

CONCURRENCY & PARALLEL PROGRAMMING

CUDA

*Auteurs: Tom Peerdeman &
René Aparicio Saez*

Datum: 08-12-2012

1 Assignment 6.1 - Wave simulation

1.1 Results

For accuracy all the experiments are ran on the same node after each other, the node was first warmed up by running the program a couple times.

In table 1 we can see the results of 10 runs with different i_{\max} settings and a t_{\max} of 1000. In previous experiments we discarded the highest and lowest value, this is not really useful in these test series since the test results are too close to each other.

We can see from the average value's in the table that the increase in calculation point leads to a decreased average run time. This is a suprising result, since we would expect that more calculations leads to a longer execution time. The decrease in time could be caused by better utilizing the GPU's recources. A GPU works best if it has multiple threads scheduled for one ALU, in the case of multiple threads it can swith very fast between threads and pipeline the instruction's. If we would have just one thread per ALU, the threadss instruction's would be scheduled as well, but when a dependency exists a lot of no-op instruction's have to be inserted. With multiple threads these no-op instructions could be replaced by useful instructions of other threads.

Table 1: Calculation times of the CUDA driven wave simulation in ms using a t_{\max} of 1000.

t _{max} = 1000, 512 threads per block				
i=1e3	i=1e4	i=1e5	i=1e6	i=1e7
4,95	5,05	4,94	3,36	3,32
3,41	3,57	3,32	3,35	3,31
3,42	3,54	3,34	3,36	3,3
4,91	5,05	3,24	3,35	3,31
3,49	3,47	3,3	3,37	3,3
4,91	3,49	3,34	3,36	3,31
4,91	3,5	4,89	3,41	3,32
4,93	5,02	4,94	3,37	3,3
3,46	5,06	4,95	3,38	3,31
4,93	5,01	4,92	3,34	3,3
Average over 10 runs:				
4,332	4,276	4,118	3,365	3,308

Since all the buffers have to be rotated after each time step all the calculating threads have to synchronize after each time step. We could therefore say that the timesteps are calculated sequentially. If we would decrease the amount of time steps by a factor of 10, we would expext a time that is 10 times lower. In table 2 the reduction by a factor 10 is executed. We can see that the increase of time is not 10, also the speed increase decreases with a larger i_{\max} . The decrease in speed increase can be explained by the discovery we made in the begin: more calculation points gives a smaller caluculation time. The increase not being 10 could be explained by utilizing caches more in de later time steps.

Table 2: Calculation times of the CUDA driven wave simulation in μ s using a t_max of 100.

t_max = 100, 512 threads per block				
i=1e3	i=1e4	i=1e5	i=1e6	i=1e7
529	555	374	392	399
391	383	373	397	406
534	554	367	405	396
530	397	541	391	574
539	540	542	403	395
388	395	373	396	383
538	384	542	404	383
379	387	538	396	399
382	554	379	395	392
527	545	375	590	386
Average over 10 runs:				
473,7	469,4	440,4	416,9	411,3
Speed increase vs t_max = 1000:				
9.15	9.11	9.35	8.07	8.04

1.2 Speed comparison

If we compare our CUDA implementation to the previous implementations based on Threads, MPI and openMP, we can see in table 3 that our CUDA implementation is by far the fastest. This result isn't surprising, the GTX 480 used has 15 SM units. Each of these SM units contains 32 ALU's, so in theory we can run $15 * 32 = 480$ threads at the same time. The next fastest is the MPI implementation with 8 nodes running each 8 processes, this counts up as 64 processes/threads. Let's say this implementation scales up perfectly by time, so if we use 7.5 times more threads/processes the time would be 7.5 times lower. The implementation would use 480 threads/processes then and the time would be 16.0187 ms. This time would still be almost 5 times slower than the CUDA implementation, while using the same amount of threads.

