Optimizing DGEMM Using Vectorized Micro-Kernels and Memory-Aware Parallelization

Arman Hovhannisyan
National Polytechnic University of Armenia
Yerevan, Armenia
e-mail: a.hovhannisyan@polytechnic.am

***Abstract*—This paper presents a high-performance implementation of the double-precision general matrix-matrix multiplication (DGEMM) operation, optimized through vectorized micro-kernels, two-level cache blocking, memory alignment, and multithreading techniques. The proposed approach leverages AVX2, fused multiply-add (FMA) instructions and a custom 6×8 micro-kernel to maximize instruction-level parallelism and register utilization. A memory-aware blocking strategy is employed to fit submatrices within L1 and L2 cache hierarchies, reducing cache misses and improving data locality. Thread-level parallelism is implemented using POSIX threads (Pthreads) with explicit workload distribution and memory affinity control. Empirical evaluations on a 12-core x86_64 architecture demonstrate a performance advantage, achieving multiplication of 4096×4096 matrices with a 13.3% increase in speed compared to the standard NumPy implementation. The results highlight the impact of fine-grained control over threading, memory, and instruction scheduling on achieving superior performance in dense linear algebra operations.**

***Keywords*—DGEMM, matrix multiplication, multithreading, high-performance computing, optimization.**

I. Introduction

Matrix multiplication is a foundational operation in scientific computing, graphics processing, machine learning, and numerical linear algebra. Among its most widely used formulations is the double-precision general matrix-matrix multiplication (DGEMM), which computes $C = \alpha AB + \beta C$, where A, B, and C are dense matrices. Due to its high computational cost and widespread use, significant research effort has been devoted to optimizing DGEMM for modern CPU architectures.

Modern processors offer extensive hardware features, such as SIMD (Single Instruction, Multiple Data) instruction sets like AVX2, multi-core parallelism, and deep memory hierarchies. However, achieving near-peak performance on such architectures requires a deep understanding of low-level optimization strategies, including vectorization, cache blocking, memory alignment, and thread-level parallelism. High-performance libraries such as Intel MKL and OpenBLAS have demonstrated the potential of such techniques, but these implementations often lack fine-grained control, especially for performance tuning and educational purposes.

This paper presents a custom DGEMM implementation that explicitly exploits AVX2 vectorization and fused multiply-add (FMA) instructions via a 6×8 micro-kernel, combined with a two-level blocking strategy aligned with L1 and L2 cache sizes. Thread-level parallelism is implemented using POSIX threads (Pthreads), allowing explicit control over workload distribution and memory locality. This approach is also compared with the highly optimized NumPy implementation (which internally uses BLAS), using large matrices of size 4096×4096 as a benchmark.

II. Related Work

Optimizing dense matrix multiplication has been a central topic in high-performance computing (HPC) for decades, owing to its ubiquity in scientific simulations, machine learning, and numerical methods. Numerous libraries and frameworks have been developed to provide optimized DGEMM implementations, with a focus on exploiting the full computational capacity of modern processors.

One of the most influential efforts is the Basic Linear Algebra Subprograms (BLAS) interface, particularly Level 3 routines such as DGEMM. Highly tuned BLAS libraries like Intel Math Kernel Library (MKL), OpenBLAS, and ATLAS leverage advanced features of the CPU, including SIMD vectorization, multi-threading, and cache-aware blocking. The GotoBLAS architecture [1], which introduced a register-level blocking strategy and loop restructuring, laid the groundwork for many of these libraries.

In terms of parallelization strategies, OpenMP is widely adopted for its simplicity and portability, offering a high-level abstraction for shared-memory parallelism. However, OpenMP's scheduling overhead and limited control over thread affinity can hinder fine-tuned performance in certain scenarios [2]. Alternatively, POSIX threads (Pthreads) provide low-level access to thread creation and synchronization primitives, enabling more precise management of workload distribution and memory locality, albeit at the cost of increased programming complexity.

