EE4C07 Lab 3 OpenCL Brain

Michiel van der Vlag (4369327), Dirk Vos (4391470), Mark Schrauwen (4419170)

November 13, 2016

1 Introduction

The human brain is a biological computer which performs many computations with a limited amount of energy. To this day simulating the human brain is a hard en power hungry activity. As can be seen table 1 the processing speed of the human brain is still 3.72 times higher than that of a super computer. And the power consumption of a Super computer is about 495000 times that of a human brain. These differences are huge and with the advancing technologies a future super computer should come up with better numbers.

In this lab we need to implement a brain neuron model simulation program based on the thesis of Du Nguyen Hoang Anh [2]. This program was originally written in Matlab and ported to C by Sebastian Isaza. An optimized CUDA version of the brain simulation code is provided which can be used for inspirational purposes.

This report describes the third assignment in which the C based version of the brain neuron model algorithm needs to be converted to an OpenCL version. Because large parts of the program could run in parallel a high speedup could be attained by running it on a GPU in stead of a CPU.

Table 1: Comparison between the human brain and a super computer

	Human Brain	Super computer	Super computer / Brain
Processing speed	2.2 billion megaflops	8.2 billion megaflops	3.72
Power consumption	20 Watts	9.9 million Watts	495000
Datastorage	3.5 quadrillion bytes	30 quadrillion bytes	8.57

As described in [2] the simulation of neural network is an "embarrassingly parallel problem". This is because each neuron can be described as a single computing element. This means that a neuron can be assigned a hardware thread on a GPU vastly improving the performance of the reference C-program.

2 Implementation

In this section we describe the steps for converting and optimizing the brain neuron model program for use on the OpenCL platform.

2.1 Implementation strategy

In a first analyses of the C-code we gathered the general structure of the brain neuron model C-implementation. The C-code uses a lot of structs to organize the data. Sadly a struct based implementation is not feasible in a OpenCL kernel [3] with OpenCL 1.5.

The OpenCL version on the TU Delft server is a couple of versions lower. This means that an array based approach should be used.

The following implementation steps were followed:

1. Profile the C based version of the neural network
2. Create an inefficient but working version of the neural network program in OpenCL

3. Optimize the OpenCL program to increase the performance (relative to the C based version)

2.2 Profiling

We started with profilin the basic C-code with the "gprof" functionality. By adding the "-pg" flag to the compiler commands extra profiling is generated which grpof can use to produce statictics. As can be seen from table 2 a few methods are associated with the performance of the program. Especially the *ComputeOneCell* method is used for 38.48% of the time. Next in order is the *CompSoma* is used for 23.10% of the time. Both methods are placed inside the ComputeOneCell [1].These methods both contain a lot a calculations. Because of their similarity they could be combined. This will reduce the readability of the code but could improve the performance.

Because the Compute(-kernel) code is much bigger then the Neighour-kernel we want to combine the two kernels to one kernel. This would reduce some kernel overhead.

Table 2: Profiling data of the C-based neural simulation program (simtime=50ms, network size=1024)

% time	Cumulative seconds	Self seconds	Self calls	Total ns/call	ns/call	name
38.49	0.10	0.10	1024000	97.74	97.74	CompDend
23.10	0.16	0.06	1024000	58.64	97.74	CompSoma
15.40	0.20	0.04				ComputeOneCell
11.55	0.23	0.03	1024000	29.32	29.32	SomaCurrVolt
7.70	0.25	0.02				neighbors
3.85	0.26	0.01	1024000	9.77	9.77	SomaCalcium

2.3 Timing

In the OpenCL based program events are used to assure timing was measured as securely as possible. Events are used to mark specific events in OpenCL and thus can guarantee that actions are finished (such as the completion of a kernel or the completion of a write/read action to the GPU device).
Timing is divided into:

- execution of the total program
- initialization of the GPU device
- execution of the kernels
- the data transfer
- the total simulation loop
- optionally also writes to a results file.

2.4 Optimization

From the thesis [2] we gathered that there are a few memory optimization techniques which had been implemented for the CUDA realization, which can be exploited for OpenCL as well.
Before we started working on the memory optimization techniques a few basic optimization have been implemented such that the memory optimization could be implemented easier. In this section basic optimization these techniques are described.

[1] This is a kernel method in the OpenCL code

2.4.1 Basic optimization

The first optimization is reducing the iApp writing to the buffer update. We now only update the iApp value if there is a change in value and have made it global. Resulting in that the clEnqueueWriteBuffer has to do an update from host to device only when this value has changed.

The second optimization is integrating the neighbor kernel (written as neighbour in the code) into the compute kernel. Since both kernels use the cellCompParams array for communicating and are executed it is trivial to merge both kernels. This optimization would mean less memory allocation and overhead for calling the extra kernel.

2.4.2 Shared Memory

One way to optimize the OpenCL implementation is to use the faster types of memory available. The slowest memory is the global memory which is available to all the threads. There is also memory that is available only to the threads in the local work group, which is called shared memory. Shared or local memory is comparable to L1 cache. From the thesis [2], we find that the C2075 platform has 48 KB of shared memory per block which could hold 6144 variables of the type double. From this number we calculate the optimum size for the number of threads in a block. All size should be a multiple of 32, since the platform has a warp of 32 threads (with a dual warp scheduler).

Our compute kernel needs a (local) array of 54 doubles. Therefore we could maximally have $6144/54 \approx 113$ threads per block to fill the shared memory to its maximum. Research points out that a block size of 2x128 or 2x256 in Fermi architecture could be optimal for performance. A low x dimension and a maximal y dimension seem to be preferable. In this case for our situation, we expect that the a block dimension of 3x32 would give the best performance, since this comes closest to the maximal multiple of 32 that could fill our shared memory and with the lowest x dimension. To check which configuration for a multiple of 32 is best, we will check the following block dimensions at our best tested network dimensions: 1x96, 2x48, 3x32, 4x24, 6x16 and 8x12.

It is important to realize one should not use the defined block-sizes during the writes to shared memory in your kernel. Memory alignment is done more dynamically and therefore the functions for obtaining the global and local id should be used. Probably the compiler is optimized such that it uses its own form of memory access. It generates its own pitch to deal with memory access.

With integrating the two kernels we thought we could kill birds with one stone. First of all, already mentioned we could exclude the building, creating, executing, writing of host to data reading, etc; all the extra overhead that is necessary for a second kernel. But also because the shared memory uses a to the compiler known local array, we could do away with the global cellCompParamsPtr array which is much slower memory.

2.4.3 Texture memory

Texture memory is the memory which could be considered as a read-only pointer to a global memory location. This type of memory is useful in situations where caching principles limits the memory access pattern and bounds performance enhancement. Such situations could occur when working with pixels calculation, where neighbor pixels are relevant to the pixel in question.

The situation of the brain simulation algorithm is similar to pixel calculations. Neighbor cells have a certain influence on the cell in question. Texture memory could have a good optimization effect on the algorithm. The texture memory is useful in this situation because it is cached by the geographic allocation of memory instead of by standard cache line [2].

Texture memory uses a two-dimensional coordination system to load certain memory locations in one load, while cache memory uses at least 3 loads [2]. In OpenCL texture memory is available in 1D, 2D and 3D management and implemented with the clCreateImage3D function.

However OpenCL version 1.2 does not support the double data-type and doesn't provide us with a simple wrapper function which allows us to use this type of memory. We could have written our own wrapper function but this is out of the time domain of this project. This also means also that we cannot optimize the branching part from the neighbor kernel. Texture memory has built in functionality that checks whether or not memory operations fall outside of the memory

allocation. Since texture memory cannot be implemented, the if statement should be maintained to do the memory range checking and that makes the code inherently slow.

2.5 Level 1 cache disablement

In addition to the default shared memory size, for CUDA the amount of extra shared memory per block can be determined inside the kernel call [2]. Level 1 cache could be disabled such that the shared memory could be extended to 64 KB. This could be an option worth exploring. With CUDA you could easily enable this by setting the compiler flag -Xptxas -dlcm=cg.

For OpenCL, however, there is no such flag available, nor is documentation about this option hard to impossible to find.

3 Results

Three versions have been used in testing: the originally supplied C-code, an OpenCL baseline version (no real optimization's implemented yet) and a more optimized OpenCL version using shared memory and one fused kernel.

After checking results with the original version, the output to a file was disabled (consumes unnecessary computation power) and timing information is averaged over multiple runs and printed on the screen.

An important note considering the results is that OpenCL uses a lot initialization time compared to the original version, thus only making a difference when a certain simulation time is passed. Which is expected since OpenCl requires a lot more to be initialized (such as platforms, devices, communication channels etc.). Unfortunately the timing on this initialization section is highly non deterministic (also in the original code), because of the malloc's used in the code. Timing jumped around a lot and sometimes a malloc from a small network took up more time then a malloc from a big network, which forced us to leave it out of the timing procedure. To still be fair the reads and writes from and to the opencl device has been included into the timing to be able to compare it with the original C code.

In figure 1 and figure 2 the results can be seen from both single as double precision from the baseline version (no optimizations). From the execution time graphs we can conclude that blocksize it not really related to anything, which is expected behaviour since no shared memory is being used so memory latency will dominate timing.

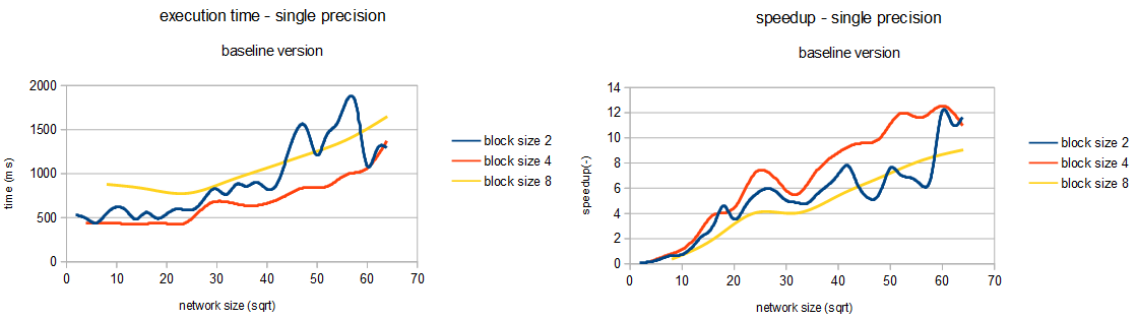

Figure 1: Opencl Baseline Version Single Precision Speedup

From the speedup graphs in figure 1 and figure 2 from the baseline version we can see that double precision does actually much better then single precision, which is not really true if we take a good look at the execution times (around the same value), so this would suggest that the cpu is actually terrible in double precision calculations. This suspicion gets confirmed by looking at the original c code execution times of double precision and single precision in figure 3.

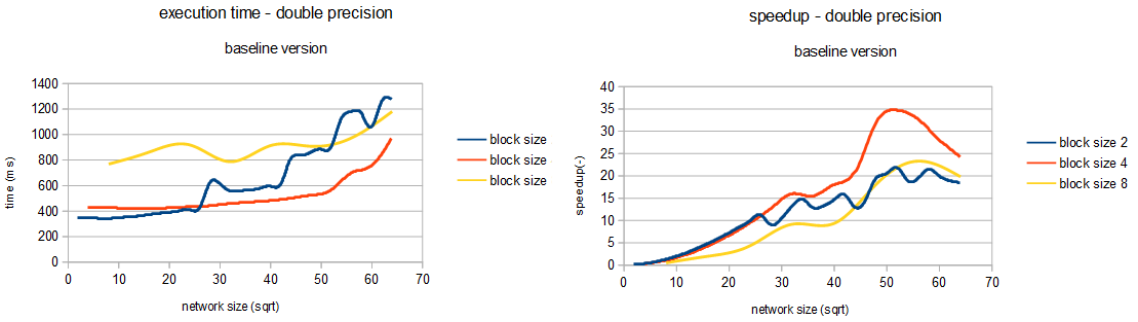

Figure 2: Opencl Baseline Version Double Precision Speedup

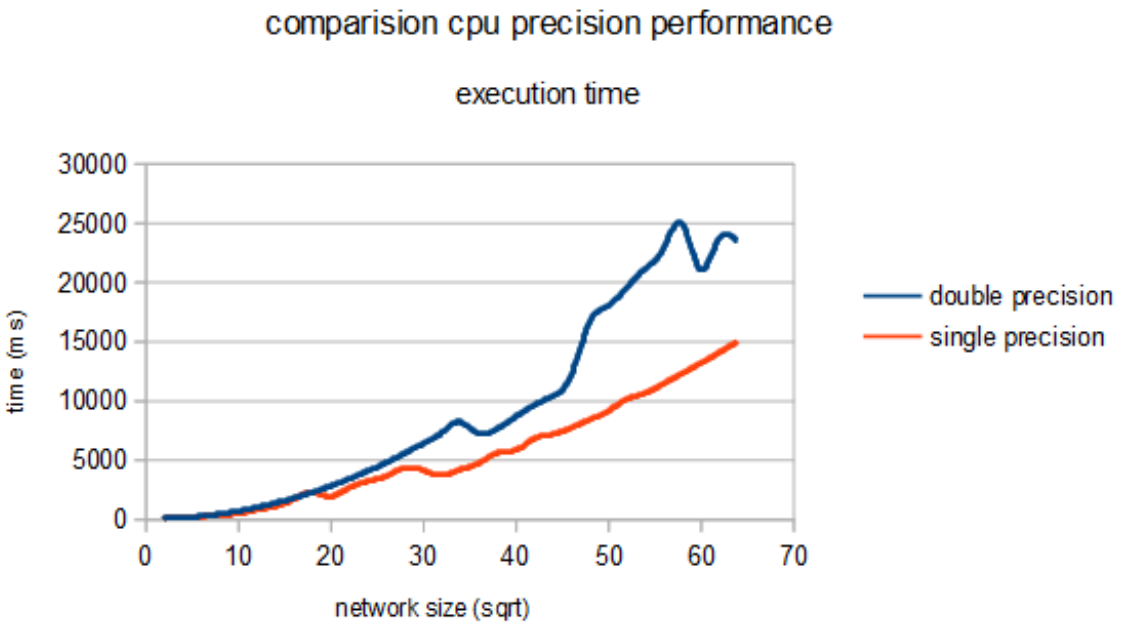

Figure 3: Double precision vs Single Precision on the CPU

The optimized version, which included shared memory and a merged kernel, did a lot better then the baseline version as can be seen in figure 4 and 5. In these graphs you can clearly see the effects of shared memory in the execution time graphs, a bigger blocksize clearly improves execution time, which is exactly what it should do. A blocksize much bigger then 8x8 was not possible since the SM processor would run out of shared memory.

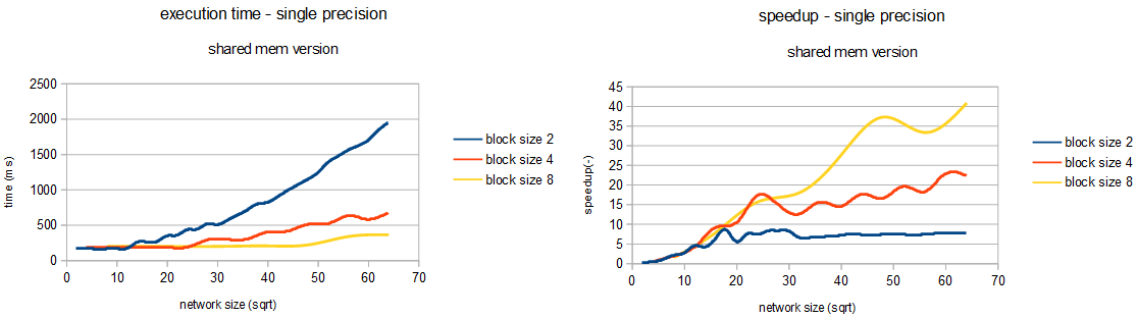

Figure 4: Opencl Shared Mem Version Single Precision Speedup

The difference between double and single precision can yet again be seen in the speedup graphs in figure 4 and 5, especially the execution time graphs show it nicely since they are almost the same. Thus the opencl device has double precision hardware onboard and the cpu does not, making a big speedup possible when looking at double precision capabillity.

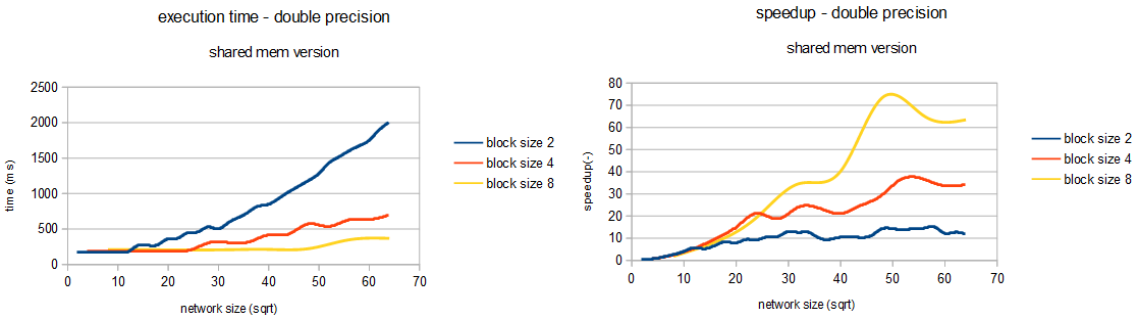

Figure 5: Opencl Shared Mem Version Double Precision Speedup

4 Conclusions

For most optimization technique that are easy implementable in CUDA, we could not find the equivalent for OpenCL. Examples are texture and level 1 cache disabling in favor of a larger shared memory. For the latter, it was hard to find any information at all. This made us realize that CUDA is a more developed language then OpenCL. The TU-Delft server used OpenCL v1.2. That is one of the first versions of OpenCL and later versions are more developed.

An important thing to realize also is that support for double operations needs to be enabled by the pragma: *#pragma OpenCL EXTENSION cl_khr_fp64*. If this line is not added, no warning will be generated. All double declaration will be treated as floats. This results in that output of cell states will be slightly different in comparison to the C output file. When a compiler makes such kind of decision it should output at least a warning.

When implementing shared memory you are easily mistaken to allocate more memory then is maximal possible. Once having accidentally done this the compiler will continue to generate the same error even when shared memory has been reduced to bare minimum; that is with a network and block dimension of 1. Eventually, we realized this could be 'reset' by disabling all optimization's with the compiler flag: *-cl-opt-disable*. Only when the code is compiled once with this flag, it made it possible to continue working with shared memory.

Sometimes, the compiler wouldn't compile the kernels, only when compiler flags were added or removed the compiler gave the corresponding error. It seems as if the compiler fails to see modifications to kernel related files and it has to be forced to re-compile by changing flags. Also the error: *ptxas application ptx input, ... error :.* is an error on which not much documentation is available and will sent you on a wild goose chase with debugging. From [2]

Hence, most of the time we spent on debugging was not on very obvious mistakes as the precision due to missing pragma or declaration, or the shared memory overload. This caused the project to miss its actual goal. The goal of implementing and figuring out the optimization's that could be done with OpenCL.

Nevertheless, we implemented the shared memory and bypassed some of the global memory allocation, we merged two kernels and did some smaller optimization's that resulted in a nice speedup.

References

[1] Du Nguyen, HA and Al-Ars, Zaid and Smaragdos, Georgios and Strydis, Christos *Accelerating complex brain-model simulations on GPU platforms*. Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition

[2] Du Nguyen, *GPU-based simulation of brain neuron models*.

[3] https://software.intel.com/en-us/forums/OpenCL/topic/280668