The sequential implementation in table 3 is given by the pThreads implementation using 1 thread. This method is truly sequential since it is executed by 1 thread, but the timing also counts some overhead given by the pThreads. Given this we should keep in mind that a pure sequential implementation would be slightly faster than the given time under sequential in table 3. Keeping this in mind, we see that the CUDA implementation is over a thousand times faster than the (almost) sequential implementation.

Table 3: Speed comparison between pthreads, MPI, openMP and CUDA, t_max = 1000, i_max = 1e6.

Method	Average time (ms)	CUDA speedup
CUDA - 512 threads per block	3.365	1.0
MPI - 8 nodes with 8 processes each	120.140	35.703
MPI - 8 nodes with 1 process each	495.634	147.291
OpenMP - 8 threads - static scheduler	661.320	196.529
pThreads - 8 threads	677.751	201.412
MPI - 1 node 8 processes	1186.726	352.667
pThreads - 1 thread / Sequential	3788.914	1125.977

1.3 Block sizes

Running a program using CUDA requires to pass the amount of threads per block into the program, in the wave simulation the number of blocks also depends on the number of threads per block. In table 4 the wave simulation was ran using different amounts of threads per block. As we can see the amount of threads does not influence the calculation time. All the result do not differentiate much from each other, certainly not if we consider some margin of error in the results.

Table 4: Calculation times of the CUDA driven wave simulation using different amounts of threads per block.

16	32	64	128	256	512	1024
3,45	3,46	3,49	3,42	3,46	3,46	3,45
3,45	3,42	3,45	3,42	3,45	3,39	3,48
3,45	3,44	3,45	3,47	3,42	3,44	3,43
3,43	3,47	3,48	3,45	3,47	3,45	3,45
3,42	3,45	3,48	3,4	3,45	3,41	3,66
3,46	3,43	3,42	3,44	3,49	3,46	3,63
3,45	3,42	3,4	3,4	3,44	3,46	3,48
3,45	3,41	3,5	3,44	3,47	3,46	3,45
3,48	3,43	3,44	3,47	3,49	3,43	3,47
3,42	3,45	3,49	3,47	3,45	3,41	3,5
Average over 10 runs:						
3,446	3,438	3,46	3,438	3,459	3,437	3,5

1.4 Results comparison

In figure 1 we can see the result of the wave simulation using the MPI implementation. If we compare this figure to the CUDA result in figure 2 we see that they don't differ very much.

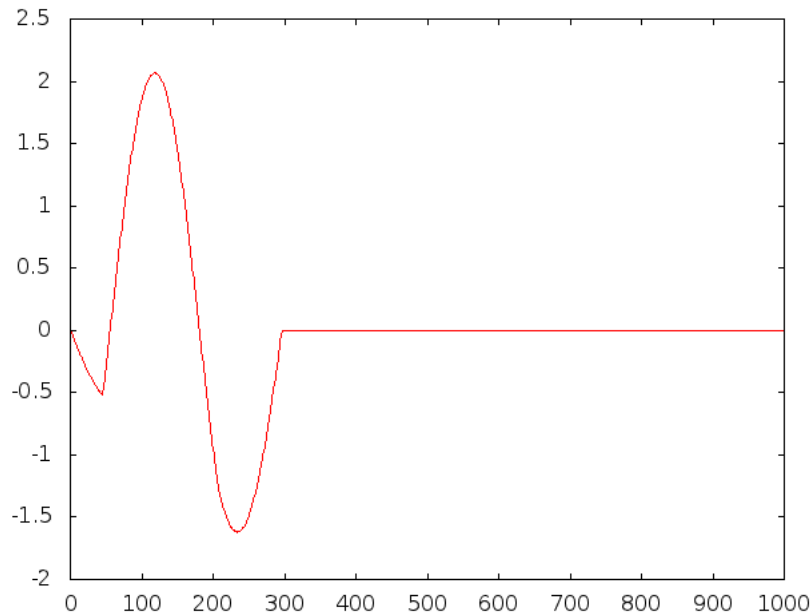


Figure 1: Result of running the wave simulation using MPI with $i_{\text{max}} = 1000$ and $t_{\text{max}} = 100$.

