ECE 665 Final Project - A Review of Matrix Multiplication for Machine Learning and Showcase of Modern Acceleration Techniques

John Murray
Electrical and Computer Engineering
University of Massachusetts Amherst
jomurray@umass.edu
ORCID ID:0009-0000-8264-0623

***Abstract*—Matrix multiplication is a core mathematical operation in many compute-intensive tasks, particularly in machine learning where it makes up fundamental network layers [1]. Accelerating this operation greatly speeds up training and inference workloads. This paper presents an implementation of matrix multiplication accelerated using SYCL, a C++ heterogeneous parallel programming framework, targeting a GPU device as an accelerator. The system runtime of a naive CPU implementation is compared against an SYCL-based GPU kernel for multiplying large matrices. The SYCL approach yields a dramatic speedup over the single-threaded CPU baseline. We detail the implementation for both CPU and GPU, including memory layout and timing methodology. Our evaluation shows the GPU execution time is about 411× faster than the CPU's, confirming the expected benefit of massive parallelism on GPUs. We also briefly discuss the role of data transfer overhead in the measured SYCL kernel time (notably, the reported kernel execution time can exclude the host-device memory copy latency under SYCL profiling [2] [3]). Finally, we outline future directions and optimizations – from algorithmic improvements like optimal matrix chain multiplication to low-level techniques such as tiling, and batched operations – that can further enhance matrix multiplication performance.**

Code is available at - https://github.com/Deltajom/ECE665ClassProject/tree/main

I. Introduction

Matrix multiplication lies at the heart of many modern computing workloads, especially in scientific computing and machine learning. Neural networks and other machine learning models spend a large portion of their execution performing matrix multiplications and as such general matrix multiply (GEMM) operations are a fundamental building block for machine learning model layers such as fully-connected layers, recurrent networks, and even convolutions [1]. As models grow in size and data, the computational demand of these matrix operations increases substantially. This has driven a need for hardware acceleration: specialized processors (GPUs, TPUs, NPUs, FPGAs) are now routinely employed to speed up linear algebra operations by taking advantage of parallelism. In fact, matrix multiplication is often cited as one of the most important and intensive computations in deep learning, accounting for the majority of FLOPs (floating-point operations) during training and inference of large models [4].

Accelerator hardware like GPUs can perform many arithmetic operations concurrently, offering a significant throughput advantage over general-purpose CPUs for data-parallel tasks. A modern GPU consists of thousands of smaller cores arranged for parallel processing, whereas a typical CPU has on the order of only tens of high-performance cores. In this work, we investigate accelerating matrix multiplication in the context of CPU versus GPU performance.

Fig. 1. SYCL logo

II. Background

A. SYCL, Khronos, and oneAPI

SYCL (pronounced "sickle") [5] is an open standard for heterogeneous computing defined by the Khronos Group. SYCL extends standard ISO C++ with high-level abstractions for accelerator platforms, in essence, SYCL allows developers to write code that can run on CPUs, GPUs, FPGAs and more, enabling portability and reuse across hardware targets. SYCL's runtime manages device selection, memory transfers, and kernel execution, providing a consistent programming interface over underlying backends such as OpenCL or CUDA.

It is worth noting the relationship between SYCL and Intel's oneAPI [6]. oneAPI is an initiative and specification by Intel to provide a unified programming model for diverse architectures. The oneAPI toolkits include the DPC++ compiler (based on Clang/LLVM) and a set of optimized libraries and tools. This means code written in SYCL/DPC++ can be compiled to run on CPUs and GPUs. In summary, SYCL defines the cross-platform heterogeneous C++ API, and oneAPI delivers a concrete ecosystem (compiler, runtime, libraries). By using SYCL in our work, we tap into a modern heterogeneous programming paradigm, gaining the flexibility to run the same code on different devices without vendor lock-in.

