

GPU programming with CUDA

MAP55616 - GPU programming with CUDA

Revision: 0.2025.02.19

J. Refojo D. Golden

Trinity Centre for High Performance Computing

Course Outline

- 1 Introduction to CUDA
- 2 Basic CUDA programming
- 3 Advanced CUDA programming
- 4 Advanced CUDA technology
- 5 Advanced CUDA programming, revisited
- 6 Introduction to OpenCL
- 7 Introduction to OpenMP for GPUs and OpenACC

Course Objectives

- Give an overview of GPU architecture and technology
- Learn when choosing the GPGPU approach is and isn't, a good idea
- How to add cuda code to your current code
- Functions that run in different parts of the hardware
- Memory stored in different parts of the hardware
- Profiling and optimization techniques for CUDA code
- (Time permitting) GPU optimization with non CUDA libraries such as OpenCL, OpenMP and OpenACC

Introduction to CUDA

First chapter: An introduction to CUDA

In this chapter, we are going to go through a series of theoretical concepts, that we will be using later on.

This chapter is meant, mostly, to be kept for use as a reference later.

GPGPU

Previous attempts to introduce special hardware to offload computation from the CPU (like, for example, FPGAs) in the high performance computing market failed because people didn't get out of their way to buy them and code for them.

GPGPU uses the reverse approach - to use hardware that people already have available to provide parallel computing.

Obviously, no approach is perfect - it's hard to compete with general purpose hardware against specific purpose hardware.

CUDA

Initially, the process of writing the software to perform calculations in a graphics card was complex and slow.

In 2005 it would take about a year to learn how to write code for it to run on a GPU.

With the introduction of the new GPGPU libraries - Nvidia's CUDA, AMD/ATI's Firestream and OpenCL - the process was greatly simplified - by using code that is similar to C, now it only takes a few weeks for a programmer who has experience with C to become a GPU programmer.

In this course, we are going to focus on CUDA, Nvidia's library, which has become the standard in the academic environment.

Firestream, HIP and ROCm

Firestream was AMD/ATI's GPGPU Library.

Even after this library achieved some early targets sooner than CUDA (such as releasing the folding at home binaries), it became less popular afterwards, and now it is almost unused (and unheard of).

Was supposed to be replaced by the Mantle graphics API, which has been abandoned ever since - it was taken over by AMD's HIP, and then by ROCm.... they all seem to have suffered the same fate, although ROCm (a real underdog!) seems to have picked up (a lot!) with the AMD based machines that were launched over the last few years.

OpenCL

OpenCL is a GPGPU library, developed by the Khronos group (same group that develops OpenGL, the 3D graphics library).

Its main advantage is that it runs in almost any brand of GPUs (Nvidia, AMD, Intel and last year's MTTS60 and MTS2000) and most modern CPUs too.

The reason why it is not as popular (at least in academia) is due to the fact that its libraries are usually not as good - since it has to be restricted to features that work in both architectures, so by using it, you don't usually get to use all the features of the latest hardware.

Its performance is usually better on AMD/ATI cards, as NVIDIA spends more time fine tuning the performance for CUDA instead.

OpenCL

The main advantage of CUDA over OpenCL is that there are more and better mathematical libraries (cuFFT, cuBLAS, cuSPARSE, cuRAND, etc).

Some people argue that integer performance is better on AMD cards, but that still depends a lot on latest version of software, drivers, etc. It is true for double precision, but usually done with ROCm.

Some applications (for example bitcoin mining) used AMD GPUs instead of NVIDIA ones because of all of this, but even AMD cards stopped being cost effective anymore for this a long time ago.

The Vulkan graphical library (which is supposed to be taking over OpenGL) is supposed to be merging OpenCL into it, so any modern card is required to have a quite modern OpenCL



Compute capability

The compute capability, which is the hardware version number of a card, is given by a pair of a major and minor revision numbers. We will refer to it as "cc" sometimes.

The mayor revision number is 1 for the Tesla (also known as T10) architecture, 2 for the Fermi, 3 for the Kepler, 5 for Maxwell, 6 for Pascal, 7 for Volta and Turing, 8 for Ampere, 8.9 for Ada Lovelace and 9 for Hopper. Blackwell, which launched on the last week of January of 2025 is cc 10.

The minor revision number increases as more features are implemented over time on each of the architectures.

In this course, we are going to start covering the features of the original CUDA architectures, and then we will move on to the features of the later ones.

NVIDIA roadmap - old - ish

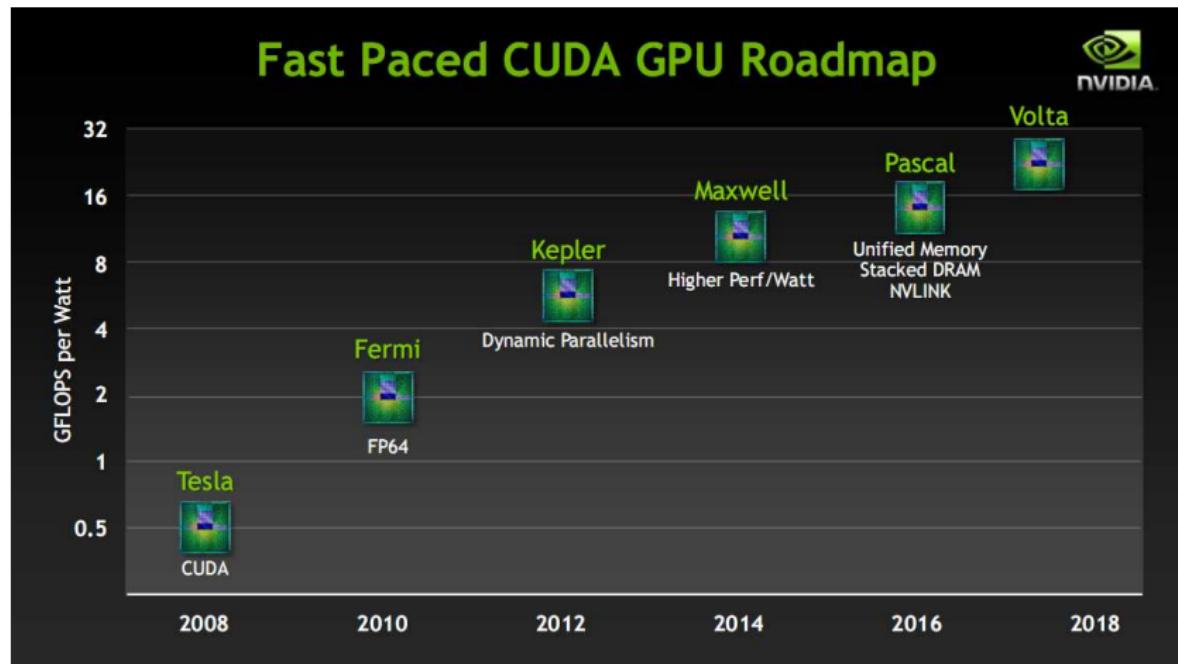


Image blatantly stolen from Bright Side of The News (tm)

NVIDIA roadmap - current - ish

MULTIPLE CHIPS. YEARLY LEAPS. ONE ARCHITECTURE.

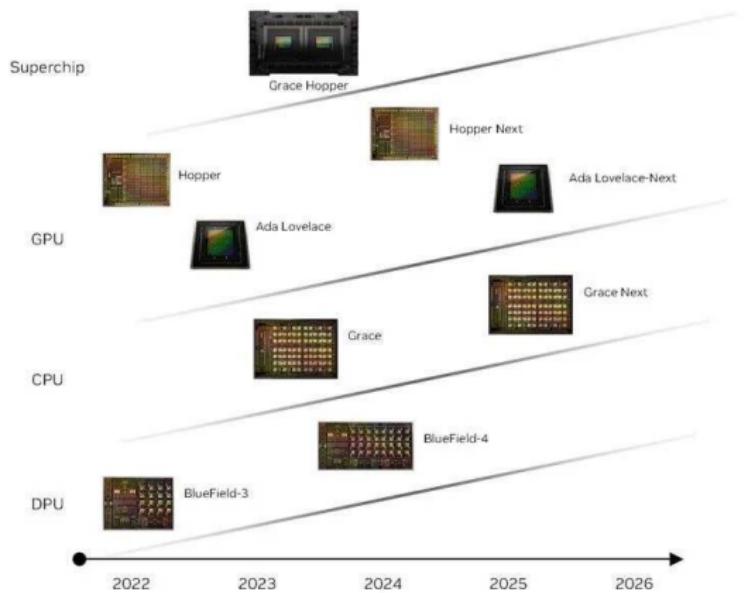


Image blatantly stolen from Nvidia (tm)

Types of Nvidia cards

We also have to differentiate between three types of cards that we can use CUDA on.

The **GeForce or RTX** cards (now thousands with tens, like RTX 4080) are mainly designed for gaming, but are mass-produced (thus cheaper) and are always the first ones to come out for each architecture.

The **Quadro** cards (now strict thousands preceded by a single letter, like A6000 or something like RTX6000) are designed and certified for professional level OpenGL rendering, are more expensive and sometimes give better double precision CUDA performance.

The **Tesla** (not to be confused with the T10 ones in cc 1.0) (now three digit model, like A100) ones are cards that lack video output connections - also, they usually have more on-board memory and are also more expensive.

Threads

Threads are the most basic unit of cuda parallel computation and represent one single software process (not hardware!) in a massively parallel processor.

Each thread is computed -in the hardware- by an individual CUDA core (which were previously called scalar processors) which belongs to a single multiprocessor (also called Streaming Multiprocessors).

Streaming Multiprocessors contain 8+ CUDA cores (which used to be called scalar processors).

There is a maximum number of threads per block, which depends on the particular graphic card architecture, and those threads can be distributed in a 1d, 2d or 3d manner.

Multi processor nomenclature

Wait, let us say that again:

What in some -not so old books- and documentation was called just a multiprocessor is now called Streaming Multiprocessor, or SM. They compute groups of threads.

Each one of those multiprocessors has a number of CUDA cores, which compute the individual threads.

The number of CUDA cores per Stream Multiprocessor varies with the architecture. Used to be 8 in T10, 32 (or 48 for a couple of models) for Fermi, 192 for Kepler, 128 for Maxwell and either 64 or 128 for Pascal (we'll see later why the number got reduced instead of increased), then 64 in Volta and Ampere and 128 in Ada Lovelace, Hopper and Blackwell.

Warps

The multiprocessors create, manage, schedule and execute threads in groups of 32 called warps.

That means that a multiprocessor always sets up and fires 32 parallel threads at any given time, even if we need fewer - which is why **ideally** you should have a number of threads (per block) which is a multiple of 32.

A half warp is either the first half or the second half of a warp.

The main reason for the threads being organized this way is that all the active threads in a warp get the same instructions at the same time (yes: SIMD) - and partially that they can share access to cached memory very close to them.

Warps

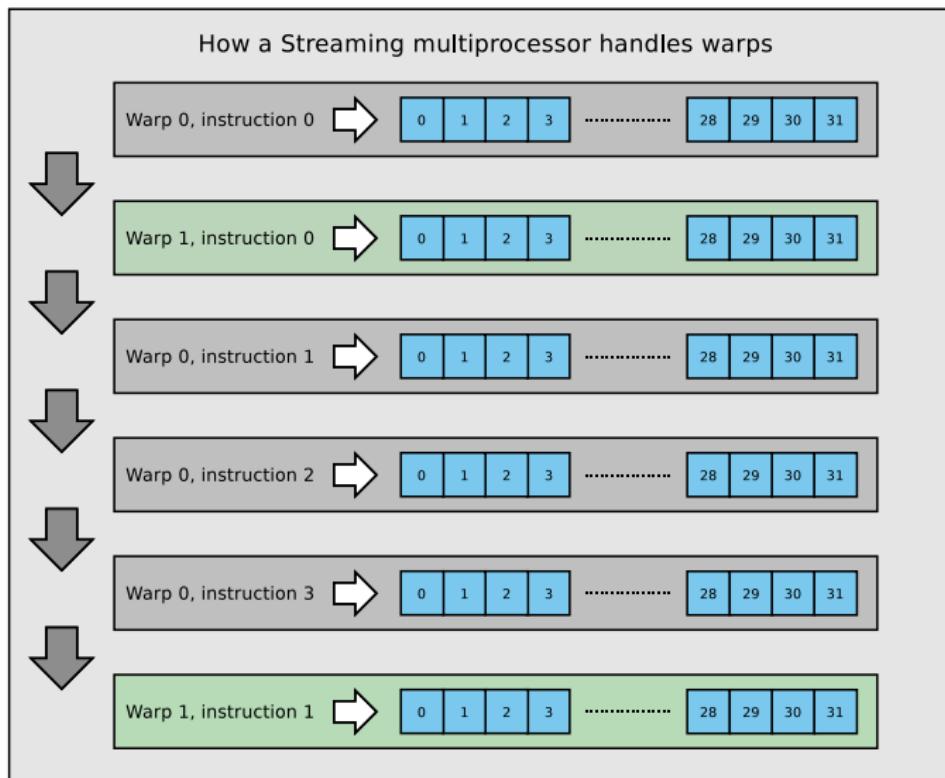
SMs execute the same instruction on all threads in a Warp at a given time - in the physical hardware.

When the next instruction has all the operands ready, the warp is eligible for execution.

Which means that we can optimize our code by making all the threads in a warp execute instructions that are as similar as possible (more about this later).

Also, if you need to use the warp size for any computation, make sure that you use the variable `warpSize` and not just 32 to future-proof your code!

Warps



Warps

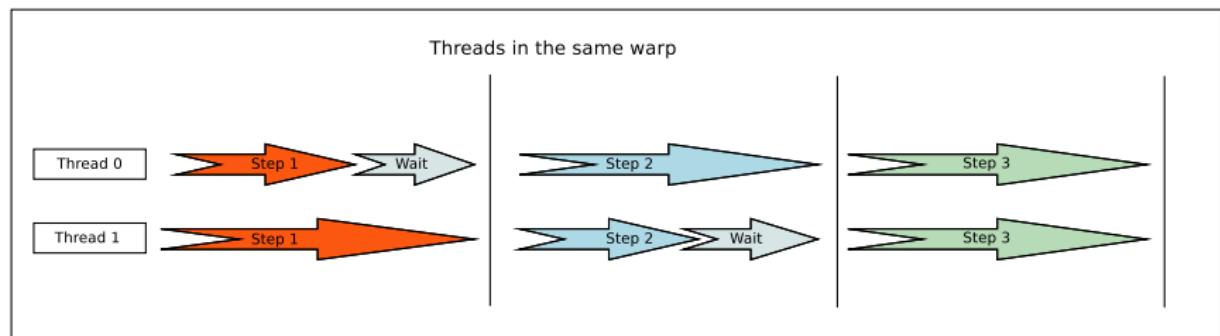
Word of warning - do not expect the warps to be running in a given order!

There can even be changes from one run to another about the order of the instructions - for example, when grabbing data that is allocated in the global memory, it might be closer or farther away from a particular multiprocessor.

Partitioning, however, is always the same - so, for example, in a matrix, element (0,0) will be run in thread 0 of block 0, but it might run in a different SMP.

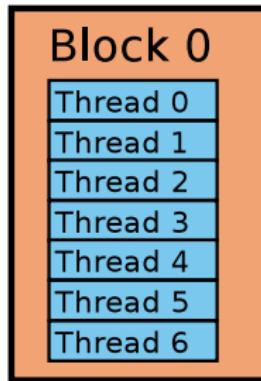
Warps

Another way to look at warps - threads in the same warp synchronize after each instruction:



Blocks

The basic building blocks of parallelisation on the GPU, they are composed of a specific number of threads that we can state, up to a limit of 1024.



There is a maximum number of blocks that we can request (usually up to millions), which depends on the graphic card, and, like the threads they can be organized again in 1D, 2D or 3D.

Blocks

A block is always executed in one single multiprocessor; and several blocks can and usually do reside concurrently on a single multiprocessor, but a block cannot be split between different MPs.

Therefore, it is usually (and this might not be true for projects with very unusual data dependencies) a good idea to use a multiple of 32 as the number of threads per block (as mentioned before, due to warps); and to have a number of blocks that is at least the number of multiprocessors in the card.

When the threads of a block terminate, new blocks are launched in the now idle multiprocessors - this dynamic multiprocessor allocation improves load balance (one simple example would be working with a triangular matrix).

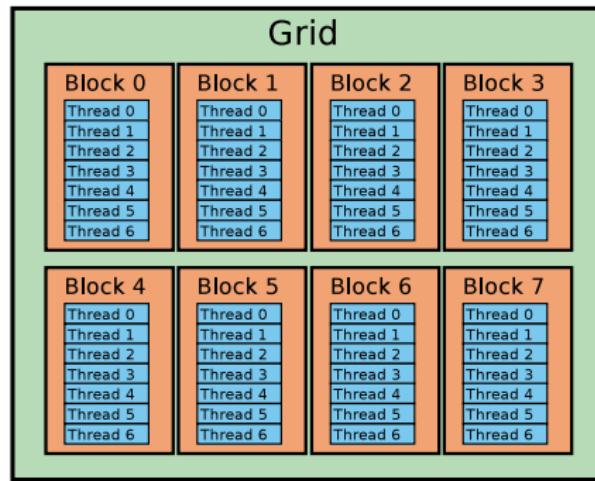
Blocks

The threads of a block mostly collaborate with each other in the following ways:

- By synchronizing, using barrier calls similar-ish (careful, this isn't a global sync!!) to `MPI_BARRIER(__syncthreads()).`
- By sharing data using an extremely fast memory, called the shared memory (for that very same reason: it is shared between threads of the same block... but not by the threads of different blocks!).

Other than that, they cannot interact with each other (so there is no direct equivalent to `MPI_SEND`, even inside a block). An exception to this has been implemented recently: it's called Cooperative Groups.

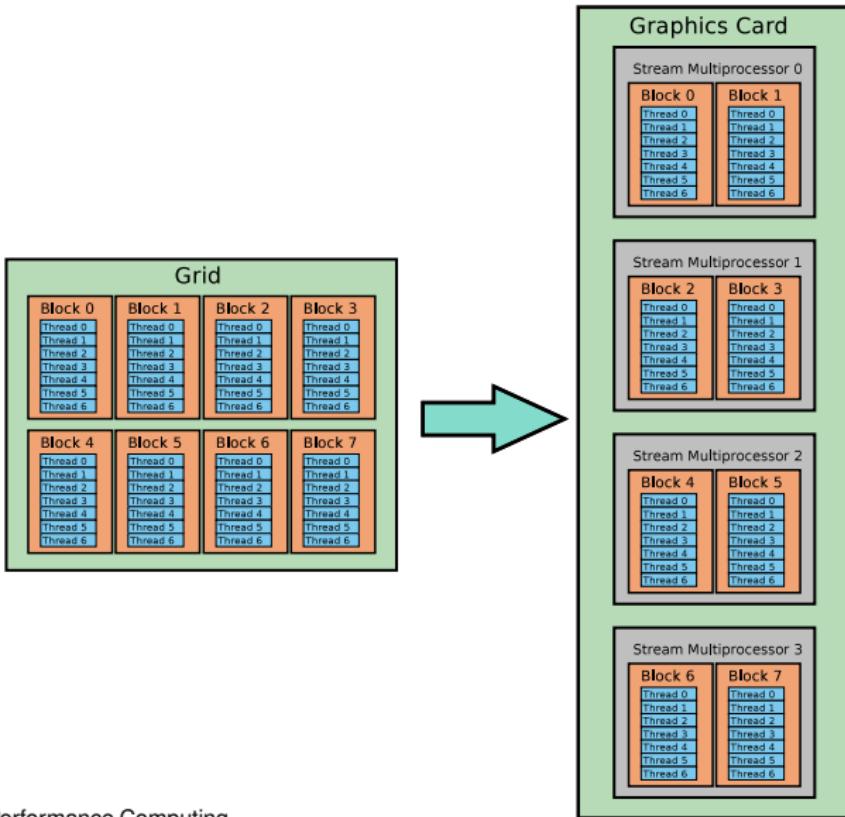
Grids



A grid is composed of blocks, which in turn are composed of threads, and runs in a single GPU.

Again, it is a good idea to have a number of blocks that is a multiple of the number of multiprocessors in the card.

Mapping to the hardware



More weird names!

Wait, so what are GPCs and TPCs, then? It turns out that Streaming Multiprocessors are also grouped into texture processing clusters (TPCs) and TPCs into GPU processing clusters (GPCs).

For example, in the Hopper architecture, we got:

The GH100 GPU had 8 GPCs, 72 TPCs (9 TPCs/GPC), 2 SMs/TPC, 144 SMs per full GPU

The H100 GPU with SXM5 had 8 GPCs, 66 TPCs (8.25 TPCs/GPC?!), 2 SMs/TPC, 132 SMs per GPU

So we got 2 SMs per TPC, but the number of TPCs per GPC could be depend not only on the architecture or model, but on a given GPC in a GPU!

This is most likely due to yields - if one TPC has manufacturing defects, it can be disabled and still sold.

More weird names!

Looking at Blackwell and assuming that we still get 2 SMs per TPC:

If the RTX 5090 Titan? (GB202) has 24,576 cuda cores: 192 SMs, 96 TPCs and 12 GPCs (8 TPC/GPC or 16 SMs/GPC)

If the RTX 5090 (GB202) has 21,760 cuda cores: 170 SMs, 85 TPCs and maybe 17 GPCs (5 TPCs per GPC)??

The RTX 5080 (GB203) has 10,752 cuda cores: 84 SMs, 42TPCs and 7 GPCs (6 TPCs per GPC?)

The RTX 5070 Ti (GB203) has 8,960 cuda cores: 70 SMs, 35TPCs and maybe 7 GPCs (5 TPCs per GPC)?? Or maybe 5GPCs (7 TPCs per GPC)??

The RTX 5070 (GB205) has 6,400 cuda cores: 50 SMs, 25TPC and 5 GPCs (5 TPCs per GPC?)

GB206 has 4,608 cuda cores: 36 SMs, 3 GPCs (12 SMs/GPC, assuming 2 SMs/TPC, hence 6 TPCs)

GB207 has 2,560 cuda cores: 20 SMs, 2 GPCs (10 SMs/GPC, assuming 2 SMs/TPC, hence 4 TPCs)

Kernel

A kernel is a C function that is going to be run N times (where N is the total number of threads) in parallel on the GPU by different threads. They are defined using the `__global__` declaration - we'll look into this properly later.

When called, they look different from any other functions because they have to be called using an "execution configuration syntax" that looks like this (yep: it's C++!):

```
<<<Grid_dimension,Block_dimension>>>
```

In devices with CUDA capability 1.X we could only run one kernel at a time; in devices with 2.X+ it is possible to run more than one - there is no term so far for a group of grids, but there might in the future.

CUDA C

CUDA C is just the extension of C so we can use cuda in C and C++ programs.

CUDA Runtime API is the high level cuda language (high here means more abstraction), and it is the one that we are going to use in the examples, as it involves less code (but it isn't as safe).

In this course, we are going to focus on CUDA Runtime API, as the code is slightly simpler and therefore better for beginners.

CUDA Driver API

CUDA Driver API is the low level cuda language, and we are not going to use a lot of it in this course, it involves more code (and better error checking) but provides safer function calls.

There used to be more of a difference between the runtime and driver versions, since you had to write your code in one or the other, but since cuda 3.0, both can be mixed, so we are not going to make a strong distinction between one and the other in this course.

Different kinds of functions

There are two clearly differentiated parts on a CUDA program.

There is code that is run on the CPU and interacts with the GPU; that is called `host code`.

And there is code that is run on the GPU; that is called `device code`.

Both are totally different from each other, as the `device code` is expected to be run in each thread of the grid (in CUDA cores) (ie, it is run several times) while the `host code` is (usually) run only once (on the CPU). So it is not like OpenMP or MPI, in which the parallel and non-parallel code are very similar, are run in the same parts of the hardware, and memory is easily shared between processes.

Different kinds of functions

However, there are three (and not two) different kinds of functions in CUDA, and they are stated with qualifiers (which means that they have a `__typeOfThisFunction__` before them, identifying the particular type), depending on from which part of the hardware they are called from and on which part they are executed.

Note that the GPU cannot call CPU functions, there is a hierarchical structure that has to be kept (starting on the CPU and finishing on the GPU).

Host functions

Host functions are executed in the CPU (therefore are `host` code) and have to be called from the CPU as well - they are pretty much the usual functions that we are used to.

There are, however, some operations that we can make on the GPU with this kind of functions - such as allocating and copying memory to and from it and setting up the thread hierarchy (grid, blocks and threads).

They can be specified with the `__host__` qualifier - but, as these are our usual C/C++ functions, no qualifier means that the function is a host one

Global functions, aka kernels

Global functions are executed in the GPU but have to be called from the CPU - so they are one of the two kinds of function that comprise the device code.

They are the kernels that we mentioned before, the functions that we call to get our CUDA implementations running - so it could be said that they are our "gateway" into the GPU.

They are specified with `__global__` qualifier, and must have void return type.

As mentioned before, they need to contain the "execution configuration syntax":

```
<<<Grid_dimension, Block_dimension>>>
```

Device functions

Device functions are executed in the GPU and have to be called from the GPU as well - they are the other kind of function that can be as well device code.

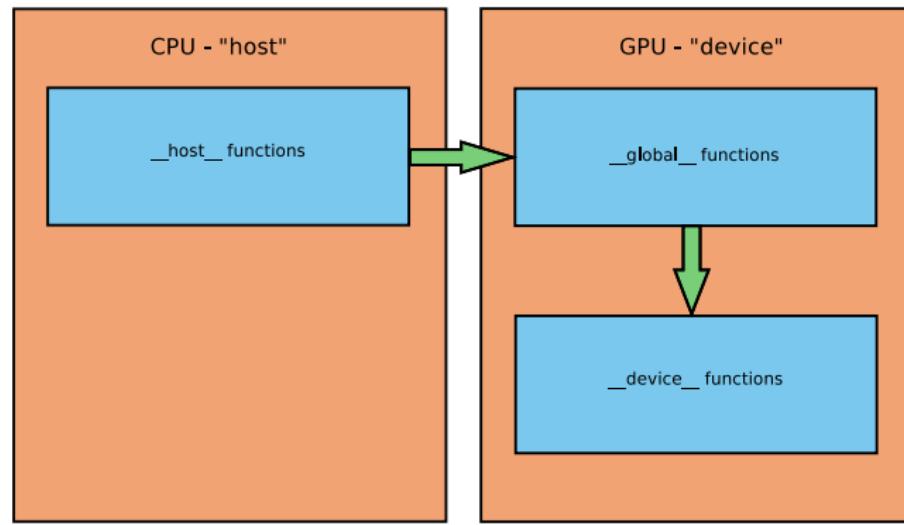
This is usually the ones in which we specify the per-thread instructions, and therefore the ones in which we have to be more careful.

They are specified with `__device__` qualifier.

The difference with the Global functions is that they look similar, but Global functions are executed by a grid, whereas device functions are executed by at least a single thread.

Considerations about the types of functions

Types of functions and where they are executed



Considerations about the types of functions

A function can have both the `__host__` and `__device__` qualifiers.

The rationale behind this is that they are some functions (like, for example, square root, fabs, etc) that we might want to call from either place.

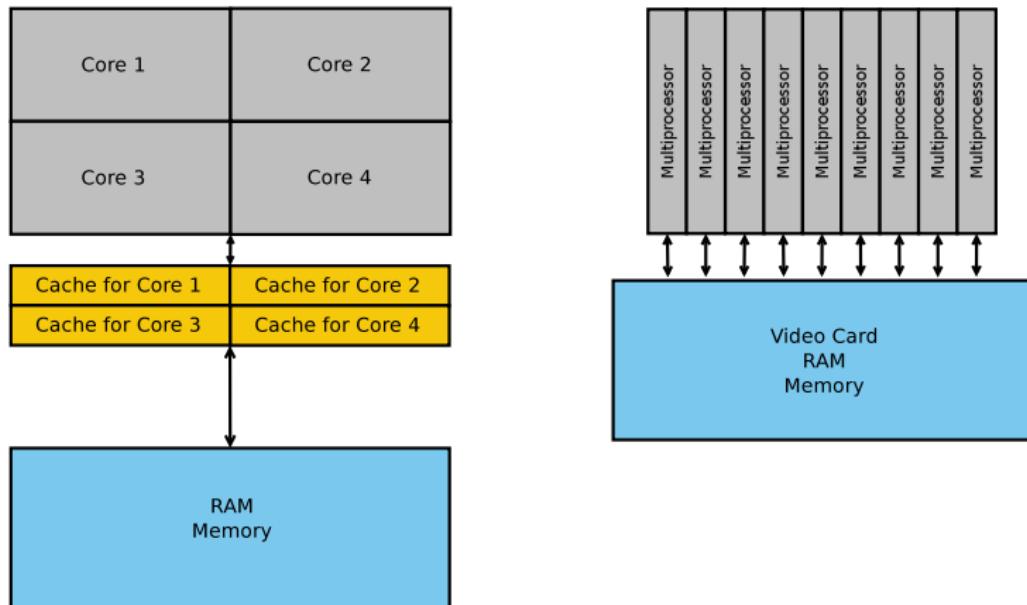
However, the combination of the `__host__` and `__global__` qualifiers is forbidden.

Considerations about the types of functions

- `__global__` functions do not support recursion... kind of.
- `__device__` functions support it starting on CUDA 2.X.
- `__global__` and `__device__` functions cannot declare static variables inside their body
- `__global__` function arguments can be passed to the device via shared memory (up to 256 bytes from cc 1.X+) or constant memory (up to 4KB on devices of cc 2.X+).
More about this memory types later.
- `__global__` function pointers can be used, but only in host code (ie not in device code).
- `__device__` function pointers can be used, but only in device code and from cc 2.X+.

Different kinds of memory

The memory on the graphics card does not work exactly in the same way as the CPU registers, cache and RAM memory.



Different kinds of memory

The CUDA cores do have registers, but there is no such thing as a cache memory that is equivalent to the caches in the CPU, that works automatically without us having to manage it (there is a L2 cache that does help move data around). The memory on the chip can be allocated in different ways, and its performance is going to be very different depending on the type that we are using.

The following slides are only an introduction to the memory types. We will revisit them in the following chapters, in which we will also show the code needed to use them.

Different kinds of memory - the bad news

We are going to give the bad news first - if normal parallelism (seen as MPI and OpenMP) involves a duplication of data between processes, in CUDA we will have at the very least an additional level of duplication - it is not uncommon to have the initial data reside on the host's RAM, the global memory, and also in another of the CUDA specific memory spaces (shared, texture, surface...).

Which means that we have to be specially careful when using data, due to potential lack of cache coherency - as we have to make sure that we are picking the most up to date one, and also that we go through all the necessary steps to bring the output all the way back to the host.

Per-thread registers

This memory is as close to the multiprocessors as possible (it's the literal registers during execution) - it is memory that becomes allocated, and used, during each thread's execution, and it can only be read by the thread that it is associated with.

As an example, if we initialize `'int i=0'` in a function that we are running for each thread in the GPU, that '`i`' will be located in the per-thread local memory.

Its main drawback is that if we use too much of it, it will overspill into the local memory - which is slower. And we do not get a warning about it.

Per-thread local memory

This memory is also accessible only by each individual thread, and is actually part of the global memory (that we will see in a moment), but isn't nearly as fast as the registers.

It is where some of the variables that we declare in our functions are stored if we run out of registers - so we have to be careful about not allocating too many per thread (or use other kinds of memory, if it is unavoidable).

Is is marked with a `__local__` declaration - and there are cases in which we might actually benefit by using it - when we have memory that we do not mind being slow (due to warp scheduling hiding the latency), and prefer to save the per-thread memory for some data that really needs to be fast.

Per-block shared memory

This memory is shared between the threads of a given block - it is really quick to access if the memory calls are done right, and it will be key in performance optimization of CUDA code. It is marked with a `__shared__` declaration.

Still, each chip has a limited amount of memory that it can allocate in this memory space, and it isn't a huge amount.

We will see later how to check the amount of shared memory available in a GPU - again, we have to be careful because it is scarce.

Another drawback that it has is that its lifetime is the lifetime of a kernel - we can't keep the data stored on it from one kernel to the next.

Device global memory

This memory resides in the DRAM or HBM of the video card - and can be read and be written to by any thread.

Unfortunately, that is the main disadvantage as well - it is the slowest kind of memory that we can use (it is still faster than non-cached CPU memory, though).

It is marked with a `__device__` declaration when it is statically allocated, but we won't usually use it that way.

Don't forget that "global" here can be a bit confusing - this memory is still in the GPU and can't be accessed directly from the CPU.

Constant memory

This memory is, as the global type, accessible by all the threads and blocks of a grid - it also resides in the GPU, but can be read from any thread.

Reading data from it is faster than from the global memory, but has the disadvantage of being very limited in size (You can expect to have only around 64KB, even in the most modern hardware!).

It is marked with a `__constant__` declaration.

It is cached into every Streaming Multiprocessor in the grid, which explains why it is so fast (and the memory limit).

Texture memory

Texture memory, like the global and constant types, is shared between all the threads and blocks of a grid - it also resides in the GPU, but can be read from any thread.

Reading data from it is also faster than from the global memory, and it is not even close to as limited in size as the constant, but it still is not as fast as the shared memory.

Texture memory

Since it works a lot like textures in computer graphics, there are more features that can be useful - such as really fast interpolation, possibility of wrapping around the boundaries (which means that if you exceed the borders of the array, you will ‘appear’ from the other side).

Is is not marked as other types, but declared like this:

```
texture<float, 2, cudaMemcpyElementType> tex;
```

It can be a bit confusing, as it looks similar to an array, but when you request data from it, you ask for positions by using floating point values and not integers.

Surface memory

We also have the option of using surface memory, which is a lot like the texture memory.

The main differences of surface memory over texture memory are:

- Surface memory is read and write, instead of read-only.
- Surface memory uses byte addressing - that means that we have to scale the floating point values that we use to recover data from the memory.

Memory types table

Memory types table

Type	Location	Cached	Speed
Registers	Dedicated hardware next to the multiprocessor	Yes	Single cycle
Shared memory	Dedicated hardware next to the multiprocessor	Yes	Single cycle
Constant memory	Dedicated hardware next to the multiprocessor	Yes	One to hundreds of cycles
Texture memory	DRAM or HBM of the graphics card	Yes	Hundreds of cycles
Surface memory	DRAM or HBM of the graphics card	Yes	Hundreds of cycles
Local memory	DRAM or HBM of the graphics card	Yes	Hundreds of cycles
Global memory	DRAM or HBM of the graphics card	No	Very slow

Simple and double precision

You also have to take into account that the results of floating point computations on the GPU will most certainly differ with the ones on the CPU.

There is a number of reasons for that - the fact that the compiler outputs and instruction sets are different, the way in which double precision arithmetic is processed in the GPU, etc.

Simple and double precision

GPUs were originally designed to compute simple precision (as double precision is not usually needed in real-time computer graphics), so it is not surprising that the earlier CUDA cards could only use single precision.

Also, you must take into account that the double precision performance can be lower in the Geforce cards than in Teslas or (sometimes) Quadros. For example, with the GeForce GTX 680 (a GK104 chip), you will get 1/24th of the performance when running in double precision mode.

Teslas and some Quadro cards are better at double precision, and you can expect 1/8th of the performance with them - this, however, means in raw compute - latency hiding due to warp scheduling might make real performance not as bad as that.

Some data about the hardware on the cards, 1 of 7

This is the technical CUDA data of a GeForce 9800 GTX, an old card (it was released the 1st of April of 2008) with CC 1.1:

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce 9800 GTX+  
GPU number of multi-processors: 16  
GPU CUDA cores per multi-processor: 8  
GPU total number of CUDA cores: 128  
GPU Compute capability: 1.1  
GPU Maximum size of each dimension of a grid: 65,535x65,535x1  
GPU Maximum size of each dimension of a block: 512x512x64  
GPU Maximum number of threads per block: 512  
GPU 32-bit registers available per SM: 8192  
GPU Shared memory available per block in bytes: 16384  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

Some data about the hardware on the cards, 2 of 7

And this is the technical data of a GeForce GTX680, which was released in March of 2012):

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce GTX 680  
GPU number of multi-processors: 8  
GPU CUDA cores per multi-processor: 192  
GPU total number of CUDA cores: 1536  
GPU Compute capability: 3.0  
GPU Maximum size of each dimension of a grid: 2,147,483,647x65,535x65,535  
GPU Maximum size of each dimension of a block: 1024x1024x64  
GPU Maximum number of threads per block: 1024  
GPU 32-bit registers available per SM: 65536  
GPU Shared memory available per block in bytes: 49152  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

This card's theoretical compute peak is 3.2 Teraflops.

