Assignment IV: Advanced CUDA

Link to GitHub repository: https://github.com/ChrisWe99/DD2360HT22

Exercise 1 – Thread Scheduling and Execution Efficiency

1. **Assume X=800 and Y=600. Assume that we decided to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. How many warps will be generated during the execution of the kernel? How many warps will have control divergence? Please explain your answers.**

For the results of division, it has to be rounded up since no partial threads exist.

#warps(per block) = #threads(per block) / #threads(per warp) = 16*16 / 32 = 8

Required blocks (size of block array):
#pixels_x / gridsize_x * pixels_y / gridsize_y = 800 / 16 * 600/16 = 50 * 38 = 1900

#warps = #warps(per block) * #blocks = 1900 * 8 = 15200

➔ 15200 warps will be generated during the execution of the kernel

Since the number of threads corresponds to the size of the image, there are 0 warps with control divergence.

2. **Now assume X=600 and Y=800 instead, how many warps will have control divergence? Please explain your answers.**

Required blocks (size of block array):
#pixels_x / gridsize_x * pixels_y / gridsize_y = 600 / 16 * 800/16 = 38 * 50 = 1900

This time, there will be control divergence because within single warps, it is required to double the pixel values (last function) but not for all of their threads. Since this only happens in y direction for the last column, the number of warps with control divergence is 50 (warps) * 8(warps per block) = 400.

3. **Now assume X=600 and Y=799, how many warps will have control divergence? Please explain your answers.**

Required blocks (size of block array):
#pixels_x / gridsize_x * pixels_y / gridsize_y = 600 / 16 * 799/16 = 38 * 50 = 1900

This time, in x direction the last column of warps leads to control divergence (reasoning similar to previous question) which leads to 8*50 = 400 warps with control divergence.
Due to the image size of 799 in one direction, 1 warp per block of the last row produces control divergence. This means 1*38 more warps with control divergence.

Depending on how to count, for the total number of warps, one warp has to be deducted because it was counted twice.

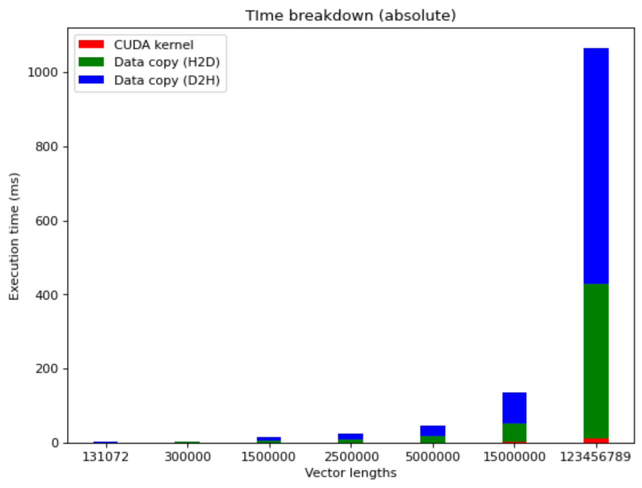
- ➔ Total number of warps with control divergence: 400 + 38 - 1 = 437.

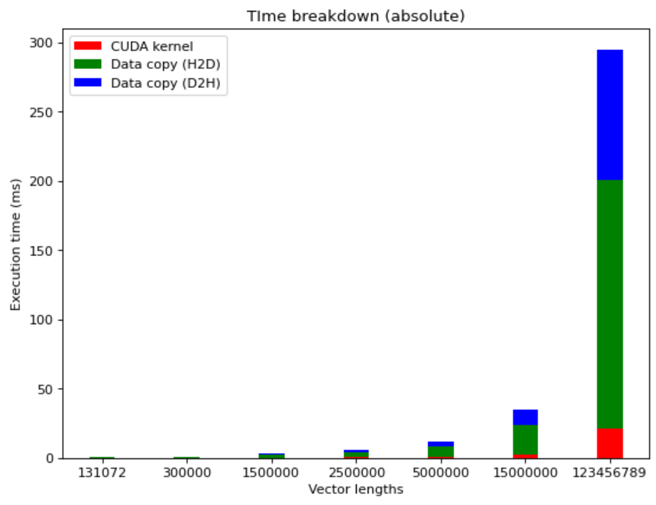
Exercise 2 – CUDA Streams

1. **Compared to the non-streamed vector addition, what performance gain do you get? Present in a plot (you may include comparison at different vector length)**

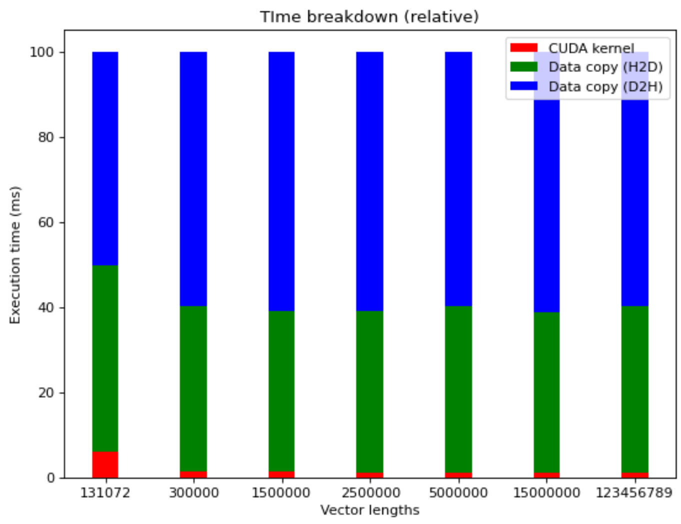
For profiling, nvprof was used: `!nvprof !nvprof ./lab4_ex2 <input_vector_length>`

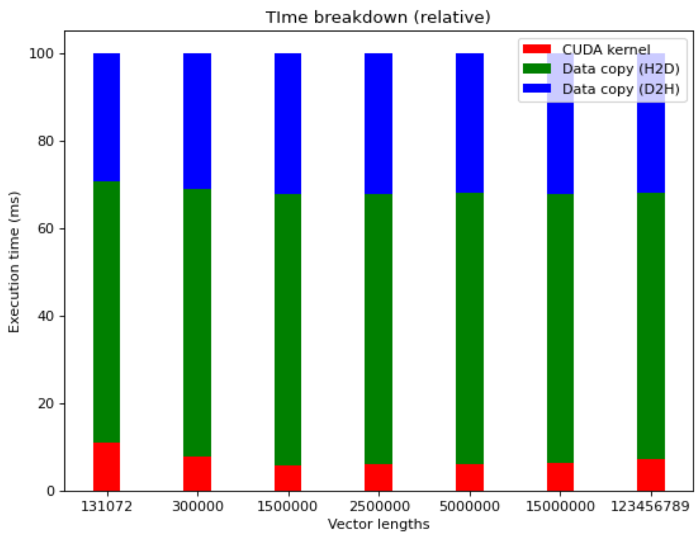
Absolute time measurements. Comparison without streams (left) and with streams (right):





Relative time measurements. Comparison without streams (left) and with streams (right):





As it can be seen, the absolute required time is drastically reduced for the streamed version (~30% of previous time required for the largest tested vector).

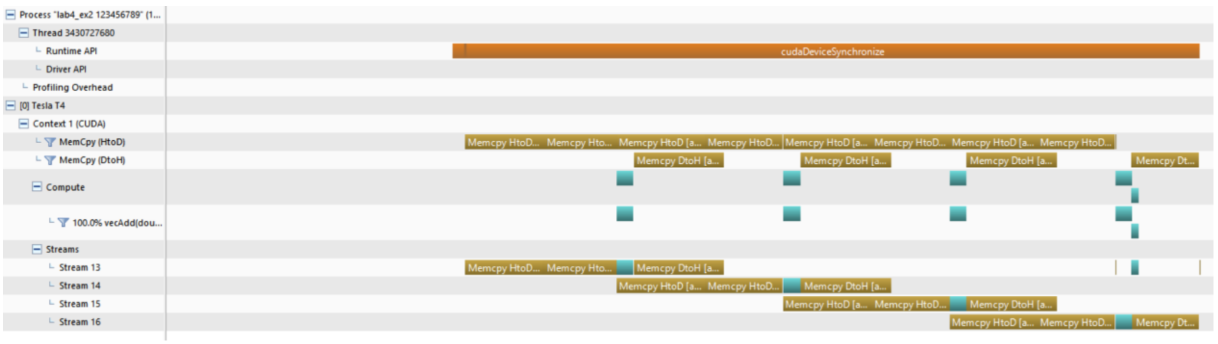
2. **Use nvprof to collect traces and the NVIDIA Visual Profiler (nvvp) to visualize the overlap of communication and computation. To use nvvp, you can check Tutorial: NVVP - Visualize nvprof Traces**

For this task, I used a vector size of 123456789.

