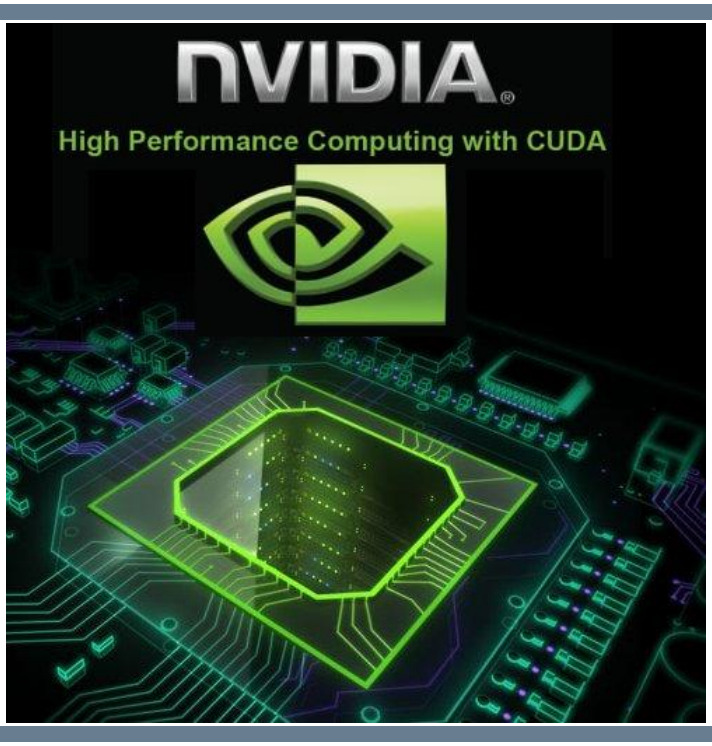



# CUDA Snippet Documentation

## GPU Acceleration Programming

Division: HuCE - OptoLab
Author: Daniel Tschupp
Date: 10.04.19

Bern University of Applied Sciences

# Versions

| Version | Date | Description | Author |
|---|---|---|---|
| 0.1 | 22.11.2017 | Template for simple reports | Daniel Tschupp |
| 0.2 | 13.04.2018 | Changed everything to english | Daniel Tschupp |

# Table of Content

**1 GPU Architecture** **1**
1.1 Cores . . . 1
1.2 Memory . . . 2
1.2.1 Memory Parallelism . . . 2
1.2.2 Memory Coalescing . . . 3
1.2.3 Corner Tuning . . . 3
1.3 Performance Considerations Overview . . . 5

**2 Programming Snippets** **6**
2.1 Hello World Snippet . . . 6
2.2 Create Random Matrix File . . . 6
2.3 Matrix Multiplication . . . 6
2.4 System Info Readout . . . 6
2.4.1 Clock Rate . . . 6
2.4.2 Number of Streaming Multiprocessors . . . 7
2.4.3 Number of Cuncurrent Kernels . . . 7
2.4.4 Warp Size . . . 7
2.4.5 Total Global Memory . . . 7
2.4.6 Total Constant Memory . . . 7
2.4.7 Shared memory per Block . . . 7
2.4.8 Memory Pitch . . . 7
2.4.9 Max Threads per Block . . . 8
2.4.10 Max Blocks per Grid . . . 8
2.4.11 Max Registers . . . 8
2.5 Matrix Calculation using Blocks . . . 8
2.6 Matrix Calculation using Shared Memory . . . 8
2.7 Matrix Calculation using Constant Memory . . . 8
2.8 Comparing different Methods for Image Substraction . . . 9
2.9 Memory Types Demonstration . . . 9
2.10 Simple Asynchron Memory transfer Example . . . 9
2.11 Asynchron Memory transfer Class . . . 10
2.12 Improved Streamer Class . . . 11
2.13 Comparing different matrix multiplication kernels . . . 12

**Bibliography** **I**

**Glossary** **II**

**Acronyms** **II**

# 1 GPU Architecture

In this chapter the architecture of a modern GPU is explained. It is crucial to understand how the GPU works to write and execute fast kernel.