Some data about the hardware on the cards, 3 of 7

And then with a Tesla K40c (released in November of 2013):

```
////////////////////////////////////////////////////////////////////////  
GPU model name: Tesla K40c  
GPU number of multi-processors: 15  
GPU CUDA cores per multi-processor: 192  
GPU total number of CUDA cores: 2880  
GPU Compute capability: 3.5  
GPU Maximum size of each dimension of a grid: 2,147,483,647x65,535x65,535  
GPU Maximum size of each dimension of a block: 1024x1024x64  
GPU Maximum number of threads per block: 1024  
GPU 32-bit registers available per SM: 65536  
GPU Shared memory available per block in bytes: 49152  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

This card's theoretical compute peak is 4.2 Teraflops (or up to 5 with Boost).

Some data about the hardware on the cards, 4 of 7

Comparing them:

- The number of multi-processors was reduced, from 16 to 8, then up again to 15. Latest cards (Ampere) has up to 108.
- The number of CUDA cores per multi-processors was increased from 8 to 192, a x24 increase - then, it dropped to 64 (Volta, Turing and Ampere).
- The total number of CUDA cores in the card increased from 128 to 1536, a x12 increase, then up to 2880 - latest (Ampere) has up to 6912.
- Maximum grid size goes from $65,535 \times 65,535 \times 1$ to $2,147,483,647 \times 65,535 \times 65,535$ (same in latest).
- Maximum block size goes from $512 \times 512 \times 64$ to $1024 \times 1024 \times 64$ (same in latest).

Some data about the hardware on the cards, 5 of 7

- Maximum number of threads per block duplicates from 512 to 1024 (...same in latest).
- Number of 32-bit registers available per block increases 8-fold (8192 to 65536) (...same in latest)
- The shared memory available per block increases from 16384 to 49152 bytes (x3), and then up again in Ampere, being configurable with up to 164 KB (we'll see later in the module how to configure this).
- The amount of constant memory and the warp size remains the same, though (so far!)

It was a very significant difference for 4 years (2008 to 2012), but has plateau'ed a little bit since.

Some data about the hardware on the cards, 6 of 7

Let's look at more modern cards, like the GeForce RTX 2080 SUPER (released in July of 2019):

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce RTX 2080 SUPER  
GPU number of multi-processors: 48  
GPU CUDA cores per multi-processor: 64  
GPU total number of CUDA cores: 3702  
GPU Compute capability: 7.5  
GPU Maximum size of each dimension of a grid: 2,147,483,647x65,535x65,535  
GPU Maximum size of each dimension of a block: 1024x1024x64  
GPU Maximum number of threads per block: 1024  
GPU 32-bit registers available per SM: 65536  
GPU Shared memory available per block in bytes: 49152  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

This card's theoretical compute peak is 10.1 Teraflops in single precision.

Some data about the hardware on the cards, 7 of 7

And finally with a GeForce RTX 3090 (released in September of 2020):

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce RTX 3090  
GPU number of multi-processors: 82  
GPU CUDA cores per multi-processor: 128  
GPU total number of CUDA cores: 10496  
GPU Compute capability: 8.6  
GPU Maximum size of each dimension of a grid: 2147483647x65535x65535  
GPU Maximum size of each dimension of a block: 1024x1024x64  
GPU Maximum number of threads per block: 1024  
GPU 32-bit registers available per SM: 65536  
GPU Shared memory available per block in bytes: 49152  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

This card's theoretical compute peak is 35.58 Teraflops in single precision.

How does Blackwell compare?

Looking at the wikipedia entry for a GeForce RTX 5090
(released on the 30th of January of 2025):

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce RTX 5090  
GPU number of multi-processors: 170  
GPU CUDA cores per multi-processor: 128  
GPU total number of CUDA cores: 21,760  
GPU Compute capability: 10.1  
GPU Maximum size of each dimension of a grid: 2147483647x65535x65535?  
GPU Maximum size of each dimension of a block: 1024x1024x64?  
GPU Maximum number of threads per block: 1024?  
GPU 32-bit registers available per SM: 65536  
GPU Shared memory available per block in bytes: 65536?????  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

This card's theoretical compute peak is 104.8 TFLOPS
Teraflops in single precision (FP32)... but only 1.637 TFLOPS
in FP64!

Installing the CUDA SDK

Installing CUDA on a machine can be broken down into the following steps:

- Making sure that your OS and hardware supports CUDA (most modern machines and OSs will do)
- (Optional) Installing an up to date NVIDIA driver (There used to be different drivers for CUDA, but now they are all the same)
- Installing the CUDA toolkit (that will depend on the OS used). Usually, an up to date driver will be installed as part of this process.

Installing the CUDA SDK: Windows

This installation will depend a lot on which software do you use under windows - Eclipse / Visual Studio / mingw, and in this course we are going to focus on HPC development - under Linux.

Installing the CUDA SDK: Linux

Requirements for CUDA under Linux:

- Make sure that you are running a supported version of Linux with a compatible gcc compiler.
- Go to the download page (<https://developer.nvidia.com/cuda-downloads>), and download and install the CUDA toolkit (which will be downloaded as a "cuda_* .run") - this will provide the cuda tools, as, for example, the compiler, and the code samples as well. It also includes an up to date driver.

The “Linux Getting Started Guide” (available from the previous link) has full details.

Installing the CUDA SDK: Linux

Also: after you download the CUDA installers, you must give the file the right permissions (especially the execute permission)

Changing permissions

```
chmod 777 cuda_12.3.2_545.23.08_linux.run
```

After doing that, you can run the installers as root by typing the following command lines (you might have to turn off the X server to install the drivers, though):

Installing

```
sudo ./cuda_12.3.2_545.23.08_linux.run
```

Installing the CUDA SDK: Linux

Reminder under Linux: After installing the CUDA toolkit, do not forget to edit and add to your .bashrc (which is a hidden file in your home folder) the following lines!!

Addition to .bashrc

```
# CUDA set up
PATH=$PATH:/usr/local/cuda-12.3/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-12.3/lib64
export PATH
export LD_LIBRARY_PATH
```

And do not forget either that you will have to update or start a new shell window for those variables to load up. Also, with 64 bit linux, you have to add /usr/local/cuda-12.3/lib64 to LD_LIBRARY_PATH as well!

Installing the CUDA SDK: Linux

Another reminder under Linux: You might also need to add the following dependencies:

Additional CUDA SDK dependencies

```
freeglut3-dev libxi-dev libXmu-dev
```

for example, under Debian or Ubuntu linux, you might have to run

Getting the additional CUDA SDK dependencies

```
sudo apt-get install freeglut3-dev libxi-dev libXmu-dev
```

Installing the CUDA SDK: Linux

Yet *another* reminder under Linux 64 bits! If you get the following error:

Common installation error

```
"undefined reference to gluErrorString"
```

You can solve it by going to
NVIDIA_GPU_Computing_SDK/C/common/common.mk and
swapping the orders of the \$(RENDERCHECKGLLIB) and
\$(OPENGLLIB)

Installing the CUDA SDK: Linux

Yet another reminder under Linux: If you get the following error:

Common installation error

```
"cannot find -lcuda"
```

You can fix it by creating the following symbolic links:

Getting the additional CUDA SDK dependencies

```
sudo ln -s /usr/lib/nvidia-current/libcuda.so /usr/lib/libcuda.so  
sudo ln -s /usr/lib/nvidia-current/libcuda.so.1 /usr/lib/libcuda.so.1
```

Installing the CUDA SDK: Linux

Also, if your machine has Nvidia Optimus - that is, you have two graphics cards (usually a lower-performance intel card and a high performance Nvidia one) so you can swap between the two of them depending on the usage that you need, you will also have to install the `bumblebee` package and then update the Nvidia driver.

After that, you can run the CUDA software by using:

Running CUDA software on Optimus systems

```
$optirun my_cuda_executable.exec
```

We have tested this with Ubuntu 12.04, we haven't tried yet other distros.

How to access the CUDA machine in TCHPC

We also have a machine with a GeForce RTX 2080 Super and a Tesla K40, so users can test CUDA compute capabilities 3.5 and 7.5 - specs are as follows:

```
CPU: Intel i5 3570K  
GPU 0: GeForce RTX 2080 SUPER / 8GD6  
GPU 1: Tesla K40c / 12GD5  
RAM: Corsair 16GB 1600MHz PC3-12800  
Mob: ASUS P8Z77-M PRO
```

So let us know if you need an account - if you have already an account on Lonsdale, the same login should work on this machine.

How to access the CUDA machine in TCHPC

Again, do not forget to set up your .bashrc (which is a hidden file in your home folder) this way:

Addition to .bashrc

```
# CUDA set up
PATH=$PATH:/usr/local/cuda-12.3/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-12.3/lib64
export PATH
export LD_LIBRARY_PATH
```

You can then log in again or type "source .bashrc" to update your environment variables.

Mixing cuda and non-cuda code

Since the nvidia compiler acts as a C++ one (which can be counter-intuitive since CUDA code is usually much closer to C than to C++), adding cuda code to our current C/C++ programs might not be trivial. There are mainly 2 approaches to do this:

Option 1: Write and compile everything as C++ (so we compile the non-cuda code with g++ and use .cpp files)

Option 2: Write and compile everything as C - we will need to use the extern C construct where C files include function definitions.

We will also show some samples about how to build the makefiles

The C++ approach, part 1

First of all, we have to let the C/C++ code know which functions that are located in the .cu files we are going to use.

This is the first problem that we have, as in the header file, there might be cuda-only function qualifiers that only nvcc understands.

To solve this problem, we will define the functions, by using `extern`, as global functions in the C++ code, as in the following example:

```
// This is main.cpp
// we declare a non-cuda function that is on a .cu file ,
// and that will call the cuda code
// we can also use inline to reduce the overhead
extern inline int add_vectors(void);

int main () {
    // Then we can use it normally
    add_vectors();
}
```

The C++ approach, part 2

And then, in the .cu file, we define the function that we want to use as "extern":

```
// This is add_vectors.cu
extern int add_vectors() {
    /* pointers to host memory */
    float *a, *b, *c;
    ....
}
```

The C++ approach, part 3

Then we will continue by defining and using different compilers
- so the cuda code gets compiled with nvcc and the rest with
g++.

```
CC=      g++
CXX=     g++
NVCC=    nvcc
```

In this case, we use g++ to compile the .c and .cpp files (as usual, and remember), but nvcc to compile the .cu files like this:

```
add_vectors.o: add_vectors.cu
$(NVCC)  add_vectors.cu -c $(NVCCFLAGS) -I$(INCPATH)
```

And finally, we link the executable with nvcc:

```
main: main.o add_vectors.o
$(NVCC) $(OBJECTS) -o add_vectors -I$(INCPATH)
```

The C approach, part 1

With this approach, we have to declare the functions as global again:

```
// This is main.c
extern inline int add_vectors(void);

int main () {
    // Then we can use it normally
    add_vectors();
}
```

and:

```
// This is add_vectors.cu
extern int add_vectors() {
    /* pointers to host memory */
    float *a, *b, *c;
    ....
}
```

The C approach, part 2

The difference in this case is that we have to tell the C++ compiler to make the functions in the .cu file that we are going to call look as if they were C and not C++.

We can do that by declaring them like this (either in the header file or in the .cu):

```
#ifdef __cplusplus
extern "C" {
    int add_vectors();
}
#endif
```

The C approach, part 3

We will again define different compilers - so the cuda code gets compiled with nvcc and the C code with gcc.

```
CC=      gcc  
NVCC=    nvcc
```

In this case, we use gcc to compile the .c (again, as usual), but nvcc to compile the .cu files like this:

```
add_vectors.o: add_vectors.cu  
$(NVCC)  add_vectors.cu -c $(NVCCFLAGS) -I$(INCPATH)
```

And again, we link the executable with nvcc:

```
main: main.o add_vectors.o  
$(NVCC) $(OBJECTS) -o add_vectors -I$(INCPATH)
```

Mixing CUDA code with OpenMP code

If you have OpenMP code mixed with cuda code in your .cu files, you will have to add the following flags in your Makefile:

```
NVCC_PARALLEL = -Xcompiler -fopenmp  
# amongst other options  
NVCCFLAGS = -O4 --use_fast_math -lm $(NVCC_PARALLEL)  
  
#to link all the files into the executable  
main: $(OBJECTS)  
    $(NVCC) $(OBJECTS) -o $(TARGET) -I$(INCPATH) $(NVCCFLAGS)  
# to compile each one of the .cu files  
myCudaCode.o: myCudaCode.cu  
    $(NVCC) -c mycode.cu $(NVCCFLAGS) -I$(INCPATH)
```

Combining OpenMP and CUDA can be extremely useful when you want your program to run code on the GPU but still do something on the CPU in the meanwhile.

Mixing CUDA code with MPI code

If you have MPI code mixed with cuda code in your .cu files, you will have to add the following to your Makefile:

```
CC = mpicc
# amongst other options
CFLAGS = -Wall -O2 -ffast-math
LDFLAGS = -lm -ffast-math

MPICC      = nvcc -Xptxas -v
#MPI_INCLUDES      = /usr/mpi/intel/mvapich2-1.2p1/include
MPI_INCLUDES      = /usr/include/openmpi
#MPI_LIBS      = /usr/mpi/intel/mvapich2-1.2p1/lib
MPI_LIBS      = /usr/lib/openmpi

# default target
cudaMpHelloWorld.exec : cudaMpHelloWorld.o
    $(MPICC) -L$(MPI_LIBS) -lmpi -o $@ *.o

%.o : %.cu
    $(MPICC) -I$(MPI_INCLUDES) -o $@ -c $<
```

If you are using MVAPICH instead of openmpi, just use the commented lines

Basic CUDA programming

Second chapter: Basic CUDA programming

We finally get to see some CUDA code!

We will start by comparing some serial code with some CUDA code, one step at a time.

Keep the chapter one notes at hand, and let's have a look at how CUDA looks like.

Add vectors program in Serial

```
void addArrays (float *in1, float *in2, float *out, int numberOfElements) {
    int i;
    for (i=0;i<numberOfElements;i++) {
        out[i]=in1[i]+in2[i];
    }
}
int main(void) {
    int i,N = 18;
    float *a, *b, *c;
    /* Allocate arrays a, b and c */
    a = (float*) malloc(N*sizeof(float));
    b = (float*) malloc(N*sizeof(float));
    c = (float*) malloc(N*sizeof(float));
    /* Initialize arrays a and b */
    for (i=0; i<N; i++) {
        a[i]= (float) i;
        b[i]= (float) 2*i;
    }
    /* Compute the addition */
    addArrays(a,b,c,N);
    for (i=0;i<N;i++) {
        printf("a[%d]=%f b[%d]=%f c[%d]=%f\n",i,a[i],i,b[i],i,c[i]);
    }
    free(a); free(b); free(c);
    return 0;
}
```

Add vectors program in Cuda, AddVectors.cu 1 of 8

First we need to state the pointers to allocate the memory both in the RAM and the GPU

```
// pointers to host memory
float *a_cpu, *b_cpu, *c_cpu;
// pointers to device memory
float *a_gpu, *b_gpu, *c_gpu;
```

Something that we have to take into account is that the declaration looks exactly the same for the memory that are allocating in both places - which is why it is important that we mark the variables that will store pointers to memory in the gpu with a "*_gpu*" after their names.

Add vectors program in Cuda, AddVectors.cu 2 of 8

Then, we allocate the memory

```
// Allocate arrays a_cpu, b_cpu and c_cpu on host
a_cpu = (float*) malloc(N*sizeof(float));
b_cpu = (float*) malloc(N*sizeof(float));
c_cpu = (float*) malloc(N*sizeof(float));

// Allocate arrays a_gpu, b_gpu and c_gpu on device
cudaMalloc ((void **) &a_gpu, N*sizeof(float));
cudaMalloc ((void **) &b_gpu, N*sizeof(float));
cudaMalloc ((void **) &c_gpu, N*sizeof(float));
```

Here where we spot the first difference: the function calls that we use to allocate the memory are different. It is also worth noting that by calling those functions we are allocating the three vectors a_gpu, b_gpu and c_gpu in the global memory of the graphics card.

We need to start being careful - the pointers a_cpu,b_cpu and c_cpu contain addresses on the main memory; but a_gpu, b_gpu and c_gpu store addresses on the video card memory. Which means that if we try to access pointers to the GPU memory from code that is running in the CPU, we will most likely get a segmentation fault!

Add vectors program in Cuda, AddVectors.cu 3 of 8

Then, we initialize the vectors a and b in the host. The vectors in the GPU cannot be initialized directly from the CPU, so what we do is:

```
// Initialize arrays a and b
for (i=0; i<N; i++) {
    a_cpu[i] = (float) i;
    b_cpu[i] = (float)2*i;
}
```

and then copy the data into the vectors in the GPU:

```
// Copy data from host memory to device memory
cudaMemcpy(a_gpu, a_cpu, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(b_gpu, b_cpu, N*sizeof(float), cudaMemcpyHostToDevice);
```

At this point, we have exactly the same kind of data in a_cpu and a_gpu, and in b_cpu and b_gpu, the only difference is where the data is located.

Also, the arguments in cudaMemcpy are not ordered like in MPI_Send: they are output, input, size, direction.

Add vectors program in Cuda, AddVectors.cu 4 of 8

Now we have to prepare the grid to run in the GPU - that means preparing the number of blocks in the grid and the number of threads per block:

```
// Compute the execution configuration  
int block_size=8;  
dim3 dimBlock(block_size);
```

This means that we are going to use 1-dimensional blocks with 8 threads per block. We will see later on examples of 2d and 3d block setups, but let's stick to 1d for now, as it is more natural for the addition of two 1d vectors.

```
// Compute the execution configuration  
dim3 dimGrid ( (N/dimBlock.x) + (! (N % dimBlock.x)?0:1) );
```

This part sets up the grid - we are going to have a grid of $N / \text{dimblock}$ blocks, or one more if we need to (and if we do, there will be idle threads on that block).

Add vectors program in Cuda, AddVectors.cu 5 of 8

```
// Add arrays a and b, store result in c  
add_arrays_gpu<<<dimGrid, dimBlock>>>(a_gpu, b_gpu, c_gpu, N);
```

This is a call to the kernel (a `__global__` function) - that means that that particular function is going to be ran in each thread of each block, and the difference between two instances running are the ids of the threads that are running it. We'll see more about this when we look at the kernel code.

```
// Copy data from device memory to host memory  
cudaMemcpy(c_cpu, c_gpu, N * sizeof(float), cudaMemcpyDeviceToHost);
```

When you are copying back from the GPU, don't forget to swap the inputs and outputs (so the outputs go first) and the `cudaMemcpyHostToDevice` to `cudaMemcpyDeviceToHost`!

Yet again, we cannot access directly from the memory that is on the GPU, so we have to copy back the memory from the GPU to the RAM before we can print it or check it

Add vectors program in Cuda, AddVectors.cu 6 of 8

```
// Print c
for (i=0; i<N; i++)
    printf(" c[%d]=%f\n",i ,c_cpu[ i ]);
```

Now, we can safely print c, as it is in the RAM.

```
// Free the memory
free(a_cpu); free(b_cpu); free(c_cpu);
cudaFree(a_gpu); cudaFree(b_gpu);cudaFree(c_gpu);
```

Here we can see the two different (albeit similar) ways in which we can free the memory depending on where it is located. Fortunately, it is almost the same instruction, the only difference is the function call that we make.

Add vectors program in Cuda, AddVectors.cu 7 of 8

Again - be careful here! Take into account that to allocate on the GPU you used:

```
// Allocate the memory in the device  
cudaMalloc ((void **) &a_gpu, sizeof(float)*N);
```

But to free the allocated memory, you used:

```
// Free the memory in the device  
cudaFree(a_gpu);
```

So in the first case, you have to pass a pointer to the array, but in the second, the array itself.

Add vectors program in Cuda, AddVectors.cu 8 of 8

```
// The Kernel
__global__ void add_arrays_gpu( float *in1, float *in2, float *out, int Ntot) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    if ( idx <Ntot )
        out[idx]=in1[idx]+in2[idx];
}
```

It is worth noting that this function is similar to the serial case, the main difference being that there is no for loop in it - each element is calculated on a different thread.

As mentioned before, this function is going to be run on each thread of each block of the grid - it is, therefore, important that we make sure that we do not replicate processes. When inside a kernel, the blockIdx and threadIdx structures will give us the position of the thread in the grid - and it is also quite important that we have a control condition (` if (idx <Ntot) `) to make sure that, if we have more threads than elements in the arrays, we are not trying to read or write unallocated positions of memory.

Restricted words in CUDA C

CUDA C has a number of words and expressions that you use to call CUDA functions, allocate the grids, set the number of blocks and threads and get information from the card.

Quite obviously, when using the library, those words are restricted - which means that you cannot use them as names for your own functions, variables and structures.

Good examples are the function and variable qualifiers (like `__global__`, `__device__`, `__constant__`), but also vector types `dim3`, `char2`, `short3`, `int3`, `long4` and other structures (like `cudaDeviceProp` to get information about the GPU).

Of particular interest for us might be structures like `float2` and `float3` for geometrical algorithms.

Restricted words in CUDA C

There are some special variables, that are accessible from every thread, and that we are going to need as well. They are:

- `gridDim` that gives us the grid size(s), or number of blocks in the grid
- `blockIdx` that gives us the block index (or indices)
- `blockDim` that gives us the block size(s), or number of threads in a block
- `threadIdx` that gives us the thread index *inside the block*

And this is how we use them:

```
/* Global id of the thread */
int idx=blockIdx.x*blockDim.x+threadIdx.x;
```

Detecting and choosing CUDA cards, findDevices.cu

1
of 4

In this part, we are going to see how to list and request information about the different graphic cards that are available in the current system. We will show some code that does that and chooses the "best" card (understood as the one with the largest number of CUDA cores... which doesn't necessarily mean that it is actually the best).

Detecting and choosing CUDA cards, findDevices.cu 2 of 4

```
cudaGetDeviceCount(&n);
printf("Found %d CUDA-enabled devices\n",n);
```

This function call (cudaGetDeviceCount) allows us to request the number of CUDA compatible graphic cards that are available in the system.

```
struct cudaDeviceProp x;
cudaGetDeviceProperties(&x, i);
```

This function call (cudaGetDeviceProperties) allows us to request the properties of the GPU that has the id "i" (the id is an integer value).

Detecting and choosing CUDA cards, findDevices.cu

3 of 4

Here is a list of different members of the structure that we can check:

```
GPU model name: x.name
GPU number of multi-processors: x.multiProcessorCount
GPU Compute capability: x.major,x.minor
GPU Clock frequency in kilohertz: x.clockRate
GPU Device can concurrently copy memory and execute a kernel x.deviceOverlap
GPU Maximum size of each dimension of a grid: x.maxGridSize[]
GPU Maximum size of each dimension of a block: x.maxThreadsDim[]
GPU Maximum number of threads per block: x.maxThreadsPerBlock
GPU Maximum pitch in bytes allowed by memory copies: x.memPitch
GPU 32-bit registers available per block x.regsPerBlock
GPU Shared memory available per block in bytes: x.sharedMemPerBlock
GPU Alignment requirement for textures: x.textureAlignment
GPU Constant memory available on device in bytes: x.totalConstMem
GPU Warp size in threads: x.warpSize
```

This function call (`cudaGetDeviceCount`) allows us to request the number of CUDA compatible graphic cards that are available in the system.

Detecting and choosing CUDA cards, findDevices.cu 4 of 4

Take into account that cudaGetDeviceCount can take a bit to run, as the data for the whole structure has to be piped through the PCI-Express bus.

If you need to query something in particular from the card as fast as possible, you should use cudaDeviceGetAttribute instead - for example:

```
cudaDeviceGetAttribute(&smemSize, cudaDevAttrMaxSharedMemoryPerBlock, devId);
```

```
cudaDeviceGetAttribute(&numProcs, cudaDevAttrMultiProcessorCount, devId);
```

Some properties are more time expensive than others, though - for example, the clock rates of the GPU core or of the memory are fairly expensive.

Transferring and handling data from and to the graphics card

We have already seen how to do this from a 1d vector to another.

One problem that we have with the memory on the GPU is that we cannot modify the values on it directly from the host, and another is that pointers to pointers in the global memory are slow when used.

The solution to this problem is to work always with dimension one arrays on the GPU, which means that we will have to calculate the relative 1d position. Let's see a few examples.

Transferring and handling 1D data

First, let's review how we did this with 1D data:

This is how we allocated the memory:

```
/* Allocate arrays a, b and c on host */
a = (float*) malloc(N*sizeof(float));
b = (float*) malloc(N*sizeof(float));
c = (float*) malloc(N*sizeof(float));

/* Allocate arrays a_d, b_d and c_d on device */
cudaMalloc ((void **) &a_d, sizeof(float)*N);
cudaMalloc ((void **) &b_d, sizeof(float)*N);
cudaMalloc ((void **) &c_d, sizeof(float)*N);
```

And this is how we copied the data to the GPU:

```
/* Copy data from host memory to device memory */
cudaMemcpy(a_d, a, sizeof(float)*N, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b, sizeof(float)*N, cudaMemcpyHostToDevice);
```

Handling 2D data, 1 of 6

One important thing to take into account when working on the GPU is that we cannot use all the functionality that we can use on the CPU. For example, an array that has been allocated in the GPU through the use of the `cudaMalloc` function, for example, such as:

```
int **aGPU;  
cudaMalloc ((void **) &aGPU, sizeof(int *)*N);  
for (i=0;i<N; i++)  
    cudaMalloc ((void **) &(aGPU[i]), sizeof(int)*M);
```

could be, theoretically, accessed safely like this:

```
for (i=0;i<N; i++)  
    cudaFree (aGPU[i]);
```

if the code is run on the CPU, since `aGPU` points to memory in the GPU, we cannot guarantee that, when calling `cudaMalloc`, "`aGPU[i]`" will work as expected, as `cudaMalloc` expects an address in the host in that position.

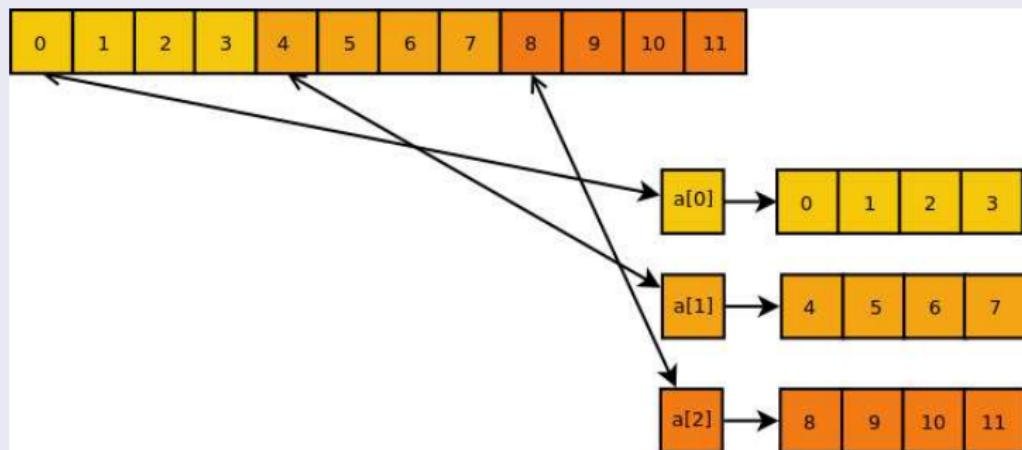
Handling 2D data, 2 of 6

Another problem of using pointers to values in CUDA is that if those pointers reside in the global memory (as in, for example, we want to use them for more than one kernel call), fetching data that is referenced by them would take considerably longer.

If fetching data from the global memory takes, say, a hundred milliseconds, we would need a hundred to get the pointer and another hundred to get the data that the pointer is pointing to.

Handling 2D data, 3 of 6

Therefore, when handling 2D data, we will have to allocate it as 1d data and deal with the appropriate transformations to request elements in appropriate, in the following way:



Handling 2D data, 4 of 6

Let's look at how to do this in the code - this is how we allocate the memory:

```
int **a;
int *a1d = NULL;
int *aGPU;

/* Allocate arrays a and a1d on host */
a1d = (int*) malloc( (N)*(M)*sizeof(int) );
a = (int**) malloc((N)*sizeof(int*));

/* Then make sure that a points to the right parts of a1d */
for (i=0;i<N; i++)
    a[i]=(&(a1d[i*M]));

/* Allocate array aGPU on device */
cudaMalloc ((void **) &aGPU, sizeof(int)*N*M);
```

And this is how we copy the data to the GPU, as a long 1d vector:

```
/* Copy data from host memory to device memory */
cudaMemcpy(aGPU,a1d, sizeof(int)*N*M, cudaMemcpyHostToDevice);
```

Handling 2D data, 5 of 6

And this is how to free the allocated space after that

```
// Free the memory allocated in 1d  
free(a1d);  
// Free the pointers allocated to point to 1d memory  
free(a);  
  
cudaFree (aGPU);
```

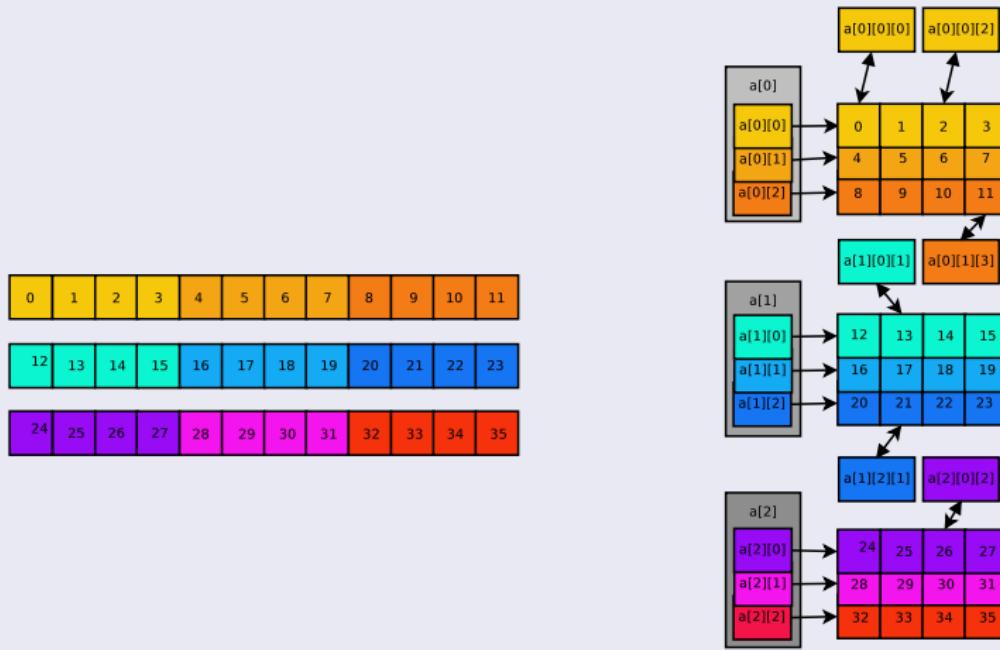
Handling 2D data, 6 of 6

Comment for C++ programmers - it is safe to use 1d stl vectors, but you have to do something similar when using multidimensional data. We are providing sample code for that - see the helloThread2DCpp sample

```
std::vector< int* > threadXIds;
std::vector< int > threadXIds1d;
...
// threadXIds1d is the one dimensional vector with all the contiguous data
threadXIds1d.resize(N*M);
// threadXIds is a vector of pointers to one dimension arrays
threadXIds.resize(N);
// The advantage here is that threadXIds works exactly the same as
// pointers to one dimension arrays
for (i=0;i<N;i++) {
    threadXIds[i]=(&(threadXIds1d[i*M]));
}
...
// We use the pointer to the first element to pass data to and from the GPU
cudaMemcpy(&(threadXIds1d[0]),threadXIdsGPU,sizeof(int)*N*M,cudaMemcpyDeviceToHost);
...
// Free the memory
threadXIds.clear();
threadXIds1d.clear();
```

Handling 3D data, 1 of 3

A similar, but slightly more complex approach, is used when dealing with 3d data:



Handling 3D data, 2 of 3

Now let's see how to do this with 3D data:

This is how we allocate the memory:

```
int ***a = NULL;
int *a1d = NULL;
int *aGPU;
    /* Allocate arrays a and a1d on host */
a1d = (int*) malloc( N*M*L*sizeof(int) );
a = (int***) malloc(N*sizeof(int**));
    /* Then make sure that a points to the right parts of a1d*/
for (i=0;i<N;i++) {
    int **aPointer = (int**) malloc(M*sizeof(int *));
    for (j=0;j<M;j++) {
        aPointerX[j]=(&(a1d[i*M+j*L]));
    }
    a[i]=aPointer;
}
    /* Allocate array aGPU on device */
cudaMalloc ((void **) &aGPU, sizeof(int)*N*M*L);
```

And this is how we copy the data to the GPU, as a really long 1d vector:

```
/* Copy data from host memory to device memory */
cudaMemcpy(aGPU,a1d, sizeof(int)*N*M*L, cudaMemcpyHostToDevice);
```

Handling 3D data, 3 of 3

And this is how to free the allocated space after that

```
// Free the memory allocated in 1d
free(a1d);
// Free the pointers allocated to point to 1d memory
for (i=0;i<N;i++) {
    free(a[i]);
}
// Free the pointers allocated to point to 2d memory
free(a);
cudaFree(aGPU);
```

Jacobi Method, 1 of 6

We have already seen how to move 2d data to the graphics card and some trivial CUDA code, now let us see how to something slightly more complex - let's write a serial Jacobi solver multiplication program and then convert it into CUDA. This algorithm will solve the system

$$Ax = b$$

Where:

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{n-1,0} \\ \vdots & \cdots & \vdots \\ a_{0,n-1} & \cdots & a_{n-1,n-1} \end{pmatrix}; x = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}; b = \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

Jacobi Method, 2 of 6

The algorithm works as follows: If we decompose the matrix A such as:

$$A = D + R$$

where D is the diagonal and R the rest of the matrix:

$$D = \begin{pmatrix} a_{0,0} & 0 & \cdots & 0 \\ 0 & a_{1,1} & \cdots & \vdots \\ \vdots & \cdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} \end{pmatrix}; R = \begin{pmatrix} 0 & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & 0 & \cdots & \vdots \\ \vdots & \cdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & 0 \end{pmatrix}$$

Jacobi Method, 3 of 6

So now we can rewrite the system as:

$$(D + R)x = b \Rightarrow Dx + Rx = b$$

$$\Rightarrow Dx = b - Rx \Rightarrow x = D^{-1}(b - Rx)$$

and then we consider the iterative method:

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)})$$

which, element-wise, can be calculated:

$$x_i^{(k+1)} = (a_{i,i}^{-1}(b_i - \sum_{j=0, j \neq i}^{n-1} a_{i,j} x_j^{(k)}))$$

Jacobi Method, 4 of 6

You have to take into account that the standard convergence condition for the Jacobi method is that the spectral radius of the iteration matrix is less than one:

$$\rho(D^{-1}R) < 1$$

That means that the

$$\rho(D^{-1}R) := \max_i(|\lambda_i|)$$

Where λ_i are the eigenvalues of $D^{-1}R$

A method is guaranteed to converge if the matrix A is strictly diagonally dominant - that is, that

$$|a_{i,i}| > \sum_{j=0, j \neq i}^{n-1} |a_{i,j}|$$

For this example, we are providing systems that converge with this method - but be warned that not all systems will converge.

Jacobi Method, 5 of 6

This is how to compute a new iteration in C - in the CPU:

```
void iterateCPU (int N, float **A, float *b, float *xOld, float *xNew) {  
    int i,j;  
    float sumatory;  
    for (i=0;i<N;i++) {  
        sumatory=b[i];  
        for (j=0;j<N;j++) {  
            if (i!=j)  
                sumatory-=(A[i][j]*xOld[j]);  
        }  
        sumatory+=(1.0f/A[i][i]);  
        xNew[i]=sumatory;  
    }  
}
```

Jacobi Method, 6 of 6

Now, using CUDA, we can split the x^{k+1} and x^k vectors into 1d blocks, and then process each iteration in the following way:

```
__global__ void iterateGPU (int N, float *A1dGPU, float *bGPU, float *xOld, float *xNew) {  
    int idx=blockIdx.x*blockDim.x+threadIdx.x;  
    int j;  
    float sumatory;  
  
    if (idx<N) {  
        sumatory=bGPU[ idx ];  
        for (j=0;j<N;j++) {  
            if (idx!=j) {  
                sumatory-=(A1dGPU[ j+idx*N ]*xOld[ j ]);  
            }  
        }  
        sumatory+=(1.0f/A1dGPU[ idx+idx*N ]);  
        xNew[ idx ]=sumatory;  
    }  
}
```

Writing to standard output from the GPU, 1 of 2

Some good news - you can use printf in the gpu code!

The printf calls will slow down the execution quite a bit, so they should be removed in production code - it is recommended to surround the calls with #ifdef conditions to make this easier.

