Parallel Computing
Assignment 2

Onno de Gouw
Laura Kolijn
Stefan Popa
Denise Verbakel

July 12, 2020

Table of Contents

Hardware and software specification

Specifications

Before we start of with discussing the different tasks of this assignment, we will specify the hardware and software that we used in order to create our programs. Since not everybody has access to GPU testing (only Stefan and Onno had dedicated GPUs available), we decided to split the tests in teams of two: Onno and Stefan tested the GPU part of this assignment and Laura and Denise tested the CPU part of this assignment. This was also due to the reason that we could not run the tests on the shootout system at `https://lb.oi.pe`: it took some time until just one run finished and we were not able to see the runtimes of every separate submission.

Hardware and Software Onno de Gouw:

- CPU: i7-6700
- Number of cores: 4
- GPU: Nvidia GeForce GTX 1070 (8GB)
- Clock Frequency: 3.40GHz
- Cache Memory: 8MB Cache
- RAM: 16GB RAM DDR4
- RAM frequency: 2133MHz
- Operating System: Windows 10
- OpenCL C Version: 1.2
- Compiler Version: Microsoft (R) C/C++ Optimizing Compiler Version 19.26.28806

Hardware and Software Stefan Popa:

- CPU: i5-4570
- Number of cores: 4
- GPU: Nvidia GeForce GTX 970 (4GB)
- Clock Frequency: 3.20GHz
- Cache Memory: 6MB Cache
- RAM: 8GB RAM DDR3
- RAM frequency: 1333MHz
- Operating System: Windows 10
- OpenCL C Version: 1.2
- Compiler Version: Microsoft (R) C/C++ Optimizing Compiler Version 19.26.28806

Hardware and Software Laura Kolijn:

- CPU: i5-6200U
- Number of cores: 4
- GPU: Intel® HD Graphics 520 (Skylake GT2)
- Clock Frequency: 2.30GHz
- Cache Memory: 3MB Cache
- RAM: 16GB RAM DDR4
- RAM frequency: 2400MHz
- Operating System: Ubuntu 18.04.4 LTS
- OpenCL C Version: OpenCL C 2.0
- Compiler Version: GCC 7.5.0

Hardware and Software Denise Verbakel:

- CPU: AMD A8
- Number of cores: 4
- GPU: Radeon R5 Graphics
- Clock Frequency: 2.0GHz
- Cache Memory: 2MB Cache
- RAM: 6GB RAM DDR3L
- RAM frequency: 1600MHz
- Operating System: Kali GNU 2020.2
- OpenCL C Version: OpenCL C 1.2 pocl
- Compiler Version: GCC 9.3.0

As can be seen above, the GPUs that were used for testing are the 'Nvidia GeForce GTX 1070 (8GB)' belonging to Onno's PC and the 'Nvidia GeForce GTX 970 (4GB)' belonging to Stefan's PC.
The CPUs that were used are the Intel CPU i5-6200U belonging to Laura's laptop and the AMD A8 CPU belonging to Denise's laptop.

Compiler flags and linkers

As the two GPU testing devices ran on Windows, the compiler flags were set by the IDE Visual Studio. A lot of options are already predefined, but for sure we can say that the option `-Wall` is explicitly used. What we also know is that the linkers to the openCL headers were properly set inside this IDE, since they needed to be setup manually. Next to this, we explicitly set the `/O2` compiler option, which "sets a combination of optimizations that optimizes code for maximum speed"[1], to make sure that also on the Window systems the compiler optimizes for execution time.

The two CPU testing devices both ran on Linux distributions. This meant that they could use a makefile:

```
# Makefile in order to make an executable called relax

# OPENCL := /opt/AMDAPPSDK-3.0
OPENCL   := /opt/intel/system_studio_2020/opencl/SDK

# C flags with strictest warnings.
CFLAGS += -O3 -Wall -g -Wextra -I$(OPENCL)/include -std=c99 -D_GNU_SOURCE

# Linker flags.
LDFLAGS += -L$(OPENCL)/lib/x86_64/sdk -L$(OPENCL)/lib64 -l OpenCL -lrt

all: relax

# Build a binary from C source.
simple.o: simple.c
	$(CC) $(CFLAGS) -std=c99 -c $^

relax: relax.c simple.o
	$(CC) $(CFLAGS) -std=c99 -o $@ $^ $(LDFLAGS)

# Remove the binary.
clean:
	$(RM) relax simple.o
```

This file thus compiled the code for them using the options `-O3`, `-Wall`, `-g`, `-Wextra` and the linkers towards the openCL headers.

[1] `https://docs.microsoft.com/en-us/cpp/build/reference/o-options-optimize-code?view=vs-2019`

Task 1: OpenCL

Implementation process

The main goal of this task was to get a running OpenCL version, in which we succeeded. In the end, we managed to get seven running OpenCL versions, on which we are proud.

For our first running OpenCL version - which can be found in the folder called `Attempt_1` - we only put the relax function in the kernel in the following way:

```
const char *KernelSource =                                          "\n"
  "__kernel void relax(                                              \n"
  "    __global double* in,                                          \n"
  "    __global double* out,                                         \n"
  "    const unsigned int count)                                     \n"
  "{                                                                 \n"
  "    int i = get_global_id(0);                                     \n"
  "    int n = get_global_size(0);                                   \n"
  "    if (i > 0 && i < n-1) {                                       \n"
  "       out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];          \n"
  "    }                                                             \n"
  "    else                                                          \n"
  "       out[i] = in[i];                                            \n"
  "}                                                                 \n"
  "\n";
```

We used the updated versions of the `simple.c` and `simple.h` files that were initially provided via BrightSpace for this version as well as for all the other versions that we will describe next. This first working version creates a work-item for each separate array element and then every work-item relaxes the element that has an index equal to the global identifier. However, this does not hold for the work-items that have as their global identifier 0 or $n-1$ (where n is the global size of the array). These work-items do not need to relax their element, since the first and last element of the array always stay the same.

We also needed to add a function in the do-while loop to release the buffers for kernel arguments every time after the relaxation took place. The reason we needed to do this, is because otherwise the device would run out of memory since it every time allocated memory in order to be able to save the updated arguments. We achieved this with splitting the code of the function `freeDevice()`: we took out the for-loop and pasted that into a new function called `release()`.

The function of `release()` in `simple.c` now looks like the following:

```
cl_int release() {
  cl_int err = CL_SUCCESS;

  for( int i=0; i< num_kernel_args; i++) {
    if( (kernel_args[i].arg_t == FloatArr)
         || (kernel_args[i].arg_t == DoubleArr))
      err = clReleaseMemObject (kernel_args[i].dev_buf);
  }

  return err;
}
```

And the `freeDevice()` function changed to:

```
cl_int freeDevice()
{
  cl_int err;

  err = clReleaseProgram (program);
  err = clReleaseCommandQueue (commands);
  err = clReleaseContext (context);

  return err;
}
```

Once we got this version running, we realized that we transferred the arrays back and forth every time we were running the do-while loop and we also needed to allocate (and free) the space for those arrays every time. Therefore, we decided to try and minimize those transfers from host to device and reuse the allocated space. The resulting code can be found in the folder `Attempt_2`. This was done by placing the function `setupKernel()` outside of the do-while loop and only putting the function `runKernel()` inside of it. We adapted the function `setupKernel()` in the following way: first we created two different kernels. The first kernel with array `a` as the first kernel argument and array `b` as the second kernel argument and the other one with the first two kernel arguments swapped. Next, we adapted the way we call the function `runKernel()` in the do-while loop:

```
do {
  if (count == 0) {
    runKernel(kernels.kernel1, 1, global, local);
    count++;
  } else {
    runKernel(kernels.kernel2, 1, global, local);
    count--;
  }

  iterations++;
} while(!isStable(a, b, n, EPS));
```

This way, we call `kernel1` (with the arguments in the normal order, so array `a` as first kernel argument and array `b` as second kernel argument) every time we are on an even number of iterations and `kernel2` (with array `a` and `b` swapped) every time we are on an odd number of iterations. This has the same effect as swapping the two arrays, but with the difference that now we do not need to transfer the two arrays back to the device again. Since we now reuse the same allocated memory every iteration, we do not need the function `release()` anymore. The freeing of kernel buffers is now placed back in the function `freeDevice()`, such that we again ended up with the original code for this function.

In our `Task_1` directory you will find the different attempts as described above.
The final OpenCL version for task 1 can be found in the file `Attempt_2`.

Task 2: Tune for GPU performance

Implementation process

Our first attempt to optimize the code for the GPU, was by including part of the functionality of `isStable()` in the kernel code. This attempt can be seen in the folder called `Attempt_1_GPU`. We did this by creating a Boolean array in which we stored for each individual element whether it is stable or not. The kernel code for this version looks as follows:

```
const char *KernelSource =                                          "\n"
  "__kernel void relax(                                              \n"
  "    __global double* in,                                          \n"
  "    __global double* out,                                         \n"
  "    __global bool* stable,                                        \n"
  "    const double eps,                                             \n"
  "    const unsigned int count)                                     \n"
  "{                                                                 \n"
  "    int i = get_global_id(0);                                     \n"
  "    int n = get_global_size(0);                                   \n"
  "    if (i > 0 && i < n-1) {                                       \n"
  "       out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];          \n"
  "    } else {                                                      \n"
  "       out[i] = in[i];                                            \n"
  "    }                                                             \n"
  "    stable[i] = fabs(in[i] - out[i]) <= eps;                      \n"
  "}                                                                 \n"
  "\n";
```

In addition to this change, we also had to adapt the function `isStable()` in the following way:

```
bool isStable(bool *stable, int n)
{
   int i;

   for(i=1; i<n-1; i++) {
      if (!stable[i])
         return false;
   }
   return true;
}
```

So, instead of computing for each element whether or not it is stable in this function, we now only check for a `false` element in the array `stable`, indicating that there is an element that is not yet stable.

After finishing this function, we realized we could make it faster by reducing the Boolean array before we send the result back to the host. This attempt can be seen in the folder `Attempt_2_GPU`. We did this by creating a local array for each work group in which we again store for each element in that work group whether it is stable or not.

