Practical Parallel Programming – Jan Lemeire

The Power of GPUs

Here we give a short introduction to GPU computing, explaining the basics of the GPU architecture which makes the enormous computational power available.

You don't have to know all the terminology but understand the important architectural differences with the CPU.

History of GPGPUs

A GPU is a graphics processing unit. Its main purpose is to show images on the screen and to carry out the necessary computations to do so. During the last 20 years the computational power of GPUs has increased enormously, and several people started to look at the possibility to use the *GPU for general purpose computations*, hence the term **GPGPU**. Initially it was necessary to cast these general-purpose problems into a graphic framework i.e. it was necessary to write the code using for example OpenGL or DirectX, APIs used to program computer graphics. This approach was very cumbersome, and efforts were made to design and implement ways to ease the creation of general-purpose programs for the GPU.

In November 2006 the first commercially available framework for general purpose computing appeared: **CUDA** from NVIDIA. CUDA stands for ``Compute Unified Device Architecture'' and presents itself as a set of extensions to the C programming language and a runtime library. CUDA is proprietary and can only be used with NVIDIA GPUs.

OpenCL is an open standard that provides a framework for writing programs that execute across heterogeneous platforms. A heterogeneous platform is basically defined as a platform that contains more than one computing device, for example a CPU and a GPU. OpenCL was initially developed by Apple, but it has become an open standard that is maintained by the non-profit technology consortium Chronos Group, that also maintains OpenGL. Version 1.0 of OpenCL was released in 2008. At the moment of writing, OpenCL is at version 2.1 and there are several commercial implementations available from for example NVIDIA, AMD, Intel and IBM.

Computational performance

While a CPU can only execute a few computations at the same time, a GPU has a lot of **Processing Elements** (PEs) which are called **CUDA Cores** by Nvidia (in their GPU data sheets). The (theoretical) peak performance can then be calculated by multiplying this number with the clock frequency:

```
Operations per second = PEs x clock frequency          (1)
```

These PEs refer to Single Precision (**SP**) computational units. SP also stands for Scalar Processors: they compute an operation on single values. Not on vectors (SIMD) although it happens in a special way: SIMT as we will discuss later.

A GPU is a collection of **Compute Units** (CUs), called **MultiProcessors** (MPs) or **Streaming Multiprocessors** (SMs) by Nvidia. The layout of a CU depends on its **architecture (generation)**. The number of PEs for a CU is fixed for each architecture. The performance can thus also be calculated as:

```
Operations per second = CUs x PEperCU x clock frequency      (2)
```

Native transcendental functions (like cos or pow) are executed on a special **SFU** unit. Also, double precision data is calculated by specific **DP** units. With the number of such units on a CU, the peak performance can be calculated by Equation 2.

The website https://www.techpowerup.com/gpu-specs/ provides you the numbers for your GPU.

The following table lists the *characteristics of a CU* for the Nvidia architectures. Since the exact number of CUs depends on the specific GPU, the peak performance cannot be mentioned here. We also mention the theoretical RAM bandwidth and the estimated SP completion latency (discussed later).

Nvidia Architecture	Clock free MHz	PEs per CU	SFUs per CU	DPs per CU	RAM band-width (GBs)	latency Λ_{SP} (cycles)
Tesla		8	?	-		24
Fermi	1147	32	8	-	144	18
Kepler	1032	192	32	64	86	9
Maxwell	1058	128	32			6
Pascal	1506	128	32	64	192	6
Turing		64	8	?		

Programming the GPU

A GPU program is a massive multi-threaded program. An important task is to decide what each thread will do. For most algorithms, this is done by assigning different parts of the data to each thread, i.e. defining a mapping of the threads on the data.
Like the threads, the data items are laid out in a 1-, 2- or 3-dimensional structure. 1-dimensional data structures are often referred to as arrays and 2-dimensional data structures as matrices. Most often it is best to use the same structure for data as for the threads. We will consider 1-dimensional structures here.