```
#ifdef WITH_MY_DEBUG
    printf ("threadIdx.x=%d  =%d\n", blockIdx.x, threadIdx.x);
#endif
```

No cout so far, though!

Writing to standard output from the GPU, 2 of 2

You can then use the option “`-DWITH_MY_DEBUG`” on the makefile tell the preprocessor to use (or not) that option, like this:

```
NVCCFLAGS = -g -G --use_fast_math -arch=sm_30 -DWITH_MY_DEBUG  
myCudaFile.o: myCudaFile.cu  
$(NVCC) myCudaFile.cu -c $(NVCCFLAGS) -I$(INCPATH)
```

So you can remove all the `printf` calls on your code just by removing the “`-DWITH_MY_DEBUG`” on the Makefile

Error handling, 1 of 4

In Cuda, errors are handled in a similar way as in OpenGL - which means that, instead of reporting them and stopping the execution straight away, we keep running the code.

This is because in computer graphics you can get away with this. The good news are that the runtime maintains an error variable that we can check.

Thus, since we don't get obvious errors (like, for example, when we get a segmentation fault on the host) on code that we run on the GPU, it is important to keep track of those error variables.

Error handling, 2 of 4

Every CUDA call -except for kernel launches, that are void functions- returns an error code of type `cudaError_t`.

Thus, we can compare the return value of any function with `cudaSuccess` - and, if the returned value is different than the success value, we have had suffered some kind of error.

We can get a description of the error by using the function
`char *cudaGetErrorString(cudaError_t code);`

However, this error handling can sometimes cause problems, as we are not guaranteed that the kernels will always execute instructions in the same order. For example, if we get two errors at almost the same time during execution, we might get one or the other, depending on which one happened last.

Error handling, 3 of 4

Another option that we have, instead of grabbing the error on the function call, is to use the `cudaGetLastError()` function to recover the last error that we got in any code that we have run in the GPU - this function is called from the host (the CPU), not from the device (GPU).

We are providing a sample function that lets the user call this function and output the error message prepended by a message from the programmer:

```
void cudaLastErrorCheck (const char *message) {
    cudaError_t err = cudaGetLastError();
    if( err!= cudaSuccess ) {
        printf("(Cuda error %s): %s\n", message, cudaGetStringError( err) );
        exit(EXIT_FAILURE);
    }
}
```

Error handling, 4 of 4

It must be taken into account that `cudaGetLastError` sets the runtime error variable back to `cudaSuccess`; if we want to keep the same error variable, we can use `cudaPeekAtLastError` instead, in the following way:

```
void lastCUDAErrorPeek (const char *message) {
    cudaError_t err = cudaPeekAtLastError();
    if( err!= cudaSuccess ) {
        printf("(Cuda error %s): %s\n", message, cudaGetStringError( err ) );
        exit(EXIT_FAILURE);
    }
}
```

Thread synchronization, 1 of 4

Sometimes, we need to sync all the threads in a block - to do this, we can use the following function call, equivalent to a MPI_Barrier for all the threads in one given block:

```
void __syncthreads();
```

Calling this function will stop instruction execution on all the threads of any block all threads reach it - which means that using it in conditional code that does not have the same condition on each thread of the block could cause the program to hang or to produce unexpected results.

Thread synchronization, 2 of 4

We also have the possibility, on devices of CC 2.0+, of synchronizing and also evaluating a predicate on all the threads of a block by using one of the following functions:

```
int __syncthreads_count(int predicate);  
int __syncthreads_and(int predicate);  
int __syncthreads_or(int predicate);
```

- `__syncthreads_count` will return the number of threads where `predicate` is non-zero.
- `__syncthreads_and` will return the non-zero if `predicate` is non-zero on all threads.
- `__syncthreads_or` will return the non-zero if `predicate` is non-zero on any threads.

Thread synchronization, 3 of 4

So what happens if we need to sync all the threads in the grid, between blocks? There isn't an easy way to do this, so what we will usually do is to terminate a kernel and start another one (another option is to use the global memory for sync'ing, but that is not considered very good practice).

We do, however, have a **very unreliable way** to try to make the reads and writes from a given thread to the global and shared memories of other threads are finished before we continue:

```
void __threadfence();
void __threadfence_block();
void __threadfence_system();
```

Thread synchronization, 4 of 4

What `__threadfence_block` **is supposed to do** is to make sure that all writes to shared and global memory made by the thread that we are calling it from are observed by all the other threads of the same block.

Calling `__threadfence` **should** guarantee the same, but for all the threads of all blocks - however, we have to be careful with it, since we have to make sure that those other threads are looking at the actual global memory and not cached versions of it - we can tell the compiler not to cache memory by declaring it `as volatile`.

`__threadfence_system` **should** do the same as `threadfence`, and also guarantee that the changes are also seen in memories that are shared with the host and that we will see later.

Matrix multiplication, 1 of 4

Now let us see how to something slightly more complex - let's write a serial matrix multiplication program and then convert it into CUDA. We'll assume that the matrices are such as:

$$C = AB; \text{ where } A \in \mathcal{M}_{n \times m}(\mathbb{R}); B \in \mathcal{M}_{m \times n}(\mathbb{R}); C \in \mathcal{M}_{n \times n}(\mathbb{R})$$

The matrix multiplication algorithm is an easy one - each element in the product matrix C is computed as

$$c_{i,j} = \sum_{k=0,1,m-1} a_{i,k} b_{k,j}$$

Matrix multiplication, 2 of 4

So the serial algorithm is just a double loop on $c_{i,j}$

```
void matmulCPU (int N,int M,float **A,float **B,float **C) {  
    int i,j,k;  
    for (i=0;i<N;i++) {  
        for (j=0;j<N;j++) {  
            C[i][j]=0.0f;  
            for (k=0;k<M;k++) {  
                C[i][j]+=A[i][k]*B[k][j];  
            }  
        }  
    }  
}
```

Matrix multiplication, 3 of 4

To turn this into cuda code, we allocate A,B and C in the GPU, and then run each $c_{i,j}$ in a thread, so we compute:

```
__global__ void computeMatMulGPU (int N,int M, float *A1d, float *B1d, float *C1d) {  
    int idx=blockIdx.x*blockDim.x+threadIdx.x;  
    int idy=blockIdx.y*blockDim.y+threadIdx.y;  
    int k;  
  
    if (idx<N) {  
        if (idy<N) {  
            float sumatory=0.0f;  
            for (k=0;k<M;k++) {  
                sumatory+=A1d[k+idx*M]*B1d[k+idy*M];  
            }  
            C1d[ idx+idy*N]=sumatory;  
        }  
    }  
}
```

(We are assuming that in this case, B has been transposed for optimal efficiency)

Matrix multiplication, 4 of 4

You can see also an example of how to use the error checking, in the following code:

```
computeMatMulGPU<<<dimGrid , dimBlock>>>(N,M,A1dGPU, BTrans1dGPU ,C1dGPU) ;  
cudaLastErrorCheck ( " (Cuda error in computeMatMulGPU) " );
```

In this example, if you set the BLOCK_SIZE to an amount larger than allowed in the GPU, you will get:

```
(Cuda error (Cuda error in computeMatMulGPU)): invalid configuration argument
```

C++ in the GPU

There is also support for a subset of C++ functionality on the GPU. It was to be planned very carefully, due to C++ usually allocating objects in memory in a more sparse way.

There are lots of restrictions and caveats; some will be removed in later versions but some other will stay. For example, run time type information, exception handling and the STL library is not supported at the time of writing.

Obviously, this means that if you write GPGPU C++ code, you could have backward compatibility problems.

C++ in the GPU

For example, you cannot use the `__device__`, `__shared__` or `__constant__` modifiers on:

- struct, class and union data members
- formal parameters (ie the parameters that you declare in a function signature).
- local variables within a function that executes on the host.

From CUDA 7 (the cuda version, not the compute capability), you can use C++11 features too. From 9, C++14; and from 12, C++20 (although no modules in GPU code.. so far!).

A word of caution

To close the basic cuda programming part, let us say that while the code that we have seen in this chapter is run on the GPU and can actually provide a substantial speedup, its performance is far from being the best attainable, due to the lack of use of the most efficient kinds of memory available in the GPU.

Assuming that everyone has understood the basics, we will continue with a chapter that is going to be focused on how to improve the performance of the code.

Also, use compute-sanitizer as much as you can!

Advanced CUDA programming

Third chapter: Advanced CUDA programming.

We now know how to run programs in the GPU, but this is only the beginning - our code is still quite far from being optimal!

So we will continue by looking at what the best strategies for increasing the performance of CUDA code are.

Optimization strategies

When trying to improve the performance of our CUDA programs, there are three main strategies that we should try to stick with when optimizing our code:

- Maximize parallel execution
- Optimize memory access and transfers
- Maximize instruction usage performance

Maximizing parallel execution, 1 of 11

What we mean by "maximizing parallel execution" is, essentially, that the usage of compute resources and load balance in the compute resources should be as good as possible.

We must always keep in mind that all the computation done in the GPU is conducted by a number of streaming multiprocessors, that can range between 1 in small cards to 192 in the most recent models (RTX 5090).

Each one of those multiprocessors will run each one of the threads blocks - so if (and this will be most likely the case) we have more blocks than multiprocessors, each multiprocessor will have to compute more than one block.

Maximizing parallel execution, 2 of 11

Let's see an example of wasted resources in our code that we have encountered already:

If you look at the sample code that we have been using, you can see that the size definition of the grid is such as:

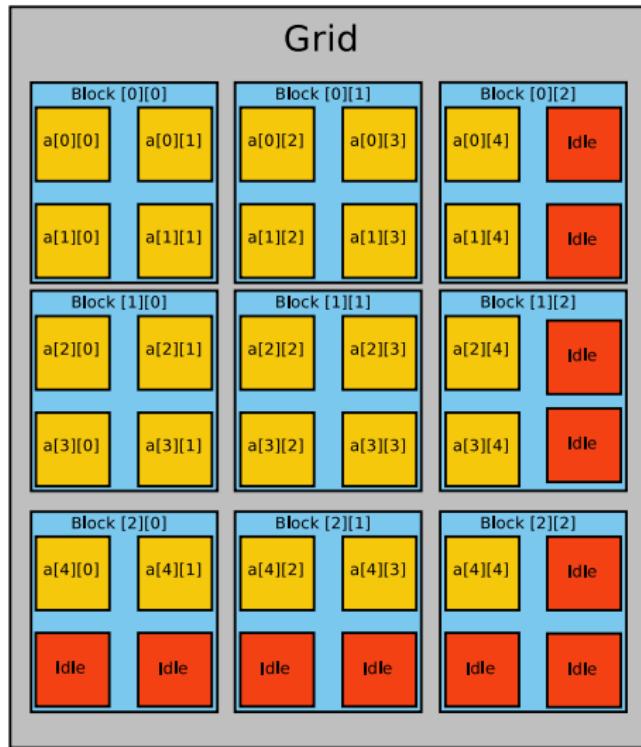
```
dim3 dimBlock(block_size,block_size);  
dim3 dimGrid ((N/dimBlock.x) + (! (N % dimBlock.x)?0:1),  
               (M/dimBlock.y) + (! (M % dimBlock.y)?0:1));
```

Let's suppose that we have a 5x5 matrix ($N=M=5$) and a block size of 2

Then, we will have a grid of 3x3 blocks of size 2x2.

Maximizing parallel execution, 3 of 11

So what really is going on is this:



Maximizing parallel execution, 4 of 11

helloThread2D.cu

This is the reason why we have been adding those two conditional evaluations in our code:

```
int idx=blockIdx.x*blockDim.x+threadIdx.x;
int idy=blockIdx.y*blockDim.y+threadIdx.y;

if ( idx < N ) {
    if ( idy < M ) {
        // Do something in array[idx+idy*N]
    }
}
```

This way, we make sure that we do not go over the boundaries of the array - an error that is very difficult to spot since we do not get segmentation faults if we do; but we still have the possibility of wasting compute time as some multiprocessors will be idle for a while.

Maximizing parallel execution, 5 of 11

Therefore, ideally, when we run a kernel, we want to have at the very least as many blocks as multiprocessors we have in the machine. This does not have to be necessarily true, though.

Also, in case you happen to need fewer blocks than usual for a particular task that doesn't require a lot of them, something that you might want to consider is the possibility of running more than one kernel at the same time, provided that you have access to a GPU with CUDA capability 2.0.

And it would be strongly advised for the number of threads per block to be a multiple of a `warpSize` (usually 32), as the blocks are processed in groups of 32 threads called warps. Let's have a look again at why.

Maximizing parallel execution, 6 of 11

Let us recap again what happens when we have our code follow two different paths in the same warp. Let us consider the following code:

doSomething.c

```
__device__ void doSomething (float *in, float *out, int numberOfElements) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    if ( idx < (warpSize/2) ) {
        out[idx]=in[idx]+threadIdx.x;
        out[idx]*=2;
    } else {
        out[idx]=-in[idx]-threadIdx.x;
    }
}
```

Maximizing parallel execution, 7 of 11

What happens at instruction level is this:

	Thread 0	...	Thread 15	Thread 16	Thread 17	...
Warp 0	$idx=0$...	$idx=15$	$idx=16$	$idx=17$...
Warp 1	$idx=32$...	$idx=47$	$idx=48$	$idx=49$...
Warp 0	$idx < 16$...	$idx < 16$	$idx \geq 16$	$idx \geq 16$...
Warp 1	$idx \geq 16$...	$idx \geq 16$	$idx \geq 16$	$idx \geq 16$...
Warp 0	$out[0]=in[0]+0$...	$out[15]=in[15]+15$	<i>*idle*</i>	<i>*idle*</i>	...
Warp 1	$out[32]=-in[32]-32$...	$out[47]=-in[47]-47$	$out[48]=-in[48]-48$	$out[49]=-in[49]-49$...
Warp 0	<i>*idle*</i>	...	<i>*idle*</i>	$out[16]=-in[16]-16$	$out[17]=-in[17]-17$...
Warp 0	$out[0]=2.0*out[0]$...	$out[1]=2.0*out[1]$	<i>*idle*</i>	<i>*idle*</i>	...

Warp 0 takes a lot longer to execute than Warp 1 due to the warps having to handle two execution paths - so try to prevent divergent execution in warps (if you can!). But, how do you do this?

Maximizing parallel execution, 8 of 11

Another option, when working with devices of compute capability 1.2 and higher, is to use Warp vote functions to make the algorithm that runs on each thread of a warp be as similar as possible.

For example, using `int __all_sync(unsigned mask, int predicate)` allows us to evaluate predicate in all active threads of a given warp and will return non-zero if and only if predicate is non-zero for all those threads.

Maximizing parallel execution, 9 of 11

Using `int __any_sync(unsigned mask, int predicate)` allows us to evaluate predicate in all active threads of a given warp and will return non-zero if predicate is non-zero in any of those threads.

Devices of compute capability 3.0+ (ie, Kepler architecture) allow you also to share variables between the threads of a warp, so you can maximize parallel execution even further.

Maximizing parallel execution, 10 of 11

Thus, it is a good idea to get all the threads of a warp to run code which is as similar as possible for best performance.

Otherwise, you will create two different execution paths, and every time that the warp goes into execution, only some of the threads in the warp will do something, while others are idle; and whenever it goes again up the queue, the opposite will happen.

Yet again, make use that you use `warpSize` and not 32 to future-proof your code!

Maximizing parallel execution, 11 of 11

This is way too much text - give me the abridged version!

Ok then - try to have as many blocks active in a given streaming multiprocessor as you can!

How do you do that? Balance! Make sure that the number of threads per block, the amount of register space used per block and the amount of shared memory per block is as low as possible so we have as many active blocks per SM as possible (we will see later on that this is called Occupancy) while at the same time as high as possible to benefit from parallelism from within each group.

Optimize memory access and transfers, 1 of 10

The first strategy, when optimizing code when thinking about the memory handling, is to minimize the number of transfers that we make between the different memories (for example, from the host to the device).

Also, in this chapter, we will learn about different kinds of memory in the GPU that have different characteristics and therefore its usage will be useful to increase the performance of different part of the program.

Let's start with a recap of the different types of memory

Optimize memory access and transfers, 2 of 10

- Registers
- Shared memory
- Constant Memory
- Texture Memory
- Surface Memory
- Global Memory

Optimize memory access and transfers, 3 of 10

Registers

They are the closest memory to the multiprocessor, and will only be used by one of them at a time.

Accessing them takes only one cycle.

This memory only lasts as long as the thread does, so if you need data to remain in the GPU, you have to use the global, constant, texture or surface memories instead.

Optimize memory access and transfers, 4 of 10

Shared memory

It is very close to the cuda cores, and can also be accessed in only one cycle. It is shared between all the threads of a single block (all the threads of a block are run in a single multiprocessor, and this is the reason - so this memory can be accessed from all of them quickly).

It is still limited in space, but it is one of the most useful types of memory that we can use (**if we can reuse its data!!**) - and the main goal of memory handling optimization in CUDA is to use the shared memory every time that it is possible.

Also, do not forget that the shared memory only lasts as long as a block does, so if you need data to remain in the GPU, you have to use the global, constant, texture or surface memories instead.

Optimize memory access and transfers, 5 of 10

Constant memory

This memory is still located in the DRAM of the device, but, compared to the global memory, it is actually much faster - the drawback is that we are really limited in how much of it (we might be talking about a limit of 64K in devices that now range between CC 1.X and 3.X). The access time can range from 1 to hundred of cycles, depending on locality of both the memory and the requesting multiprocessor

Optimize memory access and transfers, 6 of 10

Texture memory

This memory is also located in the DRAM of the device, but, yet again, compared to the global memory, it is also faster. Another advantage is that we can allocate a fairly large amount of it, as it uses memory allocation originally designed to hold read-only texture images that can be quite large and to map them to polygons as quick as possible - and that process has been gradually optimized over the last 20 years.

The access time is on the scale of hundred of cycles, making it faster than global memory and still reachable from any multiprocessor - so in some cases, it might be the best option if many multiprocessors have to read the same large data set.

Optimize memory access and transfers, 7 of 10

Surface memory

Pretty much like the texture memory (including the fact that it is cached), but you can write to it as well.

Not coherent - if you modify it, you do not know for sure that the next reads are going to have the latest values, unless you do it after a kernel call or a memory copy between the GPU and the host.

Optimize memory access and transfers, 8 of 10

Global memory

This memory is the first one that we use when moving any kind of data into the GPU. It is located in the DRAM of the device, and therefore we can allocate a large amount of space with it - plus it has the advantage of letting us allocate dynamically the memory in a really easy way.

The main drawback is that it is actually rather slow - since it is allocated in the DRAM and it is not cached near the multiprocessors that are going to be using that data.

Optimize memory access and transfers, 9 of 10

Memory types table

Type	Location	Cached	Speed
Registers	Dedicated hardware next to the multiprocessor	Yes	Single cycle
Shared memory	Dedicated hardware next to the multiprocessor	Yes	Single cycle
Constant memory	Dedicated hardware next to the multiprocessor	Yes	One to hundreds of cycles
Texture memory	DRAM of the graphics card	Yes	Hundreds of cycles
Surface memory	DRAM of the graphics card	Yes	Hundreds of cycles
Global memory	DRAM of the graphics card	No	Very slow

So the standard strategy is to use the better memory available - the shared, if the threads only need to access appropriate parts of the data, and, if not, the constant for small datasets and the texture for bigger ones.

Optimize memory access and transfers, 10 of 10

We have already seen this and it is boring - is that all that we need to know?

Nope! We also want to reduce as much as possible the number of memory transfers, specially from and to the slowest memories.

The tricky thing in this case is that we have to think of minimization on a per block basis, not at a per thread basis like we do in the CPU! So think of giving data to your groups of threads, not to each individual thread at a time.

Maximize instruction usage performance

The last, but by no means least, critical step in optimizing CUDA functions is to maximize the performance of our instructions - this can be done, for example, by minimizing the number of times that we synchronize threads in a kernel, as well as increasing as much as possible the performance of our code. Since this is not CUDA specific, we won't be going deep into this topic here.

The only part that we are going to mention is that, in some cases, it might be worth to use variables and function calls with lower precision (single instead than double), as they have lower overheads; it is also worth mentioning that cards with the most modern CUDA capabilities have a higher number of operations per clock cycle per multiprocessor than old ones, so with programs with a high arithmetic load, you might have very significant improvements just by using more modern hardware.

Using and allocating memory in the GPU, 1 of 10

In this chapter, we are going to show how to use the global memory in the GPU - we have used it before, but we are going to show a couple of methods to increase the transfer and loading performance , for those cases in which substitution for other types is not possible for any reason.

Using and allocating memory in the GPU, 2 of 10

First, let's do a recap on how we did `cudaMalloc` and did a normal transfer:

This is how we allocated an array of N floating point:

```
/* Allocate array on host */
array = (float*) malloc(N*sizeof(float));

/* Allocate array on device */
cudaMalloc ((void **) &arrayGPU, sizeof(float)*N);
```

And this is how we copied the data to the GPU:

```
/* Copy data from host memory to device memory */
cudaMemcpy(arrayGPU, array, sizeof(float)*N, cudaMemcpyHostToDevice);
```

Using and allocating memory in the GPU, 3 of 10

Another possibility is to use cudaArrays, special structures that we will use when dealing with texture memory. Let's see an example on how to use it.

cudaMallocArray

```
/* Allocate array on host */
float* floatArray;
floatArray = (float*) malloc( size*sizeof(float) );

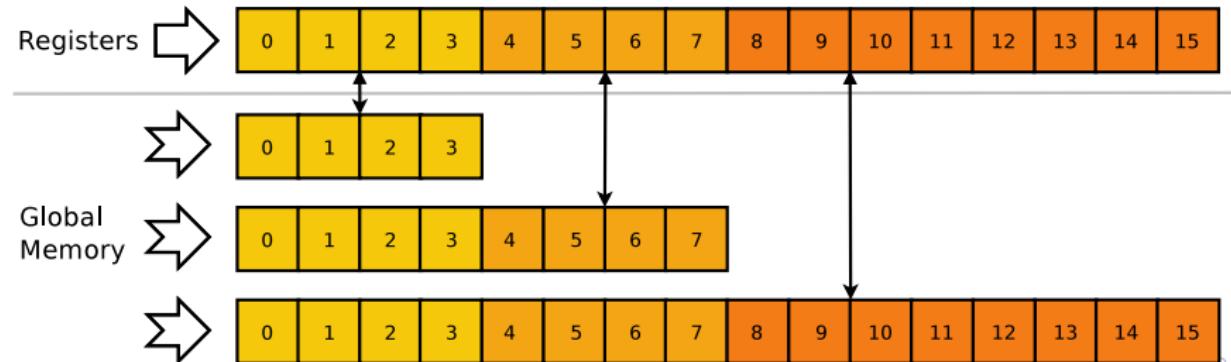
/* Allocate array on device */
cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width, height );

/* Copy data from host memory to device memory */
cudaMemcpyToArray(cu_array, 0, 0, floatArray, size*sizeof(float), cudaMemcpyHostToDevice);
```

We will use this function calls to transfer data into the texture memory, so you can check later examples to see how to use the library calls.

Using and allocating memory in the GPU, 4 of 10

One concept that has to be understood quite clearly about CUDA programming is that it is vital that the global memory is as well managed as possible, and something that we need to know about it is that the device reads 32bit (4 bytes, the size of a float), 64bit (8 bytes, the size of a double) or 128bit (16 bytes, the size of two doubles) words from the global memory into registers, and back.

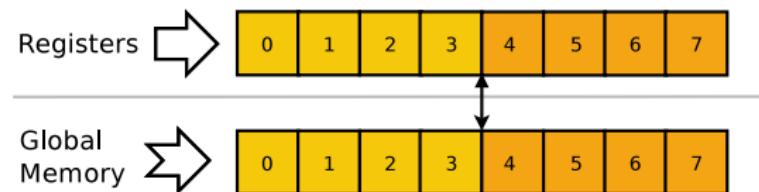


Using and allocating memory in the GPU, 5 of 10

So the first thing that we should do is to store the data on the global memory in such a way that it is aligned to 4,8 or 16 bytes. We can do this by using the prefix `__align__` on structures. For example, we could align two floats like this:

```
typedef struct __align__ (8) {  
    float x;  
    float y;  
} XY;
```

and we will be transferring:



which is actually the same of not using align at all, but:

Using and allocating memory in the GPU, 6 of 10

if we tried to transfer 3 floats, and we align to add some padding this way:

```
typedef struct __align__ (16) {  
    float x;  
    float y;  
    float z;  
} XYZ;
```

that will be stored in this way:



so we can move the three floats as a single memory transaction, instead of 2.

Using and allocating memory in the GPU, 7 of 10

A very common scenario as well is that we have to use multidimensional data, which might make things more complicated if we want to use structures - that is why cuda provides two function calls to deal specifically with that kind of datasets: `cudaMallocPitch` and `cudaMalloc3D`.

If we replace `cudaMalloc` with them, we will be getting properly aligned memory datasets.

Using and allocating memory in the GPU, 8 of 10

This is how to use `cudaMallocPitch`:

```
float* arrayGPU;           //2D array in the GPU (x,y)
size_t arrayGPUSize ;     // This variable will be set by cudaMallocPitch;
cudaMallocPitch( (void**)&arrayGPU,&arrayGPUSize , (size_t)(dimX*sizeof(float)) ,
dimY );
...
// Copy the 2d data from array into arrayGPU
cudaMemcpy2D(arrayGPU,arrayGPUSize ,array ,dimX*sizeof(float) ,dimX*sizeof(float) ,
dimY ,cudaMemcpyHostToDevice );
...
// Copy back the 2d data from arrayGPU into array
cudaMemcpy2D( array ,dimX*sizeof(float) ,arrayGPU ,arrayGPUSize ,dimX*sizeof(float) ,
dimY ,cudaMemcpyDeviceToHost );
```

Using and allocating memory in the GPU, 9 of 10

This is how to use cudaMalloc3D:

```
// We allocate 3d memory with cudaMalloc3D, with dimensions dimX x dimY x dimZ
cudaPitchedPtr array3DGPU;
cudaExtent extent = make_cudaExtent(dimX, dimY, dimZ);
cudaMalloc3D(&array3DGPU, extent);

...
// Memory transfer
// Copy arrayIn from host memory data to device memory data arrayIn => into array3DGPU
cudaMemcpy3DParms p = {0};
// describe the source pointer
p.srcPtr.ptr = arrayIn;
p.srcPtr.pitch = dimX * sizeof(float);
p.srcPtr.xsize = dimX;
p.srcPtr.ysize = dimY;
// describe the destination pointer
p.dstPtr.ptr = array3DGPU.ptr;
p.dstPtr.pitch = array3DGPU.pitch;
p.dstPtr.xsize = dimX;
p.dstPtr.ysize = dimY;
// describe the 3d
p.extent.width = dimX*sizeof(float);
p.extent.height = dimY;
p.extent.depth = dimZ;
// describe the direction of copy
p.kind = cudaMemcpyHostToDevice;
cudaMemcpy3D(&p);
```

Using and allocating memory in the GPU, 10 of 10

And this is how to copy memory out when using cudaMalloc3D:

```
////////////////////////////////////////////////////////////////////////  
// Memory transfer  
// Copy arrayIn from host memory data to device memory data arrayIn => into array3DGPU  
cudaMemcpy3DParms pOut = {0};  
// describe the source pointer  
pOut.srcPtr.ptr = array3DGPU.ptr;  
pOut.srcPtr.pitch = array3DGPU.pitch;  
pOut.srcPtr.xsize = dimX;  
pOut.srcPtr.ysize = dimY;  
// describe the destination pointer  
pOut.dstPtr.ptr = arrayIn;  
pOut.dstPtr.pitch = dimX * sizeof(float);  
pOut.dstPtr.xsize = dimX;  
pOut.dstPtr.ysize = dimY;  
// describe the 3d  
pOut.extent.width = dimX*sizeof(float);  
pOut.extent.height = dimY;  
pOut.extent.depth = dimZ;  
// describe the direction of copy  
pOut.kind = cudaMemcpyDeviceToHost;  
cudaMemcpy3D(&pOut);
```

Using and allocating memory in the host, 1 of 5

We can also allocate memory in the host RAM in ways that, depending on what we want to do with them, can be more efficient when using CUDA.

An example is pinned or locked host memory, that we can be allocated by using `cudaHostAlloc` or `cudaMallocHost()` and freed with `cudaFreeHost` - or, as an alternative, we can page-lock memory that we have malloc'ed previously by using the `cudaHostRegister` function call.

The difference between locked (or pinned) memory and non-locked (or non-pinned) memory is that the non-locked can be in the RAM or in the swap memory space, in a hard drive. Locked (or pinned) memory is guaranteed to be in the RAM, so the GPU can access it.

Using and allocating memory in the host, 2 of 5

The advantages of using this memory are:

- We can copy the memory between the host and device memories while running a kernel at the same time.
- On some devices, we can map the page-locked host memory addresses directly to the device memory, so we do not have to copy it ourselves anymore.
- By making it "Write-Combining" as well- by passing the flag `cudaHostAllocWriteCombined` to the `cudaHostAlloc` call, we prevent that memory from using the L1 and L2 caches in the CPU, but because the memory is not "snooped" (we do not have to check if the memory is marked as "dirty" and look up the caches to make sure that we have the most recent copy), we can improve transfer performance by up to 40%.

Using and allocating memory in the host, 3 of 5

- Another option is to mark a block of page-locked memory as *portable*, by passing the flag `cudaHostAllocPortable` when calling `cudaHostAlloc` or the `cudaHostRegisterPortable` to the `cudaHostRegister` call.
- Also, if we use CC 1.0+, we can mark page-locked memory as *mapped* - which means that the memory is mapped into the memory space of the device - which certainly has some advantages, like being able to copy data between the kernel and the host without having to allocate it on the GPU, but forces us to synchronize memory accesses by using streams or events.

Using and allocating memory in the host, 4 of 5

And yet another option is to use Unified memory. We can do this with the `cudaMallocManaged` function.

This memory can also be accessed from both the CPU and the GPU.

The difference between the mapped and the unified memories is that if one byte is requested and it is not already in the device, mapped will transfer one byte and unified / managed will transfer one page.

Using and allocating memory in the host, 5 of 5

Type	Benefits	Cons
Portable	Multiple GPUs can access a single block of memory	Need several devices to be useful
Write-combining	Potencial improve of +40% in transfer rate	It is dependant on CC
Mapped	Adds transparent, asynchronous I/O Extends the memory available from the GPU by using the RAM, with a bandwidth and latency limited by the PCIe bus	It is dependant on CC Should be coalesced to be fast Might require synchronization to be of any use
Unified (Managed)	Can also be accessed from the GPU or CPU	Might be slower as transferring a whole page takes longer

Using multidimensional blocks and kernels, 1 of 4

We haven't explained it yet - albeit we have used it in the examples- but both the grid of blocks and the threads inside each block can be organized in 1d, 2d or 3d structures. Here are examples of how to do it in each way:

```
/////////// 1d grid and block declaration ///////////
dim3 nBlock(block_size);
dim3 nGrid ( (N/nBlock.x) +(!(%nBlock.x)?0:1) );
/////////// 2d grid and block declaration ///////////
dim3 nBlock(block_size,block_size);
dim3 nGrid ( (N/nBlock.x) +(!(%nBlock.x)?0:1),(M/nBlock.y) +(!(%nBlock.y)?0:1) );
/////////// 3d grid and block declaration ///////////
// Block size has to be L in Z since old versions of CUDA do not allow 3d grids
dim3 nBlock(block_size,block_size,L);
// Which is why we have to use "1" as the third dimension here:
dim3 nGrid ( (N/nBlock.x) +(!(%nBlock.x)?0:1),(M/nBlock.y) +(!(%nBlock.y)?0:1),1);
```

You will notice that in the 3d case, we use "L" for the dimension in the block size, and "1" in the grid.

Using multidimensional blocks and kernels, 2 of 4

The reason for that is that, when dealing with 3d datasets, a block can have threads organized in three dimensions, but with older CCs, the grid of blocks can only have 2. If we look at the output of an information request on an old GeForce 9800 GTX+, we see that:

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce 9800 GTX+  
GPU number of multi-processors: 16  
GPU CUDA cores per multi-processor: 8  
GPU total number of CUDA cores: 128  
GPU Compute capability: 1.1  
GPU Maximum size of each dimension of a grid: 65535x65535x1  
GPU Maximum size of each dimension of a block: 512x512x64  
GPU Maximum number of threads per block: 512  
GPU 32-bit registers available per block: 8192  
GPU Shared memory available per block in bytes: 16384  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

So, while in a block, we can have threads with up to 64 levels in the Z axis, in the grid (of blocks), we can only have one - that means that we are more restricted when partitioning 3d data in cuda. This is, however, no longer the case in the most modern architectures.

Using multidimensional blocks and kernels, 3 of 4

Let us compare this with the data obtained from a new GeForce GTX780:

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce GTX 780  
GPU number of multi-processors: 12  
GPU CUDA cores per multi-processor: 192  
GPU total number of CUDA cores: 2304  
GPU Compute capability: 3.0  
GPU Maximum size of each dimension of a grid: 2147483647x65535x65535  
GPU Maximum size of each dimension of a block: 1024x1024x64  
GPU Maximum number of threads per block: 1024  
GPU 32-bit registers available per block: 65536  
GPU Shared memory available per block in bytes: 49152  
GPU Constant memory available on device in bytes: 65536  
GPU Warp size in threads: 32
```

So we are not as restricted when partitioning 3d data in cuda. However, we are still a lot more limited in the last dimension than in the first two, specially compared to the first.

Using multidimensional blocks and kernels, 4 of 4

One word of warning when working with particularly large 1D grids.

Even when the card *says* that it allows you up to 2,147,483,647 blocks in a 1D grid:

```
GPU Compute capability: 3.0
GPU Maximum size of each dimension of a grid: 2147483647x65535x65535
GPU Maximum size of each dimension of a block: 1024x1024x64
```

```
NVCCFLAGS = -O2 -arch=sm_30
```

Using shared memory, 1 of 6

As we mentioned before, there are different kinds of memory spaces available in CUDA - and both performance and usability changes quite a lot from to another.

That means that, compared with other types of parallel programming, when we design the parallel algorithms, we will have to be very careful not only about how do we split our data between different processes, but also in which memory space we are doing it.

Let's start with the shared memory, which is the one that we will always try to use if we can.

Using shared memory, 2 of 6

The shared memory has a name that can be a bit confusing. It is called shared because **it is shared inside a block**.

That means that **all the threads inside the same block can access and write into it**.

The reason for this is that, as we saw before, **a single multiprocessor runs each one of the threads of a single block, so this memory resides in that particular multiprocessor**. Which means, of course, that we are limited in how much we can use - but the advantage is that it is stored in dedicated hardware from which **it can be accessed very quickly**.

Using shared memory, 3 of 6

In the previous example, with a fairly old GeForce 9800 GTX+, we saw that:

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce 9800 GTX+  
GPU Shared memory available per block in bytes: 16384 (16KBs)  
GPU Maximum number of threads per block: 512
```

so we have $(16384/512)=32$ bytes per thread (f.ex. 8 floating point numbers per thread), if we use the maximum number of threads per block. In the RTX 2080 Super (Ampere):

```
////////////////////////////////////////////////////////////////////////  
GPU model name: NVIDIA GeForce RTX 2080 SUPER  
GPU Shared memory available per block in bytes: 49152 (48KBs)  
GPU Maximum number of threads per block: 1024
```

So in this case, we have $(49152/1024)=48$ bytes per thread (f.ex. 12 floating point numbers per thread), if we use the maximum number of threads per block. With Pascal, we have 48 KBs available per block again, and still the same limit of 1024 threads per block.

Using shared memory, 4 of 6

Let's see an example of how to allocate and use shared memory - we will use the `__shared__` prefix to do that:

```
#define BLOCK_SIZE 32
// In a global or device function
...
__shared__ float sharedMemoryArray[BLOCK_SIZE];
sharedMemoryArray[threadIdx.x] = (float)threadIdx.x;
```

- We are using a fixed allocation - while it is possible to dynamically allocate shared memory, it is usually not a very good idea until you move into production code because this way, we have better control over how much memory we allocate - and `BLOCK_SIZE*sizeof(float)` is usually safe.
- We have changed our access to `sharedMemoryArray[threadIdx.x]`; instead of the usual `sharedMemoryArray[blockIdx.x*blockDim.x+threadIdx.x]` - but quite obviously, if we did that, we will be going out of the boundaries of the array
- We have as many "sharedMemoryArray" arrays as blocks in the grid - which means that we will have to move the data from the shared memory back to the global and from it back to the RAM to get it back.

Using shared memory, 5 of 6

We can also pass pointers to shared memory as arguments to `__device__` functions:

```
////////////////////////////////////////////////////////////////  
__shared__ int sharedMemoryThread[BLOCK_SIZE];  
sharedMemoryArray[threadIdx.x] = threadIdx.x;  
  
__syncthreads();  
  
scanTheadInformationGPUtest ( &(sharedMemoryArray[0]) ,arrayOutGPU ,N) ;
```

where `scanTheadInformationGPUtest` is:

```
__device__ void scanTheadInformationGPUtest( int *sharedArray ,int *arrayOutGPU,int N ) {  
    int idx=blockIdx.x*blockDim.x+threadIdx.x;  
    if ( idx <N ) {  
        arrayOutGPU[idx]=sharedArray[threadIdx.x]+blockIdx.x*blockDim.x;  
    }  
}
```

Using shared memory, 6 of 6

This is one of the most useful memory spaces that we can use when trying to increase the performance of our application - and the most common optimization strategy in CUDA (right after trying to move as little data as possible between the CPU and the GPU) is to use the shared memory whenever possible.

However, you have to remember that this is only useful if **memory from the shared memory can be reutilized by threads of the same block**, so not all algorithms will benefit from it.

Yet again, the reason for that is that **a single multiprocessor runs each one of the threads of a single block, the shared memory is stored inside that particular multiprocessor, so it can be accessed in one single cycle.**

Using constant memory, 1 of 5

Unfortunately, there are cases in which we have to share data over multiple blocks and therefore we need to find better memory management strategies. One possibility is to use the constant memory, that can be accessed in one to hundreds of cycles.

However, as usual with all the different types of memory that CUDA offers, there are some drawbacks that we have to keep in mind - in this case, it is, on one hand, the size (It can be as little as 64KBs) and the fact that it is read only

Using constant memory, 2 of 5

To use the constant memory, first we have to declare it this way, as global variables:

```
////////////////////////////////////////////////////////////////////////  
// This is how to declare a single variable  
__device__ __constant__ int      NCONST;  
// This is how to declare an array  
__device__ __constant__ float    MatrixAConstantGPU [DIM_X][DIM_Y];  
scanTheHeadInformationGPUCTest ( &(sharedMemoryArray[0]) ,arrayOutGPU ,N ) ;
```

As in the shared memory case, it is usually good practice so used fixed size arrays, as if we set them up in any other way, we might end up trying to use more memory than we can. Once allocated, we have to initialize it from the host copying some other memory address into it (as mentioned before, from the device point of view, the memory is read-only)

```
////////////////////////////////////////////////////////////////////////  
cudaMemcpyToSymbol(NCONST           ,&N           ,   sizeof(int)      );  
cudaMemcpyToSymbol(MatrixAConstantGPU ,MatrixA1d     ,   N*M*sizeof(float) );
```

Using constant memory, 3 of 5

Then, we can call it from the GPU as any other variable:

```
////////////////////////////////////////////////////////////////  
__global__ void multiplyMatricesGPU(float *basicFloatArrayGPU) {  
    int i;  
  
    int idx=blockIdx.x*blockDim.x+threadIdx.x;  
    int idy=blockIdx.y*blockDim.y+threadIdx.y;  
  
    int myY = (idy+idx * MCONST )% NCONST ;  
    int myX = ( (idy+idx * MCONST )-myY )/ NCONST ;  
  
    for (i=0;i < MCONST ; i++ ) {  
        basicFloatArrayGPU [idy+idx*MCONST]+= MatrixAConstantGPU [myX][ i ]*  
                                         MatrixBConstantGPU [ i ][myY];  
    }  
}
```

Using constant memory, 4 of 5

And, like with the shared, we can also pass pointers to constant memory as arguments to `__device__` functions:

```
////////////////////////////////////////////////////////////////////////
// Declare MatrixAConstantGPU as a global variable
__device__ __constant__ float MatrixAConstantGPU[DIM_X][DIM_Y];
////////////////////////////////////////////////////////////////////////
// In the host code, set its value
cudaMemcpyToSymbol(MatrixAConstantGPU, MatrixA1d, N*sizeof(float));
```

and in the device function, we will simply use it like this:

```
__device__ void scanTheadInfoGPUtest(float *array, float *arrayOutGPU, int *N) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    if ( idx <(*N) ) {
        arrayOutGPU[idx]=array[threadIdx.x]+(float)(blockIdx.x*blockDim.x);
    }
}
```

Using constant memory, 5 of 5

One more warning about the constant memory: It is fast only if all the threads in a block access the same memory position when they request data from it.

If you set up an array of constant memory and then different threads of the same block request different positions of the same array, the access will be serialized, and performance will be considerably slower.

Using texture memory, 1 of 7

Another possibility, if we need to move a large amount of memory to different blocks, is to use the texture memory.

As with the constant memory, it has the advantage of being accessible by any thread in the grid, and in this case, we are allowed to allocate a much larger amount of read only memory. It is however, slower than the constant memory (hundreds of cycles), so there might still be cases in which we might prefer to use the constant over the texture.

There are, however, some additional features of the texture memory that might be worth highlighting.

Using texture memory, 2 of 7

With the proper configuration, we can use the following features of the texture memory to our advantage:

- Use floating point indices - as opposed to accessing an array the same way as usual, we can request positions with floating point indices.
- Interpolation - if the indices do not match the position of the original array, we can get back a value which is interpolated linearly between the two closest - and the interpolation is much faster than calculating it yourself.
- Going past the boundaries - if you request indices past the boundaries of the array, you can get values as if the array was continually copied after itself.
- Normalised indices - you can set up the texture memory so it accepts indices between 0 and 1 instead of 0 to size -1.

Using texture memory, 3 of 7

Those features might appear a bit confusing at first, but they are just the usual way in which textures are handled in graphical programming.

Something that has to be taken into account when thinking about texture memory is that, in the continuous performance improvement that has driven the graphic cards technology for more than 15 years, texture memory optimization had a very considerable role - the faster the textures were fetched, the faster a frame would be rendered, and the more that a card would sell.

Using texture memory, 4 of 7

Let's see now how to allocate, initialize and use texture memory.

