CPE 810 Lab 01

Zirui Wen

Question 1

In a dense matrix multiplication (computing $C = A \times B$):

- There are N multiplications.
- There are $N - 1$ additions.

Thus, the total FLOPs per element is: $2N - 1$. Since there are N^2 elements in C, the overall operation count is:

$$\text{Total FLOPs} = N^2(2N - 1) \approx 2N^3 - N^2.$$

For large N, the computational complexity is dominated by the $2N^3$ term, so the computational complexity is $O(N^3)$.

Question 2

In Naive approach, $2N$ per element, and the whole matrix have N^2 elements, so the total Global memory reads are $2N^3$ times. In a tiled matrix multiplication kernel, tiles of A and B are loaded into shared memory. Suppose each tile is of size $T \times T$, I set the tile size as 32×32. In that case, each tile is loaded once and reused by T threads. This reduces the effective number of global memory reads to:

$$\frac{2N^3}{T} = \frac{2N^3}{32}. \tag{1}$$

Question 3

We only need one time each elements for the Global memory writes. Thus, the total number of global memory writes is:

$$N^2.$$

Question 4

There are several further optimizations can improve CUDA matrix multiplication performance:

(a) **Multi-Level Tiling:**

- Using Warp tile methods to accelerate the speed.
- Implement both 1D Thread tiling and 2D Thread tiling in addition to block-level tiling.

(b) **Optimized Memory Access:**

- Ensure coalesced global memory accesses by aligning data so that threads in a warp read contiguous memory locations.
- Use vectorized loads/stores (e.g., loading a `float4` at a time) to improve bandwidth.
- Arrange shared memory accesses to avoid bank conflicts.

(c) **Double Buffering:**

- Overlap data transfers (from global to shared memory) with computation by using two sets of shared memory buffers (ping-pong buffering).

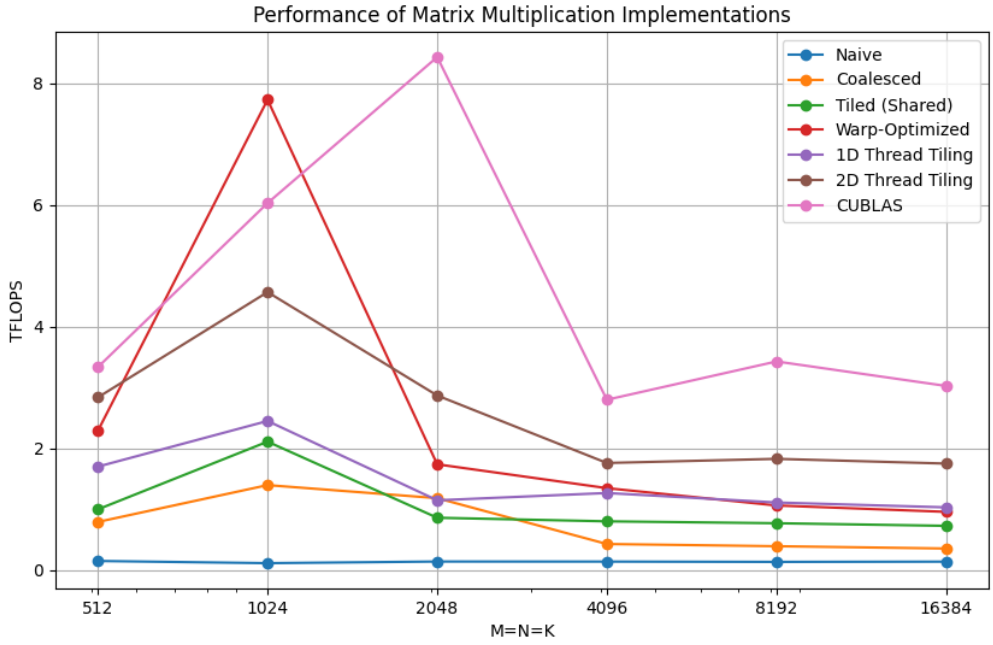
(d) **Tensor Cores:**

- Tesla V100 GPUs feature Tensor Cores, which are specialized units for matrix multiplication.
- By employing half-precision (FP16) or mixed precision (FP16 inputs with FP32 accumulation) via CUDA's WMMA API, a significant speedup (up to $8\times$) can be achieved.

In my code, I implemented the **coalesced global memory accesses**, **warp tiling** algorithm, **1D Thread tiling** and **2D Thread tiling**. arrange the share memory access to avoid **bank conflicts**. But when doing the warp tiling part, I noticed that sometimes it have abnormal speeds when calculating small matrices, and I don't know the reason. Here is the figure that shows the result:

Question 5

When dealing with this problem, we can just use tiling methods. Split the output matrix into smaller tiles that a single kernel launch can handle, and launch the kernel multiple times. The algorithm is the same as we learn in class, and the code is showned in the file.



Question 6

For this question, we can use the method we learned from linear algebra class, the Block matrix.

$$C = A \times B = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \begin{pmatrix} B_0 & B_1 \end{pmatrix} = \begin{pmatrix} A_0B_0 & A_0B_1 \\ A_1B_0 & A_1B_1 \end{pmatrix}. \tag{2}$$

If the A_0, A_1, B_0, B_1 is too big for the global memory, we can use recursion method, separate the A_0B_0 and other matrix into smaller matrices until the global memory can fit. Then use the math equation to calculate the final answer. I will show some code:

Algorithm 1 RecursiveMatrixMultiply

```
function RecursiveMatrixMultiply(A, B)
    if A and B can fit into GPU global memory then
        return GPU_Multiply(A, B)
    else
        // Split along the common (K) dimension
        [A_0, A_1] ← SplitMatrixColumns(A)
        [B_0, B_1] ← SplitMatrixRows(B)
        A_0B_0 ← RecursiveMatrixMultiply(a_0, b_0)
        A_0B_1 ...
    end if
end function
```

Also, the report have some reference, I will show the reference in the end of the report.

Some website that I referenced:

https://www.reddit.com/r/CUDA/comments/1cv77rc/optimizations_that_can_be_applied_to_the_matrix/?rdt=59131#:~:text=

https://siboehm.com/articles/22/CUDA-MMM#:~:text=5%3A%202D%20Blocktiling%20%6015971.7%60%20%6068.7,10%3A%20Warptiling%20%6021779.3%60%20%6093.7

https://github.com/wangzyon/NVIDIA_SGEMM_PRACTICE

https://leimao.github.io/article/CUDA-Matrix-Multiplication-Optimization/

https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/