# High Performance Computing for Science and Engineering II.

## Exercise Set 5

Florian Mahlknecht

2020-05-11

# 1 P1 Benchmarking

## 1.1 Benchmark A: empty kernels

On Piz Daint, the execution of benchmark A yields the following results:

```
synchronize=0 blocks=    1  threads/block=   1  iteration=2.1 us
synchronize=1 blocks=    1  threads/block=   1  iteration=5.9 us
synchronize=1 blocks=    1  threads/block=  32  iteration=5.8 us
synchronize=1 blocks=    1  threads/block=1024  iteration=6.1 us
synchronize=1 blocks=   32  threads/block=1024  iteration=6.2 us
synchronize=1 blocks= 1024  threads/block=  32  iteration=8.0 us
synchronize=1 blocks=32768  threads/block=   1  iteration=61.2 us
synchronize=1 blocks=32768  threads/block=  32  iteration=61.2 us
synchronize=1 blocks=32768  threads/block=1024  iteration=104.3 us
Empty OpenMP parallel region with 12 threads --> 1.9 us
```

**Synchronization cost** When synchronizing after *each* kernel call the device with the CPU, the required time *almost triples*, with respect to exploiting asynchronous call queuing on the GPU. This underlines the communication cost involved.

**Blocks vs Threads** The output unveils a non trivial behavior between threads and blocks. For instance, launching one block with 1024 takes 6.1 µs, and launching 32 blocks with 1024 threads each, takes just about the same amount of time with 6.1 µs. This is due to the architecture of the GPU, as long as enough blocks are physically available, there is (almost) no overhead in using them. The threads however, which are not per block used are wasted. We can see this with the inverse of 1024 blocks and only 32 blocks per thread, taking slightly longer 8.0 µs, even though the same number of parallel operations is executed.

The same phenomena is observed with 32768 blocks and 1 and 32 respective blocks per threads, taking the same amount of computation time.


fmahlknecht@student.ethz.ch
19-945-351

**OpenMP parallel region** The OpenMP parallel region creation is compared to the CUDA executions quite fast, this is perhaps due to the fact that we need to communicate with the GPU through PCI, which is principally slower than system memory operations (which are required to construct the respective thread context in the scheduler). So despite the fact that the operating system needs to take care of allocating new resources for the threads and scheduling them on the pyhiscal cores, it is faster than the communication with the GPU device.

## 1.2 Benchmark B: Memory

The complete output of the memory benchmark on Piz Daint is as follows:

```
./benchmarks_b
Using device with 1024 threads per block
upload K=      17 -->  0.02 GB/s
upload K=      65 -->  0.09 GB/s
upload K=     251 -->  0.32 GB/s
upload K=    1001 -->  1.10 GB/s
upload K=    2001 -->  1.87 GB/s
upload K=    5001 -->  4.21 GB/s
upload K=   10001 -->  6.18 GB/s
upload K=   25001 -->  8.99 GB/s
upload K=   50001 --> 10.40 GB/s
upload K=  100001 --> 11.36 GB/s
upload K=  250001 --> 11.87 GB/s
upload K=  500001 --> 12.17 GB/s
upload K= 1000001 --> 12.31 GB/s
upload K= 5000001 --> 12.45 GB/s
upload K=20000001 --> 12.47 GB/s
upload K=50000001 --> 12.47 GB/s
Case p[i]=i  -->  K=      17  [a=b_p]   0.06 GB/s  [a_p=b]   0.07 GB/s written
Case p[i]=i  -->  K=      65  [a=b_p]   0.25 GB/s  [a_p=b]   0.25 GB/s written
Case p[i]=i  -->  K=     251  [a=b_p]   0.94 GB/s  [a_p=b]   0.96 GB/s written
Case p[i]=i  -->  K=    1001  [a=b_p]   3.50 GB/s  [a_p=b]   3.76 GB/s written
Case p[i]=i  -->  K=    2001  [a=b_p]   6.95 GB/s  [a_p=b]   7.09 GB/s written
Case p[i]=i  -->  K=    5001  [a=b_p]  17.30 GB/s  [a_p=b]  17.55 GB/s written
Case p[i]=i  -->  K=   10001  [a=b_p]  34.21 GB/s  [a_p=b]  34.87 GB/s written
Case p[i]=i  -->  K=   25001  [a=b_p]  80.98 GB/s  [a_p=b]  86.06 GB/s written
Case p[i]=i  -->  K=   50001  [a=b_p] 146.37 GB/s  [a_p=b] 157.93 GB/s written
Case p[i]=i  -->  K=  100001  [a=b_p] 239.72 GB/s  [a_p=b] 246.60 GB/s written
Case p[i]=i  -->  K=  250001  [a=b_p] 180.56 GB/s  [a_p=b] 188.06 GB/s written
Case p[i]=i  -->  K=  500001  [a=b_p] 189.29 GB/s  [a_p=b] 192.12 GB/s written
Case p[i]=i  -->  K= 1000001  [a=b_p] 204.41 GB/s  [a_p=b] 205.07 GB/s written
Case p[i]=i  -->  K= 5000001  [a=b_p] 215.06 GB/s  [a_p=b] 215.71 GB/s written
Case p[i]=i  -->  K=20000001  [a=b_p] 217.59 GB/s  [a_p=b] 218.39 GB/s written
Case p[i]=i  -->  K=50000001  [a=b_p] 218.08 GB/s  [a_p=b] 218.98 GB/s written
Case p[i]=(2*i)%K  -->  K=      17  [a=b_p]   0.06 GB/s  [a_p=b]   0.06 GB/s written
Case p[i]=(2*i)%K  -->  K=      65  [a=b_p]   0.25 GB/s  [a_p=b]   0.25 GB/s written
Case p[i]=(2*i)%K  -->  K=     251  [a=b_p]   0.96 GB/s  [a_p=b]   0.96 GB/s written
Case p[i]=(2*i)%K  -->  K=    1001  [a=b_p]   3.25 GB/s  [a_p=b]   3.26 GB/s written
Case p[i]=(2*i)%K  -->  K=    2001  [a=b_p]   6.37 GB/s  [a_p=b]   6.36 GB/s written
Case p[i]=(2*i)%K  -->  K=    5001  [a=b_p]  15.01 GB/s  [a_p=b]  15.87 GB/s written
Case p[i]=(2*i)%K  -->  K=   10001  [a=b_p]  30.05 GB/s  [a_p=b]  30.08 GB/s written
Case p[i]=(2*i)%K  -->  K=   25001  [a=b_p]  70.25 GB/s  [a_p=b]  73.73 GB/s written
Case p[i]=(2*i)%K  -->  K=   50001  [a=b_p] 131.66 GB/s  [a_p=b] 125.53 GB/s written
Case p[i]=(2*i)%K  -->  K=  100001  [a=b_p] 214.88 GB/s  [a_p=b] 170.67 GB/s written
Case p[i]=(2*i)%K  -->  K=  250001  [a=b_p] 197.21 GB/s  [a_p=b] 172.61 GB/s written
Case p[i]=(2*i)%K  -->  K=  500001  [a=b_p] 138.75 GB/s  [a_p=b]  96.96 GB/s written
Case p[i]=(2*i)%K  -->  K= 1000001  [a=b_p] 148.48 GB/s  [a_p=b]  97.41 GB/s written
Case p[i]=(2*i)%K  -->  K= 5000001  [a=b_p] 156.09 GB/s  [a_p=b]  99.49 GB/s written
Case p[i]=(2*i)%K  -->  K=20000001  [a=b_p] 157.67 GB/s  [a_p=b]  99.88 GB/s written
Case p[i]=(2*i)%K  -->  K=50000001  [a=b_p] 157.98 GB/s  [a_p=b] 100.00 GB/s written
Case p[i]=(4*i)%K  -->  K=      17  [a=b_p]   0.06 GB/s  [a_p=b]   0.06 GB/s written
Case p[i]=(4*i)%K  -->  K=      65  [a=b_p]   0.25 GB/s  [a_p=b]   0.25 GB/s written
Case p[i]=(4*i)%K  -->  K=     251  [a=b_p]   0.96 GB/s  [a_p=b]   0.96 GB/s written
Case p[i]=(4*i)%K  -->  K=    1001  [a=b_p]   3.21 GB/s  [a_p=b]   2.76 GB/s written
Case p[i]=(4*i)%K  -->  K=    2001  [a=b_p]   5.95 GB/s  [a_p=b]   5.24 GB/s written
Case p[i]=(4*i)%K  -->  K=    5001  [a=b_p]  13.69 GB/s  [a_p=b]  12.92 GB/s written
Case p[i]=(4*i)%K  -->  K=   10001  [a=b_p]  26.85 GB/s  [a_p=b]  25.82 GB/s written
Case p[i]=(4*i)%K  -->  K=   25001  [a=b_p]  64.57 GB/s  [a_p=b]  62.87 GB/s written
Case p[i]=(4*i)%K  -->  K=   50001  [a=b_p] 118.28 GB/s  [a_p=b]  92.07 GB/s written
Case p[i]=(4*i)%K  -->  K=  100001  [a=b_p] 164.79 GB/s  [a_p=b] 113.70 GB/s written
Case p[i]=(4*i)%K  -->  K=  250001  [a=b_p] 176.19 GB/s  [a_p=b] 123.24 GB/s written
Case p[i]=(4*i)%K  -->  K=  500001  [a=b_p] 102.40 GB/s  [a_p=b]  68.90 GB/s written
Case p[i]=(4*i)%K  -->  K= 1000001  [a=b_p]  96.07 GB/s  [a_p=b]  57.76 GB/s written
Case p[i]=(4*i)%K  -->  K= 5000001  [a=b_p]  99.34 GB/s  [a_p=b]  58.09 GB/s written
Case p[i]=(4*i)%K  -->  K=20000001  [a=b_p]  99.97 GB/s  [a_p=b]  58.12 GB/s written
Case p[i]=(4*i)%K  -->  K=50000001  [a=b_p] 100.12 GB/s  [a_p=b]  58.12 GB/s written
Case p[i]=i, 32-shuffled  -->  K=      17  [a=b_p]   0.06 GB/s  [a_p=b]   0.06 GB/s written
Case p[i]=i, 32-shuffled  -->  K=      65  [a=b_p]   0.25 GB/s  [a_p=b]   0.25 GB/s written
Case p[i]=i, 32-shuffled  -->  K=     251  [a=b_p]   0.96 GB/s  [a_p=b]   0.96 GB/s written
Case p[i]=i, 32-shuffled  -->  K=    1001  [a=b_p]   3.50 GB/s  [a_p=b]   3.49 GB/s written
Case p[i]=i, 32-shuffled  -->  K=    2001  [a=b_p]   6.92 GB/s  [a_p=b]   6.90 GB/s written
Case p[i]=i, 32-shuffled  -->  K=    5001  [a=b_p]  17.25 GB/s  [a_p=b]  17.24 GB/s written
Case p[i]=i, 32-shuffled  -->  K=   10001  [a=b_p]  33.77 GB/s  [a_p=b]  32.77 GB/s written
Case p[i]=i, 32-shuffled  -->  K=   25001  [a=b_p]  80.47 GB/s  [a_p=b]  80.14 GB/s written
Case p[i]=i, 32-shuffled  -->  K=   50001  [a=b_p] 148.46 GB/s  [a_p=b] 155.18 GB/s written
Case p[i]=i, 32-shuffled  -->  K=  100001  [a=b_p] 240.36 GB/s  [a_p=b] 226.11 GB/s written
```


Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

```
Case p[i]=i, 32-shuffled  -->  K=  250001  [a=b_p] 179.06 GB/s  [a_p=b] 182.79 GB/s written
Case p[i]=i, 32-shuffled  -->  K=  500001  [a=b_p] 188.19 GB/s  [a_p=b] 190.83 GB/s written
Case p[i]=i, 32-shuffled  -->  K= 1000001  [a=b_p] 203.08 GB/s  [a_p=b] 204.13 GB/s written
Case p[i]=i, 32-shuffled  -->  K= 5000001  [a=b_p] 214.70 GB/s  [a_p=b] 215.46 GB/s written
Case p[i]=i, 32-shuffled  -->  K=20000001  [a=b_p] 217.35 GB/s  [a_p=b] 218.35 GB/s written
Case p[i]=i, 32-shuffled  -->  K=50000001  [a=b_p] 217.94 GB/s  [a_p=b] 218.91 GB/s written
Case fully shuffled  -->  K=      17  [a=b_p]   0.07 GB/s  [a_p=b]   0.06 GB/s written
Case fully shuffled  -->  K=      65  [a=b_p]   0.25 GB/s  [a_p=b]   0.25 GB/s written
Case fully shuffled  -->  K=     251  [a=b_p]   0.96 GB/s  [a_p=b]   0.95 GB/s written
Case fully shuffled  -->  K=    1001  [a=b_p]   3.24 GB/s  [a_p=b]   2.57 GB/s written
Case fully shuffled  -->  K=    2001  [a=b_p]   6.36 GB/s  [a_p=b]   4.92 GB/s written
Case fully shuffled  -->  K=    5001  [a=b_p]  14.70 GB/s  [a_p=b]  12.13 GB/s written
Case fully shuffled  -->  K=   10001  [a=b_p]  28.01 GB/s  [a_p=b]  24.20 GB/s written
Case fully shuffled  -->  K=   25001  [a=b_p]  65.28 GB/s  [a_p=b]  58.28 GB/s written
Case fully shuffled  -->  K=   50001  [a=b_p] 102.57 GB/s  [a_p=b]  80.69 GB/s written
Case fully shuffled  -->  K=  100001  [a=b_p] 135.31 GB/s  [a_p=b]  92.19 GB/s written
Case fully shuffled  -->  K=  250001  [a=b_p] 102.39 GB/s  [a_p=b]  98.23 GB/s written
Case fully shuffled  -->  K=  500001  [a=b_p]  78.63 GB/s  [a_p=b]  63.67 GB/s written
Case fully shuffled  -->  K= 1000001  [a=b_p]  59.16 GB/s  [a_p=b]  30.75 GB/s written
Case fully shuffled  -->  K= 5000001  [a=b_p]  37.28 GB/s  [a_p=b]  17.38 GB/s written
Case fully shuffled  -->  K=20000001  [a=b_p]  33.24 GB/s  [a_p=b]  15.82 GB/s written
Case fully shuffled  -->  K=50000001  [a=b_p]  32.56 GB/s  [a_p=b]  15.54 GB/s written
a+b  -->  K=      17  1x ->  0.0 GFLOP/s  100x ->   0.2 GFLOP/s
a+b  -->  K=      65  1x ->  0.0 GFLOP/s  100x ->   0.6 GFLOP/s
a+b  -->  K=     251  1x ->  0.1 GFLOP/s  100x ->   2.5 GFLOP/s
a+b  -->  K=    1001  1x ->  0.4 GFLOP/s  100x ->   3.2 GFLOP/s
a+b  -->  K=    2001  1x ->  0.9 GFLOP/s  100x ->   6.4 GFLOP/s
a+b  -->  K=    5001  1x ->  2.1 GFLOP/s  100x ->  15.8 GFLOP/s
a+b  -->  K=   10001  1x ->  4.0 GFLOP/s  100x ->  31.6 GFLOP/s
a+b  -->  K=   25001  1x ->  9.9 GFLOP/s  100x ->  78.6 GFLOP/s
a+b  -->  K=   50001  1x -> 19.2 GFLOP/s  100x -> 114.4 GFLOP/s
a+b  -->  K=  100001  1x -> 29.1 GFLOP/s  100x -> 115.0 GFLOP/s
a+b  -->  K=  250001  1x -> 45.6 GFLOP/s  100x -> 101.5 GFLOP/s
a+b  -->  K=  500001  1x -> 20.6 GFLOP/s  100x -> 119.5 GFLOP/s
a+b  -->  K= 1000001  1x -> 21.8 GFLOP/s  100x -> 129.0 GFLOP/s
a+b  -->  K= 5000001  1x -> 22.9 GFLOP/s  100x -> 133.6 GFLOP/s
a+b  -->  K=20000001  1x -> 23.2 GFLOP/s  100x -> 135.1 GFLOP/s
a+b  -->  K=50000001  1x -> 23.2 GFLOP/s  100x ->  19.1 GFLOP/s
```

