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. A big part of code snippets are also taken from different resources, and has not always been tested. 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.

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.

The goal of these notes is to **give us a good basic understanding of the GPU architecture, and the most important,, to try to fully depict the most common examples of CUDA code.**

Dictionary

- GPU - Graphics Processing Unit
- CUDA - Compute Unified Device Architecture. The language we use to *talk to the GPU*.
- Device - the GPU, from the software viewpoint. You may think of the notion of the device as an external kind of executor, in our case, the GPU.
- Host - the CPU, from the software viewpoint. The *machine*, that will launch the GPU from a usual `C/C++` (or any other language) program, which, by default, will execute on the CPU.
- Kernel - nothing more than a function, that will run on the device(GPU).

Small introduction

If one wants to perform computations on the GPU, one must have a way to adress it. There are various API's developed. The biggest ones are the Khronos Group's OpenCL, Microsoft's Direct Compute, and the one discussed here, the Nvidia's Compute Unified Device Architecture, or shortly - CUDA.

When discussing the necessity of the GPU for computations, many come up with the example of the car and the bus. Suppose you need to transport people from a 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 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 the 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. Like in our example with the bus and the car.

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) programming, is that one must take into account the architecture of the GPU, while writing even 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 is no any 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 improvments.

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

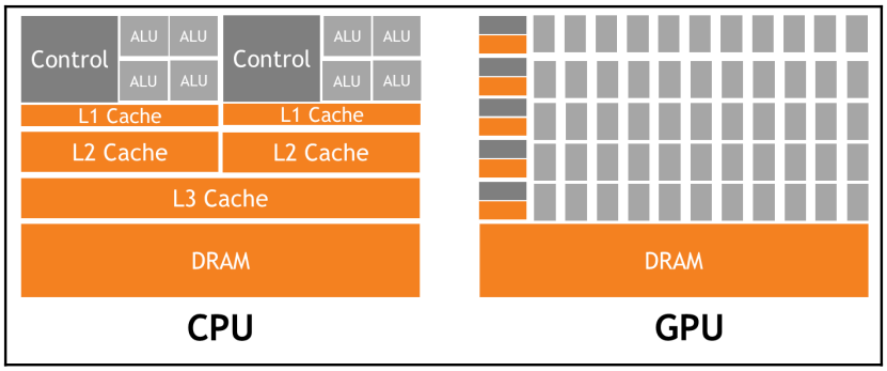


Figure 1: Schematic difference in architecture between the CPU and the GPU. Without going into details (~~as mentionned in the disclaimer, the author have not studied it in depth~~), one may be able to see that the GPU has many smaller ALU's. They are less powerful than those of the CPU and don't stand a chance in a theoretical 1v1 battle, but may do enough damage, when working together. [?]

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 Single Instruction Multiple Threads or SIMT (remember the bus and car analogy).

From the hardware viewpoint, we are distinguishing the **Device (GPU)** itself, the **Streaming Multiprocessors (SM's)** , and the **CUDA cores**. These are physical entities, having a certains structure and caracteristics. 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 mentionned part will be assigned a certain role, and will treat the software abstractions accordingly.

For the programmer, we will consider the abstraction that maps the hardware side to the software side of the program.

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

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**.

$$\text{Grid} \xrightarrow{\text{contains}} \text{Blocks} \xrightarrow{\text{constains}} \text{Threads}$$

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

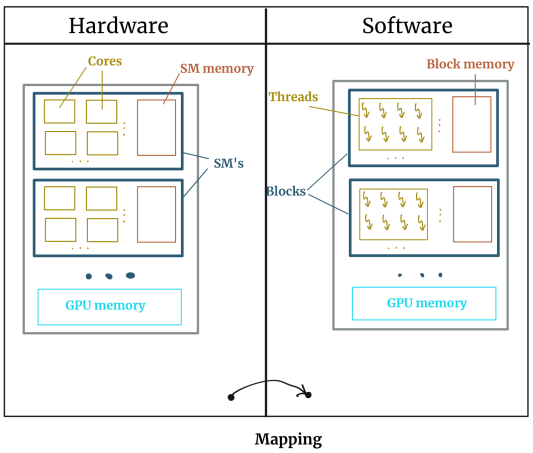


Figure 2: Hardware-software abstraction

The abstraction between the hardware and the software side of CUDA is shown Figure 2. Once the function is given, 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 SM's 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.*

However, there is a notion, which more or less guarantees the execution of threads. The Streaming Multiprocessor treats threads in groups of 32, which are called warps. Think of it as a way to handle rather than a way of grouping (as the blocks of threads) [1].

