Introduction to programming general purpose GPU accelerated applications utilizing CUDA in C++

Table of Contents

1. Introduction

1.1. Why?

The architecture of GPUs allow for a very different computational workload compared to CPUs. CPUs were originally modeled to consist of only one core which computes things in a linear fashion. Nowadays, due to a slow down in single core performance, most CPU manufacturers are releasing CPUs with multiple cores, however, the number of these cores are still not in any way comparable to those of GPUs. For example the top consumer level Nvidia GPU, the RTX 4090, has almost 15 thousand cores, compared to the usual 8 for modern consumer CPUs. This high core count lends itself to tasks that can run in parallel.

GPUs are utilized in a variety of well known workloads, among others:

- Machine Learning (Neural Networks)
- Rendering Engines & Image Processing
- Simulation Software (Fluid, Medical, Financial, Weather, etc.)

- Crypto Mining (Computation of Cryptographic Hashes)

Basically any computation which can be parallelized well is perfect for GPUs. Memory requirements are also an important factor when considering GPUs, as they are also optimized to have higher memory bandwidths than conventional CPUs.

One must however keep in mind that CUDA is a very low level programming interface for GPUs and is therefore usually only used under the hood by libraries, offering more higher-level features in a language like python. Before considering writing any CUDA code yourself check if there isn't some kind of library in python or another language that offers you the same capabilities with less effort. If you are interested in graphics rending CUDA could be interesting, but again, CUDA only handles the very low-level computational work, while a graphics pipeline like OpenGL handles the possibly more interesting higher-level stuff.

1.2. GPU architecture

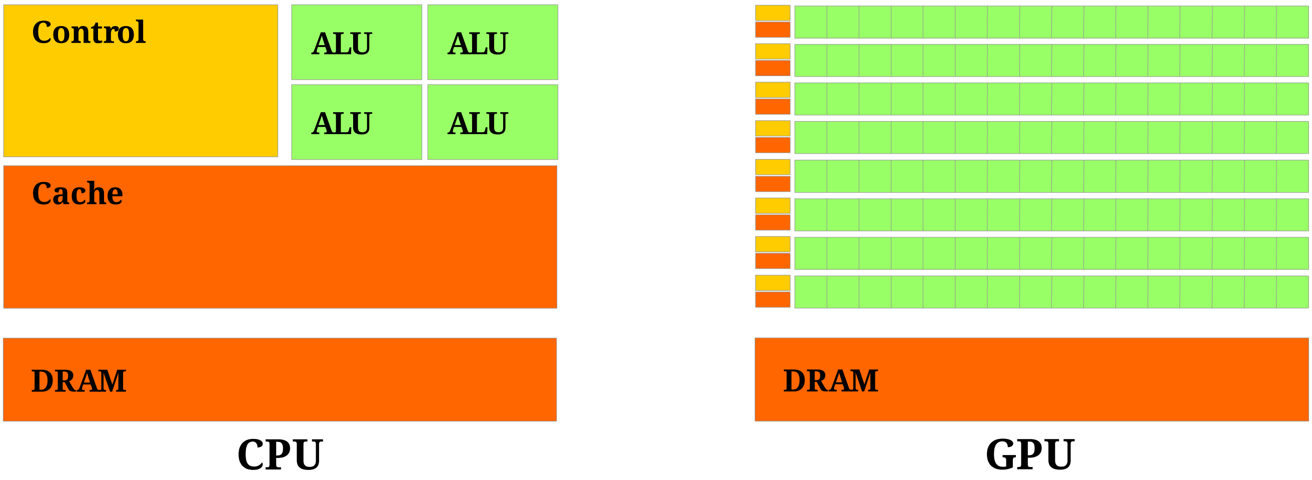


Figure 1. Simplified comparison of CPU vs GPU architecture, NVIDIA, CC BY 3.0 https://creativecommons.org/licenses/by/3.0, via Wikimedia Commons

CPUs typically feature a small number of powerful cores, designed for versatile, single-threaded performance. They excel in handling tasks with intricate branching and diverse instruction sets. In contrast, GPUs comprise a multitude of smaller and simpler cores, emphasizing parallel processing. Their architecture, with shallow instruction pipelines, enables the execution of the same instructions across multiple data points simultaneously, called Single Instruction Multiple Data (SIMD)[1], making GPUs highly efficient for data-parallel workloads.

Memory hierarchy is another differentiating factor. CPUs incorporate a complex hierarchy with various levels of cache to minimize memory access latency. This design suits low-latency tasks and diverse data types. GPUs, on the other hand, prioritize high memory bandwidth over complex hierarchies, which makes them ideal for data-intensive parallel processing but less effective for latency-sensitive tasks.

1.3. Prerequisites

To run the executables provided in the repository you will need a device which has an Nvidia GPU and also somewhat recent Nvidia drivers installed.

