GPGPU Programming with CUDA

Introduction for dummies from dummy

Leo Kruglikov

Personal notes

2022

Contents

Author's preword

These *notes* are a kind of a collection of different articles from diverse resources on this topic. More precisely, the author's interpretation of them. A big part of the code snippets are also taken from different resources. The author will do its best to try to cite the sources. Therefore it is really a *collage* of notes, articles, books on the CUDA programming. The author's main goal is to provide the most detailed possible explanation of various code snippets, as well as try to explain the main features, concepts of CUDA programming, as well as the main differences between the classical programming.

Note that this document was initially written for the author itself, who is a physics major and is a fully *self-taught guy* in programming. For the author, it was a way of learning the topic and memorize the important concepts of it. Sometimes, in order to explain different concepts, analogies will be used. Therefore, some terms will sometimes be used intercheangeably. Most importantly, some concepts, especially, when discussing architecture, can be slightly erronous. This is due to the fact that, once again, the goal of this document is **not** to provide an academically precise and correct course on the GPU's.

The goal of these notes is thus to **give us a basic understanding of the GPU architecture, and most importantly,, try to fully depict the most common examples of code snippets, using CUDA, that will run on the GPU.**.

Dictionary

- GPU - Graphics Processing Unit
- CUDA - Compute Unified Device Architecture. The language we use to *talk to the GPU*. I will often refer to it as the CUDA API. In fact, it is not a language, but an API.
- Device - the GPU, from the software viewpoint. You may think of the notion of the device as an external executor of a function, in our case, the GPU.
- Host - the CPU, from the software viewpoint. The *machine*, that will launch GPU code from a usual `C/C++` (or any other language) program, which, by default, would have been executed on the CPU.
- Kernel - nothing more than a function, that will run on the device(GPU).
- SIMT/SIMD - Single Instruction Multiple Threads/Data.

Small introduction

If one wants to perform computations on the GPU, one must have a way to address it. There are various APIs developed. The biggest ones are the Khronos Group's OpenCL, Microsoft's Direct Compute, and the one discussed here, Nvidia's Compute Unified Device Architecture, or shortly - CUDA. Do not mix it up with OpenGL, which is a slightly different thing, as it operates more on the graphics functionality.

When discussing the necessity of the GPU for computations, many come up with the example of the car and the bus [9]. Suppose you need to transport people from point A to a point B. To solve this problem, you are given a car and a bus. What would be the most optimized way to transport these people? We introduce here the notion of throughput (bandwidth) and latency. The ability to perform a certain number of operations in a certain period of time is the throughput, and the amount of time that is required to perform a single operation is the latency. In our analogy, the bus, having a smaller speed than the car, but a greater capacity has a big latency but a big throughput. On the other hand, the car has a small latency and small throughput.

So going back to our problem, we have that if the number of people to transport is significant, then the wise way to transport them is to use the bus. However, if the number of people is small enough, one should use the car, to get the small group of people faster to point B. In this analogy, the car is the CPU, and the bus is the GPU.

I am convinced that after some examples of code using CUDA, the reader will understand, how powerful actually the GPGPU model is for certain tasks' accomplishment. Like in our example with the bus and the car.

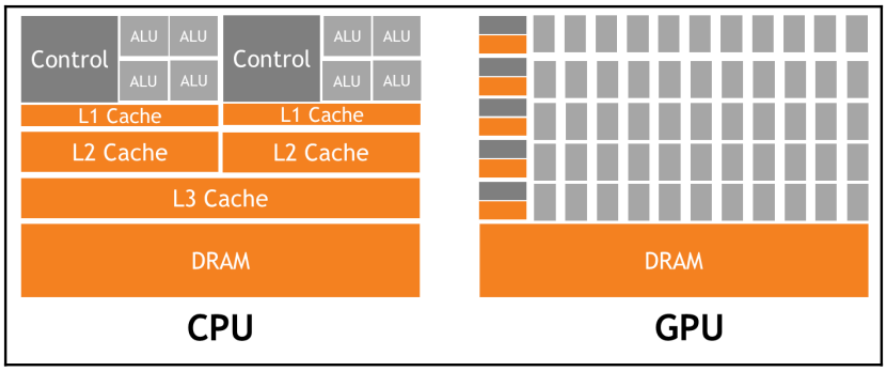


Figure 1: The schematic difference in architecture between the CPU and the GPU. Without going into details (~~as mentioned in the disclaimer, the author has not studied it in full depth~~), one can see that the GPU has many smaller ALUs. They are less powerful than those of the CPU and don't stand a chance in a theoretical 1v1 battle, but may do enough damage, when working together. Figure from [18].

1 Basics of Architecture

Before starting to consider some `C/C++` CUDA code examples, we will look into some architecture of the GPU. Indeed, one of the differences between CUDA (or GPU) and usual (sequential) programming [1], is that one must take into account the architecture of the GPU, even when writing some simple code. The GPU has a multithread architecture by default. So when the programmer is partitioning the parallel tasks, he must make sure that there are no redundant operations, and think about the way the cores will execute these tasks. If this partitioning takes into consideration all necessary aspects of the architecture and memory, it is possible to archive significant performance improvements.

The main difference between the CPU and the GPU is that the GPU has, in a way, lots of smaller CPUs in it, which are much less powerful than the actual CPU (Figure 1). [18]

1.1 Execution abstraction

As you might have noticed, the GPU is by definition a multi-threaded device. This means, it is suitable for the so-called SIMT or SIMD (remember the bus and car analogy).

From the hardware viewpoint, we are distinguishing the **Device (GPU)** itself, the **Streaming Multiprocessors (SMs)** , and the **CUDA cores**. These are physical entities, having a certain structure and features. The goal is not to give a detailed description of the GPU architecture, but rather to provide the idea of the CUDA mapping between the hardware and software world. While launching a kernel on the device, every mentioned part will be

[1] In our understanding, the *usual* programming is the code we write in C, Java, Python, etc... to run it on the CPU. It has mostly sequential instructions

assigned a certain role and will treat the software abstractions accordingly.

Us, programmers, we are writing software and operating with software abstractions. However, we still need to know how are these abstractions mapped into the *hardware world*.

Threads are fundamental units of any GPU program. It is the most primitive *executor* of a function launched on the GPU. Threads (from the software side) are executed on the **CUDA cores** (the hardware side of the program).

Blocks are grouping entities that enclose threads. When a function is asked to run on the GPU, the blocks, which *contain* threads are delegated to the corresponding **Streaming Multiprocessor** or **SM**. So by now, we get that

$$\begin{aligned} \text{Block of threads} &\xrightarrow{\text{are transmitted to}} \text{SM} \\ \text{Threads in the block} &\xrightarrow{\text{are executed on}} \text{Cores} \end{aligned}$$

So we get that the SMs are partitioning the execution of threads on the Cores at runtime. For example, suppose we have launched 8 blocks of let's say 32 threads each. Suppose our GPU has 2 SMs. Then, as mentioned above, the blocks are divided and delegated to SMs. Thus for a GPU with 2 SMs, each SM will contain ${}^{8\text{blocks}}/_{2\text{SMs}} = 4\text{blocks}$, but if our GPU has 4 SMs, each SM will contain ${}^{8\text{blocks}}/_{4\text{threads}} = 2\text{blocks}$.

Note that from the programmer's viewpoint, the threads are strictly partitioned into blocks. From the hardware's viewpoint, however, it is not said, that threads *are physically divided into blocks*. This is the exact reason, we are discussing *two worlds* and their mappings/abstractions.

Grid is the top-level abstraction layer from the software's perspective. The grid is the grouping entity that encapsulates blocks. We are thus considering that we are launching the grid on the **device**.

$$(\text{Device} \xrightarrow{\textit{contains}}) \text{Grid} \xrightarrow{\text{contains}} \text{Blocks} \xrightarrow{\text{constains}} \text{Threads}$$

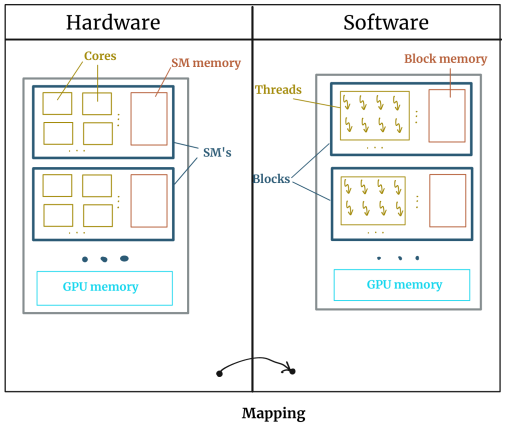


Figure 2: Hardware-software abstraction

The mapping abstraction So to recap the mentioned notions, consider the sketch of the hardware/software mapping.

The abstraction between the hardware and the software side of CUDA is shown in Figure 2. Once the function is provided, the programmer should think of the execution pipeline through threads, blocks, and the grid.

1.2 Parallel execution and warps

We briefly saw the anatomy and the terminology of some underlying elements of the CUDA kernel execution. Conceptually, the threads, to whom a kernel was assigned execute in parallel and are grouped into thread blocks. Thread blocks run concurrently with each other grouped into a grid. It is important to note (see section 1.1) that the SMs will *automatically* assign the block's execution based on the GPU resources. One may say that *there is no promise on the block's concurrent execution.* [18]. So we do not know the order in which the blocks will be run.

However, there is a notion, that more or less guarantees the order of the execution of threads. **The Streaming Multiprocessor treats threads in groups of 32, which are called warps**. Think of the warps as a way to handle the threads, rather than a way of grouping (as the blocks of threads) [2]. We will see that the warps are an extremely important concept of GPU development. We will see that the execution of kernel by threads is more efficient when we take into account the fact that they are grouped by warps.

1.3 Memory model

The memory model of the GPU is quite complicated. It has different fields of memory that have different characteristics- latency of access, write/read modes, size, scope (to whom it is visible), etc... First let's take a look into **general** notions concerning the memory model.

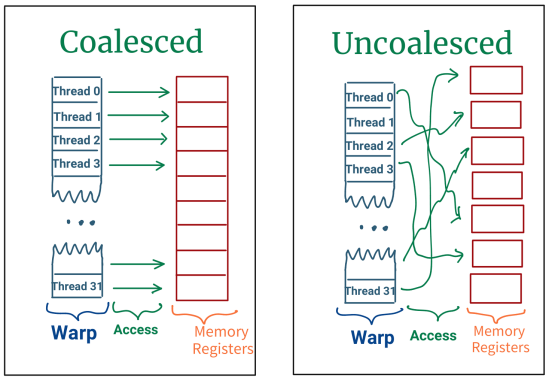


Figure 3: Memory access optimization mechanism.

Coalesced vs uncoalesced memory access. Imagine a certain number of warps are scheduled by the SM. Let's say 2 blocks of 128 threads, which gives $2\text{blocks} \cdot {}^{128\text{thr.}}/_{32} = 8\text{warps}$. These warps fetch some data from a certain place in the memory of the GPU. We know that the warp is something very grouped. Therefore it would be *nice* them to access adjacent memory addresses. This notion may seem quite confusing in the beginning, but let's see how Nvidia is describing it [2]:

> *Global memory instructions support reading or writing words* [3] *of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in*

[2] In the AMD terminology, a warp is reffered to as the wavefront. It brings more insight into the nature of warps

[3] Words can be data type of a certain size.

> *global memory compiles to a single global memory instruction* ***if and only if*** *the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size). If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing.*

Do not pay attention to the notion of **global** memory (we will discuss it soon). Try to read the Nvidia's standard above again by looking at the coalesced scheme (Figure 3) to fully understand the mechanism. One may notice that this notion is one of the most crucial in the performance of the code. Indeed, when writing the kernel, one must keep in mind this aspect and try to ensure (~~when possible~~) a coalesced memory access.

The Global memory of the GPU is the largest memory in terms of size, and yet with the greatest latency. As we've discussed, the kernels are launched **from** a host (a CPU). It would be wise to be able to share resources between the host and the device. For example, send data to the GPU from a usual `C` program and retrieve it back in a processed form, otherwise, the GPGPU programming does not make sense. This is exactly the purpose of global memory. The global memory, as the name suggests, is *global*, i.e. it is **visible to all threads from all blocks**. As we will see in practice, the usual workflow of the program is to copy the data from the global memory to some other (which is discussed below), which is faster [4] to manipulate.

The Shared memory is much faster than global one but evidently smaller. Another crucial difference between global is the scope - the shared memory is **only seen by threads in the same block**. This provides the ability for threads of the same block to share results and temporary calculations, and process the data IN PLACE. Think of the following situation: a grocery store, where the customers are threads. Every time a person wants to cook something at their house, they don't drive to the store to buy every ingredient needed. They rather go there once a week, for example, and buy the amount they need. They also make sure, that everything can fit in the fridge. Thus in this ~~wonderful~~ analogy, the fridge is the low-latency shared memory, and the grocery shop - the big and unwieldy global resource - global memory. One sometimes refers to this memory as *cache memory controlled by the programmer*. However, it is important to take note that the reduced latency of the shared memory does not guarantee better performance. Indeed, the biggest pitfall for all ~~of us~~ beginners is the *bank conflicts*[5].

---

[4] You may think of it as the `malloc()` or `calloc()` functions in C or the keyword `new` in C++. Indeed, the allocation and the access to those variables is slower than declaring on the stack:
`int * ptr_a = new int;` is slower than `int a = {};`

[5] A small disclaimer: the notion of *bank conflicts* was one of the reasons for these personal notes, as it took a very long time, for the author to understand this concept. The reader should not panic if he's missing something. The examples will be discussed later in the practice part. So one should, if necessary, come back to this "theoretical" part after going through the examples. ~~The author wants to apologize for the eventual wordiness.~~

Bank conflicts. We already discussed the notion of warps, as an execution entity encapsulating threads. One may think of the banks as the analogy of warps in memory (i.e. warps are located at the *execution level abstraction*, and the banks-at the *memory level abstraction*). Shared memory is organized into banks. One *layer bank* is a sequential field of 32 memory addresses of 4 bits ($32\#\text{addresses} \cdot 4\text{bits}$).

