EE4C07 Lab 2 CUDA

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

October 11, 2016

1 Introduction

In this report we implement an image processing pipeline in the Cuda framework for the NVIDIA GeForce GTX 750 Ti on a Intel(R) Xeon(R) CPU W3530 @ 2.80Ghz. This pipeline consists of 4 stages: gray-scaling, histogram computation for contrast enhancement and eventually smoothing to remove noise. The overall result is a gray-scaled image of the original. The sequential functions which have been provided, contain a lot of nested loops for processing the image which can be processed in parallel on the GPGPU.
This report shows next to the results also the procedure we followed to get to those results and their analysis. We describe the bottlenecks we encountered, the optimization's performed, the benchmark information comparing the sequential versus parallel version, speedup calculations and device utilization and profiling results.

2 Implementation

2.1 Implementation strategy

We tried to work following the *Incremental Build model*. Every iteration we tested the application and the new parts. This was done using the Compare application. We started of with understanding the sequential code on a global level and we tried to determine the bottle necks. One of the obvious bottlenecks was the triangular smoothing function for which the sequential version uses four nested for loops which could lead to a quartic growth rate in run time as matrix sizes grow.
Next of, we had to get familiar with the CUDA platform and tried to implement each converting stage one-by-one. We started with the RGB2GRAY code and tried to convert the code to CUDA following the other stages as mentioned above. Each stage had it own memory allocating and memory copying parts which made the first CUDA implementation much slower then the sequential one.
We lost a significant amount of time by not realizing that the sequential code actual worked on three colors (see section 2.2.3). By translating the sequential code to CUDA code one-on-one the CUDA code did not work. After realizing that we had allocate three times as much memory (for every color) the code started to work

2.2 Implementation steps

2.2.1 Getting information about the graphics card

After getting the basic CUDA code to work we realized that we did not know much about the specifications of the graphics card. We implemented a simple caller functions which printed the specifications of the

graphics card. Some specs we put in global values so they could be used in the CUDA file for calculations (see table 1). This has the advantage to make the code more portable. For instance, we get the number of Streaming Multiprocessors and Threads per Block.

Table 1: Specifications of the GeForce GTX 750 Ti Graphics Card

What		Number
Streaming Multiprocessors	SM	5
Max. Threads per Block	TPB	1024
Max. Thread dimension x	MTx	1024
Max. Thread dimension y	MTy	1024
Max. Thread dimension z	MTz	64
Max. Threads per SM	MSM	2048

2.2.2 Calculating number of threads per block

The basic working CUDA code was based on creating a grid of blocks and creating a grid with the same size as the image. This is not a smart way to do work on a GPU. We decided to divide the work between blocks and threads. The maximum number of threads per block is 1024 and we decided to make square blocks of $32 \cdot 32$ threads. Every block is allocated to one streaming multiprocessor (SM). This information was used to calculate a number of blocks for the width and the height of the image (see code).

2.2.3 Memory Mapping

The first version we build was a version which copied an entire image to and from the GPU for every different image function. This turned out to be quite inefficient since the communication between CPU and GPU is claiming most of the execution time. To optimize this we stuffed every data the GPU needed into one big copy action at the start of the process. Data like pitch, histogram size, contrast threshold etc. is squeezed into one copy action to a big chunk of data where data access is done by a simple memory mapping scheme as is shown in table 2.

Table 2: Memory mapping of different parameters

What	Index
width	0
height	1
histogram size	2
contrast threshold	3
pitch red image	4
pitch green image	5
pitch blue image	6
min (filled on device)	7
max (filled on device)	8
histogram (filled on device	9

2.2.4 Memory Alignment

Accessing individual pixels in an image has a very distinctive access pattern: use the image width and height to pinpoint a pixel. The image however is stored in memory as a big 1D array (all rows are pasted after

one other). If we want to access a pixel somewhere in this stored image, than we use the image width to calculate which row it is in and use the column number to identify the pixel in the found row. Aligning rows could work in our benefit in this case, aligning the memory in a proper way could avoid bank conflicts which means less waiting time for memory. A disadvantage however is that aligning rows in a big picture will really eat up memory. CUDA provides us with 2D memory which does exactly that: it returns a pitch, which is the aligned row length (could also be the same as the image width, depends on what the compiler thinks is best). Accessing a pixel with 2D memory works exactly the same, but now we use the provided pitch to find the pixel instead of the row length (image width). Figure 1 shows the results of using 2D memory[1].

