

Matrix-Matrix Multiplication Optimization: Iterative Analysis

Joy Mosisa

December 6, 2025

Collaborators: Parker Hix.

Instructions

For each iteration of the tuned matrix multiplication kernel, respond to the following three questions:

1. **What changed between this and the previous iteration?** Describe the code differences. Include snippets where needed.
2. **What was the change in performance?** Include a centered plot image.
3. **Why do you think that happened?** Provide a concise explanation or hypothesis.

Iteration 2: tuned_variant00_op.c

1. Code Differences

This is the baseline implementation - a straightforward triple-nested loop that computes $C = A * B + C$ (the dot product of rows of A with columns of B). It also utilizes MPI for distributed memory parallelism.

This is structured as:

- Three nested loops over matrix dimensions (m, n, k)
- Root rank distributes A and B to other ranks, each rank computes portion of C
- MPI Send/Recv for inter-rank communication
- Straightforward scalar operations with minimal optimization

The baseline serves as a reference point to measure improvements from later optimizations.

2. Performance Change

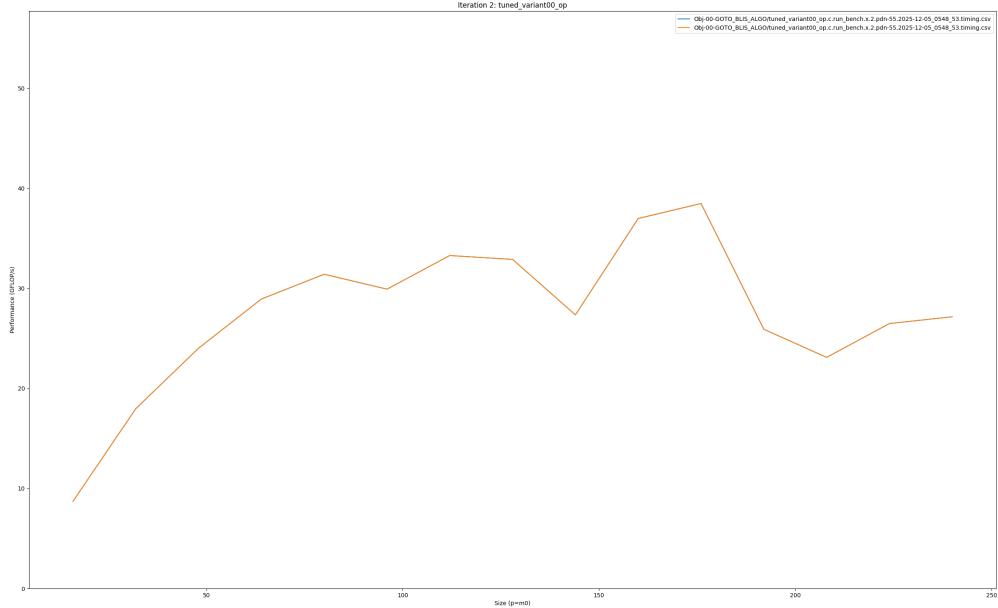


Figure 1: Performance comparison for Iteration 2

3. Explanation

This baseline performance represents the unoptimized, purely distributed-memory implementation.

Iteration 3: tuned_variant000_index_set_split.c

1. Code Differences

Introduces fringe case handling with three block size macros (`BLOCK_NC` = 192, `BLOCK_KC` = 128, `BLOCK_MC` = 128). The code splits index ranges into "steady state" regions that are multiples of block sizes, and "fringe" regions for remainders:

```
int n0_fringe_start = n0 - (n0%(BLOCK_NC));
int k0_fringe_start = k0 - (k0%(BLOCK_KC));
int m0_fringe_start = m0 - (m0%(BLOCK_MC));
```

