GPU Computing Course – Jan Lemeire

Lesson 3: Architecture and Strategy

In this chapter we will discuss how a GPU manages to get superior computation performance although having the same number of transistors as a CPU!

1 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 see later) 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.

2 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.

2.1 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 with 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 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. 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.

On the other hand, when none of the pipelines is completely filled, i.e. there are a considerable amount of idle cycles in each subsystem, we say that the kernel is **latency-bound**. To understand this, we introduce the pipeline model.

2.1.1 The pipeline model

Consider the following code:

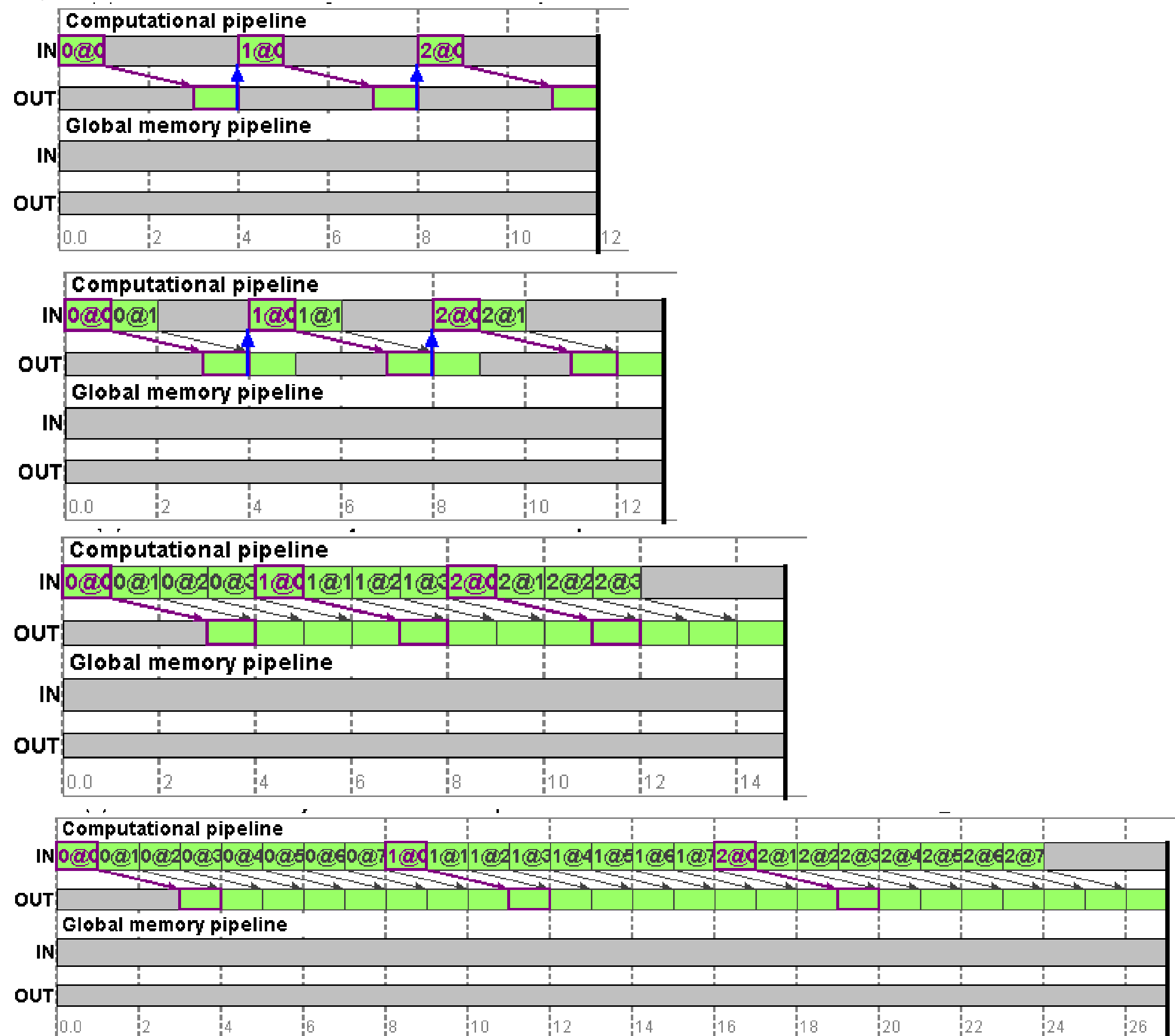
```
__kernel void scale( __global float * array, float factor)
{
        int index = get_global_id(0);
        float r = array[index ];
        r = factor * r;
        array[ index] = r;
}
```