If you want to run the provided code directly or modify it you will also need to install the Nvidia CUDA Developer Toolkit Version 12.2, found on the Nvidia website. The toolkit also seamlessly integrates into Microsoft Visual Studio 2022/2019 so using this IDE is recommended.

2. Your first CUDA program

2.1. The Task

For this example we will be looking at the simple task of vector addition, where every nth element of vector A and B are added together to result in vector C. First we need to create the two vectors, fill them, and then perform the calculation. The C++ code, that simply, in a linear fashion, loops through the vectors and adds them would look something like this:

```
#include <iostream>
#include <vector>
#include <chrono>

void vecAdd(std::vector<float>& a, std::vector<float>& b, std::vector<float>& c, int
n)
{
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}

void fillVecs(std::vector<float>& a, std::vector<float>& b, int n)
{
    for (int i = 0; i < n; i++) {
        a.push_back(static_cast<float>(i));
        // example: fill a with values 0, 1, 2, ..., n-1
        b.push_back(static_cast<float>(2 * i));
        // example: fill b with values 0, 2, 4, ..., 2n-2
    }
}

int main()
{
    const int N = 1000000000;

    std::vector<float> A, B, C;
    A.reserve(N);
    B.reserve(N);
    C.resize(N);

    fillVecs(A, B, N);

    auto start = std::chrono::high_resolution_clock::now();
    vecAdd(A, B, C, N);
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop -
start);
    std::cout << "Time taken by function: " << duration.count() << " microseconds" <<
std::endl;

    return 0;
 }
```

2.2. CUDA Implementation

For the CUDA implementation we will just be changing the vecAdd() function, adding a further kernel function and adding a few includes, the rest of the code will remain the same and will not be shown again.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
```

2.2.1. Cuda Memory

For better understanding we will be labeling our variables from here on with the prefixes h_ and d_ to differentiate between variables stored on host memory and variables stored on GPU/device memory.

Until now in our c++ implementation we have just had to handle memory in our RAM. The GPU however can during computation only access the memory which is on the device itself (nowadays multiple 8GBs or more), so previous to computation we will have to handle getting it onto the device. This is achieved through two functions: cudaMalloc and cudaMemcpy.

```
void vecadd(const std::vector<float>& h_a, const std::vector<float>& h_b, std::vector
<float>& h_c)
{
    int n = h_a.size();
    int size = n * sizeof(float);
    float* d_a, * d_b, * d_c;

    cudaMalloc((void**)&d_a, size);
    cudaMemcpy(d_a, h_a.data(), size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&d_b, size);
    cudaMemcpy(d_b, h_b.data(), size, cudaMemcpyHostToDevice);

    cudaMalloc((void**)&d_c, size);
```

cudaMalloc is very similar to the normal malloc function. It is used to first allocate space on the GPU for us to later place data into. It takes two inputs, first a pointer to a pointer is provided, which after execution of the function will be set to the location in memory that was allocated, second the size in bytes we need to allocate.

Following this we can call cudaMemcpy to take the previously allocated space and write the data into it. It takes 4 inputs: the pointer from cudaMalloc which points the the allocated memory, the pointer to the data we want to copy from the host, the size of the data we are copying, and finally the type of copy operation eg. if from host to device, device to host, host to host, or device to device.

2.2.2. CUDA Kernel

The CUDA kernel is where we actually define the operations we want to perform on our data. So in our case the simple addition of two vector elements.

CUDA execution is organized into a hierarchical structure of so called Grids, Blocks and Threads. Threads are the smallest unit of execution the GPU is capable of. In the end each thread is responsible for a small part of the larger task. Multiple threads are organized to be part of a block. Within this block threads can cooperate through shared memory and synchronization. Finally the blocks are organized into the grid, within the grid blocks operate independently and in parallel, they also do not share memory or communicate. We will look at the more complex features of this architecture later in the paper.

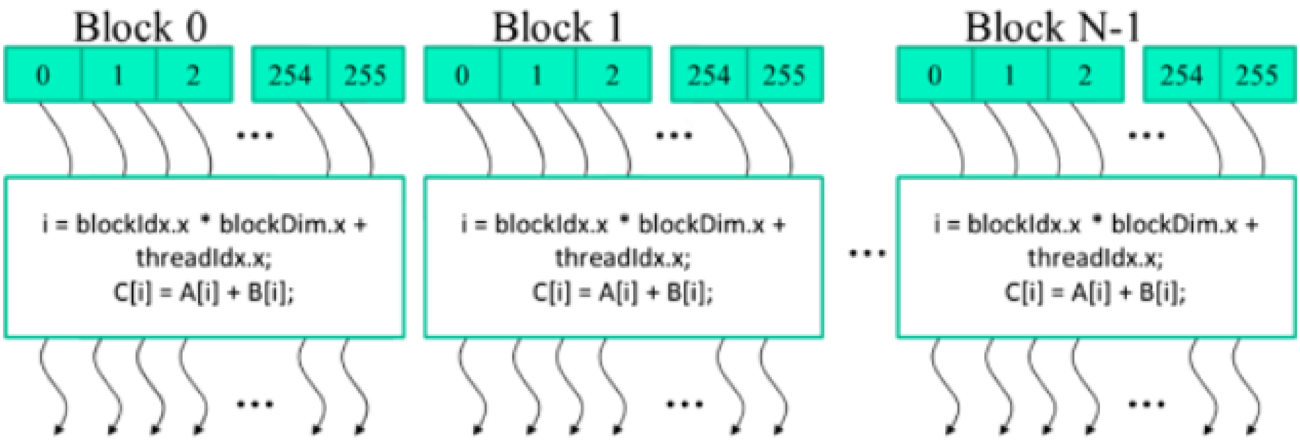


Figure 2. Structure of Blocks and their Threads, source: Programming Massively Parallel Processors

We can now define our CUDA kernel by using the __global__ tag. This designates it as being executed on our GPU device, but only as callable from our host, alternatives are __device__ for kernels only called by the device to be run on the device, and __host__ for CUDA functions called by the host to be run on the host.