Memory can serve as many simultaneous address as it has banks.

This is a very important property, so let's consider another ~~illustrative~~ analogy. Suppose in a national institution, each employee is assigned a counter, such that a single employee can serve only one client. Suppose you are the host (the person, who assigns people to desks) and you have a large group of people, whose number is the exact number as the number of desks. The wise choice would be to partition them between all the counters, right? Wouldn't that be silly to partition ,let's say 3, to one counter, 4 to another, etc... such that there are 8 free counters left. At these 8 counters, the employees will simply wait for customers, while there are customers, who are waiting in the queue. The analogy ~~may~~ not be the best, but the sketch should do the trick:

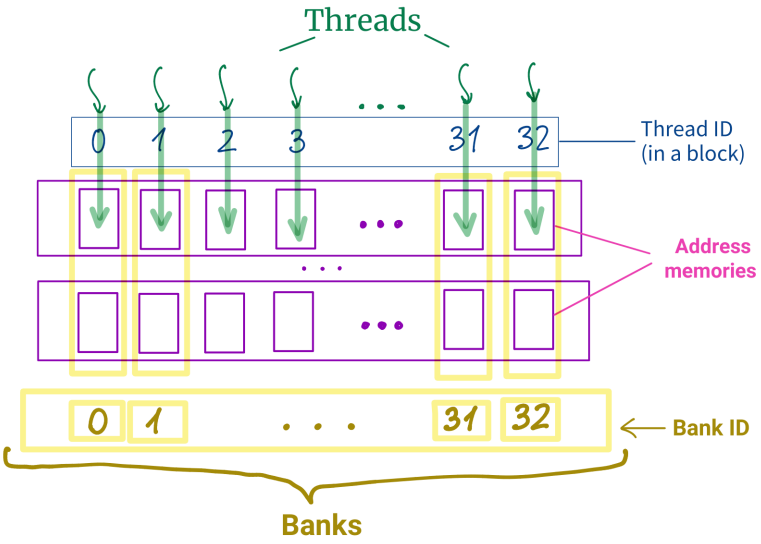


Figure 4: Memory banks serving threads. Only one thread can access a bank with a certain ID simultaneously. From the analogy, threads are clients, and banks are employees at desks, giving clients memory.

Read-only memory$*$ is, as its name suggests, can't be changed by the kernel's threads, but is loaded at compile time. This memory is not as commonly described in the GPGPU programming documentation. The name *read-only* is ambiguous, as, in different documentation, it is referred to different things. It can be referred to as the constant memory, which we will not really discuss here. Constant memory is specified with the `\_\_constant\_\_`

compiler directive. The usage of its functionally the same as some kind of constant memory in various programming languages.

The other memory, that I've seen this referring to is the *texture memory*. Texture memory is actually a very important concept, which, unfortunately, we will not treat now. Texture memory, as the name suggests, works with texture. Simply described, texture in the context of GPU is the "image", that we're attaching to some 3D object. In GPU programming related to visuals, we're often using this memory. However, as you've noticed, we are not really describing rendering here. Texture memory in CUDA is widely used together with 3rd party GPU APIs, such as OpenGL or DirectX. Indeed, there is a CUDA interoperability option with these 3rd party APIs.

Local registers By looking at the scheme of the architecture of the GPU vs the CPU (Figure 1), one may notice the number of registers in the GPU vs the CPU. The amount of registers in the GPU is incompararble with the registers of the CPU. As you might have guessed, a register is a memory, with the scope of a thread. The compiler of a CUDA program will try to optimize the number and size of registers. It is nevertheless possible, that the amount of memory in registers may fall short. Then the L1 and/or L2 caches will enter the play. This is the fastest memory available in the CUDA API, yet the most restrictive.

1.4 Memory allocation model

By now, we've made the difference between kinds of memory in terms of the scope, i.e, **who** and **when** can access various kinds of memory. This, of course, somehow impacts the way we're allocating it. However, one can classify the memory, in terms of its **allocation** procedure. There are mainly four ways of memory allocation.

We will see further that a standard program, which uses GPU resources, follows a certain path/pipeline. Roughly that is:

1. Resource declaration on the host (using `malloc`) and initialization using `memcpy`.
2. Memory allocation and initialization on the device.
3. Memory copying/transfer from the host to device.
4. Accelerated computation on the GPU and storage of the results from the device
5. Copy data from the device back to the host.

One of the steps is the allocation of memory on the GPU from the host (by calling a certain CUDA function, which will be described later).

- Pageable memory

- Pinned memory
- Mapped memory
- Unified memory

The difference in terms of the API calls, use cases, and mechanisms, will be explained further subsubsection 2.4.2.

2 Programming in CUDA

By now, we have talked about the architecture of the GPU [6] by briefly discussing different notions - the execution model of threads and kinds of memory. Now, we will try to look at examples of CUDA codes and programs. We will try to refer to all these prior concepts to get a more detailed understanding. One may need to have a look at the section above to link the theory with the practice discussed here. I will do my best to choose the most illustrative examples (of those available in different resources) to explain specific concepts, as well as introduce some tools. Also, note that I use Linux. (section).

2.1 Setup

Before starting to write some code, we must install the necessary packages and libraries. Every computer and operating system has its own subtleties, so the best way to install necessary tools, one should check the documentation for the specific system [7]. For Linux, the main packages are `CUDA` and `CUDA-toolkits`. If we've already used our computer for some time, we probably have Nvidia drivers installed.

When we run a `C` program, we need a compiler - arguably, the most common one ~~and the best one~~ is `gcc`. For `C++`, we invoke his *improved* version `g++`. For CUDA programs, however, we need NVidia's CUDA compiler - `nvcc`. As we've discussed, the program consists of the host and device codes. The device code is just plain `C`/`C++` code, so `nvcc` is also able to compile plain `C`/`C++`code. Note that the CUDA code has a `.cu` extension.

```
$nvcc -o main main.cu
$./main
```

Listing 1: Compiling with nvcc and launching a CUDA program on Linux

If both of the programs do not return an error, the code has been compiled and launched successfully.

2.1.1 Hello from CUDA

Let's create our first CUDA and C++ program, and discuss every step below.

```
1 #include <stdio.h>    //for printf()
2 #define N_THREADS 4  //number of threads
3 #define N_BLOCKS 2   //number of block
4
5 __global__           //declaration specifier
```

[6] Mainly about the Nvidia's architecture, but it can be quite well generalized to other GPU's.

[7] See the installation methods and required components on `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`.

```
6  void hi_from_gpu(){  //kernel
7      printf("Hi from GPU\n");
8  }
9  int main(){
10     printf("Hi from CPU\n");
11     hi_from_gpu<<<N_BLOCKS,N_THREADS>>>();    //invoking kernel
12     cudaDeviceSynchronize();                  //synchronize CUDA program
13     return 0;
14 }
```

In the output of the following code, we ~~should~~ get first `Hi from CPU`, followed by 8 times `Hi from GPU`. Let's discuss some aspects of the above code. Line 5 contains `__global__` declaration specifier which tells the compiler that the following kernel (function) can be launched on the device. In `main()` function, on line 11, we call the kernel. This is the semantics to invoke a CUDA kernel. The *arguments* in the angle brackets `<<<,>>>` are the dimension of threads blocks respectively that the kernel will be run on[8] & . The type of these dimensions can be either an unsigned integer, or of the `dim3` type, which we will mention in a moment. In this case, the GPU will call 2 blocks, with 4 threads each. This indeed results in $8 = 2 \cdot 4$ invocations of `printf()`. Finally, from the main function, we call the `cudaDeviceSynchronize()` method, waits for all blocks before the main function returns.

2.2 Threads & blocks indexing

Now let's dive in deeper, and access threads and block indexing by modifying our `hi_from_gpu()` function:

```
#define N_THREADS 4
#define N_BLOCKS 2
__global__
void hi_from_gpu(){
    printf("Hi from GPU\n, from thread id %d and block id %d",
    threadIdx.x, blockIdx.x);
}
```

The output after executing from `main`,

```
   $./main
   Hi from GPU, from thread id 0 and block id 0
   Hi from GPU, from thread id 1 and block id 0
   Hi from GPU, from thread id 2 and block id 0
   Hi from GPU, from thread id 0 and block id 1
   Hi from GPU, from thread id 1 and block id 1
   Hi from GPU, from thread id 2 and block id 1
```

We thus discovered how to access block's and thread's id's. The variables `threadIdx` and `blockIdx` are of type `dim3`, which is a simple structure, containing 3 unsigned int's. By

[8] We will see further, some additional stuff can be passed into these brackets, such as the size of memory and streams - concepts that will be seen in further sections

accessing its `.x` member variable, we are reffering to the 1D indexing. In general, threads are indexed using `dim3` type. Thus, for a thread, living inside a block, there exist x, y and z dimensions. And same for the blocks, living in the grid - x, y and z components. Think of threads, We can launch many threads in many blocks, but how many? This depends on the hardware we are using. To get these specifications, one can look up this information online or ask our computer.[9].

2.3 Memory

2.3.1 Vector addition

The first *useful* example that is usually introduced in CUDA tutorials is the vector addition. This exaple perfectly illustrates the need of GPU parallel model. Indeed, the component of the resulting vector x x_i does not depend on other components. So the formula for vector addition is given by $x_i = a_i + b_i$. In sequential execution, we would create a loop, iterate over all elements and do something like `x[i] = a[i] + b[i]`. In order to parallelize this workflow, we must initiate N threads, with N = number of components in the vector. In `C++`, which simply runs on the host, we create the threads using the standart `std::thread`. We know, however, that the CPU does not have many threads (probably from 4 to 12 in most cases). This is where the GPU with its multithreaded architecture comes in. The idea here is to launch N threads which are spread over many blocks.
To discover memory allocation with code examples, we will consider vector addition. And to do that, one must first cover thread & block indexing for a more general case.

A more complex indexing. We know that the number of threads in one block is limited. Let's check on Nvidia's official CUDA documentation [2] concerning threads indexing.

> *For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. 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 (Dx, Dy),the thread ID of a thread of index (x, y) is $(x + y \cdot Dx)$; for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is $(x + y \cdot Dx + z \cdot Dx \cdot Dy)$.*

[9]To do so, one must find where `cuda` is located. In my case, it is in `/opt/cuda/`. Once here, we seek for `/samples/1_Utilities/deviceQuery/` (or something similar). From here, we execute `./deviceQuery`.

Before writing some code, let's try to visualize the Nvidia's quote.

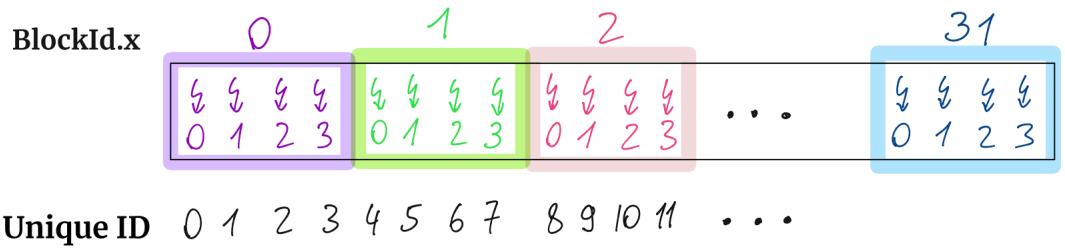


Figure 5: Simple example to illustrate a plain 1D indexing (supposing a block contains 4 threads). We see how the expression blockIdx.x*blockDim.x+threadIdx.x *is used (see below). One can easily extrapolate the indexing to 2D and 3D cases, as described in the Nvidia documentation above.*

Let's now write code to add two vectors, using a 2D indexing [10]:

Ok, there are multiple aspects to discuss...

First let's discuss the code. First on the device. Remember we discussed the global GPU memory. The line 31 we're working with memory. The classical pipeline of any CUDA program execution, is usually the following:

1. Begin code on host.
2. Allocate memory on host.
3. Allocate memory on device.
4. Calculations performed on the device.
5. Host treats the data.
6. main() gets returned.

Allocate memory on host. To allocate memory on the host, we proceed with the usual C `malloc()` function and naturally casting the returned pointer to `double*` (don't forget to free memory at the end by `free()` function). Then we call a simple function `init_host_vector()` that will just initialize the data to some values, let's say to simple $0, 1, 2, 3, ...$

Allocate memory on device. Then we allocate **global memory** on the GPU, by calling the `cudaMalloc()` function. Note the arguments that this function takes - a **pointer to pointer** - `(<type> **)` and the size in bytes.

[10] This is the first and almost the only **full** example of a cuda program. That is, the author will mostly provide the key aspects of the newly introduced features, thus skipping the usual parts of defining things, of allocating memory, freeing it, etc.

```
#include "stdio.h"
#define N_THREADS 512
#define N_BLOCKS 64
void init_host_vector(double *a, double *b);
void check_result(double *res);

__global__
void add_vec(double *a, double *b, double *res){
    //compute the index
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if(id < N_THREADS*N_BLOCKS){
        res[id] = a[id] + b[id];
    }
}

int main(){
    const int size_in_bytes =N_THREADS*N_BLOCKS*sizeof(double);
    //initialize the data on HOST
    //malloc() (C) or new (C++)
    double *hst_a = (double *)malloc(size_in_bytes);
    double *hst_b = (double *)malloc(size_in_bytes);
    double *hst_res = (double *)malloc(size_in_bytes);

    init_host_vector(hst_a, hst_b);

    //allocate memory on GPU
    double* dv_a;    cudaMalloc(&dv_a, size_in_bytes);
    double* dv_b;    cudaMalloc(&dv_b, size_in_bytes);
    double* dv_res;  cudaMalloc(&dv_res, size_in_bytes);

    cudaMemcpy(dv_a, hst_a, size_in_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(dv_b, hst_b, size_in_bytes, cudaMemcpyHostToDevice);

    add_vec<<<N_BLOCKS, N_THREADS>>>(dv_a, dv_b, dv_res);
    cudaDeviceSynchronize();
    cudaMemcpy( hst_res, dv_res, size_in_bytes, cudaMemcpyDeviceToHost );

    check_result(hst_res);

    cudaFree(dv_res);   free(hst_res);
    cudaFree(dv_a);     free(hst_a);
    cudaFree(dv_b);     free(hst_b);
    return 0;
}
```

Listing 2: Basic vector addition, using sequential thread indexing. [18]

Data copying between the host and the device is done using the `cudaMemcpy()` function. Also, pay attention to the parameters - `cudaMemcpy(void* destination, void* source, size_t size_ enum cudaMemcpyKind kind)`, with `kind` [11] specifying how to copy. In code, it is understandable that the destination is the device and the source is the host.

