Assignment IV: Advanced CUDA

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

# Exercise 1 – Thread Scheduling and Execution Efficiency

## 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.

## 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.

## 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.

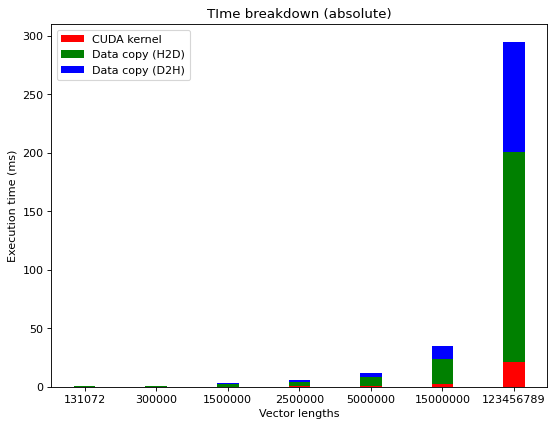
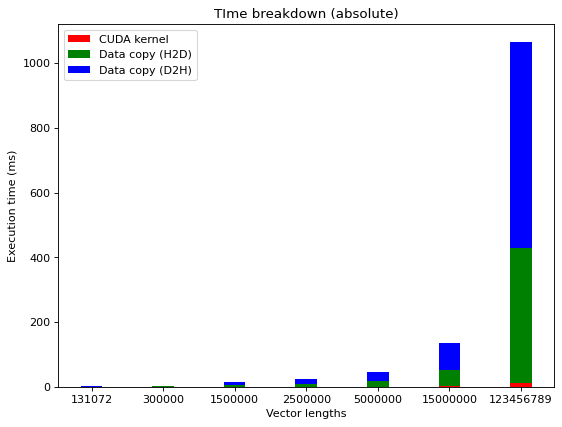
* Total number of warps with control divergence: 400 + 38 - 1 = 437.

# Exercise 2 – CUDA Streams

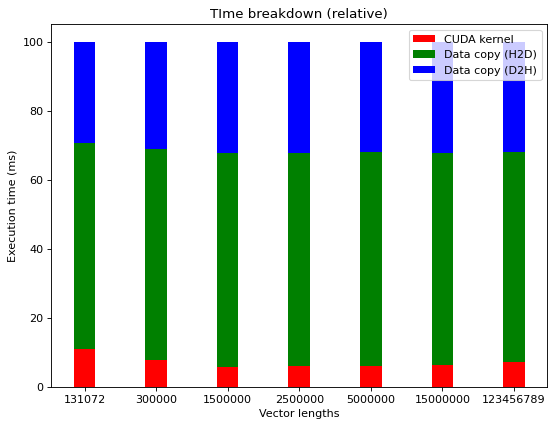
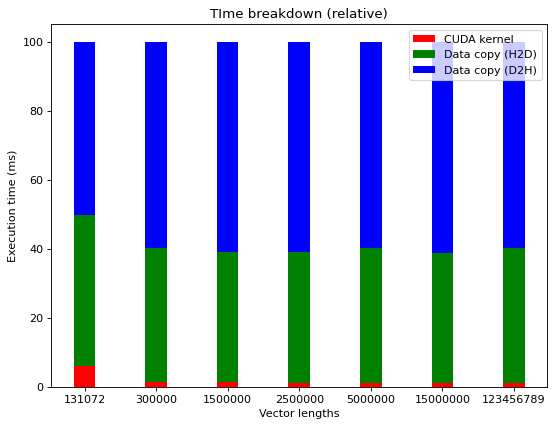
## Compared to the non-streamed vector addition, what performance gain do you get? Present in a plot ( you may include comparison at different vector length)

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).

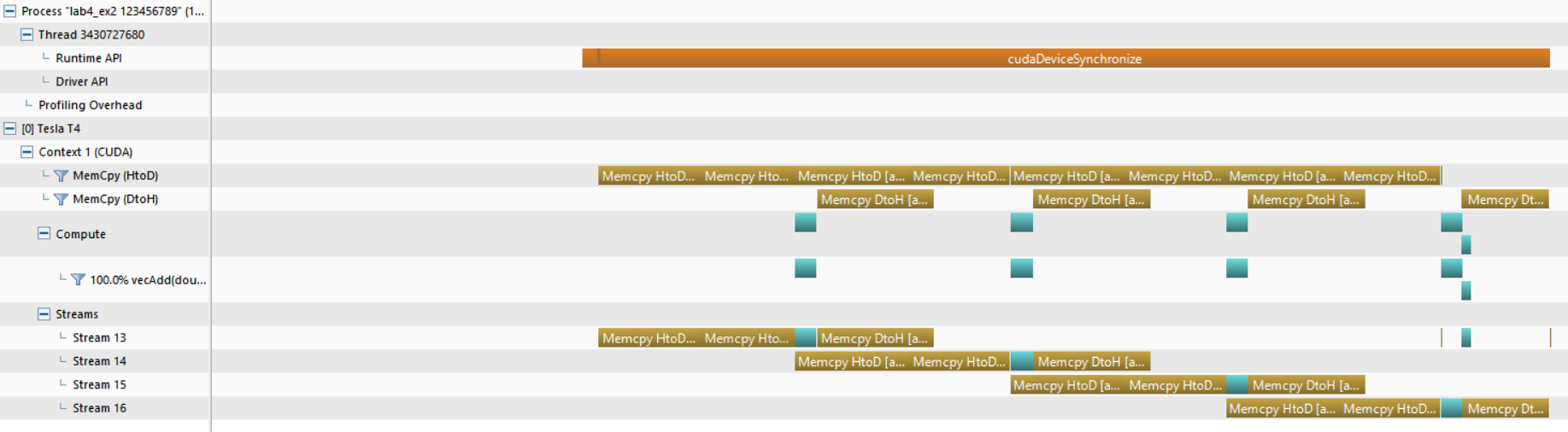
## Use nvprof to collect traces and the NVIDIA Visual Profiler (nvvp) to visualize the overlap of communication and computation. To use nvvp, you can check [Tutorial: NVVP - Visualize nvprof Traces](https://canvas.kth.se/courses/36161/pages/tutorial-nvvp-visualize-nvprof-traces)

