Report 5

Introduction to CUDA and OpenCL

Gaining performance in many possible ways.

Dariusz Biela

Karol Sawicki

1. Introduction

At the time of our sixth laboratory classes we were working with three codes, our task was to gain as much performance as possible.

The first was the vector addition code, that we had to speed up by adding more streams to the GPU execution.

The second was the SAXPY problem – Single Precision $a * X + Y$ operation, where a is a floating point number, X and Y – are vectors of floating points. We were encouraged to prepare code that executes kernel in 25 μs or less.

The last task was to prepare GPU accelerated code in the other way, by using Open ACC to speed up single-threaded code.

2. Calculations and measurements.

2.1. SAXPY

We decided to start with the SAXPY code. We decided to prepare new, more flexible kernel with stride grid loop.

To execute our kernel with the most efficient way we used cudaGetDeviceAttribute function to know how many Streaming Multiprocessors our GPU has. We launched kernel with 32 * SM as a number of blocks in grid and prepared loop to measure how many threads will be the best for.

Table 1 Execution time for diffrent thread counts

Threads in block	Kernel execution time [μs]
1	20808
2	1907
4	1002
8	617
16	411

32	284
64	244
128	250
256	235
512	235
1024	241

In the table we decided to show measurements prepared with C++ std::chrono library. The lowest time was observed for 256 threads in each block. To better data presentation we prepared column graph shown below.

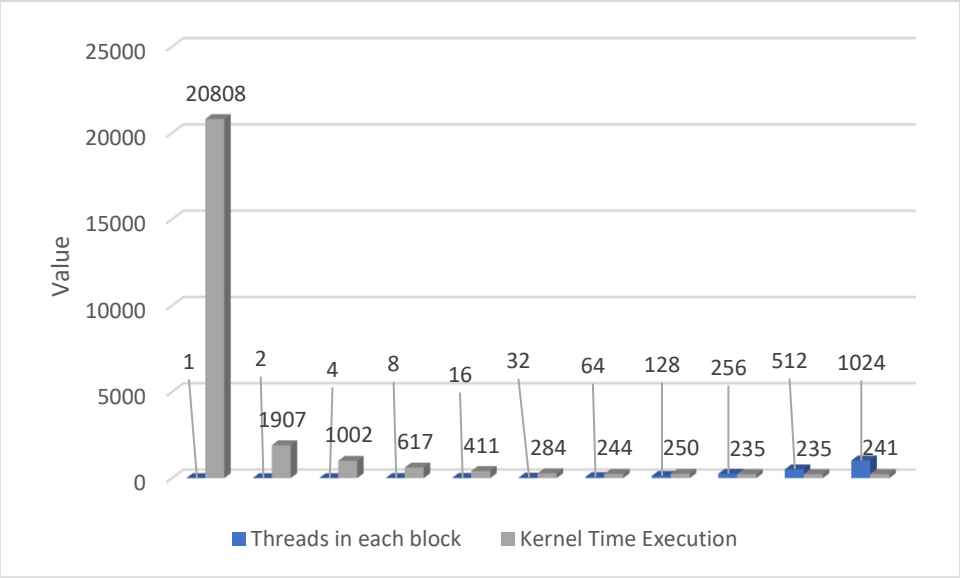

Figure 1 Ilustration of data from Table 1.

To be sure that our measurements were correct we decided to use nVidia nvprof tool to profile our code. Surprisingly the result for 256 threads was better – we obtained 166 µs. We run nvprof couple of times and the result was pretty much the same.

The original nvprof result for SAXPY kernel was somewhere around 15/16 ms. Our gain was lower than required, but the result was still much better than original.

```
==17536== Profiling application: ./saxpy.exe
==17536== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  15.375ms         1  15.375ms  15.375ms  15.375ms  saxpy(float*, float*, float*)
```

Picture 1 Example of the nvprof result screen.

Just to try, we decided to do some experiments with original kernel. Simply modification with Processing Grid gave us so much better performance, even better with modified stride style kernel with use of SMs knowledge.

```
==2448== Profiling application: ./saxpy_second.exe
==2448== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  18.763ms        11  1.7057ms  6.8480us  18.688ms  saxpy(float*, float*, float*)
```

Picture 2 nvprof result for 10 kernel runs with modified PG and the original one.

At the *Picture 2* we could see, that first run takes nearly 100% GPU execution time. The modified ones, in the other hand, are extremely fast. The used PG was the one with 128 blocks and 512 threads in each block.

2.2. GPU Streams.

After that we worked around with GPU streams and try to parallelize kernel executions.

To start working with streams we tried to profile existing code.

```
==15571== Profiling application: ./vadd.exe
==15571== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.85%  1.8409ms         3  613.63us  547.53us  655.85us  initWith(float, float*, int)
                   43.15%  1.3975ms         1  1.3975ms  1.3975ms  1.3975ms  addVectorsInto(float*, float*, float*, int)
```

Picture 3 Example of the nvprof result for adding two vectors without streams.

Picture 3 shows that in the code existed 3 calls for kernels initializing vectors, each one for other vector. Vectors were added by single kernel execution, that took somewhere around 1.4 ms.

After checking that results we tried to add more streams not only to initializing data, but to the main vector adding kernel execution too.

```
==16231== Profiling application: ./vadd_temp.exe
==16231== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.03%  1.8835ms         3  627.83us  543.59us  680.84us  initWith(float, float*, int)
                   44.97%  1.5394ms         3  513.13us  487.50us  540.04us  addVectorsInto(float*, float*, float*, int)
```

Picture 4 Example of the nvprof result for adding two vectors with 3 streams.

Picture 5 tells us, that the time was slightly worse. We were using each stream to add 1/3 of vectors. For further investigation we tried to add vectors into three separate ones. In that case our results was pretty much the same – adding streams made our execution timings worse.

At the end we decided to prepare more complex code for adding many vectors in many kernel calls. We tested adding 32 vectors with streams and without them, to compare we added operation of adding two big vectors.

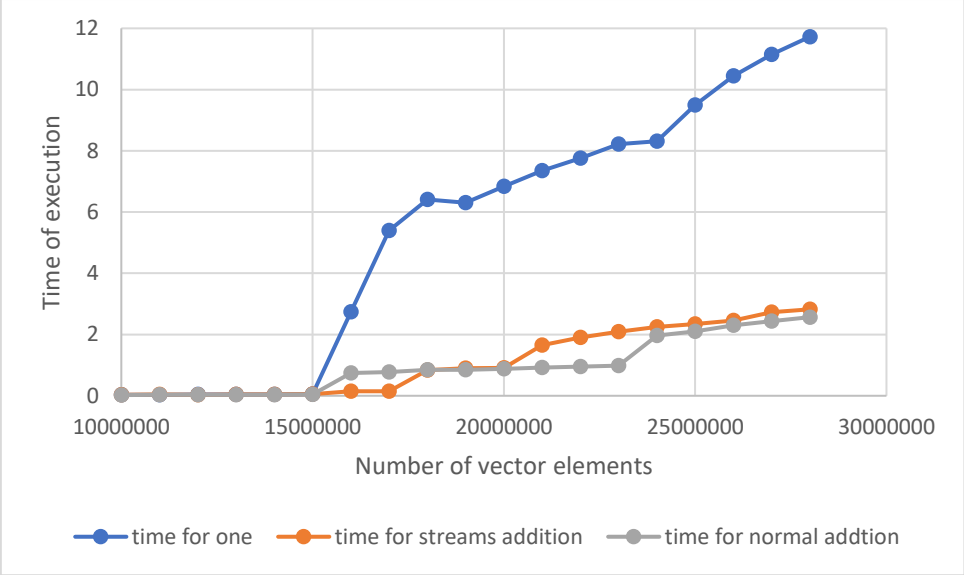

Figure 2 Measurement illustration, the size of blue one should be multiplied by 32..

Kernel parallelization must rely on many aspects, one of them must be the data size. Layout of the grid can be meaningful too. We should know that the maximum streams count is 32, but we could not be able to use all of them at the same time.

Furthermore – streams are not required to speed up our vector addition, what we could see when comparing big vector addition with adding 32 times smaller ones.

2.3. OpenACC acceleration.

At the end we decided to work with heat simulation code. We had prepared code, that only needed to be accelerated with OpenACC #pragma compiler instructions. This instructions are really simple, they not interfering with code, the compiler recognizes them and tried to prepare GPU kernel that will be automatically executed on the GPU.

With that knowledge we prepared some code using #pragma acc kernels for kernel generation to parallelize independent loops, #pragma data copyin and copyout to avoid many data movements that would not be necessary.

To compile that code we needed special compiler, that could recognize OpenACC specific instructions.

3. Conclusions.

The stride kernel is not the best way in every possible scenario. We could do some assumptions before start writing code, but we should always prepare more than one possible implementation and, at the end, compare results, just to determine, which way is the best for problem, that should be solved. In some cases the simplest optimizations could be the best ones.

Simple adding streams without thinking is not efficient. If GPU is busy, kernels are well matched by processing grid to the GPU, there could be not space for more parallelization. The newer NVIDIA GPUs could work with up to 32 streams, but there is no guarantee that they would be working with 32 streams in every case.

OpenACC is good for parallelizing complicated algorithms, because source code stays the same, what means there is no possible mistakes in implementation. To use OpenACC we should have compiler that supports it, but if we do not have one, there is no need to deleting #pragma OpenACC instructions, because compiler simply ignores them. When it could it speeds up execution, but if not, there is no way to harm code by using OpenACC.