The GPU architecture looks like this:

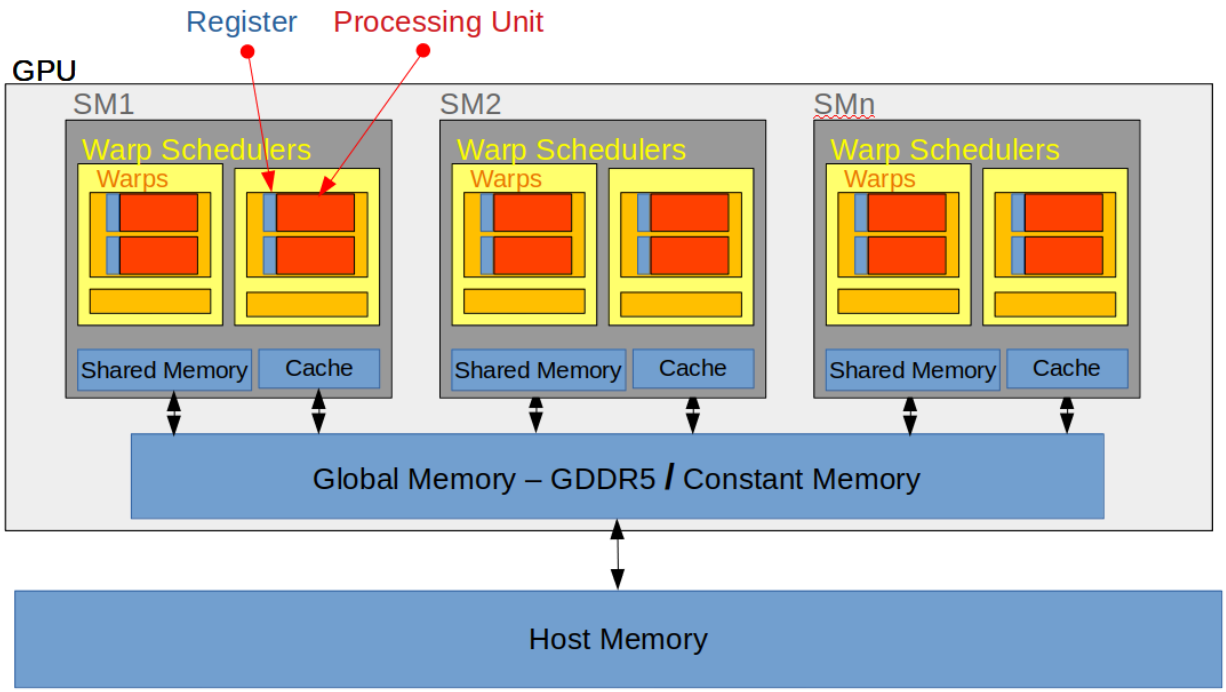



Fig. 1.1: Architecture of a GPU

[1, Parallel Computing Book]

## 1.1 Cores

Figure 1.1 shows the architecture of a GPU. A modern GPU contains multiple so called streaming multiprocessors (SM). These work after the Single Instruction Multiple Data (SIMD). This means that there's only one instruction line that will be carried out by each processing unit. Special hardware indexers can identify the data on which this instruction operates for each processing unit individually. That's how it's possible to minimize chip-space by only having one instruction core but tons of data cores working on different junks of data.
Each processing unit contains at least 1 FPU, 1 int PU. It may also contain special function units, tensor calculation units, texture filtering unit etc. The Floating point unit can calculate 1 float operation per cycle. It needs multiple cycles to calculate doubles and can calculate even 2 16bit floats per cycle. Integer processing unit is build up similarly.
The streaming processor can work on multiple blocks. Those will be split into junks of 32 threads called warp. Those warps are put into a queue and wait for execution. The SM consists of multiple warp schedulers each consisting of 32 processing units that will work through the queue. Here it's important to mention that they don't work

linearly through the actual set of threads. They just process the thread until a time consuming instruction like a load/store/branch starts. Then they switch to another set of 32 threads and continue at their actual instruction location. This is done because processing units are build up simple to minimize chip-space so they don't include stuff like branch prediction, pipe-lining etc. to work efficiently through time consuming instructions. Instead they rely on a how bunch of thread to compute in parallel to bypass such time consuming instruction. Load/store to global memory are the longest instructions. They need about 100 Cycles to complete. This means we would need about 32*100=3200 threads to really fully utilize the all processing units while loading / storing data.

## 1.2 Memory

There are 5 types of memory inside a GPU: Global Memory, Constant Memory, Shared Memory, Caches and Registers. Those can be seen in figure 1.1. The first two of those can be written to by the host CPU with an API instruction (cudaMemcpy, cudaMemcpyToSymbol). To the later two there's only access from within the kernel. As Shared memory as well as caches are on chip memories they're very fast but cost also a lot of chip-space. Therefore it's a very limited resource. Registers are even faster than shared memory and cache because there directly embedded in the computing units. The constant memory can only be written to by the host controller but not by a kernel. The kernel has only read access. That's because everything that is stored inside the constant memory will be copied to the cache when the kernel launches which makes it very fast, but also limited in space.
The last memory is the global memory which is the biggest and by far the slowest one. Therefore it has a lot of potential to improve performance.

