host side
     ```
     cudaMemcpyAsync( devBuffer, hostBuffer, bytes,
                      cudaMemcpyHostToDevice, stream[i]);
     ```
  4. hopefully a Streaming Multiprocessor or a part, is made busy;

Initially only few cores will be really busy, but as soon as the buffers and streams get full, the pressure on the GPU will increase, so we expected that workload should be enough to almost fill all of Streaming multiprocessors. For us this means that we tried to have the maximum number of active threads inside each SM, having to do some work.

To put a magnifying glass upon Figure 3.3 and to take a closer look to what we just said, check Figure 3.5.
From that we can see the order in which we issue command in a stream, and this will be the order in which they will be issued to device side too, for that stream. But we take advantage of the fact that we can overlap data transfer and/or kernel

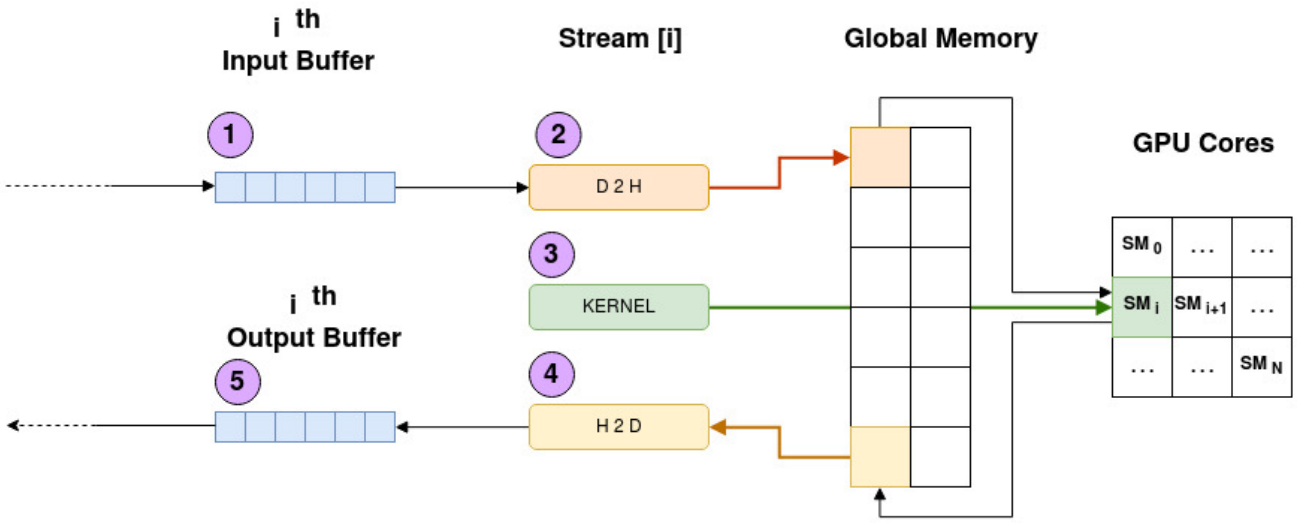

Figure 3.5: Here we can see what exactly happen in a certain CUDA Stream. Light violet numbered labels shows the order in which commands are issued by host to a certain stream.

execution in a `stream [i]` with the ones in a `stream [k]` or `stream [j]` (for some $k, j \in [0, N-1], for N = \#SMs$). Note that in the Figure 3.5, we represented a kernel execution as fully occupying an entire SM; in reality it's not exactly how it goes. We'll see how we managed and tried out the 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* [6]. 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.

[6] By "classical approach" we mean to transfer data and execute kernels without any type of overlapping. This is equivalent to send input, wait for data transfer completion on host, call kernel, send data back to host, only when they are a ready result and they are then transferred back to the waiting host.

3.4 Tunings

We followed for first some important best practices to obtain 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 done this;
- 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;
- we exploited ***Occupancy Calculator*** both in spreadsheet and API functions formats [7], [5].

Given those initial guidelines, it's important to highlight what are variable parameters in Figure 3.2, on which the tuning was made:

- the number of Streams;
- the number of threads per block (block size);
- the number of blocks (grid size);
- the buffer dimension.

After lots of attempts and experiments, for each kernel type, we extrapolated best suitable values [8].

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

[8] Check Chapter 5 to see all main values and their respective performances

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 computations; 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

Specs and code.

4.1 Stream Parallel on GPU

<<< blockSize>>>

4.2 Data Parallel un GPU

<<<dataSize/blockSize, blockSize>>>

4.2.1 CUDA Occupancy APIs

The occupancy calculator API, cudaOccupancyMaxActiveBlocksPerMultiprocessor , can provide an occupancy prediction based on the block size and shared memory usage of a kernel. This function reports occupancy in terms of the number of concurrent

thread blocks per multiprocessor. Note that this value can be converted to other metrics. Multiplying by the number of warps per block yields the number of concurrent warps per multiprocessor; further dividing concurrent warps by max warps per multiprocessor gives the occupancy as a percentage. The occupancy-based launch configurator APIs, cudaOccupancyMaxPotentialBlockSize and cudaOccupancyMaxPotentialBlockSizeVariableSMem , heuristically calculate an execution configuration that achieves the maximum multiprocessor-level occupancy.

The following code sample calculates the occupancy of MyKernel. It then reports the occupancy level with the ratio between concurrent warps versus maximum warps per multiprocessor

4.3 CPU and GPU Mix

Queue with P and Q chunk exec by respectively CPU and GPU.

CHAPTER 5

Experiments

Experiments bla bla

5.1 What and How

What and How

5.2 Results

Results

5.3 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/