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 notions concerning the memory model.

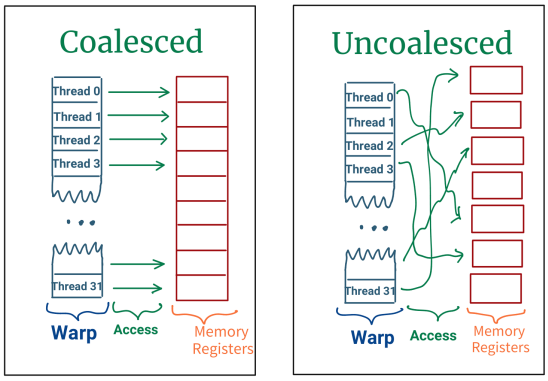


Figure 3: Memory access optimization mechanism.

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

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

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

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

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 standart above again by looking at the coalesced scheme (Figure 3) to fully understand the mechanism. One may notice that this notion is 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.

Global memory is the largest memory in terms of the size, and yet with the greatest latency. As we've discussed, the kernels are launched **from** the 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` programm and retrieve back in a processed form. This is exactly the purpose of the 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 is to copy the data from the global memory to some other (which is discussed below), which is faster [3] to manipulate.

Shared memory is much faster than the global one, but evidently smaller. Other crucial difference is that shared memory is **only seen by threads in the same block**. This provides the ability for threads to share results and temporary calculations, and process the data IN PLACE. Think of this situation as a client of a grocery store. The customers are sort of threads. Every time a person wants to cook something at their house, they don't drive to the store to buy every ingredient. They rather go there once a week, for example, and buy the amount they need and such that all 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. One sometimes refer 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 a better performance. Indeed, the biggest pitfall for all ~~of us~~ beginners are the *bank conflicts*[4].

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

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

[3] 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 = {};`

[4] A small disclaimer: the notion of *bank conflicts* was the reason for this personal document. 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.~~

This is a very important property, so lest's consider an another ~~illustrative~~ analogy. Suppose in a restaurant, each waiter is assigned a strict number of tables, such that a waiter A cannot take B's tables. Suppose you are a host at this restaurant, and you have a large group of the exact number as the resaurant's seats, that are willing to dine. The wise choice would be to partition them between all the waiters at their tables, right? Wouldn't that be silly to partition all the guests, excepting one, who will be waiting for a free place at A's waiter's table, while there are free places at B's table? The analogy ~~may~~ not be the best, but the sketch should do the trick:

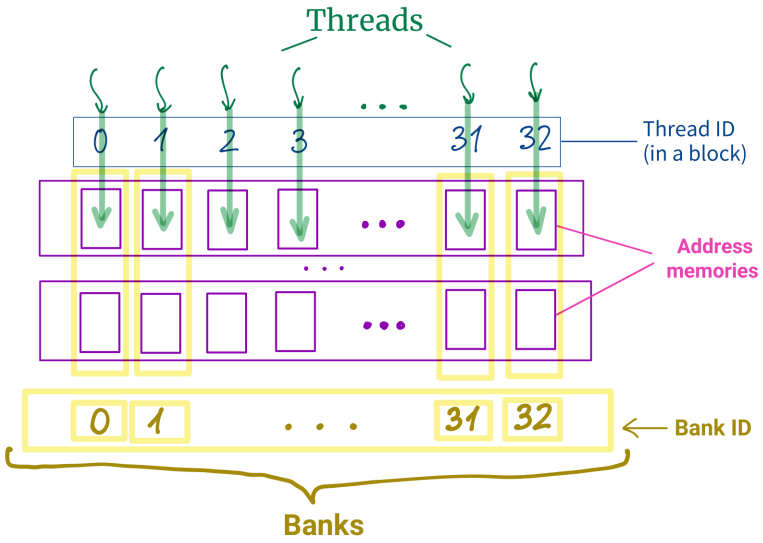


Figure 4: Memory banks serving the threads. Only one thread can access a bank with a certain ID simultaneously. From the analogy, threads are clients, and banks are waiters, giving them memory.