```
////////////////////////////////////////////////////////////////////////  
// declare texture reference for 2D float texture as global variable  
texture<float , 2, cudaMemcpyElementType> tex;  
....  
// In the host:  
// Describe the type of texture that we are going to use  
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,  
                                         cudaChannelFormatKindFloat);  
// Allocate a cuda array  
cudaArray* cu_array;  
cudaMallocArray( &cu_array, &channelDesc, width, height );  
// Then we copy an 1d array to the cudaArray  
cudaMemcpyToArray( cu_array, 0, 0, floatArrayId, size*sizeof(float),  
                  cudaMemcpyHostToDevice );  
// Set texture parameters  
tex.addressMode[0] = cudaAddressModeClamp;  
tex.addressMode[1] = cudaAddressModeClamp;  
tex.filterMode = cudaFilterModePoint;  
tex.normalized = false; // do not normalize coordinates  
// Bind the array to the texture  
cudaBindTextureToArray( tex, cu_array, channelDesc );
```

Using texture memory, 5 of 7

And this is how we access the memory:

```
////////////////////////////////////////////////////////////////  
__global__ void transformKernel( float* g_odata, int width, int height) {  
    // calculate normalized texture coordinates  
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;  
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;  
  
    // This would be how to use the normalised coordinates  
    //float u = x / (float) width;  
    //float v = y / (float) height;  
  
    if ( (x < width) && (y < height) ) {  
        g_odata[y*width+x] = tex2D(tex, x, y);  
    }  
}
```

Using texture memory, 6 of 7

Some final notes about the texture memory:

- The mentioned linear interpolation capability is separate from the scalar processors and extremely efficient.
- It is optimized for 2d arrays - the most commonly used textures in computer graphics.
- You can increase the performance even further by having multiple threads in the same wrap access memory locations next to each other.
- If you use some cuda datatypes (for example "float4"s instead of 4 "float"s) you can increase the performance even further.

In other words, this particular type of memory opens new strategies for code optimization for problems that are particularly well suited.

Using texture memory, 7 of 7

And disadvantages that have to be considered as well:

- In `tex2D(tex, x, y)`, the arrays are transposed compared to what they usually are - the "x" moves in columns, and the y in rows (like a picture in cartesian space).
- You are still limited in how large a texture can be in a graphics card - going past 131072 x 65536 might break your code (so remember to check the limit on the card with the `findDevices` code that we are providing!).
- Speed of access is still not as good as the shared memory - try to use each one in appropriate circumstances.

Using surface memory, 1 of 3

And yet another option is to use the surface memory, which is pretty much like the texture memory, but instead of being read-only, we can also write to it.

As mentioned before, since it is cached, if you write to it, memory fetches are not guaranteed to give you the latest value unless you make them after the kernel finishes or a memory copy concludes, so you might get undefined - or out of date-data.

Using surface memory, 2 of 3

The first difference is that we have to pass "cudaArraySurfaceLoadStore" as an argument to our cudaMallocArray call, like this:

Allocating the array to pass the data to surface memory

```
cudaArray* culnputArray;  
cudaMallocArray(&culnputArray , &channelDesc , width , height ,  
                cudaArraySurfaceLoadStore);
```

And that we also have to bind the surface memory like this:

Binding surface memory

```
cudaBindSurfaceToArray (inputSurfRef , culnputArray);
```

Using surface memory, 3 of 3

Then, in the GPU code, instead of using `tex2D` calls, we are going to use:

Allocating the array to pass the data to surface memory

```
// Read from input surface  
surf2Dread(&data, inputSurfRef, x * 4, y);  
// Write to output surface  
surf2Dwrite(data, outputSurfRef, x * 4, y);
```

You'll notice that we are multiplying by four in this case, when we do not need to do that when using the texture memory. It is because surface memory uses byte addressing - in the 2d case, the byte offset of the y-coordinate is calculated from the line pitch of the underlying CUDA array, that is why we only need to use it for the x coordinates.

Atomic functions, 1 of 7

Atomic functions are functions that can only be called from the device that perform one read-modify-write operation on one 32-bit or 64-bit word on the global or shared memory.

They are called "atomic" because those operations are guaranteed to be performed only by one thread until it is finished - no other thread can interfere with its execution.

Needless to say, that makes them incredibly useful for reduce operations (Min, Max, Add, etc...).

Atomic functions, 2 of 7

These functions are usually called passing the address and a value, as in the following ways:

Atomic functions

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address, unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);

int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address, unsigned int val);
```

Starting from Compute Capability 6.0 (which is the Pascal architecture), a double precision version of atomicAdd is available natively as well.

Atomic functions, 3 of 7

And this is the usage, which is pretty simple:

Atomic functions

```
__global__ void apply_add_one_atomic_operation( float *input, int Ntot) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    if ( idx <Ntot ) {
        atomicAdd (&(input[0]),1.0f);
    }
}
```

Just be careful with the return value, as it can be a bit confusing.

Atomic functions, 4 of 7

Something else that must be taken into account is that if you use Atomic operations on page-locked memory, they are only atomic for the device that you are using - so they are not guaranteed to be atomic from the point of view -nor are we guaranteed that the data will propagate to- other devices or the host.

Also, there is an Atomic function in particular that is worth commenting on over the rest - the atomic CAS (Compare and Swap); that can be used to implement almost any other atomic function that we might need.

Atomic functions, 5 of 7

Let us first have a look at our options when calling atomicCAS:

atomicCAS

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                      unsigned int compare,
                      unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

So we can use the `int` usage to read, compare and modify 32-bit (4 bytes) words, or we can use the `unsigned long long int` version to do the same with 64-bit (8 bytes) memory words.

Atomic functions, 6 of 7

So how do we implement our own atomic operations with atomicCAS?

atomicCAS

```
__device__ double atomicAddDouble(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val +
            __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

So we need to transform the values to long long ints and back, but it can be done.

Atomic functions, 7 of 7

But what does atomicCAS exactly do?

atomicCAS

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                      unsigned int compare,
                      unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

"reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old == compare ? val : old) , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap)."

CUDA Events and timing, 1 of 4

Another potentially useful tool that is part of the CUDA C specification are CUDA Events, that lets us monitor the performance of the algorithms that we run on the GPU, and provides us with accurate timing functions.

We might need them to check the performance of software that doesn't take a lot of time to run (for example, real-time operations that might take just microseconds to process).

CUDA Events and timing, 2 of 4

CUDA Events and timing

```
//create events
cudaEvent_t start, finish;
cudaEventCreate(&start);
cudaEventCreate(&finish);

//record events around kernel launch
cudaEventRecord(start, 0); // We use 0 here because it is the "default" stream
my_kernel <<< dimGrid , dimBlock >>>(...);
cudaEventRecord(finish, 0);

//synchronize
cudaEventSynchronize(start); // This is optional, we shouldn't need it
cudaEventSynchronize(finish); // This isn't – we need to wait for the event to finish

//calculate time
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, finish);
```

CUDA Events and timing, 3 of 4

But, as we are seeing, all the code that calls the events is being executed in the host - so why use them instead of our usual timing functions?

The CUDA events have better resolution (sub-microsecond) and lower latency than the CPU calls - think that it is not uncommon for GPUs to reach several thousands of frames per second when rendering some 3D scenes!

CUDA Events and timing, 4 of 4

One final warning about the CUDA events!

It can be quite computationally costly to set up and run (overheads seem to have increased in the last cuda specifications, instead of decreasing), particularly the `cudaEventCreate` calls - so be careful when you use them to measure lots of different parts of production code.

Compute-sanitizer and cuda-memcheck

The CUDA SDK provides us with an extremely useful debugging tool, called compute-sanitizer, which is the next iteration of a tool called cuda-memcheck.

This program works in a very similar way to electric fence - it will report any situation in which we go beyond the boundaries of the allocated memory.

To use it, you just have to call it in the command line when you run your cuda program, like this:

Compute-sanitizer and cuda-memcheck

```
/usr/local/cuda-12.0/bin/compute-sanitizer ./name_of_my_executable.exec  
OR  
cuda-memcheck ./name_of_my_executable.exec
```

Compute-sanitizer and cuda-memcheck

Be warned that, like many debuggers, it will make your code run much slower - specially in the case of programs that use shared memory.

Also, due to the way in which we store matrices as really large 1d chunks of memory, it might not warn us in all cases when going past memory boundaries.

It is, however, an extremely useful tool that makes cuda code debugging much easier.

Compute-sanitizer and cuda-memcheck

Also, it is worth noting that this is only the basic usage - we can use Compute-sanitizer and cuda-memcheck to find memory leaks, and to identify issues caused by race conditions on CUDA programs.

Compute-sanitizer and cuda-memcheck

The errors that we get from them can be precise or imprecise - the difference being that in the precise ones, we get a lot more information, such as which block, thread and even memory address triggered the error.

Also, the kinds of errors and leaks than can be detected are:

- Memory access errors
- Hardware exceptions
- Malloc/Free errors
- CUDA API errors
- cudaMalloc memory leaks
- Device Heap Memory leaks

Compute-sanitizer and cuda-memcheck

Memory access errors are the most common ones, and include trying to access memory positions that out of the allocated borders. Out of boundaries errors can be detected in the registers, global, local, shared or global atomic access memory.

Errors look like this (in this case, in the global memory):

Compute-sanitizer and cuda-memcheck

```
===== Invalid __global__ read of size 4
=====      at 0x000000c8 in addVectors.cu:6:add_arrays_gpu(float*, float*, float*, int)
=====      by thread (7,0,0) in block (2,0,0)
=====
=====      Address 0x70014005c is out of bounds
=====
=====      Saved host backtrace up to driver entry point at kernel launch time
=====      Host Frame:/usr/lib/libcuda.so (cuLaunchKernel + 0x3dc) [0xc9d5c]
=====      Host Frame:/usr/local/cuda/lib64/libcudart.so.5.0 [0xa18a]
=====      Host Frame:/usr/local/cuda/lib64/libcudart.so.5.0 (cudaLaunch + 0x17f) [0x2f4cf]
=====
=====      Host Frame://addVectors [0xdf8]
=====
=====      Host Frame://addVectors [0xcd2]
=====
=====      Host Frame://addVectors [0xd0e]
=====
=====      Host Frame://addVectors [0xb21]
=====
=====      Host Frame://lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xed) [0x2176d]
=====
=====      Host Frame://addVectors [0x889]
```

Compute-sanitizer and cuda-memcheck

Hardware exceptions are reported by the hardware.

Malloc/free errors are the ones that appear when we try to free twice, pass an invalid pointer to a free function call, or suffer heap corruption on the device. They can appear as well if we go past the limits on cudaMemcpy and then free the memory.

CUDA API errors are errors that are returned by any api call (say, for example, cudaMalloc or cudaMemcpy) - useful if you forgot to catch all possible return errors.

Compute-sanitizer and cuda-memcheck

However, being able to detect cudaMalloc and device heap memory leaks makes cuda-memcheck quite useful as well - to activate it, you have to run

Activating cuda-memcheck leak Checking

```
/usr/local/cuda-12.0/bin/compute-sanitizer --leak-check full ./name_of_my_executable.exec  
cuda-memcheck --leak-check full
```

and if you are using cuda-memcheck, in your code, add either:

Activating cuda-memcheck leak Checking

```
cudaDeviceReset();
```

if you are using the CUDA run time API or:

Compute-sanitizer and cuda-memcheck

Activating cuda-memcheck leak Checking

```
cuCtxDestroy();
```

If you are using the using the CUDA driver API. The memory leak reports look like this:

Activating cuda-memcheck leak Checking

```
===== Leaked 72 bytes at 0x700140200
=====     Saved host backtrace up to driver entry point at cudaMalloc time
=====     Host Frame::usr/lib/libcuda.so (cuMemAlloc_v2 + 0x266) [0xd67b6]
=====     Host Frame::usr/local/cuda/lib64/libcudart.so.5.0 [0x2371f]
=====     Host Frame::usr/local/cuda/lib64/libcudart.so.5.0 [0xa8b4]
=====     Host Frame::usr/local/cuda/lib64/libcudart.so.5.0 (cudaMalloc + 0x17a) [0x3013a]
=====     Host Frame::sharedMemory [0x9da]
=====     Host Frame::lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xed) [0x2176d]
=====     Host Frame::sharedMemory [0x899]
```

Advanced CUDA technology

In this chapter, we are going to go past the basic CUDA programming to talk about available libraries, debugging and profiling; cover a few more topics that are more advanced, and look at upcoming technologies and might just have become available.

The Thrust library

Thrust is a CUDA parallel library based on the C++ Standard Template Library (STL).

If you already know your way around the STL library, it is an extremely quick and effective way to get code running on the GPU.

It is even more similar to Boost, so if you have experience with it, it is a much more natural transition than CUDA C.

The Thrust library: Installation

Thrust being a template library, it is extremely easy to install - in fact, it comes with any install of CUDA 4.0+.

In case you still need to update it for any reason - you just have to extract the contents of the zip file in the right folder:

/usr/local/cuda/include/ on Linux and Mac OSX
C:\CUDA\include\ on Windows

The only thing left is to add the proper includes in your code, which will look like this:

Thrust sample includes

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
```



The Thrust library: Add vectors in Thrust

thrustAddVectors.cu

```
int main(void) {
    int i, N = 18;
    // Allocate vectors on host memory
    thrust::host_vector<float> a_vec(N);
    thrust::host_vector<float> b_vec(N);
    thrust::host_vector<float> c_vec(N);
    // Initialize vectors on the host
    thrust::host_vector<float>::iterator iter;
    iter = a_vec.begin(); i=0;
    for (iter = a_vec.begin(); iter < a_vec.end(); iter++) {
        *iter = (float)(i+1);
        i++;
    }
    iter = b_vec.begin(); i=0;
    for (iter = b_vec.begin(); iter < b_vec.end(); iter++) {
        *iter = (float)(i);
        i++;
    }
}
```

The Thrust library: Add vectors in Thrust

thrustAddVectors.cu

```
// Copy data from host memory to device memory
thrust::device_vector<float> a_gpu_vec = a_vec;
thrust::device_vector<float> b_gpu_vec = b_vec;

// Add arrays a and b
thrust::transform(    a_gpu_vec.begin() , a_gpu_vec.end() ,
                    b_gpu_vec.begin() , b_gpu_vec.begin() , thrust::plus<float>());

// Transfer data back to host
thrust::copy(b_gpu_vec.begin() , b_gpu_vec.end() , c_vec.begin());
```

Much simpler and cleaner, isn't it? Why are not using this all the time, then?

The Thrust library: Add vectors in Thrust

It turns out -for starters- that we cannot pass `thrust::device_vector` to our kernels - we have to turn them to standard pointers to device memory in the following way:

thrustAddVectors.cu

```
// Copy data from host memory to device memory
thrust::device_vector<float> a_gpu_vec = a_vec;
thrust::device_vector<float> b_gpu_vec = b_vec;
thrust::device_vector<float> c_gpu_vec = c_vec;
// Convert the thrust::device_vector<float> to float* in device memory
float *a_gpu_ptr = thrust::raw_pointer_cast(&(a_gpu_vec[0]));
float *b_gpu_ptr = thrust::raw_pointer_cast(&(b_gpu_vec[0]));
float *c_gpu_ptr = thrust::raw_pointer_cast(&(c_gpu_vec[0]));
// Add arrays a and b, store result in c
add_arrays_gpu<<<dimGrid, dimBlock>>>(a_gpu_ptr, b_gpu_ptr, c_gpu_ptr, N);
// This is the same as:
//thrust::transform(a_gpu_vec.begin(), a_gpu_vec.end(),
//                  b_gpu_vec.begin(), b_gpu_vec.begin(), thrust::plus<float>());
```

So... if the algorithm cannot be run as a

`thrust::transform`, we have to go back to normal CUDA C!

The Thrust library: Add vectors in Thrust

And, if we need to convert a cuda float array into a `thrust::device_vector`, we have to do it the following way:

thrustAddVectors.cu

```
// Allocate array device_a on device
float *device_a;
cudaMalloc ((void **) &device_a, sizeof(float)*N);
// Wrap raw pointer with a device_ptr
thrust::device_ptr<float> device_pointer_a(device_a);
// Add arrays a and b, store result in c
thrust::transform(device_pointer_a, device_pointer_a+N,
                  device_pointer_b, device_pointer_b, thrust::plus<float>());
// This is the same as:
//thrust::transform(a_gpu_vec.begin(), a_gpu_vec.end(),
//                  b_gpu_vec.begin(), b_gpu_vec.begin(), thrust::plus<float>());
```

So... again: if the algorithm cannot be run as a `thrust::transform`, we have to go back to normal CUDA C!

The Thrust library: Advantages

High level: Thrust code is certainly faster to write, cleaner and easier to read than CUDA C.

Simplicity: Lots of effort has been put into simplifying some often used algorithms, for example reduction operations.

Free and Open Source - under the Apache Version 2.0 License, which is quite liberal.

The Thrust library: Advantages

Highly portable: Since it is a template library, it is composed of header files, so it can be used on any platform and OS, as long as you have a nvcc compiler.

Customizable: Once you know it, you can customize it easily. Be careful with the licensing terms if you re-distribute, though.

Compatible with STL (Standard Template Library): If you are a C++ user, you can use your experience with the STL library to your advantage.

The Thrust library: Caveats and limitations

No proper multidimensional support: Like in the Standard Template Library (STL), if you need to use matrices or 3d matrices, you are by no means guaranteed that the memory is going to be contiguous, so you'll still have to handle your 3d vectors as a very long 1d one.

Asynchronous support: CUDA Streams were only added from version 1.8.

It is a template library: there is no dynamic library to point to, so any update in the Thrust library forces you to recompile your code, instead of keeping the same binaries.

The Thrust library: Caveats and limitations

It is C++ after all - which is great news if you are proficient in it already, but it will take quite some time to master if you are not.

Complex problems implemented in Thrust might actually be worse than without - the same can happen with STL (for example: VBOs).

Performance: The performance of many applications - even common ones such as reduce - is still far from perfect.

The Thrust library: An extension of STL, 1 of 6

We are not going to go deeply into C++, as that would require a different course, but let us explain a few things.

The main difference is that we have to store our data in vectors, instead of the usual C arrays; and they look a bit different, but their main advantage is that they behave in pretty much the same way, in the sense that we can access and modify elements by using the operator [] and that -at least in the 1d case- the memory is guaranteed to be contiguous.

They are what is commonly called a sequence container.

The Thrust library: An extension of STL, 2 of 6

From: <http://www.cplusplus.com/reference/vector/vector/>

stl vectors

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

The Thrust library: An extension of STL, 3 of 6

stl vectors

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity (see `push_back`).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and `forward_lists`), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and `forward_lists`.



The Thrust library: An extension of STL, 4 of 6

It is always worth mentioned that the elements of a vector can be accessed by using an integer index, as usual:

stl vector indexing

```
unsigned int ui;
for (ui=0;ui<stl_vector.size();ui++) {
    stl_vector[ui]=1;
}
```

Or by using iterators, that look like this:

stl vector indexing

```
vector<int>::iterator myIterator;
for(myIterator = stl_vector.begin(); myIterator != stl_vector.end(); myIterator++) {
    (*myIterator)=1;
}
```

The Thrust library: An extension of STL, 5 of 6

Careful, though! Iterators are not the pointers that we are used to handle, and this:

stl vector indexing

```
unsigned int ui;  
for (ui=0;ui<stl_vector.size();ui++) {  
    stl_vector[ui]++;  
}
```

Is totally different from this:

stl vector indexing

```
vector<int>::iterator myIterator;  
for(myIterator = stl_vector.begin(); myIterator != stl_vector.end(); myIterator++) {  
    *myIterator++;  
}
```

The Thrust library: An extension of STL, 6 of 6

This is because, when we do:

stl vector indexing

```
*myIterator++;
```

The operator "++" comes into place before the * takes effect, so we are affecting the iterator **before** the memory that is pointed to by the iterator is. We could solve this by using:

stl vector indexing

```
(*myIterator)++;
```

instead.

The Thrust library: Sample sort and comparison with the stl library, 1 of 3

But let's look at some more code samples with thrust, and how it compares with the very mature (it has been around since 1994!) stl library.

thrustSortRandomVectors.cu

```
// generate 32M random numbers on the host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

std::vector<int> stl_vector;
stl_vector.resize(h_vec.size());
for (unsigned int ui=0;ui<stl_vector.size();ui++) {
    stl_vector[ui]=h_vec[ui];
}
```

The Thrust library: Sample sort and comparison with the stl library, 2 of 3

thrustSortRandomVectors.cu

```
// transfer data to the device
thrust::device_vector<int> d_vec = h_vec;

// sort data on the device (846M keys per second on GeForce GTX 480)
gettimeofday(&thrustStart, NULL);
thrust::sort(d_vec.begin(), d_vec.end());
gettimeofday(&thrustEnd, NULL);

// sort data on the host
gettimeofday(&stlStart, NULL);
std::sort (stl_vector.begin(), stl_vector.end());
gettimeofday(&stlEnd, NULL);

// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

The Thrust library: Sample sort and comparison with the stl library, 3 of 3

Looking at results on a GTX680, doing several runs (with the -O4 and -fast-math options):

thrustSortRandomVectors.cu

```
GPU time , with thrust: 82454 CPU time , with stl: 2.40728e+06 speedup: 29.1955
GPU time , with thrust: 82424 CPU time , with stl: 2.40742e+06 speedup: 29.2078
GPU time , with thrust: 82486 CPU time , with stl: 2.40809e+06 speedup: 29.1939
GPU time , with thrust: 82461 CPU time , with stl: 2.40753e+06 speedup: 29.1959
GPU time , with thrust: 82440 CPU time , with stl: 2.40752e+06 speedup: 29.2033
GPU time , with thrust: 82295 CPU time , with stl: 2.4074e+06 speedup: 29.2533
```

Which isn't that bad, considering that the stl library is quite mature and the algorithm is not too parallel friendly... but it is still far from what we would expect.

The Thrust library: Sample reduce and comparison with the stl library, 1 of 9

Let's look at a thrust reduce operation now - we generate the vectors in the same way:

thrustReduce.cu

```
// generate 128M random numbers on the host
thrust::host_vector<long int> h_vec(32 << 22);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

std::vector<long int> stl_vector;
stl_vector.resize(h_vec.size());
for (ui=0;ui<stl_vector.size();ui++) {
    stl_vector[ui]=h_vec[ui];
}
```

The Thrust library: Sample reduce and comparison with the stl library, 2 of 9

thrustReduce.cu

```
// transfer data to the device
thrust::device_vector<long int> d_vec = h_vec;

// reduce data on the device
gettimeofday(&thrustStart, NULL);
thrustSum = thrust::reduce(d_vec.begin(), d_vec.end(), (long int) 0, thrust::plus<long int>());
gettimeofday(&thrustEnd, NULL);

// reduce data on the host
gettimeofday(&stlStart, NULL);
stlSum=0.0;
for (ui=0;ui<stl_vector.size();ui++) {
    stlSum+=stl_vector[ui];
}
gettimeofday(&stlEnd, NULL);

// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

The Thrust library: Sample reduce and comparison with the stl library, 3 of 9

Looking at results on a GTX680, with size 1024*1024 (again, with the `-O4` and `-fast-math` options):

thrustReduce.cu

```
GPU time , with thrust: 563 total time on the GPU, with stl: 221 speedup: 2.54751
GPU time , with thrust: 564 total time on the GPU, with stl: 222 speedup: 2.54054
GPU time , with thrust: 576 total time on the GPU, with stl: 222 speedup: 2.59459
GPU time , with thrust: 564 total time on the GPU, with stl: 223 speedup: 2.52915
GPU time , with thrust: 563 total time on the GPU, with stl: 223 speedup: 2.52466
GPU time , with thrust: 576 total time on the GPU, with stl: 222 speedup: 2.59459
```

Yet again, considering that the stl library is quite mature and the algorithm is, again, not too parallel friendly... it is way far from what we would expect.

The Thrust library: Sample reduce and comparison with the stl library, 4 of 9

How does this compare with some other reduce implementations? For example, we can consider the naive Atomic reduction kernel:

thrustReduceComparison.cu

```
__global__ void sumAtomicReduceVectorToScalarGPU ( int *vectorToReduce ,
                                                 int *reducedScalar ,
                                                 int N) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;

    if (idx<N) {
        atomicAdd ( reducedScalar , vectorToReduce[ idx ] );
    }
    return ;
}
```

The Thrust library: Sample reduce and comparison with the stl library, 5 of 9

And we can consider also the naive Atomic reduction on the shared and then the global memory kernel:

thrustReduceComparison.cu

```
__global__ void sumAtomicSharedReduceVectorToScalarGPU ( int *vectorToReduce ,
                                                       int *reducedScalar ,
                                                       int N) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;

    __shared__ int sdata[1];
    if (threadIdx.x==0) {
        sdata[0]=0;
    }
    __syncthreads();
    if (idx<N) {
        atomicAdd(&sdata[0], vectorToReduce[idx]);
    }
    __syncthreads();
    if (threadIdx.x==0) {
        atomicAdd(reducedScalar , sdata[0]);
    }
    return ;
}
```

The Thrust library: Sample reduce and comparison with the stl library, 6 of 9

Let's run some numbers again with a 1024^*1024 problem size:

thrustReduceComparison.cu

```
Total time on the CPU, with stl: 582
Total time on the GPU, with thrust: 230 speedup for thrust: 2.53043
Total time on the GPU, with atomic: 7 speedup for atomic: 83.1429
Total time on the GPU, with atomicShared: 4 speedup for atomicShared: 145.5
```

So we can easily do much better than thrust! And what happens if we apply the problem to 2^*1024^*1024 ?

thrustReduceComparison.cu

```
Total time on the CPU, with stl: 1121
Total time on the GPU, with thrust: 287 speedup for thrust: 3.90592
Total time on the GPU, with atomic: 8 speedup for atomic: 140.125
Total time on the GPU, with atomicShared: 4 speedup for atomicShared: 280.25
```

Our speedups get even better!

The Thrust library: Sample reduce and comparison with the stl library, 7 of 9

With 4*1024*1024:

thrustReduceComparison.cu

```
Total time on the CPU, with stl: 2234
Total time on the GPU, with thrust: 366 speedup for thrust: 6.10383
Total time on the GPU, with atomic: 11 speedup for atomic: 203.091
Total time on the GPU, with atomicShared: 5 speedup for atomicShared: 446.8
```

With 8*1024*1024:

thrustReduceComparison.cu

```
Total time on the CPU, with stl: 4484
Total time on the GPU, with thrust: 492 speedup for thrust: 9.11382
Total time on the GPU, with atomic: 11 speedup for atomic: 407.636
Total time on the GPU, with atomicShared: 6 speedup for atomicShared: 747.333
```

The Thrust library: Sample reduce and comparison with the stl library, 8 of 9

With $16 \times 1024 \times 1024$:

thrustReduceComparison.cu

```
Total time on the CPU, with stl: 8945
Total time on the GPU, with thrust: 733 speedup for thrust: 12.2033
Total time on the GPU, with atomic: 12 speedup for atomic: 745.417
Total time on the GPU, with atomicShared: 5 speedup for atomicShared: 1789
```

With $32 \times 1024 \times 1024$:

thrustReduceComparison.cu

```
Total time on the CPU, with stl: 17862
Total time on the GPU, with thrust: 1221 speedup for thrust: 14.629
Total time on the GPU, with atomic: 12 speedup for atomic: 1488.5
Total time on the GPU, with atomicShared: 5 speedup for atomicShared: 3572.4
```

3572 speedup versus 14.629, and we are not even using particularly smart algorithms yet!

The Thrust library: Sample reduce and comparison with the stl library, 9 of 9

So, the usual - the thrust library is not the solution to all our CUDA problems; in fact it is not even close to that.

If you know C++ and the STL library, and you can move the operations that you are planning to run on the GPU into C++ (usually template code) it is still worth looking into, but other than that, it is a good idea to keep a reasonable amount of skepticism about it.

The CURAND library

CURAND is a CUDA parallel library for pseudo-random number generating

It provides you will the tools to generate pseudo-random and quasi-random numbers

It is composed by two clearly differentiated parts: Calling a kernel provided by the library to quickly generate a large array of numbers (called the host library), and device functions to be able to generate numbers on each thread (called the device header file).

The CURAND library

To use either the host library or the device header file, you just need to install the CUDA SDK, and then add in your Makefile:

CURAND linking

`-lcurand`

when compiling and linking.

The CURAND library

To use the host library, we must add:

CURAND host library include

```
// This is the include for the host curand
# include <curand.h>
```

But then, we also need create and initialize a
curandGenerator_t structure:

CURAND host library curandGenerator_t declaration

```
// Declare and initialize the pseudo-random number generator
curandGenerator_t generator;
curandCreateGenerator (&generator,CURAND_RNG_PSEUDO_DEFAULT);
```

The CURAND library

To feed the generator, we can call:

CURAND host library seeding

```
// Set the seed  
curandSetPseudoRandomGeneratorSeed (generator,1234ULL);
```

And to extract, for example, N numbers from a normal distribution, we can call:

CURAND host library drawing numbers from a normal distribution

```
// Generate N random numbers from an uniform distribution  
curandGenerateUniform (generator, c_device , N);
```

The CURAND library

There are plenty of pseudo-random number generators and distributions that can be called - please refer to the documentation to find out about what the options are.

So with the host library, we can generate a large amount of pseudo-random numbers in the global memory very quickly. Then, we can bring them back to the host (ie, to the CPU) and use them somewhere else.

It is also worth mentioning that creating and seeding the generator usually is more computationally expensive than drawing the random numbers, so we might want to reuse it if we can.

The CURAND library

For the device includes, first of all we must add a different include:

CURAND device include

```
// This is the include for the device curand
# include <curand_kernel.h>
```

When we use CURAND in device threads, however, we have to call kernels ourselves, so we'll have to set up a grid as usual, as well as to allocate an array of `curandState` structures that we will allocate in the global memory.

CURAND device `curandState` definition

```
// Allocate deviceStates
curandState *deviceStates;
cudaMalloc (( void ** ) &deviceStates , N * sizeof ( curandState ));
```

The CURAND library

Then we write our own initialize and seed kernel (that will be run once per thread) and will be similar to this:

CURAND device initialization and seeding kernel

```
__global__ void setup_kernel ( curandState * state ) {
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    // Each thread gets same seed, a different sequence number , no offset
    curand_init (1234 , idx , 0, &state[idx]);
}
```

And that we will call the usual way:

CURAND device initialization and seeding kernel calling

```
// Kernel to setup the pseudo-random numbers on each thread
setup_kernel <<<dimGrid, dimBlock>>> ( deviceStates );
```

The CURAND library

Then, to be able to generate random numbers on each one of our threads, we just need to pass the `curandState` array to our kernel as we would any other global memory array:

CURAND device sample kernel usage

```
__global__ void generate_kernel ( curandState *state , float* results ) {
    int idx = blockIdx.x*blockDim.x+threadIdx.x;

    // Copy state to registers for efficiency – will be more useful
    curandState localState = state [idx];

    // Generate pseudo – random floats from an uniform distribution
    results[idx] = (float)(curand_uniform (& localState ));

    // Copy state back to global memory
    state [idx] = localState ;
}
```

The CURAND library

Let us compare the host cuRand library with the random number generator provided by thrust :

thrustRandomVectorsComparison.cu

```
Total time with thrust: 552.866 with cuRand :59.5185
Total time with thrust: 553.216 with cuRand :59.1049
Total time with thrust: 554.478 with cuRand :59.5632
Total time with thrust: 554.813 with cuRand :59.0194
Total time with thrust: 553.533 with cuRand :58.8404
Total time with thrust: 553.250 with cuRand :59.3486
Total time with thrust: 555.377 with cuRand :59.0594
```

So it is unsurprisingly faster.

The cuBLAS library

The cuBlas is cuda's version of the BLAS library, and one of the reasons for the success of CUDA over other GPGPU alternatives, such as OpenCL.

It has the advantage that, like with cuRAND, it can be called from the host (and from .c or .cpp files), so again, we have the possibility do not having to worry about things such as setting up the grid, which makes it a lot easier to use for people who just want some easy way to call solvers.

However, we still need to allocate memory in the global memory of the device ourselves, so using cuBLAS still requires some knowledge of CUDA, after all.

The cuBLAS library

Quite obviously, it isn't perfect either.

One problem is that it maximizes FORTAN compatibility, up to the point that it uses column-major storage and 1-based indexing, so a little bit of tweaking is required when porting Fortan code to C.

The cuBLAS library

It also must be taken into account that, from the CUBLAS 4.0 version, the API has changed a lot - in fact, the header file that we have to include is a different one.

The old header file, which is kept to support compatibility, can be accessed by using the include:

cublas before 4.0

```
#include "cublas.h"
```

And the new one, by using instead:

cublas after 4.0

```
#include <cublas_v2.h>
```

We are going to focus in the new API.

The cuBLAS library

First of all, we will still have to allocate memory in the device by using `cudaMalloc`, which by default cannot be understood by g++ or gcc, so we need to add:

cublas

```
#include <cuda_runtime.h>
```

And still compile with `nvcc` to get things compiling and linking in .cpp and .c files.

We will also need to include the following linking flag:

cublas linking flag

```
-lcublas
```

The cuBLAS library

We will also have to use a variable from the `cublasStatus_t` struct:

cublasStatus_t declaration

```
cublasStatus_t status;
```

Every cuBlas library function call will return a `cublasStatus_t` value, which means that we can easily check for errors in the following way:

cublasStatus_t testing

```
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "cuBLAS error (my function)\n");
    return EXIT_FAILURE;
}
```



The cuBLAS library

Also, we need to define and use a handler variable that comes from the `cublasHandle_t` struct:

cublas handlers

```
cublasHandle_t handle
```

This variable has to be initialized before we call any other cuBlas function call, and has to be passed explicitly to every subsequent library function call. It is specific for each host thread and device, which means that we can keep several of them, when running code which is parallel on the host, so we can execute cuBlas on multiple GPUs.

The cuBLAS library

We can initialize our handler in the following way:

cublas handler initialization

```
status = cublasCreate(&handle);
// Make sure that it was initialized properly
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! CUBLAS initialization error\n");
    return EXIT_FAILURE;
}
```

Which means that we need to use our status variable even to start our handler.

The cuBLAS library

Also, we need to destroy our handler too, in the following way:

cublas handler destruction

```
status = cublasDestroy(handle);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! shutdown error (A)\n");
    return EXIT_FAILURE;
}
```

The cuBLAS library

Then we have to handle the allocation on the host and the device as usual:

cuBlas allocation

```
host_matrix = (float *)malloc(numberOfElements * sizeof(float));  
cudaMalloc((void **)&device_matrix , numberOfElements * sizeof(float))
```

The cuBLAS library

However, we can use `cublasSetVector` to transfer data to the device:

cublasSetVector

```
status = cublasSetVector(numberOfElements, sizeof(float)),
         host_matrix , 1, device_matrix , 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! device access error (read C)\n");
    return EXIT_FAILURE;
}
```

Where the 1s mark the spacing between consecutive elements on `host_matrix` and `device_matrix` (useful if we have arrays of structures).

The cuBLAS library

And we can also perform the device to host copy operation by calling `cublasGetVector`:

cublasGetVector

```
status = cublasGetVector(numberOfElements, sizeof(float)),
         device_matrix, 1, host_matrix, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! device access error (read C)\n");
    return EXIT_FAILURE;
}
```

And the 1s here, state again the spacing between consecutive elements.

The cuBLAS library

To handle the rest of the cuBlas function calls, you just have to call the proper cuBlas function, which will look like this (we'll use `cublasSgemm` as an example):

cublasSgemm

```
status = cublasSgemm(handle , CUBLAS_OP_N, CUBLAS_OP_N,
                      matrixA_rows , matrixB_columns ,matrixA_columns ,
                      &alpha ,
                      device_matrixA , matrixA_Id ,
                      device_matrixB , matrixB_Id ,
                      &beta ,
                      device_matrixC , matrixC_Id );
```

Usage will depend on the function, and the only cuBlas specific variables that we are using here is `handle`.

The cuBLAS library

You will find a lot more information about the cuBlas library in

CUBLAS manual online documentation

<http://docs.nvidia.com/cuda/cublas/index.html>

Also, you have to take into account that the double precision performance is not as good as the single precision one, specially if you are using a GeForce card, and that double checking the accuracy of your results is almost mandatory with this libraries.

cuFFT

cuFFT is the library provided by the CUDA SDK to be able to support a range of FFT transformations by using GPUs.

It comes with the SDK as usual, so installation happens at the same time.

It is also based on FFTW.

There is also another simpler option: to use cuFFTW, which is a version of cuFFT that is designed to enable FFTW users to start using Nvidia cuda gpus with minimal effort (so it is similar to the host version of cuRand).

cuFFT

It is optimized for problem sizes that can be written in as:

$$2^a * 3^b * 5^c * 7^d$$

It is an $O(n \log n)$ algorithm, either if the problem matches the previous expression or not.

Accepts both complex and real-valued input and output.

Supports 1D, 2D and 3D transforms, batch execution, double precision, in-place and out-of place transforms, strided layout.

The API is thread safe, so you can call it from multiple threads.

Also allows streamed execution - as in the asynchronous usage of CUDA that we will see later on.

Usage is quite similar to the CUBLAS library, in this case we will have the following `cufftResult` structure to provide us with a status variable that all cuFFT functions will return:

cufftResult_t

```
typedef enum cufftResult_t {
    CUFFT_SUCCESS          = 0, // The CUFFT operation was successful
    CUFFT_INVALID_PLAN     = 1, // CUFFT was passed an invalid plan handle
    CUFFT_ALLOC_FAILED     = 2, // CUFFT failed to allocate GPU or CPU memory
    CUFFT_INVALID_TYPE     = 3, // No longer used
    CUFFT_INVALID_VALUE     = 4, // User specified an invalid pointer or parameter
    CUFFT_INTERNAL_ERROR    = 5, // Driver or internal CUFFT library error
    CUFFT_EXEC_FAILED       = 6, // Failed to execute an FFT on the GPU
    CUFFT_SETUP_FAILED      = 7, // The CUFFT library failed to initialize
    CUFFT_INVALID_SIZE      = 8, // User specified an invalid transform size
    CUFFT_UNALIGNED_DATA    = 9, // No longer used
} cufftResult;
```

cuFFT

Also, in this case we will have the following `cufftHandle` structure to provide us with a handler that we will pass as an argument to the cuFFT functions:

cufftHandle structure

```
cufftHandle plan;
```

The handler is usually called `plan` because it sets up preconfigured internal building blocks to reduce the execution time as much as possible, given the configuration of both the GPU and the problem.