If we want to know how long it takes to execute one instruction sequence or thread of this code, it is necessary to know how long it takes to execute every individual instruction. Because every instruction but the first depends on its predecessor, the execution time is simply the sum of the execution times of the individual instructions.
On modern processors multiple independent instructions from different threads running simultaneously can execute at the same time: instructions are executed by pipelined hardware, while multiple memory requests caused by memory instructions can be in flight at the same time. Thus, to determine the time needed to process multiple threads, it is necessary to model this ability of the processor. We do so by *representing a GPU by a set of independent pipelines*.

A pipeline consists of multiple stages each performing a part of the execution of an instruction. An instruction should go through all stages. Each stage takes the same time, making it possible to execute multiple instructions concurrently. If the time needed by a stage to handle one instruction is equal to one clock cycle, then the time needed to execute an instruction or its latency in clock cycles is equal to the number of pipeline stages. Given sufficient independent instructions, however, the pipeline's maximal throughput of one instruction per cycle can be attained. Because at each cycle a new instruction can be issued.

2.1.2 The occupancy roofline

The performance of a simple pipeline can be easily determined graphically. Consider a pipeline of 4 stages processing respectively 1, 2, 4 and 8 threads, each thread consisting of a sequence of 3 dependent instructions.

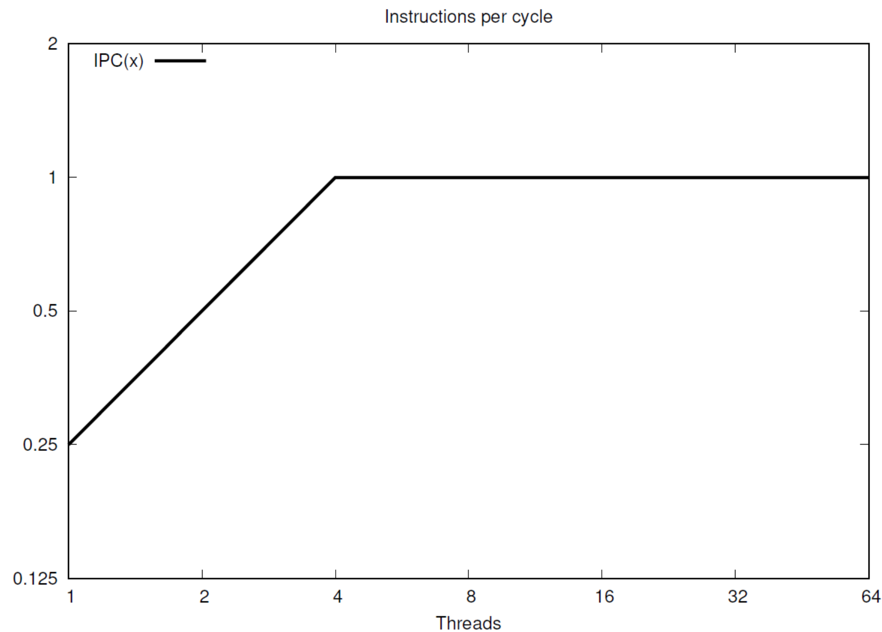


In the first case 1 instruction completes every 4 cycles, in the second case 1 instruction completes every 2 cycles, while in the third and last case 1 instruction completes every cycle. In general, the performance of a pipeline of S stages executing N threads depends on N and S.

$$IPC_{N\leq S} = S/N$$

$$IPC_{N\geq S} = 1$$

The following figure shows a plot of the performance of a pipeline with four stages in function of the number of threads.