### 1.2.1 Memory Parallelism

As the access times for DDR is very long (multiple nanoseconds) it's build up is highly parallel to reach the bandwidth modern CPUs/GPUs demand. Figure 1.2 shows a simplified DDR system. It's basic cell stores one bit of information via a capacitance. Multiple Cells are combined into a block and multiple blocks serves a channel. Each channel has it's own channel controller that is capable of issuing multiple data requests. Those requests are concurring IF they don't request data from the same block. This makes it possible to interleave data requests like it's shown in figure 1.3.

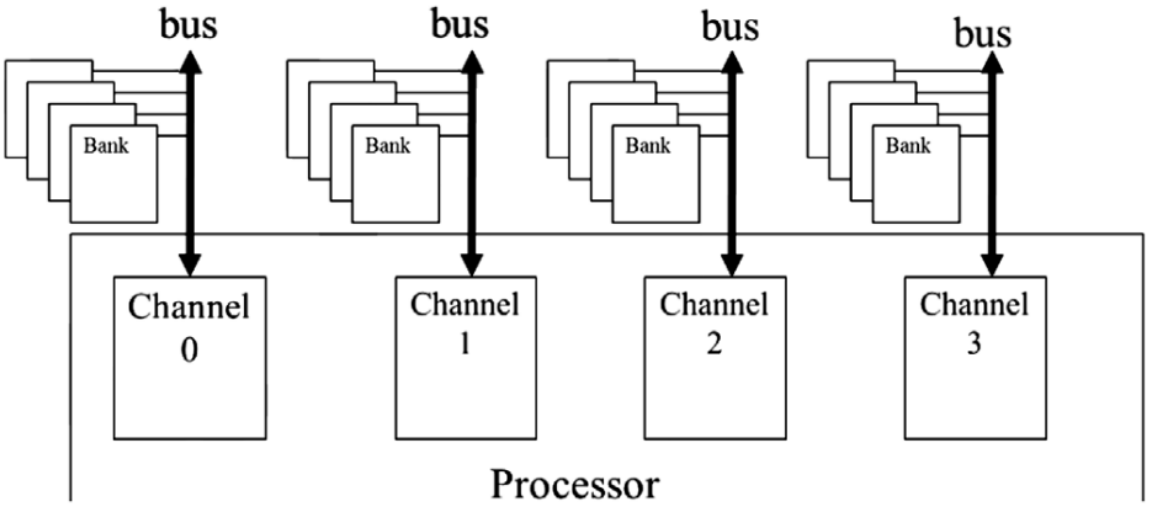



Fig. 1.2: Architecture of GDDR5. [1, Parallel Computing Book, p. 112]

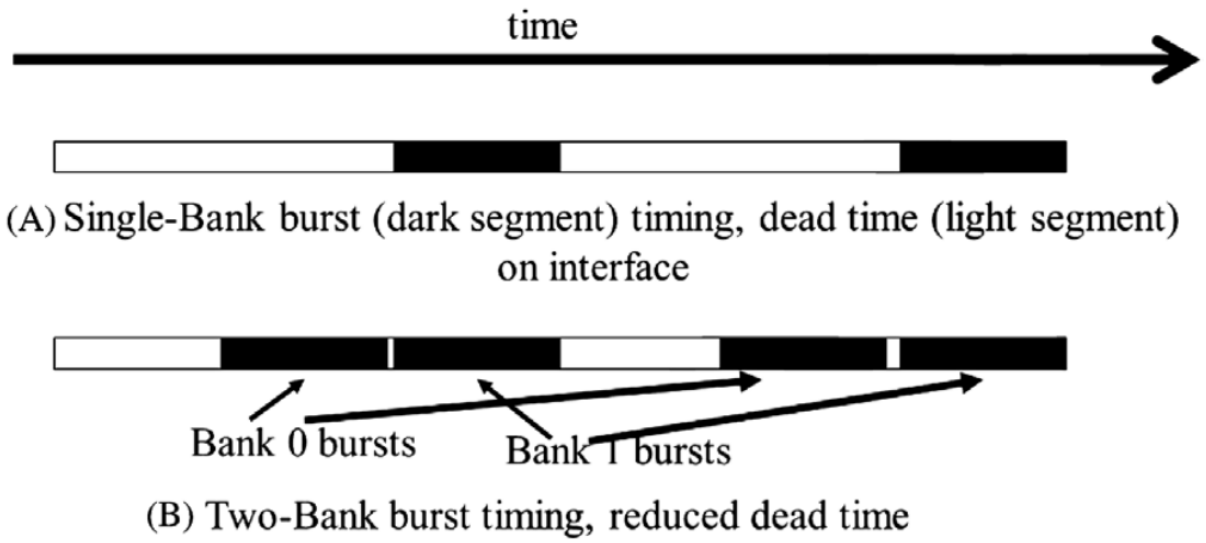



Fig. 1.3: Shows interleaved memory access.[1, Parallel Computing Book, p. 113]

As this is done automatically the programmer must only ensure that enough thread simultaneously access the memory to utilize the whole bandwidth available. As the granularity of those blanks is very fine (128byts = 32floats) the max performance will be reached if the thread slots inside the streaming multiprocessors are utilized and a whole warp accesses data that is coalesced.