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:

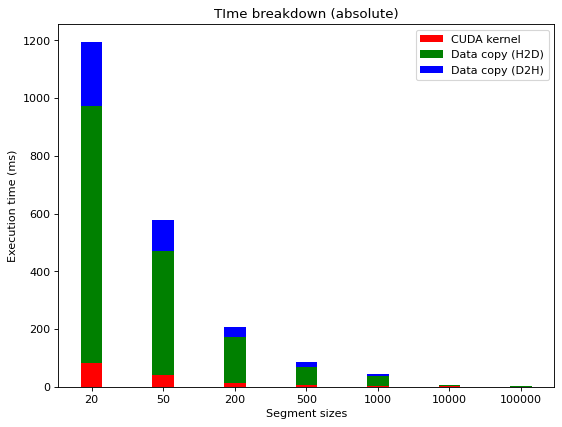
* When installing the CUDA toolkit, it has to be installed in the “expert mode” instead of the “express mode”. There, you need to disselect multiple features like the Visual Studio Integration (and according to different posts maybe some others but this seemed to be sufficient).
* The Visual Profiler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

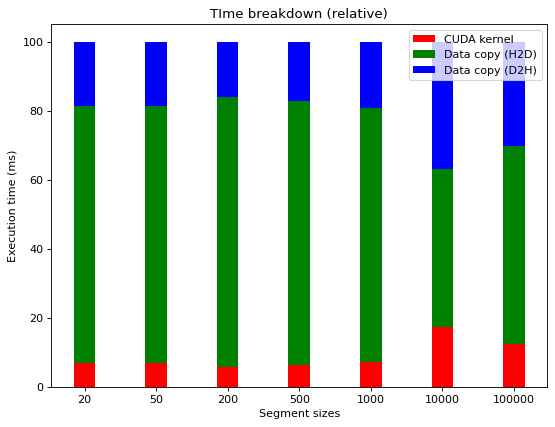
nvvp -vm "C:\Program Files\Java\jdk1.8.0\_351\jre\bin\java"

The -vm option may be required in some cases.

## 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:





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

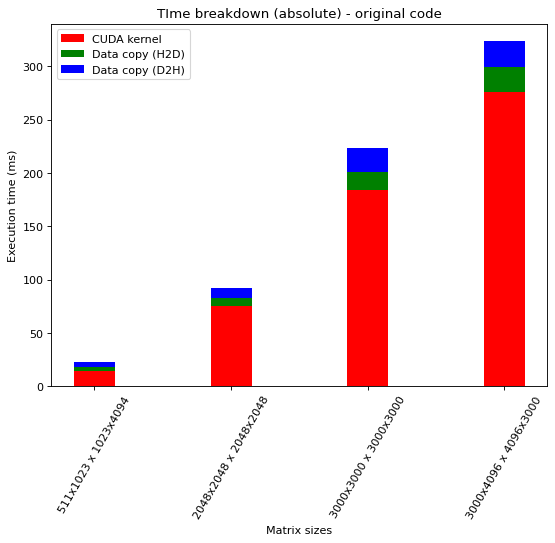
## What are the differences between pageable memory and pinned memory, what are the tradeoffs?

Inspired by the lecture slides and this [post](https://medium.com/analytics-vidhya/cuda-memory-model-823f02cef0bf): 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.

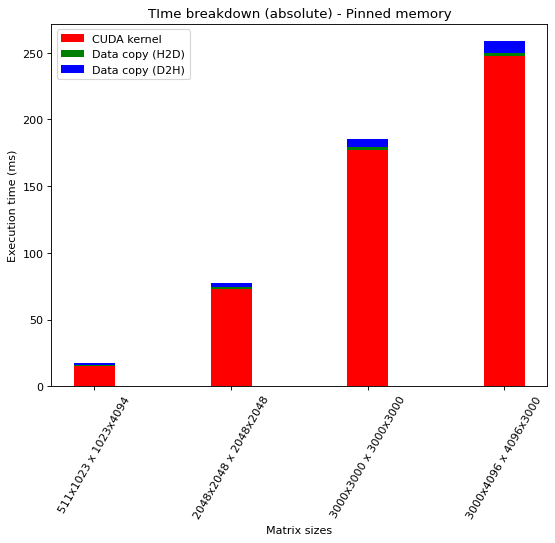
## Compare the profiling results between your original code and the new version using pinned memory. Do you see any difference in terms of the breakdown of execution time after changing to pinned memory?