Vectorization is another key factor in high-performance DGEMM implementations. Modern CPUs support wide SIMD instructions (e.g., AVX2, AVX-512), which can process multiple data elements in parallel. Studies such as [3, 4, 5, 6] have shown that manually crafted vectorized micro-kernels, when paired with appropriate loop unrolling and register blocking techniques, can significantly outperform compiler-generated SIMD code. These approaches allow fine-grained control over instruction scheduling, register reuse, and memory alignment, which are often lost in automatic code generation due to conservative compiler heuristics. These findings underscore the continued relevance of manual optimization, especially in scenarios where peak performance and architectural efficiency are paramount—such as HPC, scientific computing, and performance-critical numerical kernels.

Memory hierarchy also plays a critical role in performance [7]. Cache blocking, also known as tiling, is a well-known technique for improving the temporal and spatial locality of memory accesses. Works such as [8] demonstrate how multi-level blocking strategies, aligned with L1, L2, and even L3 cache sizes, can drastically reduce cache misses and improve throughput. By carefully partitioning matrices into tiles that fit into the different layers of the CPU's memory hierarchy, these techniques minimize costly memory accesses and increase temporal locality.

While libraries like NumPy internally rely on optimized BLAS backends, their performance is ultimately constrained by the general-purpose nature of those libraries. In contrast, custom implementations allow for architecture-specific tuning and provide valuable insights into performance bottlenecks and optimization strategies.

This paper builds upon the aforementioned techniques by integrating AVX2 vectorized micro-kernels, two-level blocking for cache efficiency, memory alignment via posix_memalign, and fine-grained parallelism using Pthreads. This work contributes to the ongoing study of low-level optimization for DGEMM by quantitatively comparing it against high-level approaches and demonstrating the performance gains achieved through full-stack optimization.

III. Details

To achieve high performance in double-precision matrix multiplication, a custom DGEMM routine was implemented, optimized across multiple architectural layers, including instruction-level parallelism, memory hierarchy, and thread-level concurrency. This section outlines the core techniques used: vectorized micro-kernel design with AVX2 and FMA, multi-level cache blocking, aligned memory allocation, and multithreading with Pthreads.

A. *Two-Level Cache Blocking*

To efficiently utilize the memory hierarchy, the matrices are partitioned into panels using two-level blocking (Fig. 1):

- Outer blocks: parameters KC, MC, and NC are tuned to fit working blocks of A, B, and C into the L2 cache.
- Inner blocks: within each panel, the micro-kernel operates on L1-cache-resident subblocks of size 6×8.

This technique reduces cache misses and increases temporal locality, ensuring that reused data stays in cache during inner loops. Block sizes were experimentally tuned based on the target CPU's cache sizes and associativity to avoid conflict misses.

B. *Vectorized Micro-Kernel with AVX2 FMA*

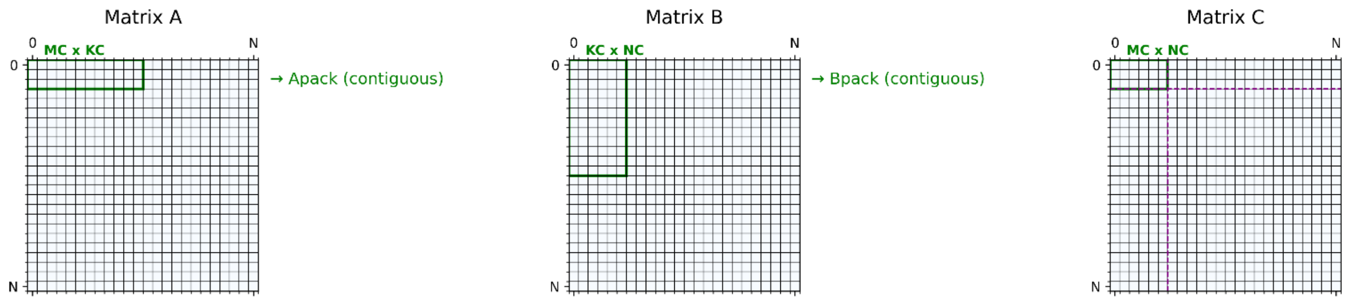
At the core of the implementation is a 6×8 register-level micro-kernel designed to utilize AVX2 instructions and fused multiply-add (FMA) operations (Fig. 2). Each micro-kernel computes a 6×8 block of the result matrix C by loading and broadcasting rows of A and performing vectorized FMA with corresponding columns of B. The use of AVX2 intrinsics allows simultaneous computation on 4 double-precision floating-point numbers, enabling a total of 24 operations per FMA instruction cycle.

AVX2 has 16 YMM registers. Since each YMM register can accommodate four double-precision floating-point values, executing a 6x8 tile requires 12 registers (6 * (8 / 4) = 12). Three registers are allocated for temporary values (a, b0, b1), leaving only a single register spare, which makes the 6x8 tile size a well-suited configuration.

Loop unrolling is applied to maximize instruction throughput, and careful register allocation ensures minimal spilling. The kernel minimizes loads by reusing values in registers and performs blind stores to write results without unnecessary reads, improving store throughput.

C. *Memory Alignment*

To ensure efficient vector loads and reduce cache line splits, all matrices are allocated using posix_memalign, guaranteeing alignment to 32-byte boundaries, which is required by AVX2 loads and stores. Aligned memory enables the use of _mm256_load_pd and _mm256_store_pd instructions without penalties.



```
for jc = 0 to N step NC:  # Column panels of B and C
    for pc = 0 to N step KC:  # Depth (K dimension)
        Pack B[pc:pc+kc, jc:jc+nc] into Bpack
        for ic = 0 to N step MC:  # Row panels of A and C
```

Fig. 1. Two-Level Cache Blocking

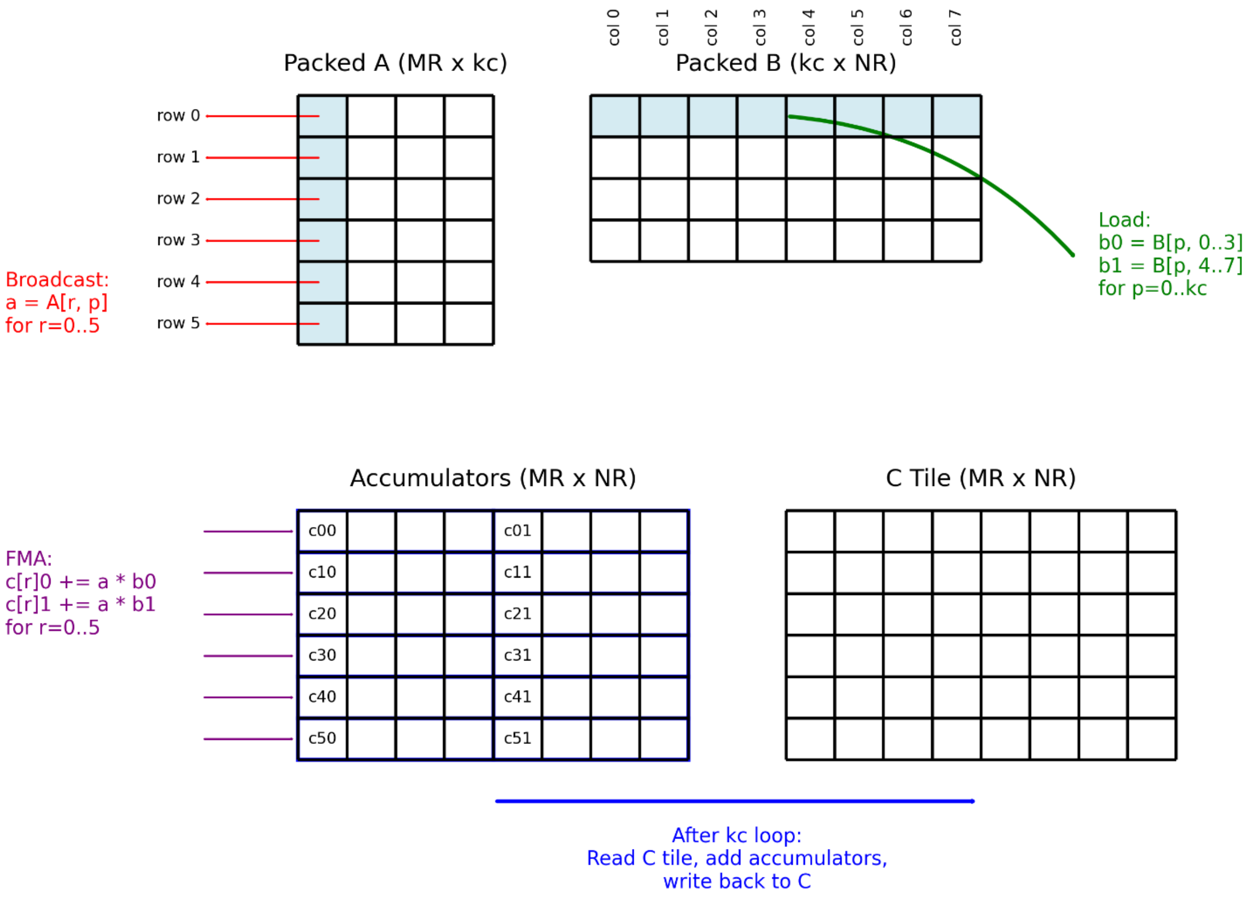


Fig. 2. Vectorized Micro-Kernel with AVX2 and FMA

Additionally, memory padding is applied to avoid false sharing and alignment conflicts across cache lines, especially in multi-threaded scenarios.

D. *Thread-Level Parallelism with Pthreads*