This plot resembles a roofline and is called the **occupancy roofline**. The term occupancy is a name for the number of threads that can execute concurrently on a processor. The ridge of the roofline is the number of threads necessary to hide the pipeline latency. The region below this point is called latency bound, while the region beyond this point is called throughput bound.
The pipeline is chosen as the central abstraction used to represent the subsystems of a processor, given its ability to model concurrent execution with two simple parameters. Note that this abstraction should not necessarily reflect reality. For example, the memory access subsystem will behave more as a queueing system than as a pipeline. Each instruction is characterized by the subsystem on which it is executed and two numbers: the issue latency and the completion latency. The former is related to the peak performance of the instruction: it provides an upper bound on the instruction throughput. The latter is the time between the moment an instruction enters the pipeline and the moment it completes. As seen in the above equations, the instruction throughput is dominated by this time in the absence of sufficient independent instructions.

2.1.3 The number of concurrent work groups

A single work group will always be executed on one core (Compute Unit). If possible, multiple work groups can be active at the same time. This will lead to more instructions that can be scheduled and thus a higher occupancy.
The number of concurrent work groups is determined by a number of limited hardware resources (typical numbers are given between brackets):

1. The amount of local memory per core (e.g. 48KB).
2. The amount of registers per core (e.g. 64KB).
3. The maximum number of work items that may be active on one core (e.g. 2048).
4. The maximum number of work groups that may be active on one core (e.g. 16).

These numbers are matched with the requirements of the kernel (1 and 2) and execution configuration (3 and 4): the ratio of *required* and *available* determines how many work groups can be run simultaneously. The most restrictive ratio of the 4 will determine the actual concurrency.
The first two resources depend on the kernel: the amount of local memory you allocate for each work group and the number of registers that the compiler allocates.
The work group size (local size in openCL) determines how many work groups are possible by the 3rd resource. If local size = 256, then there are maximally (1024/256=) 4 work groups possible. If you allocate 5KB local memory, then only (16/5=) 3 work groups can run concurrently.

We define the **occupancy** of a kernel as the number of warps that are concurrently active on a core. We also call this the *number of concurrent warps*. The **relative occupancy** is the occupancy relative to the maximum occupancy.

2.1.4 Fixing the Occupancy

The occupancy of a kernel can be fixed by reserving local memory for a work group by adding an extra argument to your kernel e.g.:

```
__kernel void matrix_sum(__global float* A, __global float *B, __global
float *C, float scale, __local float *dummy)
```

Set this extra argument as follows in the host code:

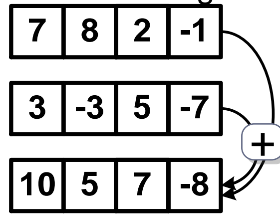
```
kernel.setArg<cl::LocalSpaceArg>(4, cl::__local(local_size));
```

The *local_size* corresponds to the number of bytes you want to reserve for shared memory. By varying local_size you can vary the occupancy. The following table shows some typical occupancy figures for different values of *local_size* and the *work group size*. It is assumed that the local memory available on one core is 16KB and that no other factors are limiting the occupancy:

Local_size (Bytes)	#Concurrent WGs	Work group size	#Concurrent warps/work items	Relative occupancy = #work items/max (1024)
4000	4	256 = 8 warps	32 / 1024	100
5000	3	256	24 / 768	75
8000	2	256	16 / 512	50
16000	1	256	8 / 256	25
2000	8	128 = 4 warps	32 / 1024	100
2280	7	128	28 / 896	87.5
2600	6	128	24 / 768	75
3200	5	128	20 / 640	62.5
4000	4	128	16 / 512	50
5000	3	128	12 / 384	37.5
8000	2	128	8 / 256	25
16000	1	128	4 / 128	12.5

2.2 Vector processors & SIMD

Let's first look at the **vector instructions** available in modern CPUs. They contain separate registers, called vector registers which each store several data elements. There are 128-bit or 256-bit and recently 512-bit wide registers.