```
__global__
void vecaddkernel(float* a, float* b, float* c, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) c[i] = a[i] + b[i];
}
```

Due to the previously explained block/thread architecture the individual thread has to somehow know what part of the data it is responsible for doing work on, in our case which nth elements in the vectors to add. This is done through block and thread Ids. Each block is given an Id starting at 0 and each thread contained in this block is given its own Id. Through the use of these two Ids and the block-dimension we can then calculate which part of the data the thread is now responsible for and assign it to the variable i. Note that due to defining i in the kernel i is private to each individual thread.

```
    vecaddkernel<<<ceil(n / 256.0), 256 >>>(d_a, d_b, d_c, n);
```

Having now defined our kernel we can now complete our vecAdd() function by calling it. We pass in two parameters, the first being how many blocks we want to create, and secondly we pass in the number of threads we want per block[2]. With this we are basically telling the GPU how many parallel computations we want to perform. And lastly we need to hand in the pointers to the inputs for the function.

And at the end of our vecAdd() function we will need to copy the result back to the host and free the memory we previously claimed on the GPU using cudaFree(), passing in the pointer to the device memory we want to free up:

```
    cudaMemcpy(h_c.data(), d_c, size, cudaMemcpyDeviceToHost);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

We have now completed our first CUDA program and are ready to build and compile.

3. Blocks, Threads & Multidimensional Execution

3.1. The Concept

You might have noticed that when previously worked with the blocks and threads we used the BlockIdx.x and ThreadIdx.x, does the .x mean there are also .y and .z for more dimensions? Yes!

Say we want to do some processing on some image, doing some computation on each individual pixel. While we could use our previous 1D approach like with the vector CUDA has a much nicer approach for splitting up the workload.

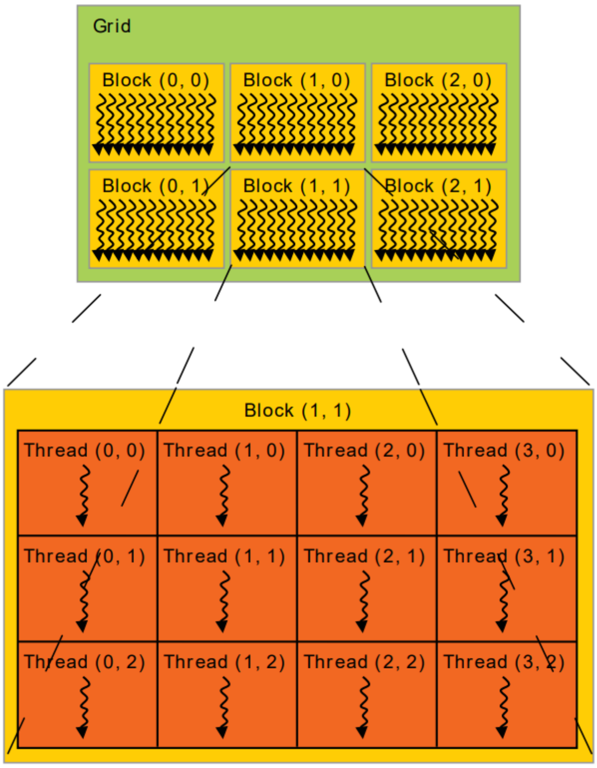


Figure 3. Structure of Blocks and their Threads, source: CUDA C Programming Guide www.nvidia.com

In the picture above is shown how it would look if you were to create a grid with block dimensions of 3x2 and thread dimension of 4x3. The brackets for each Block/Thread denote their respective BlockIdx.x and BlockIdx.y. This can be done by calling below code:

```
dim3 grid(3, 2, 1);
dim3 block(4, 3, 1)
someKernel<<<grid, block>>>(someData);
```

And below is how it would look if we process some small picture, with each thread being responsible for some kind of computation on one pixel. While not displayed in the picture we of course must keep in mind to properly compute the correct indices of the pixel we are processing like in our program from before using BlockDim.x, but additionally using BlockDim.y.

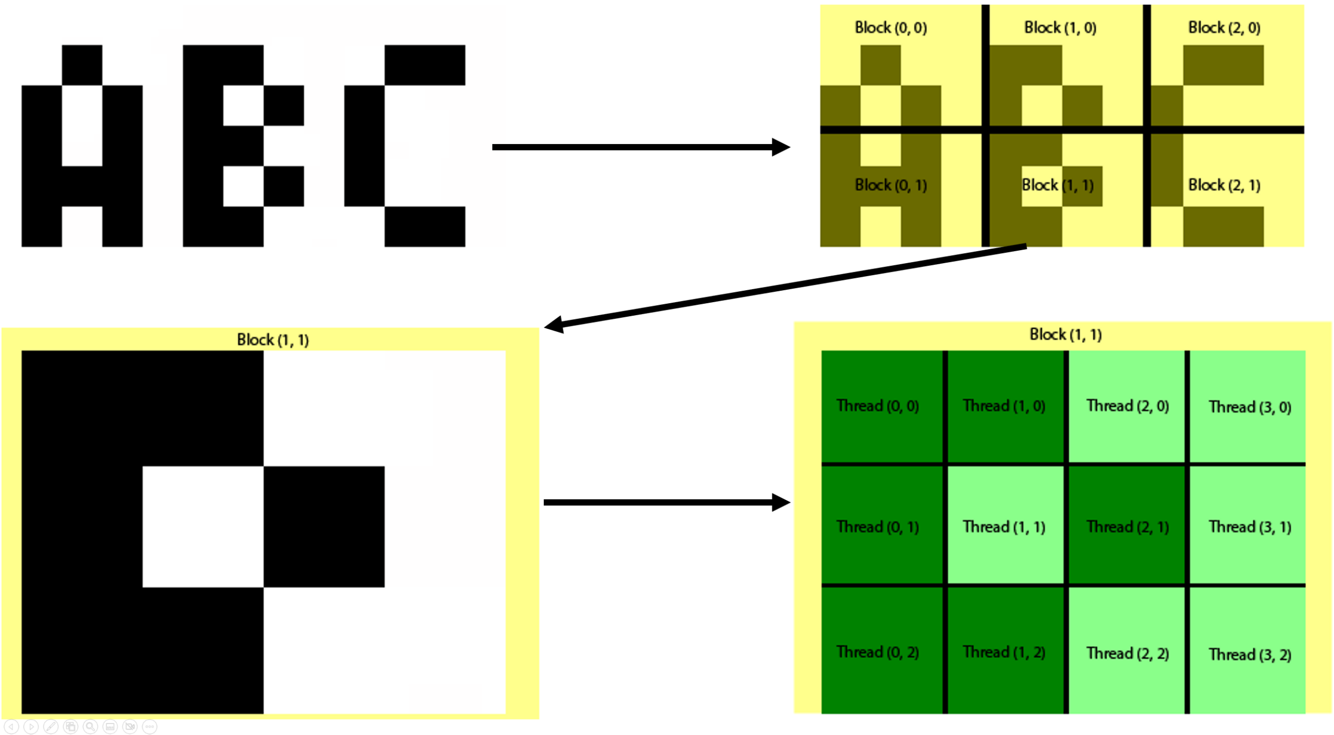


Figure 4. Example of how one would assign threads to individual pixels of a picture using a 2D Block/Thread setup, source: Felix Kunaver :)

While the absence of this feature would not necessarily keep us from doing any computations on 2D/3D data, it allows for much easier implementation of algorithms and also greatly helps when optimizing our code, say in the context of memory calls, but more on that later in the paper.

I hope the graphics help in understanding the concept of the CUDA hierarchy and how it works in multiple dimensions. While not shown here, the same process/concept works also for the 3rd dimension allowing for processing of volumes, tensors, etc.

3.2. Example Of 2D Processing: Matrix Multiplication

Here is a quick reminder of how matrix multiplication works.

Matrices are inherently 2D and lend well to a small demonstration of how one can use this CUDA feature. For sake of simplicity we are just going to handel square matrices as input. Say we have two 4x4 input matrices, also resulting in a 4x4 matrix. We can split up this resulting matrix like shown in the picture below with blocks dividing the matrix larger groups and threads handling the individual results. Of course on this small scale of matrix using multiple blocks does not really make sense, but with a rise in matrix size at some point we would have to use such a design.