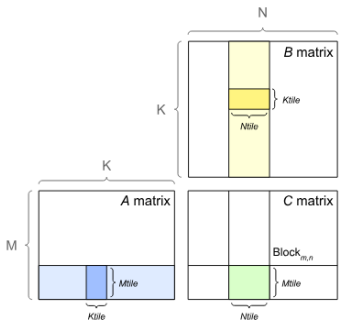

Fig. 2. Tiled GEMM computation of $C = A \cdot B$, where matrix A is partitioned into horizontal tiles, B into vertical tiles, and C into output blocks. Each C block is computed by multiplying corresponding A and B tiles.

B. Matrix Operations in Machine Learning

Matrix and tensor operations form the computational backbone of most machine learning (ML) algorithms. For example, multiplying weight matrices by input activation matrices is how fully-connected (dense) layers compute outputs, and similar patterns appear in recurrent neural networks and even convolutional layers (which can be transformed into matrix multiplications) [1]. Because these layers spend most of their runtime performing GEMM operations as shown in Figure 2, the efficiency of matrix multiplication directly impacts the training and inference speed of neural networks.

Due to the importance of matrix multiplication in ML and other domains, considerable effort has been devoted to optimizing it on various hardware. GPUs in particular are designed to excel at this operation: they can leverage thousands of parallel threads to perform the multitude of independent multiplications and additions that make up a matrix product. Modern GPUs also feature specialized hardware units (NVIDIA's Tensor Cores [7] for example) that accelerate matrix math at lower precision, further boosting throughput. The heavy reliance of AI on matrix multiplications is one of the reasons GPUs (and other accelerators) have become indispensable in the field – they can deliver dramatically higher arithmetic throughput than a CPU for large matrix sizes, as we shall demonstrate.

III. Goal and Motivation

The primary goal of this work is to compare the performance of a naive matrix multiplication on a high-end CPU versus on a GPU using SYCL, and to understand the extent of acceleration achieved by the GPU. We target a representative large matrix size (2048×2048) to ensure the problem is sufficiently compute-intensive and benefits from parallel hardware. The motivation arises from the expectation that GPUs, with their massively parallel architecture, can significantly outperform CPUs for data-parallel tasks such as matrix multiplication. By measuring and analyzing this difference, we can validate this expectation and underscore the value of GPU acceleration for compute-bound operations.

Practically, developers and researchers often need to decide whether offloading a computation to a GPU (or another accelerator) is worth the overhead of data movement and parallel programming complexity. By using SYCL, which abstracts much of the offloading complexity, we demonstrate how straightforward it can be to target a GPU and achieve huge speedups without writing device-specific code (such as CUDA or vendor-specific APIs).

In summary, the motivation is to affirm the advantage of accelerators for a computation-heavy kernel and to do so using modern, portable C++ techniques. By comparing a naive CPU implementation with an equivalent SYCL GPU implementation, we can clearly see the benefit of GPU offloading and set the stage for exploring further optimizations in the future.

IV. Algorithm Design

Naive CPU Implementation

The CPU implementation of matrix multiplication follows the classic triple-nested loop algorithm, generalized to handle arbitrary-sized matrices. Specifically, given a matrix A of size $M \times K$ and matrix B of size $K \times N$, their product $C = A \cdot B$ will have dimensions $M \times N$. The algorithm computes each entry $C[i][j]$ as the dot product of the i-th row of A and the j-th column of B:

- **for** $i = 0$ **to** $M - 1$:
  - **for** $j = 0$ **to** $N - 1$:
    - Initialize $sum = 0$
    - **for** $k = 0$ **to** $K - 1$:
      - $sum += A[i][k] \times B[k][j]$
    - $C[i][j] = sum$

In our implementation, matrices are stored as one-dimensional flat arrays in row-major order. Thus, index calculations such as $A[i \cdot K + k]$ and $B[k \cdot N + j]$ are used to access the appropriate elements. The algorithm has time complexity $O(M \cdot K \cdot N)$, and for large values (e.g., $M = N = K = 2048$), this translates to roughly 8.6×10^9 floating-point operations. The CPU implementation runs in a single thread and uses no cache tiling, SIMD, or parallelization. This makes it simple and portable, but it is not optimized for performance.

SYCL GPU Implementation

To accelerate matrix multiplication, we offload the computation to a GPU using SYCL. The SYCL kernel implements the same core algorithm but maps the computation onto a 2D grid of threads (work-items), exploiting the massive parallelism available on modern GPUs.

Each work-item is assigned a unique (i, j) coordinate that corresponds to computing a single element $C[i][j]$ in the output matrix. The kernel loops over the shared dimension K internally to compute the dot product between row i of A and column j of B:

$$C[i][j] = \sum_{k=0}^{K-1} A[i][k] \cdot B[k][j]$$

This decomposition allows the outer two loops of the CPU version (over i and j) to be parallelized across the GPU's work-items. While the total number of operations remains $O(M \cdot K \cdot N)$, the SYCL implementation leverages data-parallel execution and hardware thread scheduling to greatly reduce runtime.

Data movement is managed using SYCL buffers and accessors. The workflow is:

- Host data for A and B is copied into SYCL buffers of appropriate sizes ($M \times K$ and $K \times N$ respectively).
- A kernel is launched over a 2D range of size $M \times N$, assigning one thread per output element.
- After kernel execution, the resulting buffer C (size $M \times N$) is read back to host memory.

SYCL ensures correct memory transfers and device synchronization. All matrices remain in flat row-major layout to maintain compatibility between host and device indexing schemes. This layout enables coalesced memory access for rows of A and output C, although access patterns for B may incur performance penalties unless further optimized via shared memory tiling. While this implementation is naive and not optimized for cache locality or occupancy tuning, it demonstrates the foundational performance advantage of GPU-accelerated matrix multiplication using SYCL and forms the basis for further optimization strategies.

Timing and Execution Measurement

We measured the execution time for both the CPU and GPU implementations to evaluate performance. On the CPU side, we use a simple timing method getting high accuracy std::chrono timestamps around the triple-loop function call.

For the GPU/SYCL version, we wanted to specifically measure the kernel execution time on the device. By default, initiating a kernel in SYCL is asynchronous; to get precise timing, we used SYCL's event profiling API. We created the SYCL queue with the property, launched the matrix multiplication kernel, waited for the resulting event, and then queried the event's profiling info. In particular, we obtained the command_start and command_end timestamps of the kernel execution on the GPU. The difference between these gives the time the kernel spent running on the accelerator. It is important to note that this does not include the time to transfer data to or from the device, nor the time spent queuing the operation; it purely measures the device computation time (the duration of the parallel-for execution on GPU). In our experiment, this device execution time is what we report as the "SYCL GPU kernel time." We separately ensure that data transfers (host to device for inputs, and device to host for output) are completed, but we do not add those to the kernel time. In a real-world scenario, the total elapsed time from the host perspective would be slightly higher than the kernel time due to these transfers and any launch overhead. For completeness, we also validated correctness by checking a few output values (for example, $C[0][0]$ as shown later) against an independent calculation, to confirm that the GPU computation produced the same result as the CPU version.

V. Results and Evaluation

The matrix multiplication was performed on two platforms: (1) a CPU (Intel Core i9-12900K, a recent 12th-generation Intel Core with performance and efficiency cores) using only a single thread of execution for the naive algorithm, and (2) a GPU (NVIDIA GeForce RTX 3090) via the SYCL implementation described. The input matrices A and B were randomly generated of size 2048×2048. The output matrix C is of the same dimension. The hardware and performance results are summarized in Table I.

TABLE I
Matrix Multiplication Performance Comparison (2048×2048 matrices)

Host CPU:	Intel Core i9-12900K (32GB DDR5 RAM)
Device GPU:	NVIDIA RTX 3090 (24GB GDDR6X)
Matrix size:	2048×2048 (single precision)
CPU execution time:	6518 ms (single-threaded)
SYCL GPU kernel time:	15.8628 ms
SYCL device used:	NVIDIA RTX 3090 (CUDA backend)
Result validation:	$C[0][0] = 41252$ (correct)

As shown above, the CPU took on the order of several seconds to compute the 2048^2 matrix product, whereas the GPU kernel completed the computation in only 15.86 milliseconds. This is an astonishing difference: the raw compute kernel on the GPU is about $6518 \text{ ms}/15.8628 \text{ ms} \approx 411$ times faster than the simple CPU implementation for this problem size. In other words, what the CPU accomplished in over six seconds, the GPU did in well under one hundredth of a second. This kind of speedup is in line with expectations for large matrix operations: GPUs excel at such workloads and their advantage grows with problem size.

It is important to emphasize that the GPU timing we report is the kernel execution time on the device and does not include data transfer overhead. In our case, transferring two 2048^2 input matrices to the GPU and retrieving the result involves on the order of tens of millions of elements. If each element is 8 bytes (double precision), that's on the order of 128 MB total data to transfer (inputs + output). Even on a high-bandwidth PCIe bus, this might take on the order of tens of milliseconds. However, SYCL's asynchronous execution and our use of profiling means that the 15.86ms measured is only the time the GPU spent crunching numbers, not waiting for memory. The result $C[0][0] = 41252$ (an example element of the output) was checked against the CPU computation to ensure correctness, confirming that both implementations produce identical results.

Overall, the evaluation strongly illustrates the benefit of GPU acceleration for matrix multiplication. The speedup factor of $\sim$411× is even higher than one might initially guess, but it should be noted that our CPU implementation is single-threaded and non-vectorized, whereas modern CPUs could achieve higher performance with multithreading (e.g., using all cores of the i9-12900K) and SIMD instructions. Nevertheless, even if the CPU gained a 10× improvement from such optimizations, the GPU would still be tens of times

faster. Our findings thus align with the common understanding that offloading large matrix operations to GPUs is highly advantageous for performance. For tasks where data transfer overhead is amortized by heavy computation (as is the case here, where $O(N^3)$ operations far outweigh $O(N^2)$ memory elements), the GPU essentially dominates the computation.

VI. Discussion and Future Directions

The experimental results confirm that using a GPU via SYCL dramatically accelerates matrix multiplication, meeting or exceeding our initial expectations. Even if a multi-core CPU or hand-optimized library could reduce the gap somewhat, the fundamental advantage of the GPU's architecture remains clear. SYCL proved to be an effective vehicle for tapping into that performance with minimal code changes – we wrote a parallel kernel in C++ and the runtime handled dispatching it to the CUDA-capable device. This confirms that high-level models like SYCL can indeed realize the low-level performance of accelerators while maintaining code portability.

Beyond verifying the speedup, it's worthwhile to discuss how one might push the performance even further or apply similar techniques to more complex scenarios. Our implementation was intentionally naive, focusing on clarity and demonstrating the baseline gap. In practice, many optimizations can greatly improve matrix multiplication efficiency on both CPUs and GPUs. We briefly outline some of these strategies and future directions:

a) Matrix Chain Multiplication Optimization:: In scenarios where multiple matrix multiplications must be performed in sequence (i.e., multiplying more than two matrices, e.g. $A \times B \times C \times D$), the order of multiplication can make a huge difference in the total work. Matrix multiplication is associative, so one can parenthesize the product in different ways that change the intermediate matrix dimensions. The classic Matrix Chain Multiplication problem is to find the most efficient way to multiply a chain of matrices by minimizing the scalar multiplication operations. A dynamic programming solution (also known as the Matrix Chain Order algorithm [8]) can determine the optimal parenthesize for the chain, yielding a lower overall cost than naive left-to-right multiplication. This does not speed up a single matrix multiplication, but it significantly optimizes programs that perform many matrix multiplies together. Most optimizations using this method are performed by the compiler or graph optimizers such as cuBLAS [9], instead of being user implemented.