After that, we reduce the values in those local arrays in each work group in the first element of that array. To do this, we found a useful handout[2] which we used. In the end, we write the result of this reduction to the global array `stable` at the index corresponding to the group identifier. At first, we used barriers in order to synchronize the code. This looked like the following:

```
const char *KernelSource =                                                    "\n"
  "__kernel void relax(                                                        \n"
  "    __local  bool* stable_l,                                                \n"
  "    __global double* in,                                                    \n"
  "    __global double* out,                                                   \n"
  "    __global bool* stable,                                                  \n"
  "    const double eps,                                                       \n"
  "    const unsigned int count)                                               \n"
  "{                                                                           \n"
  "    int i = get_global_id(0);                                               \n"
  "    int n = get_global_size(0);                                             \n"
  "    int wg_size = get_local_size(0);                                        \n"
  "    int wg_i = get_local_id(0);                                             \n"
  "    int wg_num = get_group_id(0);                                           \n"
  "                                                                            \n"
  "    if (i > 0 && i < n-1) {                                                 \n"
  "       out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];                    \n"
  "    }                                                                       \n"
  "    else {                                                                  \n"
  "       out[i] = in[i];                                                      \n"
  "    }                                                                       \n"
  "    stable_l[wg_i] = fabs(in[i] - out[i]) <= eps;                           \n"
  "    for(int offset = 1; offset < wg_size; offset *= 2)                      \n"
  "    {                                                                       \n"
  "       int mask = 2*offset - 1;                                             \n"
  "       barrier(CLK_LOCAL_MEM_FENCE);                                        \n"
  "       if ((wg_i & mask) == 0)                                              \n"
  "       {                                                                    \n"
  "          stable_l[wg_i] = stable_l[wg_i] && stable_l[wg_i + offset];       \n"
  "       }                                                                    \n"
  "    }                                                                       \n"
  "    barrier(CLK_LOCAL_MEM_FENCE);                                           \n"
  "    if(wg_i == 0)                                                           \n"
  "       stable[wg_num] = stable_l[0];                                        \n"
  "}                                                                           \n"
  "\n";
```

[2] http://web.engr.oregonstate.edu/~mjb/cs575/Handouts/opencl.reduction.2pp.pdf

However, we found out that not all race-conditions are bad and thus we deleted the barriers in between the code, speeding up the final second version of our GPU optimization:

```
const char *KernelSource =                                                    "\n"
  "__kernel void relax(                                                        \n"
  "    __local  bool* stable_l,                                                \n"
  "    __global double* in,                                                    \n"
  "    __global double* out,                                                   \n"
  "    __global bool* stable,                                                  \n"
  "    const double eps,                                                       \n"
  "    const unsigned int count)                                               \n"
  "{                                                                           \n"
  "    int i = get_global_id(0);                                               \n"
  "    int n = get_global_size(0);                                             \n"
  "    int wg_size = get_local_size(0);                                        \n"
  "    int wg_i = get_local_id(0);                                             \n"
  "    int wg_num = get_group_id(0);                                           \n"
  "                                                                            \n"
  "    if (i > 0 && i < n-1) {                                                 \n"
  "       out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];                    \n"
  "    } else {                                                                \n"
  "       out[i] = in[i];                                                      \n"
  "    }                                                                       \n"
  "    stable_l[wg_i] = fabs(in[i] - out[i]) <= eps;                           \n"
  "    for(int offset = 1; offset < wg_size; offset *= 2)                      \n"
  "    {                                                                       \n"
  "       int mask = 2*offset - 1;                                             \n"
  "       if ((wg_i & mask) == 0)                                              \n"
  "       {                                                                    \n"
  "          stable_l[wg_i] = stable_l[wg_i] && stable_l[wg_i + offset];       \n"
  "       }                                                                    \n"
  "    }                                                                       \n"
  "                                                                            \n"
  "    if(wg_i == 0)                                                           \n"
  "       stable[wg_num] = stable_l[0];                                        \n"
  "}                                                                           \n"
  "\n";
```

Although this version was faster than the previous one, we still could improve the performance of this code.

The next idea we had, was to use only one Boolean value instead of reducing a whole array of Boolean values: when we encounter an unstable element, we set this value to `false` and otherwise we leave it `true`. The first work-item will initially set this value to `true`. Unfortunately, in OpenCL, if they are not constants, the arguments need to be arrays. Therefore, this Boolean value is an array of one element. After this change, the kernel code looks as follows:

```
const char *KernelSource =                                          "\n"
  "__kernel void relax(                                              \n"
  "    __global double* in,                                          \n"
  "    __global double* out,                                         \n"
  "    __global bool* stable,                                        \n"
  "    const double eps,                                             \n"
  "    const unsigned int count)                                     \n"
  "{                                                                 \n"
  "    int i = get_global_id(0);                                     \n"
  "    int n = get_global_size(0);                                   \n"
  "    if (i == 0)                                                   \n"
  "       stable[0] = true;                                          \n"
  "    if (i > 0 && i < n-1) {                                       \n"
  "       out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];          \n"
  "    } else {                                                      \n"
  "       out[i] = in[i];                                            \n"
  "    }                                                             \n"
  "    if (fabs(in[i] - out[i]) > eps)                               \n"
  "       stable[0] = false;                                         \n"
  "}                                                                 \n"
  "\n";
```

To reflect this change, we removed the function `isStable()` and instead we checked the value of `stable[0]` in the following way:

```
do {
   ...
} while(!stable[0]);
```

This change of code can be seen in the final folder `Attempt_3_GPU`. This version will be tested below.

Note that we first decided for each device separately which work group size is most suitable for testing. Also note that we tested the versions described in 'Task 3: Tune for CPU performance' on the GPU. These did not have better results on the GPU.

In our `Task_2` directory you will find the different attempts as described above.
The final OpenCL version for task 2 can be found in the file `Attempt_3_GPU`.

Results of Onno de Gouw: tested GPU version

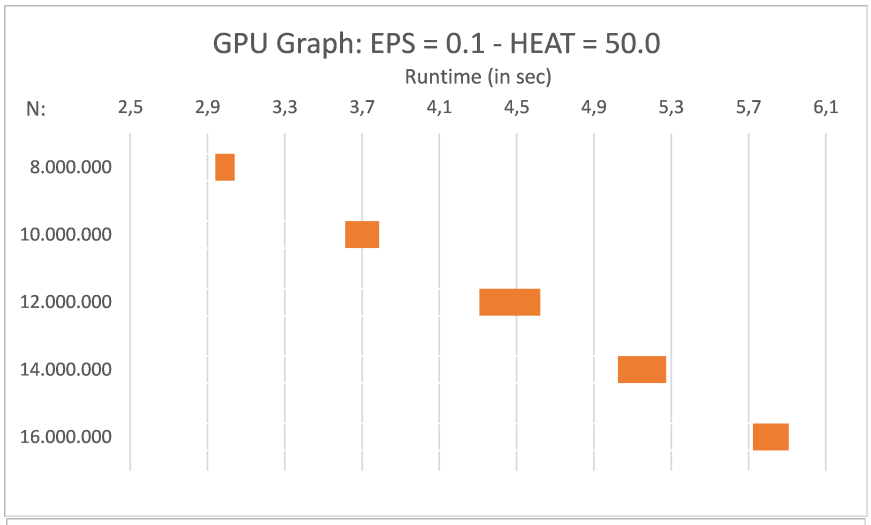
Before being able to measure the runtimes, a good work group size for the GPU had to be chosen. After running a couple of tests with different work group sizes, we were able to conclude that a size of 64 resulted in the fastest runtimes. Therefore, this value was used for testing.

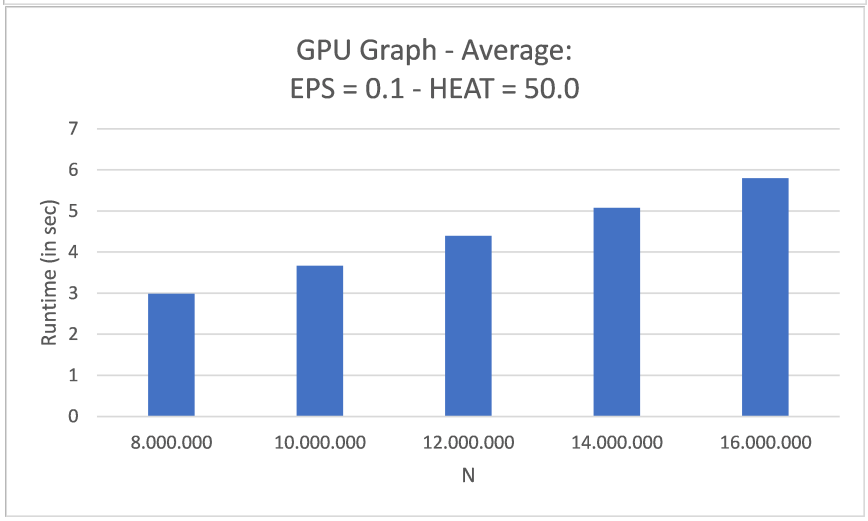
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 50.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	3.04103	3.00484	2.98409	3.03092	2.96813	2.97025	2.96936
10,000,000	3.69336	3.69422	3.68027	3.67021	3.61569	3.64845	3.61416
12,000,000	4.62201	4.37502	4.30761	4.33895	4.37004	4.36534	4.31191
14,000,000	5.27391	5.04944	5.02352	5.08096	5.06256	5.04348	5.06803
16,000,000	5.90766	5.73482	5.74038	5.76586	5.84295	5.74533	5.83288

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	2.94008	3.01188	2.96952	2.989010	3.04103	2.94008
10,000,000	3.78816	3.61311	3.64564	3.666327	3.78816	3.61311
12,000,000	4.46698	4.40479	4.35725	4.391990	4.62201	4.30761
14,000,000	5.04631	5.06577	5.03448	5.074846	5.27391	5.02352
16,000,000	5.72295	5.81617	5.85059	5.795959	5.90766	5.72295

The graphs below will display the results just obtained:



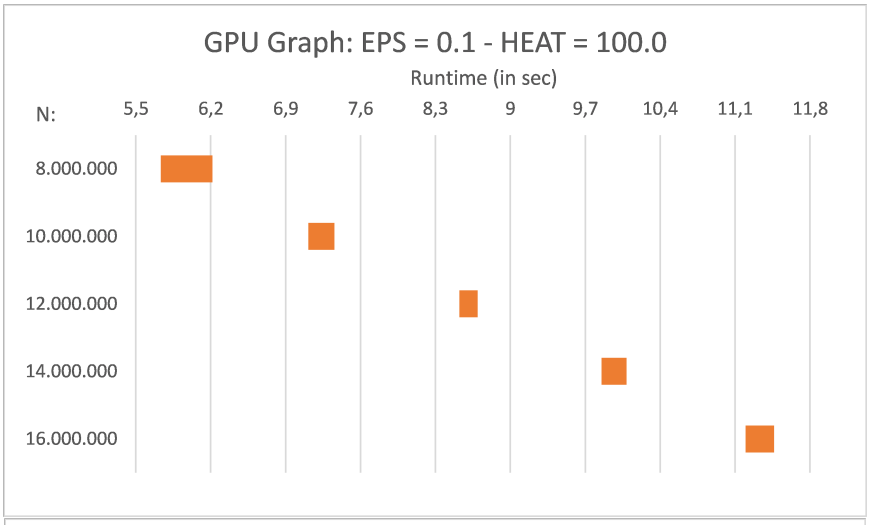


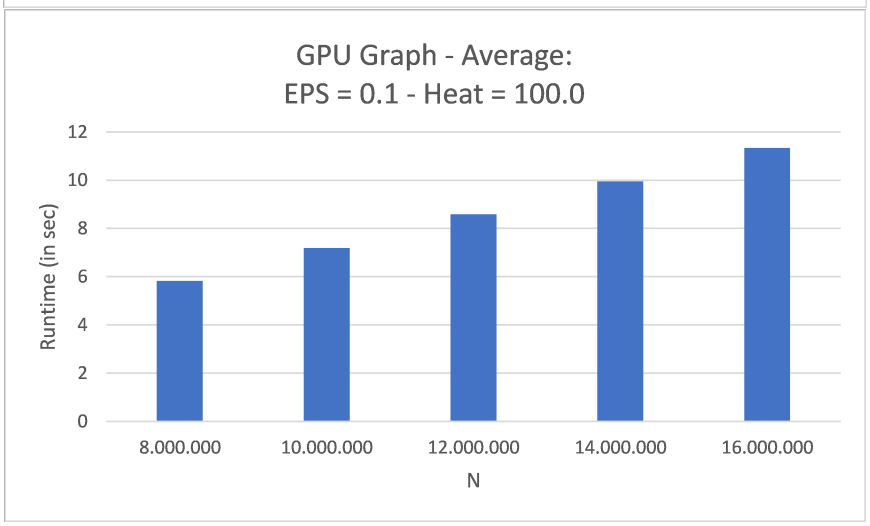
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 100.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	6.21707	5.75573	5.81291	5.76968	5.77969	5.81381	5.76022
10,000,000	7.35658	7.20145	7.25473	7.11767	7.17575	7.22701	7.15559
12,000,000	8.61426	8.52308	8.57156	8.57621	8.65761	8.55598	8.56805
14,000,000	10.08635	10.03294	9.85976	9.85271	10.01775	10.06305	9.96555
16,000,000	11.34778	11.40301	11.42899	11.33865	11.22202	11.25548	11.19712

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	5.73281	5.81643	5.80173	5.826008	6.21707	5.73281
10,000,000	7.11339	7.13792	7.11761	7.185770	7.35658	7.11339
12,000,000	8.57276	8.69491	8.57625	8.591067	8.69491	8.52308
14,000,000	9.89796	9.89201	9.85459	9.952267	10.08635	9.85271
16,000,000	11.46585	11.29134	11.45715	11.340739	11.46585	11.19712

The graphs below will display the results just obtained:



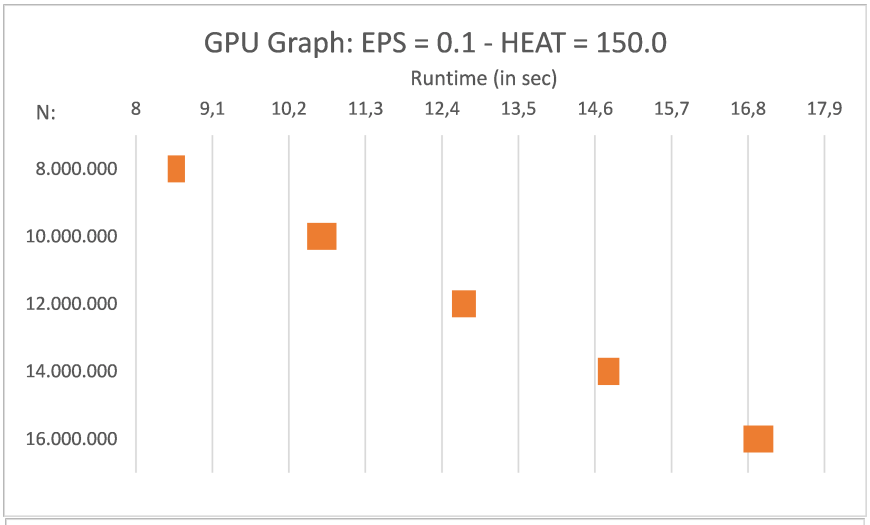


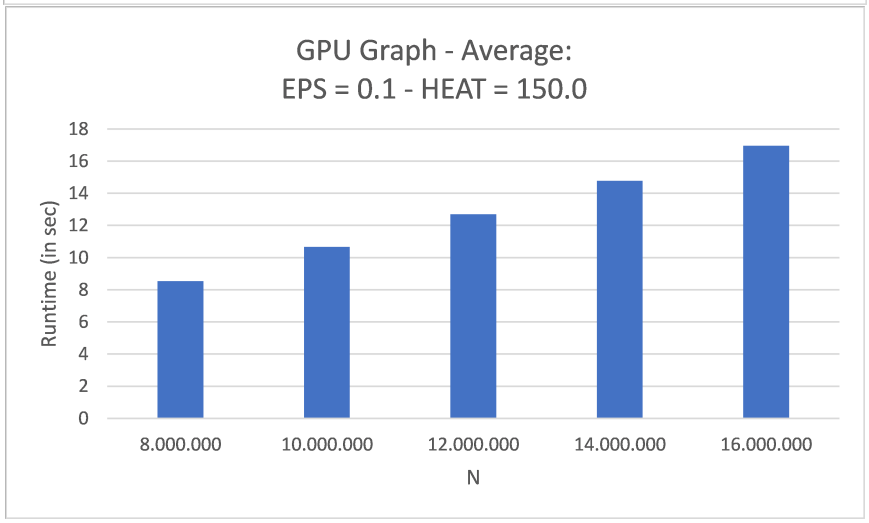
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	8.70687	8.56146	8.59486	8.52328	8.45856	8.49574	8.49061
10,000,000	10.84703	10.52563	10.46351	10.57881	10.74552	10.66831	10.62568
12,000,000	12.89035	12.89041	12.66707	12.65072	12.54356	12.58927	12.80527
14,000,000	14.95074	14.74885	14.79991	14.68003	14.81199	14.75147	14.71965
16,000,000	17.07796	16.95963	16.84905	17.16381	16.93302	16.92039	17.00031

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	8.52521	8.47977	8.55064	8.538700	8.70687	8.45856
10,000,000	10.61728	10.88499	10.69458	10.665134	10.88499	10.46351
12,000,000	12.64937	12.63867	12.60418	12.692887	12.89041	12.54356
14,000,000	14.78437	14.64007	14.87022	14.775730	14.95074	14.64007
16,000,000	16.76427	17.08674	16.73675	16.949193	17.16381	16.73675

The graphs below will display the results just obtained:





Results of Stefan Popa: tested GPU version

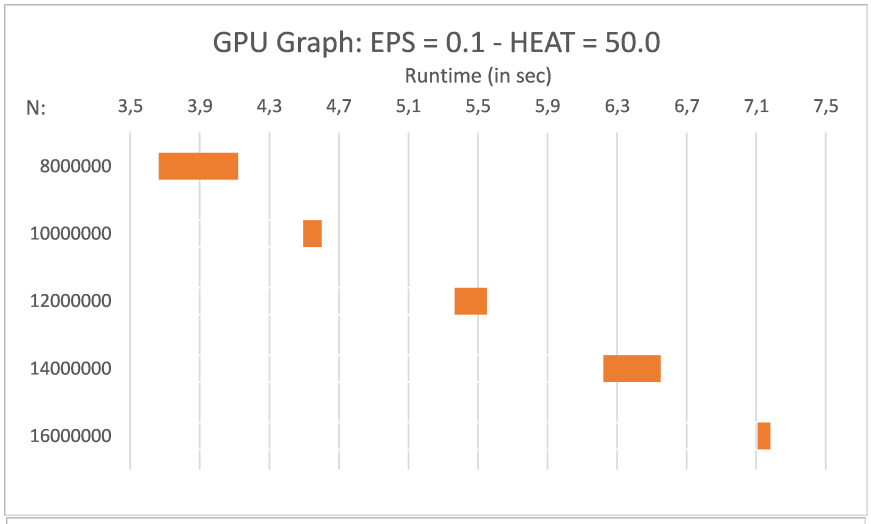
Before being able to measure the runtimes, a good work group size for the GPU had to be chosen. After running a couple of tests with different work group sizes, we were able to conclude that 32 gave the fastest runtimes. Therefore, this value was used for testing.

