Lucien Michaël Iseli, 274999
Loris Pilotto, 262651

Assignement 4 - MulArch

1) We implemented GPU_array_process with the 5 steps explained in the lecture 10 of the course that is:
    - allocate memory on the GPU
    - Copy data from CPU to GPU memory
    - Invoke the GPU kernel
    - computation on GPU
    - copying data back to the CPU's memory and deallocate memory on GPU

The GPU kernel implementation is similar to the CPU's implementation. However, the big difference is in the use of blocks. We spent a lot of time trying to figure out which size and shape were optimal.
We began with a single block and a lot of threads. We didn't expect a great performance but the goal was to make the program works. However, we had to be aware of the fact that there is a limitation in the number of threads per blocks (it's 1024 threads per blocks on the Tesla K40). So, this first implementation was not sufficient for the program because it can only run up to a 32x32 square.
Then, we tried to have square blocks with 1024 threads per block. The goal was to use the GPU at its maximum capacity and to be able to run the code on bigger arrays. This implementation worked for all sizes tested in this assignment so up to 1000x1000 square. However, we had bad performance and we can do much better. The 1000x1000 array on 10'000 iterations took 17.87 seconds.

This bad performance is probably due to bad grid design and bad caching results. It didn't exploit the cache well and the fact that a warp will use the same data. We have to take care of false sharing, and in this design there is a lot since many blocks write on the same line.

To solve that, we then decided to give one row per block because we thought writing an entire row per block would be much better and in fact closer to the CPU implementation we did. The performance was significantly better. It took 2.472 seconds to run the 1000x1000 array on 10'000 iterations.

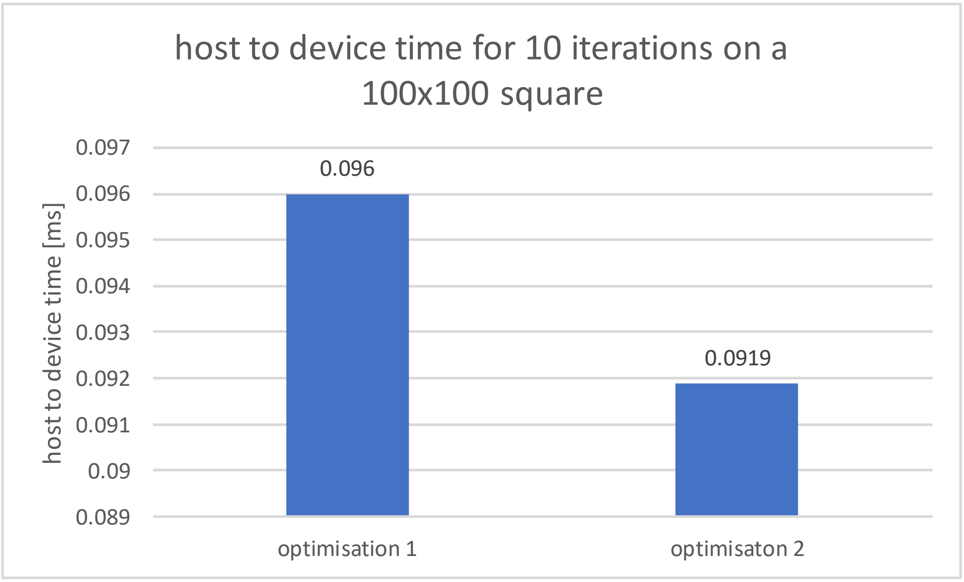
We then had a good design that is even faster than the optimized CPU version but we wanted to try out a couple more optimizations.

