High Performance Computing with GPUs
Exercise 6

Yunmin Cho Mario Alfredo Carrillo Arevalo

Technical University of Kaiserslautern
January 10, 2022

1. Why do we use GPUs for computing?

   We can compute more operations per second (higher performance) by consuming less electrical energy because GPU supports parallel computing.

2. How do we compile code for a GPU?

   We need to include our cuda code in `*.cu` files and then we can use **nvcc** NIVIDA compiler. E.g. `ncc main.cu -o out`

3. What will the following code look like on a GPU (just kernel and its call)?

```
for (i=0;i<N;i++)
    y[i]=a*x[i]+y[i];
```

```
__global__ void kernel (int N, float a, float *x, float *y) {
    int i=threadIdx.x;
    y[i]= a * x[i] + y[i];
}
kernel <<< 1, N >>>(N, a, dev_x, dev_y);
```

4. How to choose the index for each thread - and why?

   We can set a determined number of threads/blocks for our algorithm. We can use these threads through an index by using the pre-defined structure **threadIdx**. However, if we set a number of threads greater than the size of our problem **N** we need to add a condition `if(i<N)` to avoid overflow.

5. How are threads organized - groups/blocks, ...?

   The threads in GPUs are organized hierarchically with multi dimensions (`threadsIdx. (x,y,z)`). A group of threads is a **Cuda: block**, and it is a three dimensional set of threads (container), (`blockIdx. (x,y,z)`). And a three dimensional set of groups of threads (blocks) form a grid.

6. If we look at the simple loop in (3) - can we predict its overall performance?

   No, we need to check our GPU capabilities.

7. What is scalability?

It is the behavior of run time T when increasing problem sizes and the number of processors accordingly. There are two types of scalability: Amdahl's Law(strong scaling), Gustafsons' Low(weak scaling)

8. What about the time to transport data from CPU to GPU?

We can measure that time by using a basic formula.

$$T_c = T_s + size/Bandwidth$$

Where T_s is the startup time.

9. What is a SM compared to a CPUs hardware?

In a GPU, each streaming multiprocessor (SM) supervises a certain fixed number of ALUs in parallel. By the other hand, in a CPU all components are in the same unit.

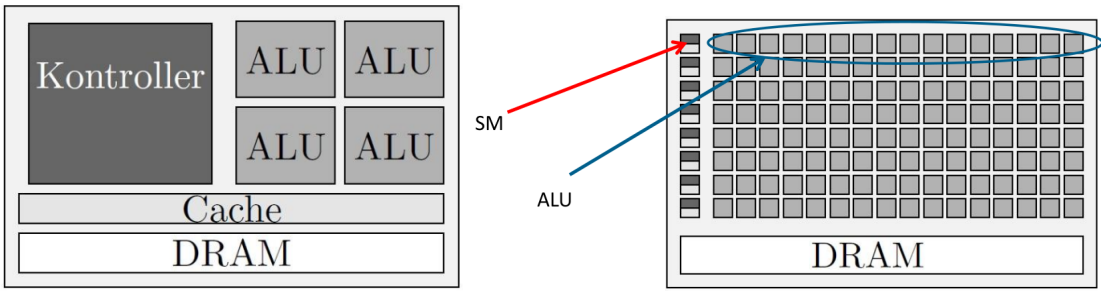


Figure 1: The left side illustrates CPU architecture and the right side GPU architecture.

10. Which types of memory exist in a GPU?

A GPU includes several types of memory such as: shared memory, cache, global memory etc. Each one has different sizes and limitations. The figure 2 illustrates its organization.

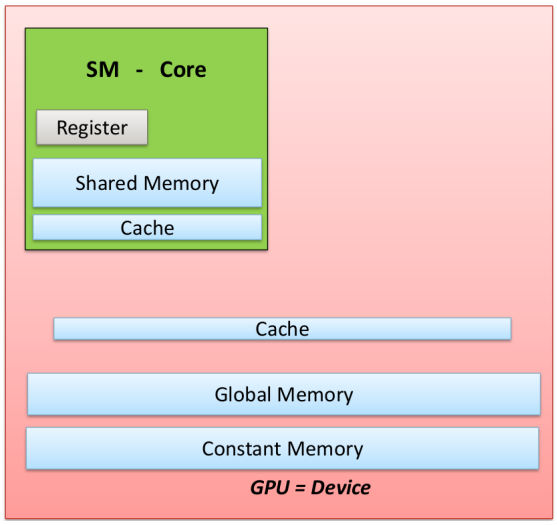


Figure 2: GPU - Memory organization.

11. How many registers has a GPU - and why are that many needed?

It depends on GPU, we need check hardware capabilities. According to the number of registers we can choose the number of threads per group as large as possible. this is to expose as many threads as possible. Therefore, we can increase parallelism. Registers are the fastest memory and each register can be accessed by a single thread. Therefore locally declared variables are kept in registers as many as possible. If we need more than available registers, locally declared variables are kept in a cache, and even further, in a global memory.

12. What is SIMD - what SIMT?

- **SIMD:** Single Instruction Multiple Data
- **SIMT:** Single Instruction Multiple Threads

13. How are threads organized and executed on a GPU?

A thread operates on each core. These threads are grouped to warps (32 threads).

14. What is an occupancy?

It means how many threads are running related to the maximal number of threads that we could run. It is defined as a *fraction (scheduled threads)/(maximal number of threads)*

15. How does the occupancy influence the performance?

If we increase the occupancy in our card we can introduce as much parallelism as possible, therefore, as we are executing independent operations we can reduce the execution time. In other words, the more threads we have, the better hiding memory latency is.

16. What are characteristics of an execution pipeline (ALU)?

1) **Latency:** How many steps we have to finish a single instruction.
2) **Throughput:** How much do I get in 1 cycle, if ALU is kepth busy.

17. What is Little's law?

It describes how much parallelism is required per GPU.

18. What is the relation between the pipeline latency and ILP?

For a memory request the latency is around 400-600 cycles. So, this prevents high performance. However we can hide this memory latency by exposing as many independent statements per thread as possible (Instruction-level parallelism (ILP)).

19. What is the relation between memory latency, Little's law and ILP?

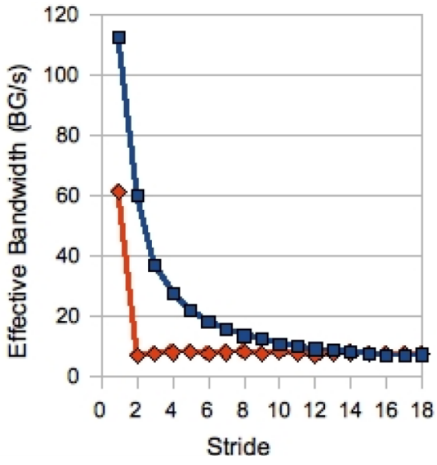
Parallelism is calculated (Little's law) multiplying latency and throughput (=number of ALUs).

20. How large is the memory latency?

It is around 400-600 cycles

21. How does the memory access pattern influence the effective (=measurable) memory bandwidth?

Non-contiguous access (stride greater than 1) reduces the bandwidth by a factor of 10.



22. What has the effective memory bandwidth to do with the index we choose for threads in (3)?

Our loop looks like `for (i=0;i<N;i++)`. So we are going to have maximal effective bandwidth due to our stride is 1.

23. What is a memory alignment?

In a GPU, the memory is accessed in groups of 32, 64 bits, and 128 bits. Howecer, if the size of the data which we are moving around is not a multiple of one of these values, then that array is going to be padded with an appropriate number of null values to round it up to one of those multiples. We want memory accesses as less as possible due to more they reduce the number of moving and copying instructions that can occur at once (the throughput). Therefore, when array pointers are not aligned, memory accesses could be slower, or it may even produce the wrong results.

24. What is a tensor core - how does it operate?

A tensor core is a hardware unit that allows to perform matrix multiplication. It produces 16 results every cycle (= 4x4 matrix) which follows the form described in the figure 3.

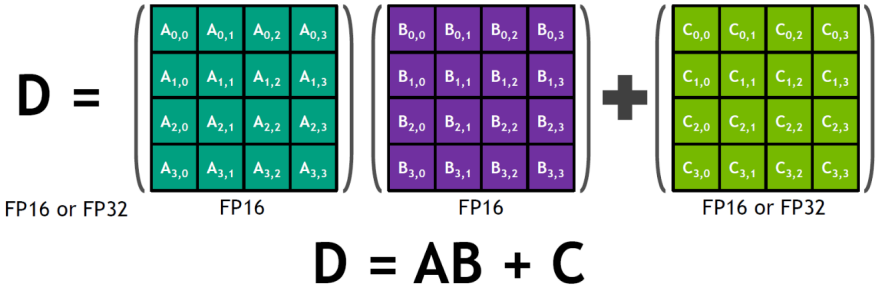


Figure 3: Tensor core - operation form.

The figure 4 describes how it works. The inputs have to be in half precision. However the multiplication of **A** and **B** work in full precision format, there is not intermediate rounding in this stage. This is a added to a floating point 32 precision matrix **C**. Finally, our result is stored in a 32 precision matrix as well.

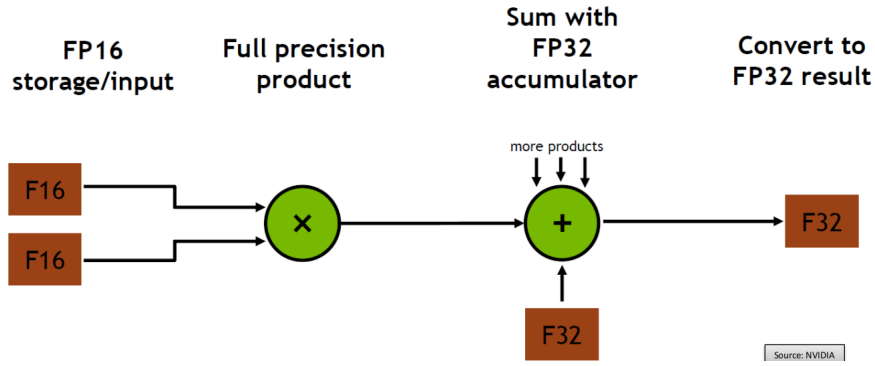


Figure 4: Tensor core - operation form.

25. What features provides the tensor core API?

The Warp Matrix Multiply Accumulate API, is a warp based API (not threads). It includes a new data type called fragments which adopt the role of registers. In addition, it contains just a few functionalities, however, they are enough to compute matrix multiplication.

- **fill_fragment:** initialize an accumulator
- **load_matrix_sync:** load input data
- **mma_sync:** perform the multiplication
- **store_matrix_sync:** store result

However it has some limitations that may vary for each Compute capability (CC)

- input A may be 8×16, 16×16 or 32×16
- input B may be 16×8, 16×16 or 16×32
- result D may be 8×32, 16×16 or 32×8
- and - like cublas - it's FORTRAN data layout
- double with $8 \times 8 \times 4$ only

26. How to use the tensor core API for a matrix multiplication?

As question (25) indicates, it is a warp based API. So, 32 Threads (= warp) work on one MMA. Each thread that cooperates in a warp-wide WMMA operation holds a part of each matrix in its registers (fragment). Now, we can summarize how to use it in 3 steps.

(a) Load the input matrices A, B, and C from memory into WMMA fragments using a **load** functionality.

(b) Perform the matrix multiply-accumulate using a **mma** functionality. The result is a fragment of the D matrix.

(c) Store the resultant D fragment to memory using a the **store** functionality.

27. What is half precision?

It is a binary floating-point computer number format that occupies 16 bits. It has some particular characteristics:

- The maximum absolut value that can be represented $\pm 65,504$.
- The smallest absolut value that can be represtend 2^{-10}.
- They have non-uniform precision loss

28. How should half precision numbers be accumulated?

No rounding during accumulation and **FP16** products are accumulated in **FP32** precision.

29. Which scaling should be used for range problems in half precision?

As was mentioned in (27), maximum absolut value for FP16 is 65504. So, for working with higher values we need to scale them. the scaling should be with powers of 2 to avoid over ot underflow.