b) Tiling and Block Matrix Multiplication:: A key optimization for matrix multiply on modern hardware is tiling (also known as blocking). Instead of computing one element of C at a time (which re-reads an entire row of A and column of B for each element), a tiled algorithm works on submatrices (blocks) so that data loaded from main memory can be reused for multiple calculations. In a tiled approach, one might partition C into, say, 16×16 tiles and treat A and B accordingly. A thread block on the GPU would then load a tile of A and a tile of B into fast on-chip memory and compute the partial product tile of C before moving on. This improves locality: each element of A and B gets used in multiple dot-products for the tile. In fact, GPU implementations of GEMM inherently partition the output matrix into tiles assigned to thread blocks [1]. By doing so, they maximize data reuse and ensure memory accesses are coalesced and aligned for efficiency. Our current SYCL kernel did not employ tiling explicitly – each work-item worked on a single C element, causing a lot of redundant reads – so there is room for improvement.

c) Batched and Parallel Computation of Multiple Multiplications:: Another strategy is to leverage batched matrix multiplication when we have to perform many small or moderate-size matrix multiplies. In deep learning, for instance, one often processes inputs in batches (say 32 or 64 inputs at a time) for efficiency. This means effectively computing many independent matrix multiplies (one per input in the batch) simultaneously. Launching a separate kernel for each might incur overhead and underutilize resources if each matrix is not very large. Instead, "batched GEMM" routines or kernels take an array of matrix pointers and perform dozens or hundreds of multiplies in one go, in a single coordinated device launch. The idea can be applied in our SYCL context as well: if we needed to multiply, say, 100 matrices of size 512×512, a batched kernel could do this more efficiently than looping 100 times on the host submitting a kernel for each.

d) Other Optimizations:: There are several additional optimization avenues worth mentioning. One is exploiting vectorization on the CPU and GPU – for instance, using SIMD instructions on CPU (AVX-512 on the i9-12900K) or using SYCL's sub-group operations on the GPU to utilize vector units. Another is to take advantage of specialized hardware: newer GPUs have matrix engines (like NVIDIA's Tensor Cores or AMD's Matrix Cores) that can multiply small matrices (e.g., 4×4 or 16×16) in one operation. SYCL is extended (through oneAPI math libraries) to allow using such Tensor Core operations. Incorporating those could drastically speed up mixed-precision matrix multiplication beyond even what we achieved.

VII. Conclusion

Overall, our study provides a baseline confirmation of the acceleration gained by moving matrix multiplication to a GPU using SYCL. The GPU's speedup met expectations and highlights why heterogeneous computing is crucial for large-scale machine learning and numerical simulations. In future work, applying the discussed optimizations (tiling, batching, etc.) within SYCL would be the next step to approach peak performance. Additionally, combining algorithm-level improvements (like optimal matrix chain ordering to reduce the number of multiplications needed) with low-level tuning can yield compound benefits. The continued evolution of SYCL and oneAPI is making it easier to write such optimized code in a platform-independent manner.

REFERENCES

[1] NVIDIA, "Matrix multiplication background user's guide," https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, 2024, accessed 15-May-2025.

[2] Saharsa, "The command_submit in the sycl::info::event_profiling: submits the whole code or just the parallel-for?" https://stackoverflow.com/questions/67742015/the-command-submit-in-the-syclinfoevent-profiling-submits-the-whole-code-or, 2021, accessed 15-May-2025.

[3] T. K. Group, "SYCL 2020 specification (revision 10)," https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2020, accessed 15-May-2025.

[4] S. Boehm, "How to optimize a cuda matmul kernel for cublas-like performance: a worklog — siboehm.com," https://siboehm.com/articles/22/CUDA-MMM#:~:text=Matrix%20multiplication%20on%20GPUs%20may,use%20the%20tensor%20cores%2C%20which, 2022, accessed 15-05-2025.

[5] K. Group, "Sycl: Single-source c++ programming for heterogeneous systems," 2020, https://www.khronos.org/sycl/.

[6] I. Corporation, "oneapi: A new era of heterogeneous computing," https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html, 2023, accessed 15-May-2025.

[7] NVIDIA, "Nvidia tensor cores," 2021, https://developer.nvidia.com/tensor-cores.

[8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[9] N. Corporation, "Using cublas in deep learning frameworks," https://docs.nvidia.com/deeplearning/frameworks/support-matrix/index.html, 2021, accessed 15-May-2025.