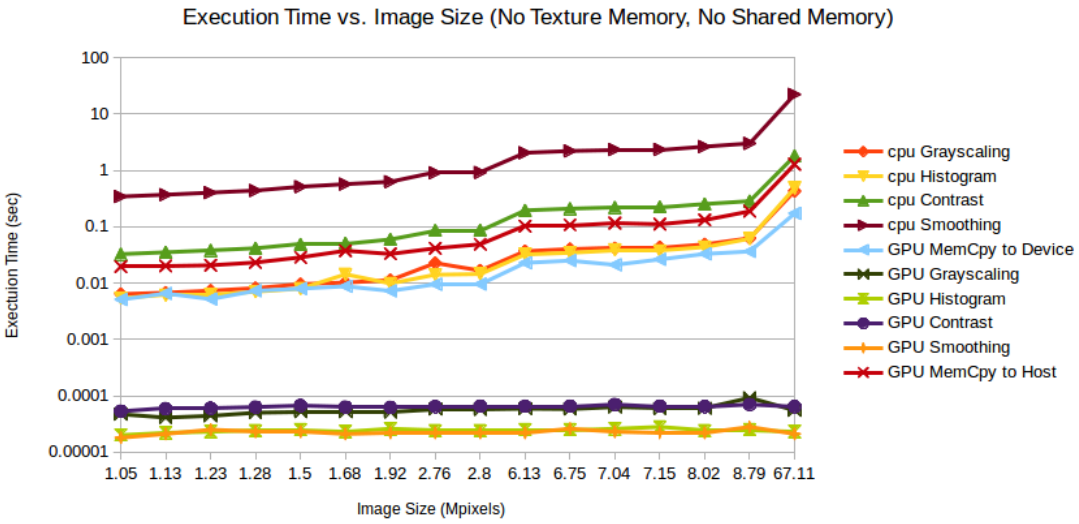

Figure 1: The use of only 2D memory.

2.2.5 Shared Memory

From the gathered results it is getting obvious that access to memory is slow. The GPU has different kinds of memory equipped as is shown in fig. 2. As can be seen in this picture, shared memory is close to the actual hardware threads used in every block. Extracting every pixel again and again from global memory could be slower than first storing it in shared memory using every hardware thread and than doing the computations on the shared memory instead of global memory. However there is some overhead involved: all threads can access shared memory (like its name suggests) so access to it should be managed using synchronize commands. Testing with and without shared memory (see figure 9) did gave some unexpected results: *shared memory actually slows down* in some cases. This could be because kernels are relatively small in computation to fully get the benefit of fast shared memory, making the extra overhead a disadvantage. Shared memory could have been beneficial for the smoothing function, because calculating a single pixel will require its neighboring pixels too. Putting all those depending pixels into shared memory will speedup accesses to them, unfortunately it is easier said then done and because of to many complexities (think about image borders and about using a custom filter instead of a regular averaging smoothing filter) we decided to drop shared memory for this specific function.

