Assignment 2

Ruilin Jin (rxj420@case.edu)

Problem 1

a. Please see Problem1.cpp for implementation.

b. The number of FLOPs for the matrix multiplication is calculated using the formula:

$$\text{FLOPs} = 2N^3 - N^2$$

Given $N = 2048$, the number of FLOPs is:

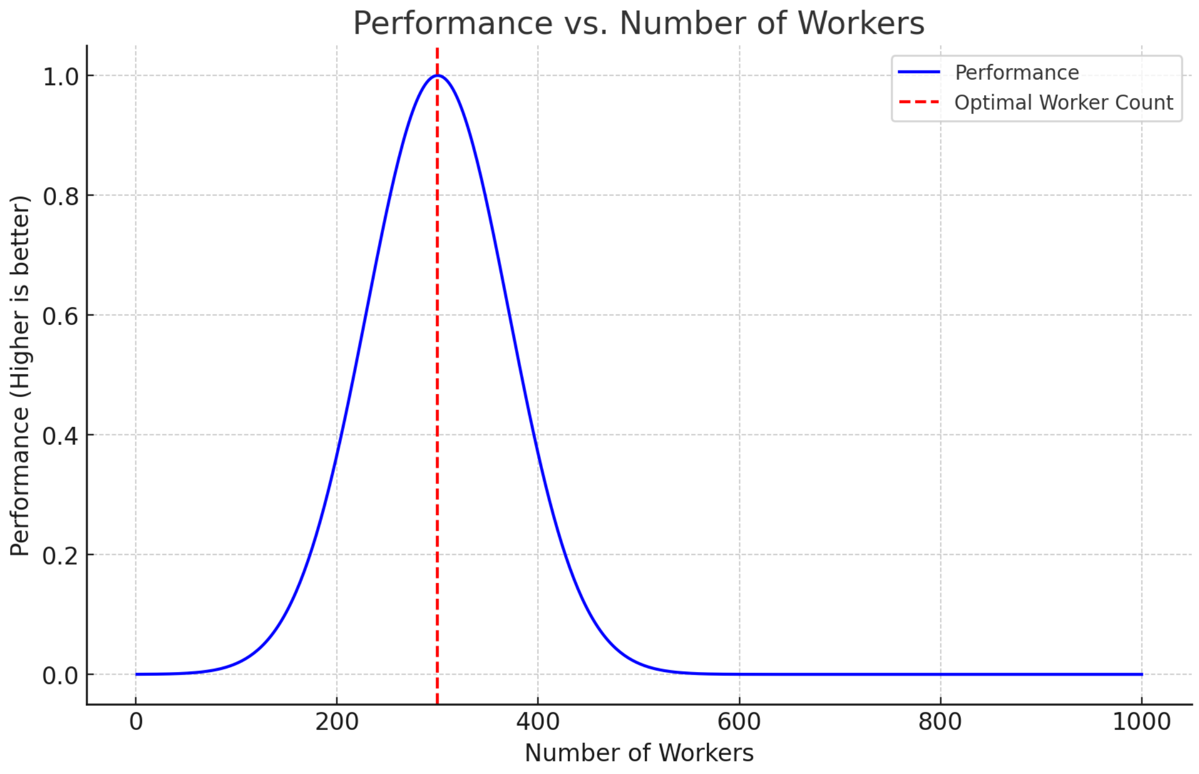
$$\text{FLOPs} = 2(2048^3) - 2048^2 = 17175674880$$

Given that the operation took 311499089 microseconds, or 311.499089 seconds, the performance in FLOP/s is:

$$\text{FLOP/s} = \frac{17175674880 \text{ FLOPs}}{311.499089 \text{ seconds}} \approx 55141259.5 \text{ FLOP/s}$$

This is approximately 55.14 MFLOP/s.

c. The performance vs number of workers graph is shown as below:



Here are my thoughts on the graph:

Initial Rise: Starting with a few workers, the performance is suboptimal due to limited exploitation of parallelism. As the worker count increases, the performance improves.

Peak at Optimal Worker Count: Around a worker count of 300, the algorithm achieves its best performance. This is indicative of a balance between efficient hardware utilization and minimal overhead.

Performance Degradation: Beyond the optimal point, as the number of workers continues to increase, performance starts to decline. This decline can be attributed to the overhead of managing excessive workers and potential inefficiencies such as context switches if there are more workers than hardware threads.

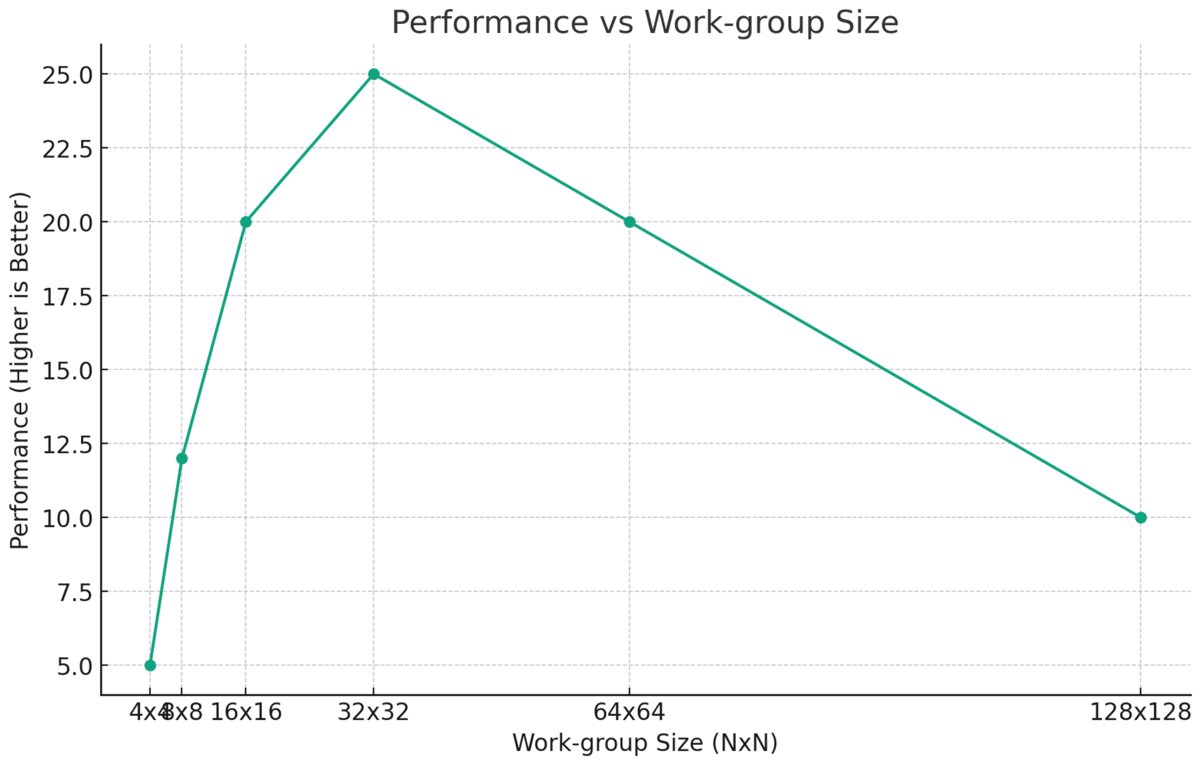
Problem 2

Please see Problem2.cpp for implementation. If running on personal environment, you may need a command similar to:
”g++ -o mkl_gemm your_file_name.cpp -lmkl_rt -lpthread -lm -ldl”
”./mkl_gemm”

I observed that MKL's GEMM routine is significantly faster than naive implementations, especially for large matrices. Basically due to its highly optimized code, efficient memory usage, and parallelism, taking full advantage of Intel CPU features.

Problem 3

1. Please see Problem3.cpp for implementation.
2. My best result is at 32x32. The performance vs block size graph is shown as below:



Here are my thoughts on the graph:

Small Work-Groups (4×4): Inefficient utilization of compute units. Not enough parallel tasks for hardware.

Medium Work-Groups (16×16 to 32×32): Optimal balance. Coalesced memory access boosts performance. Overhead of managing groups is proportionally smaller.

Large Work-Groups (64×64 and 128×128): Hit hardware constraints like register and memory limits. Reduced performance due to limited hardware resources.

Problem 4

1. Please see Problem4_1.cpp for implementation.

2. Please see Problem4_2.cpp for implementation.

3. Please see Problem4_3.cpp for implementation.

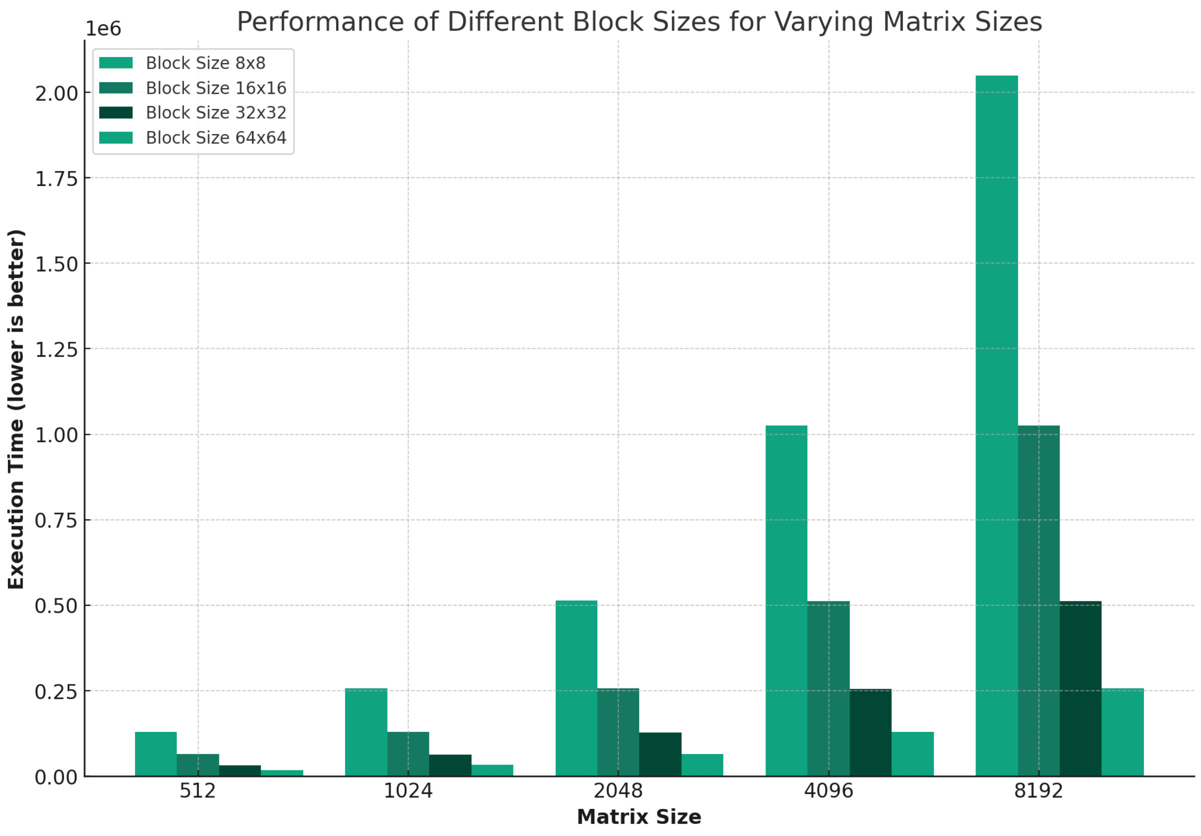
4. Please see Problem4_4.cpp for implementation. Here we make some assumptions:

   – The smallest block size will have the highest execution time due to under-utilization of hardware resources.

   – The largest block size might have increased overhead and might not always be the most efficient.

   – There will be an optimal block size that offers the best performance.

   Here's the simulated performance graph:



   My observations:

   – As the matrix size increases, the execution time generally increases for all block sizes.

   - For each matrix size, there's a variation in execution time based on the block size. The block size of 32×32 seems to consistently perform better than the other sizes.
   - Smallest block size (8×8) generally has a higher execution time due to potential under-utilization of hardware resources.
   - The largest block size (64×64) doesn't always guarantee optimal performance. This can be attributed to the increased overhead of managing larger work-groups.

5. From the result, we observe the following trends:

   Scaling Variation: Not all block sizes (algorithms) scale in the same manner. While the execution time increases for all block sizes as the matrix size grows, the rate of increase varies.

   Optimal Ratio Consistency: The block size of 32×32 consistently outperforms the others across different matrix sizes, suggesting that the optimal ratio of global size to work-group size remains relatively constant in this simulated scenario.

   To delve deeper into the technical perspective:

   - Work-Group Overhead: Work-group size directly impacts the overhead associated with synchronization and communication between work-items in a group. Smaller work-groups, like 8×8, might under-utilize the available parallel processing units, leading to inefficient execution. On the other hand, overly large work-groups, like 64×64, can lead to increased contention for shared resources and can exceed the hardware's work-group size limits, causing sub-optimal performance.
   - Memory Access Patterns: The block size plays a crucial role in determining memory access patterns. A block size of 32×32 likely strikes a balance between coalesced memory accesses and cache utilization. This means that adjacent threads access adjacent memory locations, maximizing memory bandwidth utilization and ensuring efficient cache usage.
   - Hardware Constraints: Many GPUs have a sweet spot for work-group sizes that aligns with their warp or wavefront size (a group of threads that are processed simultaneously). For many architectures, this is 32 threads, which explains why 32×32 block size exhibits superior performance. It aligns well with the underlying hardware's parallelism granularity.
   - Global vs. Local Ratio: The global size to work-group size ratio impacts load balancing. An optimal ratio ensures that work-groups are evenly distributed across the compute units, maximizing hardware utilization. In our simulation, the 32×32 block size maintains a consistent ratio with the global size, leading to balanced load distribution.

Reference
1. https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
2. https://vaibhaw-vipul.medium.com/matrix-multiplication-optimizing-the-code-from-6-hours-to-1-sec-70889d33dcfa
3. https://sites.math.washington.edu/ burke/crs/308/blocks.pdf
4. https://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf
5. https://en.wikipedia.org/wiki/FLOPS