The **one-to-one mapping** is the simplest mapping: it maps one thread onto one data item. Here an array of 24 elements and 24 threads:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

The following code in OpenCL illustrates this mapping for W = 24. W defines the array size and the number of threads. `data_index` is an array of size W and will be filled with the IDs of the threads. The array is passed as a parameter to the thread. The resulting array is shown in the figure.

```
// host code (on CPU)
unsigned int data_index[W]; // create array
cl::NDRange global(W);  // number of threads

// device code for each thread (on GPU)
unsigned int index = get_global_id(0); // retrieve thread ID
data_index[index] = index; // fill array
```

The **one-to-many mapping** maps one thread to *many* data items. The simplest way to do this is to assign contiguous data items to the same thread. This results in the following mapping:

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11

Thread 0 is mapped onto elements 0 and 1, thread 1 onto 2 and 3, …

We will not dig deeper into programming GPUs, but try to understand when and why GPUs beat CPUs.

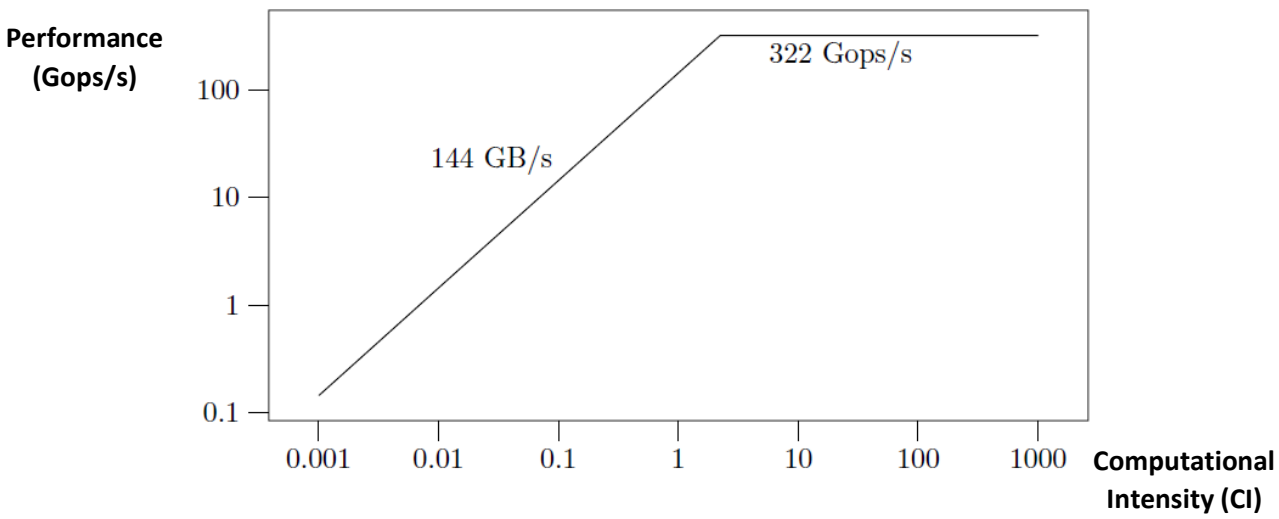
Memory Performance and Roofline Model

The tremendous computational power can only be exploited if the memory systems can provide the data in time: a certain memory bandwidth is needed to achieve the peak computational performance. Also, the possibilities for caching should be maximally used, as we will discuss later.

The performance will drop if the necessary memory bandwidth is not available. The **roofline model** provides an easy graphical representation to analyses this for a particular system. The idea of the roofline model is to predict the maximum attainable performance of a code on a system by determining whether it is **memory bound** or **compute bound**. If the code is memory bound, the maximum attainable performance is determined by the memory access subsystem. Otherwise, it is determined by the computational subsystem.

The parameter of the code that determines whether it is memory or compute bound is its **Computational Intensity** (CI). This is the number of computational operations performed per byte accessed in memory. The performance limit determined by the computational subsystem is simply the computational subsystem throughput as we calculated previously. When the performance is limited by the memory access subsystem, the computational performance is the product of the code's CI and the memory access subsystem data throughput (bandwidth BW): CI * BW. The actual maximum attainable performance will be the minimum of those two limits.

