
UPPSALA
UNIVERSITET

# Accelerator-Based Programming

## Assignment 2
## Programming in CUDA

Jinglin Gao

September 27, 2023

# 1 Performance for matrix-vector product

To measure the achieved performance of the CUDA-implemented matrix-vector product, first, we analyze the memory complexity of the code. In the implementation, we need to access a $M \times N$ matrix and a $N$ vector, the output vector has a size of $M$. So in total, we have a $MN + M + N$ complexity.

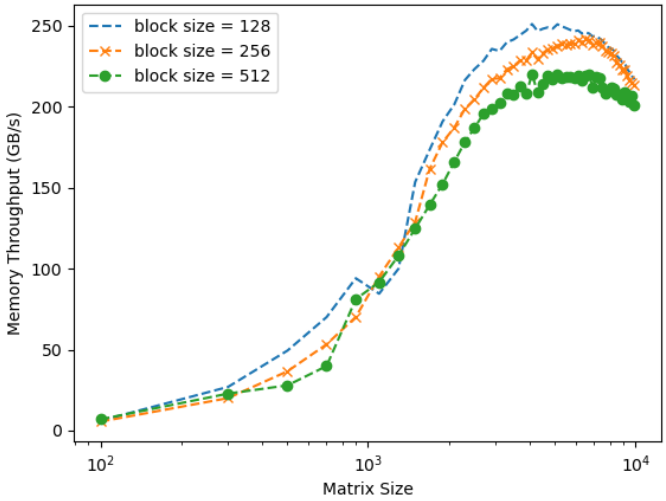


Figure 1: M = N between 100 and 10000

In this implementation, we should have $M$ parallelism since each element of the output vector can be calculated independently. Here our matrix is stored in column-major, which is preferred by GPU since GPU has a different way to access memory. All threads in the same wrap will access memory at the same time. Hence, we implement a reduction that happens at $y[row] += A[row + M * col] * x[col]$.

In the experiment, I used three different block sizes 128, 256, and 512, respectively. In the result graph, we can see that by increasing the size of the matrix, we get better performance. This is because we need to fill each block with enough work to keep the GPU working efficiently. Also, we can find out that there is a drop at around $M = N = 5000$, this is where our data can not fit into fast memory and start using RAM.

Different sizes of the thread blocks and the blocks per grid relate to our performance too. If the block dimensions are too small, there may not be enough threads to fully utilize the GPU's resources. This can result in poor performance. On the other hand, if the block dimensions are too large, there may be too many threads competing for resources, which can also result in poor performance. In our case, one SM has 64 FP32 AUs, which are able to execute 64 float32 operators each time. With a block size of 512, we probably don't have enough registers to do the calculation so we can not achieve good performance. In our case, the block size of 128 has the best performance.

## 2 cuBLAS Implementation

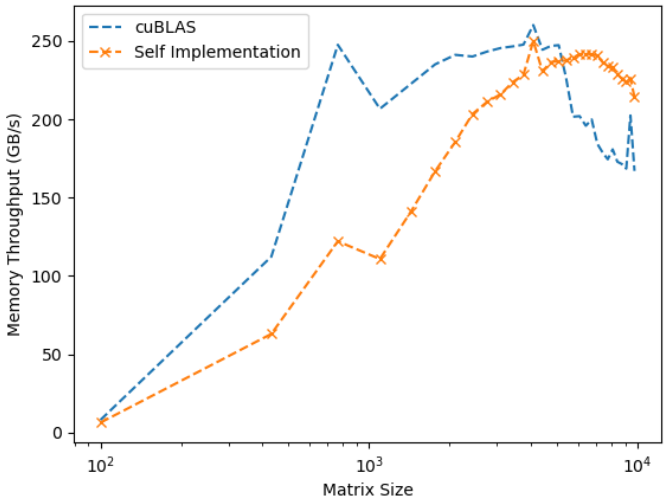


Figure 2: Comparison between cuBLAS implementation and self implementation

CuBLAS provides a set of pre-optimized linear algebra routines that are designed to work efficiently on NVIDIA GPUs. For small-size matrices, the cuBLAS library gives better performance than the code I implemented by myself. But for large-size matrices, the cuBLAS implementation can not give performance as good as mine, probably because it tries to use some complex layout but end up with too much overhead between blocks, which affects the memory throughput a lot.

Also, we can see in the figure there is obviously an L2 cache drop. Since we are using C++ float type data, it has a size of 4 bytes while our GPU has a 4MB L2 cache size. By calculation, we can know that the drop should happen around matrix size 1000. Which aligns with our result. Also when the data is too big to fit in the fast memory we can see the performance drops since we have to start fetching data from RAM.

The maximum memory throughput should be achieved at around 400 GB/s. However, in my implementation, cuBLAS did not achieve the performance it should have. The peak memory throughput is around 250 GB/s.

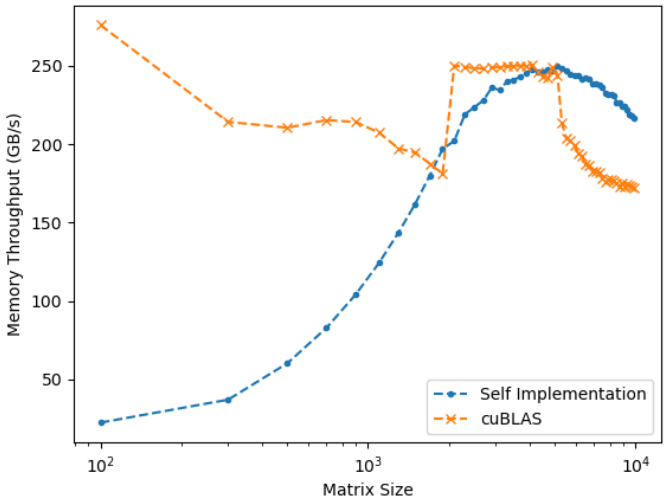


Figure 3: N = 10000 and M between 100 and 10000

Next, we try to change the size of the matrix. We give it a fixed $N = 10000$ and modify $M$ between 100 and 10000. The result is shown in Figure 3.

Since our matrix is stored in column-major, with the increase of $M$ we are actually doing more parallelization jobs. The increase of the matrix size at first helped to improve the occupancy, which led to a higher performance, this is shown in our "self-implementation" curve. The drop in my own implementation curve is because there are too much data to fit in the fast memory and we have to use RAM which leads to a worse performance.

On the other hand, for the cuBLAS implementation, we first saw a drop in performance since it reached L2 cache size and took more time to fetch data. But then we can also see there is a little peak at around $N = 2100$. This is probably because we can get more threads occupied with a larger $M$ before it finally grows too large we have to fetch data from RAM.

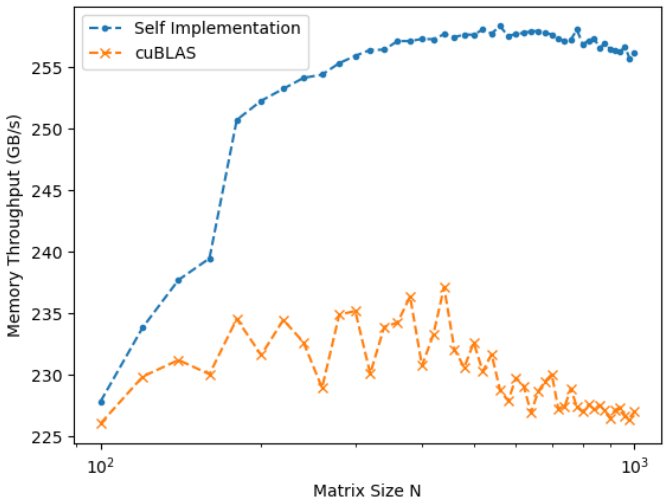


Figure 4: M = 16384 and N between 10 and 1000

In the third question, we have to change the matrix size with a fixed M = 16384 and N between 10 and 1000. The result is shown in Figure 4.

Here since $M$ is a large fixed, is the result of $128 \times 128$, our virtual layout is actually ideal in this question, and we can make sure every thread gets work. Since our implementation is ideal in this situation we can expect the performance will increase with the matrix size increasing. But for cuBLAS, there might be overhead causing the worse performance because it is trying to optimize more generally and therefore either choosing a worse layout or it took time to choose the same optimal layout for it.

# 3 Transpose Matrix and Multiply with Vector

For this task, I implemented a new kernel code to reduce by column. Since the matrix is transposed, we can now calculate each element of the result vector independently column by column. We know that a whole block of GPU will access data at the same time. So this transpose matrix-vector multiplication should give us a worse performance since now it has row-majored storage. By doing the experiment, we get the result that self-implementation has a performance of **108.805 GB/s** while cuBLAS has a performance of **243.783 GB/s**.

However, the performance of cuBLAS does not change a lot compared to the original version. This is because cuBLAS is designed to handle both storage formats efficiently and perform the necessary data transfers and computations optimally.

# 4 Matrix-Matrix Multiplication

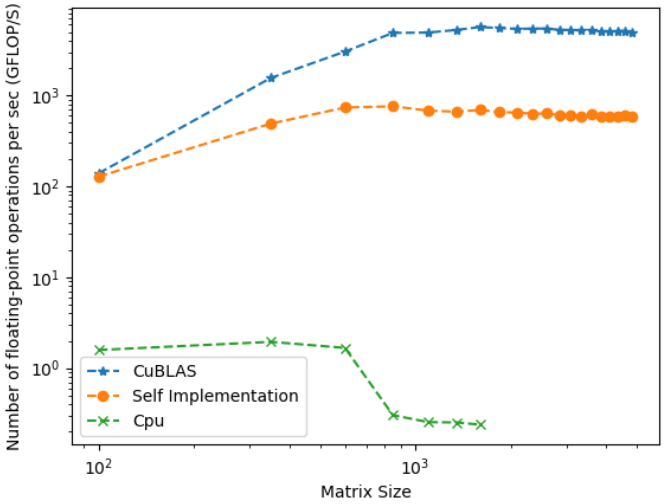


Figure 5: M = N matrix-matrix multiplication

In this task, we can see that cuBLAS has a significantly better performance than my code. This is reasonable because cuBLAS uses a highly optimized method to calculate matrix-matrix multiplication while my code just uses the most simple way to calculate it and it has a very obvious weak point.

In my code, all threads with the same *col* re-read the whole $j$th column of the matrix, this leads to memory bandwidth wasted. If we change our implementation to a more efficient way we will get better performance, for example, we can use the tile method.

But compared to CPU code, we can have a much better performance because CPU and GPU prefer a different way to access data and the column-major is preferred by GPU. In CPU we would prefer so-called `Array-of-Structure` because we can easily prefetch when we walk over it. But for GPU we prefer `Structure-of-Array` because we have contiguous access for multiple threads.