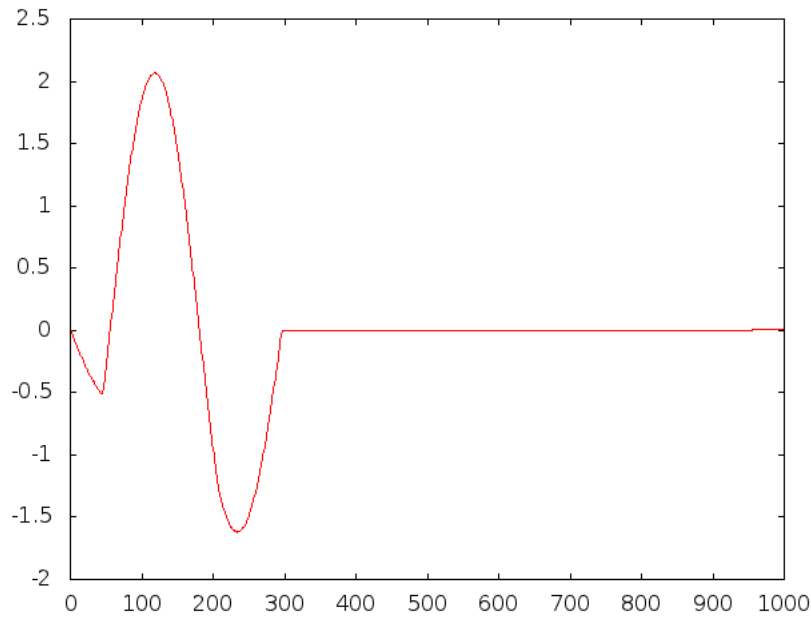


Figure 2: Result of running the wave simulation using CUDA with $i_{\max} = 1000$ and $t_{\max} = 100$.

The difference of results are therefore plotted in figure 3. We can see that the only differences lies in the begin and the end of the data.

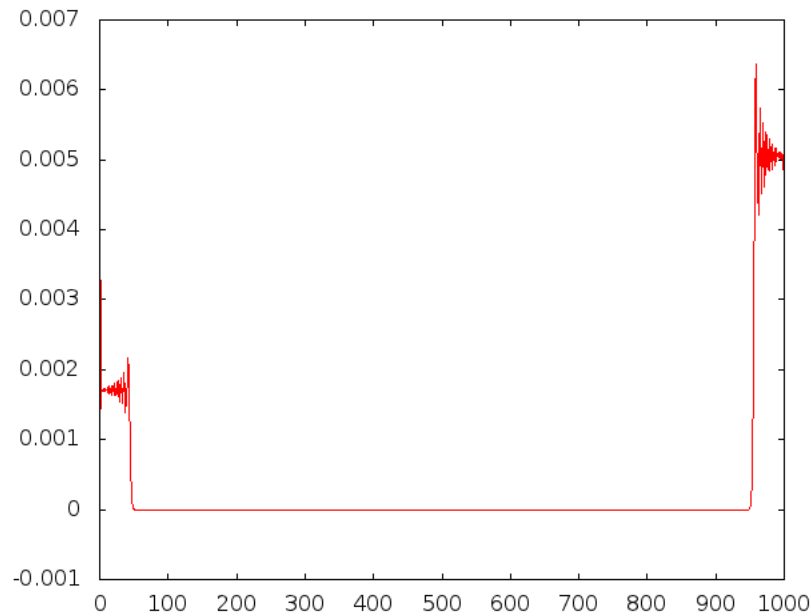


Figure 3: Absolute difference in results of running the wave simulation using MPI and CUDA with $i_{\max} = 1000$ and $t_{\max} = 100$.

These differences can be explained by the implementation of the handling of the edge values. The MPI implementation fixes the edge value's to zero each time step, the CUDA implementation simply ignores the edge values and doesn't do any calculations on it. The points next to the edge

points also utilize the value of the edge points, so these will be different in the CUDA and MPI implementation. This error propagates to the next time step where the next to the next edge value is affected, this leads to the triangle shape from the edges.