The the `__restrict__` compiler keyword from problem 2, further improves the last part:

```
a+b  -->  K=      17  1x ->  0.0 GFLOP/s  100x ->   0.7 GFLOP/s
a+b  -->  K=      65  1x ->  0.0 GFLOP/s  100x ->   2.6 GFLOP/s
a+b  -->  K=     251  1x ->  0.1 GFLOP/s  100x ->  10.0 GFLOP/s
a+b  -->  K=    1001  1x ->  0.4 GFLOP/s  100x ->  19.8 GFLOP/s
a+b  -->  K=    2001  1x ->  0.9 GFLOP/s  100x ->  39.4 GFLOP/s
a+b  -->  K=    5001  1x ->  2.0 GFLOP/s  100x ->  97.7 GFLOP/s
a+b  -->  K=   10001  1x ->  4.0 GFLOP/s  100x -> 193.1 GFLOP/s
a+b  -->  K=   25001  1x ->  9.9 GFLOP/s  100x -> 478.2 GFLOP/s
a+b  -->  K=   50001  1x -> 19.2 GFLOP/s  100x -> 935.4 GFLOP/s
a+b  -->  K=  100001  1x -> 29.1 GFLOP/s  100x -> 1155.0 GFLOP/s
a+b  -->  K=  250001  1x -> 44.7 GFLOP/s  100x -> 1315.7 GFLOP/s
a+b  -->  K=  500001  1x -> 20.7 GFLOP/s  100x -> 1351.0 GFLOP/s
a+b  -->  K= 1000001  1x -> 21.8 GFLOP/s  100x -> 1457.6 GFLOP/s
a+b  -->  K= 5000001  1x -> 22.9 GFLOP/s  100x -> 1566.6 GFLOP/s
a+b  -->  K=20000001  1x -> 23.2 GFLOP/s  100x -> 1589.6 GFLOP/s
a+b  -->  K=50000001  1x -> 23.2 GFLOP/s  100x -> 224.6 GFLOP/s
```

### 1.2.1 Discussion

**Host to device copy** Transferring data from the system's main memory to the GPU device reaches a peak bandwidth of about 12 GB/s. High bandwidths are observed when copying more than 100000 doubles, i.e. above circe 1 MB of data.

**Permuted copies on device** When shuffling in blocks of 32 elements, the *same* performance is achieved as in the linear copy case (i.e. `p[i] = i`). This can be explained by the fact that groups of 32 threads can exploit the locality, as it is physically the case, where it does not matter if permuted access are executed, since the same chunk of memory is loaded for the whole group of 32 threads within one block.

When accessing instead even positions and multiples of 4, there is a lot of wasted loads from memory, and the performance drops significantly.

**Addition** The highest performance in terms of GFLOP/s is achieved with $K = 250001$ for one addition per kernel, and $K = 50000001$ or $K = 20000001$ respectively. These numbers correspond to convenient warp scheduling opportunities for the 56 SMs. Whenever those can be kept busy


fmahlknecht@student.ethz.ch
19-945-351

as much as possible, the GFLOP/s performance peaks. It is interesting to note that in case of a single addition per kernel, the peak performance is very sharply at $K = 250001$, while in the case of 100 additions per thread, a wider range of K's achieve good performance. This is due to the fact, that with more FLOP/s in a single thread, the execution becomes less dependent on convenient memory loading and access patterns *across* SMs. In general, the performance increases significantly, in the case of more FLOP/s per thread, but not always by the factor of 100, depending on the warping situation.