The roofline model is modeling the hardware, while the software is characterized by its computational intensity. The following figure shows the roofline model for the NVIDIA GeForce GTX280 and explains where the model got its name from. Note the logarithmic scales for the X and Y axes.



The CI should be at least 3-4 to achieve maximal performance. This is called the **ridge point**.
The roofline model can serve as a first estimation of the *potential* performance. As we will discuss later, there are several performance limiters that make it hard to achieve this performance.

The modern CPU

In 1989 Intel released a *pipelined* processor with the Pentium CPUs. Later it turned it into an *out-of-order superscalar pipeline* (having a certain width). The depth of a pipeline enables shorter cycles (and a higher clock frequency), the width enables multiple instructions to be issued together. In short, the performance is multiplied with (depth x width). Note that there are (depth x width) instructions active at the same time. We say that the pipeline is *completely filled*.

However, this is only possible when there are enough independent instructions within *the same program* (called Instruction Level Parallelism), because a CPU is only processing 1 program (or 2 if it supports hardware threads like hyperthreading) at the same time. With dependent instructions or conditional branches, this is no longer true, because the result of previous instructions is needed to begin new instructions. When no novel instruction can be launched (issued), a 'bubble' will be inserted into the pipeline. We say that the pipeline is **idling** and that cycles are 'lost' because an instruction could have been issued.

To counter this problem, a modern CPU performs several optimizations to minimize the end-to-end latency of instructions:

- Forwarding: the outcome of a computation (in the ALU) is immediately fed into the input of the ALU for calculating the following instruction
- Branch prediction: the outcome of a branch is guessed so that following instructions (of the branch) can be issued. For incorrect guesses, the calculated values are rolled back.
- Caching: data is copied for reuse into faster memory.

Needless to say, a lot of additional transistors are needed to perform these runtime optimizations. A CPU is devoted to optimally run a *sequential* program.

GPU strategy for massive computations

The GPU abandons this strategy and uses most transistors for computations. To fill the pipelines and minimize the lost cycles, the GPU enables the parallel execution of thousands of threads. This will result in highly performant execution of fine-grained parallel programs.

The GPU as massive thread processor is based on (1) **simultaneous multithreading** and (2) **SIMT processing**. This happens on 1 compute unit (streaming multiprocessor). On top of that, a GPU has multiple compute units, each capable of executing several work groups at the same time.

Simultaneous multithreading

A typical CPU executes only one thread at a time. On modern processors 2 threads can run simultaneously (called hyperthreading by Intel): a dual core with hyperthreading is said to have 4 logical cores (which can be seen in the Task Manager when you open the Resource Monitor). The other threads will alternately get processor time. The change of the active thread is called a context switch which takes time since the processor state should be saved and the other one reloaded. The thread scheduling is managed by the operating system. Therefore, these threads are called **software threads**.

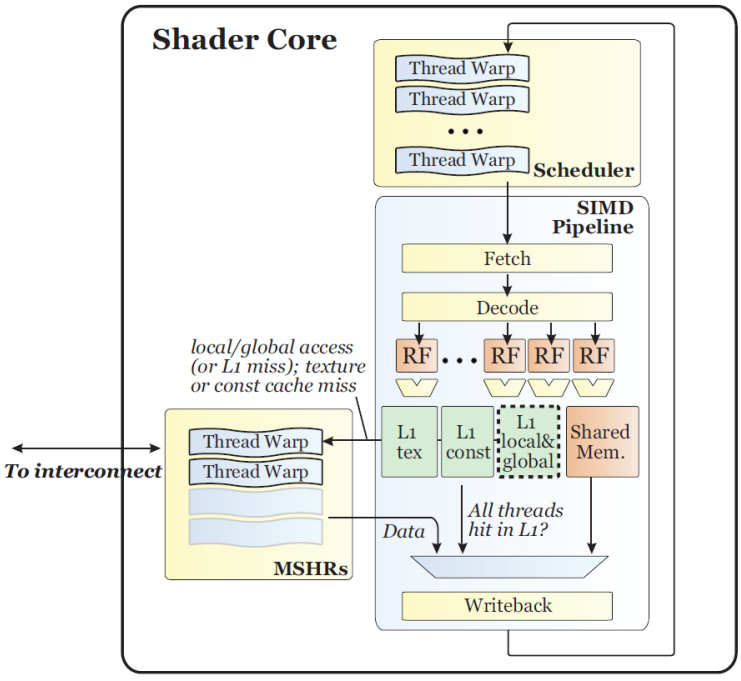
GPUs can have more than 1000 kernel threads *simultaneously* active on a single compute unit or core! These are part of **hardware threads** since the scheduling and 'switching' is completely done by the hardware. Actually, the switching is for free: instructions of different threads can be mixed together without any cost. This is called **simultaneous multithreading**. The benefit of simultaneous multithreading is **latency hiding**. While one instruction is being handled by the processor but not finished, the processor starts executing instructions of other threads. In absence of synchronization points between both threads, the instructions are independent. This mainly happens in a **pipeline** fashion. The computational subsystem is a pipeline, but also the memory system (RAM & caches) can be regarded as pipelines. An instruction starts (is issued), enters the pipeline, and new instructions can be started.