### 1.2.2 Memory Coalescing

Another important thing for accessing the global memory is to access grouped data as the device can access the global memory only in transactions of 32-, 64- or 128-bytes. Those bytes of one transaction are of sequencial memory locations (Example: N, N+1, N+2...N+64). Therefore if only one float value (4 bytes) is needed 3/4 of the transaction is wasted. Figure 1.4 shows a good example for storing a matrix in the global memory. When accessing the data from the kernel in the order the data is stored, the maximum amount of coalescing is used.

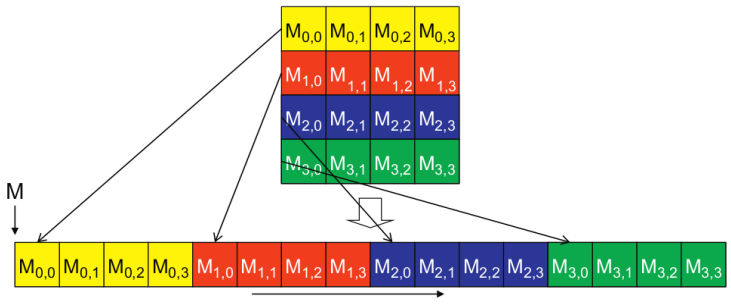



Fig. 1.4: Coalescing memory. [1, Parallel Computing Book, p. 106]

### 1.2.3 Corner Tuning

Corner Tuning is a method to access uncoalesced data in a coalesced way. This is done with using the shared memory. As access times don't differ in the shared memory for coalesced and uncoalesced data the solution is simply to copy the uncoalesced data in a coalesced way from the global memory to the shared memory and do the uncoalesced access there like it's shown in figure 1.5

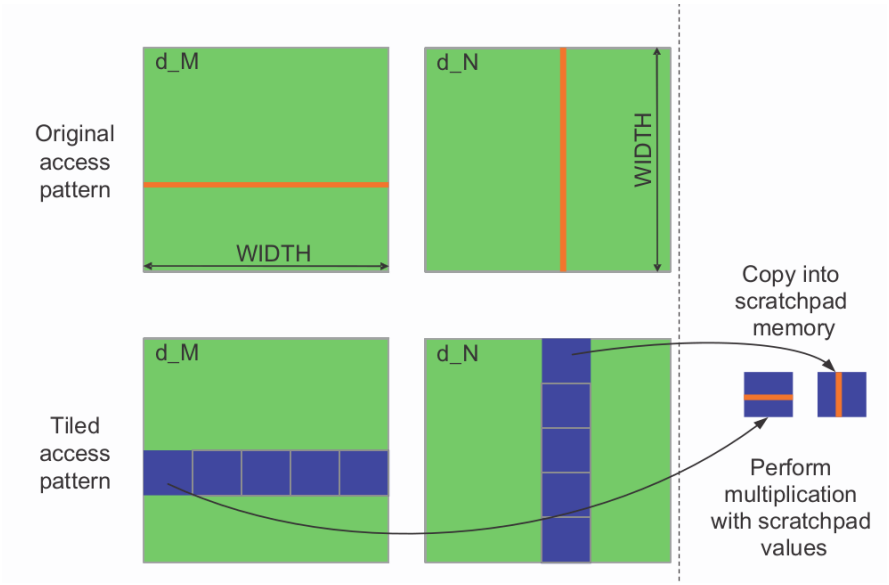



Fig. 1.5: Image shows corner tuning of uncoalesced data. [1, Parallel Computing Book, p. 110]

## 1.3 Performance Considerations Overview

The following checkpoints should be considered when optimizing an algorithm:

- Are the resident threads of each SM nearly fully used?
- Isn't the number of threads reduced by the amount of registers used inside the kernel?
- Isn't the number of threads reduced by the amount of shared memory used be the kernel?
- Isn't the number of threads reduced by the max. number of blocks that can reside inside a SM?
- Are global memory accesses in coalesced bursts of at least 128 bytes.
- Are there enough parallel global memory accesses to fully utilize the global memory access bandwidth?
- Is Host-To-Device data transfer interleaved over several kernel calls to use the maximal PCI bandwidth?
- Did you use shared / constant memory for global memory data that's accessed more than once? (Don't forget to corner tune your data)
- Did you maximize the compute-to-global-memory-access ratio? (Higher than 30 is good, higher than 50 is excellent.)
- Did you use caching effects as well as possible for similar data access that's not to the shared memory.
- Is the warp padding minimized?
- The most important performance aspect is to have a good, parallel algorithm.

