Project 10 Report: Low-Precision GEMM for Neural Networks using AVX-512

Student: Sebastian Fia (ID: 244231)
Email: sebastian.fia@unitn.it
Course: Introduction to Parallel Computing (2025–2026)

***Abstract*—This project presents a parallel implementation of General Matrix-Matrix Multiplication (GEMM) optimized for deep learning workloads on multicore CPUs. We employ a cache-aware tiling strategy enhanced with AVX-512 intrinsics to support `bf16` and `int8` data types, targeting the high-throughput requirements of modern Neural Networks. The `bf16` kernel utilizes register-level upcasting to `fp32`, while the `int8` kernel exploits VNNI instructions with dynamic range compensation. We evaluate our approach against the industry-standard oneDNN library using matrices derived from real-world models (ResNet, BERT, GPT-2). We run the experiments on a 32 core node equipped with an Intel Xeon Gold 6252N CPU. Results demonstrate the effectiveness of static load balancing and manual vectorization, highlighting the trade-offs between precision and performance. Source code and reproducibility artifacts are provided.**

***Index Terms*—Parallel Computing, AVX-512, OpenMP, GEMM, Neural Networks, Quantization, oneDNN.**

I. Introduction

Matrix multiplication is the dominant kernel in deep learning workloads, often consuming the majority of inference cycles. While training relies on `fp32`, inference is increasingly shifting towards lower precision formats like `bf16` (Brain Floating Point) [1] and `int8` [2] to maximize throughput and reduce memory bandwidth pressure.

This project aims to:

1) Implement a highly optimized, tiled GEMM kernel using C++ and AVX-512 intrinsics.
2) Develop specialized micro-kernels for `bf16` (via upcasting) and `int8` (via VNNI compensation).
3) Analyze performance characteristics comparing the implementation against Intel's oneDNN baseline [3].

II. Contribution and Methodology

We adopt a classic six-loop tiling optimization approach, condensed into a packing-based strategy to maximize L1/L2 cache hits, following the principles described by Goto and van de Geijn [4].

A. Algorithm Overview

Our implementation is structurally agnostic to the data type. We decompose the problem $C = A \times B$ into macro-tiles that fit into the L2 cache, and further into micro-tiles ($MR \times NR$) that fit into ZMM registers.

Algorithm 1 illustrates the parallel strategy. To ensure contiguous memory access during the hot loop, we pack operands A and B into temporary buffers (A_{packed}, B_{packed}). A is packed into strips of height MR (stored column-wise), and B is packed into strips of width NR (stored row-wise).

Algorithm 1 Parallel Tiled GEMM (Dtype Agnostic)

Require: Matrices $A(M \times K)$, $B(K \times N)$, $C(M \times N)$

Determine split dimension (M or N) based on aspect ratio
if Parallelize N (Typical case) **then**
  Start parallel region (#pragma omp parallel)
  Allocate thread-local A_{packed}, B_{packed}
  Calculate block ranges j_{start}, j_{end}
  #pragma omp for schedule(static)
  for j_{block} from 0 to N step NC **do**
    for k_{block} from 0 to K step KC **do**
      PackB($B \to B_{packed}$)
      for i_{block} from 0 to M step MC **do**
        PackA($A \to A_{packed}$)
        // Micro-kernel Loop
        for $i \in [i_{block}, i_{block} + MC)$ step MR **do**
          for $j \in [j_{block}, j_{block} + NC)$ step NR **do**
            MicroKernel($A_{pack}, B_{pack}, C_{sub}$)
          end for
        end for
      end if
    end for
  end for
end if

B. Parallelization Strategy

We utilize OpenMP [5] for threading. The matrices used in deep learning often have varying aspect ratios (e.g., tall-and-skinny vs. square). We implement a dynamic check: if $M > N$, we parallelize the outer loop over M; otherwise, we parallelize over N (Algorithm 1).

We explicitly use `#pragma omp for schedule(static)` with the default chunk size.

- **Regularity:** GEMM is highly regular; the workload per block is identical. Dynamic scheduling (guided or dynamic) would introduce unnecessary overhead without benefit.
- **Locality:** Our implementation already iterates through explicit cache tiles, so we can avoid tuning the omp chunk size. A static schedule ensures that threads process

contiguous chunks of the output matrix C, minimizing false sharing and maximizing packing efficiency.

C. BF16 Implementation: Register Upcasting

Native `bf16` support in hardware is not assumed. Instead, we perform operations in `fp32` using standard AVX-512 registers.

- **Load:** We load contiguous blocks of `bf16` data.
- **Upcast:** Since `bf16` is structurally identical to the upper 16 bits of an IEEE-754 `fp32`, we do not need expensive conversion instructions. We simply load the 16-bit integer values into a 512-bit register and shift them left by 16 bits (`_mm512_slli_epi32`). This effectively aligns the mantissa and exponent to the `fp32` format.
- **Compute:** Fused Multiply-Add (FMA) is performed in full `fp32` precision.

D. INT8 Implementation: VNNI & Compensation

For `int8`, we rely on the AVX-512 VNNI instruction `_mm512_dpbusd_epi32` [6]. This instruction computes the dot product of unsigned bytes ($u8$) and signed bytes ($s8$), accumulating into 32-bit integers ($s32$).

However, neural network weights and activations are typically both signed ($s8$). To use the hardware instruction which expects one unsigned operand, we apply a transformation to matrix A:

$$A_{u8} = A_{s8} + 128 \tag{1}$$

Substituting this into the dot product equation $C \mathrel{+}= \sum A_{s8} \cdot B_{s8}$:

$$C \mathrel{+}= \sum (A_{u8} - 128) \cdot B_{s8} \tag{2}$$

$$C \mathrel{+}= \underbrace{\sum (A_{u8} \cdot B_{s8})}_{\text{VNNI Instruction}} - \underbrace{128 \sum B_{s8}}_{\text{Compensation}} \tag{3}$$

Compensation Logic: During the `PackB` routine, we precalculate the row-sums of B for every column. This vector is passed to the micro-kernel, which subtracts $(128 \times \text{row\_sum})$ from the accumulator registers before storing the result. This allows us to use high-throughput unsigned arithmetic for signed data.

III. Experiments and System Description

A. Experimental Setup

The experiments are conducted on a 32 core node equipped with an Intel Xeon Gold 6252N CPU, supporting AVX-512. The operating system is CentOS Linux 7 (Core). We compile using the oneAPI DPC++ Compiler 2021.2.0. For parallelization, we use OpenMP (version 5.0). The baseline is Intel's oneDNN (2021.2.0), a highly optimized library offering high performance neural networks primitives [3].

We select 5 representative weight tensors from HuggingFace, corresponding to layers in ResNet, ViT, BERT, TinyLlama, and GPT-2 (Table I), and we generate synthetically matrices with the correct shape and data type. Doing so we cover diverse aspect ratios, testing the robustness of our parallelization strategy. See the Reproducibility section for more details about the matrices.

For each timing, we return the median time over 100 measures.

Note that the baseline against which we compare our `bf16` implementation is OneDNN's `fp32` kernel. This is because our hardware does not support native `bf16` instructions.

TABLE I
Benchmark Matrix Shapes (Generated from NN Layers)

Model	Weight	Rows (M)	Cols (N)
ResNet-50	classifier FC	1000	2048
ViT-Base	MLP up, layer 0	768	3072
BERT-Base	Attention Query, layer 0	768	768
TinyLlama	MLP down, layer 0	5632	2048
GPT-2	WTE weight	50257	768

B. Results and Discussion

1) Strong Scaling: Figures 1 and 2 illustrate the strong scaling performance for a fixed problem size ($M = 256$, Llama MLP layer).

- We observe near-linear speedup up to 16 threads in both cases.
- Beyond 32 threads, efficiency degrades due to hyperthreading, resource contention and memory bandwidth saturation.
- For both data types, our speedup curves are comparable to OneDNN's ones, so we can confidently conclude that our parallelization strategy is highly competitive, since we maintain a strong relative speedup (although in absolute terms our implementation is weaker than OneDNN's highly optimized one).

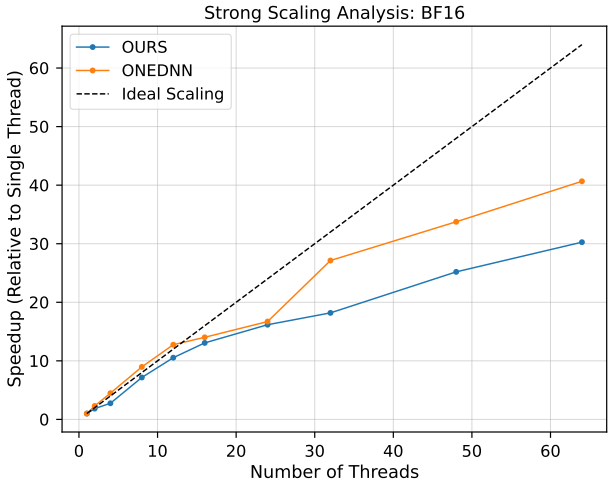

Fig. 1. Strong Scaling Analysis for `bf16` ($M = 256$).

2) Batch size scaling: Figures 3 and 4 sweep the batch size M to benchmark our implementation in different regimes (for example, during inference most neural network architectures require a batch size of 1, whereas during training higher batch

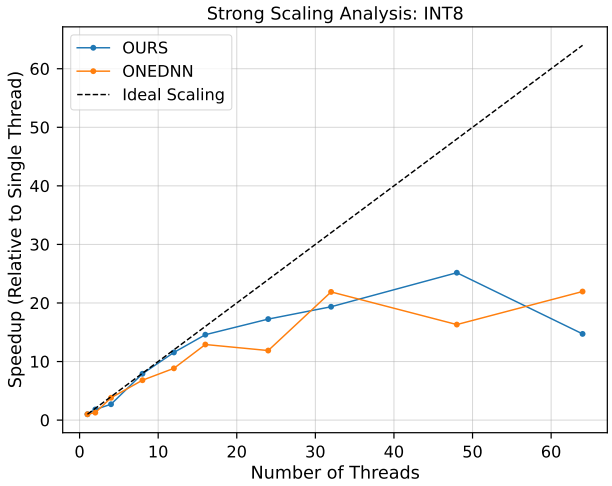

Fig. 2. Strong Scaling Analysis for `int8` ($M = 256$).

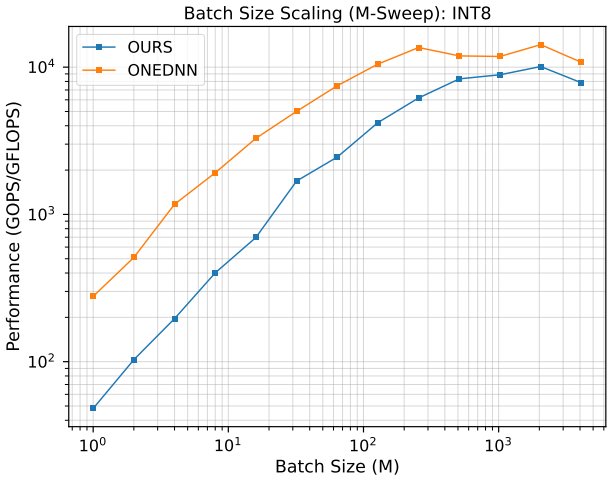

Fig. 4. Batch Size Scaling for `int8`. Comparison vs oneDNN.

sizes are more common). We keep a fixed number of 32 threads.

- **Memory Bound**: For lower M, in both cases increasing the batch size provides notable throughput increases, due to increasingly better cache re-utilization and lower pressure on memory bandwidth.
- **Compute Bound**: For higher M, performance starts to plateau. This is because we almost saturate the computing capabilities of the hardware. The `int8` kernel (Figure 4) achieves a significantly higher GFLOPS/GOPS plateau than `bf16` (Figure 3), confirming the expected 4x throughput advantage of 8 bit VNNI instructions versus the 32 bit AVX-512 FMA.

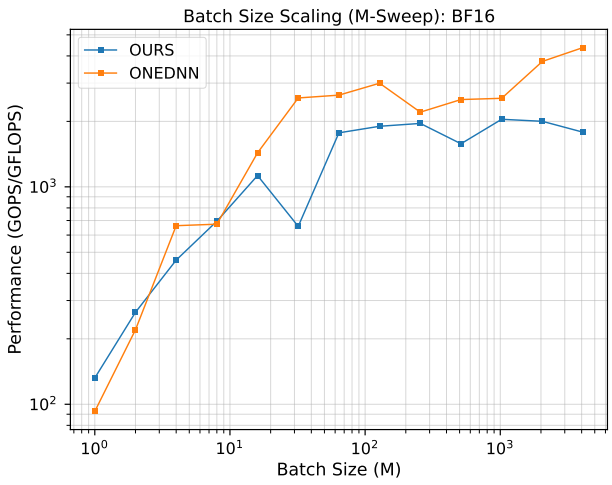

Fig. 3. Batch Size Scaling for `bf16`. Comparison vs oneDNN.

3) Workload Diversity: Figure 5 compares performance across different matrix shapes, using a fixed number of 32 threads. On average, especially for `int8`, our implementation is notably worse than OneDNN's one across the various workloads. We suppose that the main reason for this is that our packing functions are not highly optimized, and do not use AVX-512 intrinsics, since we mostly focused on optimizing the micro-kernel. Occasionally our `bf16` performance is comparable or even superior: this is due to the reduced pressure on memory bandwidth, since our implementation stores the matrices in 16 bits and upcasts only at register level, whereas the baseline stores the matrices in 32bits.

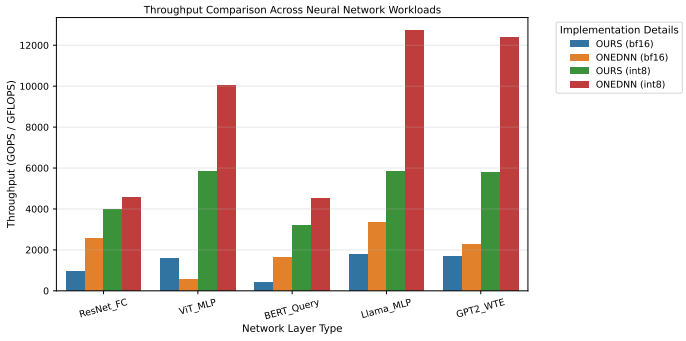

Fig. 5. Workload Diversity. Performance impact of varying aspect ratios ($K \times N$) on cache blocking efficiency.

IV. Conclusions

We successfully implemented a tiled parallel GEMM supporting `bf16` and `int8`. Our experiments validate that register-level upcasting allows effective `bf16` processing on AVX-512 hardware without native support, while the compensation-based `int8` kernel significantly outperforms higher precision variants. While the oneDNN baseline remains superior, our implementation achieves competitive scalability and demonstrates the critical role of cache-blocking and vectorization in deep learning primitives. Future work based on our findings might explore similar questions about GEMM utilizing other low precision data types, such as `fp4` or `int4`.

V. Reproducibility

The project artifacts, including source code and build scripts, are available at: https://github.com/SebastianFia/PARCO-Computing-2026--244231-.git

Detailed instructions for compilation and running the experiments are provided in the repository README.md, which also contains more thorough information about the matrices we used.

References

[1] D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[2] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 2704–2713.

[3] *oneAPI Deep Neural Network Library (oneDNN) Developer Guide and Reference*, Intel Corporation, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html

[4] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, 2008.

[5] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, Intel Corporation, 2021, Order Number: 325462-075US.