The command was copied from the given tutorial:

```
!nvprof --output-profile hw_4_ex_2_vec_123456789.nvprof -f ./lab4_ex2 123456789
```

The subsequent figure from Nvidia Visual Profiler clearly shows that communication (memory operations) and the execution of vecAdd are performed in parallel.



Note for future students: Installing Nvidia Visual Profiler on a Windows11 laptop without a dedicated GPU took me a lot of effort:

Main solutions include:

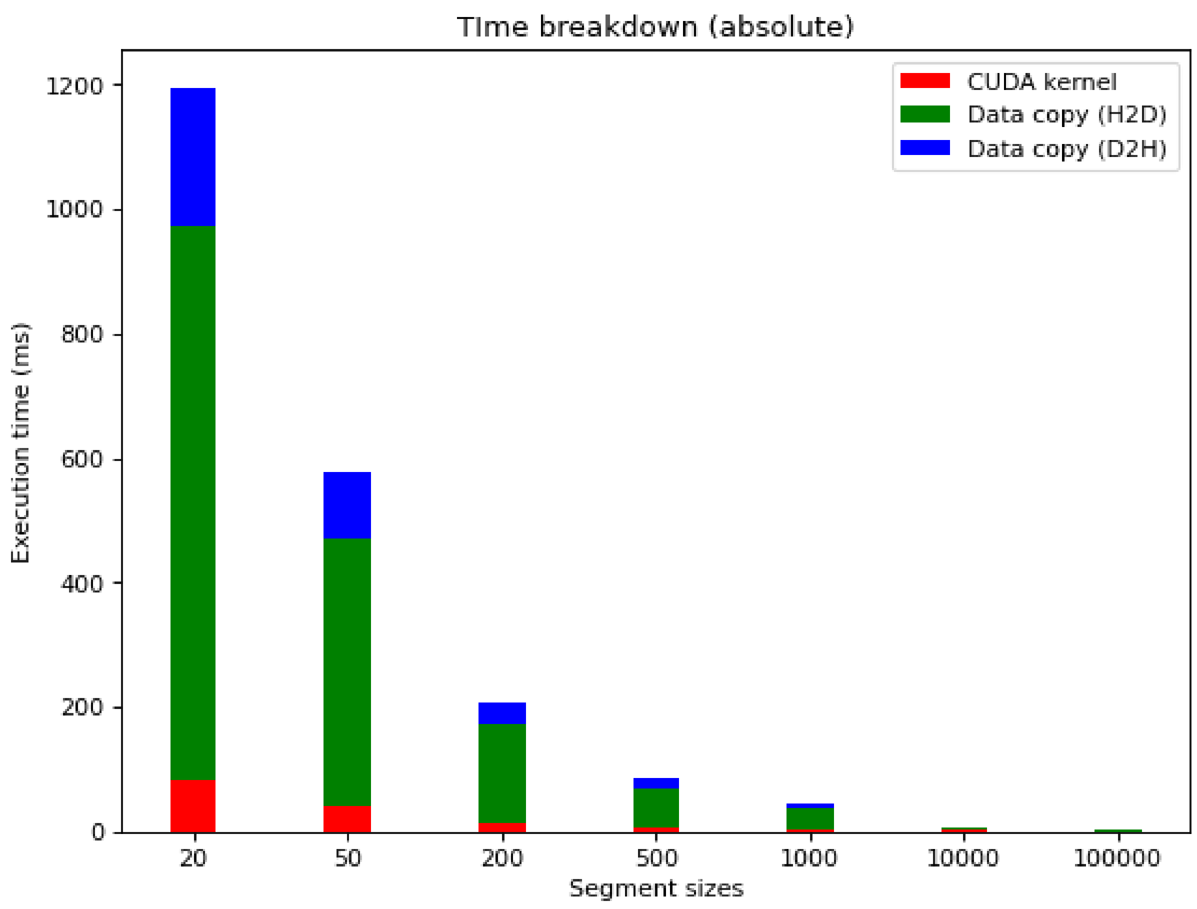
- When installing the CUDA toolkit, it has to be installed in the “expert mode” instead of the “express mode”. There, you need to disselect multiple features like the Visual Studio Integration (and according to different posts maybe some others but this seemed to be sufficient).
- The Visual Profiler can only be run from the command line using a command which points to the correct JRE. E.g.:

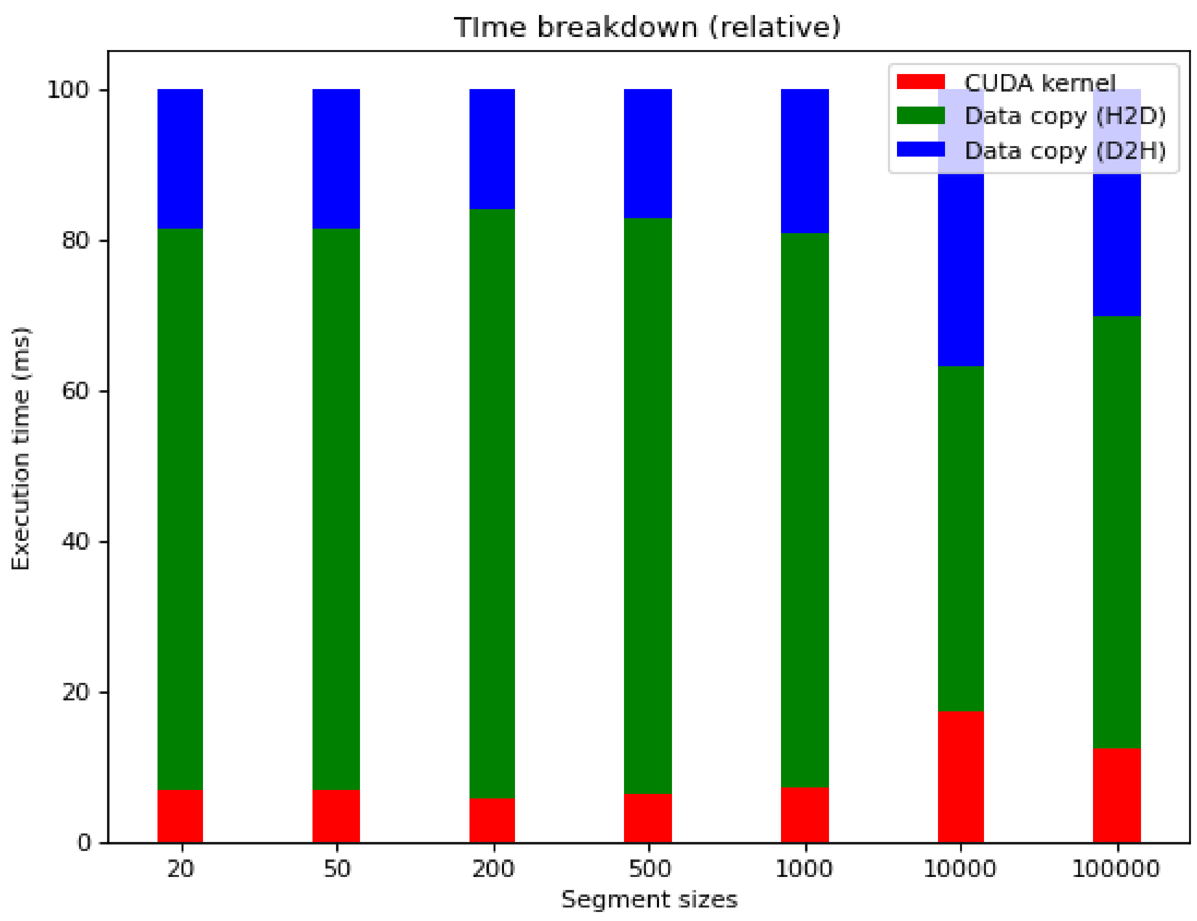
  ```
  nvvp -vm "C:\Program Files\Java\jdk1.8.0_351\jre\bin\java"
  ```

  The -vm option may be required in some cases.

