Image Processing 2013 *2016-12-12*

Team 8

Denisov Yury (優里) 0560823

Pavel Lutskov (帕維爾) 0545107

FINAL PROJECT PT1 REPORT

<u>Final result: GPU time for executing a typical convolution layer = 10 ms</u>

Contents

1. Implementation algorithm

The algorithm used to solve the task is simple.

The task of computing the convolutions is parallelized to FILTNUM (512) number of blocks, where FILTNUM is the number of filters in the input data. Each block is parallelized to FMSIZE*FMSIZE number of threads (512), where FMSIZE is the size of input data layer.

Each thread of each block computes a sum of convolutions from 1 to FMDEPTH (FMDEPTH is a filter depth, 512 in this case) for one pixel only, starting from the "plane #0" and moving further "deep".

After that, the Activation ReLU is used in the same kernel (simply by setting the result sum 0 if it is less than zero).

The last part is a 2x2 max pooling, which is also parallelized to FMDEPTH blocks with FMSIZE/2 * FMSIZE/2 threads in each block and is done by a different follow-up kernel, and done similarly to CPU realization, but in parallel instead.

The trick here is that the filter is represented and stored in the block shared memory as a space array, since it can contain a lot zeroes. This, together with another tricks, led to a dramatic increase in performance (more than 4 times immediately), and will be discussed in the section 3 of this report. Instead of storing the filter as FILTNUM*FILTNUM size array, it is stored as three arrays, each size of [TILING_FACTOR][FILTNUM*FILTNUM]. First array (ftx0) stores the x coordinate of tilter value, second (fty0) stores y coordinate of y value, and third (val0) stores the actual value. There's also an array cnt0 in size of [TILING_FACTOR] which stores the number of non-zero elements for each layer of the filter. This reduces a lot of redundant computation, like array indexes and counters.

The maximum number of threads for one block on GTX680 is 1024, but for some reason it failed to launch the kernel with 32*32*2 grid (1024 threads), so it was left as it is.

Since now the computation now is heavily parallelized (compared to CPU solution, which was consecutive), it is no surprise that the result is being computed much faster.

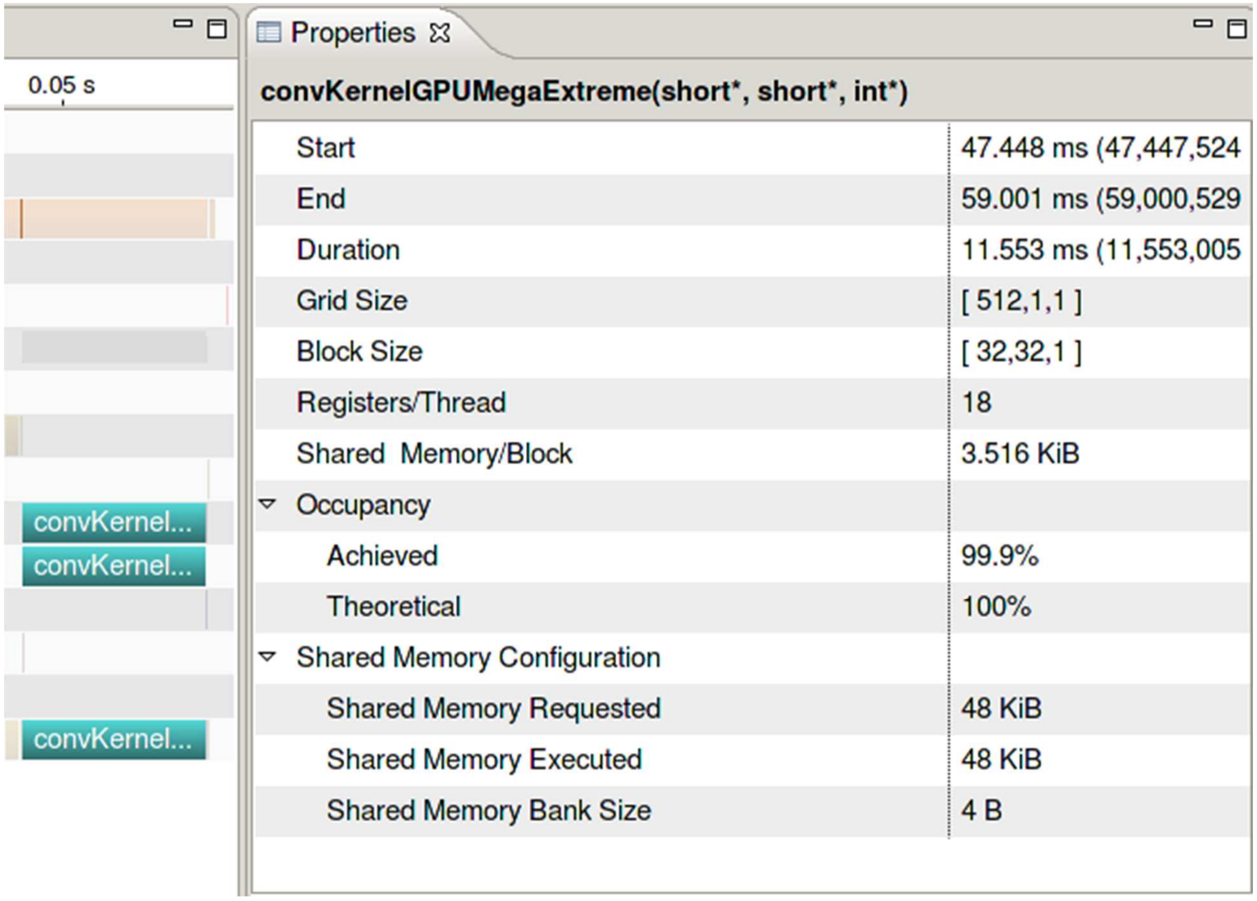
The GPU has a lot of parallel processing power at hand, and the goal of representing the computation process like this was to use it as efficient, as possible. Since the convolution computation input neurons are easily parallelizable, the task was a success.

Currently, the typical convolution layer is computed in about 10 ms, instead of 16.5 seconds in CPU, which gives a computational advantage about 1600.

The interesting fact is, that the CUDA initialization (just the initialization, without any memory allocation and copying) takes much more time than the actual computation (about 300 or so ms), but since it is still much faster than CPU, and perhaps can be used a lot, a lot of times during the neural network learning process, it is of no concern.

2. NVVP profiler usage

NVVP profiler was extremely helpful in this task. In the first realization, instead on FILTNUM blocks and FMSIZE*FMSIZE threads, the FILTNUM*FMSIZE blocks and FMSIZE threads were used. This led to occupancy of about 25% only. NCCP helped to solve this issue, and correct the blocks/grid structure so the theoretical occupancy became 100% and practical became about 99.9%. Surprisingly, the improved occupancy doesn't always mean better performance, but in this task improving the occupancy gave a dramatic increase of performance (from about 20 advantage to CPU to 90 in the early beginning of the development).

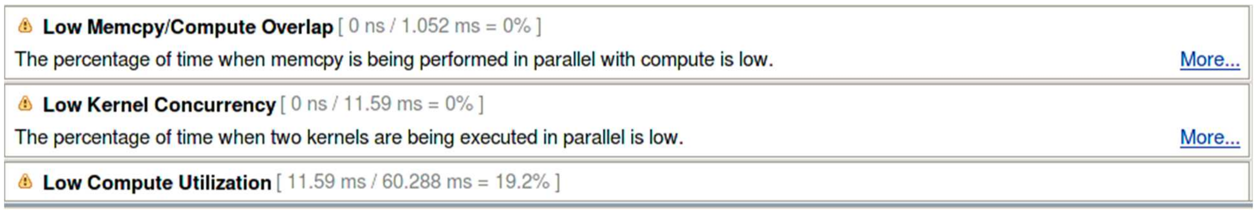


convKernelGPUMegaExtreme(short*, short*, int*)

Start	47.448 ms (47,447,524
End	59.001 ms (59,000,529
Duration	11.553 ms (11,553,005
Grid Size	[512,1,1]
Block Size	[32,32,1]
Registers/Thread	18
Shared Memory/Block	3.516 KiB
Occupancy	
Achieved	99.9%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Image 1: Stats of the convKernelGPUMegaExtreme, the main kernel.

The image 1 shows the stats of the kernel responsible for convolution and activation task, with theoretical and achieved occupancy, grid and block size and shared memory usage.



Low Memcpy/Compute Overlap [0 ns / 1.052 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

Low Kernel Concurrency [0 ns / 11.59 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Compute Utilization [11.59 ms / 60.288 ms = 19.2%]

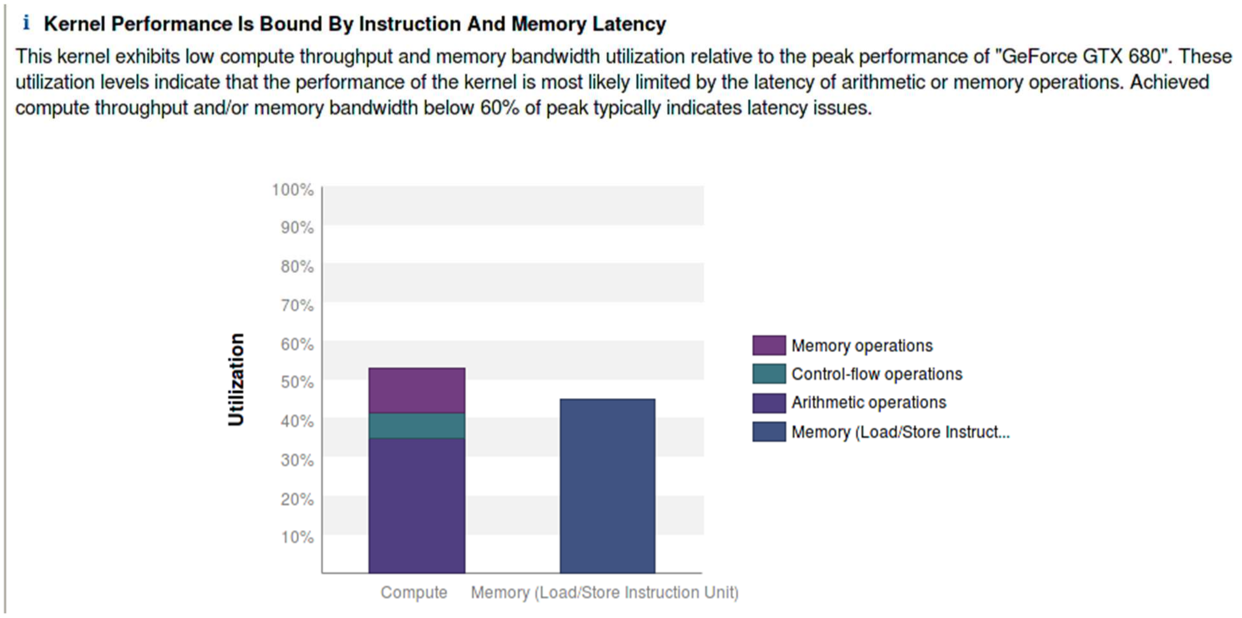
Image 2: NVVP warnings.

The NVVP also issued three warnings (shown on image 2): LowMem/Compute overlap, low kernel concurrency and low compute utilization. All three warnings can be ignored in this task for the following reasons:

Warning 1: Low MemCpy/Compute overlap. Since only the computation speed of the kernel will be measured in the contest, the data load time is not important, and all data is loaded prior to kernel launch. It is useless to optimize this section by loading and copying memory to host at the same time – the overall execution time of the program will be shorter, but the kernel running time will probably be the same or even larger. In real life this section should be optimized without a doubt, especially if there's a lot of data exchange between host and the GPU device.

Warning 2: Since we use two kernels one after another, this warning appeared. It is discussed in the part 3.1, why this warning is left untouched.

Warning 3: Low compute utilization: as was mentioned before, while data is loading, the GPU doesn't do any computation. And since the actual computation takes much less time, than CUDA initialization together with data loading, the profiler notices that for the most time of the overall task the GPU doesn't do any work. This is ignored, as, again, only the kernel performance will be measured.


Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 680". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Image 3: Kernel performance chart.

Next useful thing NVVP provides, is a deep analysis of the kernel. In the image 3, it is possible to see the actual percentage of different compute and memory operations, to understand if it is possible to improve something here. On the image 3 is an example of a kernel limited by compute or memory issues, found during the development process. The image shown hints, that the memory operations, perhaps, should be decreased in order to increase the overall performance. The good practice is to load a parts of data which is needed multiple times into the shared memory, which is much faster that regular one. The drawback is that shared memory is limited to only 48 KB, and it is usually needed to split the data to portions. But still, even is the total number of memory operations will increase because of this approach, since the memory is now much faster, the total time will decrease.

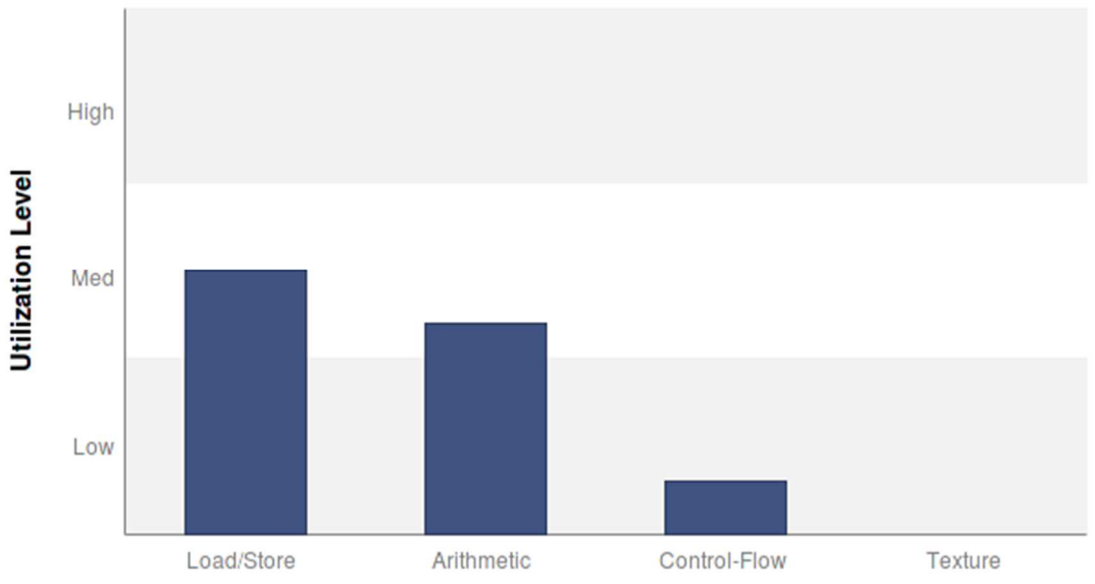


Image 4: Operations utilization level chart.

Operation utilization level is, perhaps, more useful in many case instead of percentage representation. Now it is obvious that memory operations are taking too much time, compared to other and should be decreased somehow. Using this charts, memory, arithmetic and even control-flow operations were optimized (by removing some redundant operations) during the development process.

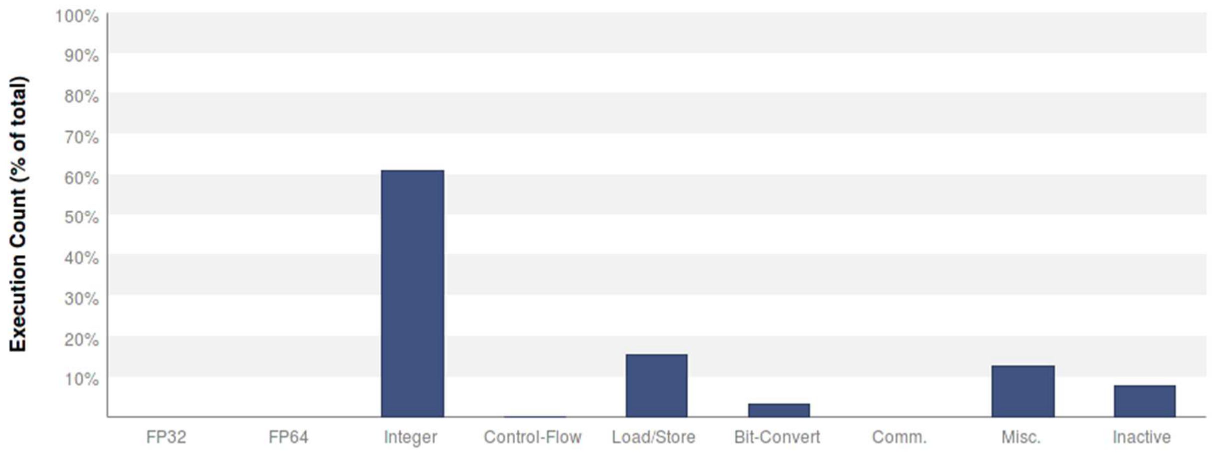
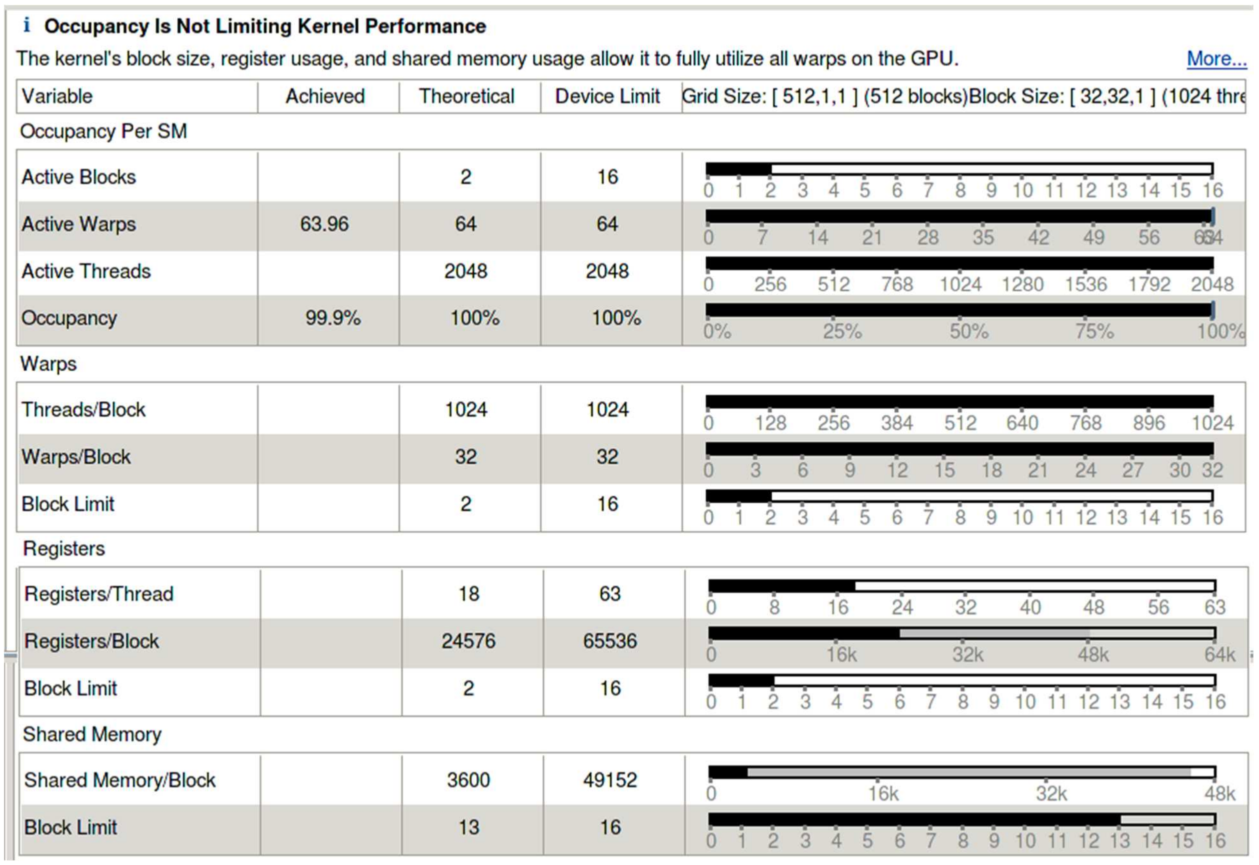


Image 5: Execution count chart.

Execution count is another useful chart. It can hint, if there is too much computation happening in the kernel. Perhaps, some data is being computed many times instead of one (like array indexes, especially when traversing multidimensional arrays).



Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [512,1,1] (512 blocks)Block Size: [32,32,1] (1024 thre
Occupancy Per SM				
Active Blocks		2	16	
Active Warps	63.96	64	64	
Active Threads		2048	2048	
Occupancy	99.9%	100%	100%	
Warps				
Threads/Block		1024	1024	
Warps/Block		32	32	
Block Limit		2	16	
Registers				
Registers/Thread		18	63	
Registers/Block		24576	65536	
Block Limit		2	16	
Shared Memory				
Shared Memory/Block		3600	49152	
Block Limit		13	16	

Image 6.1:Other useful data from deep analysis of the kernel.

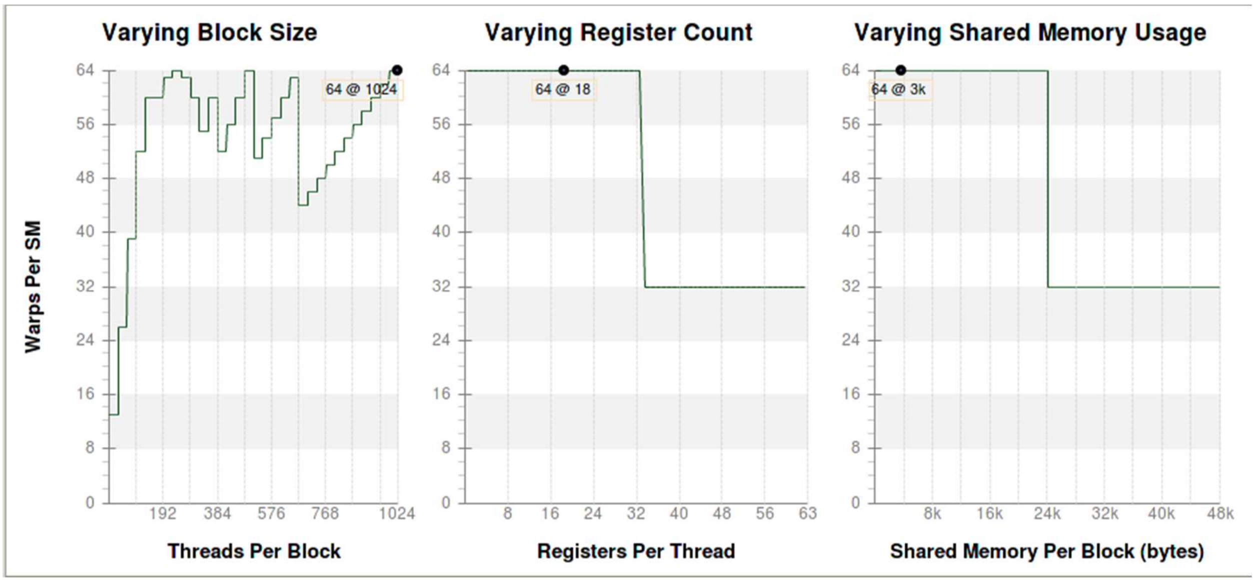


Image 6.2: Other useful data from deep analysis of the kernel.

NVVP also provides a lot of useful data to find some other issues, like shown on images 6.1 and 6.2. Using all this data, it was estimated that the good grid size should be a multiplier of 32, so the active warps will be closer to the theoretical values.

3. Optimization.

A lot of work was done in the optimization, using different techniques and tricks, all of which are listed and discussed in this section.

3.1 Two consecutive kernels instead of one.

This is the biggest part of the task, and it took a lot of time and a lot of different tricks and techniques to achieve the current result (10ms). All these tricks and techniques are listed and explained in this section.

The task is split into two kernels. First kernel, dubbed convKernelGPUMegaExtreme, does the convolution and activation. The second kernel, max2x2KernelGPU, does the Max 2x2 pooling. Initially, the task was done by one kernel, with Max 2x2 pooling embedded into it using atomic operations, but it led to a loss of precious 1-2 ms of computation time (possible due to memory conflicts, 4 atomic max operations were trying to access the same memory address at one time, ordered in line as a result and stalled the overall thread execution), so the task was split between two consecutive kernels.

The second kernel execution takes less than ms to execute (about 18 microseconds), and the first one takes much more, so the main task was to optimize the first one.

3.2 Using better architecture for the compiler.

The default settings of NVCC compiler use compute_10 architecture, in which main memory is not cached, but GTX680 supports the compute_30 instead, and the main memory in compute_30 is cached, so by using this architecture for compiling, a noticeable performance increase was noted. *–use_fast_math* key is also used, but it doesn't give any noticeable performance boosts.

3.3 Loop unrolling.

Each loop in the kernel is unrolled by a compiler (using *#pragma unroll).* This makes the resulting program quite large, and the compilation time quite noticeable, but it removes some redundant operations, like "for" loop index computations and checking of the condition after each loop. This gives a slight good boost to performance as well.

3.4 Removing all redundant operations from the kernel.

When the kernel is launched, the first thread computes some operands (mostly array index offsets) which are the same for the whole block, and stores them in the fast shared memory, and all other threads are using these values, instead of computing them anew.

3.5. Using the sparse array filter representation in the shared memory.

This is the main optimization technique, which gave a huge and enormous boost to the performance.

The filter is loaded and stored as a sparse array in the shared memory of the block. While moving through the layers of input data, first TILING_FACTOR threads load and store the filters as a sparse arrays to the shared memory, in the meantime all other threads are waiting for them. After that, each thread computes the sum of TILING_FACTOR convolutions, corresponding to different layers for one pixel this thread is responsible for. Then the process repeats, until all layers are traversed.

It is important, that (FMDEPTH % TILING_FACTOR == 0) right now, because the checking of "are we out of layers yet?" can be skipped in every cycle of computation, and can be done only after TILING_FACTOR of computations instead.

There was an attempt to load the layers into the shared memory as well, but it led to a decreased performance due to many load/store operations involved, so only the filter is loaded and stored in the shared memory. The TILING_FACTOR was found experimentally to be 128, if it is more, some delays begin to occur, and then if it is increased even more, the even filter data cannot be stored to shared memory due to its size limit.

By storing the filter as a sparse array, it is possible to dramatically reduce the total operations, since now all index calculations, and more important, load store operations will be performed for a pixel only if the corresponding filter value is not zero.

3.6 Attempt to use textures to speed up memory access.

Textures are still stored in the main memory, but are optimized and cached for localized spatial access. The attempt to use textures instead of main memory to store the data gave nothing, since the main memory has a cache too. 1D texture was used due to the ease of implementation, but, perhaps, 3D texture should be used instead. Since it is much more difficult to understand, how to use 3D texture correctly, this idea was abandoned and the brain power was directed on other techniques.

For now, these are all optimization techniques which were used in the task.

4. Feedback

The project was quite interesting, and gave a lot of new knowledge of parallel computing and computer architecture. The task was simple enough to be educational, but it still has a lot of tricky parts in it, which can be seen in the 3rd part of this report - at least 6 different optimization techniques were used together in an attempt to speed up the computation process. The project also gave some basic understanding on convolutional neural networks, which is a nice bonus.

Another funny issue was that we all got one server for about more than 10 teams, and a heavy computation task which usually uses all resources available on that server. So, if anyone is on the server running some heavy code, you need to wait. And if something goes wrong with that code – you need to wait much more. Last three days were the worst, with about 3-4 teams on a server at the same time.

The grading system is another issue. In our country, it is unheard of the "first one gets full grade, last one gets nothing" policy. Usually, the teacher gives the winner some prize (some money, something precious) and/or a maximum grade for his course together with exemption from final exams. This is still a good motivation, but it doesn't make you to hate all of your competitors.

We still really enjoyed this project, though, and want to say a big "Thank you" for all your job.

END OF DOCUMENT