Parallelization is achieved using POSIX threads, with a static workload partitioning strategy. The outermost loop over matrix rows is divided equally among threads, each operating on disjoint memory regions of matrix C. This eliminates the need for synchronization primitives during computation and ensures thread safety.

Thread affinity is optionally set to improve data locality on NUMA systems. The manual thread management provided by Pthreads offers finer control compared to OpenMP, enabling optimizations such as minimizing context switches and avoiding oversubscription.

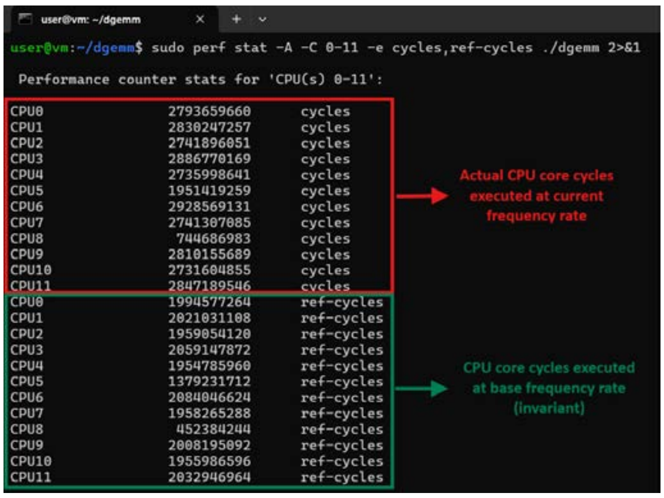
E. *Compiler Optimizations*

The program is compiled using the GCC compiler with aggressive optimization flags: *-O3 -march=native -ffast-math -mfma -funroll-loops -pthread*

These flags enable vectorization, loop unrolling, instruction scheduling, and utilization of FMA units. The -march=native flag ensures that the compiler targets the host machine's specific architecture, unlocking hardware-specific instructions and optimizations.

F. *Results*

The implementation is evaluated on an Intel Core i7 10750H CPU running WSL2 Ubuntu 22.04 OS. Performance profiling is conducted using the perf tool to obtain detailed execution metrics. Fig. 3 shows the actual CPU core cycles executed at current frequency rate (cycles) and CPU core cycles executed at base frequency rate (ref-cycles). Given a base frequency of 2.6 GHz, these measurements enable a direct calculation of the effective CPU frequency, illustrated in Fig. 4. The analysis shows that the processor sustained an average operating frequency of approximately 3.6 GHz during execution.



