Image Processing 5013 *2017-01-07*

Team 8 Denisov Yury (優里) 0560823

Pavel Lutskov (帕維爾) 0545107

FINAL PROJECT PT2

REPORT

Final result: GPU time for executing a typical convolution layer together with data transfer = 8 ms

Improvement over previous solution: 20%

Contents

1. Implementation algorithm

Since the filter scarcity was already used in the first part of the final project, there isn't much to change in the second part, and the algorithm was left unchanged. The algorithm will be presented again (copied from part 1) for information purposes.

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.

1.1 Sparse neuron and filter usage

The fact that now the filter is supplied in sparse format (COO) means that the code to transform the filter from dense format to sparse one was removed, which gave a little boost. The format of sparse filter was changed and is discussed in the "optimizations" part of this report.

Several attempts to use sparse neurons were implemented, but none of them were better than current solution, which does not use neuron sparse at all. The biggest problem was the structure of blocks and threads in the first part: it was designed that one block will process neuron layers from 0 to FMDEPTH, with a single thread for each pixel – total 512 threads for block. So one thread processes one column of the pixels. The sum is calculated using the register variable, the fastest, and only once per thread the sum is written to memory. The optimization of this scheme led to almost 100% occupancy and high efficiency of memory usage and warps. Changing this scheme to use sparse neurons is nearly impossible, the attempt to use one thread for one pixel led to unavailability to use register variable to store sum, and shared variable array was used to store sums instead. This led to a dramatic decrease of efficiency. Just quickly traversing through a "column" of pixels is impossible. So the input neurons were used again with dense format.

1.2 Data transfer time and efficient sparse format

Data transfer time with a scheme used in first part of the project takes about 0.5 ms. It seems, it is not worthy to spend any time on this part to obtain some noticeable performance boost. Still, the attempt to transfer the compressed sparse neurons and uncompressed them at the GPU was used. The sparse format which was used was changed to CSC – it removes the redundancy of the same column or row indexes in COO format. Still, the data spent on decompressing the data led to a performance decrease, and this approach was not implemented in the final version of the project, but it may be used for much larger neural networks – for the network this small as it was in this project, it makes no sense.

Overall, the time to execute the convolution kernel together with data transfer from host and back takes about 8 ms now, which is lesser than 10 ms kernel only in the first part. While this decrease is small, it still is a 20% performance improvement. The kernel itself takes about 7.5ms to execute now.

2. NVVP profiler usage

The NVVP was used in this part of the project as well, but not as vastly as in the part one since all possible optimizations ware already achieved there, and the algorithm is basically left unchanged. However, there were two use cases which helped to improve the overall performance:

1. Idea to remove control-flow operations from the convolution kernel (discussed in "optimizations" part of this report) was obtained by using NVVP.
2. A little optimization of data transfer time was also implemented – the type of some data transferred was changed from *int* to *short*.

As was mentioned in part 1 of this report, an attempt to compress the neurons, transfer them to GPU and uncompressed there was also implemented, and was tracked via NVVP, which showed its uselessness in case of such a small neural network.

Since the occupancy is near 100%, and all warps are almost fully utilized with no severe memory issues were tracked, NVVP wasn't that helpful in this part, as it was in previous one. For usage practices of NVVP, please look the previous report, since there's nothing new to show here.

3. Optimization.

Since the algorithm is basically unchanged compared to part one (which also used filter sparsity), all of the methods from previous report are still being used. They are re-listed here for information purposes.

3.1 Two consecutive kernels instead of one.

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.

New techniques are listed starting from here:

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, and it was used in the previous part of the project as well. This alone led out team to victory that time, and is the main optimization technique.

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 New sparse filter array format

The main difference between the previous part and this one is the fact that sparse array is recalculated. More than that, all of the filters have only one non-zero value, which make is a special case. Usually, for a COO format you need to know how many non-zero values you have, and you'll have three arrays of data, cols and rows. In this case, since there's only one non-zero value for each filter, this format was replaced with a much simpler one: just three arrays of data, cols and rows, with indexes of these arrays corresponding to filter number, accordingly. This makes the solution quite specialized, since now it won't work with a case where a filter sparsity is different from "only one non-zero left", but this dramatically reduced the amount of data needed to transfer and many index computation operations as well, which led to about 2ms kernel performance boost (out of 3.5 total).

3.7 Padding the neurons

The last optimization technique, which gave another 1.5ms of kernel performance boost, is use of paddings in original neurons array. Instead of using the sparsity of neurons, the opposite approach was used – the dense representation of neurons were additionally padded with 0's to remove the need of border conditions check.

For now, these are all optimization techniques which were used in the task. Again, only the sparsity of filter is used to speed up the convolution, as using the neurons sparsity was deemed too difficult to implement with the same efficiency, which was obtained by current approach in the part one of the final project. The main task was to obtain the speedup together with data transfer time over just the kernel from the part one, and this task was successfully completed by using two new optimization techniques – new sparse format for the filter and padding the neurons. Thus, all tasks of pt 2 of final project should be considered as fulfilled.

4. Feedback and comparison between pt1 and pt2

The total speedup achieved compared to pt1 is 20% (8ms instead of 10ms), which can be considered a good result based only on some minor optimizations without any fundamental algorithm change at all. Total overall speedup compared to CPU is about 1950 times.

Since we've used sparse approach in the part one, there wasn't much to do in the part 2, only to further optimize the previous solution to get even better result. Developing the completely new algorithm, which would use the sparsity of both neurons and filter was considered to be a waste of time, since there's a little chance of everybody getting ahead of current result we've got, and with the grading policy currently implemented – that means spending one-two weeks of research only to get one 1-2 points of final grade. For those who didn't implement sparse solution in part 1, the second part may seem to be very difficult to understand. This may become even more difficult, since the code for final project contains bugs, which were fixed by TA. Those who started to work on the project right ahead, had two possible scenarios: they either are so smart, that they can find a bug and tell TA's to fix it, or they'll be "stuck in hell", not understanding why their solution doesn't work at all, or, if it works, it may stop doing so after the bug is fixed, since it wasn't working properly the first time at all. The strategy we've selected was to wait until the 3rd week, and implement all code until weak 4, since it's impossible to use server properly near the deadline toe to everybody trying to test or do their projects.

Additionally, we were worn out by this project completely. The most motivation was "burned out" doing the part one, and the good suggestion for the next year will be not to split the project in parts at all. Still, since we'll probably use neural networks in our future research, this project is considered to be "useful", and we'd like to thank you for the opportunity to work on it and ann god job you've done so far.

END OF DOCUMENT