Name: Meenal Shah CSS535 Homework 1

Instructions:

- *You can hand-write (must be legible) your answers or type them.*
- ***No collaboration** with other students (or humans) in or outside our roster.*
- *Think about the questions before you "google" for answers. **If** you must use the internet, you must **cite** all sources and provide additional explanations (i.e., add your personal contribution, not just what you found online).*
- *The answer to each question **should not exceed the size of the textbox provided for that question***
- *Submit on Canvas by the specified date.*
- ***Use a standard Font, size 12 pt !!!***

1. [10p] Are the BLAS operations available through generic LAPACK single-core or multi-core oriented (i.e., are they optimized for single-thread or multi-thread)? Research and briefly discuss your findings.

The generic LAPACK packages built on BLAS are relatively old and were originally designed for single-core architectures. These implementations are typically optimized for single-core performance, with hardcoded defaults for parameters such as block size.
However, with the widespread adoption of multicore architectures, several optimized LAPACK implementations have been developed to better utilize multiple cores.
Notable examples of multicore-optimized LAPACK implementations include ATLAS, OpenBLAS, Intel MKL, Magma.
Since these optimized libraries are widely used in modern computing environments, most users—whether knowingly or unknowingly—benefit from multicore-optimized LAPACK implementations rather than the original single-core versions.
Reference: Duke University - BLAS/LAPACK

2. [20p] Think about the tiling strategy for parallelizing and optimizing matrix multiplication. So far, we've been discussing this assuming that the matrices are *dense*. Would the tiling strategy for parallelization also work for *sparse* matrices? (Think about extremely large data.) Briefly elaborate/discuss your answer.

The tiling strategy can optimize sparse matrix multiplication, but not to its full potential. This is because sparse matrices are predominantly filled with zeroes, and a direct tiling approach would result in wasted memory and processing bandwidth due to unnecessary multiplications with zero values.
Typically, tiling assumes a dense matrix, where optimization techniques involve storing matrix values efficiently in memory and leveraging SIMD (Single Instruction, Multiple Data) operations for parallel computation. However, in sparse matrix multiplication, a significant portion of values are zero, making direct tiling inefficient.
Some optimization techniques can be:

1. Separating Zero and Non-Zero Values:
   Instead of storing the entire matrix, we can a different data structure to store only non-zero values and their corresponding indices and a separate one to track the zero-value.
2. Avoiding Redundant Computation:
   Since multiplying by zero always results in zero, we can introduce a pre-check before performing multiplication, ensuring only non-zero elements contribute to the result.
3. Load Balancing for Parallel Processing:
   A direct tiling approach can lead to computational imbalance, where some threads perform significantly more non-zero multiplications than others. To mitigate this, the pre-identified non-zero elements can be assigned computations dynamically.

While the tiling strategy remains more efficient than sequential matrix multiplication, its effectiveness for sparse matrices depends heavily on additional optimizations. By incorporating zero-aware computation, different storing techniques and load-balancing techniques, we can significantly improve performance beyond traditional tiling methods.

3. [40p] Puff is a graduate student who collects old computers. He just fixed a mysterious computer, presumably from the late 90's, but he doesn't have any information about the hardware. He thinks that looking at the performance of Matrix-Matrix multiplication with double precision floating point numbers might give him some information about memory hierarchy. He ran a naïve version (3 nested loops) and then some other optimizations with different matrix sizes, he was also able to run ATLAS (with optimized BLAS) on it.

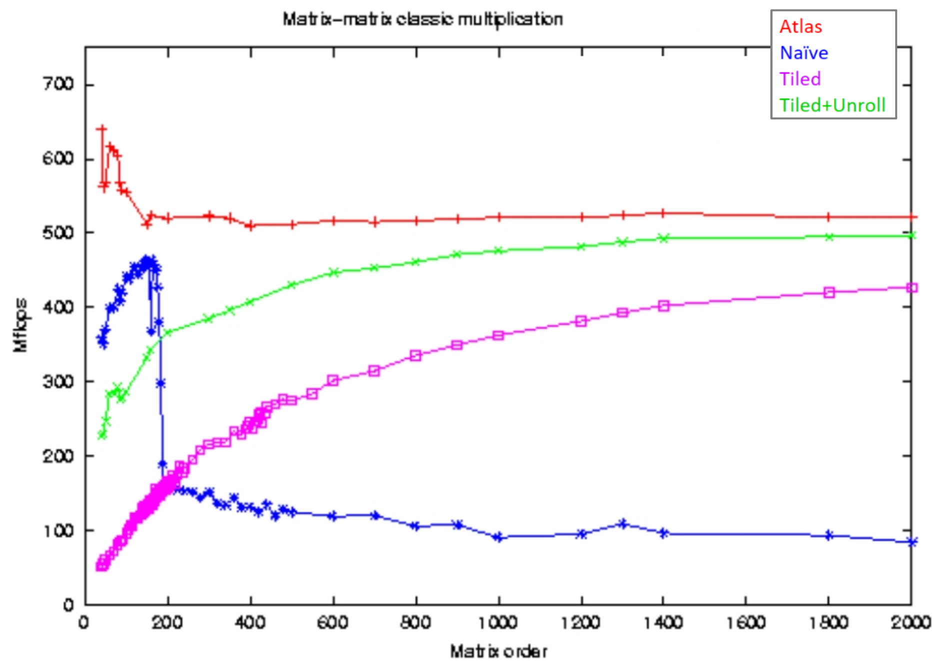

Figure 1

Figure 1 shows some of the trends Puff collected from his experiments.

Is there anything in this figure that would give him any information regarding the memory hierarchy? Explain your answer.

Do you have any additional guidance for Puff? Mention any relevant assumptions and/or considerations.

From the matrix-matrix multiplication graphs above, the following conclusions can be drawn:

1. Since Puff was able to run ATLAS on the machine, it confirms that the machine supports multi-core architecture, as the ATLAS library is multithreaded. This suggests the machine likely has L1, L2, and L3 level caches.
2. Around a matrix order of 200, we observe a significant drop in Mflops for the Naïve implementation. This is likely because the matrix order exceeds the cache size, forcing the matrix to be stored in main memory. From this, we can estimate that the L3 cache size is approximately 200 * sizeOfFloat = 200 * 16 bytes = 3.2 KB.
3. The tiled (or tiled + unrolled) approach yields results comparable to the ATLAS implementation, particularly for larger matrix orders. This indicates that the tile size and the memory consumption due to loop unrolling are likely optimized to the highest extent, assuming ATLAS is the most optimized implementation for matrix-matrix multiplication. Based on this, Puff can also approximate the sizes of the L1 and/or L2 caches.

Additional guidance for Puff:

1. To further refine cache size estimates, Puff should run matrix-matrix multiplication for smaller matrix orders (less than 200). Plotting these values on a graph similar to the one above could reveal smaller but significant dips, which may help in identifying the L1 and L2 cache sizes.

Assumptions made:

1. It is assumed that Puff's algorithms for the different execution techniques are correct.
2. It is assumed that ATLAS is installed and functioning properly, and that it is the most optimized matrix-matrix multiplication implementation available compared to other libraries.

4. [30p] Suppose you are given a picture as a 2D array of 400 by 900 pixels (i.e., each item of the 2D array represents a pixel) and a function `Foo` that applies some operation to each pixel. The size of a pixel in this scenario is 16 bits. You want to parallelize this function using CUDA. Design a kernel where each thread processes one pixel. If thread Blocks should be square and you want to use the maximum number of threads per block on the device, how would you choose the execution configuration dimensions for optimal performance? You don't have to write `Foo`'s code, just give the configuration and explain your reasoning. Mention any relevant assumptions and/or considerations. (Assume compute capability 3.7).

Assuming a compute capability of 3.7, the maximum number of threads per block is 1024. Since the warp size in CUDA is 32, we can determine that the largest and most optimal square block size is 32x32.

Using this information, we can calculate the number of blocks needed to process the entire image.

- Number of blocks spanning the image width: 400/32 = 12.5400 / 32 = 12.5400/32=12.5, which rounds up to 13 blocks.
- Number of blocks spanning the image length: 900/32 = 28.125900 / 32 = 28.125900/32=28.125, which rounds up to 29 blocks.

Therefore, the total number of blocks required is 13×29 = 377 blocks.
Thus, the kernel configuration for processing the image will be <<<377, 1024>>>.