This is just a list of points to give you an idea where to start and what to optimize.

# 2 Programming Snippets

## 2.1 Hello World Snippet

The first snippet shows how to simple transfer data synchronously to the GPU global memory and manipulate it with a kernel. It's a good place to start with and to check whether your installation of the cuda toolkit and nvidia drivers work as expected.

It simply moves a character to the GPU memory and manipulates it to change the "CPUßtring to a "GPUßtring, which will be read back and written to the console.

## 2.2 Create Random Matrix File

The second snippet won't do anything on the GPU. But for testing with kernels it's handy to have some files with matrices to compute. This Snippet provides the means to create, store such matrices, as well as write them to the console. For unknown reasons it's not possible to create matrices that are bigger as 1000x1000 elements.

## 2.3 Matrix Multiplication

The third snippet performs a first matrix multiplication kernel that computes the matrix multiplication out of the global memory of the GPU. Furthermore it compares the result to a algorithm that performs the same multiplication on the CPU. You will see, that the CPU performs faster for small matrices because there's no need to transfer the data to the GPU.

You will also see, that the maximal size of the matrix that can be calculated on the GPU is limited. This is due to the fact that in this snippet (for simplicity) we use only one block and this block is limited to 1024 Threads, that means we can only calculate matrices with up to 1024 elements (Depending on the GPU it might be less or even up to 2048).

## 2.4 System Info Readout

In order to write an optimal kernel for the given hardware it is essential to know some specifications of the GPU. This snippet provides the means to read out the important ones and store those inside a file.

Particularly we're interested in the following specs that are described below.

### 2.4.1 Clock Rate

This is the computing unit clock rate and defines the speed for calculations. It's in the area of MHz which doesn't seem very fast, but remember we do have tons of computing unit in parallel.

### 2.4.2 Number of Streaming Multiprocessors

A streaming multiprocessor is a collection of computing units called a Warp. It processes multiple given blocks. Actual GPUs can process up tu 8 blocks in "parallel". Parallel is hear in quotes because it contains a limited number of computing units which can work really in parallel, the rest of the threads will be executed in serial batches of those parallel executable threads.

### 2.4.3 Number of Cuncurrent Kernels

This parameter specifies how many different kernels can be executed by the GPU at the same time.

### 2.4.4 Warp Size

As mentioned in the Streaming Multiprocessor section a SM has one Warp that are 32 (in most actual GPUs) concurrent processing units. As the GPU doesn't have a hardware pipeline for executing commands efficiently it simply calculates another batch of data while waiting for time consuming instructions like memory loads, memory stores, branches etc. Therefor it is crucial to provide enough threads to a streaming multiprocessor in order to fully utilize the computing units. Remember a load/store instruction to the global memory needs about 100 cycles whereas a floating point instruction only needs one. When analyzing a kernel one often counts the number of global memory accesses as well as the number of floating point operations and creates the compute-to-global-memory-access ratio.

### 2.4.5 Total Global Memory

This parameter describes the maximum random access memory of the GPU. This memory is the biggest one and is the only one that the host CPU can send data to. This means CPU, as well as all Cores of the GPU have read and write access permission to it so synchronization is an issue with this type of memory.

### 2.4.6 Total Constant Memory

The constant memory is a read only memory for the GPU. The host CPU can initialize this constant memory prior to calling a kernel with special API functions. All data stored into the constant memory will be load into the L2-Cache of each streaming multiprocessor. Therefor it's always available like a global variable inside the kernel and very fast. Even faster than shared memory but slower than the registers.

### 2.4.7 Shared memory per Block

The hardware of the streaming multiprocessors contains on chip memory called Shared Memory. This kind of memory is way faster than the global GDDR5 memory. But only the streaming multiprocessor has read/write access to it. This means that the kernel has to copy the data from the global memory to the shared memory. As memory requires a lot of chip-space it's also a very expensive resource. That's why it not very big.

### 2.4.8 Memory Pitch

The global memory of a GPU may consist of multiple memory controllers which each manages a portion of the whole memory. The pitch describse how big such a portion is.

### 2.4.9 Max Threads per Block

A streaming multiprocessor is in its capability also limited in the number of maximal threads to handle, not only the number of blocks. This is due to the hardware indexers. Therefor it exists a maximal number of thread a SM can handle in each dimension (X,Y,Z) and as a whole.

