Assignment IV: Advanced CUDA

Franz Kaschner

08.01.2023

Link to Github repository: https://github.com/TheKastenkarl/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.**

Size of block array:

- In x-direction: ceil $\left(\frac{800}{16}\right) = 50$
- In y-direction: ceil $\left(\frac{600}{16}\right) = 38$

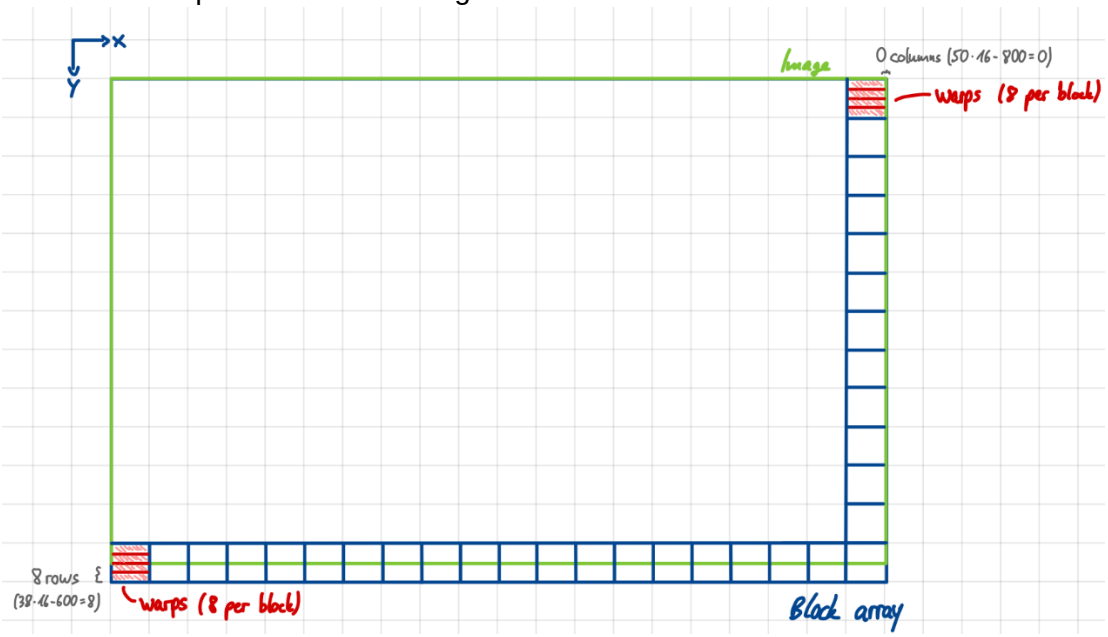
Total number of warps:

$$\text{warps} = \text{warps\_per\_block} \cdot \text{blocks} = \text{ceil}\left(\frac{\text{threads\_per\_block}}{\text{warp\_size}}\right) \cdot \text{blocks} =$$

$$= \text{ceil}\left(\frac{16 \cdot 16}{32}\right) \cdot 50 \cdot 38 = 8 \cdot 50 \cdot 38 = 15200$$

Explanation of how thread blocks are divided into warps: Link (TLDR: Threads are numbered in order within blocks so that `threadIdx.x` varies the fastest, then `threadIdx.y` the second fastest varying, and `threadIdx.z` the slowest varying.)

Number of warps with control divergence:



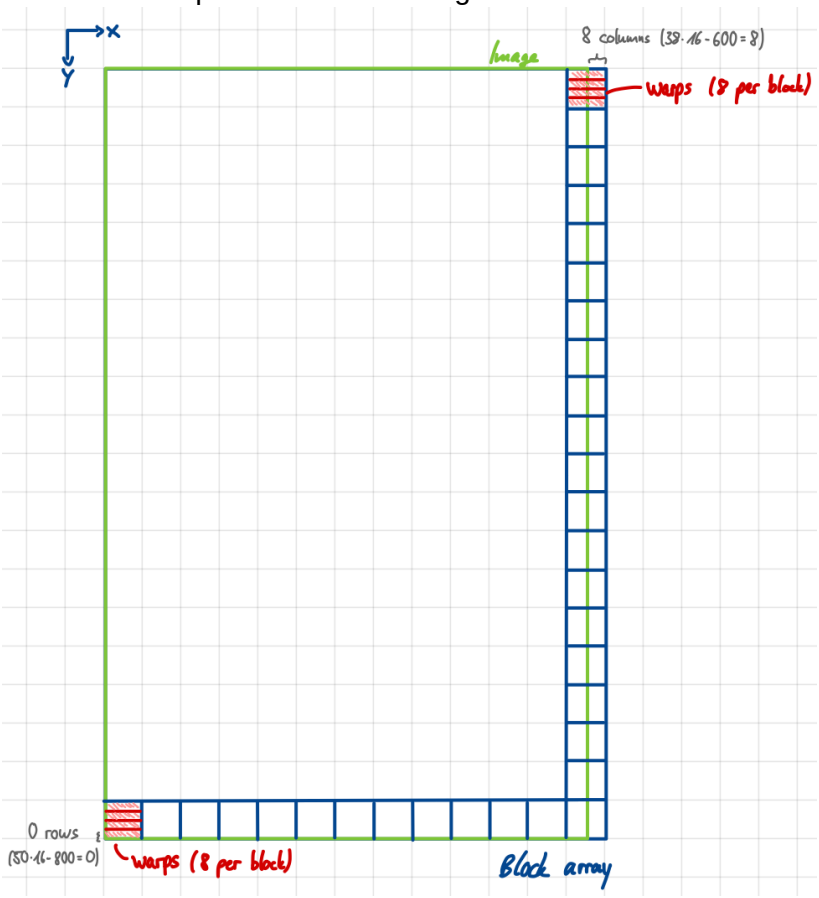
- In last row: No control divergences because a single warp either doubles pixel values in all threads or in no thread. → 0 warps
- In last column: No control divergences because the size of all threads in x-direction is equivalent to the size of the image. → 0 warps
- Total number of warps with control divergence: 0 warps

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

Size of block array:

- In x-direction: ceil $\left(\frac{600}{16}\right) = 38$
- In y-direction: ceil $\left(\frac{800}{16}\right) = 50$

Number of warps with control divergence:



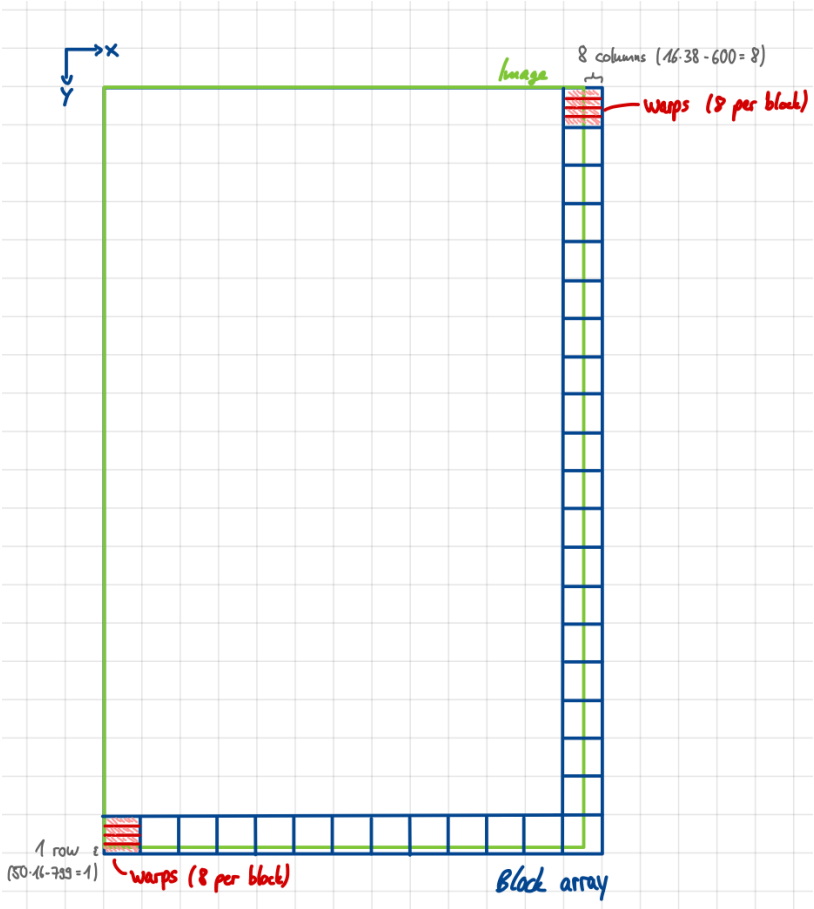
- In last row: No control divergences because the size of all threads in y-direction is equivalent to the size of the image. → 0 warps
- In last column: Control divergence in all warps because a single warp always includes threads that double pixel values and threads where this is not done. → 8 * 50 warps
- Total number of warps with control divergence: 400 warps