Again, we will have to initialize a `cufftHandle` variable that is going to depend on the FFT that we are planning to implement, and could be any of the following:

cufftHandle structure initialization

```
cufftPlanMany(); // Creates a plan supporting batched input and strided data layouts  
cufftPlan1D(); // Create a simple plan for a 1D transform  
cufftPlan2D(); // Create a simple plan for a 2D transform  
cufftPlan3D(); // Create a simple plan for a 3D transform
```

Each one of the `cufftPlan` functions have a fairly large amount to arguments that can be used to configure the transform for optimal usage and performance, the largest being `cufftPlanMany`:

cufftHandle structure initialization

```
cufftResult cufftPlanMany(cufftHandle *plan, int rank, int *n, int *inembed,  
    int istride, int idist, int *onembed, int ostride,  
    int odist, cufftType type, int batch);
```

Whereas the `cufftPlan#d` functions have less arguments, hence the "simple plan" mentioned before:

cufftHandle structure initialization

```
cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch);
cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type);
cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type);
```

And we will have to destroy our plan, just like we did with the CUBLAS handler:

cufftHandle structure initialization

```
cufftResult cufftDestroy(cufftHandle plan);
```

Execution will be conducted by calling the appropriate function, which can be any of the following:

cuFFT execution functions

```
// Complex to complex, single precision
cufftResult cufftExecC2C(cufftHandle *plan, cufftComplex *idata,
                         cufftComplex *odata, int direction);
// Complex to complex, double precision
cufftResult cufftExecZ2Z(cufftHandle *plan, cufftDoubleComplex *idata,
                         cufftDoubleComplex *odata, int direction);

// Real to complex, single precision
cufftResult cufftExecR2C(cufftHandle *plan, cufftReal *idata, cufftComplex *odata);
// Real to complex, double precision
cufftResult cufftExecD2Z(cufftHandle *plan, cufftDoubleReal *idata,
                         cufftDoubleComplex *odata);

// Complex to real, single precision
cufftResult cufftExecC2R(cufftHandle plan, cufftComplex *idata, cufftReal *odata);
// Complex to real, double precision
cufftResult cufftExecZ2D(cufftHandle plan, cufftComplex *idata, cufftReal *odata);
```

cuFFT

Again, you can find a lot more information about cuFFT in
cuFFT manual online documentation

<http://docs.nvidia.com/cuda/cufft/index.html>

Yet again, you have to take into account that the double precision performance is not as going to be as good as the single precision one, specially if you are using a GeForce card, and that you should always double check the results that you obtain with the ones that you get calculating on the CPU.

cuSparse

Another useful library is CUSPARSE - a linear algebra library of functions used for handling sparse matrices and vectors.

It is, again, installed at the same time as the CUDA runtime, and designed to be called from C, C++ or Fortran.

Yet again, there is a legacy version and a newer version, that we can access by using:

cuSparse include

```
#include <cusparse_v2.h>
```

cuSparse

Again, we have a `cusparseStatus_t` enum from which we will recover the return status of all cuSparse library function calls:

cuSparse include

```
enum    cusparseStatus_t {
CUSPARSE_STATUS_SUCCESS = 0,
CUSPARSE_STATUS_NOT_INITIALIZED = 1,
CUSPARSE_STATUS_ALLOC_FAILED = 2,
CUSPARSE_STATUS_INVALID_VALUE = 3,
CUSPARSE_STATUS_ARCH_MISMATCH = 4,
CUSPARSE_STATUS_MAPPING_ERROR = 5,
CUSPARSE_STATUS_EXECUTION_FAILED = 6,
CUSPARSE_STATUS_INTERNAL_ERROR = 7,
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED = 8
}
```

cuSparse

Usage is similar to cuBLAS, in the sense that we can call level 1 to 3 operations with different data types (float, double, cuComplex, etc), but we get as well conversion functions - which means that we can convert our (usual) matrices into a compressed format, which will reduce their size, and then (maybe) get them to fit in the GPU when we wouldn't be able to fit a larger object with 0s in many positions.

Also, the functions are thread safe from the host point of view, and can be used with asynchronous calls (cuda Streams).

cuSparse

This library has even more functions (as it includes the conversion functions, as well as preconditioners, etc) than cuBlas, so we are not going to discuss it in detail.

Documentation can be found here:

cuSparse documentation

<http://docs.nvidia.com/cuda/cusparse/index.html>

Other CUDA libraries

There are other useful CUDA libraries that we are going to mention.

The first one that we are going to mention is CULA tools, which is a proprietary library for linear algebra, that can be found in:

cula tools

<http://www.culatools.com/>

Other CUDA libraries

CULA is split in two main libraries, CULA Dense and CULA Sparse.

If you want to use it, you need to buy a license, unless you can apply for an academic one (which is free).

There is also a free trial of the CULA Dense part.

Other CUDA libraries

There are also many other alternatives, such as:
MAGMA, that is designed for heterogeneous GPU-based
architectures (so it tries to mix MPI, OpenMP and CUDA).

NVIDIA Performance Primitives (NPP) that provides
you with image and signal processing functions

NVIDIA CUDA Video Encoder (NVCUVENC) to create
video streams with various codecs

GPU AI – Path Finding and GPU AI for Board
Games, to provide some cuda tools for videogames

Other CUDA libraries

NVIDIA CUSP, an Open Source C++ library for graph, sparse linear algebra and generic parallel algorithms.

IMSL Fortran Numerical Library, which offers mathematical and statistical tools.

AccelerEyes ArrayFire, with functions for math, signal and image processing and statistics, that also provides interfaces for C, C++, Fortran and Python.

Triton Ocean SDK, that provides real time visualization of the ocean and bodies of water.

Other CUDA libraries

rCUDA, middleware library that allows machines in a cluster without GPUs to offload compute to machines with GPUs (a bit like a master and slave system).

cuSOLVER, which offers various sparse and dense linear solvers.

cuDNN (from Nvidia again), which offers a fast and easy way of implementing Deep Neural Networks.

nvJPEG (from Nvidia yet again), for JPEG encoding.

Other CUDA libraries

Random123, another library of counter-based random number generators (mostly for statistics and Monte Carlo, not recommended for cryptography, though).

Kokkos, which focuses on multi-platform development - so you can recompile the same code to be able to run it on GPUs, Xeon Phis, or many-core CPUs.

OpenACC, which allows us to mark parts of the code (in a way similar to OpenMP), to also allow it to run in massively parallel hardware.

CUB, which is another template library (similar to thrust in that sense).

Other CUDA libraries

Charm++, another library that can provides CPU and GPU parallelization.

IBM XL C/C++ compiler, which transforms OpenMP directives into more GPU friendly code.

CUDA-aware MPI, which allows you to MPI_Send and receive GPU buffers.

OpenMP, you can use OpenMP as well, but it has the problem that CPU OpenMP and GPU OpenMP are different.

Other CUDA libraries

SYCL, a high level c++ library for heterogeneous computing that uses templates and lambda functions to run GPU code.

libcu++, which extends the STL C++ library into cuda.

Using the Nvidia Nsight Compute

Another extremely useful tool that nvidia provides to help programmers achieve a better performance with their code is the Nsight Compute, a GUI that provides information about the performance of kernels that are run on a GPU.

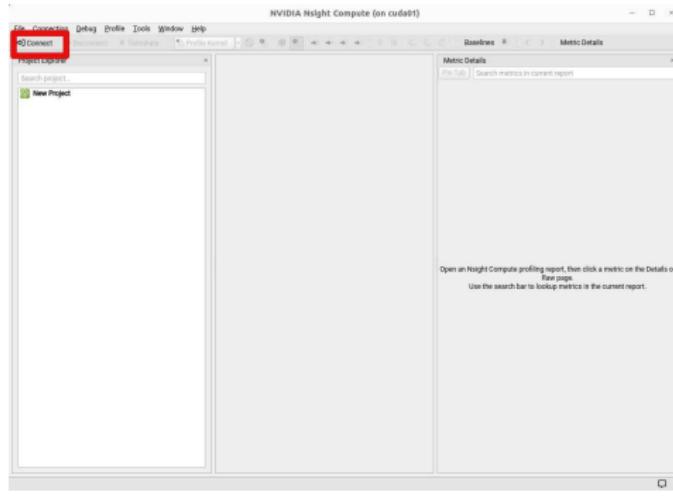
To use it, you just have to have the CUDA SDK installed. Then you can run it by typing: `ncu-ui` or `/usr/local/cuda-12.2/nsight-compute-2023.2.1/ncu-ui` on Linux and Mac OSX Or clicking on the appropriate icon under windows.

We can run it with the binaries in production mode (as in, with optimizations flags such as `-O4`), but to get the most information out of the profiler, we need to compile them with `-g`.

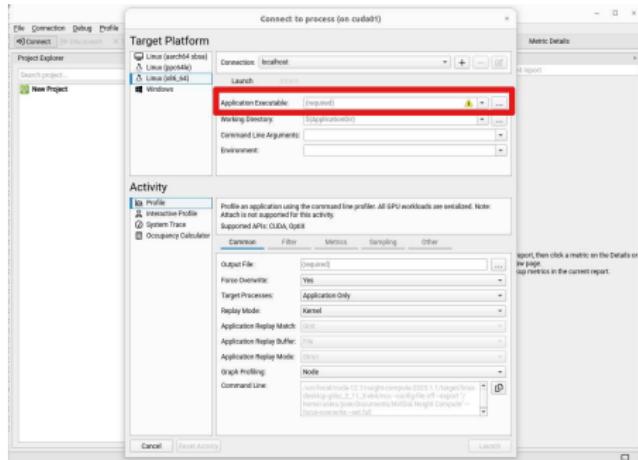
Using the Nvidia Nsight Compute

Once you get the window of the Compute Visual Profiler, you will have to create a project. You can do it by following the next steps:

- Click on "Connect"

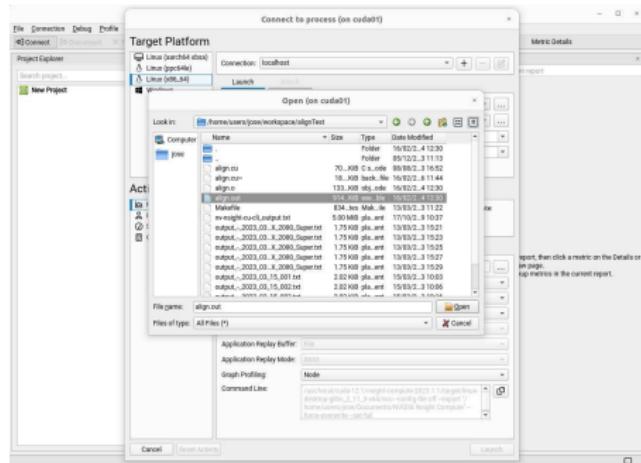


Using the Nvidia Nsight Compute



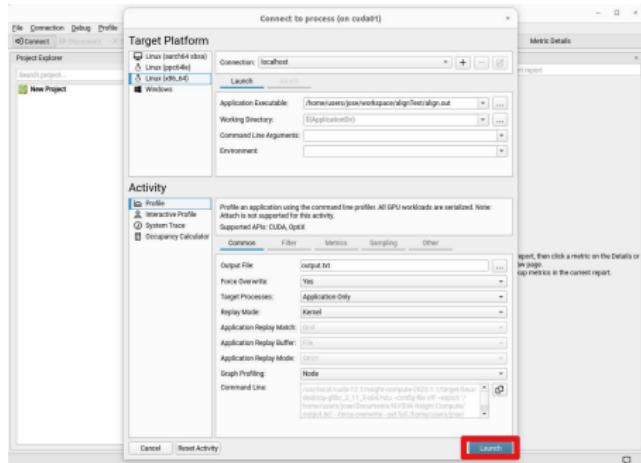
Choose the executable that you want to profile. You can just click on the button to the right that that will let you choose a file in the filesystem.

Using the Nvidia Nsight Compute



You can click on "Open" to select it.

Using the Nvidia Nsight Compute



Fill up the working directory for that executable and the output file path and name. Type in any command line arguments that you might need. Finally, click on the "Launch" button.

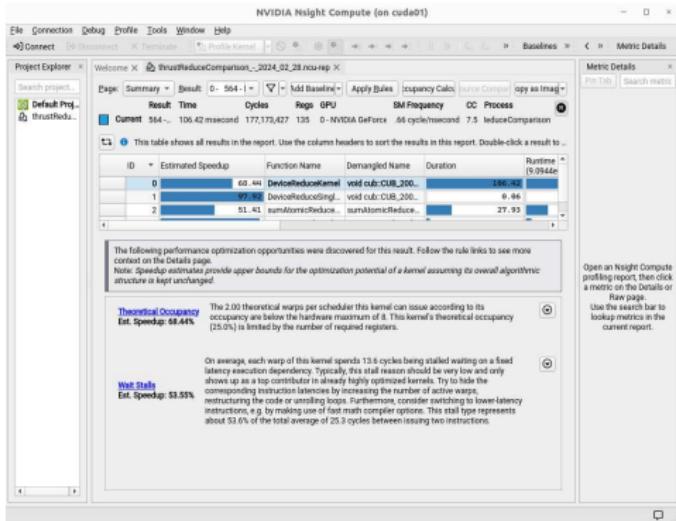
Using the Nvidia Nsight Compute

The profiler should now start running the program several times
- this can take a lot of time depending on what you are doing.



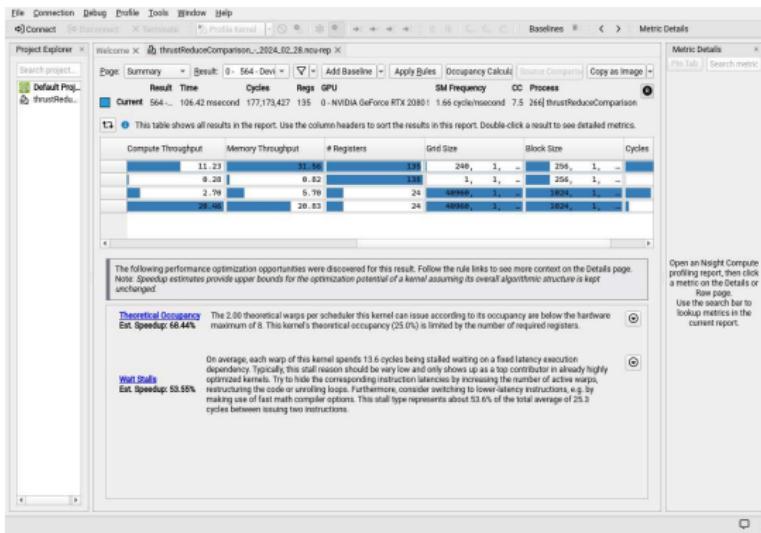
Using the Nvidia Nsight Compute

Once it finishes, you will see the "Summary", which already gives you quite a bit of useful information:



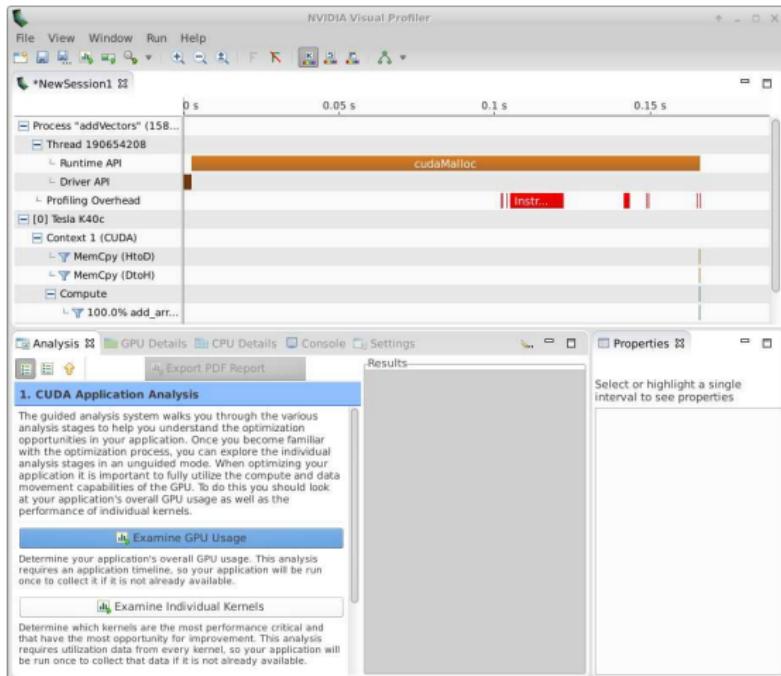
Using the Nvidia Nsight Compute

And even more so in the fields to the right:



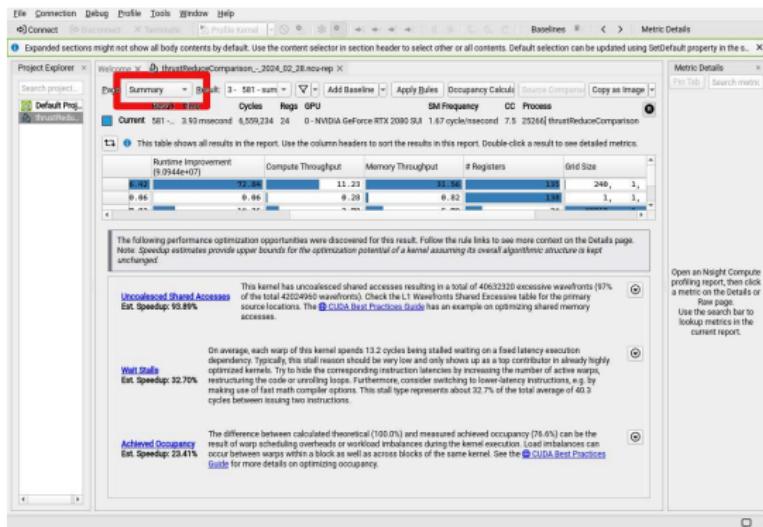
Using the Nvidia Nsight Compute

We get almost everything that we need compared to the now deprecated Nvidia Visual Profiler, except for the very useful timeline graphs - we will see later how to get them from the Nsight Systems profiler later:



Using the Nvidia Nsight Compute

Changing the menu to the other options, such as Session, Details, Source, etc, will provide us with more information. But for now, let's stick to Summary:



Using the Nvidia Nsight Compute

We can see the analysis of different kernels by choosing them in this menu or from this list:

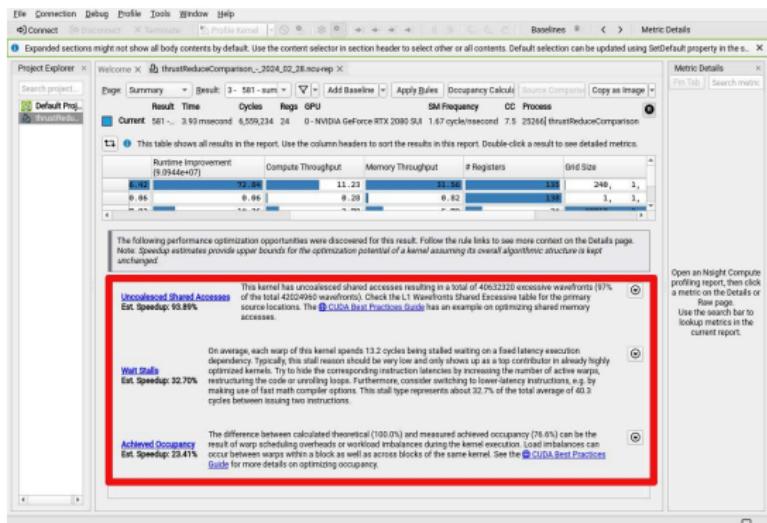
The screenshot shows the Nvidia Nsight Compute interface with the following details:

- Project Explorer:** Shows a project named "Default Project".
- Welcome:** Displays the "thrustReduceCorporation" report.
- Summary:** Shows metrics like Current (581 ms), SM Frequency (6,559,234), and CC Process (0 - NVIDIA GeForce RTX 2080 SUPER).
- Table:** A table showing performance metrics for various results. One row is highlighted with a red box:

Result	Cycles	Memory	GPU	SM Frequency	CC Process
Current	581	~3.51 ms	6,559,234	24	0 - NVIDIA GeForce RTX 2080 SUPER
- Optimization Opportunities:** Lists several items:
 - Uncoalesced Shared Accesses:** Est. Speedup: 92.89%
 - Warp Stalls:** Est. Speedup: 32.70%
 - Achieved Occupancy:** Est. Speedup: 23.41%
- Metric Details:** A panel on the right showing a metric table and a tooltip for "Achieved Occupancy".

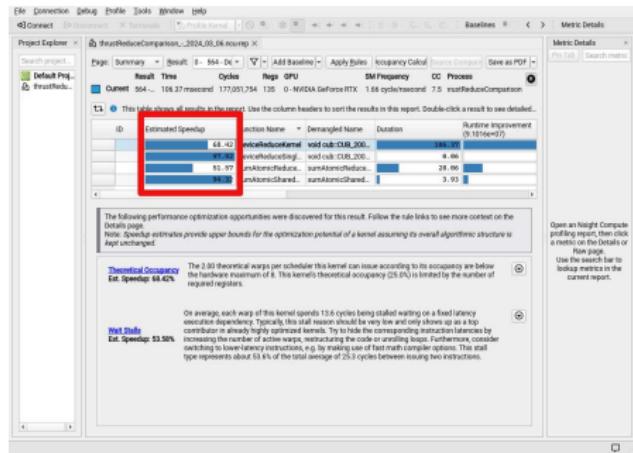
Using the Nvidia Nsight Compute

And we can get already useful information in this part of the window:



Using the Nvidia Nsight Compute

The Estimated Speedup gives us an estimation of the speedup that we could achieve by addressing the most impactful performance issues (which isn't always possible):



Using the Nvidia Nsight Compute

Common complains from the profiler are lack of efficiency when loading from and writing to the global memory:

Low Global Memory Load Efficiency, Low Global Memory Store Efficiency, Long Scoreboard Stalls, Local and Global Throttle Stalls, Wait Stalls.

Most of the times, this all refers to the same problem - the way that we using the global memory might not be the most efficient way to get the best performance out of the memory transfers - but again, depending on the algorithm, sometimes this is unavoidable.

Overlapping compute and memory transfers might help - we will see later on how to setup asynchronous memory transfers and executions.

Using the Nvidia Nsight Compute

If we are using libraries that set the block and grid size for us, we might get:

Inefficient Block Size

A classic problem what we have discussed before - since the threads are organized and launched in groups or WARP_SIZE, you want the number of threads per block to be a multiple of that number (which, as mentioned before, is so far 32 in all CCs).

Using the Nvidia Nsight Compute

Issue Slot Utilization - the schedulers in the streaming multiprocessors are issuing fewer instructions to the warps than they potentially could, because the warps are not ready to execute their next instruction yet. This could be due to warp divergence (different threads of the warp not doing the same thing at the same time), having to wait for memory to arrive, etc.

Long Scoreboard - too much time waiting for a memory operation on local, global, texture or surface operation.

Lg Throttle - The "Lg" stands for Local and Global memory. The warps have are spending too much time waiting for memory to arrive.

Using the Nvidia Nsight Compute

Thread Divergence - there is a very considerable divergence between the instructions of the 32 threads of a warp.

Uncoalesced Shared Accesses - the reads and writes to the shared memory are not particularly well aligned.

Wait - active warps are spending too much time waiting before executing the next instruction. As usual, this is because they are not ready to execute due to execution or memory dependencies.

Using the Nvidia Nsight Compute

Instruction replays

Replayed Instructions (%) This gives the percentage of instructions replayed during kernel execution. Replayed instructions are the difference between the numbers of instructions that are actually issued by the hardware to the number of instructions that are to be executed by the kernel. Ideally this should be zero. This is calculated as $100 * (\text{instructions issued} - \text{instruction executed}) / \text{instruction issued}$

Global memory replay (%) Percentage of replayed instructions caused due to global memory accesses. This is calculated as:

$100 * (\text{l1 global load miss}) / \text{instructions issued}$

Local memory replay (%) Percentage of replayed instructions caused due to local memory accesses. This is calculated as:

$100 * (\text{l1 local load miss} + \text{l1 local store miss}) / \text{instructions issued}$

Shared bank conflict replay (%) Percentage of replayed instructions caused due to shared memory bank conflicts. This is calculated as:

$100 * (\text{l1 shared conflict}) / \text{instructions issued}$

Using the Nvidia Nsight Compute

So quite often, the problem here is that the instructions have to be issued in 32 threads at a time and not all the threads are ready to go at the same time.

And what the profiler tells us is that frequently, a instruction that was waiting for global, shared or local memory was processed for some but not for the full 32 threads of a warp, which means that the same warp will have to replay (hence the name) that instruction when the data finally becomes available for the other threads.

Using the Nvidia Nsight Compute

This can be solved sometimes by optimizing the memory accesses - for example, changing our code in such a way that the 32 threads access contiguous memory addresses, so less thread divergence is caused by memory calls.

In cases in which this is not possible, due to the nature of the problem, different approaches can be considered - such as cache'ing data that is going to be requested multiple times in memory with a smaller access time.

We will have a look later at memory coalescence.

Using the Nvidia Nsight Compute

Let us have a look at some of the fields, and explain what they are about:

Duration are self-explanatory - but you need to keep in mind that running a program through the profiler makes it run slower, so in many cases, kernels that would run a lot faster if they were executed without the profiler might show a much lesser efficiency when checking those fields (specially the Duration one).

Same for grid and block size. And regarding Compute and Memory Throughput, the more, the better.

Using the Nvidia Nsight Compute

Moving on, if we go back to the summary and we look at the list of kernel calls that were done:

The screenshot shows the Nvidia Nsight Compute interface with the 'Summary' page selected. A table of kernel metrics is highlighted with a red border. The columns include Duration, Runtime Improvement, Compute Throughput, Memory Throughput, # Registers, and Grid Size.

Duration	Runtime Improvement	Compute Throughput	Memory Throughput	# Registers	Grid Size
184.93	86.82	11.40	31.29	135	249, 1, -
0.06	0.06	0.28	0.51	138	1, 1, -
27.85	27.35	2.72	5.74	24	4096, 1, -
3.92	3.69	28.56	29.94	24	4096, 1, -

The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page. Note: Speedup estimations are experimental and might overestimate optimization potential!

Issue Slot Utilization
Est. Speedup: 92.00%

Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 12.6 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 8 warps per scheduler, this kernel allocates an average of 2.00 active warps per scheduler, but only an average of 0.08 warps were issued per scheduler. This means that the scheduler is issuing less warps than it needs to. Every warp that does not have an eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the [E_Warp_Status Statistics](#) and [E_Source_Counters](#) sections can help too.

Theoretical Warp
Est. Speedup: 75.00%

The 2.00 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 8. Use the [E_Occupancy](#) section to identify what limits this kernel's theoretical occupancy.

Wait
Est. Speedup: 53.71%

On average, each warp of this kernel spends 13.6 cycles being stalled waiting on a fixed latency execution dependency. Typically, this stall reason should be very low and only shows up as a top contributor in already highly optimized kernels. Try to hide the communicating instruction instances by increasing the number of active warps, restructuring the code or unrolling loops. Furthermore, consider switching to low-latency instructions, e.g. by making use of fast math compiler options. This stall type represents about 53.7% of the total average of 25.3 cycles between issuing two instructions.

Open an Nsight Compute profiling report, then click a metric on the Details or Raw page.
Use the search bar to lookup metrics in the current report.

Using the Nvidia Nsight Compute

We can move to the "Details" tab:

The screenshot shows the NVIDIA Nsight Compute application window. The 'Details' tab is highlighted with a red box. The main pane displays a table of performance metrics for a kernel named '2 - 570 - sumAtomicReduceVectorTi'. The table includes columns for Dem. Raw, Cycles, Regs, GPU, SM Frequency, CC Process, and Grid S. Below the table, a message box lists optimization opportunities like 'Issue Slot Utilization' and 'Long Scoreboard'. On the right side, there's a 'Metric Details' panel with search and filter options.

Dem. Raw	Cycles	Regs	GPU	SM Frequency	CC Process	Grid S
void cub::CUB_200_	184.93	96.63	11.48	33.29	1.39	
void cub::CUB_200_	0.86	8.86	8.28	0.51	1.39	
sumAtomicReduce_	27.85	27.35	2.72	5.74	24	
sumAtomicShared_	3.92	3.69	38.66	28.94	24	

The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page.
Note: Speedup estimations are experimental and might overestimate optimization potential.

Issue Slot Utilization
Est. Speedup: 98.20%

Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 55.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 8 warps per block/device, this kernel allocates an average of 7.79 active warps per block/device, but only an average of 0.02 warps are issued per cycle. Eligible warps are a subset of active warps that are ready to issue their next instruction. Even though there are no eligible warps, no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the [Warp State Statistics](#) and [Source Counter](#) sections can help, too.

Long Scoreboard
Est. Speedup: 51.42%

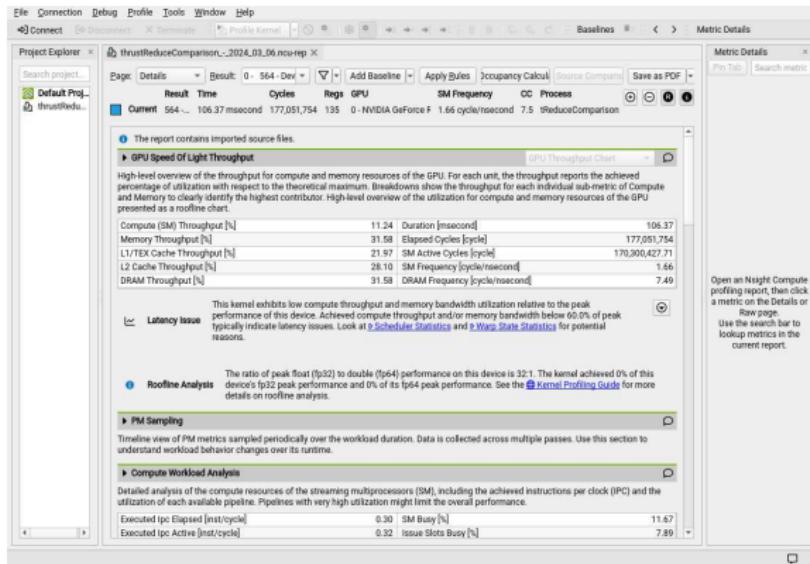
On average, each warp of this kernel spends 222.4 cycles being stalled waiting for a scoreboard dependency on a L1:TEX (local, global, surface, texture) operation. Find the instruction producing the data being waited upon to identify the culprit. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality (coalescing), or by changing the cache configuration. Consider using frequent data to shared memory. This stall type represents about 51.4% of the total average of 432.5 cycles between issuing two instructions.

Lat Throttle
Est. Speedup: 52.99%

On average, each warp of this kernel spends 152.4 cycles being stalled waiting for the L1 instruction queue for local and global (LG) memory operations to be not full. Typically, this stall occurs only when executing local or global memory instructions extremely frequently. Avoid redundant global memory accesses. Try to avoid using thread-local memory by checking if dynamically indexed arrays are declared in local scope, or if the kernel has excessive register pressure causing by spills. If

Using the Nvidia Nsight Compute

Which now gives us a lot of info about the kernel:



Using the Nvidia Nsight Compute

The screenshot shows the Nvidia Nsight Compute interface with a detailed analysis report for a kernel profile named "thrustReduceComparison_2024_03_06.ncu.rep".

Project Explorer: Shows the current project is "Default Proj..." and the selected file is "thrustReduceComparison_2024_03_06.ncu.rep".

Metric Details: A search bar at the top right allows for searching metrics.

Compute Workload Analysis: Detailed analysis of compute resources. Key metrics:

Metric	Value	Description
Executed ipc Elapsed [Inst/cycle]	0.30	SM Busy [%]
Executed ipc Active [Inst/cycle]	0.32	Issue Slots Busy [%]
Issued ipc Active [Inst/cycle]	0.32	

Low Utilization: Estimated Local Speedup: 88.33%

All compute pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the [Launch Statistics](#) and [Scheduler Statistics](#) sections for further details.

Memory Workload Analysis: Detailed analysis of memory resources. Key metrics:

Metric	Value	Description
Memory Throughput [Gbyte/second]	151.40	Mem Busy [%]
L1/TEX Hit Rate [%]	76.47	Max Bandwidth [%]
L2 Hit Rate [%]	98.79	Mem Pipes Busy [%]

Memory L2 Compression: The optional metric `lts__average_gcomp_input_sector_success_rate_pct` could not be found. Collecting it as an additional metric could enable the rule to provide more guidance.

Scheduler Statistics: Summary of the activity of the schedulers issuing instructions. Key metric:

Metric	Value	Description
Active Warps Per Scheduler [warp]	2.00	No Eligible [%]

Right Panel: Provides instructions for opening an Nsight Compute profiling report and using the search bar to look up metrics in the current report.



Using the Nvidia Nsight Compute

The screenshot shows the Nvidia Nsight Compute interface with a report titled "thrustReduceComparison_2024_03_06.ncvrep". The main window displays "Scheduler Statistics" and "Warp State Statistics".

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	2.00	No Eligible [%]	92.10
Eligible Warps Per Scheduler [warp]	0.08	One or More Eligible [%]	7.90
Issued Warp Per Scheduler	0.08		

Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 12.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 8 warps per scheduler, this kernel allocates only 2.00 active warps per scheduler, but only an average of 0.08 warps are eligible per cycle. Eligible warps are not yet active, but are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the [Warp State Statistics](#) and [Source Counters](#) sections can help, too.

Issue Slot Utilization
Est. Local Speedup: 68.42%

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	25.31	Avg. Active Threads Per Warp	31.99
Warp Cycles Per Executed Instruction [cycle]	25.31	Avg. Not Predicted Off Threads Per Warp	31.96

On average, each warp of this kernel spends 13.6 cycles being stalled waiting on a fixed latency execution dependency. Typically, this stall reason should be very low and only shows up as a top contributor in already highly optimized kernels. Try to hide the corresponding instruction latencies by increasing the number of registers, reorganizing the code or combining loops. Furthermore, consider:

Open an Nsight Compute profiling report, then click a metric on the Details or Raw page.
Use the search bar to lookup metrics in the current report.



Using the Nvidia Nsight Compute

The screenshot shows the Nvidia Nsight Compute interface with the following details:

- Project Explorer:** Shows a project named "Default Proj." containing "thrustReduceComparison".
- Metric Details:** A search bar and a list of metrics.
- Report Content:**
 - Warp State Statistics:** An analysis of warp states. It shows that each warp spends 13.6 cycles on average, which is a stall. The report suggests increasing active warps or switching to lower-latency instructions.
 - Wait Stalls:** Estimated speedup is 53.58%.
 - Warp Stall:** Check the [Warp Stall Sampling \(All Samples\)](#) table for top stall locations.
 - Instruction Statistics:** Statistics of executed low-level assembly instructions (SASS). It notes a narrow mix of instruction types and their frequency.
 - NVLink Topology:** Shows logical NVLink connections with transmit/receive throughput.
 - NVLink Tables:** NVLink topology diagram.
- Bottom Navigation:** Includes back, forward, and search icons.

Using the Nvidia Nsight Compute

The screenshot shows the Nvidia Nsight Compute interface with the following details:

- Project Explorer:** Shows a project named "Default Proj..." containing "thrustRedu...".
- Metric Details:** A search bar at the top right.
- Report Title:** "thrustReduceComparison_2024_03_06.ncvrep X".
- Result Summary:** Current result: 564 - 106.37 msecound, 177,051,754 cycles, 135regs, GPU: nVIDIA GeForce F, SM Frequency: 1.66 cycle/nsecound, CC: 7.5, Process: tReduceComparison.
- NUMA Affinity:** Non-uniform memory access (NUMA) affinities based on compute and memory distances for all GPUs.
- Launch Statistics:** Summary of the configuration used to launch the kernel. It includes:

Grid Size	240	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	135	Static Shared Memory Per Block [byte/block]	44
Block Size	256	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	61,440	Driver Shared Memory Per Block [byte/block]	0
Waves Per SM	5	Shared Memory Configuration Size [kbyte]	32.77
- Occupancy:** Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance; however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	25	Block Limit Registers [block]	1
Theoretical Active Warps per SM [warp]	8	Block Limit Shared Mem [block]	128
Achieved Occupancy [%]	24.98	Block Limit Warps [block]	4
Achieved Active Warps Per SM [warp]	8.00	Block Limit SM [block]	16

Theoretical Occupancy: 2.00 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 8. This kernel's theoretical occupancy (25.0%) is limited by the number of required registers. Est. Speedup: 69.42%
- Source Counters:** Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

Using the Nvidia Nsight Compute

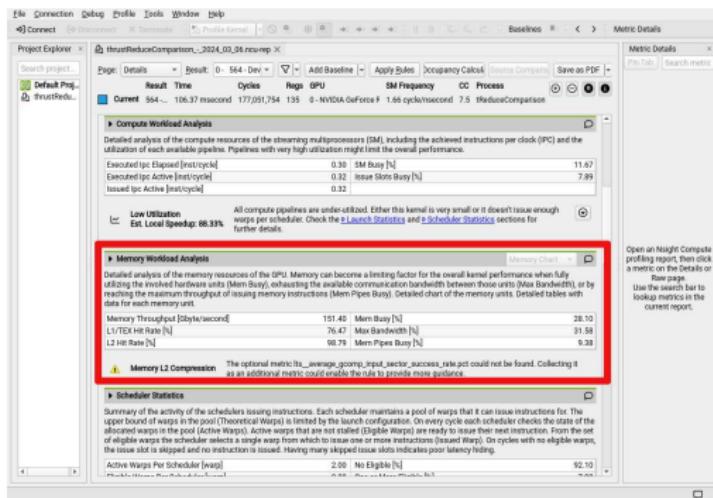
The Regs or Registers/Thread field gives us an idea about how many registers per thread our kernel is using - if we are not using many and the algorithm allows it, we might be able to get each thread to use more and increase performance that way.

Shared Memory Configuration will tell us how much shared memory we are using per block and how that memory is allocated - we will see later on how to dynamically allocate and use the shared memory.

Size and Throughput is reserved for memory transfers, and it shows the size of the memory being copied.

Using the Nvidia Nsight Compute

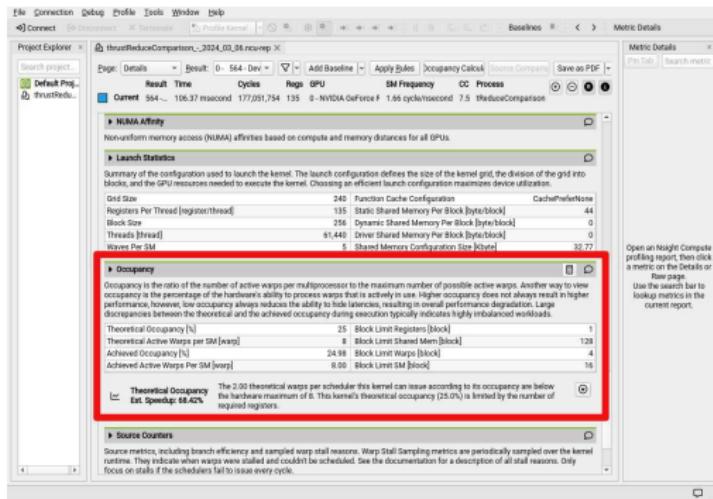
Memory gives us a breakdown of how efficient are our transfers in and out of the global memory.



We get measures for the Global Load Efficiency (as a ratio between requested to actual transactions), Global Store Efficiency (same) and DRAM Utilization (as a percentage of the theoretical peak bandwidth achievable and the real usage).

