Workshop Exercises Summaries

By Shin Kee Chung

CAASTRO (UWA)

Part 1 Summaries

- Part 1 is fairly simple, the main difficulty is grasping the concept of parallel programming.
  - It is different to conventional C programs, you are writing programs that will get executed many times (extremely similar to MPI).
- It might also be confusing that you have the graphics card series (GTX600), CUDA toolkit series (4.2) and compute capability (2.0), which are all different.

index $= bx * nby * ntx * nty + by * ntx * nty + tx * ntx + t$

index $= bx * nby * ntx * nty + by * ntx * nty + tx * ntx + ty$

1.4 Vector Addition

- By referring to the example, it makes it quite easy to complete.
- The optional section is just a little bit tricky, you should consider how the index should increase when thread indices increase, then proceed to consider block indices.
- The final indexing looks like this:
  - *index = bx*nby*ntx*nty + by*ntx*nty +tx*ntx + ty*
  - *3D:*
  - *index = bx*nby*nbz*ntx*nty*ntz + by*nbz*ntx*nty*ntz + bz*ntx*nty*ntz + tx*nty*ntz + ty*ntz + tz*

1.5 Timings

- Concept of asynchronous.
- For vector addition, it is virtually impossible for GPU to beat CPU in terms of speed at this stage. So don’t get upset for it.

1.6 2D Arrays

- Fairly straight forward, just compile, execute and observe.

1.7 Matrix Multiplication

- Matrix multiplication is one of the simplest (yet no so simple) algorithm that can be tested in GPU.
- When you start attempting it, you will most probably find that it gets complicated really quickly, especially when it is non-square matrices. This is just a “simple” non-optimized version.
  - This is part of the challenge in CUDA programming.

2.1 CUBLAS

- This part is also fairly straight forward, we only tested out some CUBLAS functions, and look at how it gets executed.
- If you did the matrix multiplication in the previous part, you will most probably find that your implementation is much much slower than CUBLAS.
- This is normal for unoptimized code.

2.2 CUFFT

- Firstly, without changing the size of the FFT operation, almost certainly CUFFT is slower.
  - Only when you get to reasonably large size, like 2^{10} then you will have some speed up.
- The batched version though, should be able to give a much better speed-up. The speed-up should reach a plateau with certain batch size.

2.3 Atomic Functions

- When multiple threads are performing some operations into the same memory space, it is almost certain that they won't be able to cooperate well. We need atomic functions.
- But atomic functions are just going to lock the memory space when certain thread access it. Making it a sequential operation which is very slow, not really what you want CUDA programs to be doing.

3.1 CUDA Profiler

- CUDA profiler is not just very useful in identifying bottlenecks, but also useful in detecting problems.
- If CUDA profiler does not output any time, or the time is not what you expect, you can be sure that there are some problems in your programs.

3.2 Threads, Blocks and Grid

- You may find that having any number from 1 to 64 threads makes no difference to the kernel execution time, then any number from 65 to 96, and 97 to 128, all incremented by warp size, 32.
- This means that threads are best chosen to be multiple of warp size, so that you don't waste computational resources.
- In general, there is a range of optimal numbers you can choose. You should not choose too many or too few.
  - The key is whether your threads and blocks can keep the GPU busy (high occupancy).

3.3 Batch Implementation

- This part actually discusses two issues.
  - Batch concept
  - Not all programs are suitable for GPUs
- Vector addition, is just a bit too simple, while the memory transfer is too much in utilizing GPUs.
- Therefore, if you have an algorithm that just consists of a number vector additions (or operations that are simple but require high memory transfer), you probably shouldn't think about optimization with GPUs.

3.4 Aligned Memory Access

- In terms of memory, we usually choose multiples of 256 bytes to avoid wasting the spaces (as CUDA always align your memory to blocks of 256 bytes as mentioned in the documentation).
- Strided access is the one that you should be aware of, as it cost at least double your memory bandwidth for the same number of data.

3.5 2D Memory

- The pitch allocation guarantees alignment requirement across different rows.
- Effectively, the pitch is your real width of your allocated memory. So your should use pitch bytes when going into next row.
- The indexing looks like this:

```
i = bx * pitch / sizeof(float) * tx;
```

3.6 Shared Memory

- Shared memory is not a magical feature that will solve everything. It still needs to read from global memory at least once.
  - Not useful for algorithms that doesn't read the memory repeatedly
- Again, it is hard to achieve great speed-up as the memory readings are only repeated 8 times, or a few times.
  - Think about matrix multiplication with size 1000 or more, which should be very worthwhile to implement this.

Part 4 Summaries

- The techniques explained in this workshop are not exhaustive, beneficial to stay up-to-date.
- There are quite a bit of optimization techniques explain in CUDA C Best Practices Guides, we have only covered a few that might not be easy to understand.

4.1 CUDA Streams

- Depending on your programs, you might want to overlap kernel execution and memory copy, or overlap two memory copies.
- While concurrent kernel can provide a bit of speed-up, you should always consider having a larger kernel execution which is usually faster.

4.2 Bank Conflict

- Bank conflict can happen when:
  - Multiple threads in the same warp accessing the same memory address.
  - You access too many successive memory space that you run out of banks.
- There is need to carefully consider whether it is really essentially to optimize for avoiding bank conflict.

4.3 Avoid Warp Divergence

- When replacing if-cases with maths operation, it might affect the overall execution speed when the warp doesn't perform diverged operations that often.
- Again, it is hard to predict in advance which methods are faster, it will come down to testing most of the time.
- Answer for the example warp divergence:

```
y = ( 1 + floor(x + 0.5) ) * x;
```