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

Size of block array:

- In x-direction: ceil $\left(\frac{600}{16}\right) = 38$
- In y-direction: ceil $\left(\frac{799}{16}\right) = 50$

Number of warps with control divergence:



- In last row: Control divergence in one warp per block because the warp which covers the last two rows of the thread block performs both pixel value doubling (second last row) and no pixel value doubling (last row). → 1 * 38 warps
- In last column: Control divergence in all warps because a single warp always includes threads that double pixel values and threads where this is not done. → 8 * 50 warps
- We counted one warp in the thread block in the bottom right corner twice, so we have to subtract one warp.
- Total number of warps with control divergence: 38 + 400 - 1 = 437 warps

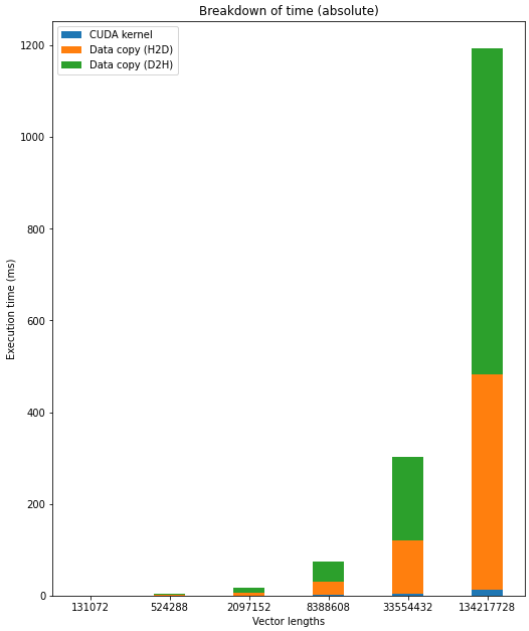
Exercise 2 - CUDA Streams

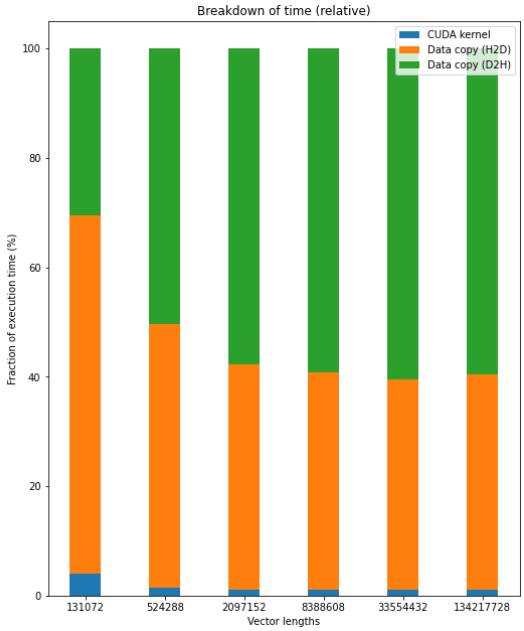
1. **Compared to the non-streamed vector addition, what performance gain do you get? Present in a plot (you may include comparison at different vector length).**

I use nvprof for measuring the times as it is simpler and more accurate (https://stackoverflow.com/questions/30371030/understanding-cuda-profiler-output-nvprof).

