Introduction to CUDA and OpenCL

Ilona Tomkowicz, Zofia Pieńkowska

October 29, 2019

Contents

1 Data structure limits

By changing the variable *numElements* we check what are the limits of data processing for a fixed configuration of kernel.

1.1 How large data are handled successfully.

The biggest data structure that could be used in sample vector add project was 2^{27}. Console output:

```
$ ./executable
[Vector addition of 268435456 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 262144 blocks of 1024 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

1.2 What is going on in this experiment.

By increasing the variable *numElements* we increase the size we want to allocate for our variables. When memory allocation on the host side does not pose any problem, the memory that we can use on the device side is limited. We can check the device memory by running the code:

```
size_t available, total;
cudaMemGetInfo(&available, &total);
fprintf(stderr,"%d,%d,%d,%d", available,total);
```

output:

```
$ 2002059264, 2076508160
```

Taking into consideration that we attempted to allocate 3 identical vectors having 2,076,508,160 bytes (equal to 500,514,816 float numbers) available on the device, by simple devision we can see that one of such vectors can have at most 166,838,272 float elements.

1.3 When does the code give errors?

The error occures when we exceed the limit of floats calculated above. For $numElements = 2^{27} = 134{,}217{,}728$ that requirement is still fulfilled. However, for $numElements = 2^{28} = 268{,}435{,}456$ not anymore.

2 Optimal grid layout search

This part of labolatories included timing experiments for various grid layout with fixed elements number ($numElements = 33{,}554{,}432$).

Each layout below was organised in threads in such a way, that number of threads was a multipier of 32 to ebanle warps be used most efficiently. Warp is a set of threads that share the same code and execution path. Merging these threads into a set (on the level of poduction) makes calcultaions faster, so that we should not split wraps into parts.

Another constrain in grid layout is the maximum thread count in one block, which is 512 for Compute Capability 1.x and 1024 for 2.x and later - our case is the latter one, 6.1 (check the report_lab2 for device details). Moreover, each block cannot consume more than 32k register memory in total.

Optimal performance for a specific device with specific program has to be found empirically. CUDA Occupancy Calculator might be helpful in this matter.

2.1 Layout experiments

Table 1: Host and device memories cooperation:

Threads	**Blocks**	**DtoH[ms]**	**HtoD[ms]**	**vecAdd[ms]**	**vecAdd[%]**	**sum [ms]**
32	1	85.4	69.2	3.3	0	157.9
32	1,048,576	60.7	42.6	6.3	5.7	109.6
128	262,144	83.7	61.6	5.3	3.5	150.6
256	131,072	71.1	55.4	5.4	4.1	131.9
512	65,536	71.3	55.9	5.2	3.9	132.4
1024	32,768	80.9	59.0	6.4	4.4	146.3

Table 2: Managed memory:

Threads	**Blocks**	**vecAdd[ms]**
128	262,144	72.3

2.2 Conclusions

For separate memories, the fastest grid version is the one with 32 threads and 1,048,576 blocks. It has such a low overall timing due to the gain in data copying. It can be seen, that in terms of vecAdd function execution that solution is the worst one, so generally for more complicated functions another layout woule be prefered.

Times for computaton do not differ much between presented layouts and do not have significant impact on the overall performance. Although in our case the most effective approach is the 2nd row (due to few calculations) the recommended choice is to form grid with 128 - 512 threads and then condcting experiments with blocks size. If we had to choose a layout for more complex function then probably the grid with 128 threads and 262,144 blocks would be the best one, providing that copying data would not make that solution gainless.

However, in our case when we have so litte operations and memory usage the best way is to use the Unified Memory. This is a single memory address space accessible from any processor in a system. Its performance is not as fast as regular GPU memory, but the gain from the lack of data copying compensates it. As a result the operation is performed in less than half of the time needed for the same grid layout with data transfer.