Using the Nvidia Nsight Compute

This is another part that is well worth having a look at:



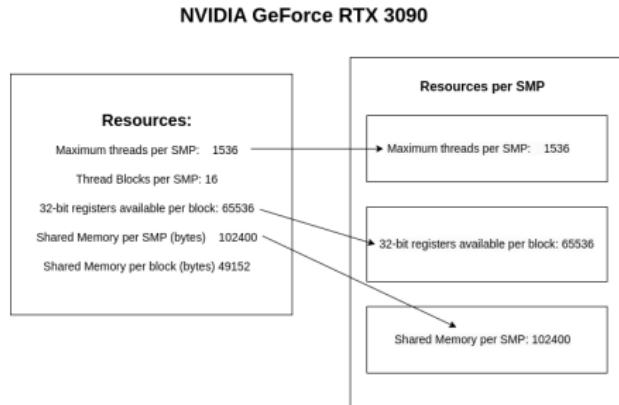
Using the Nvidia Nsight Compute

We get to finally describe Occupancy - the Achieved Occupancy is the "Ratio of average active warps per active cycle to the maximum number of warps supported on a multiprocessor"; and it is one of the main measures that we want to look at when improving the performance of our kernels.

Essentially, this means that a given card model has a maximum number of warps that can be active at a time - for example, a Fermi GPU has two warp schedulers and can have 48 active warps per Streaming Multiprocessor (or 1536 threads).

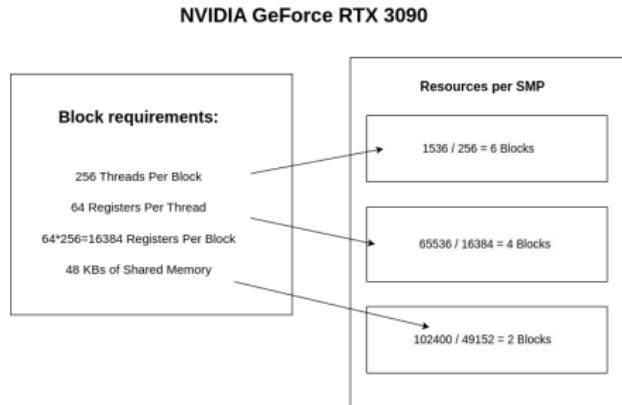
Using the Nvidia Nsight Compute

So let's assume that we look at the resources of each SM in a RTX 3090 card:



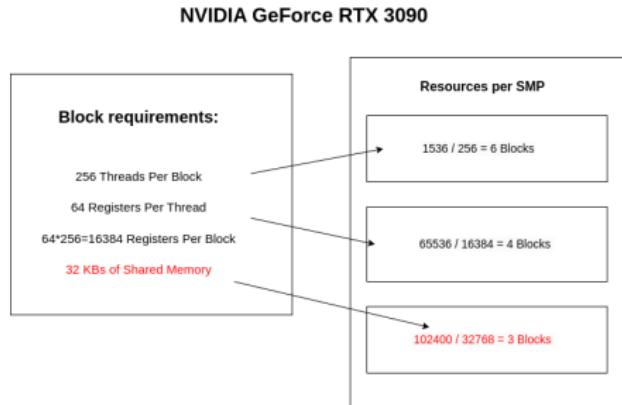
Using the Nvidia Nsight Compute

How many blocks can we keep active in a SM when it comes to each of those resources?



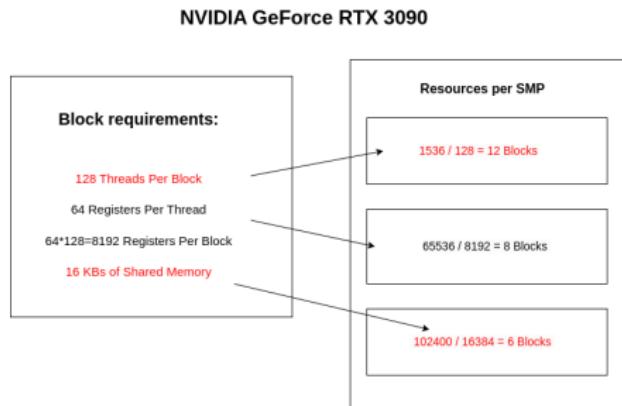
Using the Nvidia Nsight Compute

So if we reduce the amount of shared memory needed per block:



Using the Nvidia Nsight Compute

And if we reduce the amount threads per block (which would reduce as well the needed amount of shared memory):



Using the Nvidia Nsight Compute

Take into account that the architecture of the GPU might change the occupancy levels without anything else changing!

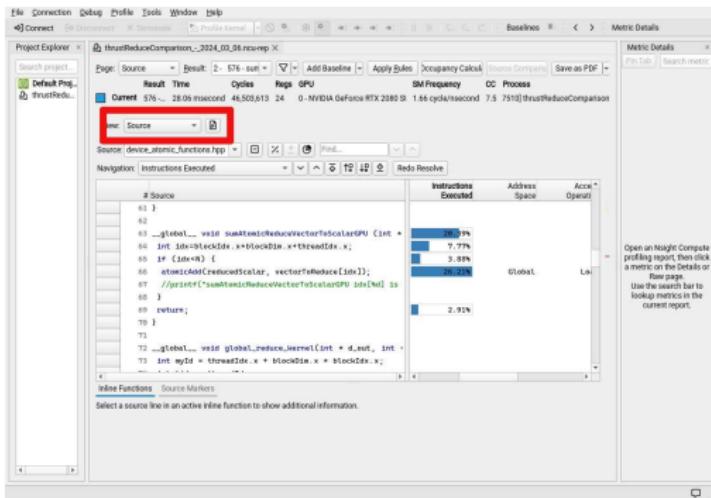
In Kepler GPUs, each multiprocessor has four warp schedulers, and each warp scheduler manages at most 16 warps, for a total of 64 warps per multiprocessor.

In Maxwell, each multiprocessor has four warp schedulers again, but since we have fewer cuda cores per multiprocessor (and the schedulers are independent of each other), we get a better ratio.

Usually, a higher occupancy means better performance, but that is not always the case - context switching (moving one warp from active but not executing to executing mode and back), while not very expensive, still causes some overheads.

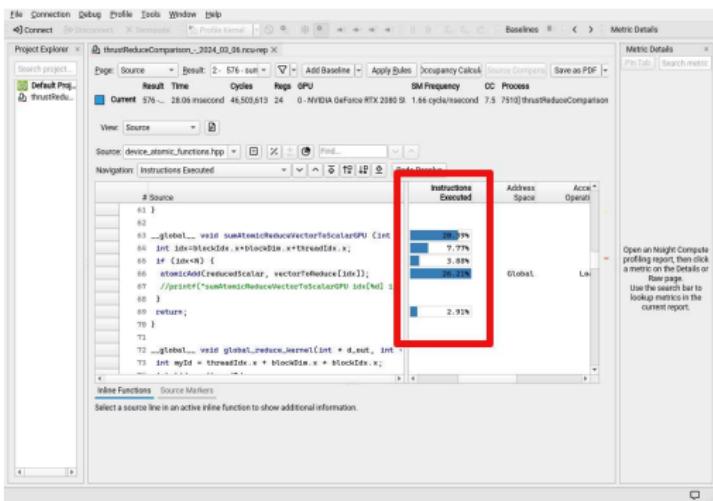
Using the Nvidia Nsight Compute

Another very useful thing that we can do is to examine the source code, which is done by selecting the "Source" tab - we can only see this if we compiled the source with -g, though!



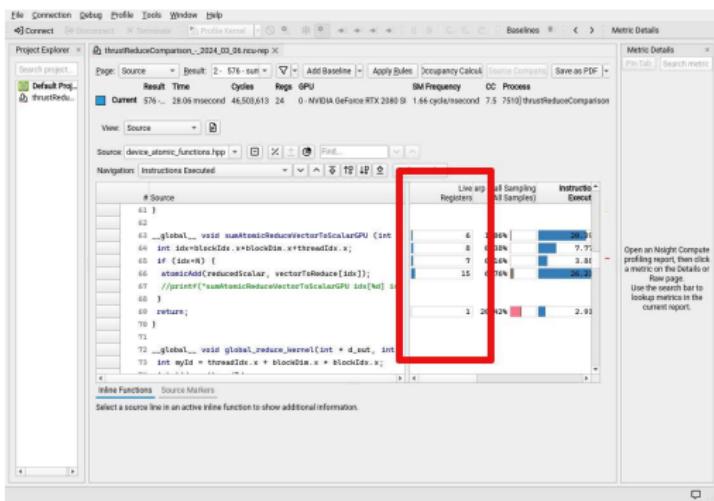
Using the Nvidia Nsight Compute

Then we can see which instructions are taking the most time by looking here:



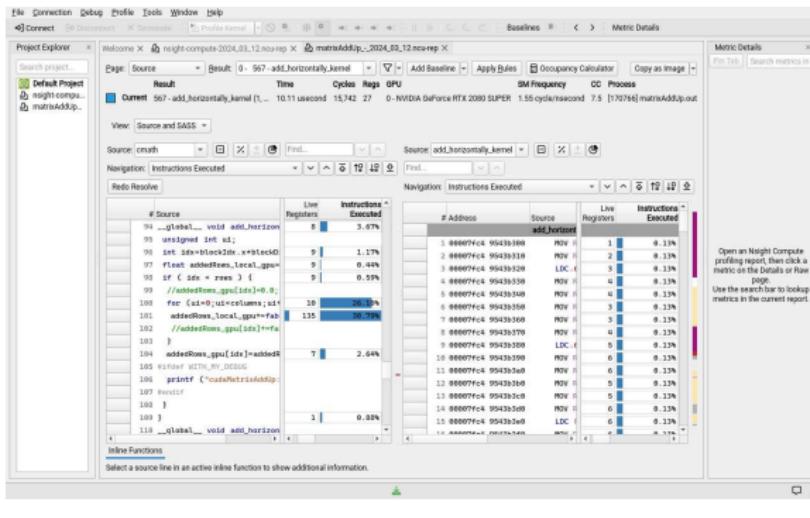
Using the Nvidia Nsight Compute

And we also check how much register space we are using if we move the slider to the left:



Using the Nvidia Nsight Compute

However! There is a weird bug in the most recent versions of the profiler that makes it report unusually high register usage for some operations such as fabs or divisions:



Using the Nvidia Nsight Compute

Fun fact - the version in cuda 12.0 seems to work just fine:

The screenshot shows the Nvidia Nsight Compute interface. At the top, there's a menu bar with File, Connection, Debug, Profile, Tools, Window, Help, and a toolbar with various icons. Below that is a Project Explorer window showing a single project named "test_ncsrep". The main area is titled "test_ncsrep" and displays the results of a performance analysis. It includes tabs for "Source" and "Instructions Executed". The "Instructions Executed" tab is active, showing two panes: one for the source code and one for the assembly instructions. The source code pane contains C++ code for a matrix addition kernel. The assembly pane shows the corresponding assembly code for the GPU. The assembly code is heavily annotated with performance metrics like SM Frequency, CC Process, and occupancy. The bottom of the interface has a toolbar with navigation icons.

```
#include <cuda.h>
#include <math.h>

__global__ void addVerticallyKernel(int *d_in, int *d_out, int rows, int columns) {
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int idz = blockIdx.z * blockDim.z + threadIdx.z;
    if (idy < rows && idz < columns) {
        for (int i=0; i<rows; i++) {
            for (int j=0; j<columns; j++) {
                d_out[idy * columns + idz] += d_in[i * columns + j];
            }
        }
    }
}
```

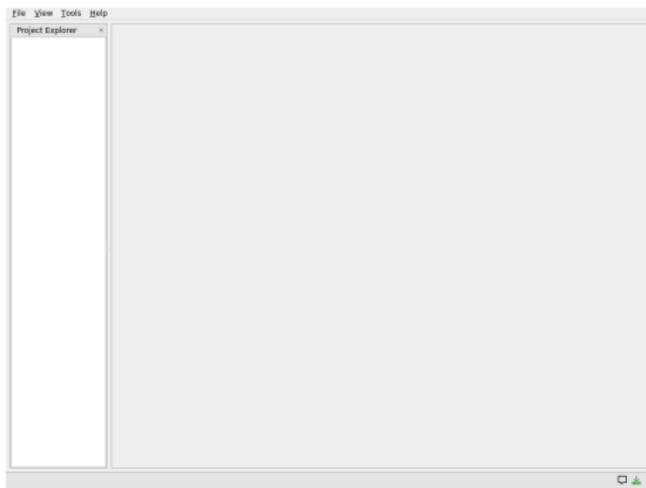
```
01 unsigned int ui;
02 int idy=blockIdx.y*32+threadIdx.y;
03 float addedColums_local,gpu;
04 if ( ui < columns ) {
05 //addedColums_gpu[ui]=0;
06 for (ui=0;ui<rows;ui++) {
07     addedColums_local+=ui;
08 //addedColums_gpu[ui]=ui;
09 }
10 addedColums_gpu[ui]=addedC;
11 #ifdef NV_DEBUG
12 printf ("casaMatrixAddUp:ad
13 #endif
14 }
15 }
```

Instructions Executed

#	Address	Source	Live Registers
1	00007f44_03442100	add_vertically_kernel	R1
2	00007f44_03442110		R2
3	00007f44_03442120		R3
4	00007f44_03442130		R4
5	00007f44_03442140		R5
6	00007f44_03442150		R6
7	00007f44_03442160		R7
8	00007f44_03442170		R8
9	00007f44_03442180		R9
10	00007f44_03442190		R10
11	00007f44_034421a0		R11
12	00007f44_034421b0		R12
13	00007f44_034421c0		R13
14	00007f44_034421d0		R14
15	00007f44_034421e0		R15
16	00007f44_034421f0		R16
17	00007f44_034421g0		R17
18	00007f44_034421h0		R18
19	00007f44_034421i0		R19

Using the Nvidia Nsight Systems

The other profiler that we can use to generate timelines is the NSight Systems, that we can launch by calling something like `/usr/local/cuda-12.3/nsight-systems-2023.3.3/bin/nsys-ui` :



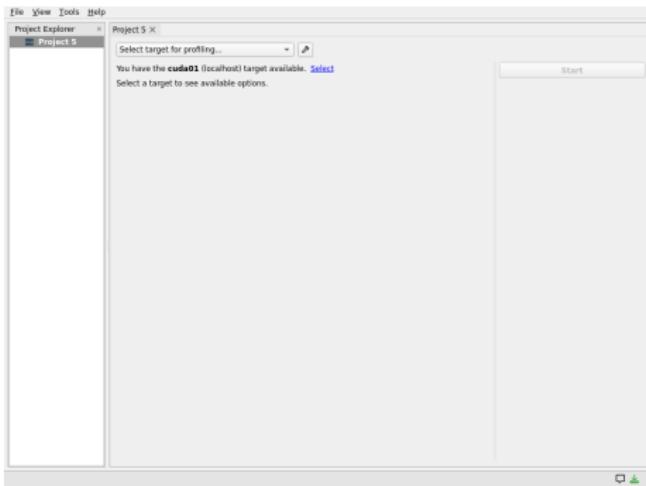
Using the Nvidia Nsight Systems

You will have to right click on the left frame and create a new project to get started:



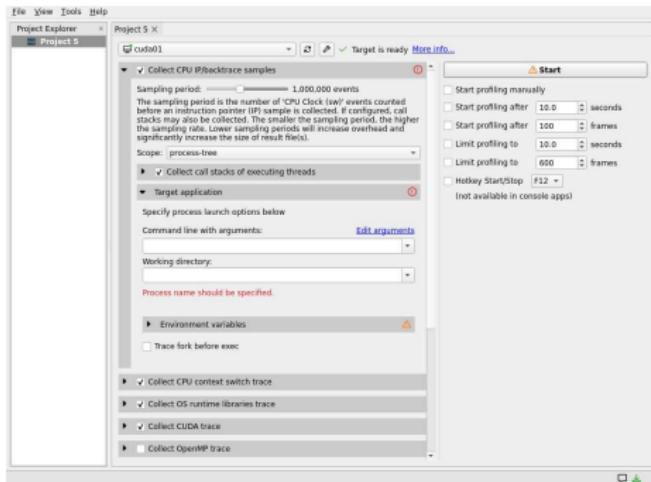
Using the Nvidia Nsight Systems

In Select target for profiling, choose the machine that you running things on:



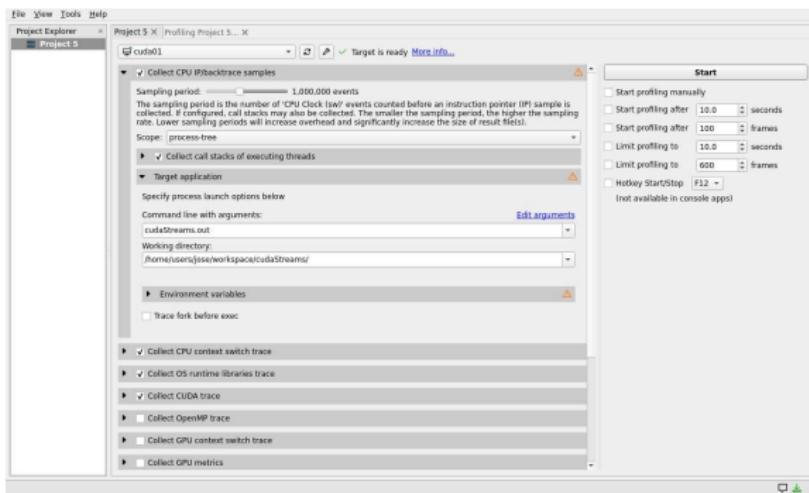
Using the Nvidia Nsight Systems

You will get lots of options - here you have to type the name of the executable and the path to the folder where it is located manually:



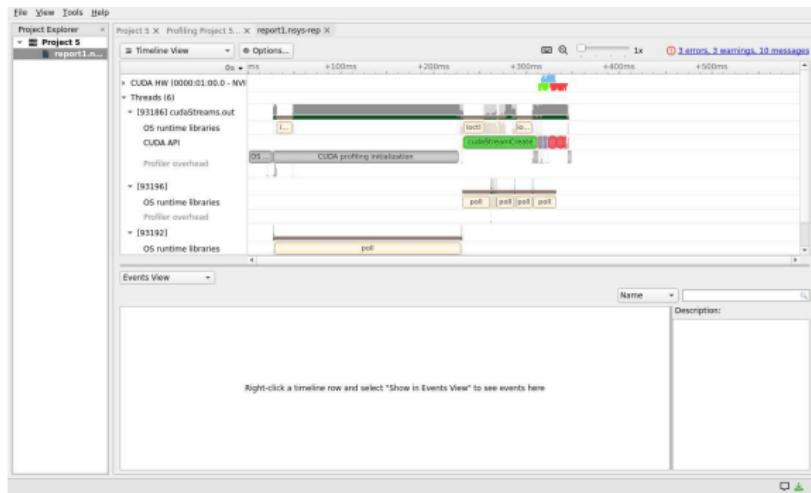
Using the Nvidia Nsight Systems

Once you have done that, just click Start to begin profiling:



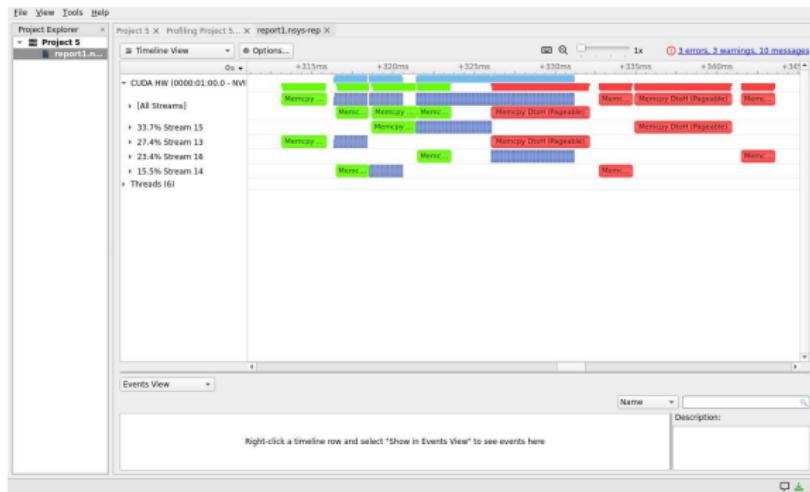
Using the Nvidia Nsight Systems

When you get the timeline, it looks like this, which isn't the best view:



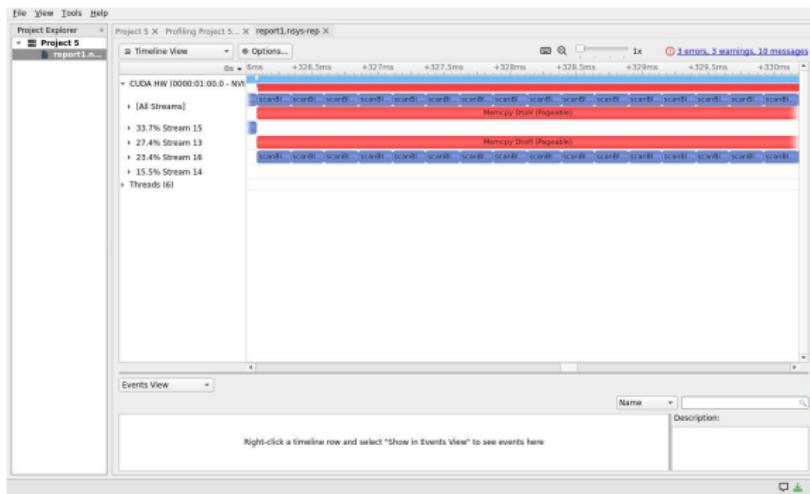
Using the Nvidia Nsight Systems

The information that we want is under CUDA HW. We can click on the arrows to hide the others, then zoom in with the mouse wheel or "Ctrl" + "+":



Using the Nvidia Nsight Systems

The very small blobs are actually calls, if we zoom in enough we can see what they are:



Using the Nvidia Profilers

Yet again, be careful when using both profilers, as the times that they give you might not be the same that we get when we are not profiling.

That means that the profiler could be pointing us in the wrong direction due to the way in which samples are taken during execution - specially if we are running the code after compiling it with `-g` so we can see the code.

Using Cuda-GDB

Another profiler-debugger that the Nvidia SDK provides is cuda-gdb, which is a version of the gdb profiler adapted to CUDA.

Using gdb is actually non-trivial, (in fact it would take a different course to cover that!) so we are going to recommend this option mostly if you already know how to use it.

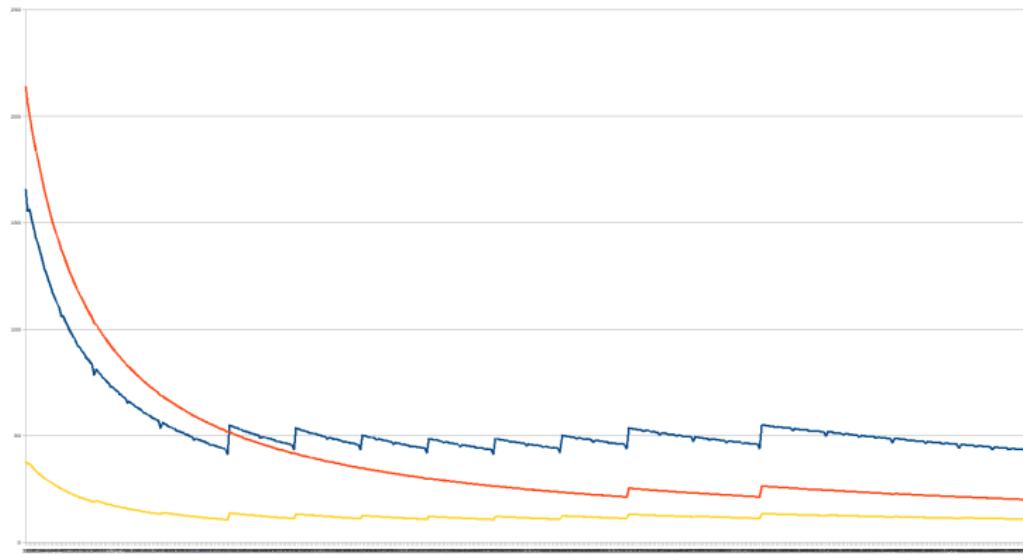
If you are interested, you can find quite a lot of documentation and tutorials in:

<http://developer.nvidia.com/cuda-gdb>

It has been deprecated, though, so it might be removed in the future.

Time measuring and speedups

Sometimes, since we know that by using profilers, we are going to get slower speeds, we just measure the efficiency of different grid sizes by looking at the time that it takes to run the kernels in production mode.



Time measuring and speedups

Speedups can be a bit tricky to calculate in CUDA, as we cannot always run the program in a single thread to get the amount of time needed to run the single threaded problem.

But the advantage is that we don't usually need to worry much about them, since we just try to get the most out of the cards that we are using anyhow.

Of course, speedups can and should be calculated when running on more than one GPU, but then we can use the standard methods.

The PCI Express Bus

PCI Express 3.0 can reach a sustained data rate of about 8 Gigatransfers per second (GT/s) or 985 MB/s per lane, but rarely manages to.

It would be an substantial advantage over PCI Express 2.0's effective 5 GT/s or 500 MB/s (per lane) if we managed to reach the peak speeds.

In a 16-lane bus, it would be 15.75 GB/s (or 128 GT/s) for PCI Express 3.0 and 31.51 GB/s (256 GT/s) for 4.0.

Time to look again at the bandwidths that we are being reported by the visual profiler!

The PCI Express Bus

PCI Express 4.0 doubled the data transfer rate again to 16GT/s or 1969 MB/s per lane - but this improvement was delayed: it was available from the end of 2017 (...it was supposed to be released in 2015!).

In a 16-lane bus, that would be almost 32 GB/s (or 256 GT/s).

Also, when you spec a machine, make sure that you check the PCI lanes on the motherboard - in some cases, the same lane is shared between two PCI Express slots, and you might get half the bandwidth for each.

The PCI Express Bus

PCI Express 5.0 doubles that again, to 64GB/s (or 512 GT/s) (on 16 lanes), and was finalised in 2019.

PCI Express 6.0 will double that, to 1024 GT/s and just about 128GB/s in x16 lanes, and the final release of the specification was in January 2022.

PCI Express 7.0, to 2048 GT/s and just about 256GB/s in x16 lanes, specs to be finalized in 2025.

DDR memory

DDR4's data transfer rates of 2133 megatransfers per second (MT/s) to 4266 MT/s compare to 800 MT/s to 2133 MT/s of DDR3.

DDR5 was released in 2020. It doubles the bandwidth of DDR4.

DDR6 should launch soon - again, it is supposed to almost? double the bandwidth of DDR5.

Why are these good news?

This should double the bandwidth between the RAM and the GPU - but the FSB of the CPU, which is in many cases the bottleneck of an application, increased only slightly.

Remember the Page-locked Write-Combining memory? We can mark parts of the memory so transfers between the RAM and the PCI Express bus are more efficient.

This means that GPGPU is going to remain strong for the next few years, but after that, it is hard to tell what is going to happen.

CUDA 5+

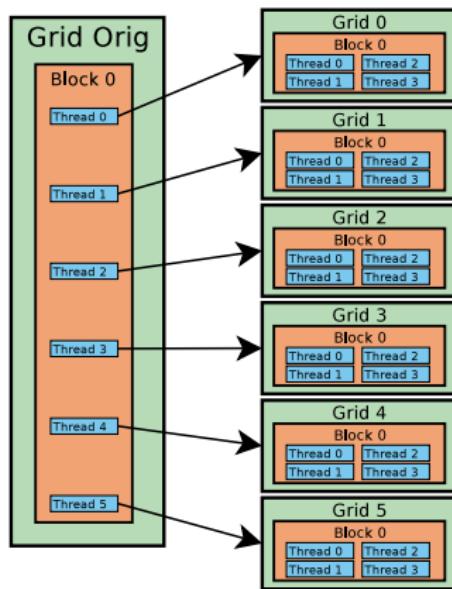
CUDA 12.4 is the latest stable version of the CUDA SDK. How much has the SDK evolved over the years?

From CUDA 5.+ , we got already a number of useful features that are worth mentioning:

- We now have the Nsight tools, to develop (with eclipse), debug and optimize (it might supersede Nvidia Compute Visual profiler eventually).
- RDMA for GPUDirect - Peer to peer memory transfers get much better - that means that we can use MPI_COPY to copy to another card in the cluster network.
- (In CC 3.5) Dynamic Parallelism - kernels will be able to spawn other kernels, which will improve grid resolution for some problems (FE, marching cubes, volume rendering...)
- (In CC 3.5) Better performance in hybrid applications (MPI + CUDA) due to the Hyper-Q technology

From CUDA 5 - Dynamic Parallelism

Out of these features, it is worth highlighting two - the first one is Dynamic Parallelism, which works like this:



From CUDA 5 - Dynamic Parallelism

Dynamic Parallelism removes one of the limitations that the original CUDA has, which is that a kernel can only be called from the host, and never from another kernel.

Quite obviously, this is going to make CUDA programs even more complex, due to the fact that it will be a lot harder to get the "global id" of threads, but the advantages are substancial - for example, in matrix operations, we can spawn a original grid with one thread per row, and then each one of them spawns as many threads as there are columns.

From CUDA 5 - RDMA for GPUDirect

RDMA for GPUDirect is also particularly potentially useful for us

The first advantage that this technology provides us with is that it makes us able to pin memory that can be accessed for read and write from any device in the PCI Express bus - for example, we could do real time image processing on the video feed obtained from a camera that is connected to a video capture card on the same computer.

Since the infiniband cards that are most commonly used in clusters also use that bus, this means that we can MPI_COPY (or other operations) to another node in such a way that the GPU in the other node will be able to get the updated data.

CUDA 6+

From CUDA 6.+ :

- Unified Memory (This is an improved version of mapped memory).
- CUDA on ARM processors (it was actually added in 5.5).
- cuBLAS can now use more than one GPU.
- Remote Development with NSight (not that useful for us, as we can ssh into machines)

CUDA 7+

From CUDA 7.+ :

- CUDA-GDB becomes deprecated.
- C++11 is enabled in CUDA (by passing the flag "`-std=c++11`" to nvcc).
- Thrust gets upgraded to version 1.8 (that can use Streams, amongst other improvements)
- The cuSOLVER library becomes available.
- The Runtime Compilation library (nvrtc) also becomes available.

CUDA 7.5+

From CUDA 7.5+ :

- 16-bit Floating Point (FP16) Data (half-precision, which will be supported natively by Pascal).
- Instruction-Level Profiling
- GPU Lambdas (still in experimental phase, though)

CUDA 8.0+

From CUDA 8.0+ :

- Support for Pascal cards (remember that there are very different types).
- Native FP16 and INT8 computation
- Improved Unified memory support
- Improved compiler performance and heterogeneous lambda support

CUDA 9.0+

From CUDA 9.0+ :

- Cooperative Groups - which is a new programming model so threads can communicate between themselves.
- A new API to program the Tensor Core matrix (half precision 4x4 matrix multiplication)
- Support for C++14 in CUDA device code

CUDA 10.0+

From CUDA 10.0+ :

- CUDA Graphs - this is actually a programming model, in which we can organize kernels in a graph, so they can be run as concurrently as possible, while respecting their dependencies.
- Vulkan and CUDA programming interoperability - so we can share global memory allocations with the Vulkan 3d graphics API.
- New nsight developer tools (Nsight Systems and Nsight Compute).
- On the A100, we can now set aside a portion of the 40-MB L2 cache to persisting data accesses to global memory. Persisting accesses have prioritized use of this portion of L2 cache, whereas normal or streaming accesses to global memory can only use this portion of L2 when unused by persisting accesses.
- Further improvements to previous features, such as CUDA Graphs, libraries for linear algebra, FFTs, and matrix multiplication, and the Nsight product family of tools.

CUDA 11.0+

From CUDA 11.0+ :

- Multi-Instance GPU (MIG) - so servers with GPUs on the cloud can share GPUs between different users.
- Ampere's Tensor Cores can now use mixed-precision matrix operations on different data types, including TF32 (which has 1 bit for the sign, 8 bits for the exponent but just 10 bits for the fraction) and Bfloat16 (1 bit sign, 8 bits exponent, 7 bits fraction) - just a reminder: FP32 is 1 bit sign, 8 bits exponent and 23 bits fraction.
- On the A100 compute card, we can aside a portion of the L2 cache to persisting data accesses to global memory, which means that we can assignate memory to a CUDA stream or a CUDA graph kernel node so it has reduced latency when being fetched.

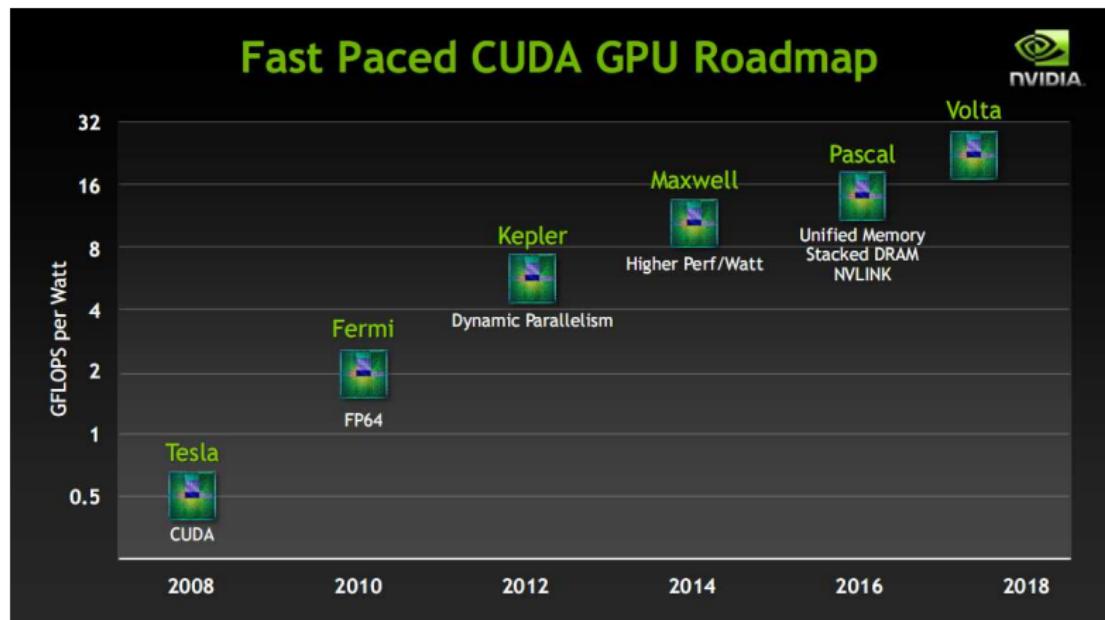
CUDA 12.0+

From CUDA 12.0+:

- Support for the Hopper architecture.
- Expansion of the cooperative groups to support the thread block clusters (didn't happen).
- Adding a feature that allows us to send from the shared memory of a block to the shared memory of another (probably within the same Streaming Multiprocessor?) (didn't happen).
- Improved dynamic parallelism
- Improved CUDA Graphs API
- Support for the GCC 12
- Support for C++20 (but still no modules on GPU code)

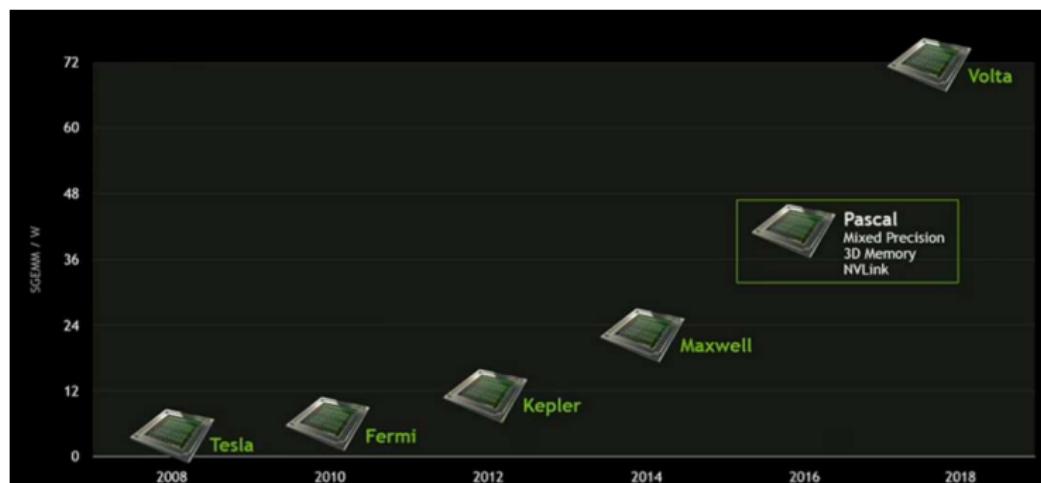
Beyond CUDA 6 - Architectures

In January of 2015, we got an update about the latest NVIDIA GPU roadmap:



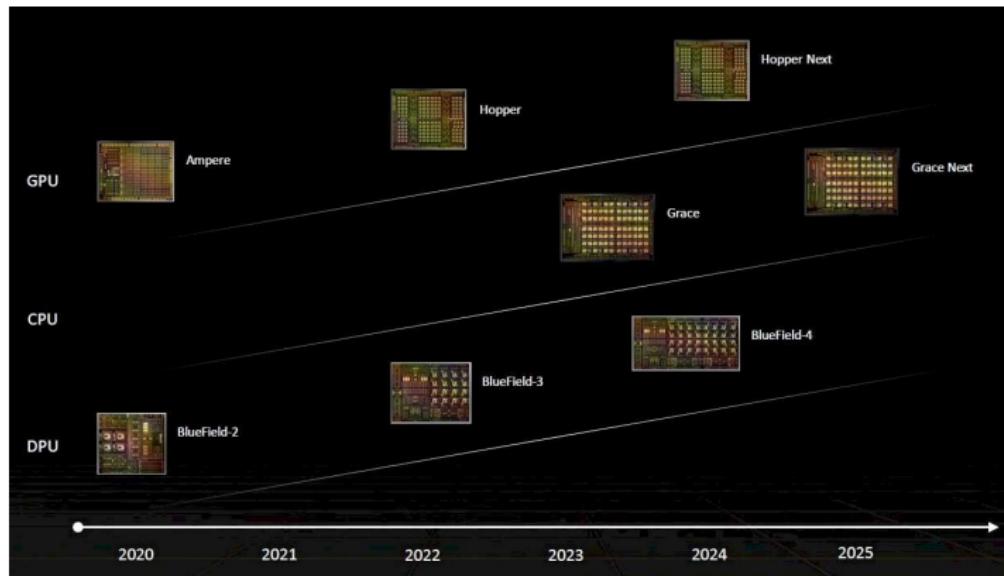
Beyond CUDA 6 - Architectures

And then, in March 2015, another one:



Beyond CUDA 6 - Architectures

The most recent one we can find still looks like this:



So the 3d stacked RAM took longer than expected, but it is finally here!

Beyond CUDA 5 - Maxwell

Also, we get improved multiprocessors in several ways- even when the number of CUDA cores per multiprocessor drops to 128, we have the same number of instruction issue slots per clock, and reduced arithmetic latencies.

Each multiprocessor still has 4 warp schedulers, but now the cuda cores are tied to a particular scheduler, which makes scheduling easier.

Each warp scheduler still has the flexibility to dual-issue (such as issuing a math operation to a CUDA Core in the same cycle as a memory operation to a load/store unit), but single-issue is now sufficient to fully utilize all CUDA Cores.

Beyond CUDA 5 - Maxwell

Also, the register file size and the maximum number of concurrent warps stay the same, (64k 32-bit registers and 64 warps), and so does the maximum number of registers per thread (at a value of 255).

However the maximum number of active thread blocks per multiprocessor has been doubled to 32, which should result in an automatic occupancy improvement for kernels that use small thread blocks of 64 or fewer threads (assuming the number of available registers and the amount of shared memory are not the occupancy limiter in a given kernel).

Beyond CUDA 5 - Maxwell

Another significant improvement in Maxwell is that it provides 64KB of dedicated shared memory per Streaming Multiprocessor.

The last two previous architectures, split the 64KB of memory between L1 cache and shared memory. Still, the per-thread-block limit remains 48KB on Maxwell, but the increase in total available shared memory can lead to occupancy improvements (in cases where we were forced to use smaller block sizes because otherwise we would be using up too much shared memory).

This was made possible in Maxwell by combining the functionality of the L1 and texture caches into a single unit.

Beyond CUDA 5 - Maxwell

Also, there is a change on how the atomic operations on the shared memory work.

The Fermi and Kepler architectures implemented shared memory atomics using a lock - update - unlock pattern that could be expensive in the presence of high contention for updates to particular locations in shared memory (as used to be the case of the atomicAdd).

However, Maxwell provides native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap (CAS), which can be used to implement other atomic functions.

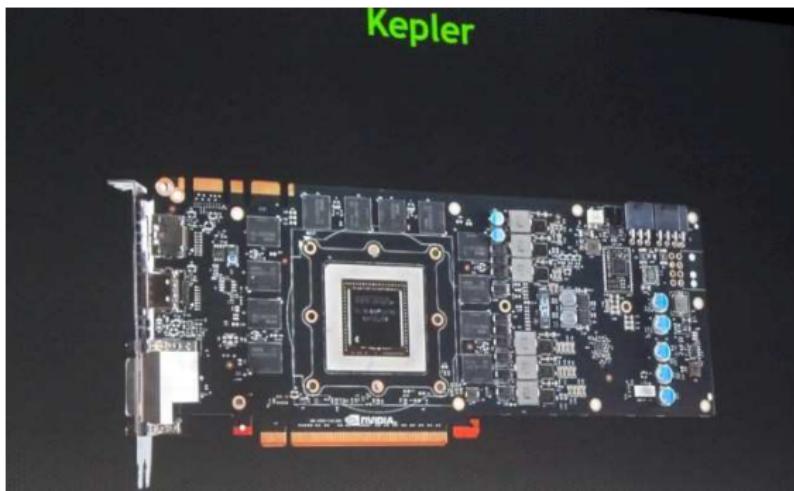
Beyond CUDA 6 - Pascal and Volta

So, in Pascal, we get the "Unified Virtual Memory" (that we were supposed to get in Maxwell already?), which means that the GPU will be able to access the host main memory, and the other way around.

We could kind of do something similar to this before, by using page-locked mapped memory, but that still had considerable problems (for starters, we do not know when the memory modifications have finished their propagations through the RAM), so we will find soon what this new technology is about.

Beyond CUDA 6 - Pascal and Volta

The big expected change was the 3D stacked DRAM in the Volta architecture (which was the next one after Pascal). The difference is that the GDDR4 is distributed around the GPU like this:



Beyond CUDA 6 - Pascal and Volta

Whereas the new architecture will move the memory closer to (in fact, on top of) the GPU, like this:

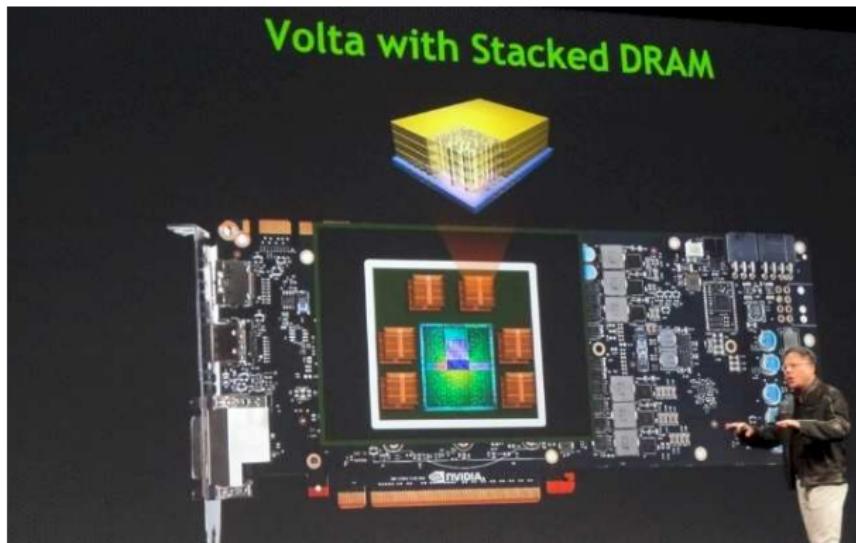


Image blatantly stolen from The Register

Beyond CUDA 6 - Pascal and Volta

By doing that, and having 6 stacked DRAM units plus a GPU in a single chip package, NVIDIA is expecting to have an aggregate bandwidth of 1TB/sec (and that B means bytes, not bits), so the performance of applications that use the global memory will be enhanced considerably.

But the 3D stacked memory is very expensive!

Beyond CUDA 6 - Turing

So the Turing architecture is, essentially, a cheaper version of Volta with GDDR6 instead of HBM2.

It's not just Geforces (including the Titan there) - there is a Tesla card (the T4), but it only has 2560 cuda cores, and it still uses GDDR6 memory.

Beyond CUDA 6 - Ampere

Ampere is a 7 nanometers architecture, and there are cards with HBM2 (like the A100) and GDDR6 (Like the RTX 3060 series).

There are three types of cards: The Geforce RTX 30X0s, the AX000 workstation cards (equivalent to the old Quadros) and the compute cards (so far the A40, and A100s with either 40GBs or 80GBs of HBM2 memory).

Beyond CUDA 6 - Hopper and Ada Lovelace

Hopper is a 4 nanometers architecture. The H100 cards have 80GBs of HBM3 (with a bandwidth of up to 3TB/s) and the H200 86GBs plus 480 GBs LPDDR5X in the CPU.

Higher TDP than the Amperes, 256 KB of combined shared memory and L1 data cache (which is 1.33x larger than A100).

Beyond CUDA 6 - Blackwell

Blackwell is still a 4 nanometers architecture. The B200 cards have 192GBs of HBM3e (with a bandwidth of up to 8TB/s) and the GB200 has 2x192GBs = 384GBs of HBM3e memory plus 480 GBs LPDDR5X (up to 512 GB/s) in the CPU.

Even higher TDP than the Hoppers, still waiting for the data on the SMs.

AFAIK there will be at least two chips, the G?B200 for datacenters (B100, GB200) and the G?B202 for gaming (Geforce RTX 50X0 series).

GDDR VS HBM memory

The cheaper cards will stick for now to GDDR memory - the latest generation of which is GDDR6X. GDDR6W is supposed to be a bit better.

GDDR6 is an improvement over GDDR5X, raising the per-pin bandwidth to 14 Gbps from 10 Gbps.

As an example, a RTX 4090 has a memory bandwidth of 1008 Tb/s with 24GBs of GDDR6X memory.

On the other hand....

Out of the 10 top 500 HPC machines in the world (on the list released in November of 2023), nine have GPUs, and six of them Nvidia ones, of which two are fairly old Volta.

In the previous list (June 2023), numbers were similar - seven with GPUs, five with Nvidia cards.

This means that GPGPU still seems to be pretty stable in HPC, so it can be hard to tell what is going to happen over the next few years - even more so for Nvidia powered clusters.

On the other hand....

And what happens if we look at the Green500 list?

In the latest list (November 2023), all of the machines in the top 10 do use GPUs. In the previous one (June 2022), all of the machines in the top did, as well.

So they are ahead by quite a bit over almost any other available technology at the moment (with only the Japanese architectures being a challenge). Worth keeping in mind when we suffer an energy crisis!

On the other hand....

But what about the Graph 500?

In the current list of the BFS test (November 2023), there are two GPU machines, and they might be running the code with CPU only. The SSSP has one.

Part of the problem might be that the benchmark that they use to create the list hasn't been ported to cuda properly, but as time passes, this seems extremely unlikely.

Or simply that the architecture isn't well suited to the problem.

On the other hand....

And the HPCG (High Performance Conjugate Gradient)?

In the current list (November 2023), there are 8 gpgpu machines in the top 10, and surprisingly enough, two of them are still running V100s. The top spot is not a GPU machine, though!

So Nvidia cards seem to be doing ok here, at the moment.

Future of CUDA in HPC

As mentioned before, the current HPC market for machines with accelerators is dominated right now by Nvidia and AMD cards, as Xeon Phis are no longer viable competition, and AMD seems to be pushing for double precision performance (all the current and planned double precision exascale machines at the moment seem to have AMD cards on them).

Another threat that was considered to be quite considerable at the times was the Japanese computer's technology, but that machine was very expensive to research and build (total investment of 112 billion yen (\$1.42 billion)!!)- although it was competitive back in the day with hybrid machines even on a per-Watt basis.

Future of CUDA in HPC

Intel has been threatening CUDA's GPGPU dominance since two years before launching Larrabee, and again with the Intel Xeon Phi, but the last generations of Intel Xeon Phis have failed to pose serious competition to other hybrid systems, and the last version (which was not a new CPU but a co-processor card that goes in a PCI-Express slot) has been discontinued. Then they released their GPUs, but they so far they seem to be aimed for low and middle range gaming, and only recently they have been considered for HPC.

AMD are finally making lots of progress in the compute sector with their new accelerators, even when their library isn't as popular (OpenCL and ROCm are both valid alternatives) and their machines in the upper parts of the list finally prove their progress.

Future of CUDA in GPGPU

Regarding libraries, OpenCL has to offer support for both NVidia and AMD/ATI chipsets (and most likely Intel ones as well), so it is always going to be steps behind in performance, at least on CUDA cards.

On the other hand, converting your code to CUDA is difficult, time-consuming, and does not always result in an improvement - one of the reasons why the Nvidia Chinese machines were not that successful (there were some reports claiming that they only had about a 10% GPU usage!!), which prompted them to try different architectures (first Xeon Phis and then their own).

But again, the IBM BlueGenes and the Xeon Phis computer could run MPI code straight away, and they weren't that successful either.

Yet again.....

Is there any other way to get 73.07 teraflops (single precision) in a normal desktop as of April 2024 for around 2,500 euros (...suggested price at launch was 1599 USD, by the way!), other than with a machine with a 4090?

CUDA still delivers an very sizable amount of single precision compute!

As previous attempts to make normal users buy additional FPGAs (remember, for example PhysX's Ageia - and its PPUs... that -ahem!- ended up being bought by Nvidia!), CUDA so far has been the most successful attempt at bringing HPC to the average desktop.

..and there is the mobile market as well

In January 2013, the Tegra 4 ("Wayne") system on a chip (SoC) was announced - it was the first Tegra chip which supported proper CUDA.

The Tegra 4 had 72 (48+24, actually) GPU cores, and they were supposed to be programmable and CUDA-compatible... but they ended up not being, somehow.

In fact, they were supposed to be based on the Kepler technology. Tegra 4 itself was built as a TSMC 28nm HPL process (Which means that it was low-powered instead of max-performance'd).

..and there is the mobile market as well

However, in March 2013, it was announced that the Tegra 5 ("Logan", later renamed to "Tegra K1") system on a chip (SoC), that supports CUDA.

The Tegra K1 is, essentially, an ARM processor with an attached Kepler streaming multiprocessor, so it had 192 cuda cores(the number of CUDA cores in one Kepler streaming multiprocessor), and has a TDP of 5W.

If we look at the Tegra roadmap, we can get an idea about where this technology is going.

SoCs (Systems on a Chip)

The Tegra X2 uses the Pascal architecture on the GPU side which means that it is 16 nm, and has 256 cuda cores, with a TDP of 7.5-15W.

Xavier uses the Volta architecture on the GPU side which means that it is 12 nm and has 512 cuda cores, with a TDP of 20-30W.

Orin uses the Ampere architecture on the GPU side which means that it is 7 nm and has 2048 cuda cores, with a TDP of 5?-45W.

Atlan was meant to have an Grace CPU and a Hopper GPU on it, but it got cancelled and we are getting Thor instead, with a Blackwell GPU (to be confirmed!) and to be released in 2025.

Jetson Nano

The Jetson Nano is a very small and cheap SoC, with a Maxwell Streaming Multiprocessor - so it has 128 cuda cores.

The CPU is a 1.43GHz quad-core ARM A57 processor, with 4GB of 64-bit LPDDR4.

It is equivalent-ish to a Raspberry Pi, and has a TDP of 5-10W.

No wireless installed by default on it, though!

Fine, so what about Grace Hopper?

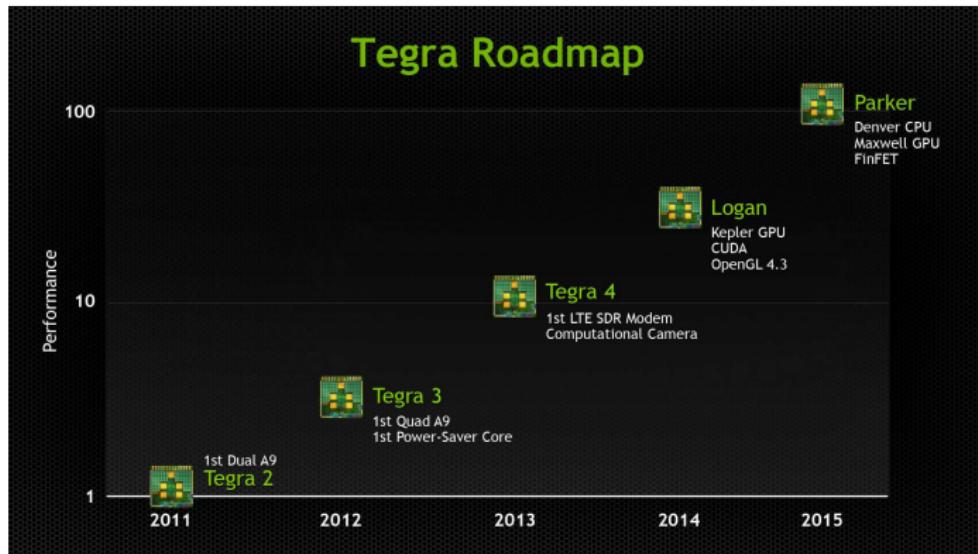
The state of the art, it's an ARM CPU and Hopper GPU in the same package.

The CPU is an ARM processor (Grace), with LPDDR5X memory (240GBs with 500GB\`s bandwidth).

The GPU is a Hopper with HBM3 memory (96GBs with 4000GB\`s bandwidth).

They link through NVLink CSC (900GB\`s bandwidth), and both can access the other one's memory.

NVIDIA Tegra roadmap



So we could expect a x10 growth on compute power in the next 2-3 years - and most of that increase is going to be in the GPU. But is this really happening, anymore?

Advanced CUDA programming, revisited

Fifth chapter: Advanced CUDA programming, revisited.

We now know how to write, run, debug and profile programs in the GPU, let's try to go a bit further and look at how at techniques to improve cuda performance and code quality.

Also, we will see features that have become available in the latest hardware generations.

Dynamic allocation of shared memory

We have seen before how to use the shared memory, but we have skipped one possibility - how to allocate the shared memory instead of in a fixed way.

We can do this by adding an int value that marks the number of bytes that we want to reserve when we call the kernel:

```
sharedMemoryKernel<<<dimGrid , dimBlock , n * sizeof (int)>>>(device_array , size );
```

So that is what the third position in the execution configuration syntax is for.

Dynamic allocation of shared memory

And then, in the kernel, we declare the shared memory array as an unsized extern array:

```
extern __shared__ int sharedMemoryThread[];  
//As opposed to __shared__ float sharedMemoryThread[BLOCK_SIZE];  
sharedMemoryThread[threadIdx.x] = threadIdx.x;
```

Dynamic allocation of shared memory

If we need to use more than one array of dynamically allocated shared memory, we can just use a larger chunk and manage it with pointers to different positions, this way:

```
extern __shared__ int sharedMemoryThread[];  
  
int *myIntegerPointer = sharedMemoryThread;  
float *myFloatPointer = &sharedMemoryThread[numberOfIntegers];
```

And then just call the kernel like this:

```
sharedMemoryKernel<<<dimGrid, dimBlock,  
    (numberOfIntegers*sizeof(int)+numberOfFloats*sizeof(float))>>>(device_array, size);
```

Dynamic global memory - reserved heap memory

If we look back to the part where we saw how to use printf from the device, we saw that there was a limit on how much output we could use.

This is because that output is dumped into a buffer in the global memory, that after the execution (and without us having to control it) is copied back to the host and then output to the console.

The reason for this is that there is a given amount of global memory that we can use for this purpose - if we ran out of it, it will behave in a circular way.

Dynamic global memory - reserved heap memory

But there is another way of using this memory in devices of CC2.X+ - we can use a heap of global memory that we can allocate dynamically.

First of all, both the printf buffer and the amount of heap memory can be found and stated from the host (once we are in a kernel, it is too late to change it!) by using the `cudaDeviceGetLimit` and `cudaDeviceSetLimit` functions.

Dynamic global memory - reserved heap memory management

Once inside the kernel we can allocate, free and manage the heap memory by using the same functions that we use in the host:

Dynamic global memory

```
void* malloc(size_t size);
void free(void* ptr);
void* memcpy(void* dest, const void* src, size_t size);
void* memset(void* ptr, int value, size_t size);
```

Dynamic global memory - per thread allocation

Now, there are two ways to do this - we can get each thread to allocate its own array.

This is really handy and gives us lots of flexibility for our cuda algorithms, but we must be careful, and we have so many threads at our disposal that it is easy to run out of memory.

And a memory leak in this circumstance could be catastrophic - so make sure to get the frees right!

Dynamic global memory - per block allocation

There is another option, which is to exploit the fact that different threads in the same block can reach the memory that the others allocated.

We can share the addresses by using the shared memory - but careful with this, as it makes our code difficult to read.

Also, a number of `__syncthreads` calls will be necessary to make sure that the memory has been allocated (and initialized) before using it.

Still, we have to take into account that the global memory is quite slow and thus we will spend lots of time waiting for memory I/O and sync.

Dynamic global memory - persistent allocation

And we have yet another option, which is to store the addresses (don't forget that we will have one per block!) to the memory in the global memory and keep it between kernel launches.

It might be a good idea to move the pointer back to the registers or shared memory when using it, to reduce overheads.

No need to mention that memory leaks or unfreed memory can be even more dangerous than in the previous cases!

Dynamic global memory - cudaMemGetInfo

It might be a good idea as well to use the cudaMemGetInfo function - which gives us information about the amount of memory being used at the moment in the chosen card.

```
size_t freeGlobalMemory, totalGlobalMemory;  
cudaMemGetInfo (&freeGlobalMemory,&totalGlobalMemory);  
  
printf ("freeGlobalMemory= %lu bytes\t",freeGlobalMemory);  
printf ("totalGlobalMemory= %lu bytes\n",totalGlobalMemory);
```

Coalesced memory

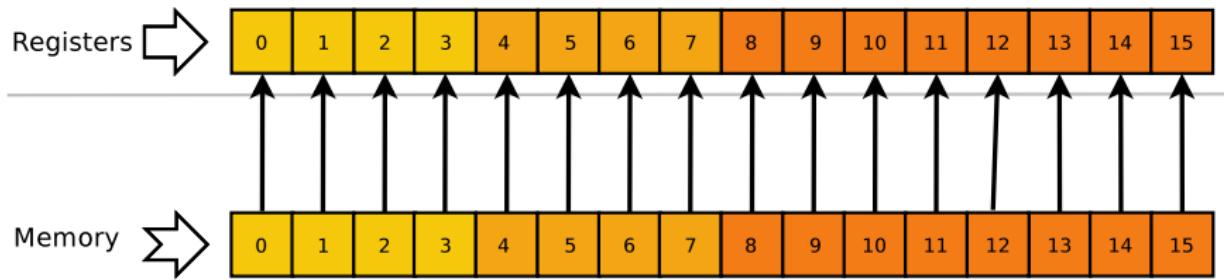
How many memory transactions are necessary and how much throughput is ultimately achieved depends on the architecture of the device.

For optimal performance, we need the memory to be well aligned with the threads that are using it - that is what we call coalesced memory.

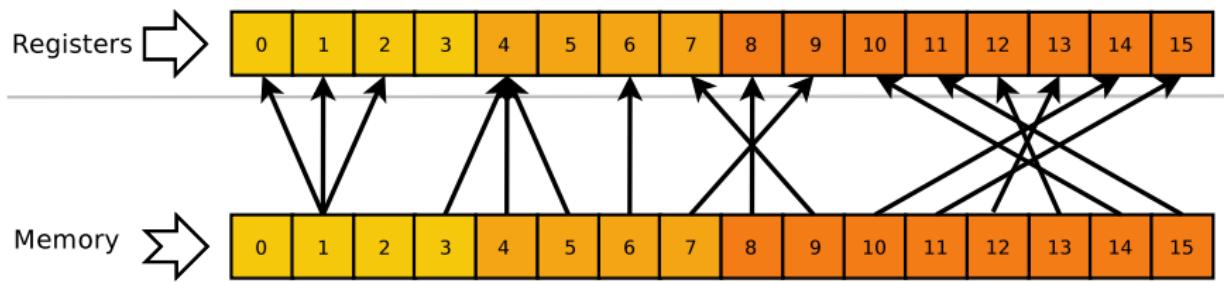
In old devices, the requirements on the distribution of the memory across the threads to get any coalescing at all were very strict.

They are much more relaxed for devices of higher compute capabilities. For devices of compute capability 2.x and higher, the memory transactions are cached, so data locality is exploited to reduce impact on throughput.

Coalesced memory



Coalesced memory



Shared memory and L1 cache configuration, 1 of 4

Let us look again at the amount of shared memory available in the old GeForce 9800 GTX+:

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce 9800 GTX+  
GPU Shared memory available per block in bytes: 16384 (16KBs)  
GPU Maximum number of threads per block: 512
```

And in the GTX 680:

```
////////////////////////////////////////////////////////////////////////  
GPU model name: GeForce GTX 680  
GPU Shared memory available per block in bytes: 49152 (48KBs)  
GPU Maximum number of threads per block: 1024
```

In fact, on devices of CC 2.x and 3.x, we have 48KBs of shared memory available, but we can reconfigure this to favor the L1 cache where the registers are stored, if we need to.

Shared memory and L1 cache configuration, 2 of 4

In devices of cc 2.x, there are two settings: 48KB shared memory / 16KB L1 cache, or 16KB shared memory / 48KB L1 cache.

The default setting is 48KB allocated to the shared memory and 16KB to the L1 cache; but this can be configured during runtime API from the host for all kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`.

These accept one of three options:

`cudaFuncCacheNone`,
`cudaFuncCacheShared`, and
`cudaFuncCacheL1`.

Shared memory and L1 cache configuration, 3 of 4

The driver will set up the shared memory and the L1 caches to match the specified preference, unless a kernel requires more shared memory per thread block than it is available in the requested configuration.

In devices of compute capability 3.x, we also have a third balance setting that allows us to specify a 32KB shared memory / 32KB L1 cache configuration, which can be requested using the option `cudaFuncCachePreferEqual`.

Shared memory and L1 cache configuration, 4 of 4

Usage of the two function calls, however, is a bit different - with `cudaDeviceSetCacheConfig` we only need to pass the mode as an argument, as it affects all the kernels:

```
////////////////////////////////////////////////////////////////////////  
// request 32KB shared memory / 32KB L1 cache configuration for all kernels  
// take note that cudaFuncCachePreferEqual should only work on cc 3.x  
cudaDeviceSetCacheConfig(cudaFuncCachePreferEqual);
```

But when we use `cudaFuncSetCacheConfig()`, we need to pass a pointer to the kernel that we want to configure the shared memory for, this way:

```
////////////////////////////////////////////////////////////////////////  
// configure scanThreadInformationGPU in particular to request more shared memory ()  
cudaFuncSetCacheConfig(&scanThreadInformationGPU, cudaFuncCachePreferShared);
```

Asynchronous Concurrent Execution - Streams

We have mentioned before that we can run several kernels in a single GPU at a given time, and that, in fact, we are actively encouraged to do so.

In fact, we can do more than that - kernel launches, memory copies and memory set function calls can all be executed concurrently.

Also, it is important to understand the difference between parallelism and concurrency - concurrency allows us to increase the load on the graphics card, than might (or might not) benefit from the parallelism on it.

Asynchronous Concurrent Execution - Streams

The very first thing that we have to do is to create cuda Streams. Here's an example about how to do it:

```
// Declare and allocate two streams
cudaStream_t stream[2];
for (i = 0; i < 2; ++i) {
    cudaStreamCreate(&stream[i]);
}
```

That also means that we have to destroy them at the end:

```
// Declare and allocate two streams
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Asynchronous Concurrent Execution - Streams

Here is how to set up asynchronous memory transfers:

```
// Copy data from host memory to device memory
cudaMemcpyAsync(array0_GPU, array0_CPU, sizeof(int)*N, cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(array1_GPU, array1_CPU, sizeof(int)*N, cudaMemcpyHostToDevice, stream[1]);
// Copy data from device memory to host memory
cudaMemcpyAsync(array0_CPU, array0_GPU, sizeof(int)*N, cudaMemcpyDeviceToHost, stream[0]);
cudaMemcpyAsync(array1_CPU, array1_GPU, sizeof(int)*N, cudaMemcpyDeviceToHost, stream[1]);
```

So business as usual.

Asynchronous Concurrent Execution - Streams

Calling the kernels is pretty straightforward too:

```
myKernel_0 <<<dimGrid ,dimBlock ,0 ,stream[0]>>>(array0_GPU , N);  
myKernel_1 <<<dimGrid ,dimBlock ,0 ,stream[1]>>>(array1_GPU , N);
```

Remember that the "0" in the third position states the amount of dynamically allocated shared memory that we are requesting for each kernel.

Running the program in the Nvidia NSight Compute shows if we are running the kernels concurrently.

Asynchronous Concurrent Execution - Streams

We have several synchronization options available too:

`cudaDeviceSynchronize` makes the execution wait until all the streams are finished with their issued instructions.

`cudaStreamSynchronize` makes one stream (passed as an argument) wait until all its issued instructions are finished.

`cudaStreamWaitEvent` takes one stream and one event (passed as arguments) and makes all the instructions added to that stream delay their execution until the event is completed, i.e. `cudaEventRecord` is called for that Event and stream.

`cudaStreamQuery` will return `cudaSuccess` if all operations in the stream passed as an argument have completed, or `cudaErrorNotReady` otherwise.

Asynchronous Concurrent Execution - Streams

We can also insert callbacks to the streams - those are custom functions that will be executed once all the threads of the specified stream have reached the point where the callback is inserted.

```
cudaStreamAddCallback(stream[0], MyCustomCallback,  
                      (void *)("Stream 0: scanThreadInformationGPU has finished\n"), 0);
```

The custom function has to be defined like this:

```
void CUDART_CB MyCustomCallback(cudaStream_t stream, cudaError_t status, void *data) {  
    printf("Inside callback: %s\n", (char *)data);  
}
```

So in this case, when Stream 0 reaches the point where we call `cudaStreamAddCallback`, we will get the following output:

```
Inside callback: Stream 0: cudaMemcpyAsync – cudaMemcpyHostToDevice has finished
```

Asynchronous Concurrent Execution - Streams

A word of warning: you need to be very careful about how do you organize the streams when you use loops. This works:

```
for (i = 0; i < 4; ++i) {
    // stream[i] copies array[i] to device
    cudaMemcpyAsync (arrayGPU[i], array[i], sizeof(float)*N,
                    cudaMemcpyHostToDevice, stream[i]);
}
for (i = 0; i < 4; ++i) {
    // stream[i] computes array[i]
    add_arrays_exp_gpu <<<dimGrid, dimBlock, 0, stream[i]>>> (arrayGPU[i],
                                                                arrayOutGPU[i], N);
}
```

But this doesn't - it gets serialized instead:

```
for (i = 0; i < 4; ++i) {
    // stream[i] copies array[i] to device
    cudaMemcpyAsync (arrayGPU[i], array[i], sizeof(float)*N,
                    cudaMemcpyHostToDevice, stream[i]);
    // stream[i] computes array[i]
    add_arrays_exp_gpu <<<dimGrid, dimBlock, 0, stream[i]>>> (arrayGPU[i],
                                                                arrayOutGPU[i], N);
}
```

Asynchronous Concurrent Execution - Streams

You can see the difference when you use Nvidia's Nsight Systems:



You can find a code example on how and how not to do this in `cudaStreamsHowNotTo`.

Dynamic Parallelism

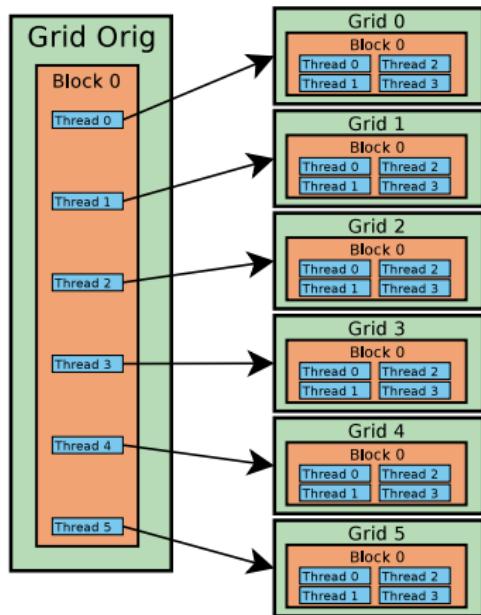
Dynamic Parallelism is an useful technique that allows us to remove the restriction of not being able to launch kernels from the device - as long as it is only from kernels (we cannot spawn a new grid from a device function).

Essentially, it means that we can spawn new grids from anywhere in our device code.

However, it is going to increase the complexity of our code quite significantly.

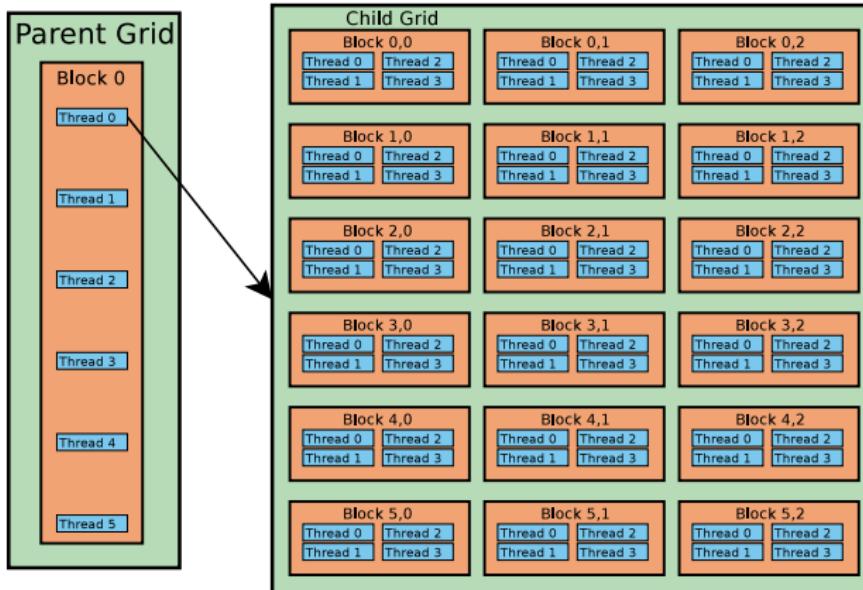
Dynamic Parallelism

This means that we can get each thread of a grid to spawn a new grid like this:



Dynamic Parallelism

Or we can just get even a single thread to spawn the new grid:



Dynamic Parallelism

The first thing that we need to consider is the way in which the grids synchronize - essentially, a parent grid is not considered complete until all the children grids that it has spawned have completed themselves.

However, we will be able to use Streams and Events created on the device to be able to manage dependencies between the threads - but only within a block.

Also, there is the subject of cuda object visibility - what a parent thread can see of the memory allocated by the children threads is about the same as what the host can see in the device (with the exception of not needing `cudaMemcpy`).

Dynamic Parallelism - Compilation

First of all, we will need to change two things in the Makefile for dynamic parallelism to work:

```
NVCCFLAGS += -arch=sm_35 --relocatable-device-code true
```

-arch=sm_35 (or later) makes perfect sense, as we know that dynamic parallelism only works from cc 3.5+, but the -relocatable-device-code true is less familiar to us.

In the latest versions of cuda, we must use that flag for compiling, but it doesn't seem to be necessary for linking.

Dynamic Parallelism - Memory

Regarding the memory, parent and child grids will share the same global and constant memory storage, but their respective register and shared memories will be different.

The texture memory can be used, but we have to be very careful with the surface as it might not be coherent.

The good news is that we can check pointers to see whether a particular address belongs to the global memory or not by using the intrinsic `__isGlobal()` function.

Dynamic Parallelism - Memory

Also, registers and local memory in a parent thread are not visible outside that particular thread, so we cannot pass a pointer to a memory address in them as an argument when launching a child kernel - this certainly a problem when we need to pass matrices or vectors as arguments.

Also, we can use dynamically allocated shared memory in the children in the usual way (by passing the right argument in the kernel call and then using extern when initializing the shared memory).

Dynamic Parallelism - Synchronization

One of the biggest differences when we start to invoke child kernels is that all calls are asynchronous - so execution does not stop until the child kernel is finished, as it (usually) happens when we execute them from the host.

We used to be able to force synchronization with the child kernel by calling `cudaDeviceSynchronize()`, but using this function in kernels or device functions got deprecated from SDK 11.6 and then removed in 12.

One option would be to run `__syncthreads()`, but so far the recommendation when trying to access global memory that the child kernel is writing to is to start another kernel after the parent one has finished.

Dynamic Parallelism - Environment Configuration

The child always inherits the same environment configuration that the parent has.

That means that the L1 / shared memory configuration, and stack size in the child kernels will be the same that we have specified for the parent.

Also, the device management is still restricted to the same device in which we are running.

Multiple device systems

So let us now consider the case of single machines that have more than one GPU available to it. How can we exploit this to our advantage?

What we can not do with those systems is to execute kernels that run on both devices at the same time - in fact, when we declare Streams, they are linked to a given device and will cause errors if we try to execute kernels on them in any device other than the one that they were originally set up.

But what we can do is to share and copy memory between them (in CC 2.X+!).

Multiple device systems - Portable memory

The first way to do that is one that we have mentioned before - by using the portable memory.

Portable memory is used in a very similar way to mapped memory, with the main difference that when we call `cudaHostAlloc`, we pass the `cudaHostAllocPortable` flag instead of the `cudaHostAllocMapped` that we used before.

By doing that, and then getting the pointer to the GPU memory by using, `cudaHostGetDevicePointer`, we can access that memory from all of the devices and the host as well.

Multiple device systems - Peer to peer access

But what if we just to be able to access the global memory of a device from another?

We can also use peer to peer memory access to the memory of device 0 (for example), that we can enable by setting another device (say 1) and then calling:

```
cudaSetDevice(1);  
cudaDeviceEnablePeerAccess(0, 0);
```

Don't forget that you will have to set device 1 before calling cudaDeviceEnablePeerAccess!

Multiple device systems - Peer to peer copy

Another option is to simply copy global memory from one device to another.

We can use any of the function `cudaMemcpyPeer`, `cudaMemcpyPeerAsync`, `cudaMemcpy3DPeer`, or `cudaMemcpy3DPeerAsync` to do that - as an example:

```
cudaMemcpyPeer(pointer1, 1, pointer0, 0, size);
```

Would copy `size` bytes from `pointer0` in device 0 to `pointer1` in device 1.

Time functions

Also, it is worth mentioning that there are two time functions that we can use in device code, and that return the number of clock cycles that have passed:

```
clock_t clock();  
long long int clock64();
```

We can use these functions to measure accurately the performance of different parts of our CUDA code - quite useful when the CUDA events are not able to do it.

Time functions

Also, take into account that `clock64` gives you a number of clock cycles. To turn them into seconds, you will need the clock rate of the card, that you can get this way (from the CPU):

```
int clocks_per_ms;  
cudaDeviceGetAttribute(&clocks_per_ms, cudaDevAttrClockRate, devicId);
```

It is recommended to use `cudaDeviceGetAttribute` instead of `cudaGetDeviceProperties`, as it takes less time to execute.

Warp vote, shuffle and match functions

We have seen before a number Warp vote functions, that we can use to improve parallel execution.

```
int __all_sync(unsigned mask, int predicate);
int __any_sync(unsigned mask, int predicate);
unsigned __ballot_sync(unsigned mask, int predicate);
unsigned __activemask();
```

Those functions use predicates (conditional expressions, or variables that are checked as if they were boolean) to share information between threads of a warp - but now we are going to see as well some shuffle functions, that we can use to share data between the threads of a warp.

Warp vote, shuffle and match functions

This is the first time that we see `ballot_sync` - which returns an unsigned int with information about the status of a given predicate for each thread of the warp in a per bit basis.

Also, `__activemask` returns a binary mask with all the threads in the warp that are active with the current executing thread.

Warp vote, shuffle and match functions

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

The unsigned mask we usually have to define ourselves, and serves as a binary mask that tell which threads in the warp should execute the order - for example, defining it as 0xffffffff (an integer with all 1s in binary representation) means all the threads. We can also call `__activemask`, which give us a mask with all the threads in a given warp that are active in the current active warp.

T, in this case, be an int, unsigned int, long, unsigned long, long long, unsigned long long, float or double. With the `cuda_fp16.h` header included, T can also be `__half` or `__half2`.

Warp vote, shuffle and match functions

`__shfl_sync (unsigned mask , T var , int srcLane , int width=warpSize)` fetches the variable `var` from the lane `srcLane` in the same warp.

`__shfl_up_sync (unsigned mask , T var , unsigned int delta , int width=warpSize)` fetches the variable `var` from the lane **minus** `delta` (careful, "up" here means lower id!) in the same warp.