Calculation. The kernel computes the sum of a vector of size $512 \cdot 64 = 32768$. Therefore 64 blocks of 512 threads are launched, and 32768 threads independently execute the `add_vec()` function. In the kernel, a unique ID is assigned to the thread(line 10) (see the indexing *policy* see 5). The ID's span from 0 to 32768, without ever repeating themselves and covering all the numbers in the interval. We also add a simple `if()` statement, to be sure, that the thread's ID does not exceed the size of the vector. Thus every thread does exactly one calculation [12]. Remember that the memory, in which these processes are happening, is the global memory.

Terminating. After the kernel, we call the usual synchronization function, which will, as usual, wait for all the threads and blocks to finish the calculations. Then, a simple check function on the host, simply to ensure, that the parallel vector addition was successful and that we've received the correct result. And finally, the `cudaFree()` method, does exactly what is expected - frees the allocated memory on the GPU, the exact same things, as the `free()` does in a usual `C` code.

Let's also try to link this piece of code to the section about warps. Remember, threads are scheduled by the SM into warps - groups of 32 threads (subsection 1.2). In this case, we're launching a total of $^{512}/_{32} = 16$warps for each block, and 64 independent blocks. While working with CUDA threads, it is advised to work with the number, which is a multiple of 32 (the thing we've done in this example). Indeed, imagine, if we were to launch 513 threads on a block. Then the SM would schedule 17 warps. This number of warps is the same as if we would launch 544 threads in a block. Indeed, this will launch $16 \cdot 32 = 512$ threads (executed simultaneously in one warp) **and** one additional thread on an almost empty/unoccupied warp.

One could also be interested in the difference in the execution time between 2 different configurations - either $64\text{threads} \cdot 512\text{blocks}$ or $32\text{threads} \cdot 1024\text{blocks}$ The benchmarking is very important and useful in GPGPU programming. Different tools and techniques for benchmarking CUDA programs will be discussed in further sections.

2.3.2 Multiple dimension memory

By now, we've looked into the global one-dimensional memory allocation (and copying) on device. Suppose you would like to work with 2D allocated memory. It is clear that we could work without the ability to explicitly allocate 2D or 3D arrays. Indeed, if we want to work

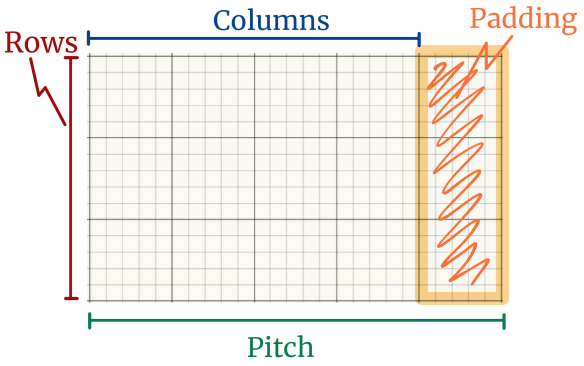
[11] Here, I described the parameters in details. However, further on, the author will not be that precise. The reference for this function was taken from `http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html`. For further function descriptions, always have a look at it.

[12] Of course, without taking into account the calculation of the id.

with, let's say, a 2D matrix, we could simply transform it into a 1D vector of size rows·cols. However, there is a feature in the CUDA API, that lets us to allocate 2D, and even 3D memory [11]. The method that implements 2D memory allocation is `cudaMallocPitch` with the following signature:

```
cudaMallocPitch( void** devicePtr,
                 size_t* pitch,
                 size_t widthInBytes,
                 size_t height);
```

The signature of function. Some new terminology given on the right.



This function is allocating **at least** width×height bytes array As we have seen, the allocation on the device is followed by the data copy from the host. There is also a special function to do so - `cudaMemcpy2D()`. Apart from allocating and initializing the device data, we should be able to access it. So let's consider a code snippet that does so:

As you might have guessed, this memory access is not extremely efficient. As one may say, THIS IS FOR EDUCATIONAL PURPOSES ONLY. In the documentation, it is recommended to use these functions to allocate and initialize the 2D memory on device. Hopefully, we all can find use cases of this memory and efficient indexing [13].

2.4 More on Memory

2.4.1 Shared memory vs global memory

For now, in the examples, we've only looked into the global (& pinned) memory. It is global and accessible for all threads in all blocks for both read and write operations, and yet having high latency, compared to more local memory types, such as the shared one. Shared memory's scope is one thread block. Thus, if, let's say, a block contains 16 threads, the shared memory can only be read and modified by 16 local threads. If one wants to operate on it locally, the data must be copied to it. Therefore, if one wants this memory to be accessed/modified by 2 thread blocks (= 32 threads), one must make sure that the data has been copied to the shared memory of the two blocks.

[13] There are a lot more of memory allocation, & memory copy methods, provided in the CUDA API. For example `cudaMallocArray()`, which has also a different signature. The different methods, dedicated to memory allocation and management can be observed `https://courses.cs.duke.edu/cps296.3/fall09/cuda_docs/html/group__CUDART__MEMORY_ge56101fe6f1ce0b48f163632f6862ae4.html#ge56101fe6f1ce0b48f163632f6862ae4`

```
int main(){
   float *A, *dA;
   size_t pitch;

   A = (float *)malloc(sizeof(float)*N*N); // allocate on host
   cudaMallocPitch(&dA, &pitch, sizeof(float)*N, N); // allocate on device

   //copy memory
   cudaMemcpy2D(dA,pitch,A,sizeof(float)*N,sizeof(float)*N, N,\
      cudaMemcpyHostToDevice);
         /*...*/
}
__global__ void access_2d(float* devPtr, size_t pitch,\
            int width, int height) {
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Listing 3: To get this straight, one should know 2D array work in pure C. Indeed, to access an element, we first identify the current row, by doing some pointer manipulations. And then, once the pointer to the very beginning of the row is identified, we iterate over this row, by accessing it sequentially. Also note that in the code, in order to identify the row for the next iteration (i.e. it's 0*'s element pointer), one uses the size of the pitch, and not of the width. We see that this 2D pitch is allocated* ***automatically*** *by the CUDA memory management system [10]*

Let's now consider a more complex and yet classical example - matrices. Recall basic matrix operation -

- If A - $m \times n$ matrix (m rows and n columns), it can be added to some other matrix B of same size $m \times n$, by adding element-wise each element $a_{i,j} + b_{i,j}$.
- One can multiply two matrix A and B - $C = A \cdot B$ by using the formula $c_{i,j} = a_{i,1} \cdot b_{1,j} + a_{i,2} \cdot b_{2,j} + ... + a_{i,n} \cdot b_{j,n}$, only if the matrix A is of size $m \times n$ and B of size $n \times p$. In short, the resulting element $c_{i,j} = \sum_{k=1}^{k=n} a_{i,k} \cdot b_{k,j}$ and yields C of size $m \times p$.

One can perform these operations using the global memory, or shared memory.

Global version code snippet is given down below.

Before moving to the code discussion, let's make a small *intermezzo* on transforming 2D array to 1D, to make things clear and easier for further analysis:

```
#include "stdio.h"
#define BLOCK_SIZE 16

typedef struct{
    int height; int width; float* element;
}Matrix;

__global__ void matmul_global(const Matrix a,\
const Matrix b, Matrix c);

int main(){
    //init matrices A and B
    Matrix A; A.height = 32; A.width = 32;
    Matrix d_A; d_A.height = A.height; d_A.width = A.width;
    int size = sizeof(float)*d_A.height*d_A.width;
    cudaMalloc(&(d_A.element), size);
    cudaMemcpy(d_A.element, A.element, size, cudaMemcpyHostToDevice);
    /* same for d_B
        ...
    */

    //prepare memory, for device to write to
    Matrix d_C; d_C.height = d_A.height; d_C.width = d_A.width;
    cudaMalloc(&(d_C.element), size);

    //prepare dimensions of the kernel (2D indexing)
    dim3 block_dim = (BLOCK_SIZE, BLOCK_SIZE);  //dimension of block
    dim3 grid_dim = (A.width/BLOCK_SIZE, A.height/BLOCK_SIZE); //dim. of blocks grid
    matmul_global<<<grid_dim, block_dim>>>(d_A, d_B, d_C);

    /*
    cudaMemcpy(...); free(...); cudaFree(...); //free the ressources
    */
}

__global__ void matmul_global(const Matrix A, const Matrix B, Matrix C){
    int row_id = blockDim.y*blockIdx.y + threadIdx.y;
    int col_id = blockDim.x*blockIdx.x + threadIdx.x;

    //accumulate sum for c_{row_id,col_id} element
    float tempsum = 0.0;
    for(int k = 0; k<A.width; k++){
        tempsum += A.element[row_id*A.width + k]*\
                   B.element[k*B.width + col_id];
    }
    C.element[row_id*C.width + col_id] = tempsum;
}
```

Listing 4: Basic, yet important, global memory usage. [18]

Suppose we are given a 2D array (matrix). But we know that, behind the scenes, (even if we're accessing a[i][j] in many programming languages), it all breaks down to contiguous, linear memory addresses. So suppose you have allocated a 1D array arr of size N. The, you could access your i'th element by doing `arr[i]` (supposing C or C++) or by doing `*(arr + i)`, where `arr` - the address of the 0'th element of the array . So the formula is given by $\mathrm{A}_{i^{th}} = A_0 + \text{sizeof(type)} \cdot i$, where A_0 - the first (0'th) memory address and sizeof-the size of one memory field (e.g. sizeof(char) = 1). Suppose now, that we've allocated a 2D memory array of size Mrows $\times$ Ncols. For the first row one can apply our previous formula - $A_{0,j} = A_0 + \text{SZ} \cdot j$. However, if j is greater than N - the number of columns, a problem occurs. If we work in 2D indexing, over a loop, we would just *reset* our index j to 0 and increment i (if $j = N-1$). However, in a flattened 2D case, in order to access $A_{i,j}$ address-wise, we use the expression $A_{i,j} = A_0 + \text{sizeof(type)} \cdot N \cdot i + j)$. One can easily check, that this expression is consistent with all the examples. This expression is for row-major arrays - accessing successive elements in a certain row.

Okay, now we can attack the code. From line 13 to 21, we are initializing the data and allocating memory. The lines 27 & 28 are initializing the dimensions (number of threads in x,y directions and the number of blocks in the x, y directions), that the kernel will be launched on. Note that we are using 2D indexing for the first time. In this case, `grid_dim` is the dimension of the grid, in which blocks are contained. The `block_dim` is the block dimension - the number of threads in the block. In this case there are $16 \cdot 16 = 256$threads and $\frac{32}{16} \cdot \frac{32}{16} = 2 \cdot 2 = 4$blocks. So there are $256 \cdot 4 = 1024$threads partitioned between 4 blocks. This is exactly what we need, because the resulting matrix C is exactly 32×32, which gives us 1024 elements. Therefore, if everything goes well, each thread will perform the calculation for each element $c_{i,j}$ of the matrix C. In theory, we would like that each thread iterates over one line of the matrix A and one column of B. Remember the 1D to 2D mapping. is going through all the elements in the row i of the matrix A (line 43).

$$A_{i,h} = A_0 + i \cdot N_{cols(A)} + h$$
$$B_{h,j} = B_0 + h \cdot N_{cols(B)} + j$$

These equations (at least the first one) are exactly predicted by 1D to 2D mapping and are represented on the lines 43 and 44 of the code snipped. Finally, once each thread runs over its own row and column, it is gathering the result by assigning this accumulated sum to the `C[i][j] = C[i*Width + j]`, where i,j - row id's and column id's respectively[14]

Shared memory is one of the key, to control the efficiency of a CUDA program. As we've mentioned several times before, the shared memory is only visible in the block's scope. That means that if, we would want to make the shared memory visible & accessible to all

[14]This aspect is not simple yet basic. This is a common CUDA pattern, which needs to be understood correctly ~~(or at least know where to look for)~~. So one should try to visualize the indexing process and the algorithm workflow.

threads in the entire kernel, we would need to allocate (and transfer the data to) shared memory, which belongs to independent blocks. Remember the analogy of the grocery store. It's up to the programmer to decide, whether it would be reasonable to spend time on algorithm design, using shared memory, and its allocation (which, of course, takes time, but has little latency, when using it), or to high-latency global memory access. So let's have a look at a shared memory code snippet[15]. demonstrating shared memory usage for matrix multiplication. Let's first discuss the general strategy. As we have said, the shared memory is much smaller in terms of size than the global one. So here, the main idea is to divide the *big* matrices A and B, that we're multiplying into smaller sub-matrices, which will be loaded into the shared memory later of each independent block.

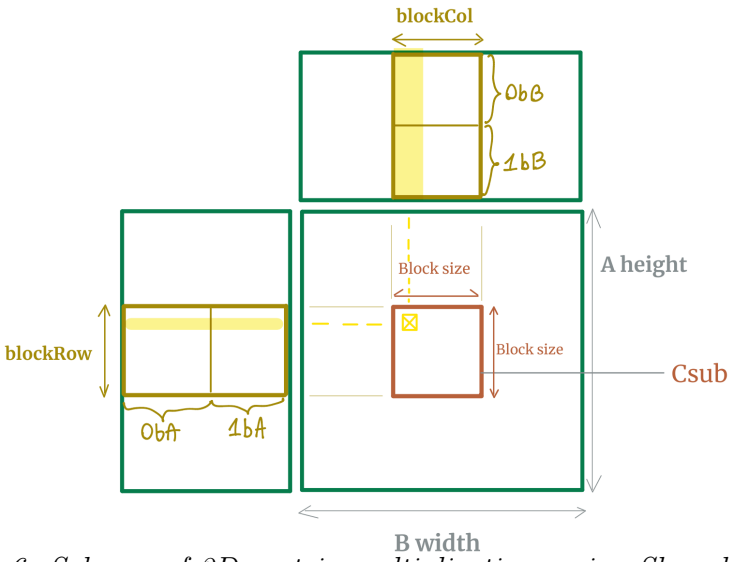


Figure 6: Scheme of 2D matrix multiplication, using Shared memory

Let's look at the code describing this strategy:

```
typedef struct{
   int width; int height; int stride;
   float *elements;
}Matrix;

__device__ float GetElement(const Matrix D, int row, int col)
{
    return D.elements[row * D.stride + col];
}

__device__ Matrix GetSubMatrix(Matrix D, int row, int col)
{
    Matrix sub;
    sub.width    = BLOCK_SIZE;
    sub.height   = BLOCK_SIZE;
    sub.stride   = D.stride;
    sub.elements = &D.elements[D.stride * BLOCK_SIZE * row
```

[15]For space-saving's sake, only a snipped is provided. The skipped pieces are usual steps of memory allocation, freeing resources, etc...

