UPPSALA
UNIVERSITET

Assignment 2:
Programming in CUDA
Accelerator-based Programming

Oskar Tegby

October 2022

1 Introduction

This assignment studies the computational properties of `CUDA` implementations of matrix-vector and matrix-matrix multiplication. The former is parallelized only over the rows, and the latter over both rows and columns, which is assumed to be faster than only parallelizing over the rows in the latter case. Notably, these operations are completely parallelizable since they are fully data parallel. Thus, we will dedicate this report almost exclusively to reporting and discussing the achieved bandwidth of the different implementations. Lastly, all tests are run on a `Nvidia Tesla T4` graphics card running on the `Snowy` server on the `UPPMAX` cluster.

The veracity of the code is tested by comparing the results for a matrix, A, and a vector, x, with strictly increasing values, since that is fully asymmetric. That is, we tested the code with e.g.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix},$$

as input values, which gives us the output

$$b = \begin{bmatrix} 30 & 70 & 110 & 150 \end{bmatrix}.$$

This indicates that the implementation is indeed correct.

Furthermore, all results are computed as the averages of 20 tests each run with 20 repeats to decrease the influence of computational noise, which allows us to rely on smaller details in the obtained graphs. Thus, it should not be seen as an explanation for unexpected behavior, as is commonly the case.

2 Tasks

2.1 Task 1

2.1.1 Part A

Figure 1 shows the throughput of the single-precision matrix-vector multiplication run with N = M ranging from 104 to 9576 obtained by running the code with `block_size` = 512. The throughput is computed as

$$(MN + N + M) \cdot \texttt{sizeof(float)} \cdot 10^{-9}/t_{\text{best}}.$$

2.1.2 Part B

Figure 1a shows a linear increase in throughput with the tensor sizes until about $N = M = 3000$. Afterwards, the throughput stagnates between above 200 and 240 GB/s. The reason for which is that we can no longer fit the entire matrix in our local cache, which means that we have to fetch the data from the graphics memory, which has a bandwidth of 320 GB/s on the Nvidia T4 graphics card. This constitutes a theoretical upper bound which we should not expect to reach.

However, we should neither expect to miss out on 80 GB/s throughput with such ideal parallelism. Given that the code is coalesced, since each thread runs the `for` loop with jumps of size N, this comes as a surprise and is a mystery to the author. Here, we must recall that we have implemented a column-major matrix layout, which means that every column is contiguously stored in memory, and each row requires a jump of size N to reach the next element. Hence, instead of the first thread accessing memory loading the first row, and the other threads using the other elements of that row, the same happens but for columns instead of rows. This causes the implementation to look uncoalesced at a first glance.

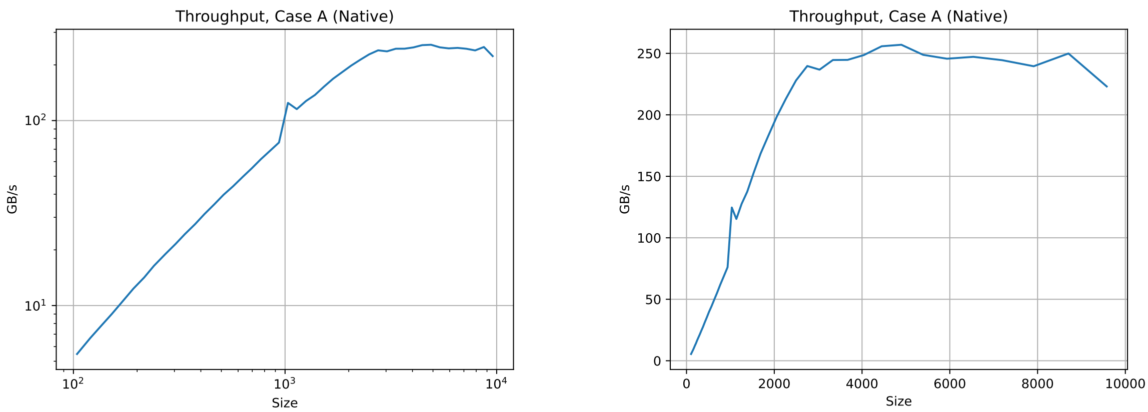
The choice of the block size 128 comes from each warp being 32 threads large and the fact that they are running on the same streaming multiprocessor (SM): We want to run several warps on each SM since we have to keep the it busy. That is, if the threads in one warp need to wait for data, then they can be swapped out to another warp which has fetched its data. At the same time, we want to make sure that the work is distributed among all SMs, so we do not want to run too many warps on each SM. However, since the maximum block size is 1028, this would only amount to 32 warps, which is a lot less than the amount of available parallel work.

The choice of the grid size as

$$\texttt{n\_blocks} = (N \cdot M + \texttt{block\_size} - 1)/\texttt{block\_size}$$

comes from dividing the number of elements from the matrix by the chosen `block_size`, and rounding up by adding $\texttt{block\_size} - 1$ in the numerator. This splits up the work into `n_blocks` blocks of work, each with a block size of `block_size`. Thus, we have `n_blocks` blocks done in parallel, and each of those are worked on in parallel since we split the block size into warps, which are run in parallel on each SM.

From these settings we expect a speedup of `n_blocks` as that is the number of blocks computed in parallel. However, the throughput that we get is limited by the bandwidth and latency of the device's memory hierarchy, as well as the memory properties of the implementation of the kernel, as discussed in the second paragraph of this subsection.



(a) The increase with loglog scale.

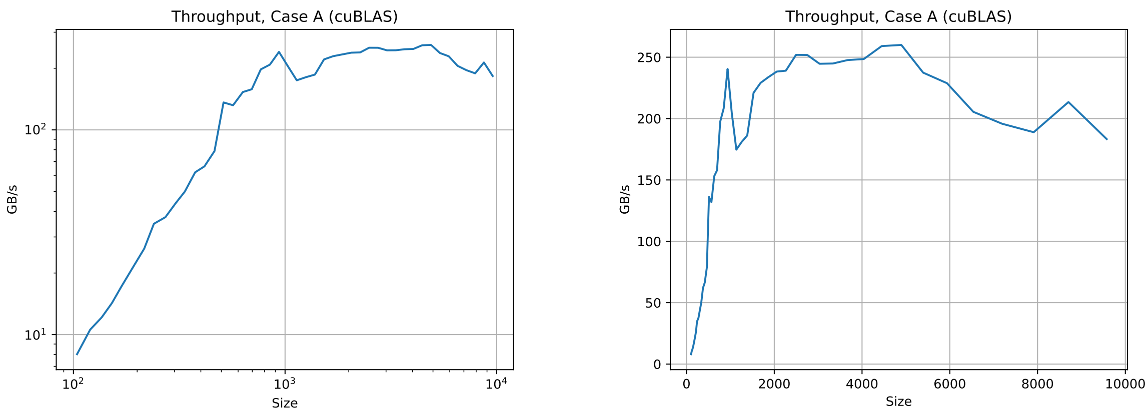
(b) The increase with linear scale.

Figure 1: Native matrix-vector multiplication with $M = N$ ranging from 100 to 10000.

2.2 Task 2

2.2.1 Part A

The results using cuBLAS are found in Figure 2. There, we observe roughly the same behavior as in Figure 1, aside from a much steeper increase in throughput and one occuring for much smaller problem sizes. That is presumably because cuBLAS is optimized for those scenarios than what any elementary native implementation could be. Furthermore, as expected, the peak throughput is also higher in this scenario. However, it is, notably, 70 GB/s short of the theoretical maximum, which still must be considered quite low given that it is only 80% of what the bandwidht allows. Importantly, this supports the notion that it was not an error in the memory coalescing that caused the relatively slow native implementation. The reason for these behaviors remains unknown to the author.



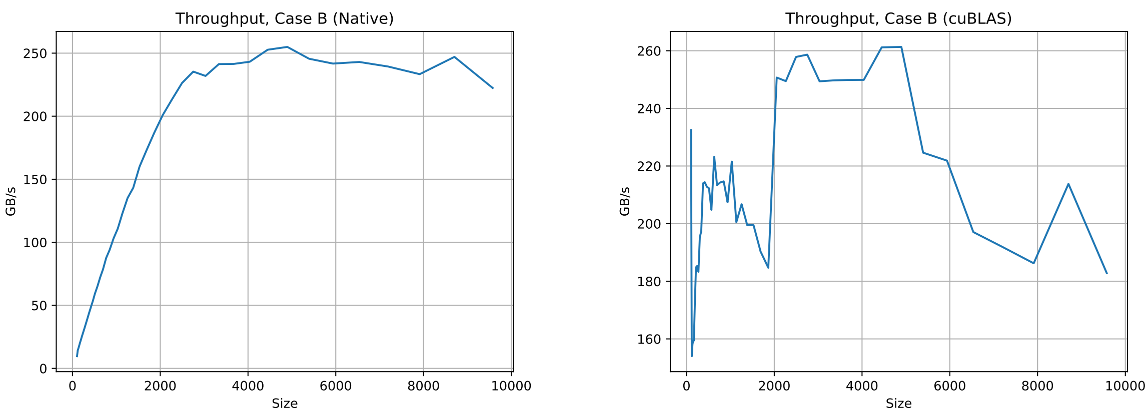
(a) The increase with loglog scale.

(b) The increase with linear scale.

Figure 2: cuBLAS matrix-vector multiplication with $M = N$ ranging from 100 to 10000.

2.2.2 Part B

As we see in Figure 3, the cuBLAS solver reaches a slightly higher peak performance, but performs worse as M increases. Notably, it performs much better for small problem sizes. The latter is most likely becuase of being better configured to deal with non-typical scenarios, but the former issue is surprising giving the high-performance nature of BLAS. No explanation to that behavior is known to the author.



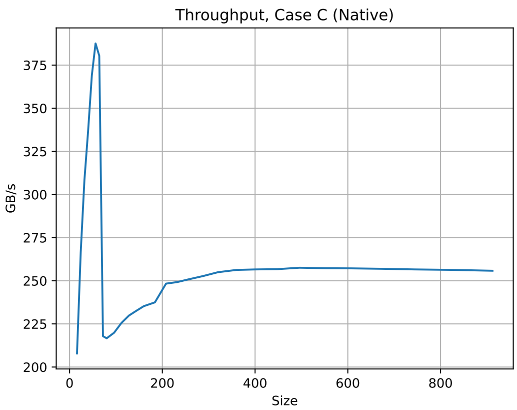
(a) The native solver.

(b) The cuBLAS solver.

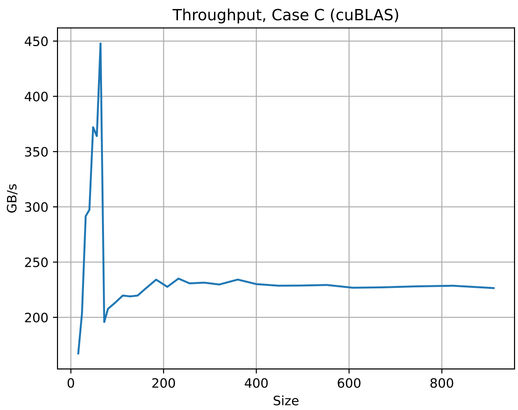
Figure 3: Matrix-vector multiplication with $N = 10000$ and M as in Task 1.

2.2.3 Part C

As seen in Figure 4, the throughput increases both for the native solver and for the cuBLAS solver as the number of rows increases. The reason for which is because we are now running the solvers with about 50% extra rows, compared to the maximum case in Part A and B, while increasing the length of the rows as we did in Part A. Here, we observe that the throughput increases linearly until we reach the capacity of the local cache, which has a bandwidht of about 800 GB/s. As seen in Assignment 1, we are bandwidth-bound, and so we observe the same phenomenom of being limited by the bandwidth to the main graphics memory, which is 320 GB/s. Again, we, surprisingly, observe that the cuBLAS implementation performs *worse* than the after the capacity of the local cache is reached. As in Part B, we offer no explanation to this behavior, but welcome one if it is available.



(a) The native solver.



(b) The cuBLAS solver.

Figure 4: The matrix-vector multiplication with $M = 15384$ and N ranging from 10 to 1000.

2.3 Task 3

Please note that this implementation only supports a multiple of the given block size as the size of the matrices. Thus, we will perform the performance experiments from $M = N = K$ between 100 and 5000, but with this constraint in order to avoid having to spend a large amount of time simply tending to edge cases which will not affect the performance in any noteworthy way.

Furthermore, please note that the main ideas for this code are greatly inspired, and partially borrowed from, the code found in NVIDIA's guide for CUDA programming called `CUDA C++ Programming Guide` (Section 3.2.3 named "Shared Memory", on pages 24-26) since it is a neat implementation using both shared memory for speed and a struct for tractability and readability.

2.4 Task 4