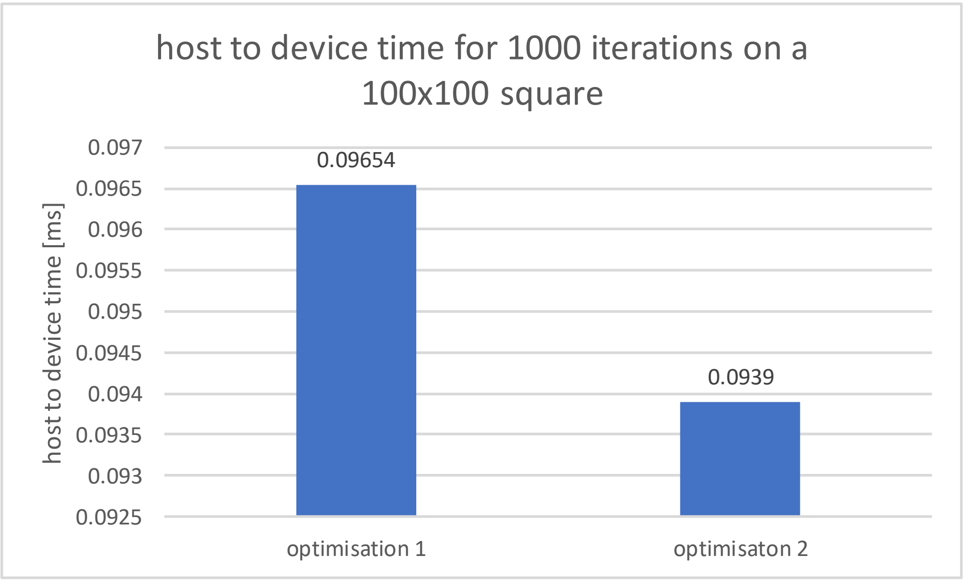
We then tried to make each block do multiple lines instead of just one. We thought it could be a bit faster by caching the inputs the lines will need. We tried with 2 and 4 lines per block but in fact the performance was scarily similar. With a couple of testing the execution time was virtually the same, we couldn't see any speedup or slowdown.

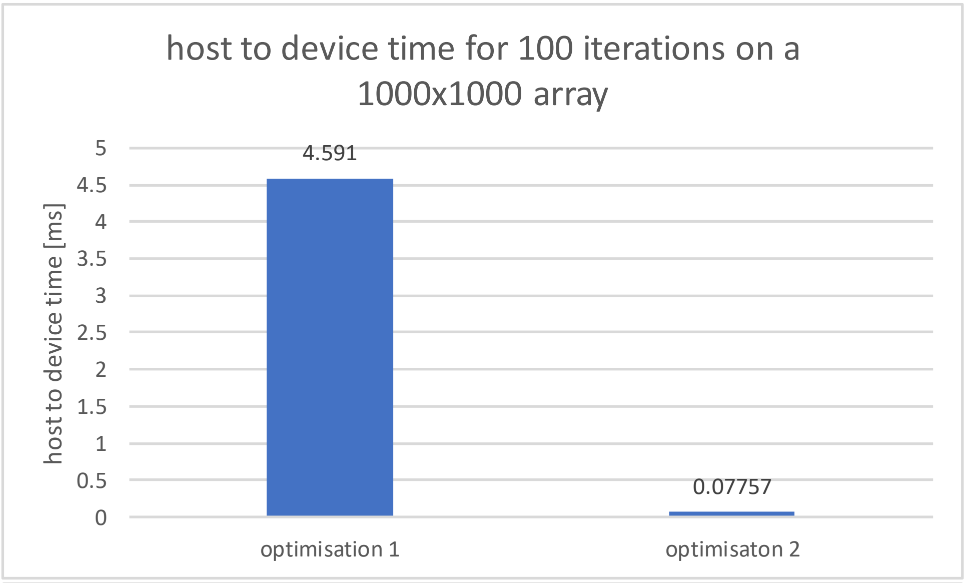
We wanted to try one last thing, in our implementation we copy the input and output of the CPU to the GPU and then copy back the GPU output to the CPU, so the MemCpy from Host to Device takes twice as long. However, copying these arrays is not really mandatory we know their content and it's easy they are empty with '1000' in the middle. Since we initialize the arrays with 0's we added a kernel to put 1000's at the center of the arrays and call this instead of copying the content. This does improve the execution time! The execution and copy device to host doesn't change but the copy from host to device is significantly faster, especially with big arrays. With 1000x1000 array we go from 4.591ms to 0.08109ms.