```
                                            + BLOCK_SIZE * col];
    return sub;
}

 __global__ void mult_global(Matrix A, Matrix B, Matrix C)
{
    // blockRow & blockCol (see image)
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Create Csub, initial matrix
    Matrix Csub;
    Csub = GetSubMatrix(C, blockRow, blockCol);
    float Cvalue = 0; //we will accumulate values (see figure above)

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B
    // Multiply each pair of sub-matrices together
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {//iterate over
                                            //sub-matrices//of A(see fig above)
        Matrix Asub=GetSubMatrix(A, blockRow, m);//Asub of A(m-the index of row)
        Matrix Bsub=GetSubMatrix(B, m, blockCol);//Bsub of B(m-the index of col)

        // shared memory to store Asub and Bsub
        __shared__ float Asub[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bsub[BLOCK_SIZE][BLOCK_SIZE];

        // Each thread loads one element of each sub-matrix Asub and Bsub
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        //All threads must be synced, to be sure all data is loaded properly
        __syncthreads();

        // Use matrix multiplication formula to get the Csub element
        for (int e = 0; e < BLOCK_SIZE; ++e){
            Cvalue += Asub[row][e] * Bsub[e][col];
        }

        __syncthreads(); //synchronize before new sub-matrices are loaded
    }
    C.elements[row * A.stride + col] = Cvalue;
}
   int main(){
      /*init matrices on host
      * init matrices on device with cudaMalloc(),
      * copy data from host to device
      */
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
   mult_global<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
   }
```

Okey, let's now discuss in detail the execution of the kernel , and try to make as more analogy

as possible with the scheme above.

1. The matrices are assumed to be multiples of BLOCK_SIZE (block dimension in fact). In this case, let's say 32×32. So, from the main function, we're launching 32 threads in each block, and $2 \times 2 = 4$ blocks in a grid. Thus we have $32 \times 32 \times 4 = 4096$ threads, organized in a grid of 4 blocks. So there is a kind of mapping between the 2D matrix and the threads partitioned into blocks. Keep that launched chunk of threads in mind.

2. The first thing we do in the kernel is attribute the block row and column ids. They correspond to blockRow and blockCol, annotated in the figure 6. There will thus be 4 sub-matrices to which, to which each block will be assigned to (as we've said that the block is of size 16×16 and the matrix size is 32×32). Remember that the shared memory is **only visible inside a block**. Thus the possible configurations of the indices are given by $(A_{row} = 0, B_{col} = 0)$, $(A_{row} = 0, B_{col} = 1)$, $(A_{row} = 1, B_{col} = 0)$ and $(A_{row} = 1, B_{col} = 1)$. These indices correspond to the *olive color* strips on the figure, labeled with BlockRow and BlockCol respectively.

3. Then we're initializing the sub-matrix C_{sub} - its dimensions, elements and stride. The stride, in our case, is the offset we need to take into account when computing the elements in the row-major form. Note that it is not always the same as matrix width. For example, when we're dealing with the submatrix, the stride does not equal to its width, but rather **the original matrix width**. Indeed, the size of the submatrix does not affect the element's addresses in memory. This sub-matrix C_{sub} is illustrated in figure 6.

   This gives us the ability to keep track of the sub-matrix within each block. Remember, the multiplication of (sub-)matrices involves the accumulation of element-wise multiplication, so we're initializing the temporary sum to 0.

4. The initialization is followed by the initialization of sub ids. These are the thread ids **within a block**. Thus these id's will be uniquely identified for each thread within the block, launched on a unique kernel instance. And, as we know, each block will be responsible for one sub-matrix of C - C_{sub}. These ids are represented by yellow rows and columns on figure.

5. Now, an important assumption will be made - the number of blocks contained in A's matrix width is the same as the number of blocks contained in the B's matrix height[16]. This is the reason why we can use the first loop, to iterate over A'th width and B'th height *simultaneously*.

6. As mentioned just above, we can simultaneously iterate over A'th width and B'th height, thus simultaneously obtaining the sub-matrices A_{sub} and B_{sub}. After getting the sub-matrices A_{sub} and B_{sub}, we allocate shared memory `As` and `Bs` for A_{sub} and

[16]This is an assumption, that is, as we say W.L.O.G - without loss of generality. Indeed, if this assumption is not the case, one could modify a bit the code for it to work without this assupmtion. However, this is neither a problem, because, one can allocate and perform operations on a bit bigger matrices, without big issues in performance

B_{sub} using the `__shared__` directive. Note here, that each thread in the block is retrieving **one, and only one element** of the matrix. Indeed, the `getElement()` method retrieves only one element from global to local memory. This is the most important step in this 2D matrix multiplication.

7. After each thread retrieves the data **within the block**, we would like all threads to be done with synchronization. Indeed, in global memory's case, each thread accesses the global memory independently, with a unique ID. Here, with the shared memory, we introduce a kind of independence between threads, by making them all access to the same block-local shared memory, by making them all retrieve local data. So, for example, suppose we have a 4×4 sub-matrix. Each thread within the block loads one element into shared memory (seen by the other 15 threads). So before making the calculations, we want all the 16 data to be loaded into shared memory.

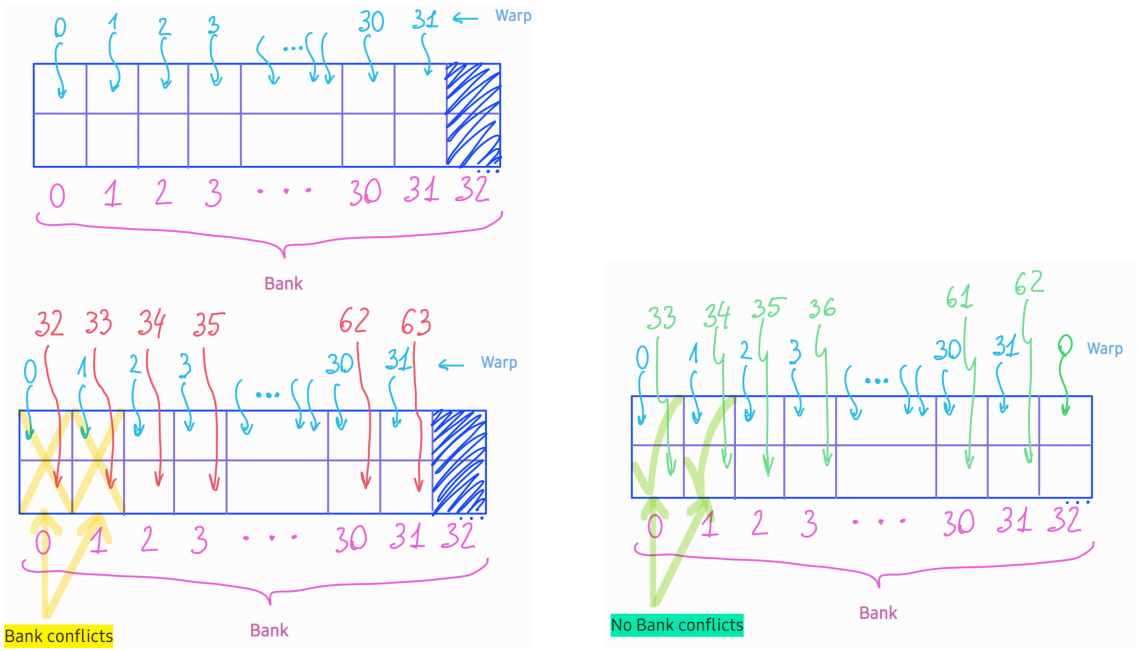
8. Afterwards, we are finally doing calculations on the sub-matrix, using the previously retrieved elements of A_{sub} and B_{sub} into `As` and `Bs` respectively. Remember here that `row` and `col` are unique for each thread. Once again - This step is illustrated in the figure: the iteration is performed through the yellow line's multiplication element-wise. And the **unique** elements, attributed to the threads are the `row` and `col` indices. This is the final result we want to achieve: *each thread within the block multiplies the **row** of A_{sub} and the **col** of B_{sub} element-wise, to get only one element of the C_{sub} sub-matrix.*

9. The final step of the kernel is to copy each element of the C_{sub} sub-matrix to the global memory. Once again, as each thread is doing its own $Csub_{i,j}$ element, it is just copying only **one** element to global memory.

Note that apart of the `__shared__` directive, other, new technical aspects were used. First - the `__device__` directive. It declares a function, which can **only** be called from the device function (the one declared with `__global__`). These device functions can return different types. Second - the shared memory with the `__shared__` directive. As discussed above, this is shared memory, only seen within the block.

As mentioned, this is a very fundamental example, yet not simple, and takes time to understand. The reader ~~and the author~~ must go through the code and the illustration several times. As well as going through multiple particular instances of the code, with particular instances of the block and threads ids.

Bank conflicts. We have practically discussed several applications of the theoretical notions described in the section on architecture. How about the bank conflicts? Remember - the warps are chunks of threads (way of scheduling/organizing them). The banks are chunks of memory and a way to structure them. Also remember that, ideally, 32 threads, organized in one warp, access simultaneously one "column" at a time Figure 4. So if two threads in a warp access simultaneously one bank (one column), there is a bank conflict, and the warp will be forced to get split into two execution periods. The classical bank conflict is often illustrated by the 2D array initialization in shared memory.



(a) *(b)*

Figure 7: Bank conflicts

Once again: suppose we are simply allocating a 1D array (either in global or shared memory) of size 32 bytes. Then one warp is accessing one bank of memory. Suppose now we're allocating 64 or 128 bytes of memory, then 2 or 4 warps respectively will access the memory sequentially, i.e. 2 and 4 periods respectively.

So, in order to avoid these bank conflicts in 2D memory (of course, it is always possible to map a 2D array into 1D array, which is much easier to deal with), one should allocate *a bit more memory* than required. Take a look at the Figure 7: the two first figures (a & b) show a 1D and 2D array memory access respectively. On the LHS, the memory access is a multiple of 32. Therefore, at the same time, more than 2 columns - **banks**, are accessed by the same warp (multiple of 32). In order to get rid of a bank conflict in a 2D memory access, whose size is multiples of 32 (warp's size), we allocate one additional byte to each row (or 4 bytes if it's a float). To do so, we replace `__shared__ float Mat[BL_SIZE][BL_SIZE]`, with `__shared__ float Mat[BL_SIZE][BL_SIZE+1]`. Indeed, if the addresses are multiples of 32, each warp will have to access the same bank/memory address as the previous and the next bank. So, what one can do, is to allocate memory of size 33 rows. Thus, the threads, seeking memory, will access different banks(columns).

The author does fully understand and does know that this is a quite subtle aspect, and one needs time to get used to these concepts. It is also clear that these things are difficult ~~or even impossible~~ to debug and track, once the code is written, and the programmer does not have any ideas of where to look for the performance issues. In most cases, the *solution* is to simply benchmark the execution of the program, and consequently test the program with different configurations.

2.4.2 Memory mapping

We have discussed multiple kinds of memory - **Global**, **Shared**, **thread-local** memories. All of them have different scopes, bandwidths, and properties(such as bank access). These memories are classified by their scope, not by the way they are allocated and transferred (see 1.4).

Lets now treat and have a look at different memory allocation types [17]:

- **Pageable data transfer.** When we invoke the *classical* `cudaMalloc()`, we're acessing pageable memory. It is the default behaviour. Under the hood, it is allocated twice - first to pinned memory(see below), then, from the pinned one, to the device global memory.

$$\text{HOST} \xrightarrow{allocates} \text{Pageable} \Rightarrow \text{Pinned} \xrightarrow{goes\ to} \text{DEVICE} \tag{1}$$

  One can think of it as the longest path between the host data, and the required device data. The syntax for this process is:

  1. In order to allocate memory on the host, one uses `malloc()` or `new`.
  2. In order to allocate the memory on the device `cudaMalloc()`.
  3. In order to copy memory from the host to device and vice-versa, one use the `cudaMemcpy()`.

- **Pinned data transfer** lets us avoid 2-way memory allocation between the host and the device. We are thus skipping one step in the memory allocation pipeline. Pinned memory is faster, yet reduces the performance from the host side, as the amount of memory for host processing is reduced. The syntax for pinned memory is given by:

  1. In order to allocate memory on the host, one uses the `cudaMallocHost()`.
  2. In order to allocate the memory on the device `cudaMalloc()`.
  3. In order to copy memory from the host to device and vice-versa, one use the `cudaMemcpy()`.

- **Mapped memory**, also called *zero access memory* [17], is pinned memory, which is **directly** initialized/mapped to the device address. This gives the ability to leverage the device's memory using the host if it is not big enough. This also gives a lower memory transfer time. As there is in a way, **no** memory transfer. The syntax for the zero access memory usage is given by:

  1. In order to allocate the memory, one uses the `cudaHostAlloc(void** host_ptr, size_t size, int flags)`, where the flags, e.g. `cudaHostAllocMapped`.
  2. In order to get the communication channel between the host and device, one uses `cudaHostGetDevicePointer()`.