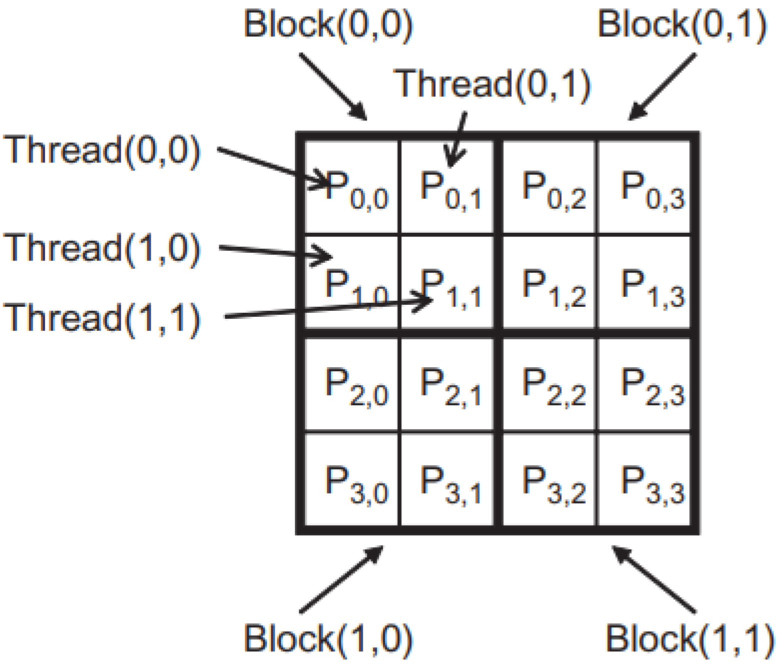

Block(0,0)
Block(0,1)
Thread(0,1)
Thread(0,0)
Thread(1,0)
Thread(1,1)
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$
Block(1,0)
Block(1,1)

Figure 5. Block/Thread view of a 4x4 result Matrix, source: Programming Massively Parallel Processors

Now with our matrices and our computational setup is 2D, optimally we would also like to access our matrices using a two-dimensional array like InputA[][]. However due to the ANSI C version that CUDA is based on, for this to be possible the array sizes have to be known at compile time. So we will still be handling our data as a one-dimensional array by "flattening" it as shown in the picture below. So with a 4x4 Matrix if we would have liked to access an element by matrix[2][3], we will be doing it like this: matrix[2 * Width + 3].

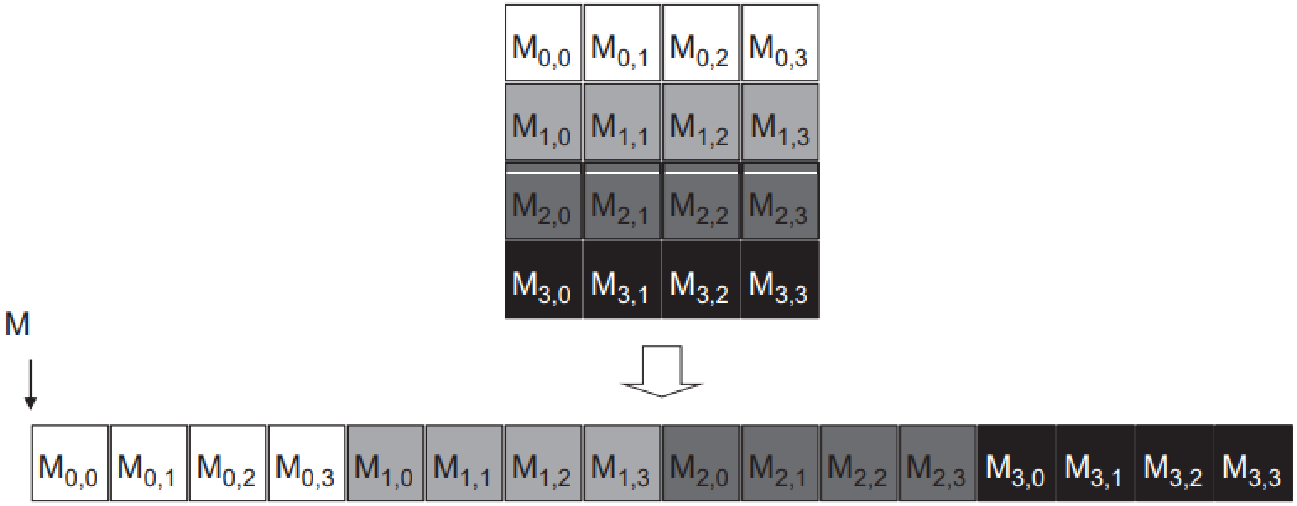

$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Figure 6. Flattening of 2D array into 1D array, source: Programming Massively Parallel Processors