### 2.4.10 Max Blocks per Grid

Similar to the max threads per block, there exists a maximal number of blocks an SM can handle. This number is usually very big and not limiting.
Remember this number has nothing to do with the parallel computation capabilities of the GPU. The number of SM, the Clock and the Warp size divine that. The Max Blocks per Grid can be interpreted as a maximal queue length for blocks waiting to be computed by the SMs.

### 2.4.11 Max Registers

This number describes the total count of registers that are available inside a streaming multiprocessor. This resource is shared between all the Treads of a block, not all the thread inside a warp to enable fast switching between different warp sets. So pay attention that the register memory won't limit the number of threads/blocks that can reside inside a streaming multiprocessor.

## 2.5 Matrix Calculation using Blocks

To be able to compute bigger matrices than the maximal number of threads per block we obviously need multiple blocks. This snippet shows how to declare a grid of blocks and how to use the block index hardware signal to address the correct data portion to process.

## 2.6 Matrix Calculation using Shared Memory

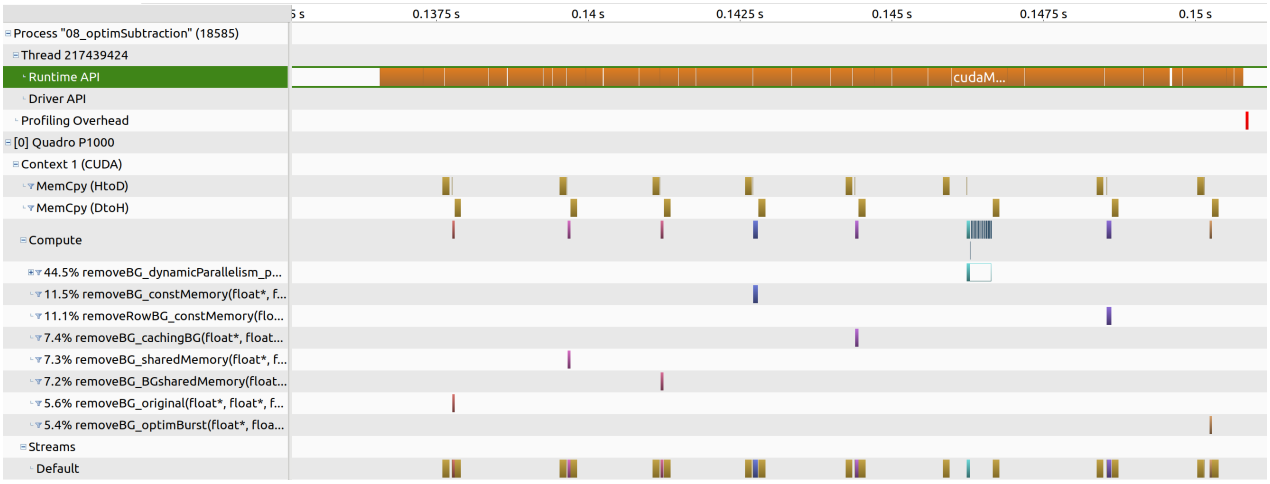
As we have seen in the GPU Architecture part that a streaming multiprocesser contains on-chip-memory which is way faster than the global memory we want to use that to minimize global memory access. This snippet shows how to split the data into small portions called tiles. For those tiles memory will be allocated inside the kernel and the input data from the global memory will be copied into the shared memory of the tile.

## 2.7 Matrix Calculation using Constant Memory

The last type of memory we can access is the fast constant memory. This snippet shows an example how to access it.

## 2.8 Comparing different Methods for Image Substraction

It's a common task to remove a background light intensity of a camera image. Therefore this snippet shows how tho subtract a vector from each row of an image efficiently.




As it can be seen by the analysis it's not possible to speed up this kind of kernel though shared memory. The constant memory approach doesn't help much neither because the vector isn't very big, so it will be stored inside the cache anyway.
So the only thing speeding up this kind of kernel are optimal burst transfers. The burst transfers are already very optimal in the original kernel, that's why the optimBurst kernel is only slightly better.

## 2.9 Memory Types Demonstration

This snippet shows how to prepare the host memory for best possible transfer speeds.