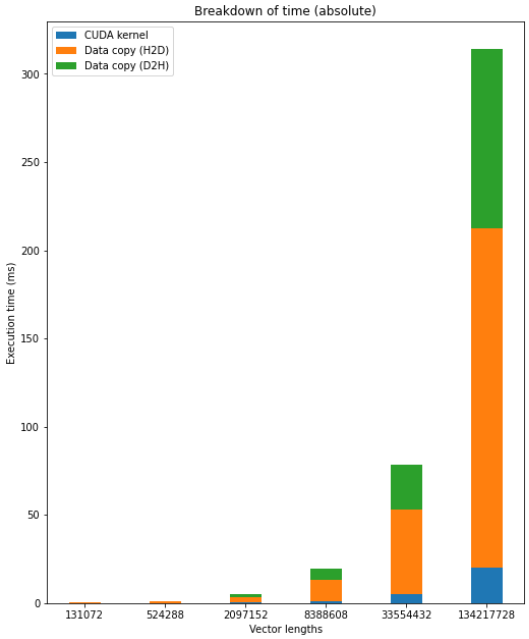
Used segment size: `S_seg = inputLength / NUMSTREAMS`
Command: `nvprof ./lab4_ex2 <vector-length>`

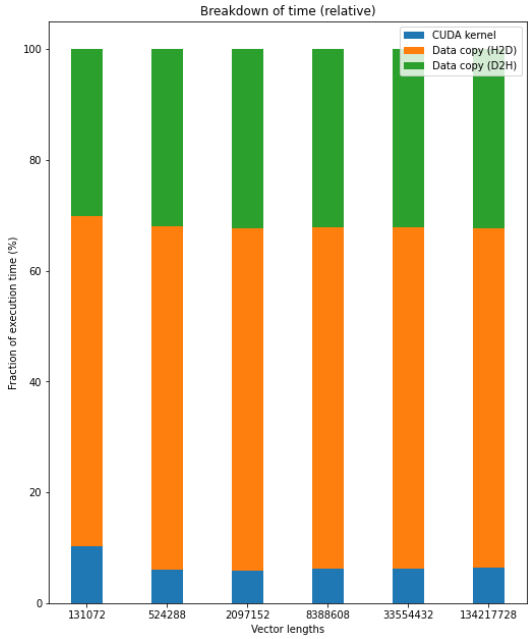
Non-streamed vector addition:





Streamed vector addition:

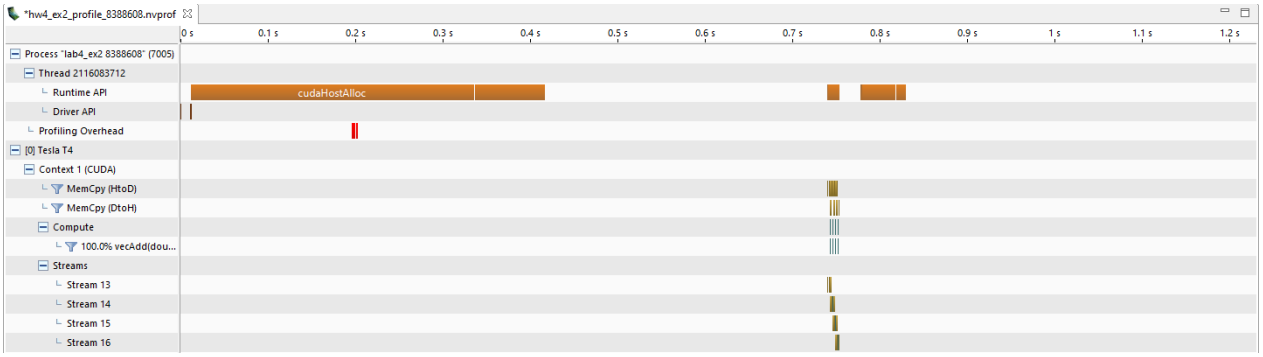




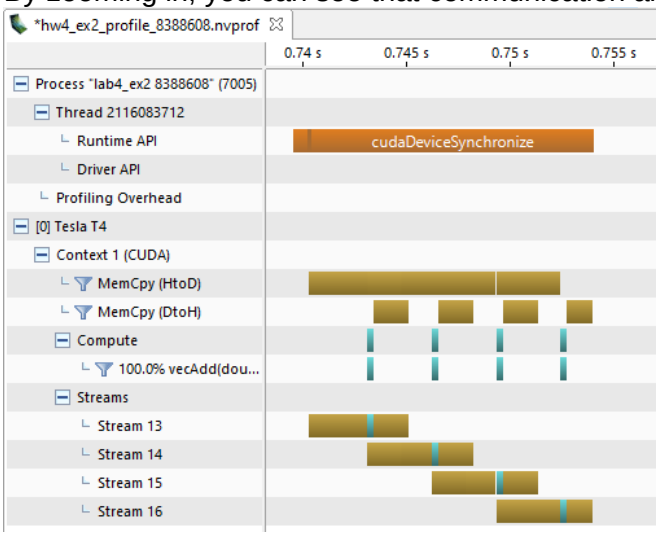
As you can see, the streamed vector addition is much faster than the non-streamed vector addition, especially for larger vector sizes.

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.**

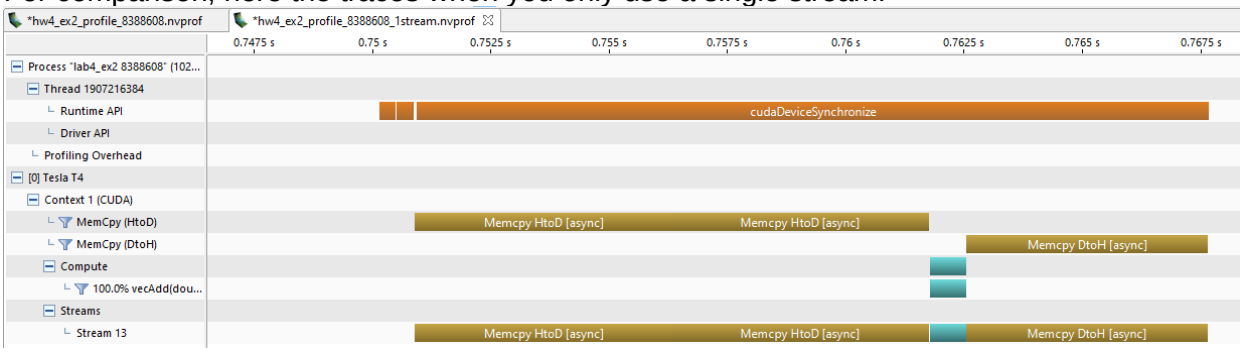
Used segment size: `S_seg = inputLength / NUMSTREAMS`
Used vector length: 8388608
Command: `nvprof --output-profile hw4_ex2_profile_8388608.nvprof -f ./lab4_ex2 8388608`



By zooming in, you can see that communication and computation overlap:

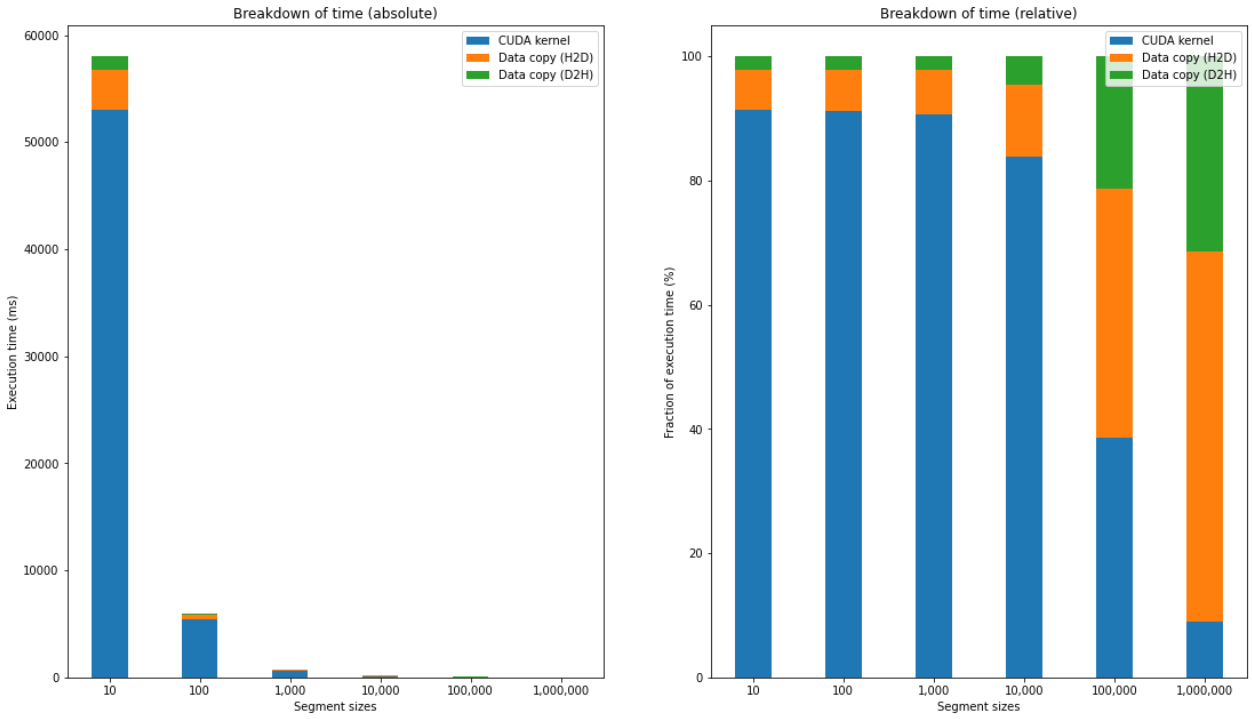


For comparison, here the traces when you only use a single stream:



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).

Used vector length: 8388608
Command: `nvprof --output-profile hw4_ex2_profile_8388608.nvprof -f ./lab4_ex2 8388608`



You can clearly see that the segment size should not be chosen too small. This is because with small segment sizes you get a lot of overhead as a result of the large number of CUDA API calls (like `cudaMemcpyAsync`).

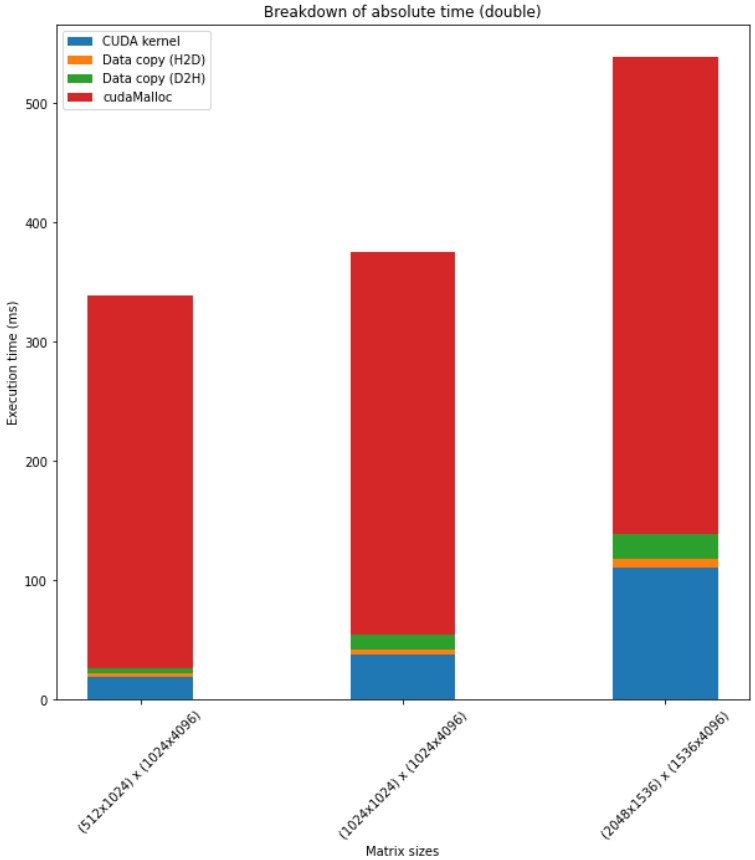
Exercise 3 - Pinned Memory and Unified Memory