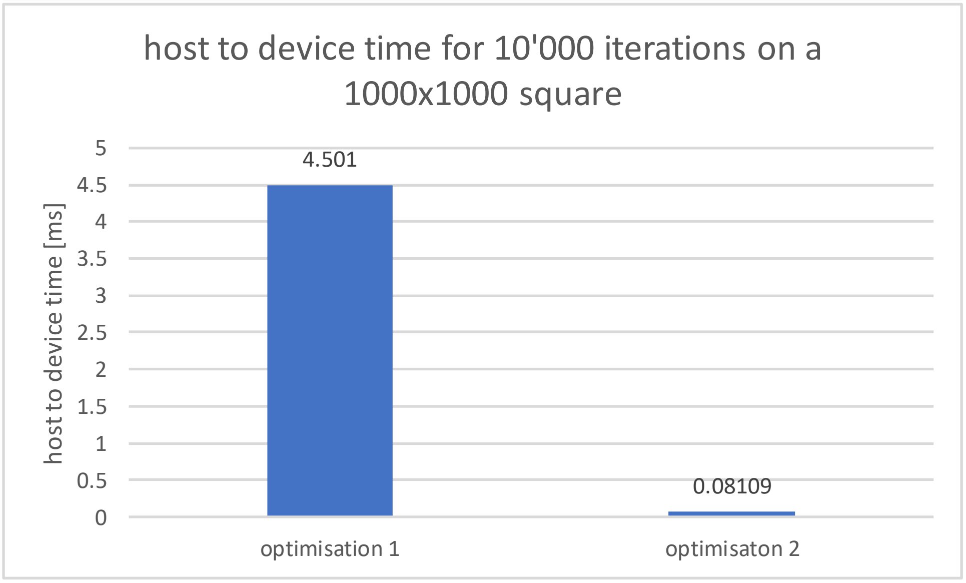
2)

1. Simple square blocks with 1024 threads -> one line per block: Execution time went from 17.87 seconds to 2.472 seconds on 1000x1000 array for 10'000 iterations. So this is our major optimization and it improves better everywhere so we don't think it's really useful to add graphs for that.
2. Each block does 2/4 lines. We saw virtually no improvement or worse performance so we didn't use this change.
3. Removing MemCpy Host to Device and replace it with kernel. This speeds up the host to device copy of array. The computation and device to host remain virtually the same:



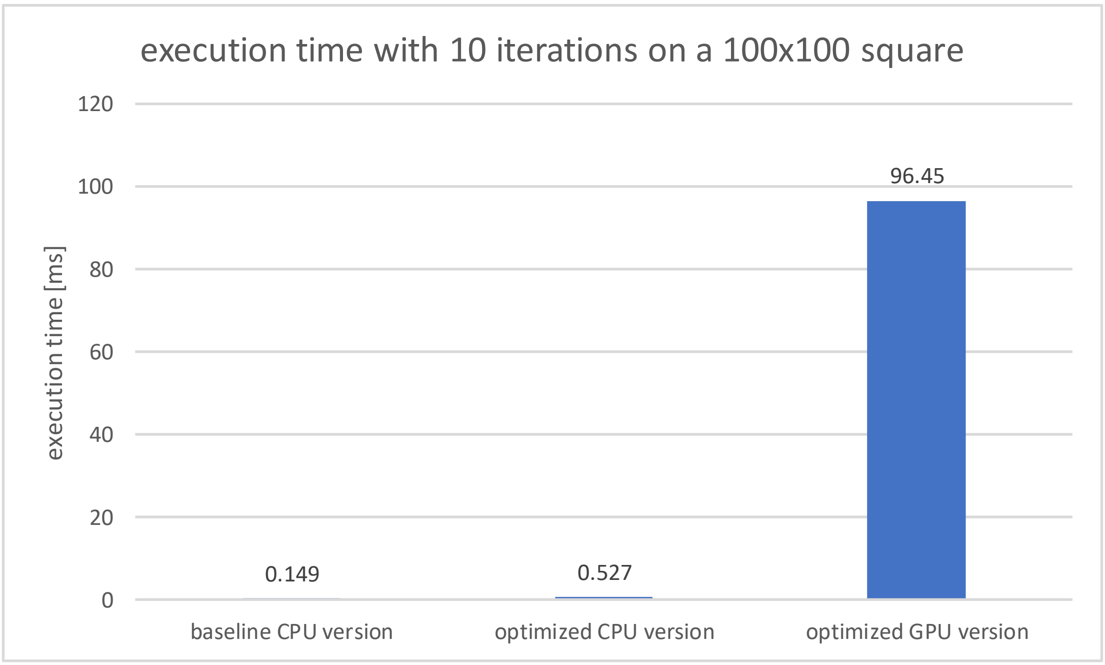


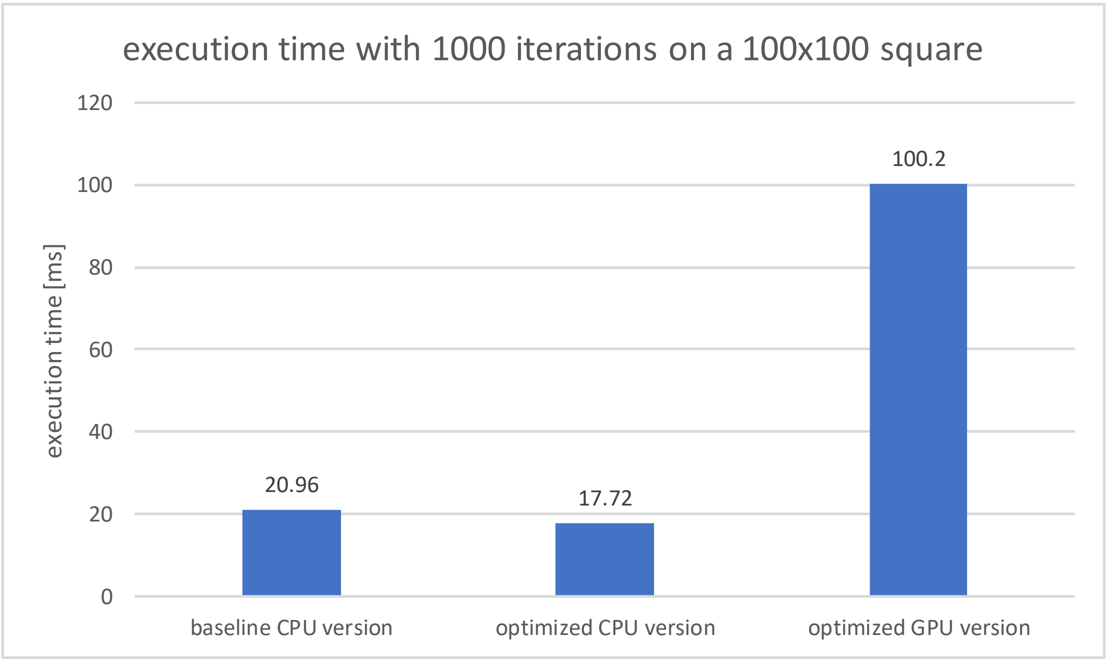


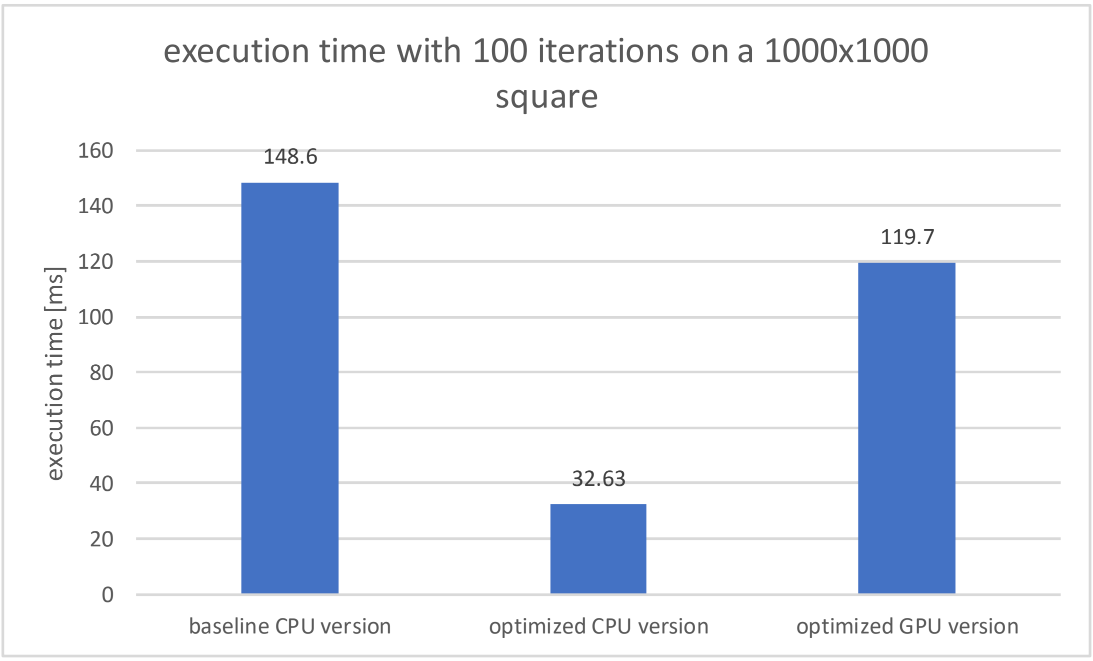


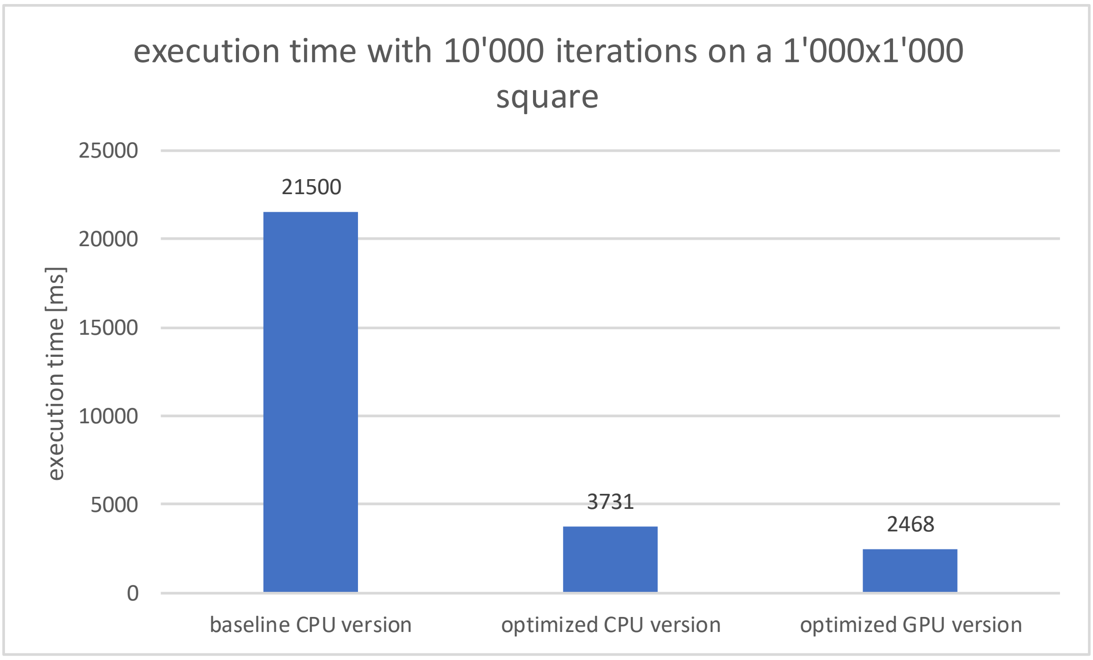
We can see major improvement in the execution time of the copy from host to device due to the fact that we improved something that took linear time on the size of the input to something that takes constant time. However, because of Amdahl's Law the overall execution time doesn't change so drastically because most of the time is spent in the computation phase. The overall execution went from 2.472 seconds to 2.468 seconds on the 1000x1000 array for 10'000 iterations. So we still get a speedup!

3) a) We decided to put each <iteration, length> combinations in a different graph since the time's difference is really big between small inputs like <10, 100> and big inputs like <1'000, 10'000>:

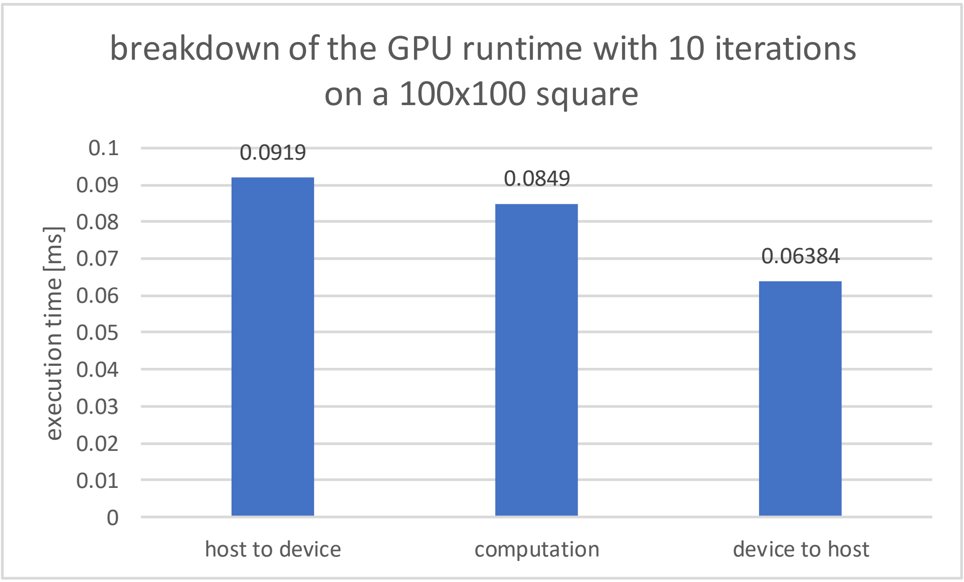

execution time with 10 iterations on a 100x100 square
execution time [ms]
120
100
80
60
40
20
0
0.149
0.527
96.45
baseline CPU version
optimized CPU version
optimized GPU version

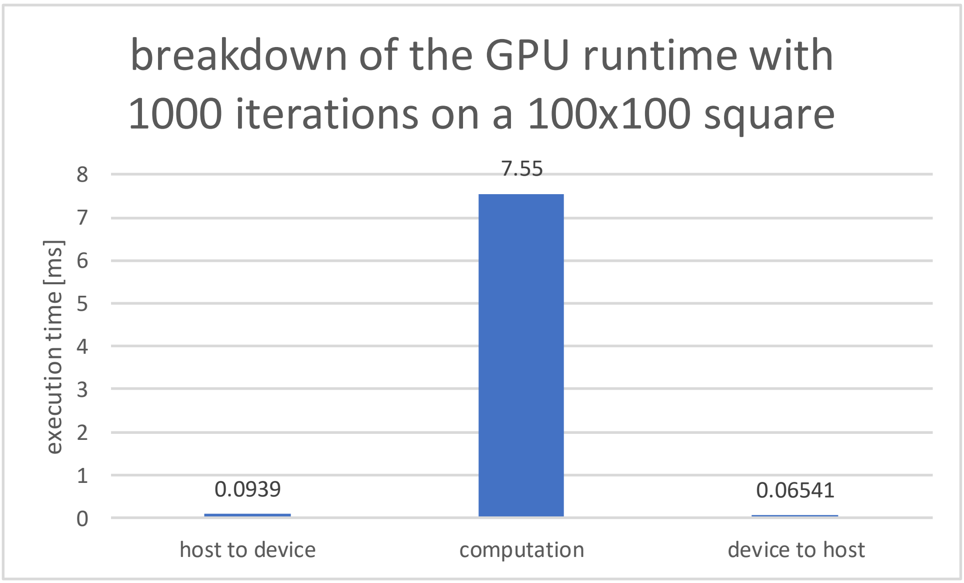

execution time with 1000 iterations on a 100x100 square
execution time [ms]
120
100
80
60
40
20
0
20.96
17.72
100.2
baseline CPU version
optimized CPU version
optimized GPU version

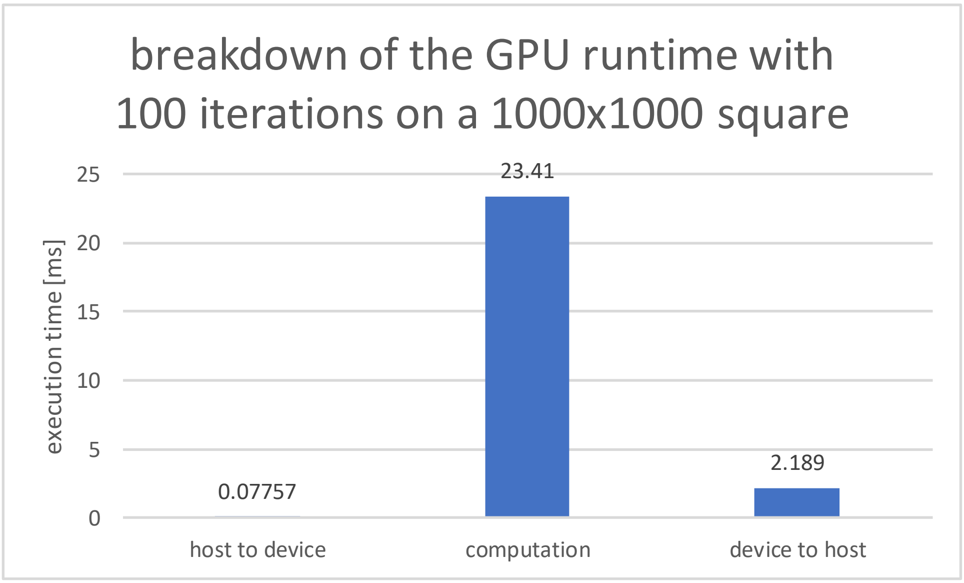

execution time with 100 iterations on a 1000x1000 square
execution time [ms]
0
20
40
60
80
100
120
140
160
148.6
32.63
119.7
baseline CPU version
optimized CPU version
optimized GPU version