Read-only memory$*$ is, as its name suggests, can't be changed by the kernel's threads. This memory is not as common in GPGPU programming. It is often encapsulated in GPU API's, such as OpenGL. Thus by using those API's, the programming is indirectly operating with the read-only memory. The texture memory is also known as the texture memory.

Local registers . By looking at the scheme of the architecture of the GPU vs the CPU Figure 1, one may notice the amount of registers in the GPU vs the CPU. Indeed, it is normal due to the number of threads in the GPU. As you might have guessed, a register is 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.

1.4 Memor allocation model

By now, we've made the difference between kinds of memory in terms of the scope, i.e, who is able to access/read certain memory. This of course impacts the way we're allocating it. In terms of the memory **allocation**, there are mainly fours ways of allocation, which reffers to the global device memory. We will see further that a usual code, which uses GPU resources, follows a certain path. One of the steps is the allocation of memory on the GPU from the host (by calling a certain CUDA function).

- Pageable memory
- Pinned memory
- Mapped memory
- Unified memory

I will mention the difference in these further.

2 Programming in CUDA

By now, we have talked about the architecture of the GPU [5] 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. We will mainly refer to all these concepts in order to get a more detailed understanding. One will maybe need to refer to the section above to link the theory with a bit of practice. I will do my best to choose the most illustrative examples to explain specific concepts, as well as introducing with some tools. Also note that I use Linux (**??**).

2.1 Setup

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

When we run a `C` program, we need a compiler - the most common one ~~and the best one~~ is `gcc`. For `C++`, we invoke his improved `g++`. For CUDA programs, however, we need `nvcc`. As we've discussed, the program consists of host and device code. The device host is just

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

plain C/C++ code, so nvcc is also able to compile plain code. Note that CUDA codes have a .cu extension.

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

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

If this terminates with success, the program is successfully launched and terminated.

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  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 aspect 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. 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, that finishes all the 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 and thread id's. The variables `threadIdx` and `blockIdx` are of type `dim3`, which is a simple structure, containing 3 unsigned int's. By aceessing its `.x` member variable, we are reffering to the 1D indexing. In general, threads are indexed using `dim3` type. Thus, for a thread, there exists x, y and z dimensions. 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 this information online or asking our computer.[6].

2.3 Memory

2.3.1 Vector addition

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. 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 parallel 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. But we know that the number of threads in one block is limited. Let's check on Nvidia's official CUDA documentation [**?**] concerning threads indexing.

[6]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`.

> *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 $(x + y \cdot Dx + z \cdot Dx \cdot Dy)$.*

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

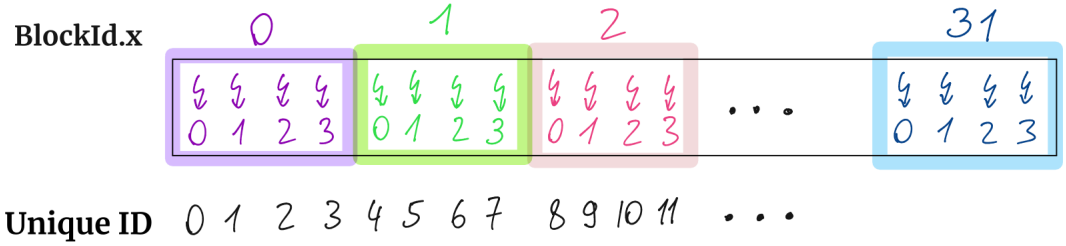


Figure 5: Simple example to illustrate a simple 1D indexing (supposing a block contains 4 threads). We see how the expression blockIdx.x*blockDim.x+threadIdx.x *is used (see below). One can easily extrapolate the indexing to 2D and 3D cases, as described in the Nvidia documentation above.*

Let's now write code to add a vector, using a 2D indexing:

Ok, there are multiple aspects to discuss...