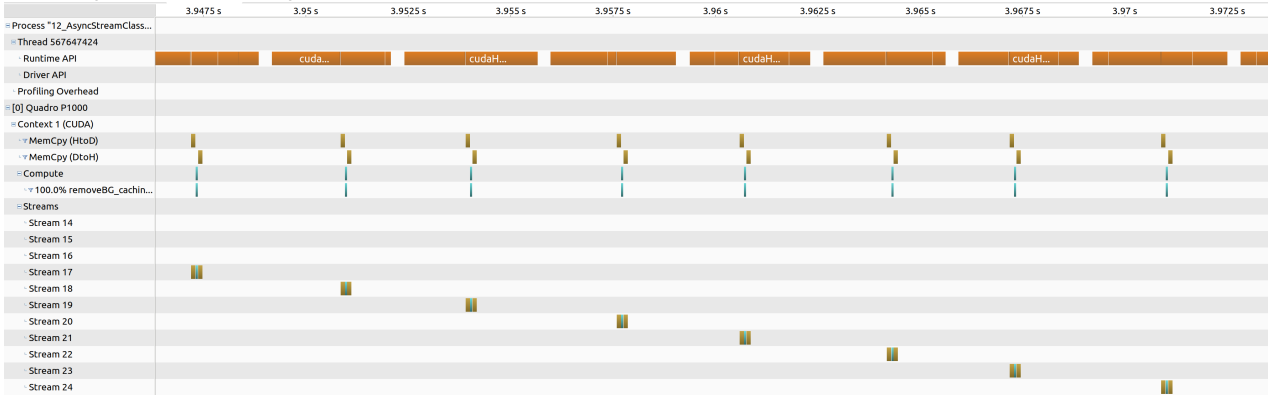
## 2.10 Simple Asynchron Memory transfer Example

Streams are the tool of CUDA to enable asynchronous (interleaved) data transfer from host memory to GPU memory. This snippet shows how to create streams and assign kernels to the streams.
In the examples before we used a stream to, but it was hidden. We used the default stream which is always synchronous. [2, CUDA Programming Guide, chapter 3.2.5.5ff]

## 2.11 Asynchron Memory transfer Class

This snippet implements a streamer class that creates a stream and manages the data transfers and kernel executions.
When analyzing the code with the nvidia-profiler tool we see that data transfers and kernel execution ar the smallest part of the whole execution time. In the next snippet we're this will be corrected.

The output of the nvidia-profiler:




We can see now, that the kernels indeed execute in different streams concurrently. But the main time is used for allocation etc, so it's not more efficient than the sequential kernel execution.

Output readings:

| | |
|---|---|
| cudaHostRegister input image: | 403us |
| cudaMalloc input image: | 213us |
| cudaHostRegister output image: | 843us |
| cudaMalloc output image: | 193us |
| memCopy input image: | 50.5us |
| kernel: | 193us |
| memCopy output image: | 50.5us |
| cudaHostUnregister input image: | 580us |
| cudaFree input image: | 136us |
| cudaHostUnregister output image: | 359us |
| cudaMalloc output image: | 140us |

We can calculate the theoretical throughput for BScans with 640 A-Scans per BScan: $C = \frac{1}{2*50.5us+192us} * 512 \approx 1.7M$ AScans per second.
But we lose a lot of time with allocation and memory pinning in this implementation which limits us to 160k AScans per second. [2, CUDA Programming Guide, chapter 3.2.5.5ff]

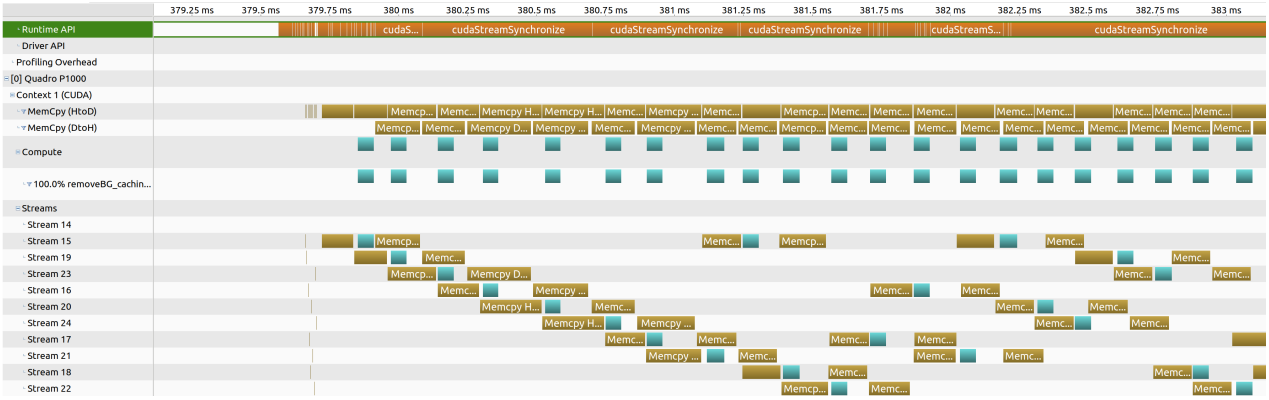
## 2.12 Improved Streamer Class

As it was shown in the Streamer class section the streamer class needs improvement. By allocating the device memory just once for the input and output image per stream and using already pinned memory on the CPU side it was possilbe to improve the class a lot!