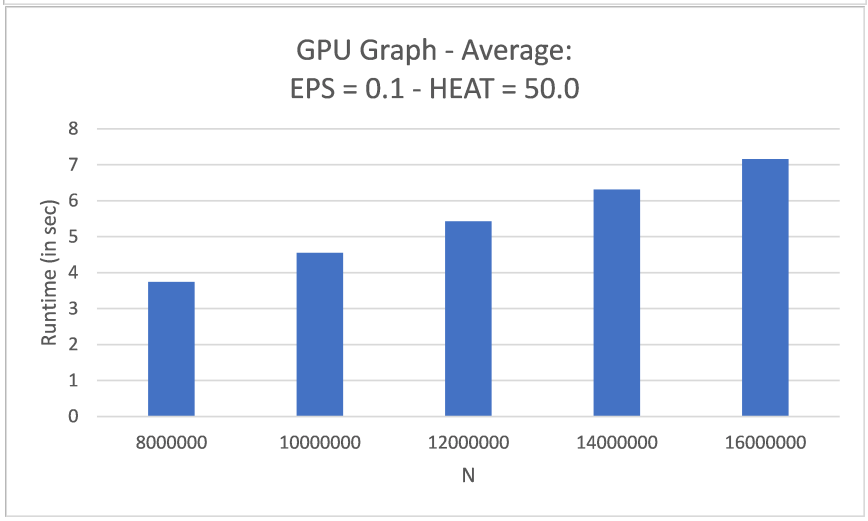
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 50.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	4.12029	3.70057	3.70824	3.68738	3.66458	3.69011	3.73791
10,000,000	4.56903	4.56482	4.56463	4.52394	4.60183	4.53080	4.53527
12,000,000	5.41405	5.41019	5.55132	5.46258	5.37710	5.45155	5.41942
14,000,000	6.22125	6.33330	6.30045	6.26399	6.27038	6.55159	6.23263
16,000,000	7.15117	7.16955	7.18104	7.17157	7.16184	7.10742	7.16761

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	3.67465	3.68152	3.75908	3.742433	4.12029	3.66458
10,000,000	4.55575	4.49475	4.54163	4.548245	4.60183	4.49475
12,000,000	5.41888	5.36492	5.38850	5.425851	5.55132	5.36492
14,000,000	6.29447	6.34305	6.27370	6.308481	6.55159	6.22125
16,000,000	7.13930	7.17610	7.17484	7.160044	7.18104	7.10742

The graphs below will display the results just obtained:



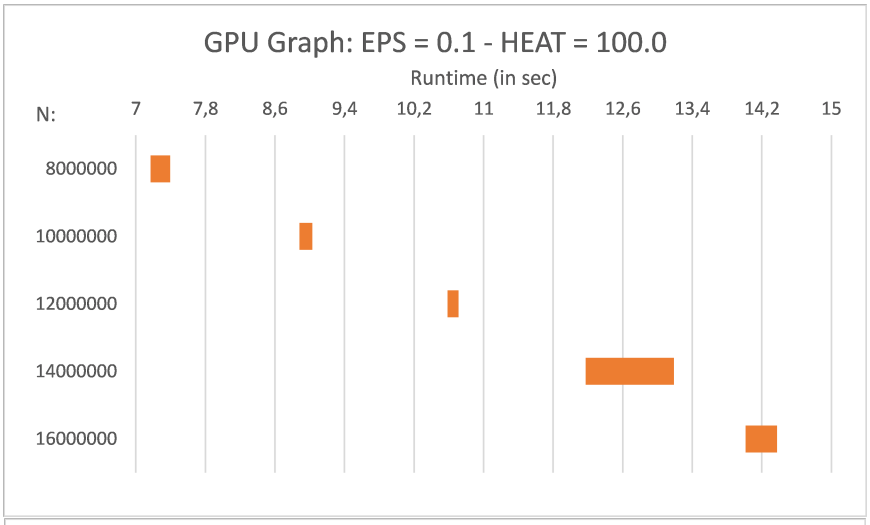


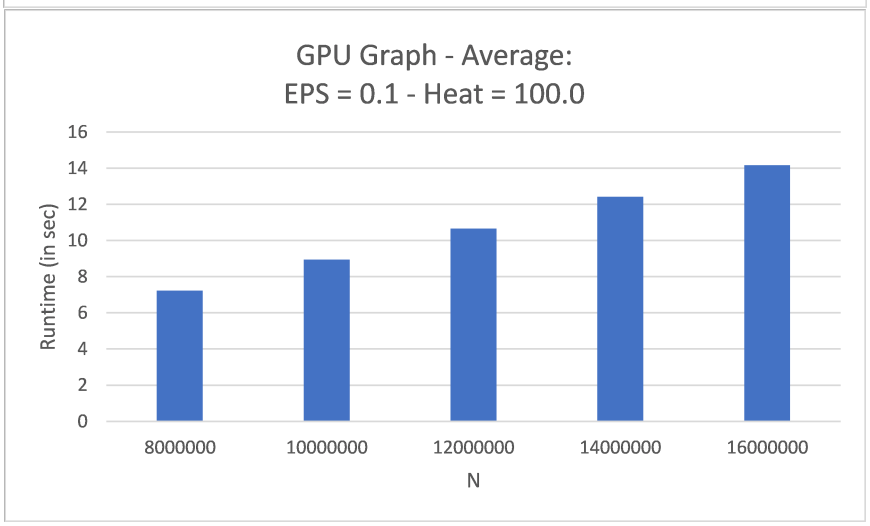
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 100.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	7.39682	7.19915	7.19496	7.17024	7.30391	7.23976	7.18004
10,000,000	8.91883	8.92797	8.98724	8.95295	8.88105	9.03120	8.88946
12,000,000	10.71174	10.69544	10.70103	10.62526	10.63083	10.58402	10.67648
14,000,000	12.32794	12.42428	12.30082	12.17271	12.39319	12.41167	12.37250
16,000,000	14.16790	14.14625	14.37508	14.35025	14.06621	14.12019	14.28729

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	7.19933	7.22573	7.22559	7.233553	7.39682	7.17024
10,000,000	8.95042	8.92119	8.92534	8.938565	9.03120	8.88105
12,000,000	10.64777	10.63919	10.63980	10.655156	10.71174	10.58402
14,000,000	13.18814	12.28466	12.36000	12.423591	13.18814	12.17271
16,000,000	14.01434	14.06359	14.06858	14.165968	14.37508	14.01434

The graphs below will display the results just obtained:



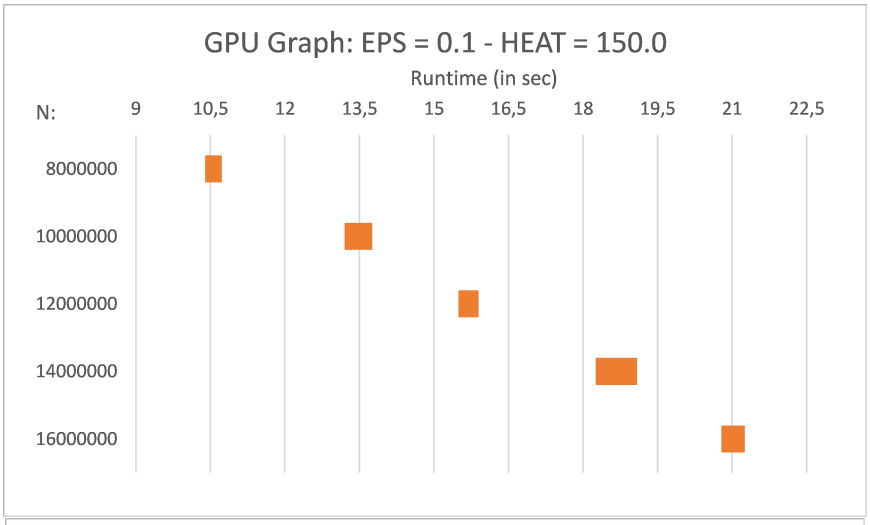


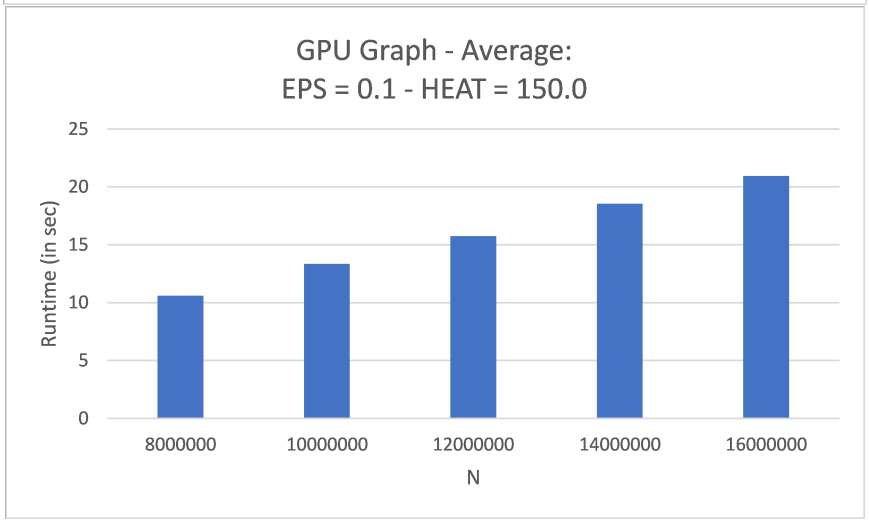
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	10.72829	10.55338	10.62351	10.55567	10.72072	10.62830	10.58581
10,000,000	13.23436	13.75822	13.19810	13.37476	13.30603	13.42020	13.20112
12,000,000	15.75912	15.75920	15.49429	15.86638	15.69308	15.69102	15.75186
14,000,000	18.32535	19.09085	18.47756	18.64582	18.36491	18.56390	18.87921
16,000,000	20.92718	20.97169	20.93457	20.95922	20.79800	21.25431	20.82753

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	10.58997	10.47045	10.39581	10.585191	10.72829	10.39581
10,000,000	13.37476	13.28699	13.40228	13.355682	13.75822	13.19810
12,000,000	15.72374	15.89946	15.72393	15.736208	15.89946	15.49429
14,000,000	18.25233	18.57564	18.25233	18.542790	19.09085	18.25233
16,000,000	20.94242	20.78313	20.92137	20.931942	21.25431	20.78313

The graphs below will display the results just obtained:





Task 3: Tune for CPU performance

Implementation process

Since the versions for the GPU did not get good results on our CPU, our first attempt to optimize the code for the CPU, was by splitting the global array into chunks. We did this in order to keep the thread space smaller on the CPU. We decreased this size with the following code:

```
global[0] = n/10;
```

We divided the global size with 10: this seemed like a logical choice for us, since we can always divide a million by 10. After tweaking a little with this value, the runtime was not affected a lot, so we decided to leave it at 10. Next, we adapted the kernel code in such a way that each work-item now relaxes ten elements instead of only one. We also decided that the first work-item should set the last element to its previous value such that this index will definitely not be relaxed. This way, we gave each work item more work to do, such that runtimes on the CPU could be improved. The resulting kernel code looks like the following:

```
const char *KernelSource =                                          "\n"
  "__kernel void relax(                                              \n"
  "    __global double* in,                                          \n"
  "    __global double* out,                                         \n"
  "    const unsigned int count)                                     \n"
  "{                                                                 \n"
  "    int i = get_global_id(0);                                     \n"
  "    int n = get_global_size(0);                                   \n"
  "    if(i == 0) {                                                  \n"
  "       for(int j = i*10 + 1; j <= i*10 + 9; j++)                  \n"
  "          out[j] = 0.25*in[j-1] + 0.5*in[j] + 0.25*in[j+1];       \n"
  "       out[n*10 - 1] = in[n*10 - 1];                              \n"
  "    } else if (i < n - 1) {                                       \n"
  "       for(int j = i*10 ; j <= i*10 + 9; j++)                     \n"
  "          out[j] = 0.25*in[j-1] + 0.5*in[j] + 0.25*in[j+1];       \n"
  "    }                                                             \n"
  "}                                                                 \n"
  "\n";
```

This attempt can be seen in the folder `Attempt_1_CPU`. This version was faster than the original function without any improvements.