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.

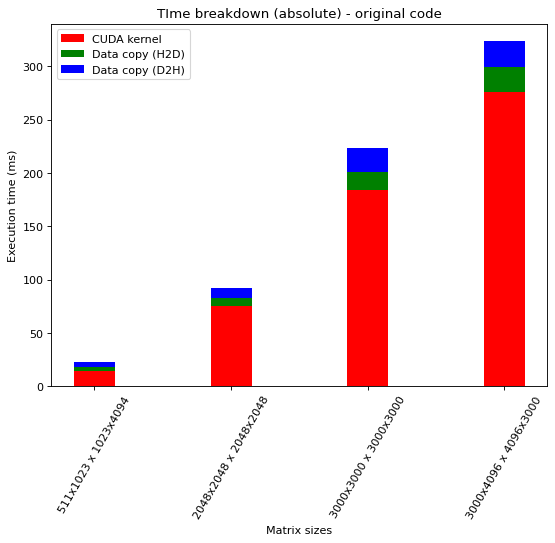
## What is a managed memory? What are the implications of using managed memory?

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.

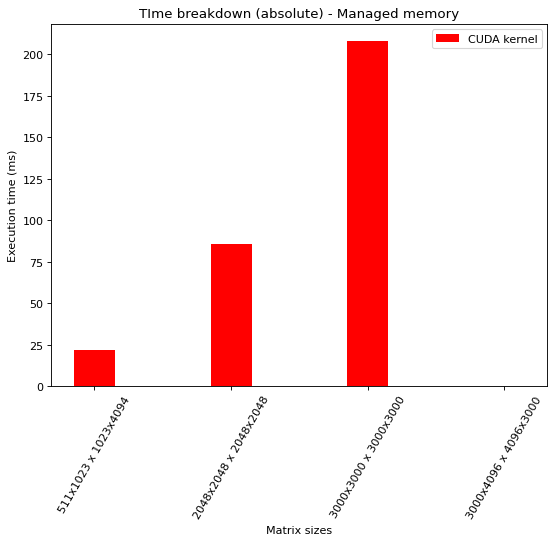
## Compare the profiling results between your original code and the new version using managed memory. What do you observe in the profiling results?

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](https://forums.developer.nvidia.com/t/nvprof-error-code-139-but-memcheck-ok/50329/14) and other posts.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

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:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Page faults for 2048x2048 2048x2408 matrices:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

# Exercise 4 – Heat Equation with using NVIDIA libraries

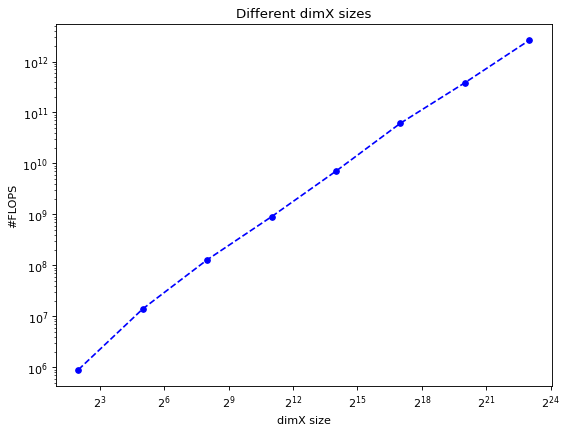
## Run the program with different dimX values. For each one, approximate the FLOPS (floating-point operation per second) achieved in computing the SMPV (sparse matrix multiplication). Report FLOPS at different input sizes in a FLOPS. What do you see compared to the peak throughput you report in Lab2?

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.

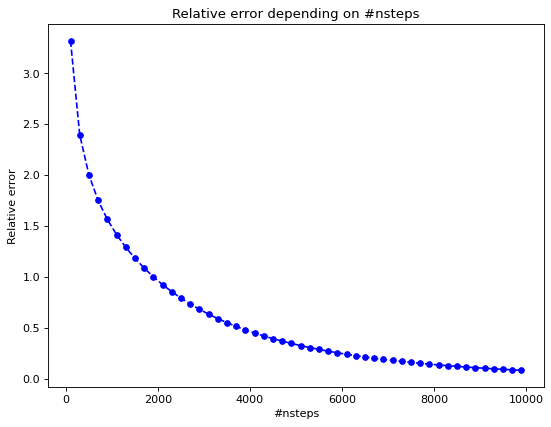
For the number of FLOPS comparison with different dimXs, I fixed the nsteps to 100 as it was the lowest suggest size in the next question:



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.

## 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.

## 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:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

With prefetching:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung