Optimizing Single Precision Matrix Multiplication on a CPU

Sangeen Khan

December 31, 2025

1 Overview

This report presents a small study of dense matrix multiplication $C \leftarrow AB$ in single precision (SGEMM) on a laptop class CPU. The goal is to implement a set of progressively optimized kernels and benchmark them against OpenBLAS on the same inputs and data layout. The report also includes a cache locality analysis using Valgrind Cachegrind and a thread scaling study.

2 Target system and experimental setup

All experiments were run on:

- **CPU:** Intel Core i5-6300U (2 physical cores, 4 hardware threads)
- **Cache:** L2 512 KB, L3 3 MB (as reported by the platform)
- **OS:** Windows with WSL2 (Ubuntu)

The implementation and scripts used in this work are contained in `matmul_system.cpp` (custom kernels) and `blas_ref.cpp` (OpenBLAS reference). The benchmark results were exported to `all_results.csv` and `thread_scaling.csv`. Unless otherwise stated, all matrices are stored in column major order.

3 Correctness and Fairness of the BLAS Comparison

3.1 Data layout

The project uses a consistent **column major** layout for all implementations. In the reference code, OpenBLAS is called through `cblas_sgemm` with `CblasColMajor` and with $\alpha = 1$ and $\beta = 0$, which matches the semantics of the custom kernels.

Algorithm 1 Naive SGEMM (column major)

```
for j = 0 to N − 1 do
    for i = 0 to N − 1 do
        s ← 0
        for k = 0 to N − 1 do
            s ← s + A(i, k) · B(k, j)
        end for
        C(i, j) ← s
    end for
end for
```

3.2 Correctness Checks

The driver supports an optional correctness mode (`--check`). In this mode, the output matrix C is validated by recomputing a set of randomly sampled entries using a straightforward dot product and comparing them using a relative tolerance. This provides high confidence that the optimized kernels produce numerically consistent results, while keeping validation overhead small for large N.

3.3 Fair comparison

OpenBLAS is a production tuned library and typically uses vectorization and multithreading internally. For fairness, the custom kernels are measured in the same process and on the same inputs, and the OpenBLAS thread count can be controlled using `OPENBLAS_NUM_THREADS`. In WSL2, CPU frequency scaling and scheduler noise can introduce run to run variation, so the results should be interpreted as representative measurements rather than strict bounds.

4 Implemented kernels

Let $A, B, C \in \mathbb{R}^{N \times N}$ (single precision). We compute $C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$.

4.1 Baseline Naive

The baseline is a triple loop implementation using the access pattern that is natural for column major storage. Algorithm 1 summarizes the baseline 3 loop SGEMM used as the performance reference.

4.2 JPI Unrolling

The `jpi` kernel applies a register friendly loop ordering and partial unrolling to reduce loop overhead and improve instruction level parallelism.

Algorithm 2 Packed SGEMM with panel packing

for $j_c = 0$ to $N - 1$ step N_C **do**
  for $p_c = 0$ to $N - 1$ step K_C **do**
    $B_p \leftarrow \text{PackPanelB}(B[p_c : p_c + K_C,\ j_c : j_c + N_C])$
    for $i_c = 0$ to $N - 1$ step M_C **do**
      $A_p \leftarrow \text{PackBlockA}(A[i_c : i_c + M_C,\ p_c : p_c + K_C])$
      for $j_r = 0$ to $N_C - 1$ step N_R **do**
        for $i_r = 0$ to $M_C - 1$ step M_R **do**
          $\text{MicroKernel}(M_R, N_R, K_C, A_p, B_p, C)$
        end for
      end for
    end for
  end for
end for

4.3 Cache Blocked

The `blocked` kernel applies tiling so that sub blocks of A and B are reused while they remain in cache.

4.4 Packed Micro Kernel Variants

The best performing custom kernels are `packed_ic` and `packed_jc`. They follow a BLIS style structure: pack panels of B into contiguous buffers, pack blocks of A, and compute on small tiles using a fixed size micro kernel (in this project a 16×6 kernel). Packing improves spatial locality and allows the micro kernel to stream through packed buffers with predictable strides 2.

4.5 OpenBLAS Reference

The reference implementation in `blas_ref.cpp` calls `cblas_sgemm`. This is used as the performance target and for the performance gap plots.

5 Benchmark Methodology

For each N, the benchmark measures wall clock time for the matrix multiplication kernel and reports both time in milliseconds and throughput in GFLOP/s computed as $2N^3/t$. The data in this report comes directly from the provided CSV logs.

Controlling threading for a fair comparison: OpenMP kernels use `OMP_NUM_THREADS`. OpenBLAS uses `OPENBLAS_NUM_THREADS` (and may also honor `OMP_NUM_THREADS` depending on the build). For the main size sweep (`all_results.csv`), we run all variants in single thread mode by exporting `OMP_NUM_THREADS=1` and `OPENBLAS_NUM_THREADS=1`. For the

Algorithm 3 OpenMP parallel JPI (thread safe)

```
#pragma omp parallel for schedule(static)
for j = 0 to N − 1 do
    for i = 0 to N − 1 do
        C(i, j) ← 0
    end for
    for k = 0 to N − 1 do
        b ← B(k, j)
        for i = 0 to N − 1 do
            C(i, j) ← C(i, j) + A(i, k) · b
        end for
    end for
end for
```

thread scaling experiment (`thread_scaling.csv`), we set both variables to the same value (1, 2, and 4) to compare scaling behavior at $N = 2048$.

5.1 Parallelization

The code supports both single thread and multi thread execution using OpenMP (3 to 5). All parallel regions are designed so that each thread writes to a disjoint tile of C, while the reduction dimension (k or p_c) remains sequential inside each tile update.

Thread safe loop choices: For $C \leftarrow AB$ with accumulation over k, parallelizing over the reduction dimension is unsafe unless each thread accumulates into a private buffer and a reduction is performed. The implementations in `matmul_system.cpp` use the following thread safe choices:

- parallelize over columns j (or column tiles j_c),
- parallelize over row blocks i_c once the packed B panel for the current (j_c, p_c) is available,
- keep k (or p_c) sequential inside each tile update.

Parallel JPI (column parallel): The JPI ordering is parallelized over j. Each thread owns full columns of C.

Parallel packed by j_c (packed_jc): The packed_jc strategy parallelizes the outermost column tile loop. Each thread owns one or more j_c tiles and therefore writes to disjoint regions of C. Packing buffers are private per thread, which avoids synchronization, at the cost of extra total packing work when many threads are used.

Parallel packed by i_c inside each (j_c, p_c) (packed_ic): The packed_ic strategy shares the packed B panel across threads, then distributes the row blocks i_c across threads. This reduces redundant packing of B, but introduces a synchronization point for each (j_c, p_c).

Algorithm 4 OpenMP parallel packed_jc (thread private packing)

```
#pragma omp parallel for schedule(static)
for j_c = 0 to N − 1 step N_C do
    allocate thread private packA and packB
    for p_c = 0 to N − 1 step K_C do
        packB ← PackPanelB(B[p_c : p_c + K_C, j_c : j_c + N_C])
        for i_c = 0 to N − 1 step M_C do
            packA ← PackBlockA(A[i_c : i_c + M_C, p_c : p_c + K_C])
            call the M_R × N_R microkernel over sub tiles of C[i_c : i_c + M_C, j_c : j_c + N_C]
        end for
    end for
end for
```

Table 1: Measured throughput in GFLOP/s for each implementation and matrix size N. Missing entries indicate that a configuration was not benchmarked for that N.

N	naive	jpi	blocked	packed_ic	packed_jc	openblas
256	1.50	19.66	19.49	35.50	47.59	65.45
512	0.91	11.21	15.90	40.13	43.05	72.98
1024	0.33	9.99	13.47	47.68	48.77	78.20
2048	–	–	–	44.46	44.77	77.74
4096	–	–	–	40.31	42.32	76.76

6 Performance results

6.1 Gap to OpenBLAS

OpenBLAS remains the strongest reference. Over the tested sizes, the best custom kernel achieves between 55.1% and 72.7% of OpenBLAS throughput (average 61.4%) 2. This corresponds to a performance gap of roughly 1.38x to 1.81x. The remaining gap is expected because OpenBLAS typically uses architecture specific assembly kernels, deeper register blocking, careful prefetching, and tuned block sizes, while this project uses a single portable microkernel and conservative block sizes. Table 1 reports throughput (GFLOP/s) across matrix sizes for all variants along with the OpenBLAS.

6.2 Technical Interpretation

This section explains what each plot is showing and connects the trends back to the implementation choices in `matmul_system.cpp` and to the target CPU cache hierarchy (L2 512 KB, L3 3 MB).

Figure 1: Throughput versus N. The baseline `naive` kernel is orders of magnitude slower and its throughput drops as N grows. For example, it goes from about 1.50 GFLOP/s at $N = 256$ to about 0.33 GFLOP/s at $N = 1024$. This is the expected behavior for an unblocked dot product style loop: as the working set exceeds L2 and L3, most operand loads

Algorithm 5 OpenMP parallel packed_ic (shared B panel, row block parallel)

allocate shared `packB` for one (j_c, p_c) panel
#pragma omp parallel
allocate thread private `packA`
for $j_c = 0$ to $N - 1$ step N_C **do**
    for $p_c = 0$ to $N - 1$ step K_C **do**
        #pragma omp for schedule(static)
        for each subpanel of $B[p_c : p_c + K_C,\ j_c : j_c + N_C]$ **do**
            pack into shared `packB`
        end for
        #pragma omp barrier
        #pragma omp for schedule(static)
        for $i_c = 0$ to $N - 1$ step M_C **do**
            `packA` $\leftarrow$ PackBlockA($A[i_c : i_c + M_C,\ p_c : p_c + K_C]$)
            call the microkernel over $C[i_c : i_c + M_C,\ j_c : j_c + N_C]$
        end for
    end for
end for

Table 2: Best custom kernel versus OpenBLAS. Ratios are computed from the sweep in `all_results.csv`.

N	Best variant	Best (GFLOP/s)	OpenBLAS (GFLOP/s)	Ratio
256	packed_jc	47.59	65.45	72.7%
512	packed_jc	43.05	72.98	59.0%
1024	packed_jc	48.77	78.20	62.4%
2048	packed_jc	44.77	77.74	57.6%
4096	packed_jc	42.32	76.76	55.1%

stop hitting cache, so performance becomes dominated by memory traffic and cache miss latency rather than floating point throughput.

The mid level kernels `jpi` and `blocked` improve locality with loop reordering and tiling, so they jump to the 10 to 20 GFLOP/s range for smaller sizes. Their curves still slope downward with N because they continue to read A and B from their original strided layout, which limits effective bandwidth and makes prefetching less predictable.

The packed microkernel variants `packed_ic` and `packed_jc` deliver the highest custom throughput because they convert strided accesses into sequential accesses and concentrate work inside a vectorized 16×6 AVX2 plus FMA microkernel. Their best region is around $N = 1024$ where `packed_jc` reaches about 48.77 GFLOP/s. At very large sizes (for example $N = 4096$), throughput decreases (42.32 GFLOP/s) because packing traffic and L3 capacity pressure increase, and the computation becomes more bandwidth limited.

OpenBLAS stays above the custom kernels across all tested sizes (about 65.45 to 78.20 GFLOP/s in the first three sizes), which is consistent with highly tuned assembly microkernels, deeper blocking, and mature prefetch and scheduling heuristics.

Table 3: Measured runtime in milliseconds for each implementation and matrix size N.

N	naive	jpi	blocked	packed_ic	packed_jc	openblas
256	22.302	1.707	1.722	0.945	0.705	0.513
512	293.788	23.950	16.880	6.689	6.236	3.678
1024	6438.758	214.891	159.374	45.041	44.030	27.463
2048	–	–	–	386.370	383.703	220.999
4096	–	–	–	3409.761	3247.342	1790.514

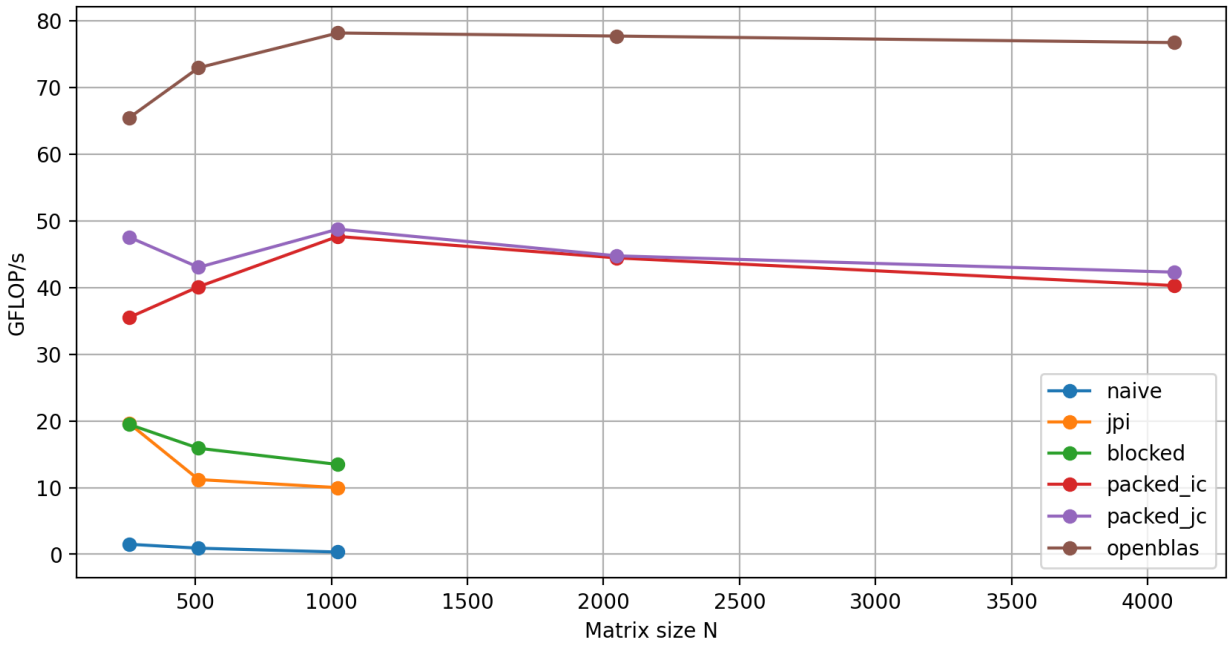

Figure 1: Throughput (GFLOP/s) versus matrix size N for custom kernels and OpenBLAS.

Figure 2: Runtime versus N on a log scale. The log scale makes the near cubic growth easy to see. Doubling N ideally multiplies the arithmetic by 8, so for a compute dominated GEMM you expect time to increase by about 8 times. OpenBLAS is close to this ideal trend for the larger sizes (for example from $N = 1024$ to $N = 2048$ the time increases from 27.46 ms to 221.00 ms, which is about 8.0 times). The packed kernels show a similar trend once N is large enough that packing overhead is amortized (3).

The naive curve is extremely steep because the same arithmetic growth is combined with rapidly worsening cache behavior. This is also why naive was only benchmarked up to $N = 1024$ in this run set, since beyond that it becomes impractically slow on the target laptop CPU.

Figure 3: Speedup relative to naive. Speedup increases with N because the naive kernel degrades faster than the optimized kernels as the working set exceeds cache. At $N = 256$, `packed_jc` is about 31.7 times faster than naive (in GFLOP/s terms). At $N = 1024$, `packed_jc` is about 147.8 times faster, and OpenBLAS is about 237.0 times faster. This widening gap is a hallmark of cache optimized matrix multiplication: packing and blocking keep reuse high even when N is large, while the naive kernel repeatedly reloads operands

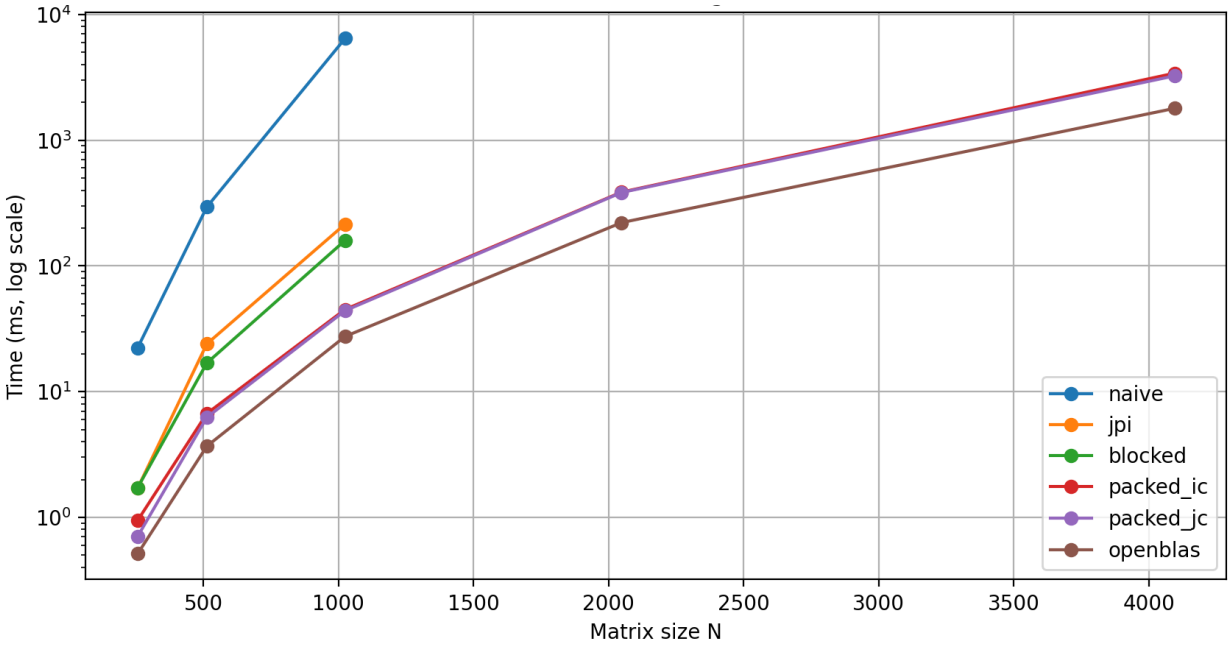

Figure 2: Runtime versus N on a log scale.

Table 4: Cachegrind summary for $N = 512$. Miss rates are computed as misses divided by references for reads and writes separately. Cachegrind uses a cache model, so these are good indicators of locality trends rather than exact hardware counter totals.

Variant	Ir	D refs	L1D read miss	LL read miss	L1D write miss	LL write miss
naive	289,877,453	274,235,260	49.36%	0.00%	17.94%	3.07%
packed_jc	61,584,329	65,821,407	2.63%	0.07%	1.26%	0.89%
openblas	69,546,131	24,968,198	6.95%	0.36%	2.14%	0.94%

from memory.

Figure 4: Gap to OpenBLAS. This plot shows how close the best custom kernels get to OpenBLAS. For `packed_jc`, the ratio is about 72.7% at $N = 256$, about 62.4% at $N = 1024$, and about 55.1% at $N = 4096$. The ratio is usually highest in the mid range where packing overhead is amortized but the problem still fits the cache blocking strategy well. The ratio drops at the largest sizes because OpenBLAS typically has more refined blocking, packing, and prefetch behavior for very large N, and because its microkernels are optimized at the assembly level.

7 Cache Locality Analysis

To better understand why packing helps, Cachegrind was run for $N = 512$ for `naive`, `packed_jc`, and `openblas`. The overall comparative analysis is give in the Table (4).

The cache behavior explains most of the speedups:

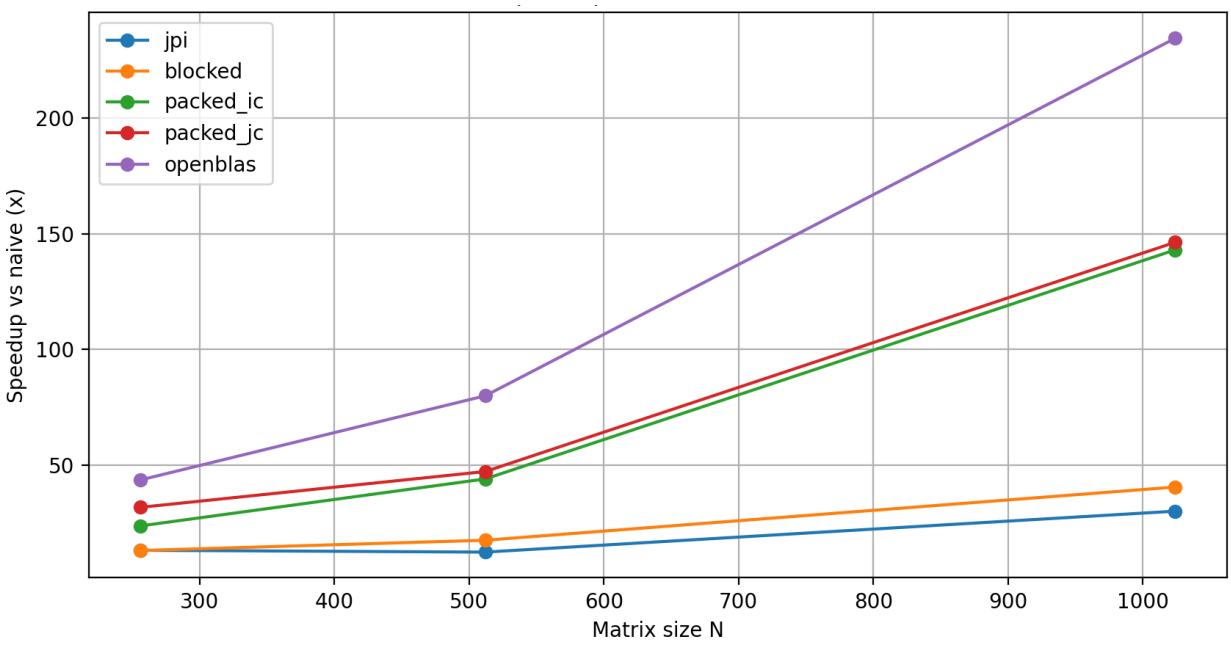

Figure 3: Speedup relative to the naive kernel for sizes where naive was benchmarked.

- The naive kernel has a very high L1D read miss rate (49.36%). This matches the access pattern where $B(k, j)$ is read with a large stride in memory.
- Packing reduces the L1D read miss rate to 2.63% (about 18.8x fewer read misses than naive) by converting microkernel inputs into contiguous streams. It also reduces the total number of data references by about 4.17x, which indicates that more values are kept in registers and reused.
- OpenBLAS shows a higher L1D read miss rate than packed_jc, but fewer total data references. This is consistent with stronger register blocking and reuse inside its assembly microkernels.

The packed kernel substantially reduces the L1 data miss rate compared with the naive baseline, which is consistent with the intended effect of packing panels into contiguous buffers. OpenBLAS also shows good locality, although the details depend on internal blocking choices.

8 Conclusion

On the Intel Core i5-6300U, the best custom kernel (`packed_jc`) reaches roughly half to two thirds of OpenBLAS throughput across the tested sizes, which is a strong result for a compact educational implementation. The main remaining performance gap likely comes from deeper micro kernel tuning, more careful choice of blocking parameters, and more aggressive vectorization.

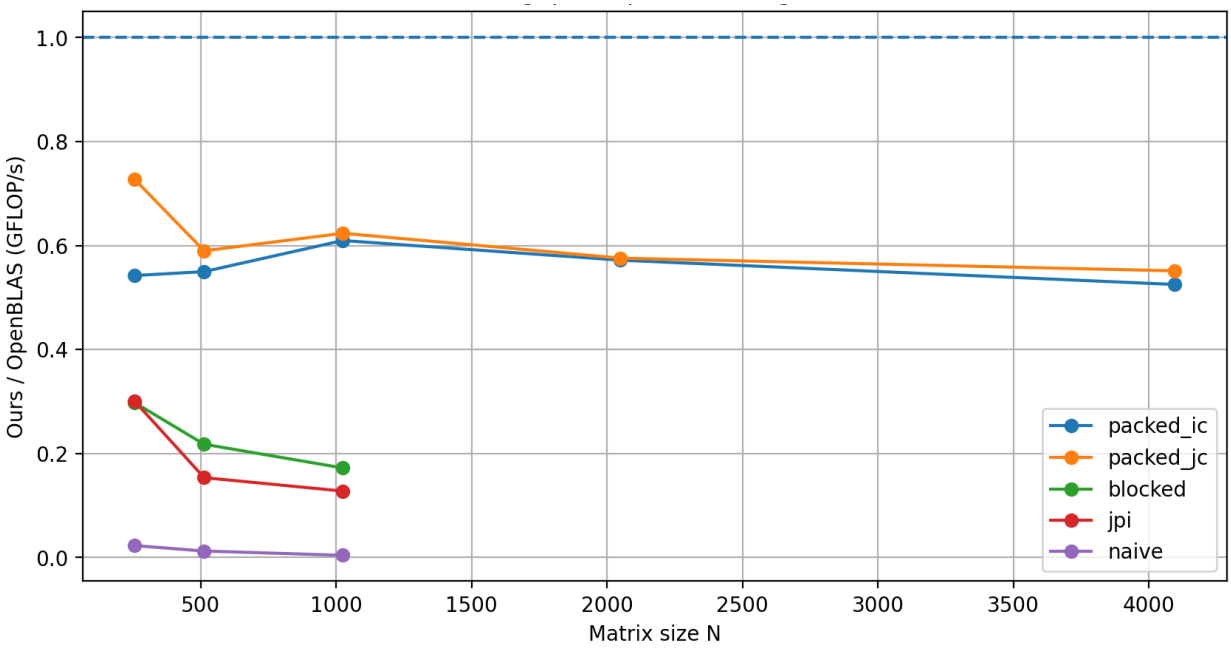

Figure 4: Ratio of custom kernel throughput to OpenBLAS throughput. Higher is better.

8.1 Future Work

- Pin the process to a fixed core set and disable frequency scaling during benchmarks to reduce noise.
- Run multiple repetitions per configuration and report median and percentiles.
- Tune M_C, N_C, K_C to fit the L2 and L3 cache sizes of the i5-6300U.
- Add explicit vector intrinsics for the 16×6 micro kernel and align packed buffers for AVX2.
- Add a separate outer parallel loop over j_c panels for the packed kernels so that multi-threading scales cleanly.