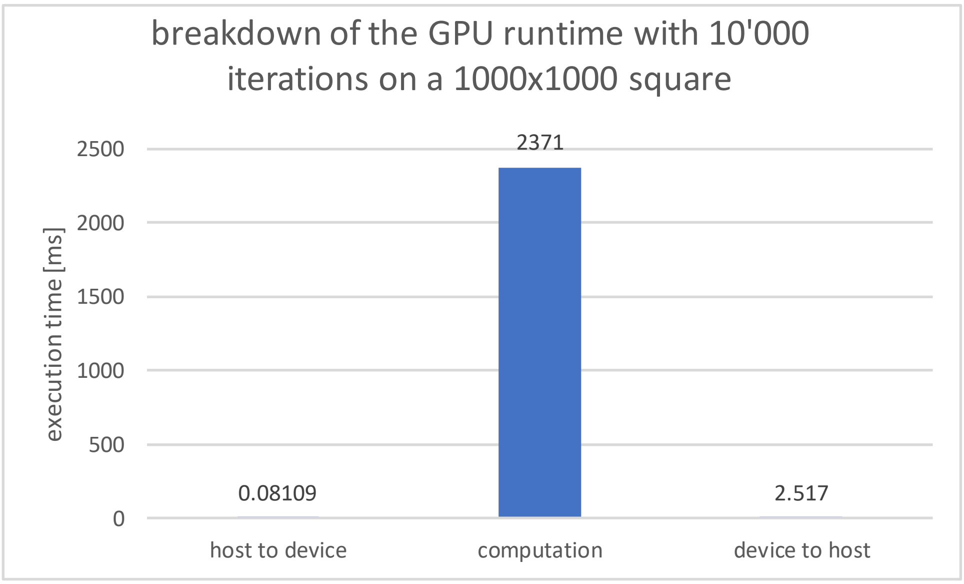

execution time with 10'000 iterations on a 1'000x1'000 square
execution time [ms]
0
5000
10000
15000
20000
25000
21500
3731
2468
baseline CPU version
optimized CPU version
optimized GPU version

b)


breakdown of the GPU runtime with 10 iterations on a 100x100 square
execution time [ms]
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0.0919
0.0849
0.06384
host to device
computation
device to host


breakdown of the GPU runtime with 1000 iterations on a 100x100 square
execution time [ms]
8
7
6
5
4
3
2
1
0
7.55
0.0939
0.06541
host to device
computation
device to host


breakdown of the GPU runtime with 100 iterations on a 1000x1000 square
execution time [ms]
25
20
15
10
5
0
23.41
0.07757
2.189
host to device
computation
device to host



4)

We can see that the baseline CPU is better with very small inputs and iterations, but the optimized CPU quickly becomes better and for big inputs the GPU performs even better that the optimized CPU. This is due to the fact that the parallel CPU has some overhead and the GPU has a LOT of overhead so calling the GPU for a small work isn't useful at all. To check that, we can see that if we sum the GPU execution time of computing and copying array we are very far from the complete execution time it took, due to pre and post processing of creating arrays on the GPU, initializing them to 0, free them etc… However, when the overhead starts becoming minor compared to the computation phase the GPU really stands up and performs better that CPU. This is why even with the 1000x1000 array on 100 iterations the GPU is still worse, it's because there isn't much computation to do, so because of Amdahl's law the % of time the overhead take hugely impacts the overall performance. However, with 10'000 iterations the computation takes a bit of time and now the GPU is faster. So as we saw in class the GPU is indeed really good when there is a lot of similar computation to do.

About the detailed execution times of the GPU, we can see that thanks to our optimization the host to device copy takes about constant time, the device to host of the output takes some time that's close to the size of the array. However it's not completely linear, for example on the 1000x1000 array with 100 iterations it takes 2.189 ms and with 10'000 2.517 ms. We think this is due to the fact that with 100 iterations the array will be mostly empty so the copy is probably better optimized if many of the values of the array are the same. For computation time, we can see that it depends on the number of iterations and on the size of the array. This is normal because the more iteration the more we have to run the kernel so it takes more time. And the bigger the array the more blocks we have, and we of course have many more blocks that we have SMs so it takes longer to run. (We have one block per line).