Applied GPU Programming - Assignment III

DD2360 HT20

Pontus Asp

November 25, 2020

1 Git repository

I uploaded my git repository to GitHub. I use the same git repository for the entire course but the folder structure requested is still followed under the root folder. I also have 2 extra directories, one for this report and one where I have code from tutorials.

Here is the link to my git repository:
`https://github.com/pontusasp/kth-dd2360/tree/master/Assignment_3`

2 Exercise 1

In the code we were given, the kernel functions are called with the mappings like: `kernel<<<grid, block>>>(...)`.

The `grid` variable is defined as a 2D grid where each axis corresponds to the number of threads in that "direction", in the x-axis we set the grid to have the same number of threads as the width of the input image, and in the y-axis we set the grid to have the same number of threads as the height of the input image.

The grid is also divided into blocks where a number of threads will be grouped together, and the `block` variable defines the dimension of this grouping. If we for instance have a grid with the size of 2000x1000 and a block size of 50x100 then the grid will get a structure of blocks like 40x100. All threads in a block will be executed on the same SM, and each SM also has a shared memory that each block can allocate memory on, which we utilize in the exercise.

Theoretically we should be able to increase performance with this shared memory since it is stored closer to the SM than the global memory that we would otherwise need to use. We also avoid using up more memory bandwidth to the global memory by utilizing the shared memory.

After we first implemented shared memory our image output looked like a grid. The explanation for this is that when we first added our shared memory we only copied the pixels "in range" of our block, i.e. a 16x16 grid of pixels. but the problem is that the filter we wanted to apply relied on looking on nearby pixels to decide the new color of the pixel we applied the filter on. So when we applied the filter on the border of our block we accessed pixels that we did not copy - so those pixels looked black to the filter and got applied to the pixel we were currently updating. I fixed this by making the last two threads in each row and column copy an extra pixel and placing it 2 pixels away in their individual direction. But, this does have an edge case where the

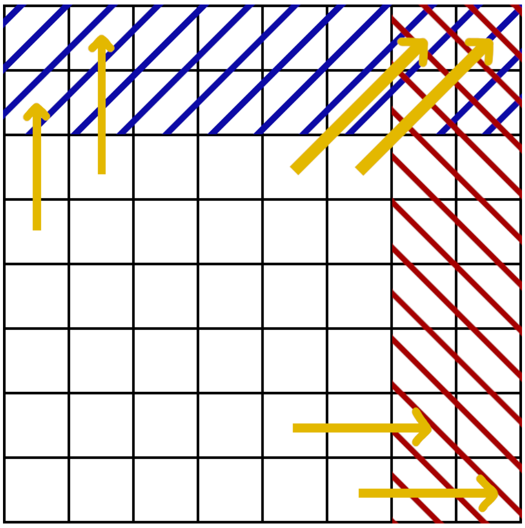

Figure 1: Visual example of copying pixels to shared memory.

pixels will not get copied in a 2x2 square where the two other problematic rows and columns intersect. I solved this by having the threads handling these intersecting rows and columns of pixels also copy a third pixel 2 steps diagonally from their own pixel. See Figure 1 for a visual example.

When running the program with all the different images it seems to be the case that the time it takes to finish correlates linearly to the size of the images, which, is not weird since the work needed to be done also grows linearly with the size of the image.

3 Exercise 2

The difference between pageable memory and pinned memory is essentially that the pinned memory is always loaded in physical memory. The good thing about pinned memory is that since it will always be loaded in the physical memory it will always be ready to be used. Which removes the delay of needing to move it back to memory if it is not present. But, with pros usually cons come as well and this is no exception. The bad thing about pinned memory is that we unfortunately do not have infinite memory, so when we use pinned memory the available memory decreases - it is a limited resource. That is what pageable memory solves. Pageable memory is not restricted to the size of the memory but can be unloaded and loaded into the physical memory, but at the cost of being slower if the memory asked for is not currently in physical memory.

When changing from pageable memory to pinned memory the execution time for the GPU particle simulation almost got cut in half, which is a 50% increase - that is, really good! When testing this I used a block size of 16, 1000 iterations and $100k$ particles.

Managed memory is another kind of memory in the cuda API. Managed memory is memory that is accessible by both the GPU and CPU, this works by the memory being copied over from the CPU or the GPU memory automatically - removing work for the programmer, but also abstracting away some of the lower level control that in some cases can be useful as well.

I am not using Tegner nor a lab computer, but I am assuming that the use of managed memory will implicitly copy the memory before a kernel launch because the resources are shared - so to not waste the limited space on the GPU, the allocated memory will only be present on the GPU when really needed, i.e. when a kernel needs to run. So therefore it will be copied when launching a kernel.

4 Exercise 3

The advantages of using CUDA streams and asynchronous memory copying is that the GPU can perform operations at the same time as it is transferring memory, but so far we have not been utilizing this fact. What we have done is copy the data to the GPU, and when it is finished we have launched our kernel, and then we wait for all threads to finish before copying the data back. What we can do with CUDA streams and async memory copies is we can asynchronously start copying batches of data, and instruct cuda to launch a kernel when the memory is finished copying, and then copy the data back. Since all this is asynchronous we can do this for a number of streams basically in paralell (sequentially on the CPU side but paralell on the GPU). What then happens is that the GPU will start copying part of the full data and then start working on that part while at the same time transferring the next part. So by doing this we can utilize that the GPU can transfer data and perform work at the same time, and we could also perform work on the CPU while the memory is asynchronously transferred back and before trying to access the memory we synchronize the CPU with the GPU to make sure all the streams are done executing.

When I implemented the CUDA stream the performance improvement was very big. In my tests the performance improved with approximately 50%, which I would consider a great improvement.

Disclaimer: I was not sure if batch size was meant to be the number of streams or the size of data per stream, in my case I picked that batch size is the size of the data per stream.

The impact on the performance of the batch size did not change too much in my testings, unless it deviated a lot from the size of the data. In my testings the optimal batch size seemed to be at around $20k$ when doing a simulation on $100k$ particles. When changing the batch size to $50k$ the time increased with 3%, there was approximately the same result when changing the batch size to $10k$. However if I decreased the batch size to 100 the time increased with 1700%. Which shows that there can definetily be too many streams with too little data for the problem, considering that if we set the batch size equal to the size of the data (so we only use 1 stream) then we simply get approximately the same result as not using any other CUDA stream other than the default one, i.e. basically the worst case in low amount of streams.

5 Bonus Exercise

The matrix size has to be a multiple of 16 to make the code a bit more simple. It specifically becomes simpler in the `shared_sgemm_kernel` function since allocating dynamically sized shared data is more complex than creating statically sized shared data in CUDA. It also makes it easier to partition the matrices since we will always have sub-matrices with the dimensions `A_(m x TILE_SIZE), B_(TILE_SIZE x n)`. If these matrices were not static in one dimension we would not be able to utilize the `#pragma unroll` directive either.

The `#pragma unroll` directive can potentially improve performance by optimizing away the `for-loop` at compile time. Since we can get rid of the loop we can therefore avoid branching which can speed the computation up. This works because the loop is static in how many times it will loop over the data.

Continuing talking about the `shared_sgemm_kernel` function, there are two `__syncthreads` calls in the loop. They both play an important role to make the code reliable (or even work at all). The first call synchronizes the threads after they copy over the current matrix partition (every thread copies one value from each matrix), since each thread will need to cross-access data that the other threads will copy over as well. They need to access this since each thread will need to access the entire row and column of the matrices B and A when calculating the value for the output matrix C in a specific position. The second synchronization happens after the calculation

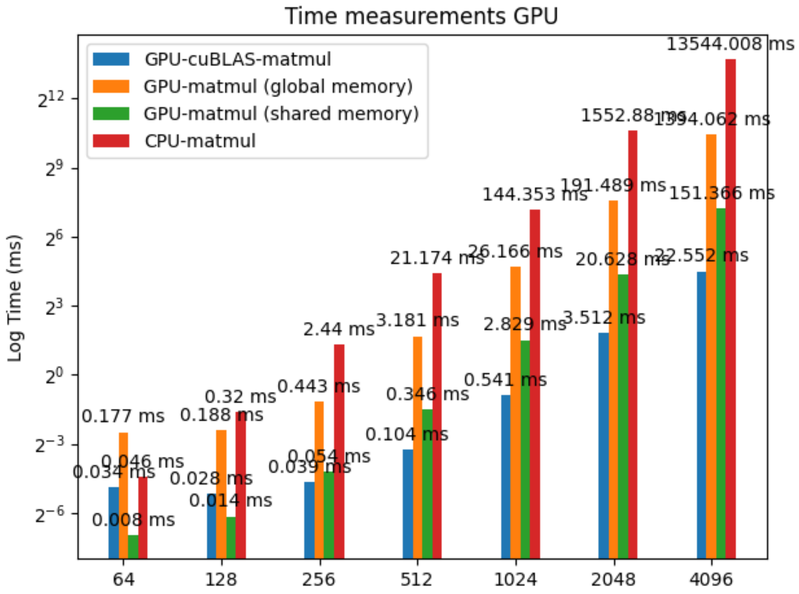

Figure 2: Graph of the different calculation times for different matrix sizes.

of the value for C, this is because we need to make sure all the threads are finished using the shared data before going over the loop again. Since when we start the loop again we will start overwriting the shared memory, and if some threads are not done using the shared memory yet we will not get the expected result.

When calling `cublas_sgemm` we were instructured to compute $C = BA$ instead of $C = AB$. This has to do with the fact that cuBLAS is a port of another library called BLAS. The original BLAS library was written in FORTRAN, which stores multi-dimensional arrays differently than C/C++ does. However cuBLAS could of course change the way this works in the port to C/C++ but they decided to stick to the same order the BLAS library uses for compability reasons. Now, why does this matter to us? Well FORTRAN uses column-major storage while C/C++ uses row-major storage. So if we want to send our matrices to cuBLAS we would have to transpose both of our matrices to make it work, but because of how matrix multiplication works we can instead swap places of the two matrices and it will give us the same effect as transposing them both, this is both easier and faster for us to do, which is why we did it.

I made a plot of all the different times recorded by running the program with the flag -v, with the sizes varying from 64 up until 4096. The GPU alone had a very steep curve in execution time so I decided to plot the graph with logarithmic time in base 2, since we doubled the matrix sizes with 2 each time - i.e. our X-axis is in base 2 as well. By doing so the CPU time also fit well into the graph. Looking at the graph it now looks linear, but we do see that the bars increase faster in the Y-axis than in the X-axis, so if we would look at this data in linear scales then they would actually be growing exponentially.

In general the way the time benchmark is implemented in this exercise is not optimal because it requires some overhead, which gets included in the time measurement. For instance, if we measure the time it takes between copying data to the GPU from the host, then calling the kernel, then synchronizing, and finally copying data back to the CPU - then we will loose some time spent by the CPU as well, so it will look like the GPU was slower than it actually was. In this exercise the time difference should be very small though, so for our purposes of looking how long time the different methods of calculating matrix multiplications takes it is OK, since we are doing it for learning purposes.