2 Assignment 6.2 - Parallel reduction

Calculating the maximum from a list of numbers in CUDA is not an easy task. CUDA does not have global communication, therefore results cannot be exchanged between thread blocks. This does not mean that calculating the maximum from a list of numbers is impossible. If CUDA is used in a tree-based approach, we can work around the communication problem.

Below are the results of a tree-based CUDA program versus a sequential program to find the maximum in a list of floats. All experiments are done on the same node.

Table 5: Time needed to calculate the maximum value of a list of floats

List length	Unoptimized or Optimized	Data					Average
10	CPU	38.2 ns	38.3 ns	38.3 ns	38.1 ns	38.3 ns	38.3 ns
	GPU	33.8 μ s	33.8 μ s	33.7 μ s	33.8 μ s	33.9 μ s	33.8 μ s
100	CPU	363 ns	365 ns	366 ns	366 ns	365 ns	365 ns
	GPU	33.4 μ s	33.6 μ s	34 μ s	42.9 μ s	40.6 μ s	37.3 μ s
1000	CPU	4.11 μ	4.11 μ	4.11 μ	4.11 μ	4.11 μ	4.11 μ
	GPU	42.5 μ s	34.3 μ s	32.6 μ s	34.4 μ s	43.7 μ s	37.5 μ s
10000	CPU	41.6 μ	41.7 μ	41.5 μ	41.6 μ	41.6 μ	41.6 μ
	GPU	33.6 μ s	34.3 μ s	39.9 μ s	34.4 μ s	40.1 μ s	36.5 μ s
100000	CPU	428 μ	430 μ	431 μ	428 μ	428 μ	429 μ
	GPU	35.9 μ s	36.4 μ s	43.5 μ s	36.7 μ s	36.6 μ s	37.8 μ s
1000000	CPU	4.18 ms	4.18 ms	4.16 ms	4.18 ms	4.2 ms	4.18 ms
	GPU	40 μ s	39.6 μ s	39.3 μ s	39.3 μ s	40.7 μ s	39.9 μ s

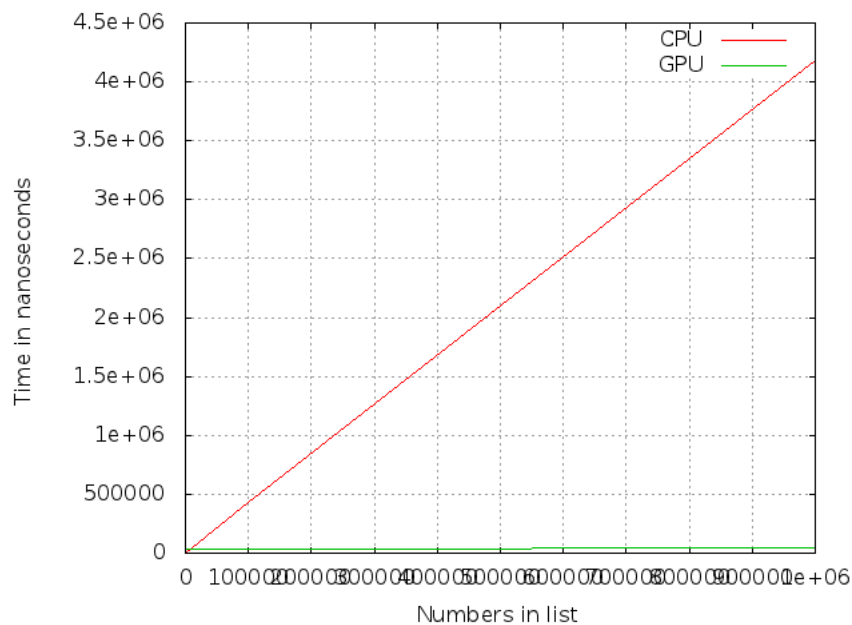


Figure 4: Comparison between a GPU and a CPU to get the maximum from a list

No matter how long the list of numbers is, the speed CUDA needs does not grow a lot. The sequential program however grows linearly. The bigger the list gets, the better CUDA is compared to the sequential program.

2.1 Optimizing code

It is possible to make the CUDA program run even faster. One of the possibilities is to make the list of numbers shared within a thread block. The table of data below shows the time results with this optimization.

Table 6: Time needed to calculate the maximum value of a list of floats using optimized code

List length	Unoptimized or Optimized	Data					Average
10	Unoptimized	33.8 μ s	33.8 μ s	33.7 μ s	33.8 μ s	33.9 μ s	33.8 μ s
	Optimized	33.1 μ s	33.1 μ s	33.6 μ s	33.3 μ s	33.1 μ s	33.3 μ s
100	Unoptimized	33.4 μ s	33.6 μ s	34 μ s	42.9 μ s	40.6 μ s	37.3 μ s
	Optimized	33.7 μ s	33.5 μ s	39.6 μ s	33.2 μ s	34 μ s	34.8 μ s
1000	Unoptimized	42.5 μ s	34.3 μ s	32.6 μ s	34.4 μ s	43.7 μ s	37.5 μ s
	Optimized	39.9 μ s	34.3 μ s	39.8 μ s	33.8 μ s	33.8 μ s	36.3 μ s
10000	Unoptimized	33.6 μ s	34.3 μ s	39.9 μ s	34.4 μ s	40.1 μ s	36.5 μ s
	Optimized	33.5 μ s	34 μ s	33.9 μ s	33.8 μ s	34.2 μ s	33.9 μ s
100000	Unoptimized	35.9 μ s	36.4 μ s	43.5 μ s	36.7 μ s	36.6 μ s	37.8 μ s
	Optimized	37.3 μ s	37.3 μ s	36.5 μ s	37 μ s	37.2	37.1 μ s
1000000	Unoptimized	40 μ s	39.6 μ s	39.3 μ s	39.3 μ s	40.7 μ s	39.9 μ s
	Optimized	39.1 μ s	39.3 μ s	38.7 μ s	38.8 μ s	37.9 μ s	38.8 μ s

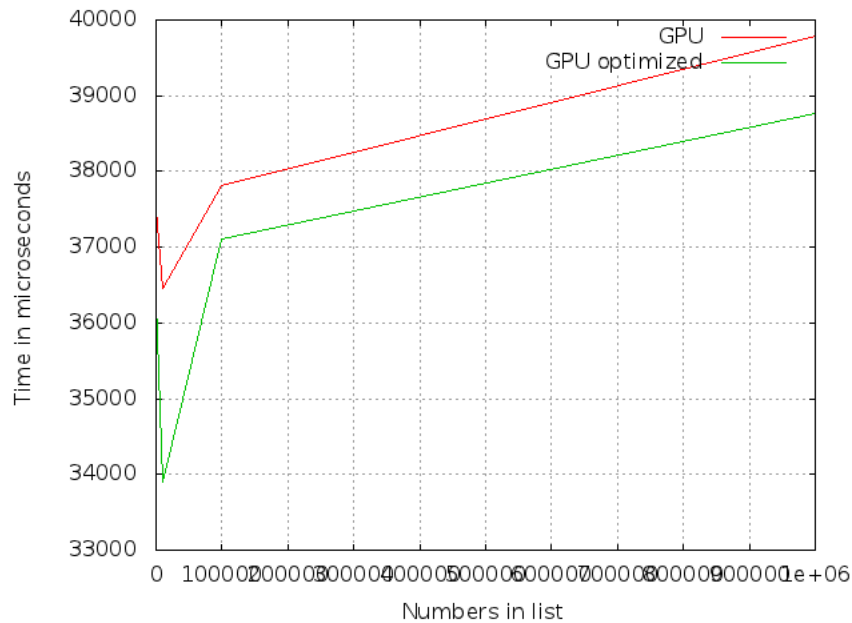


Figure 5: Comparison between unoptimized code and optimized to get the maximum from a list

The optimization will not result in that big of a speed up. But any speedup is welcome.