After this, we wanted to try to include the `isStable()` functionality back in the kernel, like we did in the optimized version for the GPU. Again, the value for `stable[0]` is set to `true` by the work-item that has as global identifier 0. Then, we let each work-item check whether some index is not yet stable, and if so, we set the value of `stable[0]` to `false`. The kernel code looks like the following:

```
const char *KernelSource =                                          "\n"
  "__kernel void relax(                                              \n"
  "    __global double* in,                                          \n"
  "    __global double* out,                                         \n"
  "    __global bool* stable,                                        \n"
  "    const double eps,                                             \n"
  "    const unsigned int count)                                     \n"
  "{                                                                 \n"
  "    int i = get_global_id(0);                                     \n"
  "    int n = get_global_size(0);                                   \n"
  "    if(i == 0) {                                                  \n"
  "       stable[0] = true;                                          \n"
  "       for(int j = i*10 + 1; j <= i*10 + 9; j++) {                \n"
  "          out[j] = 0.25*in[j-1] + 0.5*in[j] + 0.25*in[j+1];       \n"
  "          if (fabs(in[j] - out[j]) > eps)                         \n"
  "             stable[0] = false;                                   \n"
  "       }                                                          \n"
  "       out[n*10 - 1] = in[n*10 - 1];                              \n"
  "    } else if (i < n - 1) {                                       \n"
  "       for(int j = i*10 ; j <= i*10 + 9; j++) {                   \n"
  "          out[j] = 0.25*in[j-1] + 0.5*in[j] + 0.25*in[j+1];       \n"
  "          if (fabs(in[j] - out[j]) > eps)                         \n"
  "             stable[0] = false;                                   \n"
  "       }                                                          \n"
  "    }                                                             \n"
  "}                                                                 \n"
  "\n";
```

As in Task 2: Tune for GPU performance, to reflect this change, we removed the function `isStable()` and instead we checked the value of `stable[0]` in the following way:

```
do {
   ...
} while(!stable[0]);
```

However, this version that can be seen in `Attempt_2_CPU`, did not improve our runtimes, so we ran our tests with the version that we described above, without the `isStable()` functionality in the kernel source.

Note here that we first decided for each device separately which work group size is most suitable for testing. As said in Task 3: Tune for CPU performance, note that we also tested the versions described in 'Task 2: Tune for GPU performance' on the CPU. These did not have better results on the CPU.

In our `Task_3` directory you will find the different attempts as described above.
The final OpenCL version for task 3 can be found in the file `Attempt_1_CPU`.

Results of Laura Kolijn: tested CPU version

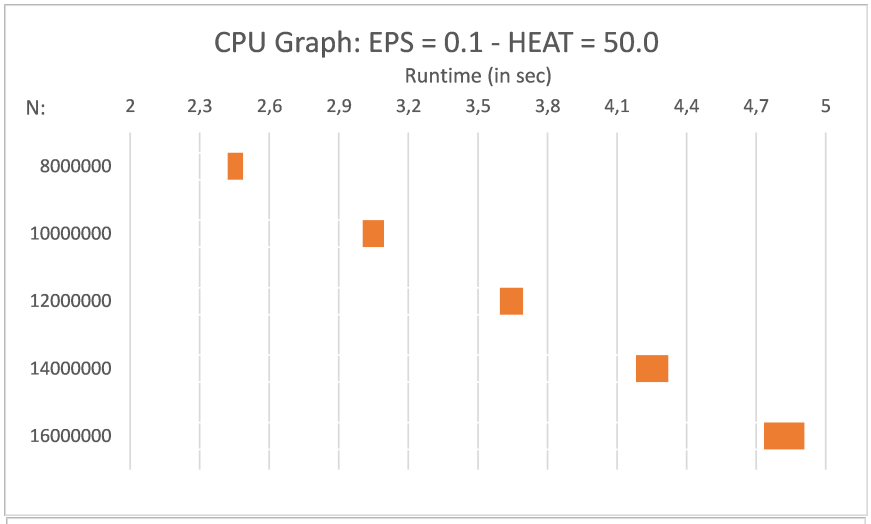
Before being able to measure the runtimes, a good work group size for the CPU had to be chosen. After running a couple of tests with different work group sizes, we were able to conclude that 64 gave the fastest runtimes. Therefore, this value was used for testing.

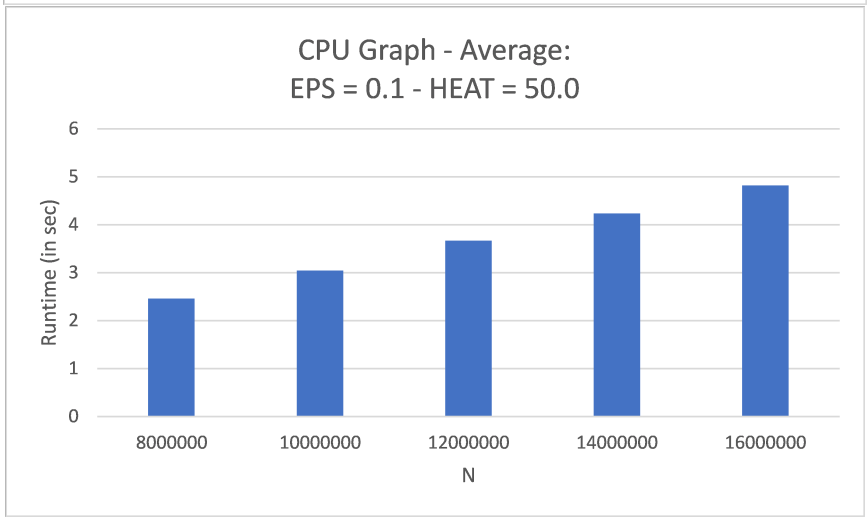
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 50.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	2.47515	2.42261	2.47616	2.47491	2.47619	2.42121	2.47736
10,000,000	3.05117	3.02988	3.00724	3.09500	3.00344	3.01257	3.05904
12,000,000	3.66877	3.66481	3.68494	3.65280	3.66988	3.69440	3.59510
14,000,000	4.28636	4.19815	4.19038	4.21206	4.20567	4.26564	4.18194
16,000,000	4.90811	4.86057	4.78519	4.82125	4.75763	4.90392	4.84444

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	2.42585	2.42912	2.48737	2.456593	2.48737	2.42121
10,000,000	3.06303	3.06329	3.05298	3.043764	3.09500	3.00344
12,000,000	3.68143	3.68349	3.64532	3.664094	3.69440	3.59510
14,000,000	4.28591	4.18557	4.32149	4.233317	4.32149	4.18194
16,000,000	4.82826	4.73333	4.75402	4.819672	4.90811	4.73333

The graphs below will display the results just obtained:



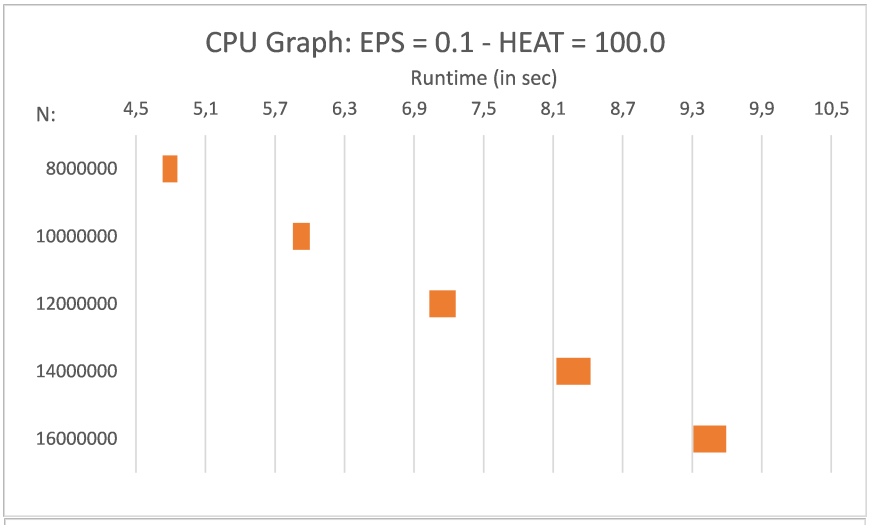


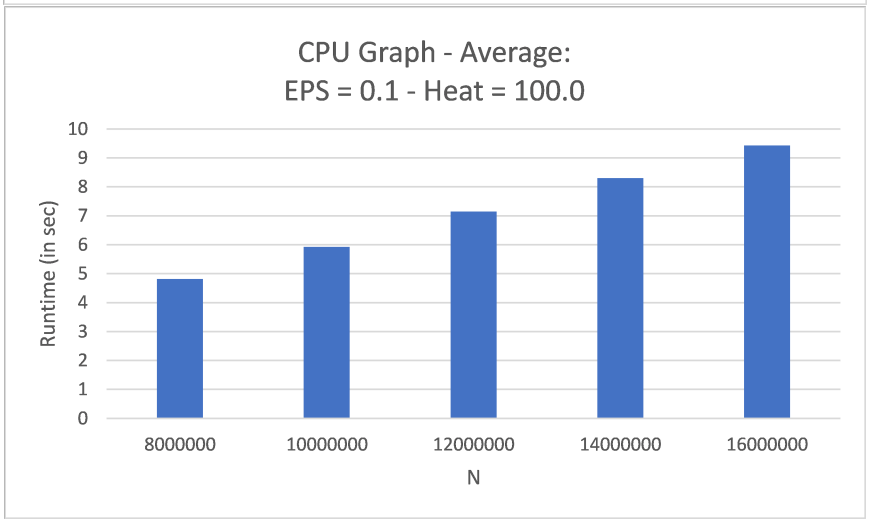
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 100.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	4.84277	4.85705	4.85861	4.80014	4.85066	4.82476	4.81760
10,000,000	5.98947	5.85578	5.91474	6.00136	5.89571	5.86714	5.91443
12,000,000	7.15668	7.06482	7.25847	7.17883	7.15418	7.15418	7.03125
14,000,000	8.35474	8.42313	8.23246	8.32984	8.20048	8.33808	8.35237
16,000,000	9.36220	9.59299	9.31057	9.50121	9.31057	9.50121	9.39394