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

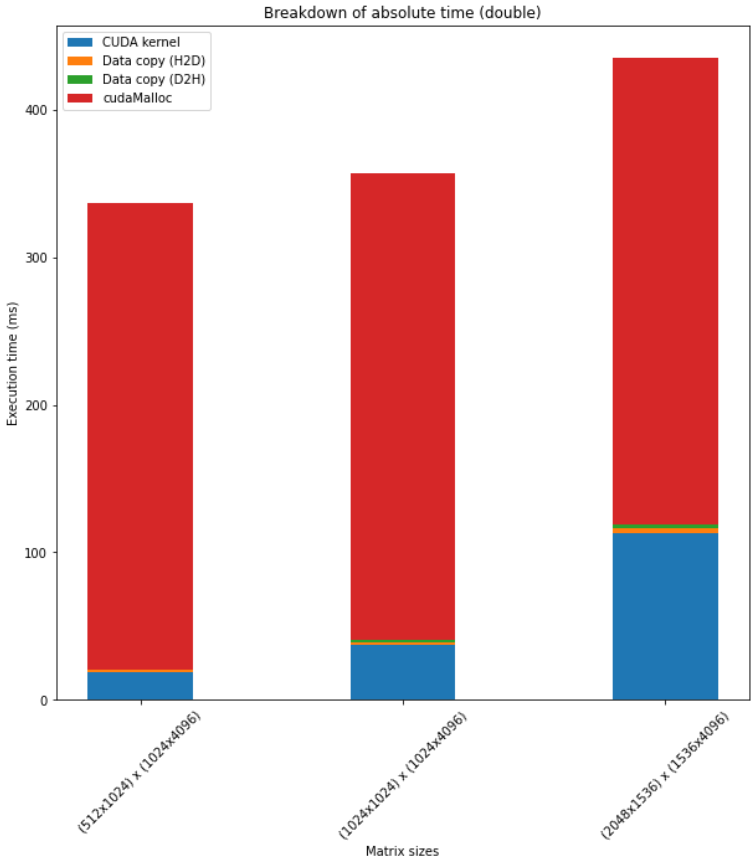
Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out. Pageable memory can be paged out. If a source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory (→ extra overhead). However, pinned memory is a limited resource and it is much more expensive to allocate and deallocate. [Source: Lecture slides]

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?**

Original code:



New code using pinned memory:



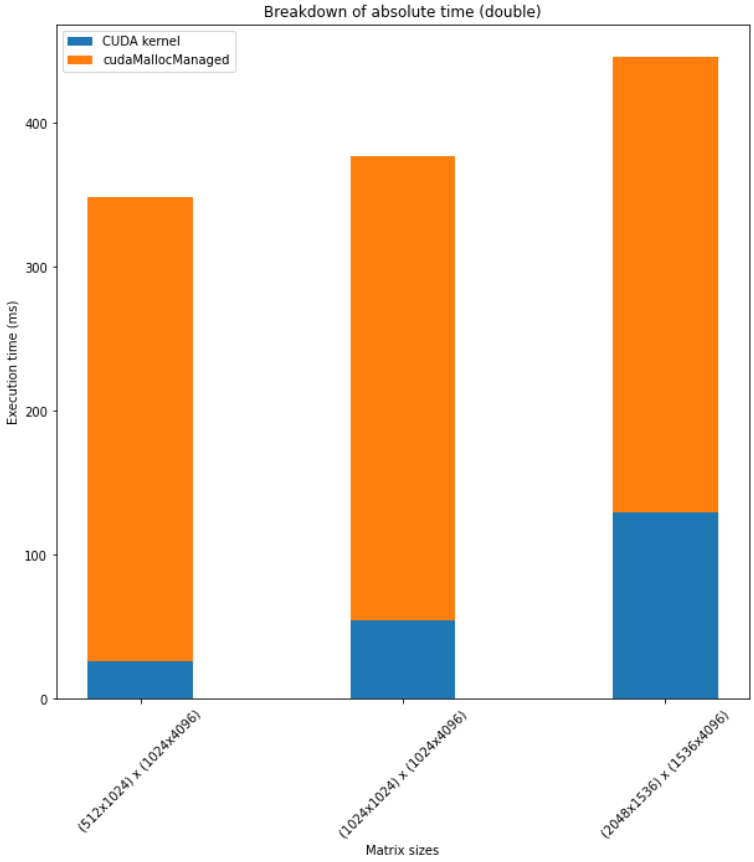
The memory copying operations are much faster in the new code using pinned memory (device to host approx. by a factor of 8 and host to device approx. by a factor of 2). I would have expected that memory allocation takes longer using pinned memory, but at least in this example this is not the case. The required times for memory allocation are really similar for the two smaller matrix sizes, and for the larger matrix sizes the memory allocation using pinned memory is even faster by approx. 20%.

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

Managed memory (or Unified Memory) is a single memory address space accessible from any processor in a system. Hence, allocated data can be read or written from code running on either CPUs or GPUs. The CUDA system software takes care of migrating memory pages to the memory of the accessing processor. [Source: Lecture slides]

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

New code using Unified Memory:

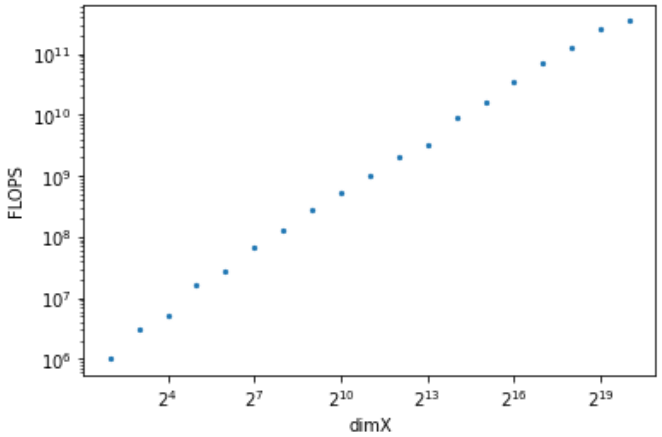


In this Nvidia blogpost it is explained that “The cost of the migrations is included in the kernel run time”. Hence, the execution time of the kernel in the new code is a bit longer than the execution time of the kernel in the original code. However, in total the code using Unified Memory is a little bit faster compared with the original code. Especially, for the largest matrix size the time for memory allocation is lower. In the “Unified Memory profiling result” from nvprof you can see that we have quite a lot of Gpu page fault groups. Thus, optimization regarding this aspect would be helpful. See this Nvidia blogpost for tips on doing that.

Exercise 4 - Heat Equation with using NVIDIA libraries

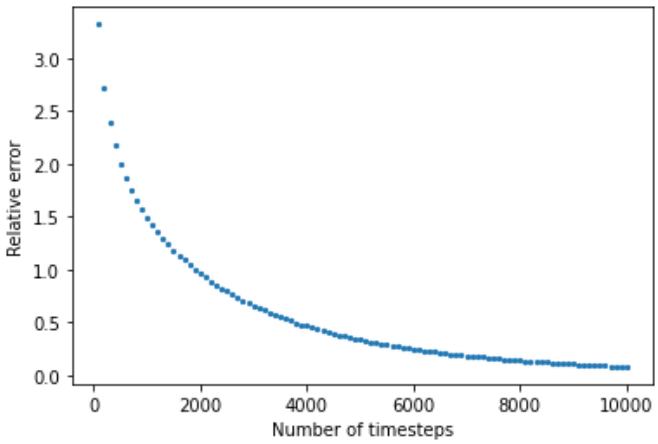
1. **Run the program with different dimX values. For each one, approximate the FLOPS (floating-point operation per second) achieved in computing the SMPV (sparse matrix multiplication). Report FLOPS at different input sizes in a FLOPS. What do you see compared to the peak throughput you report in Lab2?**

To calculate the FLOPS I first measure the time required to compute the SMPV (sum of all SMPVs over all time steps). Then, the following formula is used to calculate the FLOPS: `num_timesteps * (3 * dimX - 6) / time`
This formula can be used because for every nonzero element we have one multiplication and there are `(3 * dimX - 6)` nonzero elements.



For larger values of dimX, we get a higher number of FLOPS. We don't reach the peak throughput of 4.07 * 10^{12} FLOPS of the Nvidia Tesla T4.

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?**



The relative error gets smaller with a higher number of timesteps.

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]**

Command: `!nvprof ./lab4-ex4 128 1000`

With prefetching the time for “Initializing the sparse matrix on the host.” is only 2µs. Without prefetching it is 85µs. However, the time for prefetching is 64µs. Hence, we only have a small speedup. A reason for the speedup is that we don’t have any CPU Page faults when we use prefetching.