[1] `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`

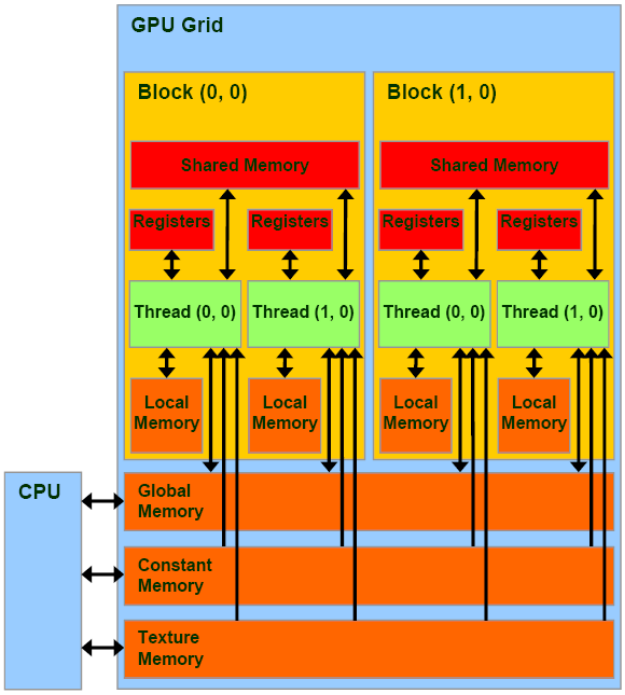

Figure 2: Memory types on the GPU

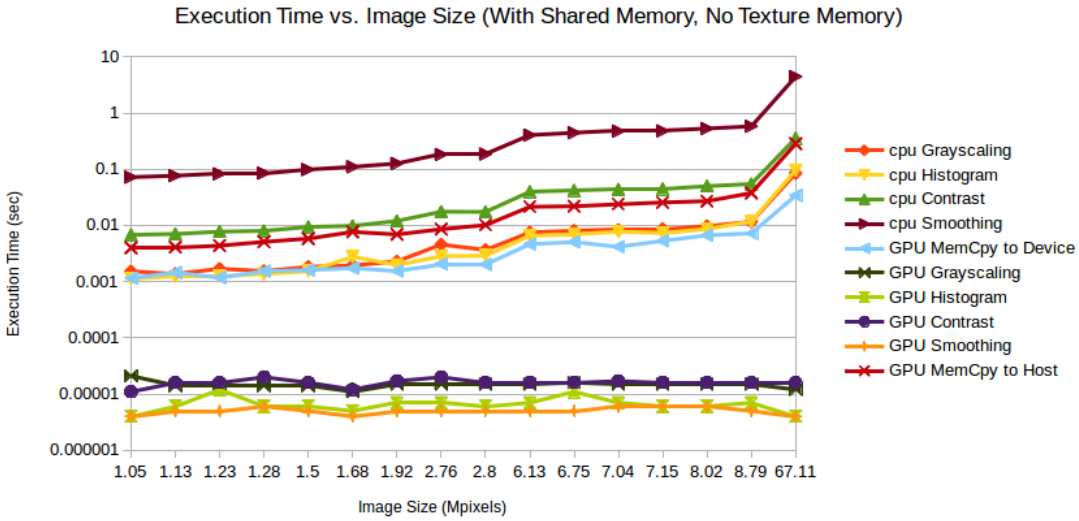

Figure 3: Execution time of a version with shared memory and no texture memory. Every image provided is processed and ordered based on the pixel size. Every measurement per image is repeated five times.

2.2.6 Texture Memory

Another interesting memory optimization is using texture memory instead of global memory. Texture memory is optimized for memory reads. If the kernels will read from memory a lot, than implementing texture

memory could give some benefits. Since all kernels do a lot of read - modify - write operations, texture could be a nice optimization. Accessing pixels using texture memory differs from the other memories: with texture memory you just have to provide the row and column and than the internal mechanism will calculate it to a pixel. Yet again, results were surprising (see figure 9). Texture memory does not look to contribute to any speedup. This is probably because the amount of reads were still to low in comparison with the amount of writes (around 50% - 50%). Figures 4 and 5 show results from using texture memory: one with shared memory and one without. The results from the version without shared memory can be found earlier in figures 1 and 3, these also with and without shared memory. Comparison of speedup between these versions can be found somewhere at the end of the report in figure 9.