3. What is the impact of segment size on performance? Present in a plot (you may choose a large vector and compare 4-8 different segment sizes)

For this task, I fixed the vector size to 1234567 and tried out different segment sizes which can be seen in the following plots:


TIme breakdown (absolute)
CUDA kernel
Data copy (H2D)
Data copy (D2H)
Execution time (ms)
0
200
400
600
800
1000
1200
20
50
200
500
1000
10000
100000
Segment sizes



As it can be seen, a small segment size creates overhead since many memory operations / cuda API calls are required.

Exercise 3 – Pinned Memory and Unified Memory

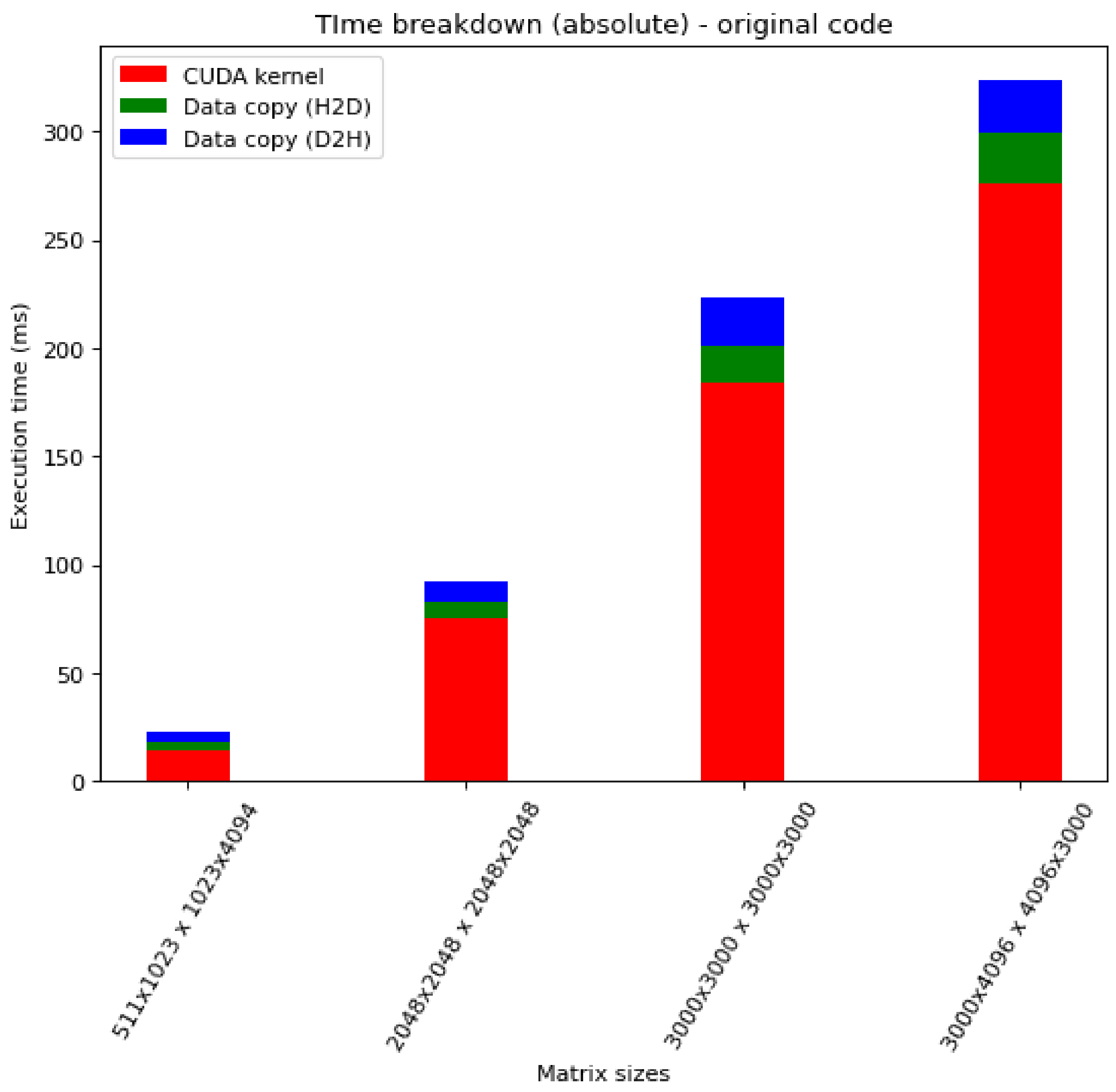
1. What are the differences between pageable memory and pinned memory, what are the tradeoffs?

Inspired by the lecture slides and this post: Pinned memory allows to avoid intermediate transfers so that small transfers can be batched in one large data transfer. If a source or destination of cudaMemcpy() in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory which generates extra everhead. However, if the host memory source/destination is already in pinned memory, it is faster. Compared to pageable memory, pinned memory is much more limited and yields e.g. a reduction of the memory availability for the host processing.

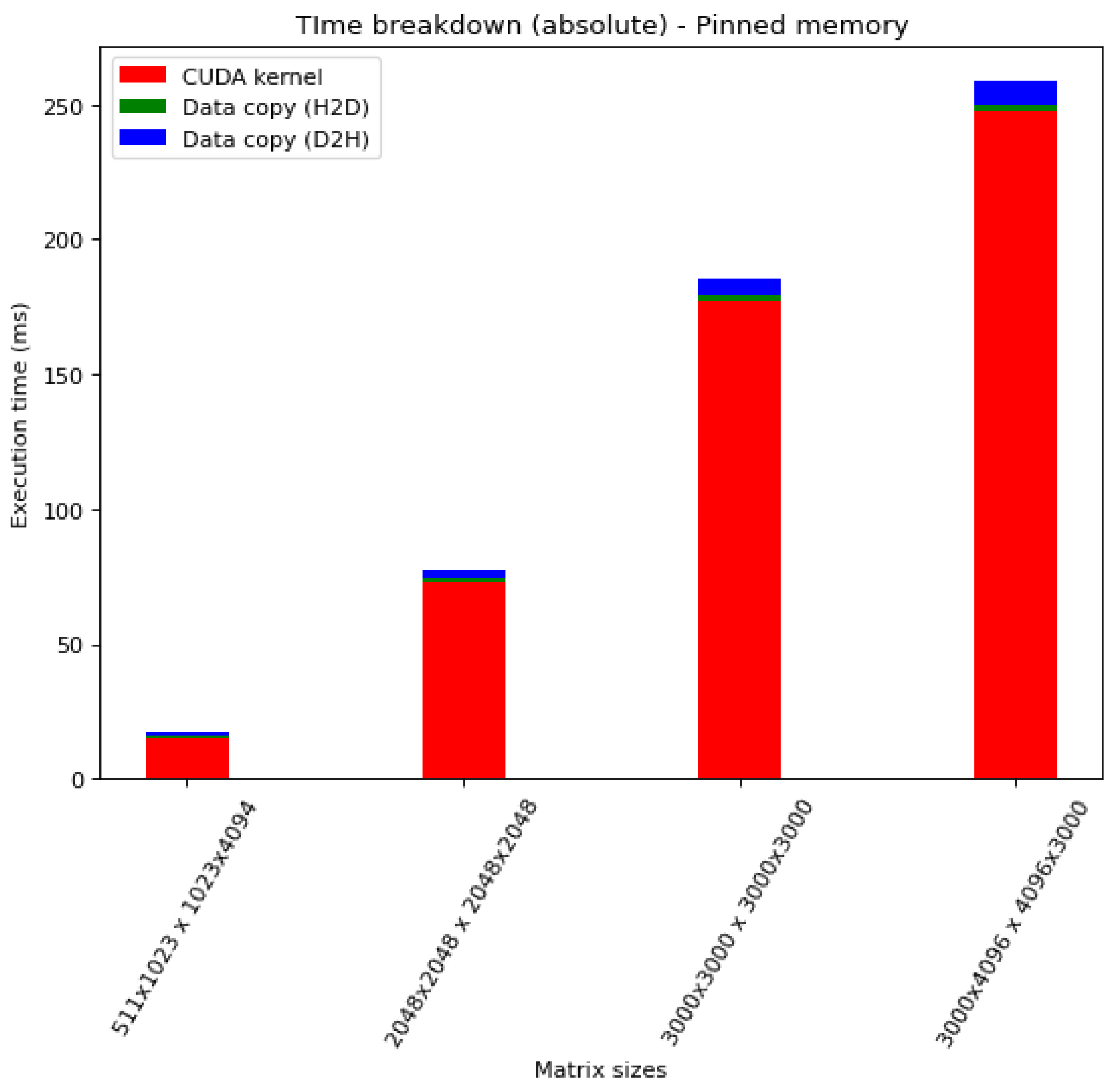
2. **Compare the profiling results between your original code and the new version using pinned memory. Do you see any difference in terms of the breakdown of execution time after changing to pinned memory?**

Used datatype: float (not double)

Original profiling result:



Pinned memory result:



It can be observed that the D2H and H2D copying is a lot faster than for the original code. Especially H2D improves a lot due to the faster transfer. This is due to the speedup of memory access as it was explained in the previous question.

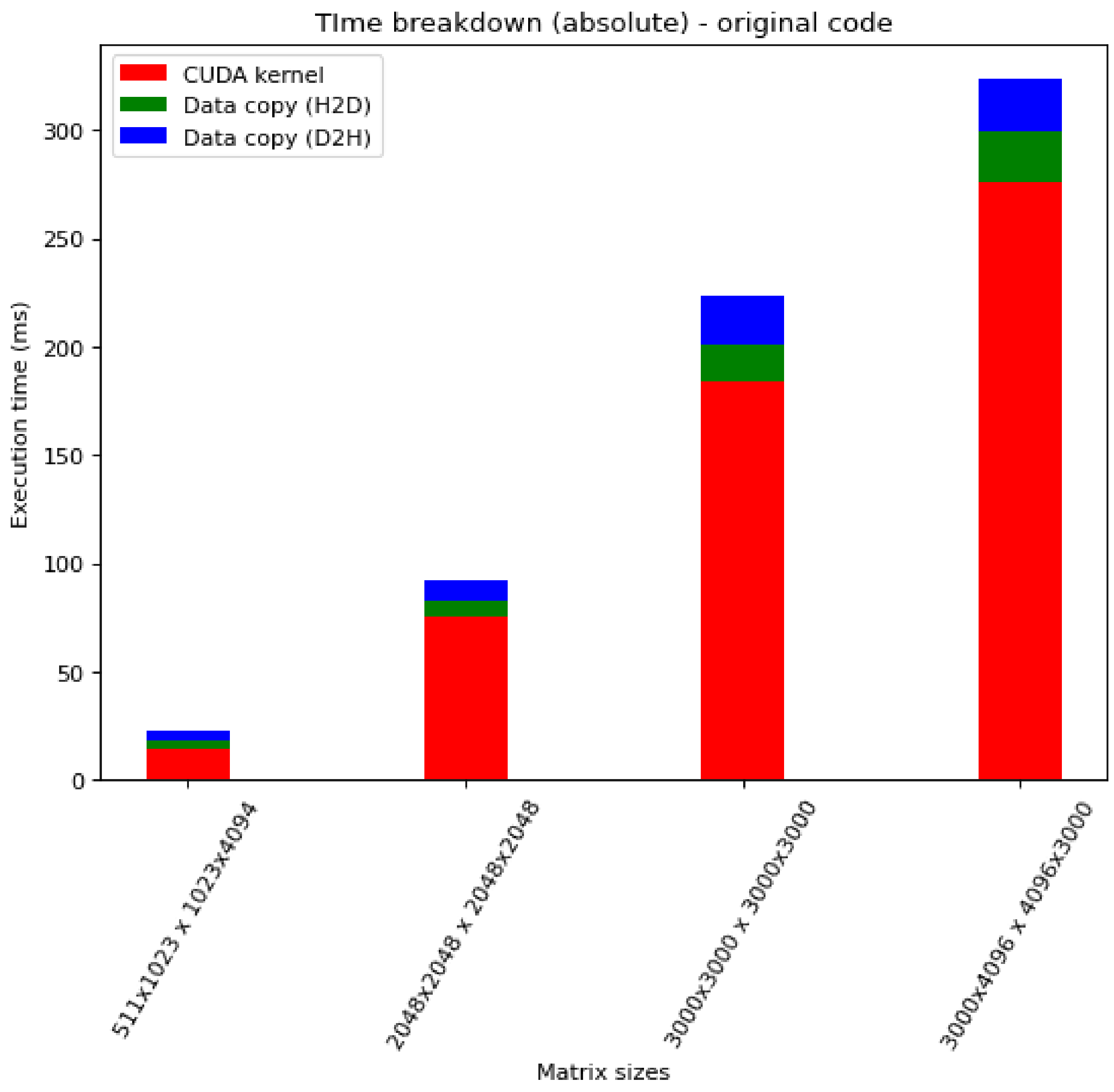
3. What is a managed memory? What are the implications of using managed memory?

Inspired by the lecture slides: Managed memory (= unified memory) is a single memory space for host and device memories. This means that data can be written and read from both, the CPU and GPU without explicit transfer. The CUDA system software takes care of migrating memory pages to the memory of the accessing processor.

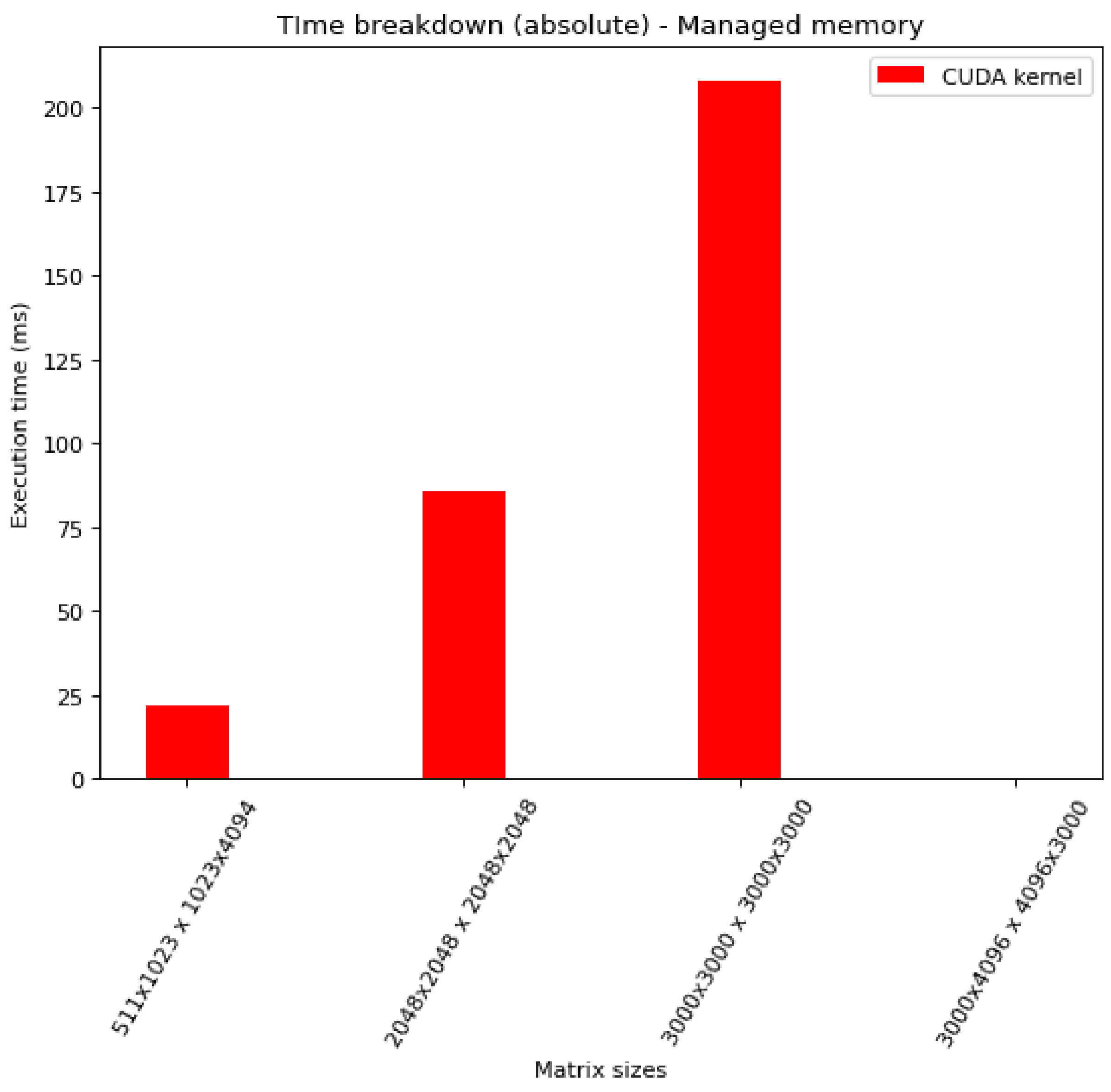
4. Compare the profiling results between your original code and the new version using managed memory. What do you observe in the profiling results?

Used datatype: float (not double)

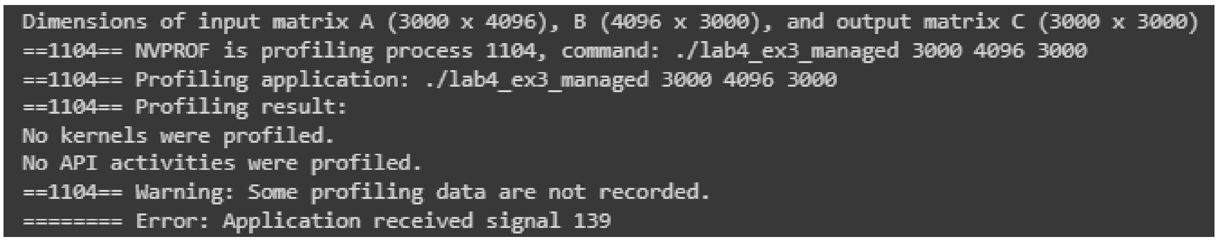
Original profiling result:



Managed memory result:



Please note: For the largest input matrices, I always received the following error which seems to be a common problem with nvprof profiling larger managed memory operations according to this and other posts.

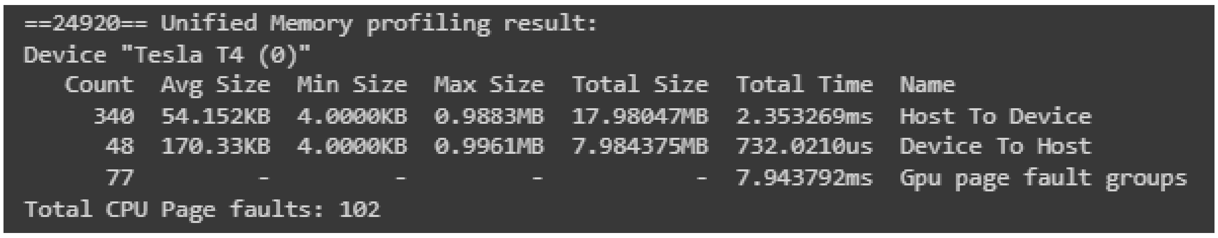


```
Dimensions of input matrix A (3000 x 4096), B (4096 x 3000), and output matrix C (3000 x 3000)
==1104== NVPROF is profiling process 1104, command: ./lab4_ex3_managed 3000 4096 3000
==1104== Profiling application: ./lab4_ex3_managed 3000 4096 3000
==1104== Profiling result:
No kernels were profiled.
No API activities were profiled.
==1104== Warning: Some profiling data are not recorded.
======== Error: Application received signal 139
```

I tried many different fixes but none of them worked or were possible inside Google Colab.

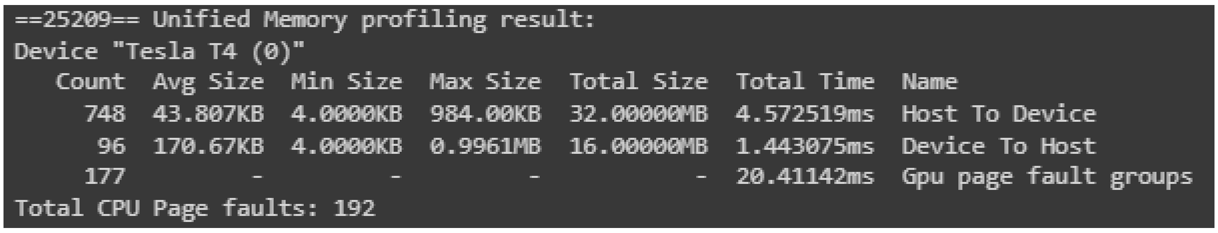
Since the kernel now takes over the data copy operations, the kernel execution time for managed memory is slightly more than for the original code. However, it can be observed that the total required time is faster.
It is interesting to see that there are a lot of CPU and GPU page faults monitored. This could be optimized for further speedup. For larger matrices, the number of faults increases.

Page faults for 511x1023 1023x4094 matrices:



```
==24920== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     340  54.152KB  4.0000KB  0.9883MB  17.98047MB  2.353269ms  Host To Device
      48  170.33KB  4.0000KB  0.9961MB  7.984375MB  732.0210us  Device To Host
      77         -         -         -           -  7.943792ms  Gpu page fault groups
Total CPU Page faults: 102
```

Page faults for 2048x2048 2048x2408 matrices:



```
==25209== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     748  43.807KB  4.0000KB  984.00KB  32.00000MB  4.572519ms  Host To Device
      96  170.67KB  4.0000KB  0.9961MB  16.00000MB  1.443075ms  Device To Host
     177         -         -         -           -  20.41142ms  Gpu page fault groups
Total CPU Page faults: 192
```

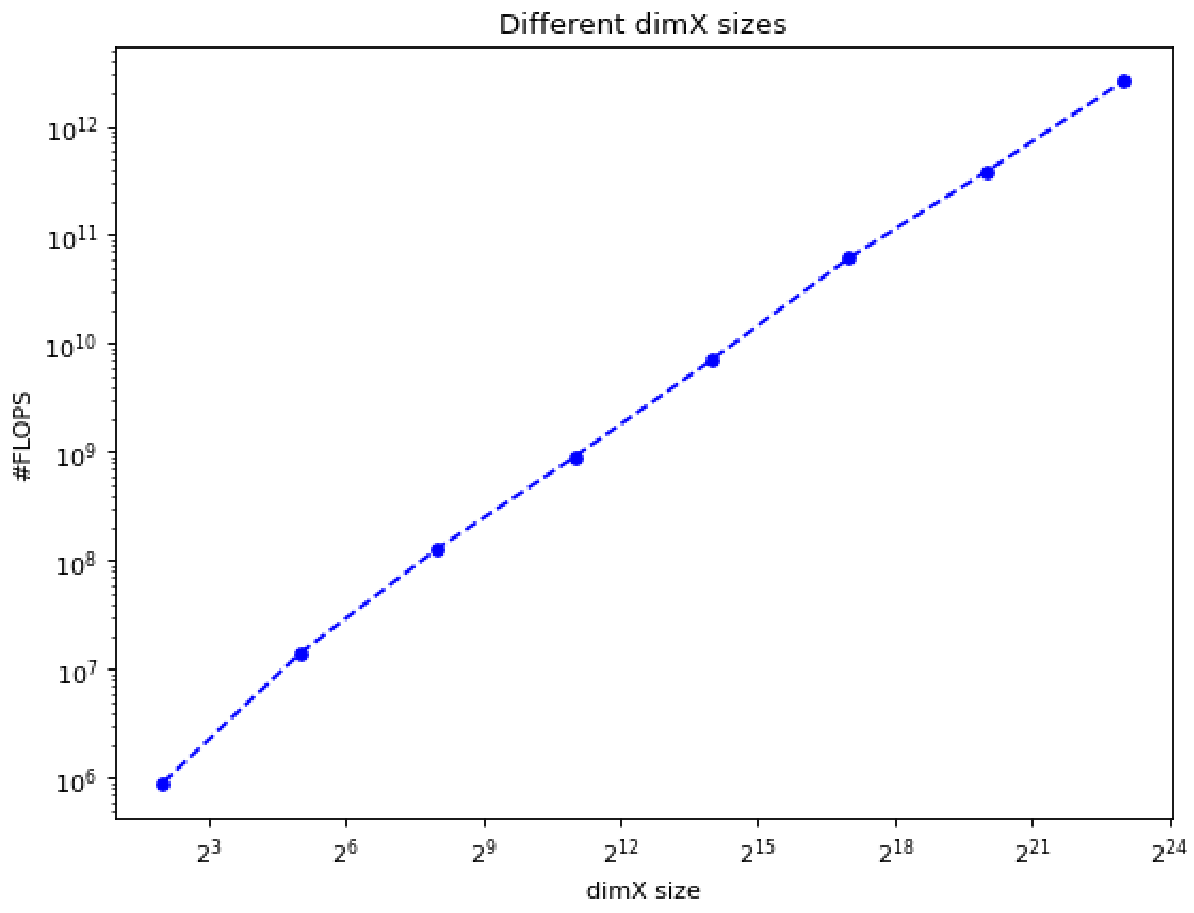
Exercise 4 – Heat Equation with using NVIDIA libraries

1. **Run the program with different dimX values. For each one, approximate the FLOPS (floating-point operation per second) achieved in computing the SMPV (sparse matrix multiplication). Report FLOPS at different input sizes in a FLOPS. What do you see compared to the peak throughput you report in Lab2?**

The number of FLOPS corresponds to the following formula:
nsteps * (3 * dimX – 6) / total_time

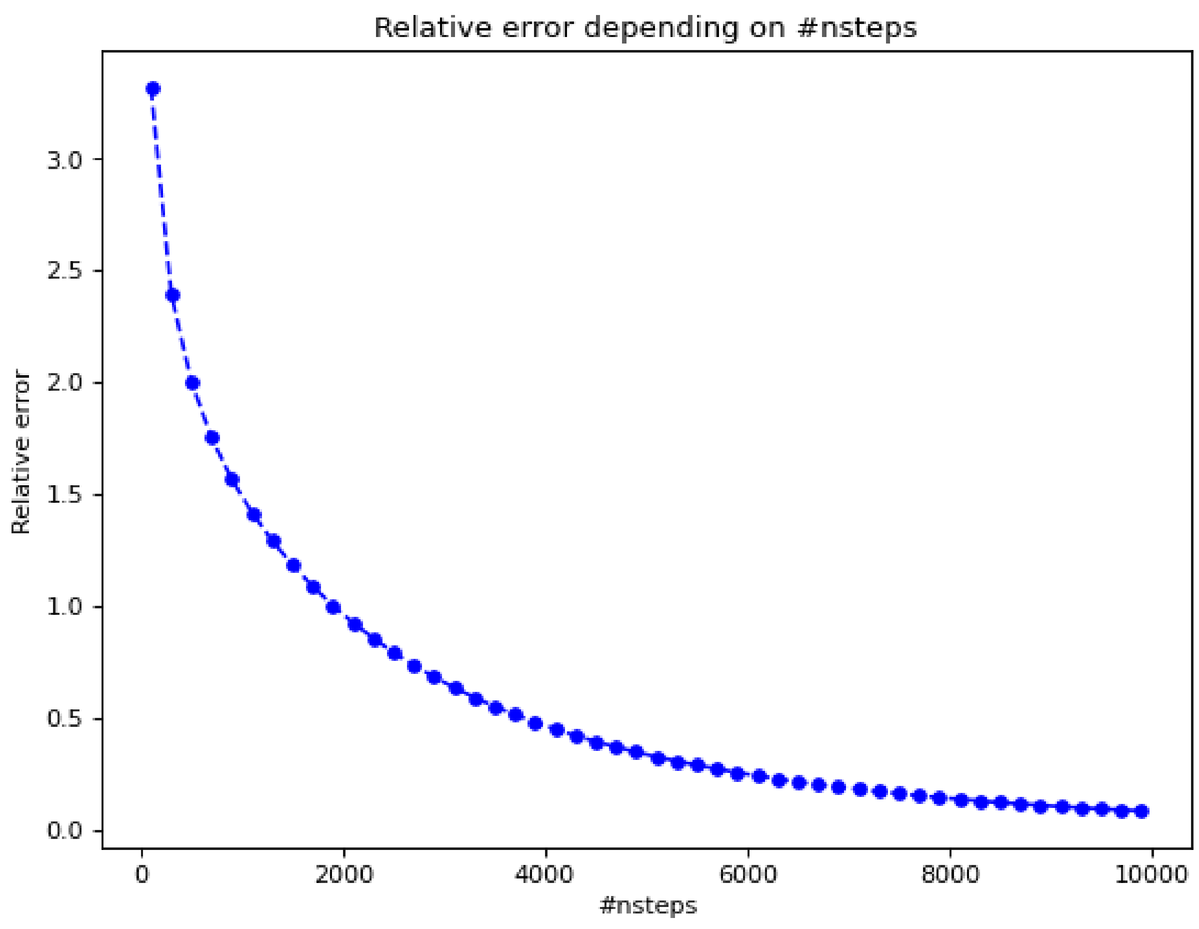
The total_time is the time that is required for the whole for loop of calculating the SMPVs.

For the number of FLOPS comparison with different dimXs, I fixed the nsteps to 100 as it was the lowest suggest size in the next question:



It can be observed that larger dimX values lead to a higher number of FLOPS. Compared to the peak throughout in Lab2, we are not able to reach this peak but to get close to it.

2. **Run the program with dimX=128 and vary nsteps from 100 to 10000. Plot the relative error of the approximation at different nstep. What do you observe?**