### 1.3 Benchmark C: Leibniz kernel

This benchmark unveils the power of GPUs:

```
./benchmarks_c
CUDA blocks= 1024  threads/block= 128  iter/thread=16384  pi=3.141592653124  rel error=-1.5e-10  Gterms/s=136.4
OpenMP threads=12  pi=3.141592653123193  rel error=-1.5e-10  Gterms/s=2.5
```

It's more than 50 times faster than the computation using OpenMP threads.

Changing the number of blocks and thread per block does not impact the performance significantly anymore. After several runs, the best result was:

```
./benchmarks_c
CUDA blocks= 8192  threads/block=1024  iter/thread=  256  pi=3.141592653124  rel error=-1.5e-10  Gterms/s=177.3
OpenMP threads=12  pi=3.141592653123193  rel error=-1.5e-10  Gterms/s=2.5
```

However, this result seemed a bit dependent on previous executions (hot cache). There was no clear trend or strategy in the choice of threads and blocks, after reaching a large enough number of iterations per thread.

### 1.4 Kernel evaluation time balancing

In the described example the issue lies in the parallelization architecture of a GPU: threads execute the same code, and the whole block only finishes when *all* threads are done. If we take 1024 threads per block as in our case, and proceed with the random sampling, this means that it is likely to get a high value when sampling randomly from the interval $[1, 10^6]$. In the worst case, when dividing the computation into 1000 blocks and 1000 threads, each of the blocks contains a number close to $10^6$, i.e. the computation time is $\propto 10^6$ for each block, yielding a total computation time of $T \propto 10^9$.

If we now sort the samples first, we expect the worst thread of block $i$ to yield $T_i \propto 1000\, i$, where $1 \leq i \leq 1000$. This means that the total computation time is the summation of this sequence, i.e. $T \propto \frac{10^6}{2} 10^3 = \frac{1}{2} 10^9$. This is indeed half of the previous estimate, which makes the example results plausible.

## 2 N-Body code performance optimization

The output for the unoptimized skeleton code on Piz daint yields:

```
./nbody_0
Statistics: <F>=-1.76055e-10 2.29097e-10 6.4493e-09  <F^2>=3.69391e+08
Average execution time: 5213.8 ms
```

The manual aliasing fix yields:


Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

```
./nbody_a
Statistics: <F>=-1.76055e-10 2.29097e-10 6.4493e-09  <F^2>=3.69391e+08
Average execution time: 711.1 ms
```

with the copmiler keyword `__restrict__`, we get almost the same results, however also repetitive execution shows a 10 ms increase in computation time with respect to the manual fix:

```
./nbody_a
Statistics: <F>=-1.76055e-10 2.29097e-10 6.4493e-09  <F^2>=3.69391e+08
Average execution time: 721.0 ms
```

Reusing the computation of $\frac{1}{r^3}$ yields:

```
./nbody_b
Statistics: <F>=-1.74563e-10 1.94596e-10 5.99638e-09  <F^2>=3.69391e+08
Average execution time: 670.5 ms
```

Using `rsqrt()` yields:

```
./nbody_c
Statistics: <F>=-1.7755e-10 2.91852e-10 5.95999e-09  <F^2>=3.69391e+08
Average execution time: 499.5 ms
```

Consecutively loading the particles for one block into shared memory gives additional improvements:

```
./nbody_d
Statistics: <F>=-1.7755e-10 2.91852e-10 5.95999e-09  <F^2>=3.69391e+08
Average execution time: 367.9 ms
```

## 2.1 Summary and discussion

The given statistics in the output are – to a small error – the same. Minor deviations are introduced by computing $\frac{1}{r^3}$ once and reusing it (most likely due to some floating point calculation artifacts), as well as using `rsqrt()`. However, the overall statistics remain basically the same.