In this way the pipelines are filled with instructions of different threads and there is less need of CPU optimizations: long latencies are hidden by instructions of other threads. So, to reach the peak performance of a GPU, we should execute more threads than there are CUDA cores (also called PEs) ! The more threads the better the latencies can be hidden.

When one of the pipelines, computational or memory, is almost always full, we say that the GPU kernel is respectively **computation-** or **memory-bound**. In chapter 1 this was modelled by the roofline model.

SIMT

The pipeline of a GPU core also has a certain *width*. There are 8, 32, 128 or 192 Processing Elements (PEs, called RF in the diagram) on a Streaming Multiprocessor.



As can be seen in the diagram, there is only 1 instruction fetch and decode unit, meaning that *one* instruction is issued and executed *simultaneously* by all Processing Elements (indicated as RF in the picture). This looks like SIMD processing as with vector processors. We will however call it **Single Instruction Multiple Thread** (**SIMT**) because there are subtle differences with vector processing.

The kernel that you program will be executed by each thread. In most cases each thread works on different data elements. For instance, by using its ID as index, but this can also be more complex mappings from threads onto the data (see chapter 2). But: a number of threads will be executed in

lock step all executing the same instruction, very similar to vector processing. The main difference is that we do not have to worry about the data layout or vector instructions. We program a thread *as if it will be executed in an independent thread*. This is not the case; a hardware thread will execute a number of threads together at the same time. Nvidia GPUs execute 32 threads, called a **warp**, while AMD GPUs execute 64 threads in lock step (called wavefronts). For Intel GPUs it is more complicated, it can be 8/16/24/32 threads. Other OpenCL compilers (e.g. for CPUs) will try to vectorize groups of threads, which is only possible in case of contiguous data access.

Performance limiters

As a rule of thumb, a GPU has a peak performance 100x higher than that of 1 CPU core. But this power is often difficult to unleash. We discuss here the most important overheads.

As was already discussed, data must be transferred to the GPU and from the GPUs RAM to the processors. This takes time so that programs often are memory-bound.

Secondly, since a lot of data-hungry threads are simultaneously active, the memory system must be able to serve all the concurrent memory requests simultaneously. This is not always possible; it depends on the access pattern of the concurrent memory requests.
Shared memory is organized in banks (see slides). In order to maximize the reading and writing of shared memory the threads of a warp should access different banks. If this is not the case, the concurrent access will be serialized. This is known as *bank conflicts*.

A third performance limiter comes from threads taking part in the same SIMT not following the same sequence of instructions. If the code contains *branches* and threads of the same warp follow different branches, then the execution of these branches is serialized. Both the then- and else-part are executed. Some threads will only commit in the then-part and others to the else-part (called **conditional processing**). This results in lost cycles, because on average only half of the threads are active. This is even worse for a larger number of branches. A kernel with a while-loop might result in different iterations for each thread of the warp. The warp will execute the maximal number of iterations. So, if one thread has to do 1000 iterations and all other threads only a few iterations, 31 over 32 cycles are lost. The efficiency drops to only 3%!