Output readings:

| | |
|---|---|
| memCopy input image: | 118us |
| kernel: | 57us |
| memCopy output image: | 154us |

This results in full bandwidth usage of the PCI bus and a speed of: $C = \frac{1}{118+57+154} * 512 = 1.55$MAScans per second.




[2, CUDA Programming Guide, chapter 3.2.5.5ff]

## 2.13 Comparing different matrix multiplication kernels

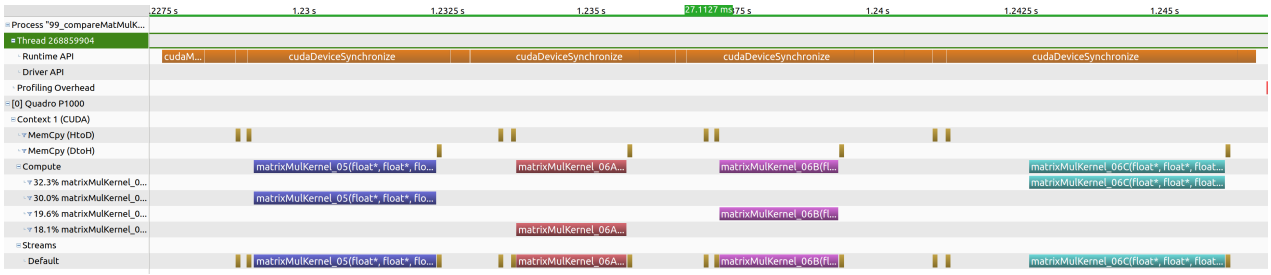
This snippet compares the total execution time of different matrix multiplication kernels. It provides also a class for easy measuring times in CPP.

The output of this snippet executed on a HP ZBook Studio G5 with a mobile Nvidia P1000 GPU is as followed:

Kernel Description:
- Kernel 05: Global Memory access kernel.
- Kernel 06A: Shared memory with bursts for both tiles.
- Kernel 06B: Shared memory with bursts for one tile.
- Kernel 06C Shared memory with no bursts.

```
Traditional Way: 1021.78ms. Value: 12812
Kernel 05: 3.53315ms. Value: 12812
Kernel 06A: 2.04691ms. Value: 12812
Kernel 06B: 2.72592ms. Value: 12812
Kernel 06C: 3.58989ms. Value: 12812 Slower because no burst transfers!"
```

But be aware compared to the nvidia-profiler output we see that this method is not very accurate for small kernels.




The readouts results from the tool are as followed:
- Kernel 05: 3.178ms
- Kernel 06A: 1.92ms
- Kernel 06B: 2.073ms
- Kernel 06C 3.415ms

Here we can definitely see that the using of shared memory in combination with burst transfers for global memory access is important to speed up GPU processing!

# References

[1] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors*. MPS Limited, Chennai, India: Katey Birtcher, 2017.

[2] NVIDIA-Corporation, "Cuda c programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

# CUDA Glossary

**boundary conditions** If the data isn't a multiple of the warp size there will be some pading threads that would access the wrong data. Boundary conditions prohibits that. .

**coalesces** Coalesces means that memory is sequencial order (N, N+1, N+2...).

**control flow** The control flow is the program execution path of control structures like loops and branches. .

**corner turning** Technique used to make efficient memory accesses. .

**device** Device always relates to the GPU that executes the kernels.. .

**DRAM bursts** DRAM bursts are multiple data transfers from DRAM grouped together. .

**host** Host always relates to the CPU that lauches the kernels. .

**interleaved data distribution** Storing data in separate groups in the memory to use differen banks when accesing the data. .

**row major** Row major means that the different rows of an matrix is stored sequencial in a 1d array..

**Thread Divergence** Threads of a warp may diverge when they follow different control flows. .

**warp padding** If the data isn't a multiple of the warp size the last warp will be filled regardlessly with threads. That's called padding of warps. .

# CUDA Acronyms

**CPU** Central Processing Unit.

**CUDA** Compute Unified Device Architecture.

**DRAM** Dynamic Random Access Memory.

**FPU** Floating Point Unit.

**GDDR SDRAM** Graphical Double Data Rate - Synchronous Dynamic Random Access Memory.

**GPU** Graphical Processing Unit.

**PU** Processing Unit.

**SIMD** Single Instruction Multiple Data.

**SM** Streaming Multiprocessor.