**Reuse computation optimization** The floating point artifacts, as mentioned, are most likely also the reason why the compiler does *not* perform this optimization of reusing the local computation in this context automatically: it is not equivalent in floating point multiplication, due to finite mantissa representation, computing the latter part first and then multiplying by $dx, dy, dz$ respectively. Depending on the specific values of $dx$, higher resolution might be maintained, when sequentially multiplying by $\frac{1}{r}$. This is reflected also in the slightly changing statistics.


fmahlknecht@student.ethz.ch
19-945-351

**Result discussion** The speedup results are summarized in table 1.

| Version | Optimization Technique | Execution time [ms] |
|---|---|---|
| 0 | n/a | 5213.8 |
| a | fix aliasing | 711.1 |
| a | fix aliasing with `__restrict__` | 721.0 |
| b | reuse local computations | 670.5 |
| c | use `rsqrt()` | 499.5 |
| d | shared memory | 367.9 |

Table 1: N-Body kernel optimization summary

The most significant improvement is introduced by the fix of aliasing, and also the following optimization combined up to shared memory, provide an additional factor 2 speedup.

# 3 Jacobi Method

The implementation is provided in the source code attached.

## 3.1 Non-zero elements

We have:

$$A_{ij} = \begin{cases} -\frac{4}{h^2} & \text{for } i = j \Rightarrow i_x = j_x,\ i_y = j_y \\ \frac{1}{h^2} & \text{for } \begin{cases} i_y = j_y,\ i_x = j_x + 1 \\ i_y = j_y,\ i_x = j_x - 1 \\ i_x = j_x,\ i_y = j_y + 1 \\ i_x = j_x,\ i_y = j_y - 1 \end{cases} \end{cases} \tag{1}$$

And the forward step equation becomes:

$$x^{(k+1)}_{i_x,i_y} = \frac{1}{A_{i_x i_y, i_x i_y}} \left( b_{i_x i_y} - \sum_{(j_x,j_y)\neq(i_x,i_Y y)} A_{i_x i_y, j_x j_y} x^{(k)}_{j_x j_y} \right) \tag{2}$$

$$= -\frac{h^2}{4} \left( b_{i_x i_y} - \frac{1}{h^2} \left( x^{(k)}_{i_x-1\, i_y} + x^{(k)}_{i_x+1\, i_y} + x^{(k)}_{i_x\, i_y-1} + x^{(k)}_{i_x\, i_y+1} \right) \right) \tag{3}$$

So finally we get with $x = \varphi$ and $b_{i_x i_y} = -\rho_{i_x i_y}$

$$\varphi^{(k+1)}_{i_x,i_y} = \frac{h^2}{4} \rho_{i_x i_y} + \frac{1}{4} \left( \varphi^{(k)}_{i_x-1\, i_y} + \varphi^{(k)}_{i_x+1\, i_y} + \varphi^{(k)}_{i_x\, i_y-1} + \varphi^{(k)}_{i_x\, i_y+1} \right) \tag{4}$$

## 3.2 Results

As desired the norms decrease with more iterations:


Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

```
./electrostatics
01000  Aphi - -rho  ==>  L1=    4.7798  L2=  0.003348
02000  Aphi - -rho  ==>  L1=     4.378  L2= 0.0016121
03000  Aphi - -rho  ==>  L1=    4.0593  L2=0.00098464
[...]
498000  Aphi - -rho  ==>  L1=3.8354e-07  L2=2.7985e-18
499000  Aphi - -rho  ==>  L1=3.7189e-07  L2=2.6311e-18
500000  Aphi - -rho  ==>  L1= 3.606e-07  L2=2.4737e-18
```

Figure 1 shows a frame of the video rendered by the visualization python script.

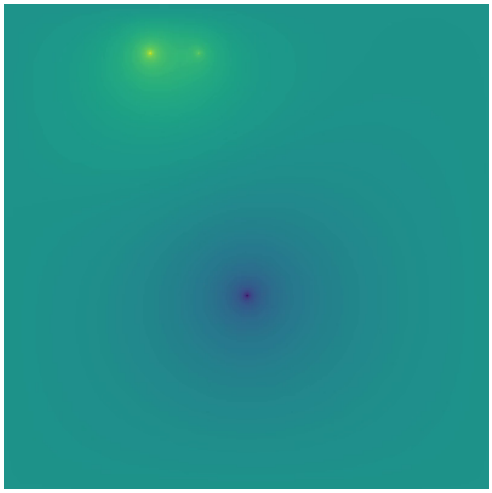

Figure 1: Video output capture


fmahlknecht@student.ethz.ch
19-945-351