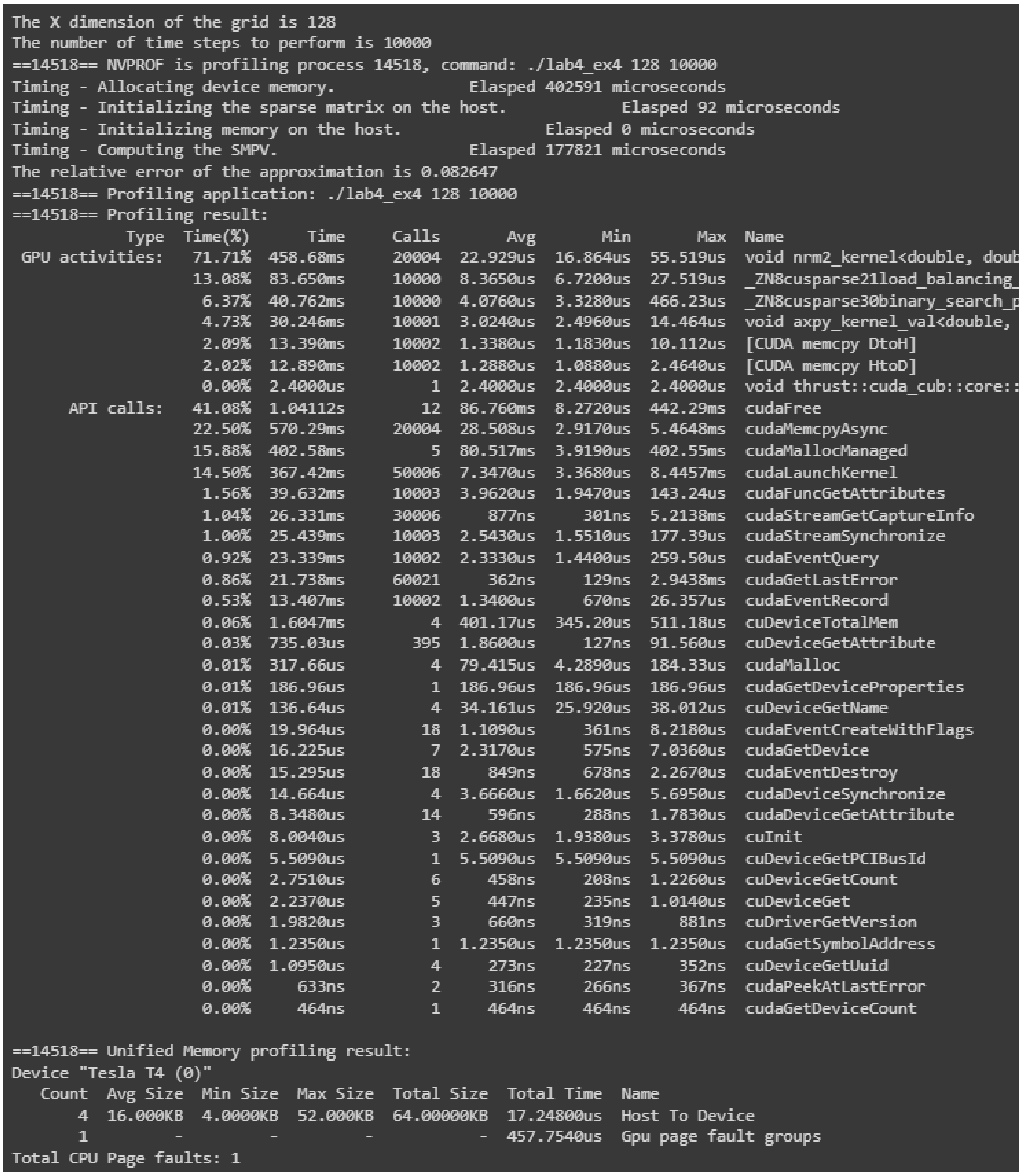
A larger number of nsteps yields a smaller error.

3. Compare the performance with and without the prefetching in Unified Memory. How is the performance impact? [Optional: using nvprof to get metrics on UM]

Prefetching helps to avoid CPU Page faults. Consequently, I expect a speedup which can also be observed.

I ran the application both times with the parameters dimX = 128 and nsteps = 10000. The time for the GPU activities doesn't change significantly. However, the time for initializing the sparse matrix is reduced a lot with prefetching (92 mikroseconds to 2 mikroseconds). Though, it has to be considered that prefetching requires some time as well which reduces the speedup.

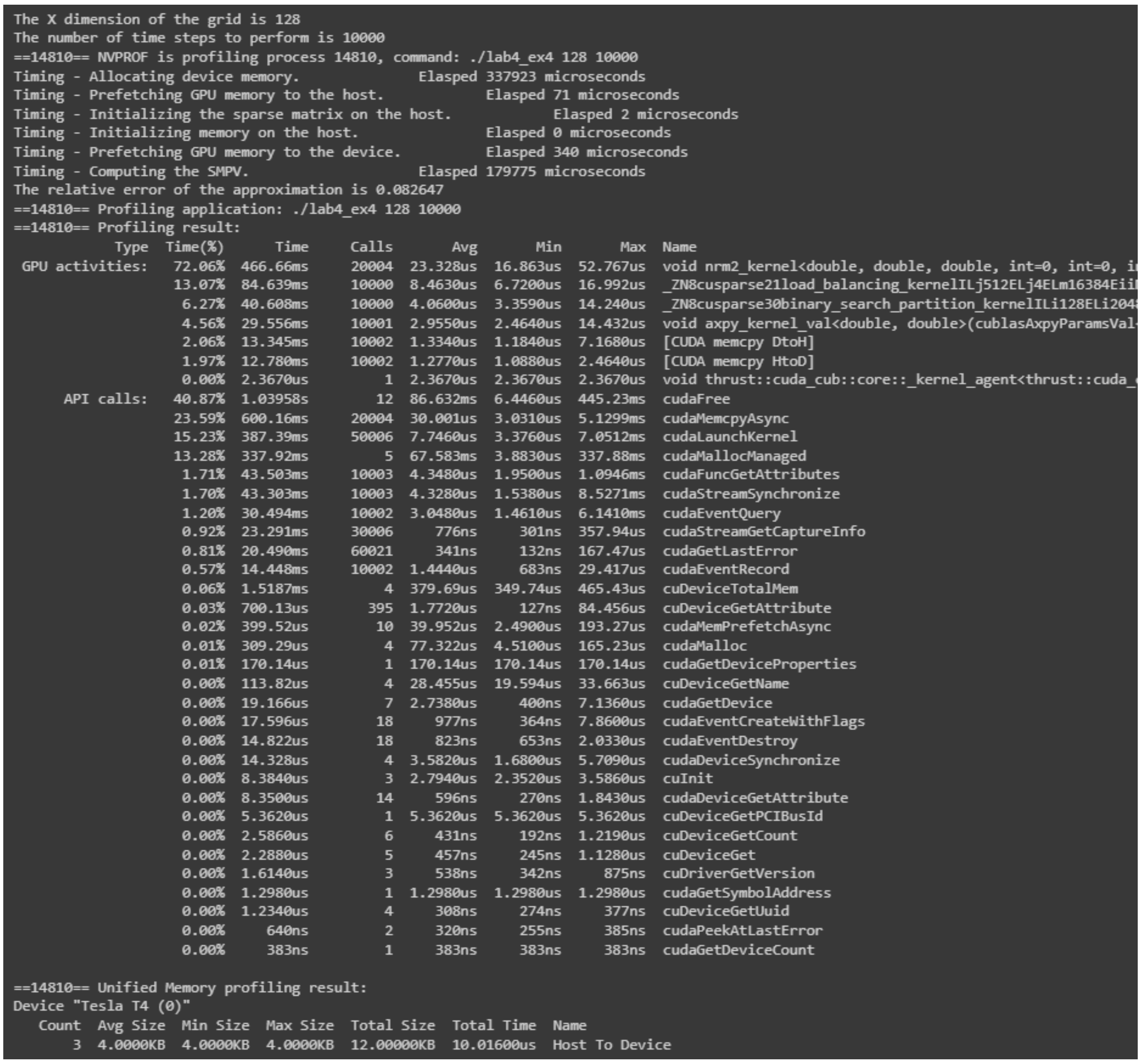
Without prefetching:



```
The X dimension of the grid is 128
The number of time steps to perform is 10000
==14518== NVPROF is profiling process 14518, command: ./lab4_ex4 128 10000
Timing - Allocating device memory.              Elasped 402591 microseconds
Timing - Initializing the sparse matrix on the host.            Elasped 92 microseconds
Timing - Initializing memory on the host.               Elasped 0 microseconds
Timing - Computing the SMPV.                    Elasped 177821 microseconds
The relative error of the approximation is 0.082647
==14518== Profiling application: ./lab4_ex4 128 10000
==14518== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   71.71%  458.68ms     20004  22.929us  16.864us  55.519us  void nrm2_kernel<double, doub
                   13.08%  83.650ms     10000  8.3650us  6.7200us  27.519us  _ZN8cusparse21load_balancing_
                    6.37%  40.762ms     10000  4.0760us  3.3280us  466.23us  _ZN8cusparse30binary_search_p
                    4.73%  30.246ms     10001  3.0240us  2.4960us  14.464us  void axpy_kernel_val<double,
                    2.09%  13.390ms     10002  1.3380us  1.1830us  10.112us  [CUDA memcpy DtoH]
                    2.02%  12.890ms     10002  1.2880us  1.0880us  2.4640us  [CUDA memcpy HtoD]
                    0.00%  2.4000us         1  2.4000us  2.4000us  2.4000us  void thrust::cuda_cub::core::
      API calls:   41.08%  1.04112s        12  86.760ms  8.2720us  442.29ms  cudaFree
                   22.50%  570.29ms     20004  28.508us  2.9170us  5.4648ms  cudaMemcpyAsync
                   15.88%  402.58ms         5  80.517ms  3.9190us  402.55ms  cudaMallocManaged
                   14.50%  367.42ms     50006  7.3470us  3.3680us  8.4457ms  cudaLaunchKernel
                    1.56%  39.632ms     10003  3.9620us  1.9470us  143.24us  cudaFuncGetAttributes
                    1.04%  26.331ms     30006     877ns     301ns  5.2138ms  cudaStreamGetCaptureInfo
                    1.00%  25.439ms     10003  2.5430us  1.5510us  177.39us  cudaStreamSynchronize
                    0.92%  23.339ms     10002  2.3330us  1.4400us  259.50us  cudaEventQuery
                    0.86%  21.738ms     60021     362ns     129ns  2.9438ms  cudaGetLastError
                    0.53%  13.407ms     10002  1.3400us     670ns  26.357us  cudaEventRecord
                    0.06%  1.6047ms         4  401.17us  345.20us  511.18us  cuDeviceTotalMem
                    0.03%  735.03us       395  1.8600us     127ns  91.560us  cuDeviceGetAttribute
                    0.01%  317.66us         4  79.415us  4.2890us  184.33us  cudaMalloc
                    0.01%  186.96us         1  186.96us  186.96us  186.96us  cudaGetDeviceProperties
                    0.01%  136.64us         4  34.161us  25.920us  38.012us  cuDeviceGetName
                    0.00%  19.964us        18  1.1090us     361ns  8.2180us  cudaEventCreateWithFlags
                    0.00%  16.225us         7  2.3170us     575ns  7.0360us  cudaGetDevice
                    0.00%  15.295us        18     849ns     678ns  2.2670us  cudaEventDestroy
                    0.00%  14.664us         4  3.6660us  1.6620us  5.6950us  cudaDeviceSynchronize
                    0.00%  8.3480us        14     596ns     288ns  1.7830us  cudaDeviceGetAttribute
                    0.00%  8.0040us         3  2.6680us  1.9380us  3.3780us  cuInit
                    0.00%  5.5090us         1  5.5090us  5.5090us  5.5090us  cuDeviceGetPCIBusId
                    0.00%  2.7510us         6     458ns     208ns  1.2260us  cuDeviceGetCount
                    0.00%  2.2370us         5     447ns     235ns  1.0140us  cuDeviceGet
                    0.00%  1.9820us         3     660ns     319ns     881ns  cuDriverGetVersion
                    0.00%  1.2350us         1  1.2350us  1.2350us  1.2350us  cudaGetSymbolAddress
                    0.00%  1.0950us         4     273ns     227ns     352ns  cuDeviceGetUuid
                    0.00%     633ns         2     316ns     266ns     367ns  cudaPeekAtLastError
                    0.00%     464ns         1     464ns     464ns     464ns  cudaGetDeviceCount

==14518== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  16.000KB  4.0000KB  52.000KB  64.00000KB  17.24800us  Host To Device
       1         -         -         -           -  457.7540us  Gpu page fault groups
Total CPU Page faults: 1
```

With prefetching:



```
The X dimension of the grid is 128
The number of time steps to perform is 10000
==14810== NVPROF is profiling process 14810, command: ./lab4_ex4 128 10000
Timing - Allocating device memory.              Elasped 337923 microseconds
Timing - Prefetching GPU memory to the host.            Elasped 71 microseconds
Timing - Initializing the sparse matrix on the host.            Elasped 2 microseconds
Timing - Initializing memory on the host.               Elasped 0 microseconds
Timing - Prefetching GPU memory to the device.          Elasped 340 microseconds
Timing - Computing the SMPV.                    Elasped 179775 microseconds
The relative error of the approximation is 0.082647
==14810== Profiling application: ./lab4_ex4 128 10000
==14810== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   72.06%  466.66ms     20004  23.328us  16.863us  52.767us  void nrm2_kernel<double, double, double, int=0, int=0, i
                   13.07%  84.639ms     10000  8.4630us  6.7200us  16.992us  _ZN8cusparse21load_balancing_kernelILj512ELj4ELm16384EiiE
                    6.27%  40.608ms     10000  4.0600us  3.3590us  14.240us  _ZN8cusparse30binary_search_partition_kernelILi128ELi204
                    4.56%  29.556ms     10001  2.9550us  2.4640us  14.432us  void axpy_kernel_val<double, double>(cublasAxpyParamsVal
                    2.06%  13.345ms     10002  1.3340us  1.1840us  7.1680us  [CUDA memcpy DtoH]
                    1.97%  12.780ms     10002  1.2770us  1.0880us  2.4640us  [CUDA memcpy HtoD]
                    0.00%  2.3670us         1  2.3670us  2.3670us  2.3670us  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_
      API calls:   40.87%  1.03958s        12  86.632ms  6.4460us  445.23ms  cudaFree
                   23.59%  600.16ms     20004  30.001us  3.0310us  5.1299ms  cudaMemcpyAsync
                   15.23%  387.39ms     50006  7.7460us  3.3760us  7.0512ms  cudaLaunchKernel
                   13.28%  337.92ms         5  67.583ms  3.8830us  337.88ms  cudaMallocManaged
                    1.71%  43.503ms     10003  4.3480us  1.9500us  1.0946ms  cudaFuncGetAttributes
                    1.70%  43.303ms     10003  4.3280us  1.5380us  8.5271ms  cudaStreamSynchronize
                    1.20%  30.494ms     10002  3.0480us  1.4610us  6.1410ms  cudaEventQuery
                    0.92%  23.291ms     30006     776ns     301ns  357.94us  cudaStreamGetCaptureInfo
                    0.81%  20.490ms     60021     341ns     132ns  167.47us  cudaGetLastError
                    0.57%  14.448ms     10002  1.4440us     683ns  29.417us  cudaEventRecord
                    0.06%  1.5187ms         4  379.69us  349.74us  465.43us  cuDeviceTotalMem
                    0.03%  700.13us       395  1.7720us     127ns  84.456us  cuDeviceGetAttribute
                    0.02%  399.52us        10  39.952us  2.4900us  193.27us  cudaMemPrefetchAsync
                    0.01%  309.29us         4  77.322us  4.5100us  165.23us  cudaMalloc
                    0.01%  170.14us         1  170.14us  170.14us  170.14us  cudaGetDeviceProperties
                    0.00%  113.82us         4  28.455us  19.594us  33.663us  cuDeviceGetName
                    0.00%  19.166us         7  2.7380us     400ns  7.1360us  cudaGetDevice
                    0.00%  17.596us        18     977ns     364ns  7.8600us  cudaEventCreateWithFlags
                    0.00%  14.822us        18     823ns     653ns  2.0330us  cudaEventDestroy
                    0.00%  14.328us         4  3.5820us  1.6800us  5.7090us  cudaDeviceSynchronize
                    0.00%  8.3840us         3  2.7940us  2.3520us  3.5860us  cuInit
                    0.00%  8.3500us        14     596ns     270ns  1.8430us  cudaDeviceGetAttribute
                    0.00%  5.3620us         1  5.3620us  5.3620us  5.3620us  cuDeviceGetPCIBusId
                    0.00%  2.5860us         6     431ns     192ns  1.2190us  cuDeviceGetCount
                    0.00%  2.2880us         5     457ns     245ns  1.1280us  cuDeviceGet
                    0.00%  1.6140us         3     538ns     342ns     875ns  cuDriverGetVersion
                    0.00%  1.2980us         1  1.2980us  1.2980us  1.2980us  cudaGetSymbolAddress
                    0.00%  1.2340us         4     308ns     274ns     377ns  cuDeviceGetUuid
                    0.00%     640ns         2     320ns     255ns     385ns  cudaPeekAtLastError
                    0.00%     383ns         1     383ns     383ns     383ns  cudaGetDeviceCount

==14810== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       3  4.0000KB  4.0000KB  4.0000KB  12.00000KB  10.01600us  Host To Device
```