```
for (int i = 0; i < nStreams; ++i){
  int offset = i * streamSize; //offset for memory copy
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,\
    cudaMemcpyHostToDevice, stream[i]);
  kernel<<<streamSize/blockSize, blockSize, 0,\
    stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,\
    cudaMemcpyDeviceToHost, stream[i]);
}
```

*Listing 5: The code snippet [16] is giving an example of how to use the asynchronous memory copy. Thanks to the streams combination with the async memory copy, the only synchronization that is happening is the memory copy and kernel execution, inside a separate stream. That is, if no streams are specified, every iteration is will wait for the previous one to finish. For details, see [**?**].*

You may notice that there is no 2 memory allocations - from the host and device sides. Instead, there is only one memory allocation, and one *pointer sharing*. This emphasizes the *zero mapping* fact of the mapped memory. The flaw of this method is the increase of processing time, as the data is simply mapped, and not copied.

- **Unified memory** is, as its name suggests, memory available both to the host and the device. The biggest advantage of it, is the reduction of the memory allocation pipeline. However, the latency of memory access is highly increased. In order to use and invoke the unified memory, one only uses the `cudaMallocManaged()`. There is no other functions to invoke, as the memory is initialized for both the host and the device.

2.4.3 Asynchronous memory

We have seen that both the memory access and memory allocation will affect the program execution time. Therefore when designing a program, one must take into account not only the memory access but also the memory allocation time. The asynchronous memory allocation may reduce the time of program execution, as it is asynchronous. And non-blocking. This concept of asynchronous memory allocation involves the concept of streams discussed further. Therefore the main part of this functionality is discussed in the CUDA streams section subsubsection 5.2.1. The asynchronous memory copy is a non-blocking memory copy. The async memory copy is invoked with the `cudaMemcpyAsync()` function. This can be for example used, when one wants to invoke many kernels, with many memory copies. Let's have a look at an abstract example, without going into details, of what is the kernels doing.

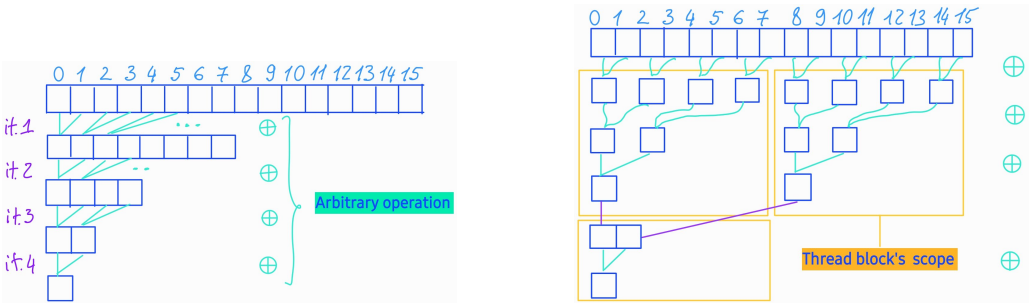


Figure 8: Reduce algorithm with differnet strategies.

3 Basic parallel algorithms & patterns

3.1 Reduce

Consider the situation, when we need to add all the array's elements together. In `C++`, one could use the STL algorithm `accumulate()` function. In a more primitive implementation, we would do a single for-loop and accumulate the results in one variable. The complexity of such an algorithm would then be about $O(n)$, where n - the length of the array. However, one could add some parallelism to this algorithm. Indeed, at first iteration, what we would do is add the 0th element with the 1st, the 2nd with the 3rd, the 4th with the 5th, etc... Notice that all of these $N/2$ additions can be done in parallel (that is, each thread does one addition). The next *iteration* would be summing the result of the sum of 0th and 1st with the sum of 2nd and 3rd, thus giving a total of $N/4$ additions (instead of $N/2$) . Therefore, the number of necessary divisions in the next step will be half of those, during the previous ones. As those divisions are performed in parallel, this algorithm looks like a log-scale complexity.

Global memory reduce

Let's first have a look at a ~~not-so-~~naive implementation of the discussed reduce algorithm on the GPU. Once again, we're looking at a simplified version of the code, without implementing memory allocation, copy, etc... (note that in this case, we use simple global memory with `cudaMalloc()`). The implementation of this is given by 6.

Okay, let's discuss the 6. First, in the host function, we define the number of blocks. In this case, this number is not so important. It would be important to optimize the execution, by taking into account the notion of warps, etc... The important part is the loop in the host code and the device kernel. We will try to do the debugger's job and inspect the steps. During the first iteration, the value of `stride` is 1. The important point is that the thread id, computed in the kernel does not depend on the value of the stride. This is because we're working with the global memory, and the access is global.

Suppose the size is 32, partitioned into 1 block. Then for the first iteration, we'll get,

```
__global__ void reduce_global_kernel(float *data_out,\
                float *data_in, int stride, int size) {
int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
if(idx_x + stride < size){
data_out[idx_x] += data_in[idx_x + stride];
}
}

void reduce_global(float *d_out, float *d_in, int n_threads, int size) {
int n_blocks = (size + n_threads - 1) / n_threads;
for (int stride = 1; stride < size; stride *= 2){
   reduce_global_kernel<<<n_blocks, n_threads>>>(d_out, d_in, stride, size);
}
}

//main()
```

Listing 6: Global memory reduction. [18]

`out[0] = [0]+[1]`, `out[1] = [1]+[2]`, ... `out[30] = [30] + [31]`. This is exactly the first iteration, shown in Figure 9 (the case of size=8). Then, during the next iteration, we're *jumping* over 2 next elements and adding them, in order to get the sum of N_{stride} elements, and save them into the `data_out[0]`. Let me mention, that this process is illustrated in the image above Figure 9.

This algorithm is, maybe, not easy to understand, but is very fundamental parallel algorithm. Both the algorithm and the way of analyzing the problem. However, it can be optimized using the block's shared memory.

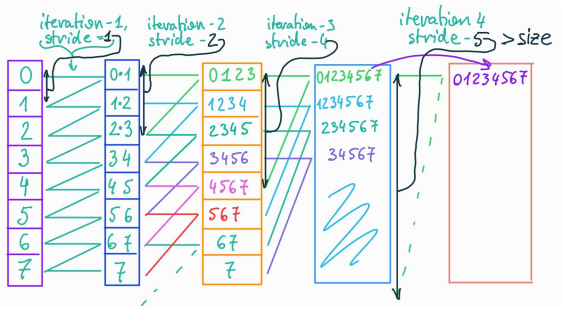


Figure 9: The description of every iteration, for the global memory reuction kernel. Note the how stride is doubling every iteration, ,and how the elements are accumulating in the very first (0th) element of the output data. One should understand, that the code works for arbitrary number of blocks, as we're working with global memory, visible to all threads in all blocks

Shared memory reduce

As we've discussed several times above, shared memory access has low latency. The idea is thus to do a local copy of the data to each block. As the shared memory's size is limited, we **map** the data pieces to each block (see figure).

Now, let's try to understand it together in more detail, using well-defined numbers of threads and blocks, to make things more illustrative. Trying to keep in mind both the illustration (Figure 10).

We omit the `main()` function and the host/device memory allocation/copy. Line 2 declares the

```
   /*Perform the necessary declarations, main(), before/after, etc...*/
void reduction(float *d_out, float *d_in, int n_thr, int size){
   cudaMemcpy(d_out, d_in, size * sizeof(float), cudaMemcpyDeviceToDevice);
   while(size > 1){
      int n_bl = (size + n_thr - 1)/n_thr;
      reduce_shared<<<n_bl, n_thr, n_thr*sizeof(float), 0>>>(d_out, d_out, size);
      size = n_bl;
   }
}

__global__ void reduce_shared(float* d_out, float* d_in, unsigned int size){
   int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
   extern __shared__ float s_data[];
   s_data[threadIdx.x] = (idx_x < size) ? d_in[idx_x] : 0.f;
      __syncthreads();

// do reduction
   for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
      // thread synchronous reduction
      if ( (idx_x % (stride * 2)) == 0 ){
         s_data[threadIdx.x] += s_data[threadIdx.x + stride];
      }
      __syncthreads();
   }
   if(threadIdx.x == 0){
      d_out[blockIdx.x] = s_data[0];
   }
}
```

Listing 7: Optimized reduce, with shared memory. [18]

function, which will be calling the kernel. Line 3
copies the memory `cudaMemcpyDeviceToDevice`. This is done in order to make sure we're working with the same memory locally allocated on the device. This is illustrated with the bottom blue arrow.

First iteration For the first iteration, we're associating the variable `int size` to be the litteral number of elements to be reduced. In our case, it is 32. The next declared variable `int n_bl` is, as the name suggests, the size of the block. As we're working with the shared memory, each shared memory will be allocated for one specific block, as shown in figure. In our case, we've chosen *nice* numbers, so that `n_thr * n_bl = size = 32`, with `n_thr` - the number of threads per each block. On line 6, we're finally invoking the kernel with the initial 8×4 dimensions (the first 2 parameters in the angle brackets). The 3rd parameter in the angle brackets is the size of shared memory, that will be assigned to each block. We see this syntax for the first time. This is how we allocate the **dynamic shared memory**, which we will discuss a bit later, as well as the last parameter 0 (ignore that too for the moment). For now, this is just shared memory allocation, outside the kernel itself. So we've allocated $n_{threads} \times size_{float}$ - the exact amount of shared memory, that will be accessed by the 4 threads.

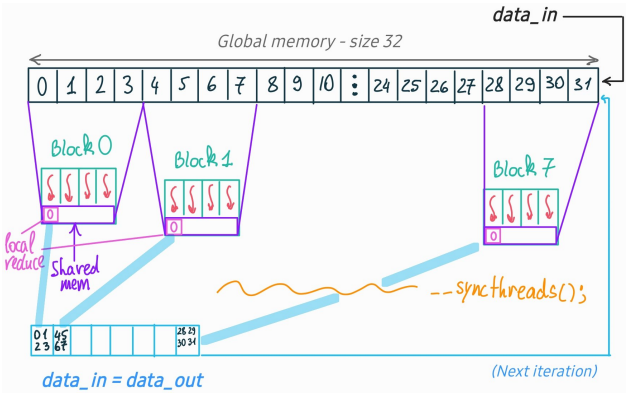


Figure 10: Reduce algorithm using shared memory.

Let's move to the kernel body itself. First, on line 12, our usual procedure, we're assigning a personal ID to each thread. Line 13 declares shared memory with size of $n_{threads} \times size_{float}$, specified outside, at the kernel call. This is the dynamic allocation of shared memory with `extern` keyword (as said, we'll discuss it later). On Line 14, we're copying the data to shared memory. Remember, the size of shared memory is the size of threads in the block (line 6). Thus the operation SH_DATA[LOCAL_THR_ID] is performed. Line 6 operation is illustrated with the violet lines - the mapping between the global to shared memory ($[0:3]_{global} \mapsto [0:3]_{block=0}$, $[4:7]_{global} \mapsto [0:3]_{block=1}, ...$). After copying, we are making sure that **all** the threads are done copying with `__syncthreads()`. Without that, some problems can occur (e.g.if we're starting reducing in 0'th block **before** all the threads within this blocks are done copying its data).

On line 17, we're starting to perform reduction. Let's first try to ignore the `if` condition on line 20, in order to better understand, why is it, and should be here. The dummy variable `stride` varies from 1 to the size of the block - $1 \to 2 \to 4 \to 8 \to 16 \to ...$ (in our case till 8).For example, for a local thread with local ID = 0, the sum of SH_MEM[0], SH_MEM[1], SH_MEM[2] will be accumulated within the loop. For thread with local ID = 1, the sum of SH_MEM[1], SH_MEM[2], SH_MEM[3] will be accumulated. For thread with local ID = 2,

the sum of SH_MEM[2], SH_MEM[3], SH_MEM[4], will be accumulated, and so on. We have now several problems, e.g. the access to SH_MEM[4], which is out of bounds, as the size of it is the save as number of threads (4 in our case).

Consider now the loop on the global scale/scope, as on line 20, we're checking the global thread ID. Then which threads will access the line 21, for the dummy variable `stride` = 1? These are threads with global ID $0, 2, 4, 6, 8, 10, 12, 14, 16, ...$, which corresponds to local threads $\{0, 2\}_{block=0}$, $\{0, 2\}_{block=1}, ...$ (see violet lines Figure 10). On next iteration of the dummy variable `stride` = 2, only threads with global ID's $0, 4, 8, 12, 16$ will access the line 21, which corresponds to local threads $\{0\}_{block=0}$, $\{0\}_{block=1}, ...$

Now, taking into account the two aspects, we can say, that for the first iteration of the dummy variable `stride`=1, the the threads with local ID's $0, 2$, will accumulate SH_MEM[0], SH_MEM[1] into SH_MEM[0] and SH_MEM[2], SH_MEM[3] into SH_MEM[2] respectively. For the next iteration of the dummy variable `stride`=2, only the thread with local ID's 0 will accumulate SH_MEM[0] and SH_MEM[2] into SH_MEM[0]. **But remember:** in the previous iteration, the sum of SH_MEM[0], SH_MEM[1] was stored in SH_MEM[0] **and** SH_MEM[2], SH_MEM[3] into SH_MEM[2]. Thus, during the last iteration, we've performed the reducing operation within the block and accumulated the sum into SH_MEM[0]. The line 25 will simply write the value of SH_MEM[0] to the output, global memory. The kernel is done.

Next iterations operate the same way as the first. The only thing that is changing is the number of blocks, that the GPU will operate with. This does neither change the workflow, nor even the size of shared memory within each block.

It is quite hard ~~even maybe very hard~~ to understand the pipeline of the method execution. We should re-read the text above again and again, and try to associate it with the Figure 10. To recap the reduce process with shared memory:

- Define the initial number of threads and blocks, such that they cover the whole array to be reduced (note that #threads - number of elements on which the reduction will be performed locally).
- At every iteration, launch the kernel with the #blocks and update the new size of the array, which contains the previously reduced elements.
- In the kernel :

    1. Assign global thread id
    2. Copy data to the shared memory, from the global memory.
    3. Perform the reduction in the shared memory locally. With the if condition, make sure that

        (a) The memory access does not overflow
        (b) The elements are not added more than once (only add 2 consecutive elements)

4. Copy the data at 0'th location (the location of the elements accumulated within the block) to the global memory. (in Figure 10, this corresponds to the blue grid below)

- Repeat the kernel, by adjusting the size of the global array, (accessed by kernel at the beginning), controlled by the # of blocks.
- End when the #blocks has reached 1 - when we're left with 1 block, on which we must perform the reduction and store at the 0'th element.

Okay, let's now discuss various performance aspects:

Memory. Clearly the main difference between the two implementations is the usage of memory. The first implementation uses global memory. Every thread goes to the global memory to take data, which, of course, takes time. In the shared memory implementation, the algorithm spends time to initialize the shared memory, which takes time. However, further on, it has lower latency than the global one. In general, the performance of the shared memory implementation is better than the global memory. Nevertheless, it is almost always advised to implement benchmarking into the code and/or use some debugging/benchmarking tools.

Warps. One may analyze the code under the warp's viewpoint. Remember, the threads are scheduled on the SM, partitioned into groups of 32 - warps. Ideally, they all run in parallel and don't have any barriers. Suppose that some threads in the scheduled warp, have some conditions that stops them, so they finish earlier than those, who haven't entered the condition. This means that some threads are idle, plus this requires additional, potential rescheduling. This problem, which causes throughput inefficiency, is called **WARP DIVERGENCE**. Let's now quickly try to detect warp divergence in both codes.

In the shared memory implementation, we've got one potential `if()` condition. Which may cause warp divergence. Indeed, the greater the stride is, the more threads will fail the `if(id_x+ stride<size)` condition, thus being idle & waiting for other threads, who have entered the condition.

The similar issue is in the second code. Indeed, on line 20, we have a condition, `if()`. The warp divergence is pretty big [17], as at every loop, only some threads will *enter* the condition (see the code discussion above), while other will become idle.

To attack these issues, one may use various techniques, potentially discussed in further sections. Some of these methods may be very tricky, sometimes requiring built-in CUDA features (unknown for us at the moment), and sometime very *primitive* techniques (e.g. section 8)

[17] One could potentially deduce the mathematical formulation of warp divergence, and evaluate, where is the divergence more present. But for the moment, we will stick with the qualitative approach.

To be fully honest, there is almost never a way to complpetely get rid of warp divergence. However, it is possible to do small changes to reduce them. In this case, we will follow a strategy, that has changed a bit the access of the elements and gather/reduce them together. Remember, in the shared memory implementation of reduce, we were looking for elements, which are located next to each other. We want to modify the memory access of threads, such that the pairs of elements are not necessarily next to each other. To do that, we are dividing the block in 2 parts, and we're adding (reducing) the 0'th element of the first half and the 0'th element of the second half. We call the dimension of the half of the block - the stride. Thus we get that SH_MEM[0] = SH_MEM[0+stride], SH_MEM[threadId.x] = SH_MEM[threadId.x+stride]. Note that this expression may cause bad memory access, if `threadId.x + stride` is greater than the size of the shared memory. To prevent that, we're adding an additional condition - `if(threadId.x<stride)`. And this process will be done at every iteration of the for loop in the kernel. Here we are doing nothing but a litteral reduction - *Dividing the block in half, adding(or any arbitrary operation) the one-to-one elements. When done, "throw" away the right block and perform the same reduce on the newly created block.* From the warp divergence perspective, one may notice that there is still a condition, that will potentially lead to warp divergence. However, looking at this condition, we can make a statement about **when** will this divergence occur. As the stride vary from `blockDim.x` to 0 being every time divided by 2 (e.g. $64, 32, 16, 8, 4, 2, 1, 0$). The `if` condition will be omitted **if and only if** the warp size is less than the stride. Thus, at iterations, when the stride ¿ $size_{warp}$ no warp divergence will occur, as **all the threads will pass the condition and there won't be idle threads**. As the warp size is 32, one can choose the most optimal block dimensions. Supposedly, the bigger the bloc dimension is, the less iterations will cause warp divergence, the better it is.

Personally speaking, this code/approach is much easier to understand and visualize than the previous ones, and in addition a bit faster. However, I wanted to roughly take some course/book's paths, where the reduction is presented in this specific order.

```
__device__ int reduce_sum(
    cooperative_groups::thread_group gr, \
                        int *temp, int val){
    int lane = g.thread_rank();

    for (int i = g.size() / 2; i > 0; i /= 2){
//map each element in the first "semi" block
//to it's corresponding element in the second one
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if(lane<i) val += temp[lane + i];
        g.sync(); // wait for all threads in to load
    }
    return val; //only thread 0 will return full sum
}
```

Listing 8: This kernel, is ~~almost~~ the same as the first, optimized version of the reduce algorithm, using the shared memory (section 3.1). Therefore, one must note that this reduce_sum() method must be called for the array temp, located in the shared memory.*

4 CUDA synchronization mechanisms

As we've discussed in the section on architecture, when writing kernels, we always need to think about the GPU's hardware, scheduling, etc... By now, with the basic examples we've considered only *basic* operations. Indeed, the only API function to synchronize execution, that we've used in the kernel is the `__syncthreads()`. This is a simple syncing mechanism, that ensures that all the threads **within the block** are done before this function invocation.

4.1 Cooperative Groups

Cooperative groups is a relatively new feature to the CUDA API. As the name suggests it, this feature enables us to group threads, with the ability to perform common, collective operations (or simply collectives). We can also perform synchronization between the threads, belonging to the same cooperative groups. With these API features, one can simplify the code, thus avoiding common mistakes and making it more readable. For example, in the code snippet 8, we perform the exact same algorithm as in the section on reduce algorithm, by using some utility of the CUDA API. In this case, the `cooperative_groups::thread_group` class (do not pay attention to how we created this object and/or how it is declared). The `thread_rank()` function gets the ID/rank of the thread within the thread group `g` (the same way as `threadId.x` within a block). Then we're calling the `sync()` function, which ensures that all the threads within the thread group will be done setting the `val`, and the second to ensure that all threads are done reducing. It is important to understand, that all the threads will return the val. However, only the thread 0 will accumulate all the `val`'s. [10]

```
auto tb = this_thread_block(); // gets the thread block in kernel
tb.sync() // same method as in cooperative_groups
cooperative_groups::synchronize(tb);
this_thread_block().synchronize();
cooperative_groups::synchronize(this_thread_block());
```

Thread blocks

We've already seen the notion of a thread block many times. This notion was always quite *abstract* and implicit. Indeed, while launching the kernel, we've always kept the notion of thread blocks in our mind, but never actually accessed it explicitly. However, in newly introduced features, we can "access" the thread block explicitly. Remember the legacy `__syncthreads()` function. Well, syncing the `this_thread_block()` does the same thing as the `__syncthreads()`. Thus, there are several ways/semantics to synchronize the threads. The following function calls are synonyms.

One can also mention other synonyms `dim3 threadIdx` ≡ `dim3 thread_index()` and `dim3 blockIdx` ≡ `dim3 group_index()`. Thus one can easily replace these built-in keywords with these new methods, without any noticeable performance issues.

Partitioning

For these cooperative groups, a partitioning feature is also available. For instance, if we've created a thread thread block, by invoking `auto tb = this_thread_block()`, one can divide it into more small parts, for instance, into groups of 16 threads. This is done using the `cooperative_groups::partition()`, method, which takes the subject itself (the one to be partitioned into groups) and the number of threads per group. For instance, `cooperative_groups::partition(tb, 16)` divides the thread block into groups of 16 threads (so if e.g. a block has a max of 64 threads, this function will create) 4 groups of 16 threads in each.

The object returned is a `thread_group`. By accessing this object, it is possible to get the thread's rank, **within the obtained thread_group** (for instance, if we divide the thread block into groups of 16 threads and, by passing this object to a device function, print the `thread_rank()`, method, we will see numbers varying from 0 to 15).

The utility of these features overall is to reduce the errors in code. Indeed, the NVidia documentation states that the usage of these features significantly reduces the risk of deadlocks. The concept of deadlocks is probably well-known to the reader. This is a typical situation when we don't want different threads to access a critical section, and ask them to be synchronized before accessing them. Consider the two pieces of code:

We clearly see a deadlock in the first piece of code, as there are only threads with ID's less

```
__device__ int sum(int *x, int n)
{   ...
    __syncthreads();
    ...
}
__global__ void parallel_kernel(float *x, int n)
{
    if (threadIdx.x < blockDim.x / 2){
        sum(x, count);  // Half of the threads enter and
                        //the other half-not
    }
}
```

```
__device__ int sum(thread_block block, int *x, int n)
{   ...
    block.sync();
    ...
}
__global__ void parallel_kernel(float *x, int n)
{
    sum(this_thread_block(), x, count); //OK
}
```

Listing 9: Synchronizing using the legacy mechanism vs using the cooperative groups. [10]

than the half of the block dimension, which will enter the `if` condition. Those threads will perform their piece of code independently and then will wait for all the other threads in the block to be completed and synchronized. This is a big issue, as there are threads, which will never start the `sum()` kernel and will just sit up there, waiting for those, who have. So the two chunks of threads will just sit and wait for each other infinitely.

This is why one may find an application for the previously discussed primitives. In the second piece of code, one call the method `sum()` with a thread block. However, we could have divided it into groups, using the CUDA functionality discussed above, **and only synchronize that block**, which, we are sure, will be run by all threads in the block.

Warp synchronizations

While programing with CUDA, one never gets tired to think about warps, and how to optimize their execution. I do agree that it is not an easy task, to think of it while doing even some basic operations. The new NVidia architectures and new versions of the CUDA API, provides a simple way to navigate through these concepts.

We have discussed a lot of the advantages of shared memory (e.g. for the speed and efficiency of the reduce algorithm). However, the new utilities give us a faster, or even more local way to perform some operations. Remember, shared memory is block-local memory. Remember

also that every thread has some kind of register to store small intermediate values while performing a kernel, for example, when we used to store the local thread ID. The so-called *warp level synchronization primitives* allow us to access a certain thread's local register from an-another thread, as long as they are in the same warp, without the usage of the shared memory. Again, there are many things to keep in mind, but if such a function is called, it is doing everything *atomically*, in the sense that it is a primitive operation, performed locally on the threads in the warp. We therefore introduce here the notion of the **lane** (in fact, we used it briefly above). A lane is the thread id within the warp.

A little disclaimer: There are various Warp-level primitive functions. We will note that many function's names are similar, and only differ by the postfix `_sync()`. For instance, `__shfl_xor()` and `__shfl_xor_sync()` [3], [7]. Indeed, the ones with the `_sync()` postfix are an improvement of the former. It is recommended to use the newer version instead. I will not go into great detail about these differences. I will just mention that there are differences in parameters [18](see further examples).

The `__activemask()` primitive/function is actually not a synchronization mechanism, but more of filtering mechanisms. This function returns the *indices* of the active threads in the warp, where it was referenced from[19]. So the result returned from the `__activemask()` is used to call other synchronization functions, to *give them the corresponding threads, that are active in the warp*. So, for example, one could call the `__syncwarp(MASK)`, thus asking to sync all the threads meeting the `__activemask()` condition. One can go to the NVidia's developer's guide [2] and find the following:

> *Returns a 32-bit integer mask of all currently active threads in the calling warp. The Nth bit is set if the Nth lane in the warp is active when __activemask() is called. Inactive threads are represented by 0 bits in the returned mask.*

So let's have a quick look and example of what did NVidia provide us with [20]

- `__all_sync()`, or in previous CUDA versions, `__all()`, being now deprecated. The signature of this function is `int __all_sync(unsigned mask, int predicate)`, where `mask` is the filtering mechanism. Based on this filter, only threads, which fulfill it will be eligible to get eventually syncrhonized. This passed `mask` can be the result from, for example, the `__activemask()` function. The `predicate` is the condition, that will be evaluated in all threads, that have passed the filter condition. See example section 7.

[18]Frankly speaking, the author hasn't seen this additional parameter being used in a very extensive way. This *extra* parameter is often replaced with some kind of hardcoded value

[19]One can think of the mask as the python's numpy feature, while doing a filter : `arr[:]<1.0`, which will return an array of booleans, which will be used to access later the elements of interest

[20]There are many such primitives available. As usual, the best way to get information about such CUDA functionality, is to look it up in the NVidia's developer's guide [2].

The function *will return non-zero if and only if the predicate evaluates non-zero to* ***all*** *of them* [2].

- `__any_sync()` is a similar version, having the same signature as the `__all_sync()`. But the difference is that *the function will return non-zero if and only if the predicate evaluates non-zero to* ***any*** *of them* [2]. See example section 7.

- A more complex function, which gives more information, about the evaluation of the predicate, it the `__ballot_sync()`. The signature of this function is `unsigned __ballot_sync(unsi` The difference between this function and the previous ones, is that *the function returns an integer, whose N'th bit is set if and only if the predicate evaluates to non-zero for the N'th thread of the warp, and the N'th thread is active* [2]. See example section 7.

- `__shfl_sync()`, or, in previous CUDA versions, `__shfl()` is a tool, to "broadcast", or "spread" a certain value from a certain thread (identified with its lane) to all others in the warp. For example, in a certain warp, all the threads have a variable `int b = //some random int`, unique for all the threads. And I want all these thread's variable `b`, to be the same as the one in the thread 4 (its lane number or ID within the warp). The best way to do that is to use the provided function: `__shfl_sync(0xffffffff, b, 4)` (or `__shfl(b,4)`). The second parameter is the variable to be broadcasted and the third one is the lane number to take the value from (because, of course, all the threads have this local variable `b`, which is different for all of them). So we're replacing all the b's with THE b of the thread 4. The first parameter is actually the mask/filter, that tells the processor (or core I should say) which threads will be involved in this operation. This can be used by passing the result of e.g. `__activemask()` function (there are multiple *filtering* functions) or by passing it the *default* value in hex notation, which corresponds to the maximum number that can be displayed in binary notation (all the 32 bits are 1, thus we're saying that all the threads are active).

- `__shfl_up_sync()` or in previous CUDA versions `__shfl_up()` is a function to shift the values of the warp by an offset, with a lower line ID. For instance, let's say that in the warp, I want the 4'rd thread to have the value from 0'th thread, the 5'th thread the value of the 1'st, the 6'th thread the value from the 2'nd thread, etc ... (from higher id to lower) Then we would want to use the `__shfl_up_sync()` function, with the same parameters as in the `__shfl_sync()` function described above.

- `__shfl_down_sync()` Is the same idea as the `__shfl_up_sync()`. The difference is that we would use it to shift from higher to lower. This is the classical mechanism to perform a reduce. The reduce will performed on a warp level (see Figure 11).

These primitives come in various shapes and forms. It would take quite a time to discuss them all here. The idea for them all, however, follows quite well the patterns, we've discussed just above. It is important to understand that these operations are sort of atomic, because, as we've seen, these are warp-local primitives, which is the most fundamental part of the execution scheduling model.

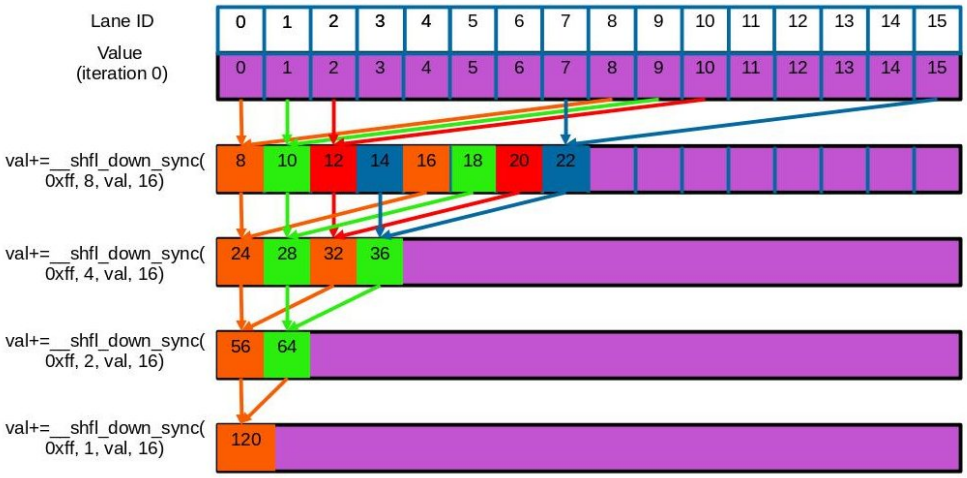


Figure 11: The implemented reduce algorithm using the shuffle-down method. Note that this example is warp local. Thus it is impossible to use this mechanism for bigger arrays. However, this example illustrates well the idea of shuffling within a warp. In addition, it is possible to use this concept for greater arrays, by breaking them down into simpler, warp-level arrays. In this example, for illustration's simplicity's sake, I've supposed that the warp is of size 16, instead of the correct value of 32. The first argument passed in the function is the mask, which is -1 in hex notation, meaning that this addresses all the threads in the warp. The next arguments is the value to be added and the offset to the value, of the second thread. The last argument is the value of the warp size. In this case, it is 16 (by default it is 32). Other synchronization functions can be used to achieve a similar goal. For example the Xor shuffle. The geometric interpretation will however be different. [15], [13]

Atomics

When learning multithreaded programing, one of the first notions, that one must understand is the notion of atomics or atomic operations [21] . To make it short, these operations are operations, which, we are sure, will be performed *in the smallest possible period* and nothing will be able to affect the execution of these operations. For example, in C++, there is a `std::atomic<bool>` template specialization.

In the CUDA API, multiple atomic operations are provided. Arithmetic atomic operations, such as `atomicAdd()`, `atomicSub()`, `atomicMin()`, ... ,bitwise functions, such as `atomicAnd()`, `atomicXor()`, ... (for more: [2]). All these operations have the same (or almost same) signatures: `int <or any primitive type> <function_name>(int* old_adress, int value)` So these functions take the old address as the first parameter (for example, the address of the value, we want to add to) and the value as the second. The function thus performs an arithmetic or logical operation atomically, and stores the result in the old address. It also returns the value.

The atomic operations are very expensive, as the scheduler must perform sequential memory access. It is thus very advisable to use these operations, only when needed (just as in usual `C++` code). For example, let's say we've performed some kind of reduction on multiple blocks locally. We know that if we've used some shared of memory on this reduction, one must in addition add the block results, to get the full answer of our reduction. One way to do that is to force an atomic operation in all blocks, so they store it in the first address of the global memory, which we will later *consume* from the CPU. So if one supposes that we've accumulated the thread-local reduce into `sum` and the pointer to the initial data `g_out`, we would do something like

```
if(this_thread_block().thread_rank()==0){
    atomicAdd(&g_out[0], sum);
}
```

5 Streams

5.1 Concept of streams

CUDA streams is another fundamental concept that we've used previously in an implicit manner. Indeed, let me provide another example using C++: when a "Hello World" program is written in C++, no one cares about the thread usage. We do not create a "main" thead, when printing a character to the standart output. In the context of the CUDA

[21] Intuitively, this comes from the word *atom*. This is a greek word, meaning something like *uncuttable*, *undivisible*. In fact, this is almost exactly, what an atomic operation corresponds to [14] .

API, **any** program is by definition multithreaded. Therefore the analogy between the *usual* C/C++/Python/... and a GPU program does not make sense. The stream concept is more of a thread **over the GPU threads**. By now, all the programs we've considered were launched in one single CUDA stream. So one can launch a program (e.g. a simple vector addition) in one stream, which will be performed in parallel. However one can add a second vector addition program, which will be done in parallel, with the first vector addition.

A CUDA stream is thus a way to make two (or more) kernels run in parallel. (Remember that the kernels themselves are already run in parallel following the CUDA architecture). By now, we've used one implicit stream - the default stream.

5.2 Basic usage of streams

Let's try to explicitly create a stream and look at its initialization syntax:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
foo_kernel<<< grid_size, block_size, 0, stream>>>();
cudaStreamDestroy(stream);
```

Therefore to create the CUDA stream, one must first initialize the stream, then create the stream, i.e. allocate the memory via the `cudaStreamCreate()` method by passing the pointer of the stream. In order to call the kernel and make it run in a particular stream, one passes it as the last parameter in the angle brackets. If nothing is passed in the angle brackets for the stream, the default stream is used.

The simplest usage example of CUDA streams is when one would want to consecutively call a certain kernel multiple times (for example N times). the way to do it using the default stream is to simply loop N times and call the kernel at every iteration. Keep in mind one important property of the default stream - it is by default sequential and can not overlap with other streams. The way to do it using streams is to create an array of streams and call the kernel by assigning it to different streams. The kernels will then be run in parallel without waiting for the previous kernel to return. The execution of the kernels So these functions take the old address as the first parameter (for example, the address of the value, we want to add to) and the value as the second. The function thus performs an arithmetic or logical operation atomically, and stores the result in the old address. It also returns the value.

5.2.1 Asynchronous streams

We've already mentioned the asynchronous memory transfer mechanism, when discussing the memory models (see 2.4.3). Let's get back to the same example, and illustrate in more

```
for (int i = 0; i < nStreams; ++i){
  int offset = i * streamSize; //offset for memory copy
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,\
    cudaMemcpyHostToDevice, stream[i]);
  kernel<<<streamSize/blockSize, blockSize, 0,\
    stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,\
    cudaMemcpyDeviceToHost, stream[i]);
}
```

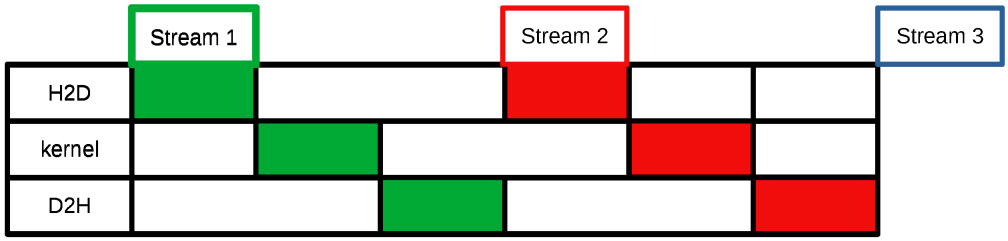


Figure 12: Scenario of the execution pipeline, if no streams and asynchronous memory are involved.

detail on what was happening in the codes.

Without the streams and the asynchronous memory copy, all the operations invoked in the loop would have been sequential. Suppose that one of requirements is not fulfilled. That is, suppose one replaces the `cudaMemcpyAsync()` with something else, and/or remove the stream functionality. Then the iteration of `i=1` will wait for the `i=0` iteration and so on.

One may see that the sequential version does not fully use the potential of the GPU. That is, some of the memory transfer mechanisms are waiting for others to finish. To solve this problem, one uses the asynchronous memory version. In this case, the execution pipeline would look like that:

Note that this Figure 13 pipeline is not always true. That is, based on different architectures, different results can be expected. For example, some of the versions have only a single

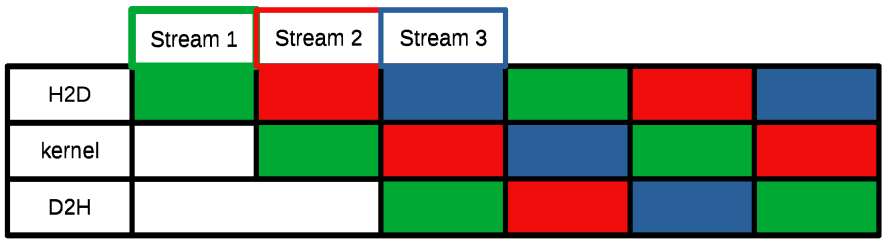


Figure 13: Scenario of the execution pipeline, which is fully asynchronous and fully uses the memory transfer mechanism's resources.

memory copy and kernel engines. This implies that the async functionality does not make sense for these versions (note that these versions are not very common anymore).

Okay, but what if we want to not only parallelize the CUDA API calls for memory copy, but also the kernels? Suppose we're calling two kernels sequentially in our host code. Then they will run sequentially, that is, the second kernel call will wait for the first to return, before getting launched [22] . This case is fully ok, if one knows that the kernel is extremely efficient, such that it takes 100% of our GPU's resources, memory, etc... But what if we somehow know that it is not the case, and we want to parallelize 2 kernels, which each take less than 50% of our GPU's resources? This is once again where the streams will come into play. It is clear that if we launch two such kernels in the same stream, the calls will not be parallel. Thus, we would need two streams, so that each kernel is assigned to a separate one. The subtle thing here is that in order for this to work, one needs to **explicitly** create 2 streams and **explicitly** assign a kernel to each of them. What is meant here, is that the default stream will not do the job. As remember, the default stream is by default sequential. This is a simple small detail, that one needs to take int account, when making our code parallel. Unfortunately, it is not it. If we do as we've just said, that is, to create two **separate** streams and pass two kernels to it (not the default one), the kernels will still run sequentially. In order to make the stream overlap for kernel calls, one must create them with `cudaStreamCreateWithFlags()` function. This function has two parameters - the pointer to the stream (exactly as the `cudaStreamCreate`), and a flag. In our case, as we want them to be able to overlap with each other, we will pass the flag `cudaStreamNonBlocking` [4].

[22]This is not fully true. Indeed, by default, the host is asynchronous with the kernel calls. The host "does not care", whether and when will the kernel return - it's not his responsability. This is the exact reason, why is the very beginning, we were calling a special function `cudaDeviceSynchronize()`, when we knew we were done with the GPU code.

What I meant here, by *return*, is that, in fact, it is the default stream, (that the kernel has been launched on), which is sequential, and will wait for the previous kernel to return.

6 Programming tools

We have seen that there are many subtleties in the GPGPU CUDA programming. Indeed, when I first discovered GPU programming, the concept of scheduling a process on the big number of threads was new to me. It is quite hard to debug these applications. The processes are happening on a very low-level scale. Often, algorithms to be implemented, are complex enough, to have multiple ways to implement them. This is where it gets interesting and hard. We all know that the process of debugging and profiling a program/software is a huge part of the development. Debugging a sequential program is already not easy (especially if it implements some kind of parallelism). You can imagine, how hard CUDA programs are to debug. In order to debug a plain `C/C++` program, we use `gdb`. For CUDA programs, we will use `CUDA-gdb`, which is available in the installed CUDA toolkits. Apart from debugging on a low level, we would also to have an overview of the overall program pipeline. For that, we do profiling. The legacy NVidia's tool for profiling is the `nvprof`, which comes in both cli and ui versions. However, it has been replaced by 2 tools (also coming in cli and UI versions) - NSight Systems and NSight Compute. They are invoked with `nsys` or `nsys-ui` and `ncu` or `ncu-ui` respectively. *Note: The section here will be kind of a "soup" of different additional tools, that are interesting and, in a way, necessary to know in order to successfully start the CUDA programming journey.*

6.1 Handling errors

Before we start to look into different tools that a CUDA programmer should be familiar with, one first must learn how to handle errors, when CUDA API calls are invoked. Suppose you're allocating memory on the host, using, let's say `cudaMalloc()`. What if some kind of error happens during this process. It is not guaranteed that the program will throw an error. The function `cudaMalloc()` returns a `cudaError_t`, which is a part of a CUDA errors enum.