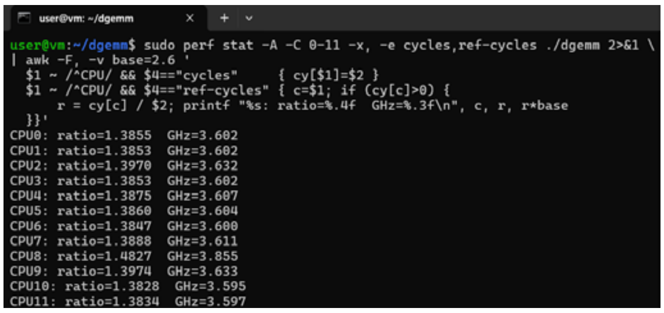
user@vm: ~/dgemm
Actual CPU core cycles executed at current frequency rate
CPU core cycles executed at base frequency rate (invariant)

```
user@vm:~/dgemm$ sudo perf stat -A -C 0-11 -e cycles,ref-cycles ./dgemm 2>&1

 Performance counter stats for 'CPU(s) 0-11':

CPU0                   2793659660      cycles
CPU1                   2830247257      cycles
CPU2                   2741896051      cycles
CPU3                   2886770169      cycles
CPU4                   2735998641      cycles
CPU5                   1951419259      cycles
CPU6                   2928569131      cycles
CPU7                   2741307085      cycles
CPU8                    744686983      cycles
CPU9                   2810155689      cycles
CPU10                  2731604855      cycles
CPU11                  2847189546      cycles
CPU0                   1994577264      ref-cycles
CPU1                   2021031108      ref-cycles
CPU2                   1959054120      ref-cycles
CPU3                   2059147872      ref-cycles
CPU4                   1954785960      ref-cycles
CPU5                   1379231712      ref-cycles
CPU6                   2084046624      ref-cycles
CPU7                   1958265288      ref-cycles
CPU8                    452384244      ref-cycles
CPU9                   2008195092      ref-cycles
CPU10                  1955986596      ref-cycles
CPU11                  2032946964      ref-cycles
```

Fig. 3. Cycles per logical processor



```
user@vm:~/dgemm$ sudo perf stat -A -C 0-11 -x, -e cycles,ref-cycles ./dgemm 2>&1 \
| awk -F, -v base=2.6 '
  $1 ~ /^CPU/ && $4=="cycles"     { cy[$1]=$2 }
  $1 ~ /^CPU/ && $4=="ref-cycles" { c=$1; if (cy[c]>0) {
      r = cy[c] / $2; printf "%s: ratio=%.4f  GHz=%.3f\n", c, r, r*base
  }}'
CPU0: ratio=1.3855  GHz=3.602
CPU1: ratio=1.3853  GHz=3.602
CPU2: ratio=1.3970  GHz=3.632
CPU3: ratio=1.3853  GHz=3.602
CPU4: ratio=1.3875  GHz=3.607
CPU5: ratio=1.3860  GHz=3.604
CPU6: ratio=1.3847  GHz=3.600
CPU7: ratio=1.3888  GHz=3.611
CPU8: ratio=1.4827  GHz=3.855
CPU9: ratio=1.3974  GHz=3.633
CPU10: ratio=1.3828  GHz=3.595
CPU11: ratio=1.3834  GHz=3.597
```

Fig. 4. Frequencies per logical processor

The DGEMM operation involves $2n^3$ floating point operations (FLOPs). Using the total FLOP count together with the measured average CPU frequency and an execution time of 0.85 seconds, FLOPs per core per cycle can be calculated, as presented in equation (1).

$$c = \frac{FLOPs}{t * f * cores} = \frac{2 * 4096^2}{0.85 * 3.6 * 10^9 * 6} \approx 7.5 \quad (1)$$

The AVX2 YMM register accommodates four double-precision (fp64) values. Each fused multiply-add (FMA) instruction performs two FLOPs. The throughput of FMA instructions on the Comet Lake microarchitecture is 0.5 cycles, meaning that two FMA instructions can be issued per cycle [9]. Consequently, the theoretical peak performance is 4×2×2=16 FLOPs per core per cycle. The implementation achieves 46.8% of this theoretical maximum.