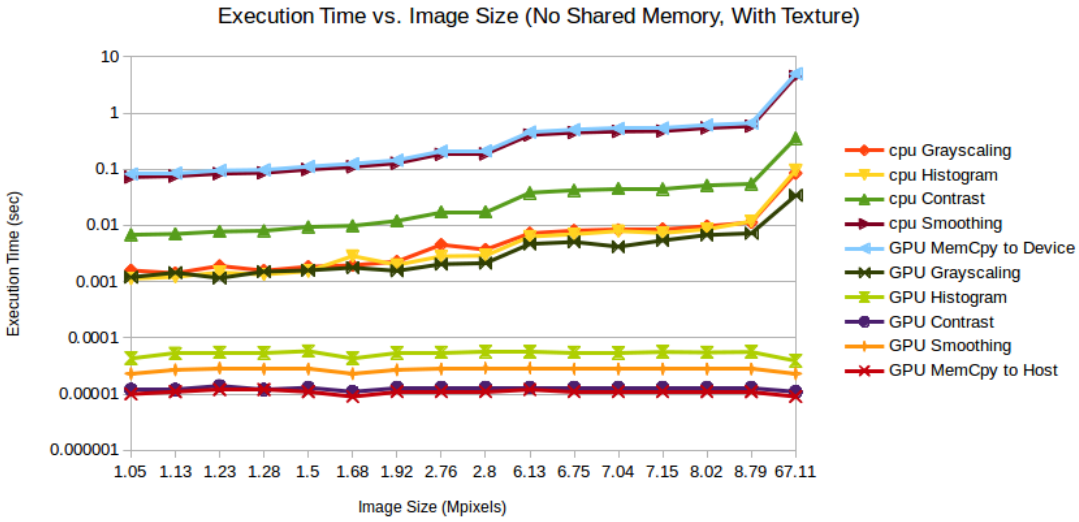

Figure 4: No shared memory With Texture Memory

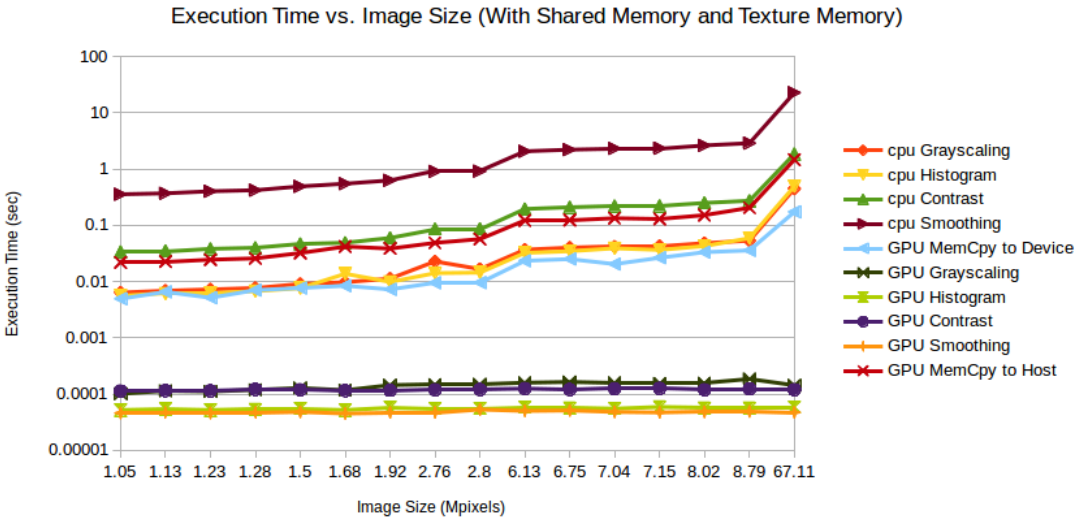

Figure 5: The individual execution times of every function for CPU and for GPU without Shared Memory but with Texture Memory.

2.2.7 Histogram implementation

To convert the sequential histogram code we had to find a way to do addition operation on the GPU. The problem arises (see listing 1.) that a thread execution can not be predicted and different threads can operate on the same variable at the same time.

```
for ( int y = 0; y < height; y++ )
{
        for ( int x = 0; x < width; x++ )
        {
                histogram[static_cast< unsigned int >(grayImage[(y * width) + x])]
                    += 1;
        }
}
```

Listing 1: sequential histogram code

To solve this problem we converted the sequential code (listing 1) to the CUDA version (see: listing 2). By using an atomicAdd() we solved the problem of accessing different variables in an unpredictable manner. The atomicAdd() guarantees that reading and writing to a memory address in global or shared memory happens without any intervention of other threads.

```
// local variables, using memory mapping
unsigned int *histogram = (unsigned int *)&image[2 * width * height];

// add pixel to histogram in one threadsafe operation
atomicAdd(&histogram[static_cast< unsigned int >(image[(y * width) + x])], 1);
```

Listing 2: CUDA histogram code

2.3 Profiling

We have profiled our implementation of our application that uses global, shared and texture memory for CUDA kernel implementation for the largest image and studied the nvprof.trace file in the visual profiling application of Nvidia.

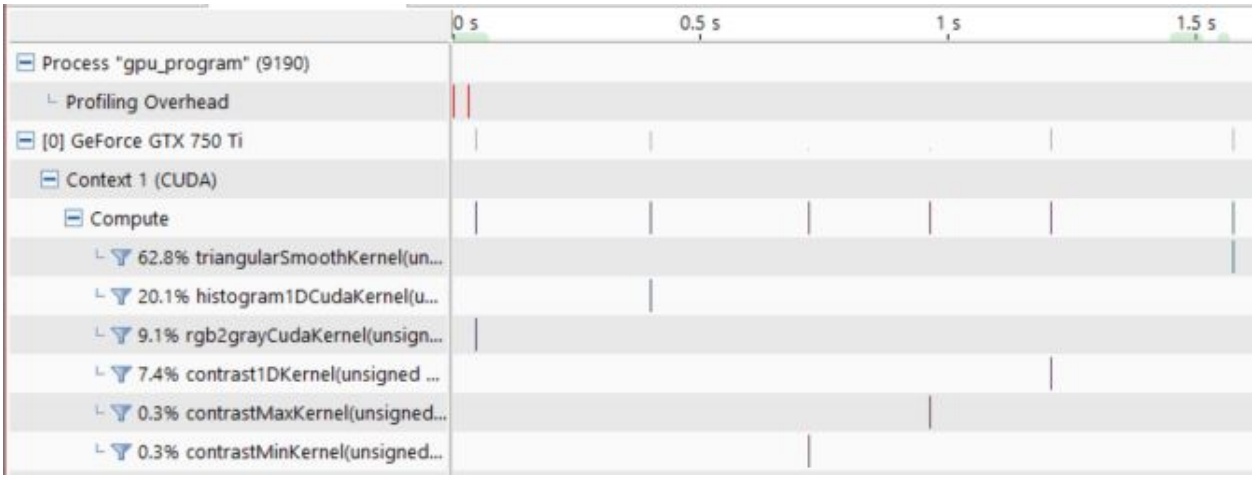

Figure 6: Kernel timing information

In fig. 6 the percentage of utilization of the kernels and also the execution sequence and moment where every kernel is executed are visualized. As we expected the kernel for smoothing is the heaviest kernel that takes the most processing time even when parallelized in Cuda.
The lines represent the moments on which the kernels are executed. The time between execution of the kernels can be explained as the time take for memory reads and writes. The largest part of the execution time is spend on memory overhead, like copying data from CPU to GPU and vice versa.
Also probably the profiling information gathering takes some time to execute which is also reflected in the time between kernels. This would explain also why the total execution time is a few micro seconds longer then normal execution without profiling options enabled.

	triangularSmoothKernel	histogram1DCudaKernel	rgb2grayCudaKernel	contrast1DKernel
Start	1.575 s (1,575,382,415 ns)	397.748 ms (397,748,431 ns)	45.414 ms (45,413,583 ns)	1.206 s (1,206,179,599 ns)
End	1.578 s (1,577,747,370 ns)	398.504 ms (398,503,696 ns)	45.757 ms (45,757,243 ns)	1.206 s (1,206,459,717 ns)
Duration	2.365 ms (2,364,955 ns)	755.265 µs	343.66 µs	280.118 µs
Stream	Default	Default	Default	Default
Grid Size	[35,35,1]	[35,35,1]	[35,35,1]	[35,35,1]
Block Size	[30,30,1]	[30,30,1]	[30,30,1]	[30,30,1]
Registers/Thread	20	10	13	14
Shared Memory/Block	0 B	1 KiB	3 KiB	1 KiB
◢ Efficiency				
Global Load Efficiency	⚠ 12.5%	⚠ 12.5%	⚠ 12.5%	⚠ 12.5%
Global Store Efficiency	⚠ 31.3%	n/a	⚠ 31.3%	⚠ 31.3%
Shared Efficiency	n/a	⚠ 3.4%	⚠ 3.4%	⚠ 3.4%
Warp Execution Efficiency	94.7%	95.6%	95.3%	95.1%
Non-Predicated Warp Executio	89.8%	90.9%	92%	88.6%
◢ Occupancy				
Achieved	84.2%	72.5%	83.9%	83.8%
Theoretical	90.6%	90.6%	90.6%	90.6%
◢ Shared Memory Configuration				
Shared Memory Requested	64 KiB	64 KiB	64 KiB	64 KiB
Shared Memory Executed	64 KiB	64 KiB	64 KiB	64 KiB
Shared Memory Bank Size	4 B	4 B	4 B	4 B

Figure 7: Individual kernel profiling information

In fig. 7 the profiling information for each kernel can be found. In these results it is interesting to see that the occupancy of our kernels approximates the theoretical possible occupancy. This would explain also the high warp execution efficiency. Almost all the threads that are active are also used. This would mean that our grid construction is an efficient one.