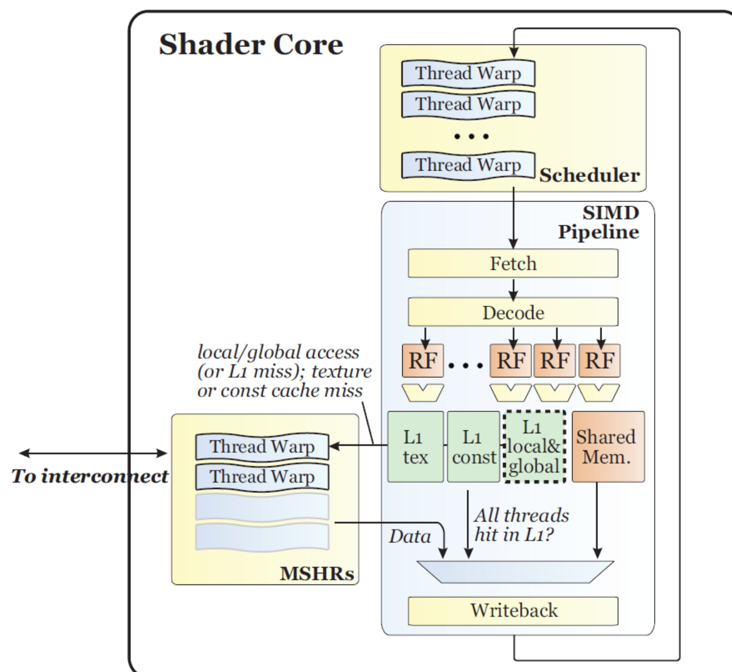


Vector instructions are performed at once on all elements of the vector registers. Instead of iterating over the vector (for-loop), one instruction is sufficient. This way of processing is called **Single Instruction on Multiple Data** (**SIMD**).

The way to program them is with specific datatypes used for reading data from memory into vector registers and specific vector operations to execute vector instructions on the vector registers. It is quite tedious to program and less flexible than OpenCL which is based on SIMT.

2.3 Hardware threads & SIMT (Level 3 of GPU programming)

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 Nvidia core (called Streaming Multiprocessor by Nvidia).



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 PEs. 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 work item. In most cases each work item works on different data elements. For instance, by using its global ID as index, but this can also be more complex mappings from work items onto the data (see chapter 2). But: a number of work items 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 work item *as if it will be executed in an independent thread*. This is not the case; a hardware thread will execute a number of work items together at the same time. Nvidia

GPUs execute 32 work items, called a **warp**, while AMD GPUs execute 64 work items in lock step (called wavefronts). For Intel GPUs it is more complicated, it can be 8/16/24/32 work items. Other OpenCL compilers (e.g. for CPUs) will try to vectorize groups of work items, which is only possible in case of contiguous data access. We will use both terms hardware thread and warp; they mean exactly the same.

While programming you might see a work item as a **kernel thread,** but since it is not a real thread, it has several consequences for the *performance* of your GPU program.

1. Running 1 work item or 32 work items takes the same amount of time, therefore create work groups which are multiples of 32 or 64 for AMD GPUs.
2. 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 work items 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 work item of the warp. The warp will execute the maximal number of iterations. So, if one work item has to do 1000 iterations and all other threads only a few iterations, 31 over 32 cycles are lost. The efficiency is then only 3%!
3. Some **memory access patterns** cannot be served at the full bandwidth. This will be discussed in lesson 5. 'Nice', contiguous memory access is fine, but other patterns will lead to longer latencies. To understand this, the notion of a warp is essential.

Concluding, the GPU will schedule the work groups among the GPU cores (**work group scheduler**). The number of work groups that can run together is bounded. This amount is scheduled on each core and each time a work group finishes; another work group is launched. Until all work groups of the workspace have been executed. Within a core, all warps of the concurrent work groups are active at the same time. The **warp scheduler** will choose among those warps which instructions are issued.

3 Measuring the lambdas of the pipeline model

The computational or memory performance is measured in GOps or GBs. More interesting, however, is the **Cycles Per Warp Instruction** (**$CPWI_{core}$**) because it reflects what's happening within the GPU: a core schedules an instruction of an active warp. How many cycles does it take? In the best case (maximal throughput) it is the issue latency, in the worst case it is the completion latency.

Calculating **$CPWI_{core}$**:

#Warp Instructions = #instructions/# cores/warp size **$CPWI_{core}$** = #cycles / #Warp Instructions = #cycles/#instructions * #cores * warp size

#cycles is the total number of cycles of executing the kernel (run time * clock frequency).

#instructions is the total number of instructions of the kernel.