The implementation was further compared against the NumPy library (backed by the OpenBLAS implementation). Experimental results (Fig. 5) indicate that NumPy completes the 4096×4096 DGEMM in 0.98 seconds, whereas the proposed implementation achieves the same operation in 0.85 seconds, corresponding to a 13.3% performance improvement.

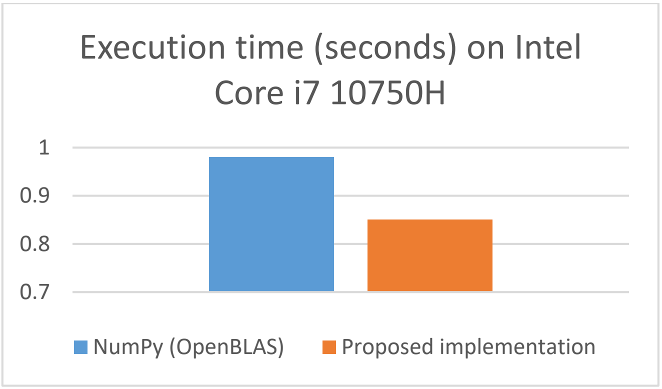


Fig. 5. Execution time comparison

IV. CONSLUSION

This paper presents a highly optimized implementation of the double-precision general matrix multiplication (DGEMM) operation that leverages low-level architectural features of modern CPUs, including AVX2 vector instructions, fused multiply-add (FMA) micro-kernels, cache-aware blocking, memory alignment, and multithreaded execution using Pthreads.

Experimental results demonstrate that meticulous control over data movement, memory layout, and thread scheduling can lead to substantial performance gains. Specifically, on the Intel Core i7 10750H CPU, the implementation outperforms a widely used NumPy BLAS backend, achieving a 13.3% performance improvement.

This work reinforces the effectiveness of combining architecture-aware blocking strategies with custom SIMD kernels and thread-level optimizations for compute-intensive workloads. The implementation is available at [10].

REFERENCES

[1] K. Goto and R. Van De Geijn, “High-performance implementation of the Level-3 BLAS,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 1, pp. 1–14, 2008.

[2] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey “The Design of OpenMP Thread Affinity”, *Proceedings of IWOMP 2012, Lecture Notes in Computer Science,* vol 7312. Springer, Berlin, Heidelberg, pp.15-28, 2012.

[3] S. Boehm, (2022) Fast Multidimensional Matrix Multiplication on CPU from Scratch. [Online]. Available: https://siboehm.com/articles/22/Fast-MMM-on-CPU

[4] X. Su, X. Liao and J. Xue, “Automatic generation of fast BLAS3-GEMM: A portable compiler approach,” *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Austin, TX, USA, pp. 122-133, 2017.

[5] H. Martínez, S. Catalán, F.D. Igual, J.R. Herrero, R. Rodríguez-Sánchez, and E.S. Quintana-Ortí, “Co-Design of the Dense Linear Algebra Software Stack for Multicore Processors,” 2023, arXiv:2304.14480.

[6] G. Alaejos, A. Castelló, H. Martínez, P. Alonso-Jordá, F. D. Igual, and E. S. Quintana-Ortí, “Micro-kernels for portable and efficient matrix multiplication in deep learning,” *The Journal of Supercomputing*, vol. 79, pp. 8124–8147, 2023.

[7] U. Drepper, “What every programmer should know about memory”, 2007. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[8] S. Williams et al., “Optimization of sparse matrix–vector multiplication on emerging multicore platforms,” *Proceedings of SC'07*, pp. 1–12, 2007.

[9] A. Abel and J. Reineke, “uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures”, *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, pp. 673-686, 2019.

[10] Proposed implementation, [Online]. Available: https://github.com/aahovhannisyan/dgemm