`__shfl_down_sync (unsigned mask , T var , unsigned int delta , int width=warpSize)` fetches the variable `var` from the lane **plus** `delta` (careful, "down" here means higher id!) in the same warp.

Warp vote, shuffle and match functions

The match functions look like this:

```
unsigned int __match_any_sync(unsigned mask, T value);  
unsigned int __match_all_sync(unsigned mask, T value, int *pred);
```

Where T, for the match case, can be an int, unsigned int, long, unsigned long, long long, unsigned long long, float or double.

The match functions return binary masks that we can use with other warp functions - `match_any` returns the mask of threads that have same value of `value`, and `match_all` returns mask if all threads in mask have the same value for `value`; otherwise 0 is returned. Predicate `pred` is set to true if all threads in mask have the same value of `value`; otherwise the predicate is set to false.

You'll need to use the flag "`-arch=sm_70`" to get the match functions to compile, though!

Warp vote, shuffle and match functions

Finally, we have a function that can provide warp-wide barrier synchronization:

```
void __syncwarp(unsigned mask=FULL_MASK);
```

We can still setup the mask and sync only some of the threads in the warp, if that is what we need.

Warp vote, shuffle and match functions

Now, the first novelty in warp-level programming is that we have a new term - the lanes.

A lane is a thread id respective to the warp - which means that we have yet another thread id to worry about.

delta, in the up and down functions, represents how many lanes up or down we want to get the data from.

value, would be the per-thread variable that we want to share - and this (the value from the calling thread) is the value that is returned when either sourceLane or the thread lane plus or minus the delta value are illegal.

Warp vote, shuffle and match functions

The binary masks were introduced due to lots of problems with different threads in warps not being at the same execution point in the code when calling the warp functions.

The previous warp level functions didn't require them, but they have been made deprecated now, so we are going to ignore them.

Warp vote, shuffle and match functions

The new functions also introduce some potential problems - for example, if an active thread calls a warp level function passing a mask that has its own bit marked as zero, the result is undetermined.

Also, newer architectures (Volta and later) handle threads in a warp in different ways (independent thread scheduling and such), so, if you experience erratic behaviour, you might want to try compiling with "`-arch=compute_60 -code=sm_70`".

Unified Memory

Unified memory is another option that allows us to allocate memory on both the host and the gpu (or even gpus, if there is more than one).

This has a number of advantages, amongst them, that it can simplify the code quite a bit; and disadvantages, such as us losing control over how and when the data is transferred over the PCI-Express bus.

In SoC systems, this option is a considerable advantage, as the memory space is shared between the CPU and the GPU, there is no transfer to worry about.

Unified Memory

It requires a 64 bit system and a GPU with CC 3.0 or more - but now that Fermi and 32-bit systems have been deprecated by NVIDIA's drivers, that means pretty much any modern system.

GPUs with CC 6.0 (Pascal) or more have additional features, such as on-demand page migration and GPU memory oversubscription (which allows us to allocate more memory than we have in the GPU, transferring from the system memory when requested).

Also, take into account that the most advanced features might be Linux only.

Unified Memory

In fact, from CUDA 8.0 (latest is 10.1), and on CC 6.0 or more (so, Pascal onwards), it is activated by default by the "malloc" or "new" calls from the system (so you can use the same pointer from either the CPU or GPU).

In machines that do not support this, we can still explicitly use the unified memory in two ways. One is to call the following function when allocating:

```
cudaError_t cudaMallocManaged(void **pointer,  
                             size_t size,  
                             unsigned int flags=0);
```

The pointer that `cudaMallocManaged` provides is a pointer to the global memory of the (currently set) GPU, but that is valid when used from the CPU and any other GPUs in the system.



Unified Memory

Another possibility is to define the memory as a global variable, by the use of the `__device__ __managed__` keywords:

```
__device__ __managed__ float unifiedMemory[100];  
  
__global__ void myAddValueKernel (float value) {  
    int idx = blockIdx.x*blockDim.x+threadIdx.x;  
    unifiedMemory[idx] += value;  
}
```

Which is certainly simpler, but gives us less flexibility compared to dynamically allocating the memory.

Unified Memory

Either way, we will need to sync the memory after we call a kernel before we can safely access it from the CPU. There are two ways to do this too - one is to call `cudaDeviceSynchronize()` before we request it:

```
myAddValueKernel<<<dimGrid , dimBlock>>>(value );  
// Try to comment this out and see if still works, I dare you!  
cudaDeviceSynchronize();
```

The other option is to set the environment variable `CUDA_LAUNCH_BLOCKING` to 1 - this way, after each kernel call, the memory gets sync'ed; but this is mostly for debugging, and shouldn't be used in production code:

```
export "CUDA_LAUNCH_BLOCKING=1"
```

Unified Memory

There are a few differences on how the Unified memory is managed, compared to the usual mapped memory. For starters, on compute capability lower than 6.0, we cannot have another GPU or the CPU access unified memory while a kernel is running.

On devices with CC 6.0+, we can (there's a concurrentManagedAccess flag that we can check), but we still need to take care of provide proper synchronization, which can be, obviously, an issue.

Unified Memory

There are other options to sync the memory, other than just calling `cudaDeviceSynchronize()` before we access it from the CPU.

The main resource that we have is the `cudaStreamAttachMemAsync` function, that we can use to attach allocated unified memory to particular cuda streams - which also means that that part of the memory won't be available in other streams or other GPUs, though.

Unified Memory

```
cudaError_t cudaStreamAttachMemAsync( cudaStream_t stream ,  
                                     void *pointer ,  
                                     size_t length=0,  
                                     unsigned int flags=0);
```

Where stream is the stream we want to associate the memory with, pointer the pointer to the memory and length the amount that we want to assign (so far, only 0 -for the whole chunk- can be used, but this should change in the future).

MPI and cuda

We have seen before how to add cuda code to MPI code and how to choose different cards and even how to run various kernels at the same time, but - what if we want to do all of that at the same time?

The key thing to remember is that in MPI each MPI process runs a copy of the host code (not unlike each thread runs a copy of the device code), so we will have to handle the inter MPI process communication by using MPI functions.

MPI and cuda

It also means that (if the system allows it) we have the advantage of being able to spawn more MPI threads than CPU cores we have in the machine; and use them to call the cuda kernels and let those spare threads wait for the result from the cards, exploiting that way all the available hardware (both CPUs and GPUs), and then combine everything at the end with MPI.

Or, if we are in a multi-device system, we can run blocking kernels on both devices at the same time.

MPI and cuda

Either way, it is worth mentioning that since, more often than not, we have more memory available in the host than in the device, in some cases we might want to split the problem in pieces and solve it one at a time in GPUs.

Quite obviously, this is going to increase the complexity of our code even more - since we are adding an extra layer or memory multiplicity.

Also, to run the compute sanitizer, it is recommended to do it like this (so each MPI process runs compute-sanitizer by itself):

```
mpirun -np 5 /usr/local/cuda-12.2/bin/compute-sanitizer mpiMultipleDevices.exe
```

GPU boost

GPU Boost is a new user controllable feature that allows us to change the processor clock speed on the Tesla K40 GPU (so far, only this card seems to be supporting it).

Essentially, the K40 has a list of available clock speeds, and we can set any of them by using the NVIDIA's System Management Interface in the command line (`nvidia-smi`).

The major disadvantage that this technology has is that it has to be run as root - but it can be set up so normal users can use it as well.

GPU boost

To request the list of available clock speeds in the card, we execute:

```
nvidia-smi -q -d SUPPORTED_CLOCKS
```

This will give us the list of available GPUs, whether they support GPU boost or not, and all the available clock speeds for both the streaming multiprocessors and the memory.

GPU boost

To get an idea about what we can do with this, the default value for the SMs of a Tesla K40 is 745Hz, and the list of available speeds is 666,745,810 and 875 MHz - so we can improve the performance of our code without even having to change the code.

We can set a different clock speed by running (as superuser):

```
sudo nvidia-smi --clocks <Memory_clock_speed, Graphics_clock_speed>
```

The good news is that we can't set a clock speed that isn't allowed - if we try to do that, we will get this error:

```
Specified clock combination "(MEM 3004, SM 888)" is not supported for GPU 0000:05:00.0.  
Run 'nvidia-smi -q -d SUPPORTED_CLOCKS' to see list of supported clock combinations  
Terminating early due to previous errors.
```



GPU boost

To get the current clock speeds, we can run:

```
nvidia-smi -q -d CLOCK
```

Take into account that once we set up a clock speed, the card will remain running at that speed until we reboot or unload the driver, and that we are paying the price of the higher clock speeds in energy consumption - so it is recommended to set it back to a lower value (or reset it to default) after running our cuda code (or write a script that wraps the cuda execution on the clock speed set and reset calls).

GPU boost

To reset all clocks in the card, we can run:

```
nvidia-smi -rac
```

Also, an application can also change the clock speeds during execution by using the Nvidia Management Library (NVML):

```
https://developer.nvidia.com/nvidia-management-library-nvml
```

but that requires the cuda code to be executed with appropriate permissions.

Cooperative Groups

The Cooperative Groups methodology is a new programming model that allows us to arrange together groups of threads so they can communicate between themselves.

Essentially, this allows us to create groups of threads and then synchronize them the way that we usually do with `__syncthreads()`.

It isn't perfect, though - in architectures older than Pascal and Volta, we can only organize groups of threads below blocks - but from Pascal onward, we can enable grid wide and even multi-GPU synchronizing groups.

Cooperative Groups

Volta, in particular, gives us an even wider choice, letting us set up thread groups even at cross-warp and sub-warp levels. Synchronization in this case can be conducted at thread level: you can even synchronize threads from divergent instruction paths.

Let's start with intra-block groups, as we can use this technique in any architecture.

Cooperative Groups

To be able to use Cooperative Groups, we are going to consider 6 things:

- Additional includes that we need to use for cooperative groups to work
- The data type that represents the groups of cooperating threads
- The default groups provided by CUDA (thread blocks for now)
- How to divide those default groups into new ones
- The barrier operation that synchronizes threads of a group
- How to scan for properties of a group

Overall, it is not that different from managing topologies in MPI.

Cooperative Groups

Additional includes that we need to use for cooperative groups to work:

Additional includes

```
#include <cooperative_groups.h>
using namespace cooperative_groups;
```

Setting an arch doesn't seem to be necessary at this stage, which seems a bit suspicious.

Cooperative Groups

The data type that represents the groups of cooperating threads is `thread_group` (`cooperative_groups::thread_group`, in the namespace). The default group provided by CUDA blocks is `this_thread_block()`:

The `thread_group` datatype, and the provided block thread group

```
thread_group my_block = this_thread_block();
```

Cooperative Groups

To split a thread group into smaller adjacent parts (called tiles, at least, for now), we can use tiled_partition :

The tiled_partition function

```
// This splits our block into adjacent subgroups of 32 threads each
thread_group my_tile = tiled_partition(my_block, 32);
```

One restriction here is that so far we can only use sizes that are a power of 2 and no larger than 32.

Cooperative Groups

The barrier operation is the method `.sync()` of the cooperative group class :

The `.sync` barrier operation

```
// This syncs our tile  
my_tile.sync();
```

Cooperative Groups

To scan for properties of a group, we can use the following functions:

Scanning for properties

```
unsigned size();           // Gives us the number of threads in this group
unsigned thread_rank();   // The rank of the calling thread within [0, size]
bool is_valid();          // Whether this group has violated any API constraints
```

Cooperative Groups

There is also a templated (as in C++ templates) version of tiled_partition, that we can call like this:

tiled_partition

```
thread_block_tile<32> tile32 = tiled_partition<32>(this_thread_block());  
thread_block_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```

Which add the following functionality (as we had for warps):

tiled_partition

```
.shfl()  
.shfl_down()  
.shfl_up()  
.shfl_xor()  
.any()  
.all()  
.ballot()  
.match_any()  
.match_all()
```



Cooperative Groups

Another advantage of cooperative groups is that we can check for branch divergence in a given warp :

Scanning for branch divergence

```
coalesced_group active = coalesced_threads();
```

In this case, active would be the thread group of threads that are all active on the instruction path when it is called (and they are called coalesced threads).

Cooperative Groups

We can also use a grid-wide cooperative group to enable sync-wide synchronization:

Grid wide synchronization

```
grid_group grid = this_grid();  
.....  
grid.sync();
```

However, this requires us to call the kernel by using a function called `cudaLaunchCooperativeKernel`, to use special compiler and linker options (`-arch=sm_61 -rdc=true`), and we should check that the card supports it (CC >6.0) - and then, we need to use a number of blocks that can be in execution at the same time - so we need to check how many blocks can fit in a SM and how many SMs we have in the card.

Cooperative Groups

It is also possible to perform multi-device synchronization, but the amount of requirements and changes needed are even larger.

For example, the same kernel has to be called in all devices, with block, grid, and dynamically allocated shared memory being the same for all the devices.

We also need to call the kernels by using an special function - cuLaunchCooperativeKernelMultiDevice, in this case.

Compilation flags

Recommended compilation flags:

Depending on architecture

====> On a RTX 3090:

```
NVCCFLAGS:= --use_fast_math -O4 -gencode arch=compute_86,code=sm_86  
          --extra-device-vectorization
```

====> On a RTX 2080Super:

```
NVCCFLAGS:= --use_fast_math -O4 -gencode arch=compute_75,code=sm_75  
          --extra-device-vectorization
```

Let's check the difference with debug flags!

You can get a full description of all the nvcc compiler options here:

<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/>

As of cuda 12.4.1, you want to read part 4.2.7: Options for Steering GPU Code Generation.

Compilation flags

The "`--gen-code arch=compute_xx,code=sm_xx`" option is equivalent to both "`--gpu-architecture=arch --gpu-code=arch`", so it is usually better to use it rather than the other options...

The "`--use_fast_math`" and "`-O4`" flags work just like in the CPU.

Compilation flags

Other compilation flags that might be useful:

The "-resource-usage" or "-res-usage)" will give you some extra information about the resources used by your code.

The "-maxrregcount", which sets the maximum amount of register space that is allowed, can be used to check if memory is being spilled into the local memory - or if the amount of register space reported by NSight Compute is actually correct.

Introduction to OpenCL

An introduction to OpenCL

In this chapter, we are going to have a look at OpenCL

Our current knowledge of CUDA is going to be pretty useful, as some things are pretty similar, but not all of them.

Introduction to OpenCL

The biggest difference that we find, when moving from CUDA to OpenCL, is that the source code doesn't necessarily have to be organized in the same way.

In CUDA, we have to set our CUDA source code in .cu files, and then compile it with nvcc.

In OpenCL, we set our OpenCL source code in .cl files, but they can be compiled during execution.

That means that, during execution, we can set up a virtual machine that will load the .cl file and compile it.

Introduction to OpenCL

Also, one word of warning - OpenCL is designed for C, not for C++, which can be a problem if you want to add the GPU code to an already existing C++ project.

There is a C++ for OpenCL project (latest version is C++ for OpenCL 2021), which is built on top of OpenCL 3.0 and implements up to C++17.

You can find the documentation for that project here:

https://www.khronos.org/opencl/assets/CXX_for_OpenCL.html

Introduction to OpenCL

Be careful! There are a few C++ features that are not supported. In the last version (C++ for OpenCL 2021), they are the following: Virtual functions, References to functions, Pointers to class member functions (in addition to the regular non-member functions that are already restricted in OpenCL C), Exceptions, dynamic_cast operator, Non-placement new and delete operators, thread_local storage class specifier and Standard C++ libraries.

Introduction to OpenCL

Quite obviously, the biggest advantage of OpenCL over CUDA is that you can run OpenCL code in many more devices, and not only GPUs (Intel Xeon Phi cards and IBM PowerLinux systems as well) - you can find a list of OpenCL compliant products in the Khronos Group webpage.

In fact, since some CPUs also run OpenCL, you can run OpenCL kernels in both devices at the same time.

OpenCL data structures

Platform (`cl_platform_id`): A platform id allows us to access the devices - think that it could store something like "intel cpu", "nvidia gpu" or "ati gpu".

Device (`cl_device_id`): An OpenCL device is each of the devices that can execute a kernel. Usually, this will be each particular CPU or GPU.

Kernel (`cl_kernel_id`): OpenCL kernels are similar to the CUDA ones - we give devices kernels to execute.

OpenCL data structures

Program (`cl_kernel_id`): An OpenCL program is the list of functions from where the kernels are picked.

Command queue (`cl_command_queue`): A command queue for a device consists of a queue of kernels, that specify the order in which they are going to be executed.

Context (`cl_context`): A context is generated for a number of devices, and allows them to receive kernels and transfer data in and out.

OpenCL Execution model

Kernels are executed by at least one work-item, and work-items are organized into work-groups.

Work-items would be equivalent to CUDA threads.

Work-groups would be equivalent to CUDA blocks.

OpenCL Memory model

Memory has to be stored in one of the four following types:

Global memory, which would be equivalent to the CUDA memory of the same name.

Constant memory would be like CUDA's global memory, but read-only.

Local memory, which is like the CUDA shared memory.

Private memory, which would be equivalent to the CUDA per-thread registers and local memory.

Add vectors program in OpenCL

AddVectorsOpenCl.c

```
int main (void) {
    // OpenCL structures
    cl_device_id device;
    cl_context context;
    cl_program program;
    cl_kernel kernel;
    cl_command_queue queue;
    cl_int i, err;
    size_t local_size, global_size;

    // Data and buffers
    int problem_size_cpu = ARRAY_SIZE;
    float data1_cpu[ARRAY_SIZE];
    float data2_cpu[ARRAY_SIZE];
    float final_cpu[ARRAY_SIZE];
    cl_mem data1_gpu, data2_gpu;
    cl_mem final_gpu; // Final result
    cl_mem size_gpu; // Total problem size
    cl_int num_groups;
```

Add vectors program in OpenCL

AddVectorsOpenCl.c

```
// Initialize data
for(i=0; i<problem_size_cpu; i++) {
    data1_cpu[i] = (1.0f*i)+1;
    data2_cpu[i] = (2.0f*i)+2;
    final_cpu[i] = 0.0;
}

// Create device and context
device = create_device();
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
if(err < 0) {
    perror("Couldn't create a context");
    exit(1);
}
```

So `clCreateContext` creates the context, but, what does `create_device` do? Turns out it isn't an OpenCL function call, but one that is written by us:

Add vectors program in OpenCL

AddVectorsOpenCl.c

```
// Find a GPU or CPU associated with the first available platform */
cl_device_id create_device() {
    cl_platform_id platform;
    cl_device_id dev;
    int err;

    // Identify a platform
    err = clGetPlatformIDs(1, &platform, NULL);
    if(err < 0) {
        perror("Couldn't identify a platform");
        exit(1);
    }
}
```

So `clGetPlatformIDs` requests and creates the platform - we will need this to set up a device.

Add vectors program in OpenCL

AddVectorsOpenCl.c

```
// Access a device
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &dev, NULL);
if(err == CL_DEVICE_NOT_FOUND) {
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &dev, NULL);
}
if(err < 0) {
    perror("Couldn't access any devices");
    exit(1);
}

return dev;
}
```

And `clGetDeviceIDs` sets up the device in the obtained platform.

Add vectors program in OpenCL

The next thing that we need to do is to create a context:

```
context = clCreateContext(NULL, 1, &device , NULL, NULL, &err);
if(err < 0) {
    perror("Couldn't create a context");
    exit(1);
}
```

Contexts are similar to the cuda ones (except for the fact that in cuda, usually, we do not need to specify them), they allow us to state kernels to be run and to transfer data into and out of the gpu.

Add vectors program in OpenCL

The next thing that we need to do is to create a program.

OpenCL programs are kind of similar to the pointers to functions that we usually get in C or C++, with the major difference that in this case, we have to store them into a `cl_program` structure.

There are a number of ways to store the device code in OpenCL - for this sample case, we are going to load it from an `.cl` file while we are executing the host code - yet again, this is something that we do not do in cuda.

Add vectors program in OpenCL

```
cl_program build_program(cl_context ctx , cl_device_id dev, const char* filename) {  
    cl_program program;  
    FILE *program_handle;  
    char *program_buffer, *program_log;  
    size_t program_size, log_size;  
    int err;  
  
    // Read program text file and place content into buffer (char*)  
    program_handle = fopen(filename , "r");  
    if(program_handle == NULL) {  
        perror("Couldn't find the program file");  
        exit(1);  
    }  
    fseek(program_handle , 0, SEEK_END);  
    program_size = ftell(program_handle);  
    rewind(program_handle);  
    program_buffer = (char*) malloc(program_size + 1);  
    program_buffer[program_size] = '\0';  
    fread(program_buffer, sizeof(char), program_size, program_handle);  
    fclose(program_handle);
```

Add vectors program in OpenCL

Then, we need to create the program object:

```
// Create program from file
program = clCreateProgramWithSource(context, 1,
    (const char**)&program_buffer, &program_size, &err);
if (err < 0) {
    perror("Couldn't create the program");
    exit(1);
}
free(program_buffer);
```

Add vectors program in OpenCL

Then, build the program:

```
// Build program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if(err < 0) {
    // Find size of log and print to std output
    clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG,
        0, NULL, &log_size);
    program_log = (char*) malloc(log_size + 1);
    program_log[log_size] = '\0';
    clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG,
        log_size + 1, program_log, NULL);
    printf("%s\n", program_log);
    free(program_log);
    exit(1);
}
return program;
```

Add vectors program in OpenCL

Then we can proceed to set up our work items and work groups:

```
// Set up work items and work group sizes:  
global_size = problem_size_cpu; // This is the total size of the problem  
local_size = 1024;  
num_groups = global_size/local_size;
```

And allocate the global memory:

```
// Create global data buffer  
data1_gpu = clCreateBuffer(context, CL_MEM_READ_ONLY |  
    CL_MEM_COPY_HOST_PTR, ARRAY_SIZE * sizeof(float), data1_cpu, &err);  
data2_gpu = clCreateBuffer(context, CL_MEM_READ_WRITE |  
    CL_MEM_COPY_HOST_PTR, num_groups * sizeof(float), data2_cpu, &err);  
if (err < 0) {  
    perror("Couldn't create a buffer");  
    exit(1);  
};
```

Add vectors program in OpenCL

...and we create the command queue:

```
// Create a command queue
queue = clCreateCommandQueue(context, device, 0, &err);
if (err < 0) {
    perror("Couldn't create a command queue");
    exit(1);
};
```

..the kernel...

```
// Create a kernel
kernel = clCreateKernel(program, KERNEL_FUNC, &err);
if (err < 0) {
    perror("Couldn't create a kernel");
    exit(1);
};
```

Add vectors program in OpenCL

Set up the kernel arguments (as if we were calling the kernel from the command line!):

```
// Create kernel arguments
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &data1_gpu);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &data2_gpu);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &final_gpu);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &size_gpu);
if(err < 0) {
    perror("Couldn't create a kernel argument");
    exit(1);
}
```

Then put the kernel in the execution queue:

```
// Enqueue kernel
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size,
                             &local_size, 0, NULL, NULL);
if(err < 0) {
    perror("Couldn't enqueue the kernel");
    exit(1);
}
```

Add vectors program in OpenCL

Copy the data back into the host

```
// Read the kernel's output
err = clEnqueueReadBuffer(queue, sum_buffer, CL_TRUE, 0,
    ARRAY_SIZE * sizeof(float), final_cpu, 0, NULL, NULL);
if(err < 0) {
    perror("Couldn't read the buffer");
    exit(1);
}
```

And we shouldn't forget to free all the OpenCL objects:

```
// Deallocate resources
clReleaseKernel(kernel);

clReleaseMemObject(data1_gpu);
clReleaseMemObject(data2_gpu);
clReleaseMemObject(final_gpu);
clReleaseMemObject(size_gpu);

clReleaseCommandQueue(queue);
clReleaseProgram(program);
clReleaseContext(context);
```



Add vectors program in OpenCL

Now, if we have a look at the .cl file:

AddVectorsOpenCl.cl

```
__kernel void addVectors( __global float* data1,__global float* data2,
 __global float* sum_result,__global int* problemSize) {

    float sum;
    float input1 , input2 , sum_vector;
    uint global_addr , local_addr;

    global_addr = get_global_id(0);
    if (global_addr<problemSize[0]) {
        input1 = data1[global_addr];
        input2 = data2[global_addr];
        sum_vector = input1 + input2 ;
        sum_result[global_addr]=sum_vector;
    }
}
```

So: the kernel is named in pretty much the way that we are used to, memories are specified (global, local, etc) and we can get the global and local ids with function calls.

Add vectors program in OpenCL

AddVectorsOpenCl.cl

```
local_result[local_addr] = sum_vector.s0 + sum_vector.s1 +
                           sum_vector.s2 + sum_vector.s3;
barrier(CLK_LOCAL_MEM_FENCE);
if(get_local_id(0) == 0) {
    sum = 0.0f;
    for(int i=0; i<get_local_size(0); i++) {
        sum += local_result[i];
    }
    group_result[get_group_id(0)] = sum;
}
}
```

And we see here how to use barriers.

How to scan for more than one platform

So, let's go into a bit more of detail - the very first thing that we do is identify the platforms that are available. We used to be able to get them as a vector (the C++ template), in the following way (not sure if this works anymore!):

```
cl::vector< cl::Platform > platformList;
cl::Platform::get(&platformList);
cout << "platformList.size()=" << platformList.size() << endl;
if (platformList.empty()) {
    std::cerr << "No OpenCL platforms were found." << std::endl;
    return 1;
}
// Other way to do this with checkErr
checkErr( platformList.size()!=0 ? CL_SUCCESS : -1, "cl::Platform::get" );
```

How to scan for more than one platform

Another one of the differences with cuda is that we can actually run OpenCL in both GPUs and CPUs - and in fact, in many cases we might want to do both at the same time if we can, so all the other OpenCl structures should also be resized to the number of available platforms.

```
unsigned int ui;
cl::vector< std::string > platformVendors (platformList.size ());
cl::vector< cl_platform_id > platformIds (platformList.size ());
cl::vector< cl_device_id > deviceIds (platformList.size ());
cl::vector< bool > deviceIsGpu (platformList.size ());
cl::vector< cl_context > contexts (platformList.size ());
```

OpenCL Error Checking

OpenCL error checking works in a fairly similar way than in CUDA; in this case, instead of having a state in the card that we can request and test, we have the return value of the OpenCL API calls.

Like in CUDA, we can write a function to test that returned value:

```
inline void checkError(cl_int error, const char * myString) {
    if (err != CL_SUCCESS) {
        std::cerr << "ERROR: " << name << " (" << err << ")" << std::endl;
        exit(EXIT_FAILURE);
    }
}
```

OpenCL - global memory usage

Global memory is pretty much the same as the CUDA global memory - and this is how we allocate, transfer it to the gpu:

```
// In the .c file
// Allocate and transfer
global_memory_gpu = clCreateBuffer( context , CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                                    array_size * sizeof(float) , global_memory_cpu , &error);
...
// Set it so it is passed as an argument to the kernel
error = clSetKernelArg(kernel , 0, sizeof(cl_mem) , &global_memory_gpu);
```

and use it in the kernel:

```
__kernel void globalMemoryKernel(      __global float* global_memory_gpu) {
    // Do something
    global_memory_gpu[0]=1;
}
```

OpenCL - constant memory usage

Constant (which would be more or less equivalent to texture memory in CUDA - probably with less memory size restrictions) is also very easy to use:

```
// In the .c file
// Allocate and transfer
constant_memory_gpu = clCreateBuffer( context , CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                      array_size * sizeof(float) , global_memory_cpu , &error );
...
// Set it so it is passed as an argument to the kernel
error = clSetKernelArg(kernel , 0 , sizeof(cl_mem) , &constant_memory_gpu );
```

and then use it in the kernel like this:

```
__kernel void constantMemoryKernel( __constant float* constant_memory_gpu ) {
    // Do something
    float float_variable;
    float_variable=constant_memory_gpu[0];
}
```

OpenCL - local memory usage

Local memory is a bit different. We have to specify its size on the .c code when passing it to the kernel, like this:

```
// local memory i- note the value of NULL and different size  
error = clSetKernelArg(kernel , 0, local_size * sizeof(cl_uint), NULL);
```

and then use it in the kernel like this:

```
__kernel void localMemoryKernel( __local uint* local_array) {  
    // Do something  
    uint global_addr,local_addr;  
    local_addr = get_local_id(0);  
    local_array[local_addr]=2+local_addr;  
}
```

OpenCL - private memory usage

Private memory is also quite easy to use, just like CUDA's registers - any variable you allocate inside the kernel is, by default, in it:

```
__kernel void privateMemoryKernel( ) {
    __private uint global_addr,local_addr,work_dim;
    global_addr = get_global_id(0);
    local_addr = get_local_id(0);
    work_dim = get_work_dim();
}
```

2d work groups

To use 2d work groups and work items, we need to do a few things - the first one is to define arrays for the `global_work_size` and `local_work_size` values.

```
size_t local_size[2], global_size[2];  
  
global_size[0] = numberOfWorkElements[0];  
global_size[1] = numberOfWorkElements[1];  
local_size[0] = local_size[0];  
local_size[1] = local_size[1];
```

and then, when calling `clEnqueueNDRangeKernel`, pass a 2 as the third argument (which is a `cl_uint work_dim`):

```
error = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global_size, local_size, 0, NULL, N
```

2d work groups

And then, in the kernels, we can get our 2d local and global id by passing the dimension as a parameter to the `get_global_id` or `get_local_id` functions, and from them figure out our position in the 2d array as usual:

```
__private uint global_addr_x,local_addr_x;
__private uint global_addr_y,local_addr_y;
__private uint global_addr;

global_addr_x = get_global_id(0);
global_addr_y = get_global_id(1);

local_addr_x = get_local_id(0);
local_addr_y = get_local_id(1);

global_addr = global_addr_y+global_addr_x*N[1];
```

3d work groups

And for the 3d case, we just extend that to three dimensions instead of 2.

```
size_t local_size[3], global_size[3];  
  
global_size[0] = numberOfRows[0];  
global_size[1] = numberOfRows[1];  
global_size[2] = numberOfRows[2];  
local_size[0] = local_size[0];  
local_size[1] = local_size[1];  
local_size[2] = local_size[2];
```

and then, when calling `clEnqueueNDRangeKernel`, pass a 3 as the third argument (which is, again, a `cl_uint work_dim`):

```
error = clEnqueueNDRangeKernel(queue, kernel, 3, NULL, global_size, local_size, 0, NULL, N
```

3d work groups

And again, we can get our 3d local and global id by passing the second and third dimensions as a parameter to the `get_global_id` or `get_local_id` functions, and from them figure out our position in the 3d array as usual:

```
__private uint global_addr_x,local_addr_x;
__private uint global_addr_y,local_addr_y;
__private uint global_addr_z,local_addr_z;
__private uint global_addr;

global_addr_x = get_global_id(0);
global_addr_y = get_global_id(1);
global_addr_z = get_global_id(2);

local_addr_x = get_local_id(0);
local_addr_y = get_local_id(1);
local_addr_z = get_local_id(2);

global_addr = global_addr_z+global_addr_y*N[2]+global_addr_x*N[2]*N[1];
```

OpenMP for GPUs

In this chapter, we are going to have a look at how to use OpenMP to run things on GPUs.

The main problem that we have with OpenMP for GPUs is that the code that we write for GPUs is different to the OpenMP code that we write to run on CPUs, which means that it goes against the main principle of OpenMP programming, which is to keep a single code base in which you tell OpenMP what you want to parallelize and then let it do it for you in different architectures. The problem here, as it is usual in GPGPU, is memory management in the GPU.

Installing OpenMP for GPUs

Installing OpenMP for GPUs is actually much harder than the default OpenMP, so here are some steps that might help:

First, we need to install the libomp-dev library, in a way similar to this:

```
sudo apt-get install libomp-dev
```

to check that it installed correctly:

```
dpkg --status libomp-dev
```

Installing OpenMP for GPUs

Then, we need to install the gcc-offload-nvptx library:

```
sudo apt install gcc-offload-nvptx
```

That should install nvptx-tools. Remember that you will have to tell GCC where to find it:

```
--with-build-time-tools=[install-nvptx-tools]/nvptx-none/bin.
```

Installing OpenMP for GPUs

Then, we need to compile gcc for GPUs:

```
sudo apt install gcc-offload-nvptx
```

That *should* install nvptx-tools. Remember that you will have to tell GCC where to find it:

```
--with-build-time-tools=[install-nvptx-tools]/nvptx-none/bin.
```

If that doesn't work (and it is pretty likely it won't), you will need to download, compile and install nvptx-tools and newlib-cygwin on your own - which usually involves a lot of googling errors!

Basic OpenMP for GPUs

Either way, once you have it installed, your Makefiles will look like this:

```
CC= /home/jose/gcc-offload/install/bin/gcc
CFLAGS= -g -pedantic -W -Wall -L/home/jose/gcc-offload/install/lib64 /
INCPATH      = -I. -I.

TARGET=addVectorsCpp.o
EXEC=prog_debug

all: $(TARGET)
    $(CC) -Wall -o ${EXEC} ${TARGET} -fopenmp -foffload=-lm

%.o: %.cpp Makefile
    $(CC) $(CFLAGS) -c $(INCPATH) -fopenmp -foffload=-lm $<

install:

clean:
    rm -f *.o ${TARGET}
```

Basic OpenMP for GPUs

While your code will look like:

```
int main() {  
    int i;  
    // Allocate arrays a, b and c on host  
    float *a, *b, *c;  
    a = (float*) malloc(N*sizeof(float));  
    b = (float*) malloc(N*sizeof(float));  
    c = (float*) malloc(N*sizeof(float));  
    // Initialize arrays a and b  
    for (i=0; i<N; i++) {  
        a[i]= (float) i;  
        b[i]=+(float) 2*i;  
    }  
    #pragma omp target map(tofrom: a[0:N], b[0:N], c[0:N])  
    for (i=0; i<N; i++) {  
        c[i]= a[i]+b[i];  
    }  
  
    // Print c  
    printf("addVectorsfloat will generate two vectors, and add them together in the GPU\n");  
    for (i=0; i<N; i++) {  
        printf(" a[%2d](%10f) + b[%2d](%10f) = c[%2d](%10f)\n", i, a[i], i, b[i], i, c[i]);  
    }  
  
    // Free the memory  
    free(a); free(b); free(c);  
}
```

Basic OpenMP for GPUs

So, as you can see, we still have to move the data to and from the GPU, using a syntax that is very different to our usual OpenMP, as we will use `pragma omp target map(tofrom: a[0:N], b[0:N], c[0:N])`

This remains a problem in even more advanced OpenMP for GPUs - we will still have to specify many GPU operations to get the best performance.

Also, as we learnt from the first assignment, changing the order of our loops might improve or decrease the performance considerably.

Do not forget to double check with `nvprof` that it is actually running on the GPU!

Basic OpenMP for GPUs

Don't forget that the code gets lot more complicated as we need to add instructions to increase the level of parallel optimization - a not too far reaching example would be:

```
void addVectors(float* a, float* b, float* c, int size) {
#pragma omp target teams map(to:a[0:size], b[0:size]) map(from:c[0:size]) num_teams(ntteams)
{
    int blockSize = n / omp_get_num_teams(); // n assumed to be multiple of #teams
#pragma omp distribute
    for (int i = 0; i < size; i += blockSize) {
        #pragma omp parallel for simd firstprivate(i,blockSize)
        for (int j = i; j < i + blockSize; j++) {
            c[j] = a[j] + b[j];
        }
    }
}
```

Basic OpenMP for GPUs

We can also add Asynchronous calls, which would look like this:

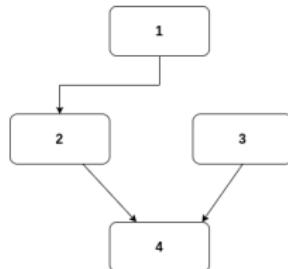
```
#pragma omp task depend(out:a)
calculate_1(a);

#pragma omp target map(to:a[:N]) map(from:x[:N]) nowait depend(in:a) depend(out:x)
calculate_2(a, c, N);

#pragma omp target map(to:b[:N]) map(from:z[:N]) nowait depend(out:b)
calculate_3(b, d, N);

#pragma omp target map(to:y[:N]) map(to:z[:N]) nowait depend(in:x) depend(in:b)
calculate_4(d, c, b, N);

#pragma omp taskwait
```



OpenACC for GPUs

OpenACC is going to be fairly similar - you used to have to download and install the pgi linux tools, but nowadays, you just have to install the NVIDIA HPC Software Development Kit the following way:

```
$ wget https://developer.download.nvidia.com/hpc-sdk/21.3/nvhpc-21-3_21.3_amd64.deb
```

Then run:

```
$ sudo apt-get install ./nvhpc-21-3_21.3_amd64.deb ./nvhpc-2021_21.3_amd64.deb
```

Basic OpenACC for GPUs

In this case, the Makefile will look like this (paths might differ):

```
CC= /opt/pgi/linux86-64/19.10/bin/pgcc
CFLAGS= -acc

INCPATH      = -I. -I.

TARGET=addVectorsOACC.o
EXEC=prog_debug

%.o: %.c %.cpp Makefile
    $(CC) $(CFLAGS) $< -c $(INCPATH)

all: $(TARGET)
    $(CC) $(CFLAGS) ${TARGET} -o ${EXEC}

install:

clean:
    rm -f *.o ${TARGET}
```

Basic OpenACC for GPUs

Whereas code will look very much like the OpenMP:

```
int main( int argc, char* argv[] ) {
    int i,n = 200000000;
    double * restrict a, *restrict b, *restrict c;
    a = (double*) malloc(n*sizeof(double));
    b = (double*) malloc(n*sizeof(double));
    c = (double*) malloc(n*sizeof(double));

    // Initialize content of input vectors, vector a[i] = sin(i)^2 vector b[i] = cos(i)^2
    for(i=0; i<n; i++) {
        a[i] = sin(i)*sin(i);
        b[i] = cos(i)*cos(i);
    }
#pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
    for(i=0; i<n; i++) {
        c[i] = a[i] + b[i];
    }
    // Sum up vector c and print result divided by n, this should equal 1 within error
    double sum = 0.0;
    for(i=0; i<n; i++) {
        sum += c[i];
    }
    sum = sum/n;
    printf("final result: %f\n", sum);

    free(a); free(b); free(c);
}
```

Basic OpenACC for GPUs

So, again, we still have to move the data to and from the GPU, in this case with a pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])

Again, this is even more the case the deeper that you delve into OpenACC on GPUs - we will still have to plan and specify GPU memory operations to get the best performance.

Try changing the order of the loops and see if that affects performance!

Yet again, after compiling, do not forget to double check with nvprof that it is actually running on the GPU!

Review of Course Topics

- 1 Introduction to CUDA
- 2 Basic CUDA programming
- 3 Advanced CUDA programming
- 4 Advanced CUDA technology
- 5 Advanced CUDA programming, revisited
- 6 Introduction to OpenCL
- 7 Introduction to OpenMP for GPUs and OpenACC