Another way to access the error and/or the error code is the `cudaGetLastError()`. This function returns the last error produced by any of the CUDA runtime calls in the same host thread, and resets it to `cudaSuccess`. A string can also be retrieved from the CUDA error code. This can be done with `cudaGetErrorString(cudasError_t err)`. Let's have a look at a very simple example at 10.

6.2 Events

Another thing we would want to do, related to debugging and profiling is to trace different events/milestones that our code will go through. What I mean by tracing is to *see/follow*, what our code is doing, step by step. It is very useful, because, as we've already mentioned, the parallel GPU applications are quite hard to debug, in the way we're doing it with plain

```
cudaError_t status;
status = cudaMalloc((void**)d_ptr, size*sizeof(float));

if(status != cudaSuccess){
  char* err_str[1024];
  err_str = cudaGetLastError(status);
  std::cerr<<"Error occured"<<std::endl;
  std::cerr<<err_str<<std::endl;
  return (int)status;
}

```

Listing 10: Retrieving error codes from the CUDA runtime API.

C/C++ code. This gives us an additional tool to debug, trace and benchmark CUDA code.

```
  cudaEvent_t start_evnt;
  cudaEvent_t stop_evnt;

  //memory allocation on host, device
  //memory copy initialization, etc...

  //record the start event
  cudaEventRecord(start_evnt, stream);
  kernel_to_benchmark<<<NG, NT, 0, stream>>>();

  //record the stop event
  cudaEventRecord(stop_evnt, stream);
  //wait for the event to be synced (done)
  cudaEventSynchronize(stop_evnt);

  float miliseconds = 0;
  cudaEventEllapsedTime(&miliseconds, start, stop);
  std::cout<<"Time elapsed given by "<<miliseconds\
    std::endl;

```

Listing 11: The code is quite simple to understand. On line 2 *and* 3*, we're creating the 2 CUDA events, which will keep track of flags, that we will set. On line* 7*, we are enqueuing the start event into the stream, which will stick to the provided stream. Then we're calling the kernel with the specified stream, and once it is returned (we know that the function/kernel calls on the host are sequential), we are recording the stop event, and on lines* 15 *and* 16*, we're measuring the elapsed time between two event registration. There is however a call on line* 13*, that may look unfamiliar. We will discuss it shortly. [5]*

The mean, through which, we're implementing these concepts are already familiar **streams** and **events** (CUDA streams and CUDA events). We've seen that a CUDA program can be parallel at the kernel execution level (we've discussed threads, blocks, grouping, etc...), as well as on the *device* level. What I mean here, is that one can launch CUDA functions, such as the `cudaMemcpyAsync()`, in parallel. These asynchronous functions will be sched-

uled automatically by the CUDA scheduling mechanism, and will perform asynchronously. The CUDA event is a CUDA type `cudaEvent_t`, which helps to control the CUDA stream execution. The events can be considered as some kind of flags/points living in the stream execution timeline. The way the events can be used is for example benchmark the execution time of the kernel, by creating 2 flags in a certain stream. The first flag will be *registered* as soon as it was created, and the second flag will be *registered*, once the kernel that we want to benchmark has been completed and returned. Then we have two events that have been registered at different moments of time in the specific stream, and we can take the difference between them [6]. Let's have a look at a simple example of using CUDA events as a benchmarking technique.

The events are not only used for performance metrics, as in the code snippet above [5], but also for synchronizations between streams. Indeed, using the `cudaEventRecord()`, one can enqueue the event into a stream, and further refer to it. We manipulate the event/stream manipulations using three methods [4]:

- `cudaEventQuery(event)` returns `CUDA_SUCCESS`, if the event passed as the parameter has occurred (this could be interesting for some kind of conditional waiting).
- `cudaEventSynchronize(event)` will block the **host**, until all CUDA calls are completed.
- `cudaStreamWaitEvent(stream, event)` will block the stream until the event will occur. Note that this will not block the host, and this function will block only those calls, launched after this function call.

6.3 Other tools

6.3.1 PTX

Programming languages have different levels of abstraction and different levels of closeness to the hardware. I think we can all agree that on of the approximate rankings in terms of *low-leveliness* would be `Python` → `Java` → `C/C++` → `Assembly` → `Machine/Hardware code`. Don't judge this classification too strictly, as this is a more intuitive rank, mostly based on my opinion. When writing `C/C++` code, it gets translated to Assembly, then to machine code, which gets interpreted by the processor. You may imagine that the GPU understands a slight variation of this Assembly language.

Indeed, the equivalent Assembly for the CUDA GPU is the PTX [23]. Despite being very proprietary, it may however be very useful to sometimes inspect the PTX code, even for a better understanding of how the in-thread code works. All the extensive information on

[23]PTX stands for Parallel Thread Execution, and is not really a language. It is more of a proprietary instruction set (see sources [8], [12])

PTX is available in the CUDA developers guide [8]. However, I must mention, how to obtain a PTX code from a `.cu` file(s). To do that, we invoke the CUDA's nvcc:

```
$ nvcc --ptx <filename>.cu
```

6.3.2 Profilers

[24] A very important part of CUDA programming is the use of ~~not-so-~~external tools to profile CUDA code. NVidia provides developers with tools to profile execution of a program. The legacy tool for profiling is the NVidia profiling tool - nvprof. This tool is still often described in the literature, as might often be helpful. Indeed, the author has always used it as a profiler, when writing code. Nvprof is both a command line and a GUI tool. In order to make a program to be *debuggable*, one must pass flags, when compiling it. The `-G`, when compiling and creating the CUDA executable, lets us fully use the CUDA NSight Compute tool to debug and profile the program. [25] *Note: The NVidia profiling tools is an extremely important and rich tool to debug CUDA programs. I could have spent at least half of the lenght of the notes describing its features and how to profile them. Instead, as for almost any programming guide/book, it targets one specific aspect or goal. The same goes for this one - to provide main possibilities regarding CUDA programming.*

[24] The profiling aspect is extremely important in development. The process can be very complex. However, I will unfortunately not discuss the extremely rich features of the different tools of it, as I have not used them extensively (only the most basic features). Once we know how to launch them, it is possible to discover the features of these tools empirically. Also, as with several notions discussed in these notes, there is no extensive documentation. However, with these tools, it is possible to read the `man` documentation, which may be very helpful, together with Googling.

[25] As mentioned many times, The goal is to not give a precise detailed course on CUDA tools, rather provide a path with different possibilities, which can be looked further into through sources [1].

7 Appendix

Improving with primitive operations

The modulo operation `%` is expensive for an arithmetic operation. One could replace it by something much easier for the GPU architecture to execute. In this case, with a bit of knowledge of binary number representation, we can verify that `if( (id_x%(stride*2) ) == 0)` is equivalent to `if( (id_x&(stride*2 -1) ) == 0)`. It would impossible for me to come up with this small optimization on my own. Maybe for CS majors, it is evident.

Warp level filtering and syncronization

Let's have a look at three functions for warp-local syncronizations - `__all_sync()`, `__any_sync()` and `__ballot_sync()`. The aim of the piece of code is to show the working principle of these functions, with made-up examples.

```
#include "stdio.h"
#include <cuda.h>

__global__ void all_any_example(){
  int id = threadIdx.x;

  //0 if odd, 1 if even
  int id_is_even = !(id%2);

  //get the mask of active threads
  unsigned int mask = __activemask();

  //get the result of functins of interest
  int any = __any_sync(mask, id_is_even);
  int all = __all_sync(mask, id_is_even);
  __syncthreads();

  //the result is the same for all threads
  //thus only print the result from 1 thread
  if(id == 0){
    printf("Result of __any_sync: %d\n", any);
    printf("Result of __all_sync: %d\n", all);
  }
}

__device__ void by_bit_output(int result){
  int size_in_bits = 32;

  printf("\n");
  for(int i = 0; i<size_in_bits; i++){
    printf("%d\t", (result & ( 1 << i )) >> i);
  }
  printf("\n");

```

```
}

__global__ void ballot_example(){
  int id = threadIdx.x;

  //the condition to be true is the
  //lane index to be equal to 12
  int target_thread_id = 12;

  //the predicate computed
  int predicate = (target_thread_id == id);

  //get the mask of active threads
  unsigned int mask = __activemask();
  int result_int = __ballot_sync(mask, predicate);

  __syncthreads();

  //print the result once
  if(id == 0){
    by_bit_output(result_int);
  }
}

int main(){
  int n_threads = 32;
  int n_blocks = 1;
  all_any_example<<<n_blocks, n_threads>>>();
  printf("\n\n");
  ballot_example<<<n_blocks, n_threads>>>();

  cudaDeviceSynchronize();

  return 0;
}
```

The output of this program on my computer is given by:

```
Result of __any_sync: 1
Result of __all_sync: 0

0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The code is quite straightforward to understand, but let's break it down. The program contains 2 independent functions - one to demonstate the `__all_sync()` and `__any_sync()` functions, and one the `__ballot_sync()`. Let's start with the first one.

The functions runs on 32 threads, that is, one warp. This is not surprising, as we're dealing with warp-local primitives, which we want to illustrate. The example consists of applying the 2 functions of interest to determine, whether there are functions having an even ID. Which is, of course true, as indices in a single lane go from 0 to 31. So first we're retrieving the index of the thread and constructing the variable, `id_is_even`, which will be `true` if the id

is even, and `false` otherwise. It is clear, that there will be 16 threads, in which it evaluates to `true`, and 16, where it evaluates to `false`. Next, we're asking the CUDA runtime, to give us the mask with active threads. In our case, all of the launched threads will be active. Thus the `mask` evaluates to `-1` or `0xffffffff` in hex. Finally, were using the two functions of interest. From each thread, the function will "look into other's threads" and evaluate the predicate. From that it is clear, that the return values from the specific function will be the same in all threads. After applying the functions, we're synchronizing threads, which is, maybe, unnecessary. Finally we're printing the results of the 2 evaluations (we're doing it only once, as the results are the same in all threads within the warp). The result is not surprising. Indeed, the function `__all_sync()` will return false, as **not all thread's id is even** (there are also odd id's). The function `__any_sync()`, on the other hand, returns true, as there is at least one function, that fulfills the condition of having an even index.

The second function is similar, yet more *more precise*. That is, we can retrieve the precise index of the thread within the lane, where the predicate has been evaluated to `true`. In this case, the predicate will be true if the thread's lane index is equal to the arbitrary chosen value 12. There will be only one thread, which fulfills this condition, namely - the one with the lane id 12. Thus, similarly to the previous example, we retrieve the id and compute the predicate. Then retrieve the mask for active threads, and pass it, together with the predicate, to the `__ballot_sync()`. Then printing the result (we're doing it only once, as the result is the same for all the threads within the warp). The function that prints the results simply accesses integer's consecutive bits. That is, in order to obtain integer's n i'th bit, one writes `(n & (1 << i) ) >> 1`. The result is expected: all bits are 0, except the 12'th in the sequence.

References

[1] URL: https://indico.cern.ch/event/962112/contributions/4110591/attachments/2159863/3643851/CERN_Nsight_Compute.pdf.

[2] Cuda c++ programming guide. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

[3] Cuda __shfl_xor __shfl __shfl_up() __shfl_down(). URL: https://blog.csdn.net/jqw11/article/details/103071556.

[4] Cuda streams: Best practices and common pitfalls. URL: https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[5] How to implement performance metrics in cuda c/c++ — nvidia ... URL: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc.

[6] Introduction to stream and event. URL: https://www.programmerall.com/article/4005578780/.

[7] Kepler's shuffle (shfl): Tips and tricks — gtc 2013. URL: http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf.

[8] Parallel thread execution isa version 7.7. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[9] Paraljeljnoje programmirovanije s cuda. chastj 1: Vvedenije, Apr 2015. URL: https://habr.com/ru/company/epam_systems/blog/245503/.

[10] Cooperative groups: Flexible cuda thread programming, Aug 2020. URL: https://developer.nvidia.com/blog/cooperative-groups/.

[11] Fang's notebook. nichijou.co, May 2022. URL: https://nichijou.co/cudaRandom-memAlign/.

[12] Parallel thread execution, May 2022. URL: https://en.wikipedia.org/wiki/Parallel_Thread_Execution.

[13] Was Armour. Warp shuffles, reduction and scan operations. URL: https://people.maths.ox.ac.uk/gilesm/cuda/2019/lecture_04.pdf.

[14] Brijendar BakchodiaBrijendar Bakchodia1 and AmadanAmadan. What are atomic operations for newbies?, Apr 2018. URL: https://stackoverflow.com/questions/52196678/what-are-atomic-operations-for-newbies.

[15] Mike Giles. Warp shuffles, reduction and scan operations. URL: https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec4.pdf.

[16] Mark Harris. How to overlap data transfers in cuda c/c++, Dec 2013. URL: https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/.

[17] Raj Prasanna Ponnuraj. Cuda-memory model, Oct 2020. URL: https://medium.com/analytics-vidhya/cuda-memory-model-823f02cef0bf#:~:text=Pageable%20memory,-The%20memory%20allocated&text=The%20data%20at%20this%20memory,allocation%20and%20transfer%20is%20slow.

[18] Brian Tuomanen. *Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA*. Packt Publishing Ltd, 2018.