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	4.72922	4.78078	4.81548	4.817707	4.85861	4.72922
10,000,000	5.91531	5.85455	5.99452	5.920301	6.00136	5.85455
12,000,000	7.07643	7.23322	7.18694	7.149500	7.25847	7.03125
14,000,000	8.36755	8.29745	8.12826	8.302436	8.42313	8.12826
16,000,000	9.34440	9.56158	9.36573	9.424440	9.59299	9.31057

The graphs below will display the results just obtained:



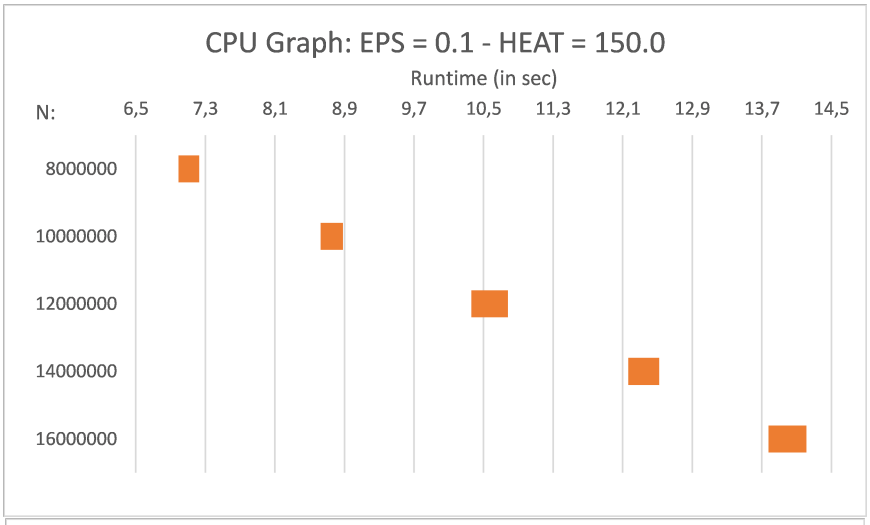


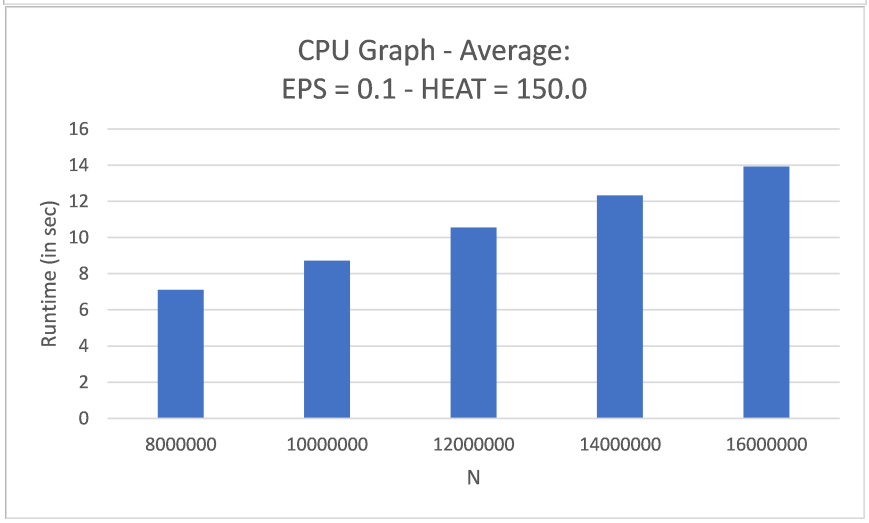
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	7.11670	7.23082	7.20950	7.14095	7.08685	7.08794	7.00082
10,000,000	8.83089	8.68269	8.86736	8.65134	8.64829	8.62554	8.66045
12,000,000	10.37501	10.36175	10.64012	10.36010	10.57817	10.47141	10.77976
14,000,000	12.34979	12.26983	12.26725	12.36324	12.52005	12.16528	12.37349
16,000,000	13.78249	13.79764	13.77740	13.80181	13.98406	14.21303	13.87164

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	7.05036	7.20321	6.99084	7.111799	7.23082	6.99084
10,000,000	8.66687	8.88120	8.65921	8.717384	8.88120	8.62554
12,000,000	10.64544	10.67898	10.63213	10.552287	10.77976	10.36010
14,000,000	12.21444	12.45625	12.26419	12.324381	12.52005	12.16528
16,000,000	13.91010	14.01331	14.07815	13.922963	14.21303	13.77740

The graphs below will display the results just obtained:





Results of Denise Verbakel: tested CPU version

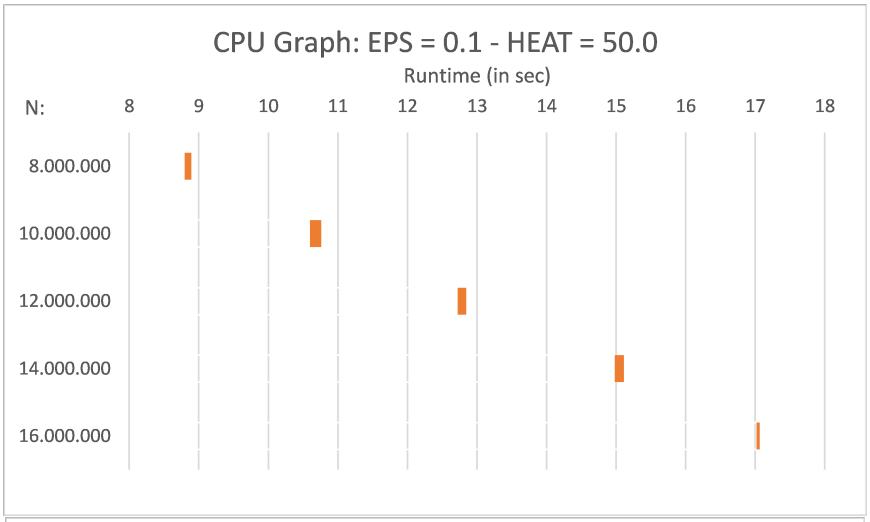
Before being able to measure the runtimes, a good work group size for the CPU had to be chosen. After running a couple of tests with different work group sizes, we were able to conclude that 32 gave the fastest runtimes. Therefore, this value was used for testing.

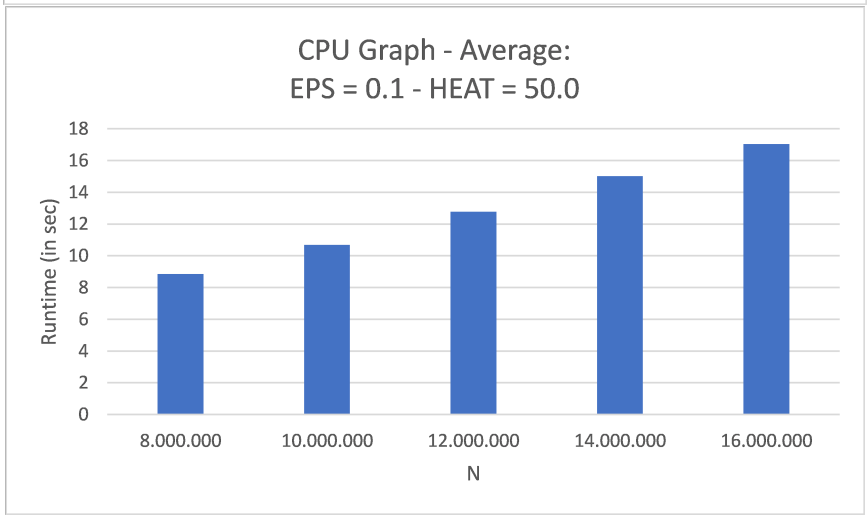
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 50.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	8.89229	8.83248	8.87329	8.81737	8.86102	8.81710	8.82161
10,000,000	10.66465	10.61678	10.60012	10.67428	10.64408	10.70835	10.71229
12,000,000	12.76660	12.73741	12.74770	12.76449	12.74779	12.84550	12.72372
14,000,000	14.98260	15.01235	15.02969	15.00287	14.98083	15.11383	15.01558
16,000,000	17.04505	17.01907	17.06531	17.03698	17.06338	17.04414	17.04357

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	8.86883	8.79891	8.89476	8.847766	8.89476	8.79891
10,000,000	10.75455	10.63318	10.75966	10.676794	10.75966	10.60012
12,000,000	12.81263	12.72131	12.80063	12.765748	12.84550	12.72131
14,000,000	14.98195	15.01342	15.01690	15.015002	15.11383	14.98083
16,000,000	17.02292	17.03127	17.02852	17.040021	17.06531	17.01907

The graphs below will display the results just obtained:



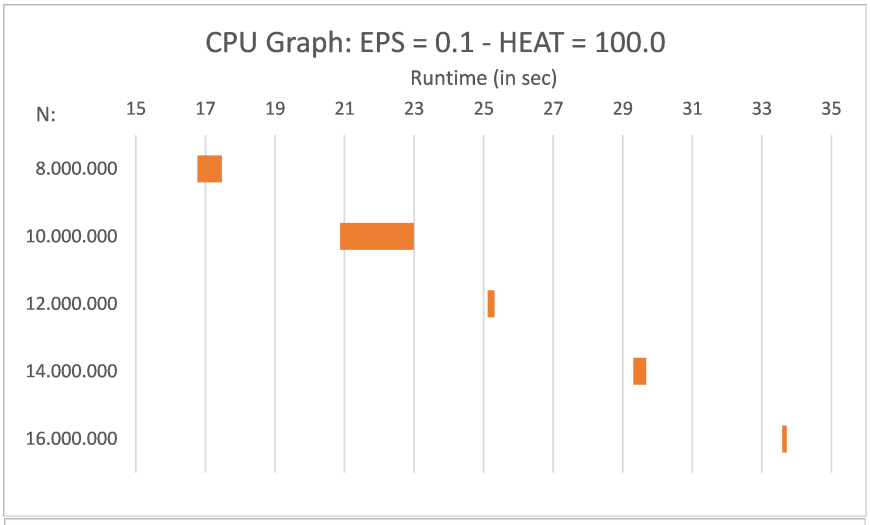


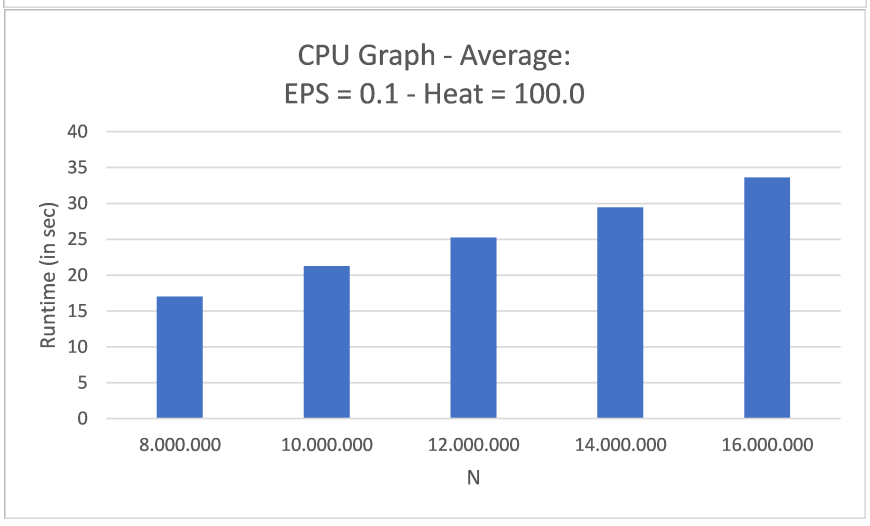
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 100.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	17.05699	16.90108	17.04347	17.47369	17.08291	16.89896	17.10863
10,000,000	21.12244	21.05942	21.17014	20.87643	21.09094	21.03015	21.22281
12,000,000	25.31787	25.15197	25.11458	25.30632	25.25469	25.19411	25.26885
14,000,000	29.45606	29.47795	29.41860	29.52503	29.36904	29.52113	29.43273
16,000,000	33.58823	33.59878	33.58629	33.63838	33.71527	33.66268	33.58629

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	16.77374	17.02992	16.92777	17.029716	17.47369	16.77374
10,000,000	22.99524	21.08360	20.92123	21.257240	22.99524	20.87643
12,000,000	25.17880	25.30950	25.22200	25.231869	25.31787	25.11458
14,000,000	29.50520	29.30646	29.67623	29.468843	29.67623	29.30646
16,000,000	33.58348	33.59522	33.58617	33.614079	33.71527	33.58348

The graphs below will display the results just obtained:



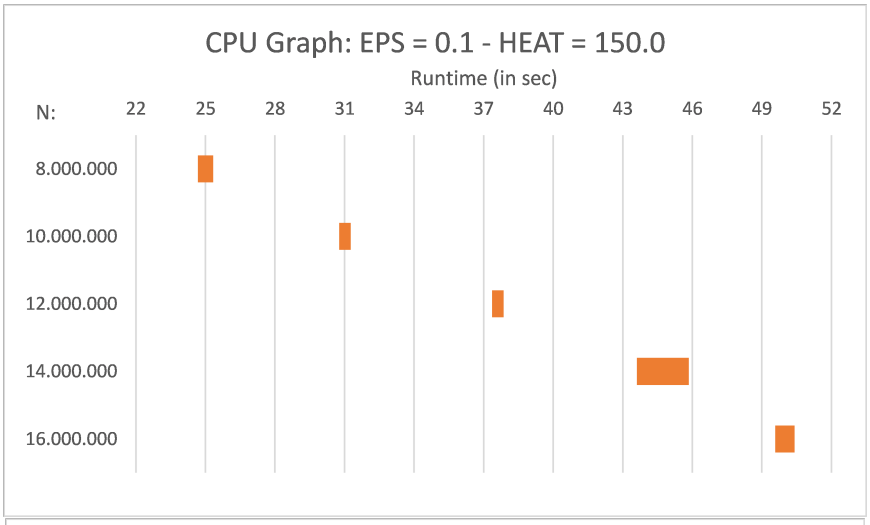


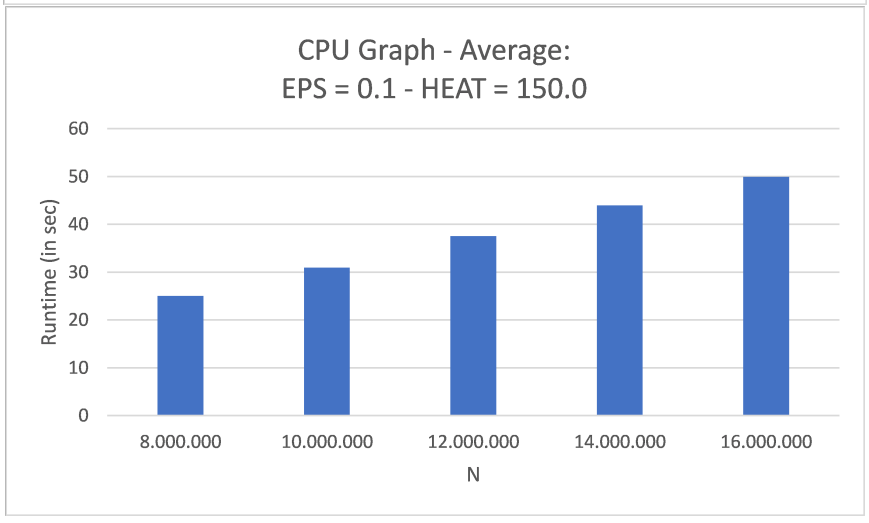
The following results (in seconds) are obtained with the parameters `EPS = 0.1` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
8,000,000	25.33162	24.89952	25.20202	24.88427	25.13187	24.86139	25.04519
10,000,000	31.07962	30.81660	31.17677	30.78233	30.90089	30.76753	31.06371
12,000,000	37.86323	37.36085	37.49753	37.60579	37.48193	37.45865	37.47032
14,000,000	43.69835	43.95087	43.61351	43.87661	43.69557	43.83570	43.71202
16,000,000	49.83467	49.57802	50.18457	49.74983	49.81592	49.87563	49.73543

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
8,000,000	24.67973	25.00749	24.98604	25.002914	25.33162	24.67973
10,000,000	30.76459	31.26918	30.95090	30.957212	31.26918	30.76459
12,000,000	37.45590	37.70842	37.39204	37.529466	37.86323	37.36085
14,000,000	43.91880	43.65448	45.85206	43.980797	45.85206	43.61351
16,000,000	50.41451	49.87080	50.02817	49.908755	50.41451	49.57802

The graphs below will display the results just obtained:





Task 4: Performance

Absolute performance measured in FLOPS

In order to start our discussion on the performance, we first computed the absolute performance we achieved on the two systems (CPU and GPU). For that purpose, we use FLOPS (floating point operations per second) as measure and rewrite these results into gigaFLOPS. We computed these FLOPS as follows:

$$\frac{n \cdot 5 \cdot \text{nr\_iterations}}{\text{avg\_runtime}}$$

Below, we computed every FLOPS measurement for each unique combination of `EPS`, `HEAT` and `N`. Here, nr_iterations is the number of iterations printed by our program and avg_runtime is the average runtime that we obtained in Task 2: Tune for GPU performance and Task 3: Tune for CPU performance.

FLOPS for `EPS = 0.01`, `HEAT = 50.0` and `N = 8,000,000`:

- Onno (GPU): $\frac{8,000,000 \cdot 5 \cdot 121}{2.98901} = \frac{4,840,000,000}{2.98901} \approx 1.6 \times 10^9$ FLOPS (= 1.6 gigaFLOPS)
- Stefan (GPU): $\frac{8,000,000 \cdot 5 \cdot 121}{3.742433} = \frac{4,840,000,000}{3.742433} \approx 1.3 \times 10^9$ FLOPS (= 1.3 gigaFLOPS)
- Laura (CPU): $\frac{8,000,000 \cdot 5 \cdot 121}{2.456593} = \frac{4,840,000,000}{2.456593} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{8,000,000 \cdot 5 \cdot 121}{8.847766} = \frac{4,840,000,000}{8.847766} \approx 5.5 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 50.0` and `N = 10,000,000`:

- Onno (GPU): $\frac{10,000,000 \cdot 5 \cdot 121}{3.666327} = \frac{6,050,000,000}{3.666327} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{10,000,000 \cdot 5 \cdot 121}{4.548245} = \frac{6,050,000,000}{4.548245} \approx 1.3 \times 10^9$ FLOPS (= 1.3 gigaFLOPS)
- Laura (CPU): $\frac{10,000,000 \cdot 5 \cdot 121}{3.043764} = \frac{6,050,000,000}{3.043764} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{10,000,000 \cdot 5 \cdot 121}{10.676794} = \frac{6,050,000,000}{10.676794} \approx 5.7 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 50.0` and `N = 12,000,000`:

- Onno (GPU): $\frac{12,000,000 \cdot 5 \cdot 121}{4.39199} = \frac{7,260,000,000}{4.39199} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{12,000,000 \cdot 5 \cdot 121}{5.425851} = \frac{7,260,000,000}{5.425851} \approx 1.3 \times 10^9$ FLOPS (= 1.3 gigaFLOPS)
- Laura (CPU): $\frac{12,000,000 \cdot 5 \cdot 121}{3.664094} = \frac{7,260,000,000}{3.664094} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{12,000,000 \cdot 5 \cdot 121}{12.765748} = \frac{7,260,000,000}{12.765748} \approx 5.7 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 50.0` and `N = 14,000,000`:

- Onno (GPU): $\frac{14,000,000 \cdot 5 \cdot 121}{5.074846} = \frac{8,470,000,000}{5.074846} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{14,000,000 \cdot 5 \cdot 121}{6.308481} = \frac{8,470,000,000}{6.308481} \approx 1.3 \times 10^9$ FLOPS (= 1.3 gigaFLOPS)
- Laura (CPU): $\frac{14,000,000 \cdot 5 \cdot 121}{4.233317} = \frac{8,470,000,000}{4.233317} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{14,000,000 \cdot 5 \cdot 121}{15.015002} = \frac{8,470,000,000}{15.015002} \approx 5.6 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 50.0` and `N = 16,000,000`:

- Onno (GPU): $\frac{16,000,000 \cdot 5 \cdot 121}{5.795959} = \frac{9,680,000,000}{5.795959} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{16,000,000 \cdot 5 \cdot 121}{7.160044} = \frac{9,680,000,000}{7.160044} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{16,000,000 \cdot 5 \cdot 121}{4.819672} = \frac{9,680,000,000}{4.819672} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{16,000,000 \cdot 5 \cdot 121}{17.040021} = \frac{9,680,000,000}{17.040021} \approx 5.7 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 100.0` and `N = 8,000,000`:

- Onno (GPU): $\frac{8,000,000 \cdot 5 \cdot 243}{5.826008} = \frac{9,720,000,000}{5.826008} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{8,000,000 \cdot 5 \cdot 243}{7.233553} = \frac{9,720,000,000}{7.233553} \approx 1.3 \times 10^9$ FLOPS (= 1.3 gigaFLOPS)
- Laura (CPU): $\frac{8,000,000 \cdot 5 \cdot 243}{4.817707} = \frac{9,720,000,000}{4.817707} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{8,000,000 \cdot 5 \cdot 243}{17.029716} = \frac{9,720,000,000}{17.029716} \approx 5.7 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 100.0` and `N = 10,000,000`:

- Onno (GPU): $\frac{10,000,000 \cdot 5 \cdot 243}{7.18577} = \frac{12,150,000,000}{7.18577} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{10,000,000 \cdot 5 \cdot 243}{8.938565} = \frac{12,150,000,000}{8.938565} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{10,000,000 \cdot 5 \cdot 243}{5.920301} = \frac{12,150,000,000}{5.920301} \approx 2.1 \times 10^9$ FLOPS (= 2.1 gigaFLOPS)
- Denise (CPU): $\frac{10,000,000 \cdot 5 \cdot 243}{21.257240} = \frac{12,150,000,000}{21.257240} \approx 5.7 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 100.0` and `N = 12,000,000`:

- Onno (GPU): $\frac{12,000,000 \cdot 5 \cdot 243}{8.591067} = \frac{14,580,000,000}{8.591067} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{12,000,000 \cdot 5 \cdot 243}{10.655156} = \frac{14,580,000,000}{10.655156} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{12,000,000 \cdot 5 \cdot 243}{7.149500} = \frac{14,580,000,000}{7.149500} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{12,000,000 \cdot 5 \cdot 243}{25.231869} = \frac{14,580,000,000}{25.231869} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 100.0` and `N = 14,000,000`:

- Onno (GPU): $\frac{14,000,000 \cdot 5 \cdot 243}{9.952267} = \frac{17,010,000,000}{9.952267} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{14,000,000 \cdot 5 \cdot 243}{12.423591} = \frac{17,010,000,000}{12.423591} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{14,000,000 \cdot 5 \cdot 243}{8.302436} = \frac{17,010,000,000}{8.302436} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{14,000,000 \cdot 5 \cdot 243}{29.468843} = \frac{17,010,000,000}{29.468843} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 100.0` and `N = 16,000,000`:

- Onno (GPU): $\frac{16,000,000 \cdot 5 \cdot 243}{11.340739} = \frac{19,440,000,000}{11.340739} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{16,000,000 \cdot 5 \cdot 243}{14.165968} = \frac{19,440,000,000}{14.165968} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{16,000,000 \cdot 5 \cdot 243}{9.424440} = \frac{19,440,000,000}{9.424440} \approx 2.1 \times 10^9$ FLOPS (= 2.1 gigaFLOPS)
- Denise (CPU): $\frac{16,000,000 \cdot 5 \cdot 243}{33.614079} = \frac{19,440,000,000}{33.614079} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 150.0` and `N = 8,000,000`:

- Onno (GPU): $\frac{8,000,000 \cdot 5 \cdot 363}{8.538700} = \frac{14,520,000,000}{8.538700} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{8,000,000 \cdot 5 \cdot 363}{10.585191} = \frac{14,520,000,000}{10.585191} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{8,000,000 \cdot 5 \cdot 363}{7.111799} = \frac{14,520,000,000}{7.111799} \approx 2.0 \times 10^9$ FLOPS (= 2.0 gigaFLOPS)
- Denise (CPU): $\frac{8,000,000 \cdot 5 \cdot 363}{25.002914} = \frac{14,520,000,000}{25.002914} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 150.0` and `N = 10,000,000`:

- Onno (GPU): $\frac{10,000,000 \cdot 5 \cdot 363}{10.665134} = \frac{18,150,000,000}{10.665134} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{10,000,000 \cdot 5 \cdot 363}{13.355682} = \frac{18,150,000,000}{13.355682} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{10,000,000 \cdot 5 \cdot 363}{8.717384} = \frac{18,150,000,000}{8.717384} \approx 2.1 \times 10^9$ FLOPS (= 2.1 gigaFLOPS)
- Denise (CPU): $\frac{10,000,000 \cdot 5 \cdot 363}{30.957212} = \frac{18,150,000,000}{30.957212} \approx 5.9 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 150.0` and `N = 12,000,000`:

- Onno (GPU): $\frac{12,000,000 \cdot 5 \cdot 363}{12.692887} = \frac{21,780,000,000}{12.692887} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{12,000,000 \cdot 5 \cdot 363}{15.736208} = \frac{21,780,000,000}{15.736208} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{12,000,000 \cdot 5 \cdot 363}{10.552287} = \frac{21,780,000,000}{10.552287} \approx 2.1 \times 10^9$ FLOPS (= 2.1 gigaFLOPS)
- Denise (CPU): $\frac{12,000,000 \cdot 5 \cdot 363}{37.529466} = \frac{21,780,000,000}{37.529466} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 150.0` and `N = 14,000,000`:

- Onno (GPU): $\frac{14,000,000 \cdot 5 \cdot 363}{14.77573} = \frac{25,410,000,000}{14.77573} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{14,000,000 \cdot 5 \cdot 363}{18.542790} = \frac{25,410,000,000}{18.542790} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{14,000,000 \cdot 5 \cdot 363}{12.324381} = \frac{25,410,000,000}{12.324381} \approx 2.1 \times 10^9$ FLOPS (= 2.1 gigaFLOPS)
- Denise (CPU): $\frac{14,000,000 \cdot 5 \cdot 363}{43.980797} = \frac{25,410,000,000}{43.980797} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

FLOPS for `EPS = 0.01`, `HEAT = 150.0` and `N = 16,000,000`:

- Onno (GPU): $\frac{16,000,000 \cdot 5 \cdot 363}{16.949193} = \frac{29,040,000,000}{16.949193} \approx 1.7 \times 10^9$ FLOPS (= 1.7 gigaFLOPS)
- Stefan (GPU): $\frac{16,000,000 \cdot 5 \cdot 363}{20.931942} = \frac{29,040,000,000}{20.931942} \approx 1.4 \times 10^9$ FLOPS (= 1.4 gigaFLOPS)
- Laura (CPU): $\frac{16,000,000 \cdot 5 \cdot 363}{13.922963} = \frac{29,040,000,000}{13.922963} \approx 2.1 \times 10^9$ FLOPS (= 2.1 gigaFLOPS)
- Denise (CPU): $\frac{16,000,000 \cdot 5 \cdot 363}{49.908755} = \frac{29,040,000,000}{49.908755} \approx 5.8 \times 10^8$ FLOPS (= 0.6 gigaFLOPS)

As can be seen above in the gigaFLOPS computation we obtained the following results:

- Onno (GPU): about 1.6 or 1.7 gigaFLOPS for every test.
- Stefan (GPU): about 1.3 or 1.4 gigaFLOPS for every test.
- Laura (CPU): about 2.0 or 2.1 gigaFLOPS for every test.
- Denise (CPU): about 0.6 gigaFLOPS for every test.

From these results per person we can draw the conclusion that for everyone within our group the floating point operations per second are quite constant. There is not much difference between all combinations of different `HEAT` and `N` values.

Discussion

First, we want to explain why we chose the `HEAT` and `N` values for testing. We chose these values because they result in doable runtimes for both the GPU and CPU, while also offering a good overview of different runtime results.

As can be seen from the implementation process, a GPU performs best when the code is highly parallelized, whereas a CPU performs better when the workload of a single thread is slightly higher and the thread space is smaller. From this we can also conclude that parallel computing (with small programs) is not always a good solution. This can especially be seen in the results of Denise on the CPU. We noticed that our implementation was not fully parallelizable and that there will not and does not really get significantly faster.

Finally, we will answer the question:

> *Do the `N` and `HEAT` values affect the absolute performance on either system?*

As the section 'Absolute performance measured in FLOPS' showed, we can answer this question with a simple and short answer: no. All four testing devices got around the same values for FLOPS every time the tests were run, no matter which `HEAT` and `N` values were used. The only noticeable difference in increasing the `HEAT` and `N` values was that the runtime every time gradually increased, matching with the conclusion that the FLOPS stay around the same value.

Further directions of investigation

First of all, a possible further direction of investigation is to figure out how a 2D version of our OpenCL implementation would affect the runtimes. Next to this, we could also try to be more verbose with our OpenCL calls. By avoiding abstraction, it might be easier to make our program faster. Finally, as in the previous assignment, we could also investigate how the runtime differs in the case where the assumption holds that the heat is always placed in the beginning.

Task 5: Team

Division of work

In this task, we will describe on how we divided up the work.

As in our previous assignment, our main idea was to work together (so all four of us) as much as we could since we decided that nobody should get a much higher or lower grade: we are in this together and decided to make this assignment as a team. Since with this assignment we could not really split the tasks, we started working together on Task 1. We optimized this version before starting to work on the optimization of the GPU and/or CPU. After having finished this optimized version, we went on by implementing the improved GPU version together as described in Task 2: Tune for GPU performance. Next, we implemented the versions for the CPU as described in Task 3: Tune for CPU performance.

After implementing everything together, we split up in the following groups of two:

- GPU testing group: Onno de Gouw and Stefan Popa
- CPU testing group: Laura Kolijn and Denise Verbakel

After running different tests and finding the FLOPS, we again grouped together and wrote this report.

Also this time, we all believe we can say that we solved this whole assignment as a team. Every member of the team was very involved with the project and has thought about different ideas and possibilities. We agree that we all roughly put in the same amount of effort and that the communication went well between us.

Team name on server

Our team name on the test server was 'Sole Survivors'. We created a logo for it and we like to include it in our report: