CSE260 Assignment2 - GPU Matrix Multiplication

Ho-Lun Wu
A53271935
hlwu@eng.ucsd.edu

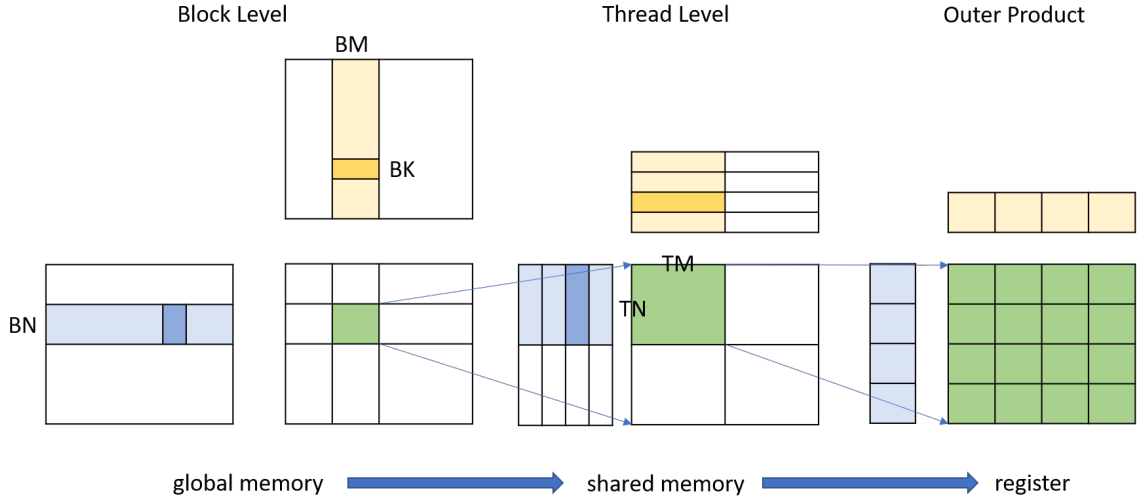
Abstract

In this assignment, we are going to optimize the square matrix multiplication ($C = A * B$ where A,B and C are $N \times N$ matrices) with GPU. By applying shared memory, hierarchical memory, and instruction-level parallelism, our program achieved a peak of 844 Gflops with a single GPU which is better than the requirement 360 Gflops.

Section 1

Program Behavior

Figure 1: Hierarchical Memory



To optimize the target program, we use the hierarchical memory structure to most utilize the cache of GPU. In the block level, the left of Figure 1, we fetch the partial matrix elements into

Algorithm 1 Psuedo Code

1: **procedure** BLOCKHIERARCHICALMEMORY
2: $\_\_shared\_\_$ $double$ $sharedA[BN][BK]$, $sharedB[BK][BM]$
3: $double$ $regA[BN]$, $regB[BM]$, $c[BN][BM]$
4: **for** $k = 1$ **to** $\frac{N}{BK}$ **do**
5: load matrix into shared memory $sharedA$ and $sharedB$ corresponding to k-index
6: $\_\_syncthreads()$
7: **for** $i = 1$ **to** BK **do**
8: load vector into $regA$ and $regB$ from $shraedA$ and $sharedB$ corresponding to i-index
9: **end for**
10: $c = c + a \times b$ ▷ add outer product of a and b into c
11: $\_\_syncthreads()$
12: **end for**
13: merge c into global memory
14: **end procedure**

the shared memory in each iteration in each block. In the thread level, the middle of Figure 1, we fetch the partial vector elements from shared memory into local registers. In the last level, the right of Figure 1, we calculate the outer product of the registers and add the results into the local registers. According to Figure 1, there are several parameters, BN, BM, BK, TN and TM, which decide the size of the shared memory and registers in each block and each thread. In our final implementation, we assign two sets of parameters to deal with the different size of matrices and we will discuss the mixed parameters in the later sections.

Optimization

Hierarchical Memory

The hierarchical memory structure is the main part of the whole optimization. The main idea is pre-fetch the part of the matrix which is used frequently to the faster memory. For example, in the first level (block level), line 5 of Algorithm 1, we fetch a part of the matrices A and B into the shared memory in the block, so can access the part of the matrices more efficiently. In the next level (thread level), line 8 of Algorithm 1, each thread will compute small part of the block by outer product. Then, we need two vectors whose sizes are $1 \times TN$ and $1 \times TM$ respectively to calculate a small part of matrix C which is $TN \times TM$. Copying the two vectors from shared memory into registers allows us to access the vectors faster. Also, we will keep the small part of matrix C which is $TN \times TM$ in the registers so we are able to update the outer product more efficiently.

Instruction-Level Parallelism

In the GPU processor, if they get an instruction which is dependent on the result of the previous instruction, the processor will stall and wait until the result of the previous instruction is available. In this situation, the processors will operate in low utilization. For example, if there are two consecutive instruction: $a = b + c, d = a + e$, the second instruction needs to wait until the result of the first instruction is available. To avoid this low utilization, we can reorder the instructions

such that each instruction isn't dependent on several previous instructions. In our case, we will calculate several values interleaving. For example, the outer product (line 10 in Algorithm 1), each instruction isn't dependent to the previous instructions.

Pointer Aliasing

When passing the pointers with *const__restrict__* decorator as parameters, which is called pointer aliasing, the compiler knows that these pointers will not be used to access overlapping regions so the compiler is able to optimize the memory access. Also, it tells the compiler that this region of memory is read-only, so the compiler is able to use *__ldg*() to load the data with a read-only cache.

For Loop Unroll

Adding *#pragmaunroll* to the front of the for loop with a constant number of iterations to tell the compiler to unroll the for the loop. By doing so, there is less number of the branch instruction which needs to flush the processors' pipeline. However, we can't unroll the for loop without a constant number of iterations, since it will decrease the performance.

Different Block Sizes

Because we notice that the different size of the matrix has its own best parameters, we define several sets of parameters.

Template Programming

In order to use different parameters and reduce the length of the code, we use template the program a device function which will be called in the global function instead of passing parameters as function parameters or writing several functions.

Inline Function

For each function, we add __inline__ to the function to reduce the call instructions.

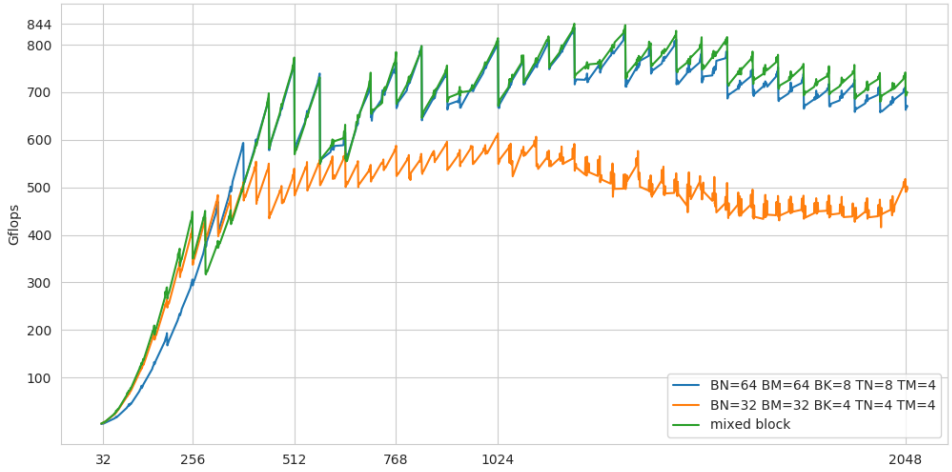
Development Process

At the beginning of this assignment, we try several different block sizes, but the performance is still around 70 Gflops. To obtain better performance, we apply the shared memory into our program according to the method discussed in the lecture. The shared memory version can achieve around 250 Gflops with the block size = 32. In this version, each thread only computes one element in the C matrix. To increase the throughput, we need to consider the trade-off of how many elements need to be calculated in one thread. Due to that, we unroll the shared memory version to make each thread calculate more than one element. In our experiments, we find that the program can achieve the best performance (around 450 Gflops) when block size = 32 and each thread calculates 8 elements. Then, we notice that only a small number of registers is used in each thread, that is, we need to access more times of shared memory or global memory. Finally, we are inspired by CUTLASS [1], developed by NVIDIA, which calculates the most inner block by outer product and apply hierarchical memory structure (global memory, shared memory and registers) to improve our program. After tuning the parameters, our program is able to achieve a peak of 800 Gflops.

Section 2

In this section, we will plot three sets of parameters, including the smaller block, the larger block, and the mixed block. In our experiments, we notice that the parameters with smaller block size perform better on the smaller matrices and parameters with larger block size work better on the larger matrices. The parameters of the larger block are $BN = 64$, $BM = 64$, $BK = 8$, $TN = 8$ and $TM = 4$. The another set of parameters is $BN = 32$, $BM = 32$, $BK = 4$, $TN = 4$ and $TM = 2$. We decide these parameters based on several criteria: the number of blocks and the number of threads in one block. Too less number of block make the multiprocessors wait for __syncthreads() at most time. However, too many blocks imply there is less number of threads in one block which there aren't enough warps to be scheduled in each block. On the other hand, too many threads in each block imply each thread does less computation. Nonetheless, too less number of threads means there isn't enough number of warps to be scheduled. Also, we need to consider the number of registers in each block and in each thread. Finally, we decide to use smaller block parameters for the matrices whose size is less than 384 and use the bigger for the others.

Figure 2: Block Size Selection



According to Figure 2, we find out the the blue line and the orange line intersect at around $N = 384$, so we use the parameters $BN = 32$, $BM = 32$, $BK = 4$, $TN = 4$ and $TM = 2$ for $N < 384$ and choose parameters $BN = 64$, $BM = 64$, $BK = 8$, $TN = 8$ and $TM = 4$ for $N \geq 384$. Finally, the green line is the performance of the mixed parameter version which has the best performance.

Section 3

Table 1: Compare to Naive program

N	our implementation	naive with $BX = BY = 32$
256	449	65
512	773	77.4
768	784.5	76.6
1023	810.2	56.6
1024	813.8	69.3
1025	671.2	55.3
2047	730.2	38.7
2048	741.6	47.3
2049	693.6	38.6

According to Table 2, we notice that our implementation performs much better than the naive one because the naive program computes only one element in matrix C in each thread and it accesses the global memory directly without caching by shared memory or registers which leads to poor memory bandwidth.

Section 4

Table 2: Compare to BLAS program

N	BLAS (GFlops)	our implementation (GFLOPS)
31		4.6
32		5.2
33		5.3
256	5.84	449
257		350.2
512	17.4	773
768	45.3	784.5
1023	73.7	810.2
1024	73.6	813.8
1025	73.5	671.2
1212		834.6
1213		834.1
1214		837.8
1215		840.8
1216		844.3
1341		831.2
1343		833.1
1344		841.2
2047	171	730.3
2048	182	741.6
2049	175	693.6

Figure 3: Compare to BLAS

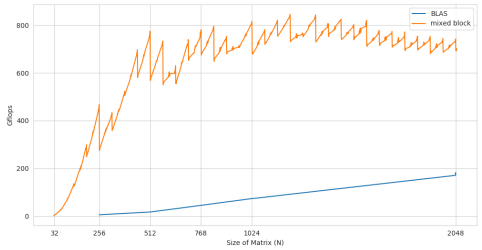
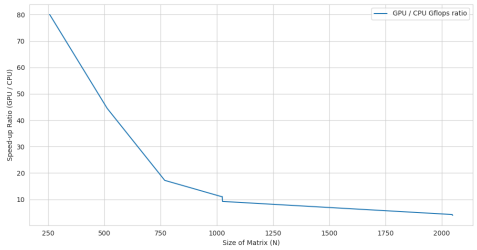


Figure 4: Speedup Ratio



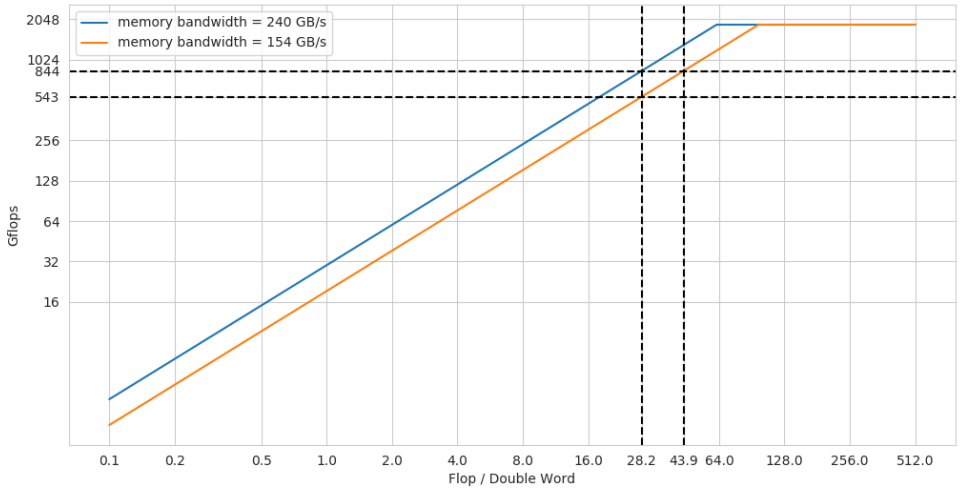
In Figure 3, the curve of BLAS keeps increasing, but the curve of our implementation drops slightly when N is larger than about 1500. We think it might need a set of parameters which has the larger size of block. Then, we notice the speedup ratio curve in Figure 4 keeps dropping when N is increasing. We think the reason is that the multi-core version of BLAS hasn't reached the

peak performance; however, our implementation reaches the peak performance more quickly than BLAS.

Also, we figure out that the performance of our implementation is extremely low when N is small because the overhead of launching kernel is too large for a small matrix and the clock speed of CPU is much higher than the clock speed of GPU.

Section 5

Figure 5: Roofline Model



According to Figure 5, the blue line is the curve with maximum memory bandwidth = 240 GB/s (Tesla K80) and the orange line is the curve with the maximum memory bandwidth = 154 GB/s (Volkov thesis). Assume the real memory bandwidth is equal to the peak memory bandwidth (240 GB/s), the Q value will be 28.2 $Flop/Double\ Word$. If the real memory bandwidth is equal to the value measured by Volkov (154 GB/s), the Q value will be 43.9 $Flop/Double\ Word$. Actually, the Q value will be a constant value corresponding to the program. Due to that, the same program is able to achieve higher performance with higher memory bandwidth.

Future Work

Multiple GPU

In this assignment, we only use one GPU for matrix multiplication. Actually, we are able to use multiple GPUs to compute matrix multiplication, that is, each GPU compute a small part of the matrix. theoretically, we are able to reach K times speedup if we use K GPUs

Overlapping of Memcpy and Computation

In this assignment, we copy the matrix from the host to the device before the computation and calculate the performance according to only the time consumption of matrix multiplication without memory copy. However, we can overlap the memory copy and computation. In the other word, the execution sequence will be *copy* $A_1, B_1 \rightarrow$ *compute* $C_1 \rightarrow$ *copy* $A_2, B_2 \rightarrow$ *compute* $C_2 \rightarrow \ldots \rightarrow$ *copy* $A_n, B_n \rightarrow$ *compute* $C_n \rightarrow$. By doing so, the CUDA library can copy the memory and do multiplication at the same time.

References

[1] Julien Demouth Andrew Kerr, Duane Merrill and John Tran. Cutlass: Fast linear algebra in cuda c++, 2017.