Name	Global Load Throughput	Global Store Throughput	Unified Cache Hit Rate
contrast1DKernel(unsigned cha...	20.183 GB/s	11.963 GB/s	69.6%
rgb2grayCudaKernel(unsigned ...	9.835 GB/s	9.751 GB/s	70%
histogram1DCudaKernel(unsig...	3.008 GB/s	0 B/s	87%
triangularSmoothKernel(unsign...	1.429 GB/s	1.417 GB/s	64.8%

Figure 8: Info about the throughput of the application

In figure 8 some additional information about the throughput can be found.

2.4 Speedup

We calculated the speedup for every image found in the image/ folder. The results are presented in fig. 9. The data is presented based on the image size in Mega Pixels. The speedup is calculated by dividing the execution time of the sequential code by the execution time of the CUDA code. Every calculation on every image is repeated for 5 times to get a good average. As can be seen in fig. 9 the maximum speedup is 20 times. The speedup does vary between 13 and 20 times. These differences are due to timing differences between the sequential version and the CUDA version. It is remarkable that bigger image size (e.g. larger than 10 MB) does not influence the speedup. This should however be more examined more in depth.

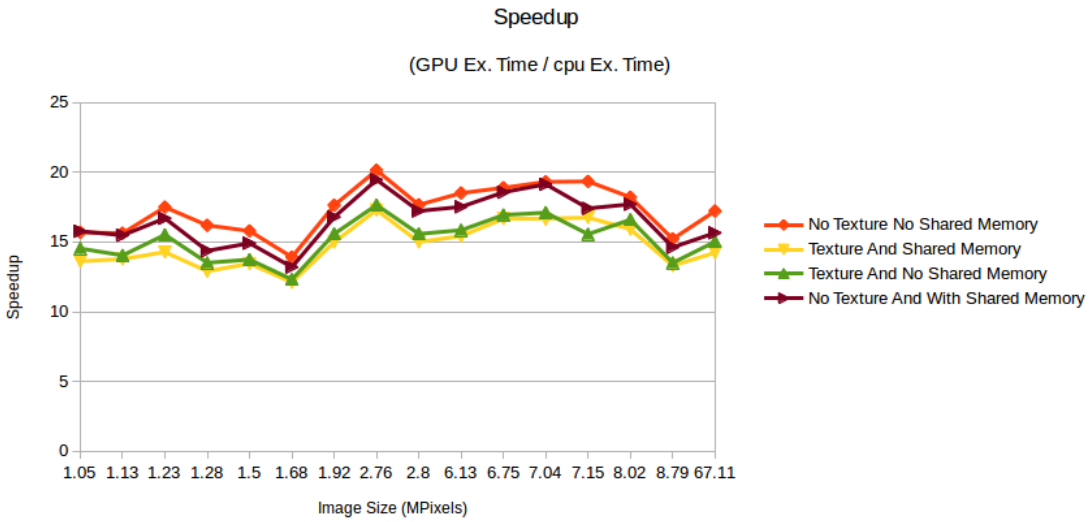

Figure 9: The speedup of all the images with different implementation of the CUDA code.

3 Conclusions

The GPU performance can greatly exceed the CPU performance. By dividing the work among a number of multi-streaming processor and a number of threads the sequential code gets parallelized with great speed-up as a result. Using the Cuda framework could really speed-up applications when a lot of the same computations, for example for graphic calculations, are done.
Aligning memory really works, but this is only an option when enough memory is available. If you have to align a lot of partial used pieces of memory, you would need a lot of extra memory. For example if you would align a row of 1000 to 1024 for 1024 rows would take a lot of extra memory.
To our surprise the use of shared and/or texture memory didn't gave the speed-up as expected. And in some cases the shared memory actually slowed down the execution of the program for the relatively small kernels due to the overhead. An idea to try out could be to use a big kernel that uses only shared memory directly for all calculations instead of writing the results back to global memory after each computation.
The less communication between CPU and GPU is better. As we see from the profiling information the largest part of the execution time is spend on memory copies to and from CPU. One should try to eliminate the memory overhead as much as possible.