First let's discuss the code. First on the device. Remember we discussed about the global GPU memory. The line 31 we're working with memory. The classical pipeline of any CUDA program execution, is iusually 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 host, we proceed with the usual C `malloc()` function and naturally casting the returned pointer to `double*` (don't forget to

```
#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. [**?**]*

free memory by `free()` function). Then we call a simple function `init_host_vector()` that will just initialize the data to some values.

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

Data copying is done using the `cudaMemcpy()` function. Pay also attention to the parameters - `cudaMemcpy(void* destination, void* source, size_t size_in_bytes, enum cudaMemcpyKind kind)`, with `kind` [7]specifies how to copy. In code, it is understandable that we the destination is device and the source is 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 execute the `add_vec()` function. In the kernel, each an ID is assigned to the thread (line 10) (see the indexing *policy* **??**). We write a simple `if()` statement, to be sure, that the thread does not go out of bounds. Thus every thread does exactly one calculation [8] Remember that the memory, in which these processes are happening, is the global memory.

Terminating. After the kernel, we call the usual synchronization function. Then, a simple check function on host, simply to ensure, that the parallel vector addition was successful. And finally, the `cudaFree()` method, which does exactly what it is expected - frees the *dynamically* allocated memory on the GPU.

Let's also link this piece of code to the section about warps. Remember, threads are scheduled by the SM by 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 multiple of 32. Indeed, imagine, if we were to launch 513 threads on a block. Then the SM would schedule 17 warps. This number 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 a 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 $32\text{threads} \cdot 1024\text{blocks}$

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 do without this feature. Indeed, if we want to work with, let's say a 2D matrix, we could

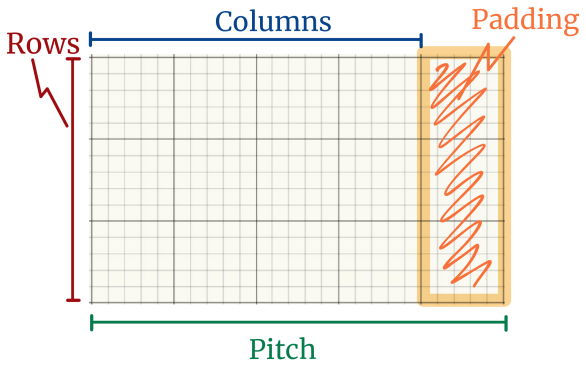
[7]Here, I described the parameters in details. However, further on, the author will not be that precise. The reference for this function was taken from `http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html`

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

simply transform with a 1D vector of size rows·cols. However, there is a feature in the CUDA API, to allocate 2D, and even 3D memory [**?**]. 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 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 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:

```
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 each element, we access it using LINE by casting to CHAR*, *which gives us the very first element of all the pitch. We see that this 2D pitch is allocated* ***automatically*** *by the cuda memory management system. [**?**]*

As you might have guessed, this memory access is not extremely efficient as it is here, as here, really, all threads & blocks that the kernel will be launched with, will **all** access all the

elements of the pitch memory. As one may say, THIS IS FOR EDUCATIONAL PURPOSES ONLY. In the documentation, it 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.

2.4 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 great latency, comparing 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.

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. [?]

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 arr. So the formula is given by $\mathrm{A}_{i^{th}} = A_0 + \mathrm{SZ} \cdot i$, where A_0 - the first (0'th) memory address and SZ-the size of one memory field (e.g. $\mathrm{SZ}_{\mathrm{char}} = 1$). Suppose now, that we've allocated a 2D memory array of size $M\mathrm{rows} \times N\mathrm{cols}$. For the first row one can apply our previous formula - $A_{0,j} = A_0 + \mathrm{SZ} \cdot j$. However, if j is greater than N - the number of columns, a problem occurs. If we work in 2D, we would just *reset* our index j to 0 and increment i (if $j = N - 1$). However, in 2D case, in order to access $A_{i,j}$ address-wise, we use the expression $A_{i,j} = A_0 + \mathrm{SZ} \cdot N \cdot i + j$). 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 one can attack the code. From line 13 to 21, we are initiating the data and allocating memory (don't forget to initialize the data in your code). The lines 27 & 28 are initializing the *pitch*, that the kernel will be launched on. 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. Each thread must iterate 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 putting together 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[9].

Shared memory is one of the keys, to control the efficiency of a CUDA programming. As we've mentionned several times before, the shared memory is only visible in block's scope. That means that if, *supposing*, we would like to make the shared memory visible & accessible to all threads, we would need to allocate (and transfer to) shared memory for each block. Remember the analogy of the grocery store. It's to the programmer to decide, whether it

[9]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. The author always draws a sketch to visualize the indexing process on a piece of paper

would be reasonable to spend time on shared memory allocation (which, of course, takes time, but has little latency), or to the memory access. So let's have a look at a shared memory code snippet[10]. demonstrating shared memory usage for matrix multiplication. Let's first discuss the general strategy. As we have discussed, the shared memory is much smaller in terms of size than the global ones. 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.

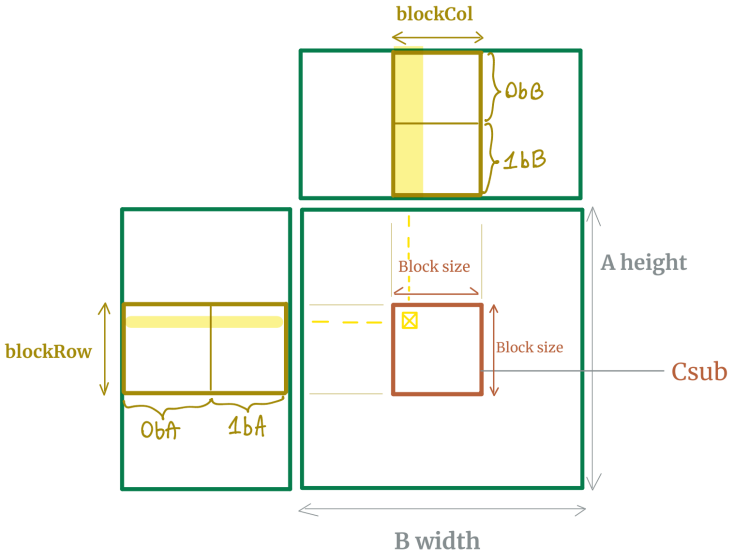


Figure 6: Scheme of 2D matrix multiplication, using Shared memory

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
                                        + BLOCK_SIZE * col];
    return sub;
}

```

[10]For space-saving's sake, only a snipped is provided. The skipped pieces are usual pattern steps.

```
 __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 dimensioin 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 2 blocks, each containing threads positioned in a 2D manner. Keep that launched chunk of threads in mind.

2. The first thing we do in the kernel is attributing the block row and colums id's. 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(think about what could the stride mean). This sub-matrix C_{sub} is illustrated in the figure 6.

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

4. The initialization is followed by the initialization of sub-id's. These are the thread id's **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 for one sub-matrix of C - C_{sub}. These id's are represented by yellow lines and cols on the 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[11]. This is the reason why we can use the first loop, to iterate over A'th width and B'th height *simultaneously*.

6. As mentionned 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 B_{sub} using the `__shared__` directive. Note here, that each thread in 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 sinchronization. Indeed, with global memory's case, each thread accesses the global memory independently, with a unique ID. Here, with the shared memory,

[11] 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 (as we say in fancy high school books -you can check it yourself as an exercice ~~(the author is, maybe, too lazy to do that)~~.) However, this is neither a problem, because, one can allocate and perform operations on a bit bigger matrices, without big issues in performance

we introduce a kind of independence between threads, by making them all access to the same block-local shared memory, by making them all retrieving 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.

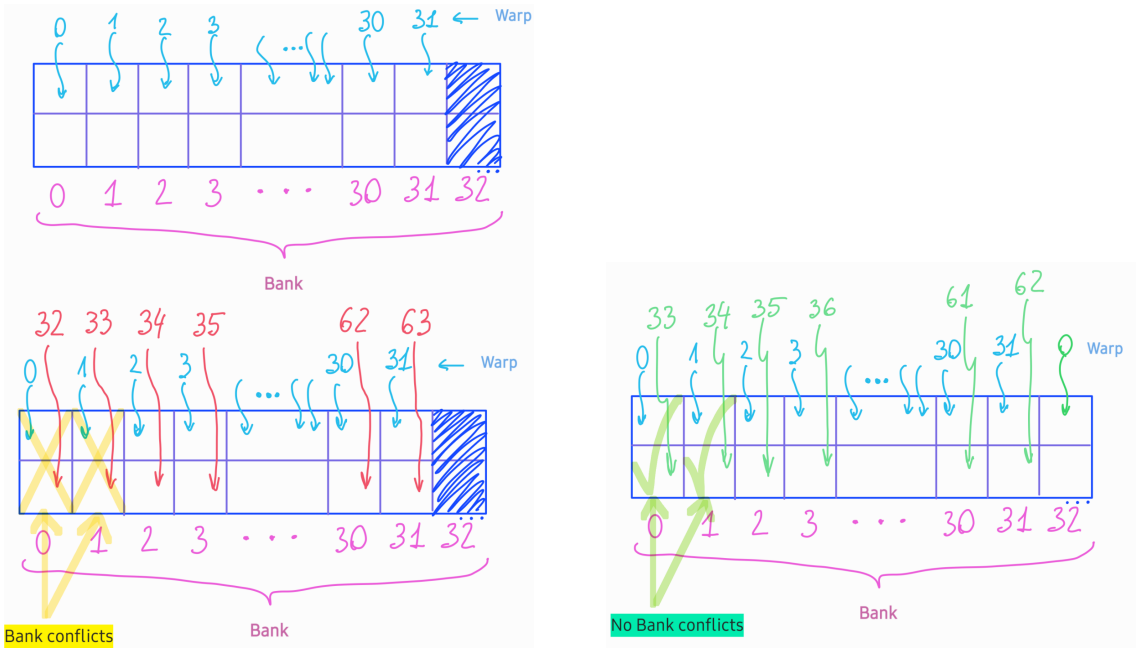
Note that apart of the `__shared__` directive, other, new technical aspects were used. First - the `__device__` directive. It declares a function, which can **only** be called from the device function (the one declared with `__global__`). These device functions can return different types. Second - the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

As mentionned, this is a very fundamental example, yet no 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 thread id's.

Bank conflicts. We have shown several applications of the theoretical notion 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 splitted into two execution periods. The classical bank conflict is often illustrated by the 2D array initialization in shared memory.

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 then 2 and 4 warps respectively will access the memory sequentially, i.e. 2 and 4 periods respectively.

So, in order to avoid these bank conficts 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*



(a) *(b)*

Figure 7: Bank conflicts

more memory than required. Take a look at the **??**: 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 conflit in a 2D memory access, whose size are 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 for memory, will access different banks(columns).

The author does fully understand and does know that this is a quite complicated 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. In most cases, the *solution* is to simply benkchmark the execution of the program.

2.5 More remarks on memory

We have discussed multiple kinds of memory - **Global**, **Shares**, **thread-local** memories. All of them have different scopes, bandwidths and properties(such as bank access). Until now, we have discussed, that shared memory is invoked with `__shared__` compiler directive. The local memory is thread-local memory, when, for example, we initiate simple memory on

stack in a **kernel**, e.g. `int a = 64` (this `int` a is allocated to each thread's register locally).

The global memory is invoked with `cudaMalloc()` function, called from host. Think of global memory as a *long* transfer between the host and the device. The CUDA API provides, in some ways, to make this process faster. Note that the following classification is concerning the global memory. *channelling*[12].

$$\text{HOST} \overset{\text{owns}}{\Longrightarrow} \text{Pageable\& Pinned} \xrightarrow{ALLOCATES} \text{DEVICE}$$

- **Pageable data transfer** When we invoke the *classical* `cudaMalloc()`, we're acessing pageable memory. Under the hood, it is allocated in two times - first to pinned memory(see below), then, from the pinned one, to the device global memory. One can think of it as the longest path between the host data, and the required device data.
- **Pinned data transfer** lets us avoid 2-way memory allocation between the host and the device. Pinned memory is faster, yet less safe than the pageable. We are thus skipping one step in the memory allocation pipeline 2.5. Indeed, the `cudaMalloc()` function is the most primitive and sure memory allocation. The pinned memory allocation is done with `cudaMallocHost()` function, which has the exact same signature and purpose as the `cudaMalloc()` function.
- **Mapped memory**, also called the *zero access memory*[REEF], is pinned memory, which is **directly** initialized/mapped to device address. You may think of it as dynamic memory in `C/C++`. It may ~~(or may not)~~ be faster than the previously mentionned memory policies. It is however more vulnerable. The mapped memory is directly called from the host with `cudaHostAlloc()`, and one get the pointer, used by the device, using `cudaHostGetDevicePointer()`.
- **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 increased. This unified memory is invoked with `cudaMallocManaged()` function [13]

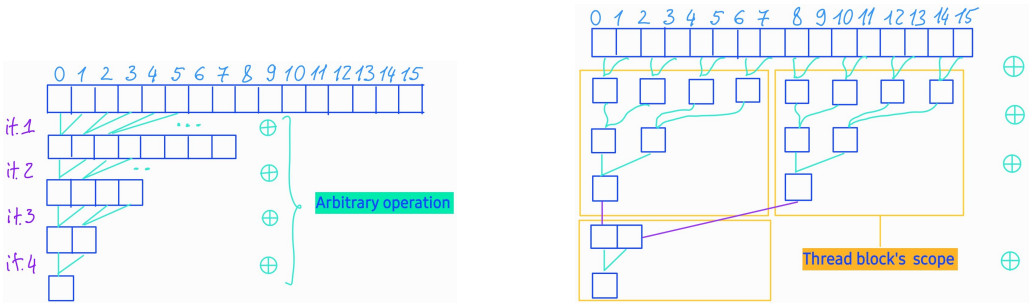
3 Basic parallel algorithms & patterns

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

[12]The way the host and the device are communicating with each other, to allocate memory

[13][REEEEFF]. As the GPGPU programming implies some kind of communication between the host and the device, we try to optimize the time spent on kernel execution by the device and memory allocation by the host.



(a) Fixed rod length and varying mass *(b) Fixed mass and varying rod length*

Figure 8: Reduce algorithm with differnet strategies.

```
__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 5: Global memory reduction. [?]

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. At first iteration, what we would do is to add the 0th element with the 1st, the 2nd with the 3rd, the 4th with the 5th, etc... Notice that all of these additions ($^N/_2$ additions) can be done in parallel. 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. 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 mamory with `cudaMalloc()`) The host and device functions are given by

Okay, let's discuss the implementation. 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, `out[0] = [0]+[1]`, `out[1] = [1]+[2]`, ... `out[30] = [30] + [31]`. This is exactly the first iteration, showed 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.

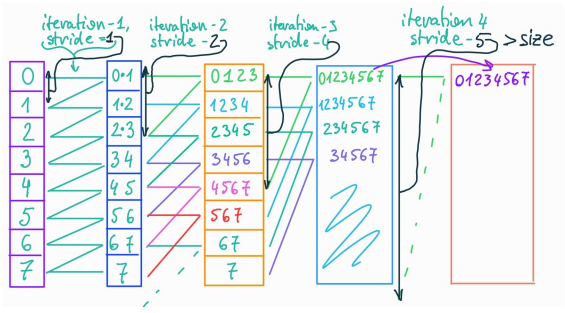


Figure 9: The description of every iterationion, for the global memory reuction kernel. Note the how stride is doubling every iteration, ,and how the elements are accumulating in the very first (0th) element of the output data. One should understand, that the code works for arbitrary number of blocks,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

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

Shared memory reduce

As we've discussed several times above, the 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 function, which will be calling the kernel. Line 3 copies 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

```
   /*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 6: Optimized reduce, with shared memory. [?]

`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.

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).

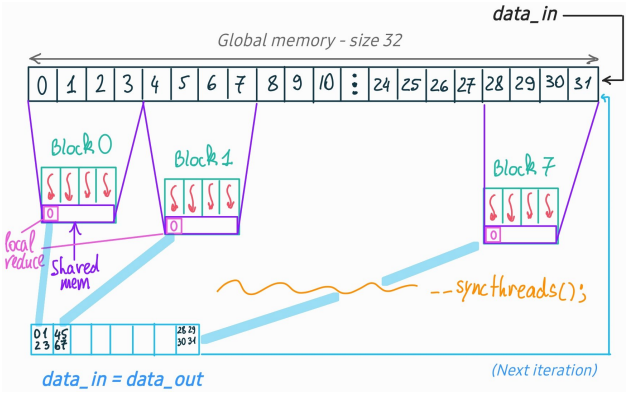


Figure 10: Reduce algorithm using shared memory.

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 [14], 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 6)

To be fully honest, there is almost never a way to get rid of complpetely 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 of accessing 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

[14] 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.

```
__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 7: This method, is ~~almost~~ the same as the first, optimized version of the reduce algorithm, using the shared memory **??***. Therefore, one must note that this reduce_sum() method must be called for the array temp*, located in the shared memory.*

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 divergence perspective, one may notice that there is still a condition, that will potentially lead to warp divergence. Yes, indeed. 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 divergence, the better it is.

thrust::reduce Personally speaking, this code/approach is much easier to understand and visualize, 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.

4 CUDA synchronization mechanisms

As we've discussed in the section on architecture, when writing kernels, we need always 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 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 barrier. This is block-level synchronization barrier.

```
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());
```

Cooperative groups is a relatively new feature to the CUDA API. As the name suggests it, this feature enables us to group threads into groups, with the ability to perform common, collective operations (or simply collectives). We can also perform synchronization between the threads in the same cooperative groups. With these API features, one can simplify the code, thus avoiding common mistakes. For example, one can make the code easier to read. For example, in the code snippet section **??**, we perform the exact same algorithm as in **??**, 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 in the thread group `g`. Then we're calling the `sync()` function, which ensures that all the threads within the block 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. [**?**]

Thread blocks

We've already seen the notion of a thread block many times. This notion was always quite *abstract*. Indeed, while launching the kernel, we've always kept the notion of thread blocks in our mind, but never actually accessed it. However, in newly introduced features, one "access" the thread block explicitly. A quoted "access", because we can actually manage synchronization of the thread blocks. 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 avaliable. 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 that it is less easier to make errors. 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:

```
__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 8: Synchronizing using the legacy mechanism vs using the cooperative groups. [?]

In a nutshell, we clearly see a deadlock in the first piece of code, as there are only threads with ID's less 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 chuncks of threads will just wait for each other.

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 into groups, using either CUDA functionality discussed above, **and only synchronize that block**, which, we are sure, will by run by all threads in the block.

Warp synchronizations

While programming with CUDA, one never gets tired to 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, provide a simple way to navigate through these concepts.

We have discussed a lot the advantages of the shared memory (e.g. for the speed and efficiency of the reduce algorithm). However, the new utilities give us a more faster, or even a 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 name are similar, and only differ by the postfix `_sync()`. For instance, `__shfl_xor()` and `__shfl_xor_sync()`. Indeed, the ones with the `_sync()` postfix is a improvement of the former. It is recommended to use the newer version instead. I will not go into great details between these differences. I will just mention that there are differences in parameters [15](see further examples).

The `__activemask()` primitive/function is actually not a synchronization mechanism, but

[15]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

more of a filtering mechanisms. This function returns the *indices* of the active threads in the warp, where it was referenced from[16]. 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 [**?**] 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 at what did NVidia provide us with [17]

- `__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. 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 ... 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 if we wanted e.g. the thread 29 to have the value 31. (see figure for better understanding).

[16] 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

[17] 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 [**?**].

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.

Atomics

When learning multithreaded programming, one of the first notions, that one must understand is the notion of atomics or atomic operations [18] . 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: [**?**]). 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 adress 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 adress. It also returns the value.

The atomic operations are however very expensive, as the scheduler must perform sequentialize the memory access. It is thus very advisable to use these operations, only when needed. 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, that we will later *consume* from the CPU. So if one suppose 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);
    }
```

[18] 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 [**?**] .

5 Streams

CUDA streams is an another fundamental concept that we've used previously in an implicit manner. Indeed, let me provide an another example using C++: when a "Hello World" programm 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 API, **any** programm is by definition multithreaded. Therefore the analogy between the *usual* C/C++/Python/... and a GPU programm 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 programm (e.g. a simple vector addition) in one stream, which will be performed in parallel. However one can add a second vector addition programm, which will be done in parallel, with the first vector addition.

5.1 Concept of streams

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

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 pass it as the last parameter in the angle brackets. If nothing is passed in the 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 concecutively 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. 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 the previous kernel return. The execition of the kernels are thus independent within the GPU.

The CUDA streams are also able to be synchronized between each other. The call of the `cudaStreamSynchronize()` function which takes a stream as a parameter will wait for the stream to be done.

We've seen that before every kernel execution one needs to transfer data between the host and the device and vice-versa. We've also seen that this operation may take a significant amount of time, compared to the kernel execution time. This time impact can be reduced with streams, by distributing this load between different CUDA streams. This is how we will introduce the concept of asynchronous memory allocation, widely used in the CUDA

code development. This will also introduce the practical concept of the pinned memory allocation/data transfer, theoretically discussed previously 2.5. This implements the data transfer between host-device and device-host within a user-created stream. The detailed sketch and the comments will be listed below (as usual the author takes the code from multiple sources [**?**]). For now, let's simply invoke the functions needed for this operation.

The asynchronous memory copy may not seem very useful in this specific example, as we're only performing one memory copy. However this is useful, when multiple kernels are called, and multiple host to device allocations are needed. In this case, one copy operation will be assigned to every stream.

6 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.