We can now define our cuda kernel:

```
__global__ void MatrixMulKernel(float* InputA, float* InputB, float* Result, int
Width) {
    // Calculate the row index of Result and InputA
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate the column index of Result and InputB
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float ResultValue = 0;

        // Loop through the row/column and compute the resulting value
        for (int k = 0; k < Width; ++k) {
            ResultValue += InputA[Row * Width + k] * InputB[k * Width + Col];
        }

        Result[Row * Width + Col] = ResultValue;
    }
}
```

Each thread, responsible for one element of the result matrix, calculates which row and column it is responsible for and then iterates through all elements in the corresponding input matrices and adds the result together. In contrast to the first example, since we are using 2D blocks and threads as shown in the code below. We now utilize not only the x index of our threads and blocks but also the y index.

```
    // Define grid and block dimensions & Launch kernel
    dim3 dimGrid(2, 2, 1);
    dim3 dimBlock(2, 2, 1);
    TiledMatrixMulKernel <<<dimGrid, dimBlock >>> (d_InputA, d_InputB, d_Result,
Width);
```

The memory handling, eg. allocation, copying and freeing stays the same as in the first example, so it is not shown but can be found in the repository.

4. Another Look At Memory

4.1. Memory Hierarchy

When programming CUDA we always have to keep in mind the hardware architecture our code will be running on. One important aspect of this is the memory design on modern Nvidia CUDA GPUs.

A CUDA GPU has several different types of memory organized into a straightforward hierarchy displayed in the picture below. Knowing where to optimally place ones data can have huge performance implications and is one of the most important topics when looking at optimization.

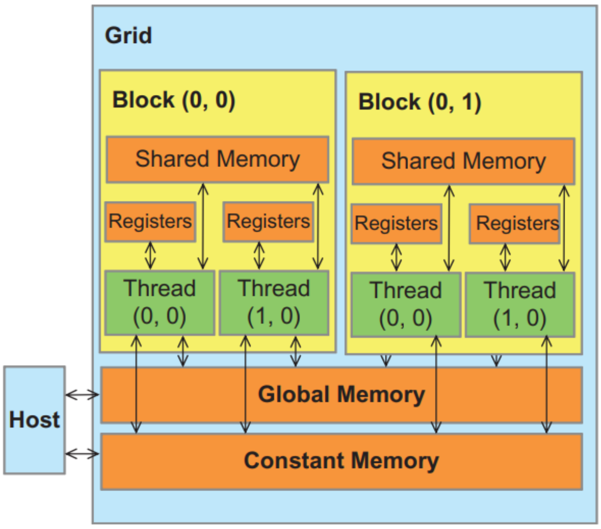


Figure 7. Diagram of the CUDA memory hierarchy, source: Programming Massively Parallel Processors

Lets take a look at the roles of the diffrent memory types:

- **Registers** are the fastest type of memory found on the GPU. Each thread has its one registers that are private to it. Registers can only hold a small amount of data[3]. When we create a scalar variable inside our kernel function, such as the index in our first example, then these are stored within the threads register.
- **Shared Memory** is accessible by all threads in one block. Access to it is faster than access to the global memory, but slower than to a threads register. Using the __shared__ keyword in front of a variable we set in a kernel function makes it accessible to the other threads in the same block. More on the use of this memory in the next chapter on Tiling.
- **Global Memory** is the memory we can write into using cudaMalloc and cudaMemcp. It is the slowest of the memory types available on the CUDA GPU, but is accessible from all threads and blocks and is the largest. In our programs we will usually load our data in here initially but try and get it closer to the threads and into faster memory as quickly as possible. All non-scalar, eg. array variables, we create in our kernel function are stored here by default.
- **Constant Memory** is similar to global memory, with the difference that it can only store constants. Due to this it has slightly faster read times than the global memory. It is set by declaring a variable outside any function body with the __constant__ keyword.

4.2. Tiling

With our previous matrix multiplication example we are reading elements from our input matrices multiple times. For example, as shown in the picture below, both thread(0, 0) and thread(0, 1) will read the first row of our A input matrix from global memory. This is inefficient and is likely to cause memory bandwidth issues. With tiling we will get our threads to cooperate on this reading, and load parts("tiles") of our data into shared local memory for faster overall access.

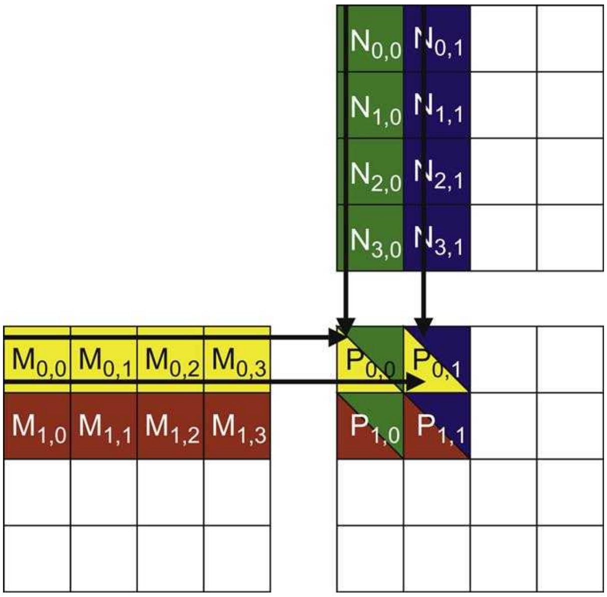
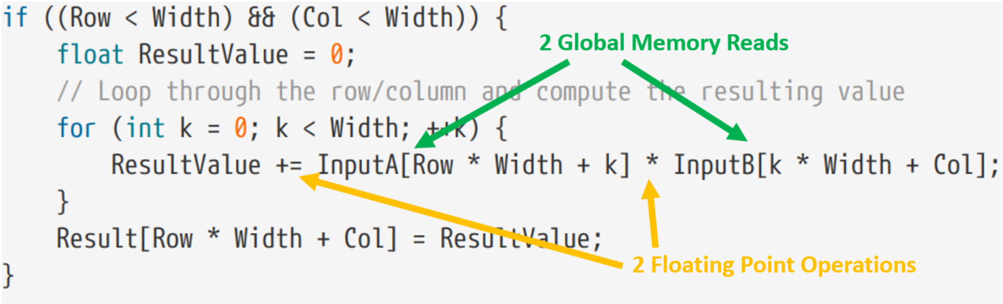


Figure 8. Multiple threads reading same data, source: https://nichijou.co/cuda7-tiling/

One factor to look at whether to implement tiling is the ratio of floating point operations to global memory loads for a kernel. Taking a look at our original code as shown below we can see both two read operations to our global memory and two floating point operations bringing us to an overall "Compute to Global Memory Access Ratio" (CGAM Ratio) of 1. The lower this ratio the more likely we are to have our performance be memory bound, the higher and we are compute bound.

To see where we lie we have to take a look at the hardware we are running our code on, for this example lets look at the top of the line Nvidia H100. It has a global memory bandwidth of 3TB/sec and compute performance of 60Terra floating-point operations per second (FLOPS). Each floating point element we are reading is 4 bytes so we can at maximum compute at 0.75Terra FLOPS because 3TB/4B = 0.75T, and our CGAM Ratio of 1. This is way below what the H100 would be capable of with its 60Terra FLOPS but we are severely memory bound. If we can get our CGAM Ratio to be higher, meaning more operations per memory load, then we would be able to use more of our compute capabilities. Say with a ratio of 10, we would be able to compute at 7.5Terra FLOPS instead of our original 0.75TFLOPS.



```
if ((Row < Width) && (Col < Width)) {
    float ResultValue = 0;
    // Loop through the row/column and compute the resulting value
    for (int k = 0; k < Width; ++k) {
        ResultValue += InputA[Row * Width + k] * InputB[k * Width + Col];
    }
    Result[Row * Width + Col] = ResultValue;
}
```

Figure 9. Diagram of the CUDA memory hierarchy, source: Felix Kunaver :)

When tiling we alleviate this issue somewhat by getting rid of duplicate global memory reads by different threads in the same block.

Since for our simplified example we assume the input and output matrices to be of the same size, we can solve this duplicate memory issue by giving each thread first the task to load their respective elements from input into shared memory. Thread(0, 0), computing element (0, 0) in the output matrix loads element (0, 0) from both input A and B, thread(0, 1) does the same with elements (0, 1) and so on. However, since our shared memory is obviously smaller than the global one we will usually not be able to load all the data into shared memory at once so we only load a smaller chunk aka. "Tile". This way no two threads will have to read the same data twice from global memory.

4.3. Tiling Example: Matrix Multiplication Optimization

We start off our tiled kernel by declaring our shared memory versions of our input matrices using the __shared__ keyword. Note that we can use a two-dimensional array for the shared matrices as the size, eg. the tile size, is known at compile time. One could also modify the program so that the tile size is adjusted based on the provided hardware, then the code would also have to be modified to use our "flattend" array as previously shown.

```
__global__ void TiledMatrixMulKernel(float* d_InputA, float* d_InputB, float*
d_Result, int Width) {
    __shared__ float SharedA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float SharedB[TILE_WIDTH][TILE_WIDTH];
```

Next we compute the row and column we are working on. This has to be done since our input is "flattend"/one-dimensional.

```
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_Result element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float ResultValue = 0;
```

Next we enter the loop and each thread loads their part of the data from the global memory into our arrays in the blocks shared memory. This is followed by a call to __syncthreads()

As previously stated the threads in a block actually have the capacity to communicate and share memory, this allows for more complex algorithms and optimizations, however we sometimes need to synchronize the activity of our threads. This can be achieved through the __syncthreads() function. When called each thread in a block will wait until all other threads of the block have reached this point in the code until continuing. If our kernel code includes any if else we have to make sure that all threads will hit their __syncthreads() calls, as long as at least one of our threads in a block is occupied doing something else all other threads will be waiting for it.

In our code we need to use __syncthreads() since we need to be sure that all of the threads have put their part of the data into shared memory before we start accessing it. Would we not sync here we could run into issues where one thread that was a bit faster is already reading an element from shared memory before another thread has actually written the correct value into shared memory.

```
    // Loop over the d_InputA and d_InputB tiles required to compute d_Result element
    for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {

        // Collaborative loading of d_InputA and d_InputB tiles into shared memory
        SharedA[ty][tx] = d_InputA[Row * Width + ph * TILE_WIDTH + tx];
        SharedB[ty][tx] = d_InputB[(ph * TILE_WIDTH + ty) * Width + Col];
        __syncthreads();
```

Finally we can do our computation as also previously done. Once we have finished with our tile, we have to sync again. This is done because, depending on our tileSize and input, our threads will do multiple iterations before coming to their individual results and we do not want fast threads to overwrite our shared memory while some other thread might still need to read the overwritten element.

Finally we can save our result back into global device memory and our kernel is done computing.

```
        // Same computation as previously
        for (int k = 0; k < TILE_WIDTH; ++k) {
            ResultValue += SharedA[ty][k] * SharedB[k][tx];
        }
        __syncthreads();
    }
    d_Result[Row * Width + Col] = ResultValue;
}
```

The kernel call works the exact same as with our previous example.

We can now take another look at our CGAM Ratio. We have two global memory reads per iteration in our outer loop, and two shared memory reads, which are not counted, as well as two floating pint operations in our inner loop. As the number of times the inner loop is run depends on the tile width, we can determine our new CGAM is equal to the tile width, for example with a tile width of 16, we would be getting a CGAM Ratio also of 16. This is a massive improvement over what we had previously.

Below is the kernel in its entirety for better reading:

```
__global__ void TiledMatrixMulKernel(float* d_InputA, float* d_InputB, float*
d_Result, int Width) {
    __shared__ float SharedA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float SharedB[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_Result element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float ResultValue = 0;
    // Loop over the d_InputA and d_InputB tiles required to compute d_Result element
    for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {

        // Collaborative loading of d_InputA and d_InputB tiles into shared memory
        SharedA[ty][tx] = d_InputA[Row * Width + ph * TILE_WIDTH + tx];
        SharedB[ty][tx] = d_InputB[(ph * TILE_WIDTH + ty) * Width + Col];
        __syncthreads();

        // Same computation as previously
        for (int k = 0; k < TILE_WIDTH; ++k) {
            ResultValue += SharedA[ty][k] * SharedB[k][tx];
        }
        __syncthreads();
    }
    d_Result[Row * Width + Col] = ResultValue;
}
```

5. Next Steps & Further Resources

You should now be able to grasp the basics of CUDA programming in C++. While this is by no means an in depth guide it should allow you to write at least some simple CUDA kernels.

If you want to learn more about CUDA in general you can find some good resources below, which were also used as sources for this text. A lot of details had to be omitted to get to such a concise text and reading these books/guides will fill in a lot.

- Programming Massively Parallel Processors by David B. Kirk & Wen-mei W. Hwu
- The official Nvidia C++ CUDA Guide
- GithubRepo with sample implementations for algorithms ranging from linear algebra, to simple rendering engines, fluid simulations, finance and much more
- A university students notes on CUDA, much shorter than the above two, but longer and more detailed than this text

Below are some more topics that might be interesting to look at if some particular aspect of this paper looked interesting:

- You want to know more about machine learning/neural networks then look at cuDNN, the state of the art library under the hood of most machine learning applications today
- Math interest you? Take a look at cuBLAS a linear algebra library that forms the bedrock of most more complex algorithms
- If you are interested in diving deeper into the memory optimization take a look at memory read coalescing
- If you have been bothered throughout this paper by the question how to choose the number of threads to have in a block take a look at "Streaming Multiprocessors"(SMs), "Streaming Processors"(SPs) and "warps".

- For anything else check the sources above

[1] Cuda utilizes the paradigm of Single Program Multiple Data (SPMD), the main difference being that not the exact same instruction has to be executed in each thread, but the same program, this allows for branching and more.

[2] This factor depends on the algorithm one is running. Depending on the amount of shared memory, registries and communication the individual threads need, the amount should be higher or lower. Most modern Nvidia GPUs allow for up to 1024-4096 maximum threads per block.

[3] While we technically don’t have a limit of how big the resister of a single thread is, due to the usage of SMs (Streaming Processors), it is preferable to keep the amount of register memory used as low as possible. For more details of SMs look at the chapter next steps.