2. Performance Change

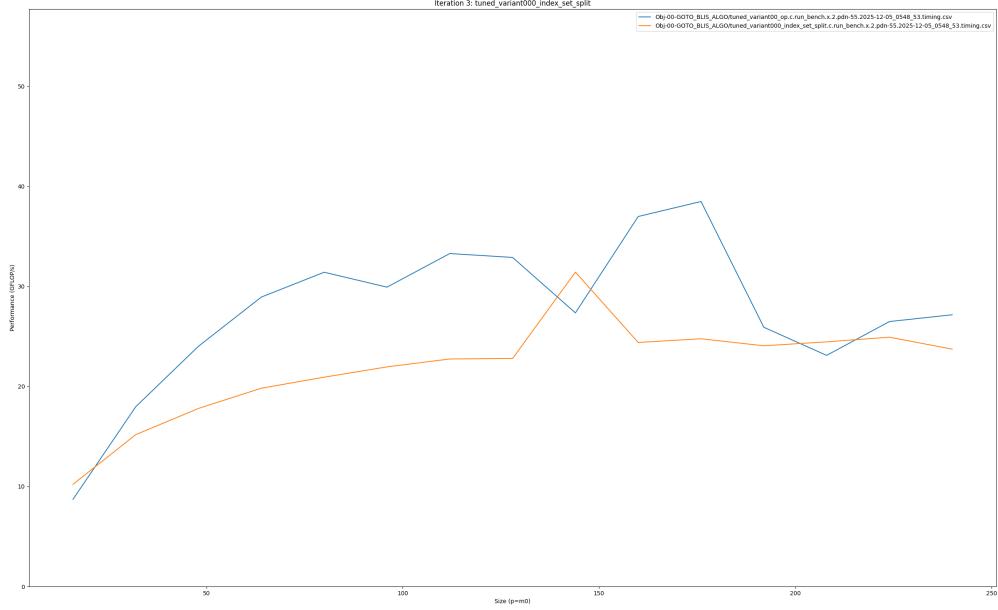


Figure 2: Performance comparison for Iteration 3

3. Explanation

The fringe case handling is done by splitting the index sets into main blocks and remainder blocks. The separate loops handle the steady-state (0 to `fringe_start`) and fringe (`fringe_start` to dimension size) computations, potentially reducing branching overhead in the main loop as it avoids conditional logic. Despite this, there seems to be an overall regression in comparison to iteration 2, likely due to the duplicate loop structure increasing instruction cache overhead and fetch latency, making it more difficult to unroll effectively.

Iteration 4: tuned_variant01_op.c

1. Code Differences

Introduces the first level of cache blocking by adding `BLOCK_NC` (n-dimension blocking, default 192). The outer `j`-loop now iterates in blocks of `BLOCK_NC`:

```
int n0_fringe_start = n0 - (n0%(BLOCK_NC));
// Steady State
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
for( int p0 = 0; p0 < k0; ++p0 )
```

```

for( int i0 = 0; i0 < m0; ++i0 )
for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
{
int j0 = j0_o + j0_i;
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
}

// Fringe for n0
for( int j0 = n0_fringe_start; j0 < n0; ++j0 )
for( int p0 = 0; p0 < k0; ++p0 )
for( int i0 = 0; i0 < m0; ++i0 )
{
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
}

```

2. Performance Change



Figure 3: Performance comparison for Iteration 4

3. Explanation

The use of BLOCK_NC enables blocking in the n-dimension (columns of C). This improves cache locality by working on vertical panels of C that better fit L3 cache, reducing cache misses when accessing C repeatedly. That being said, there is a sharp drop in throughput at the bigger sizes, likely due to it being the only dimension blocked.

Iteration 5: tuned_variant02_op.c

1. Code Differences

Adds BLOCK_KC (k-dimension blocking) to the existing BLOCK_NC blocking. Now both the k-loop and n-loop are blocked:

```
int n0_fringe_start = n0 - (n0%(BLOCK_NC));
int k0_fringe_start = k0 - (k0%(BLOCK_KC));

// Steady State
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{ // Steady State
    for( int p0_o = 0; p0_o < k0_fringe_start; p0_o += BLOCK_KC )
        for( int i0 = 0; i0 < m0; ++i0 )
            for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
                for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
                {
                    int j0 = j0_o + j0_i;
                    int p0 = p0_o + p0_i;
                    float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
                    float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
                    C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
                }
}

// Fringe for k0
for( int p0 = k0_fringe_start; p0 < k0; ++p0 )
    for( int i0 = 0; i0 < m0; ++i0 )
        for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
        {
            int j0 = j0_o + j0_i;
            float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
            float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
            C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
        }
}
```

2. Performance Change

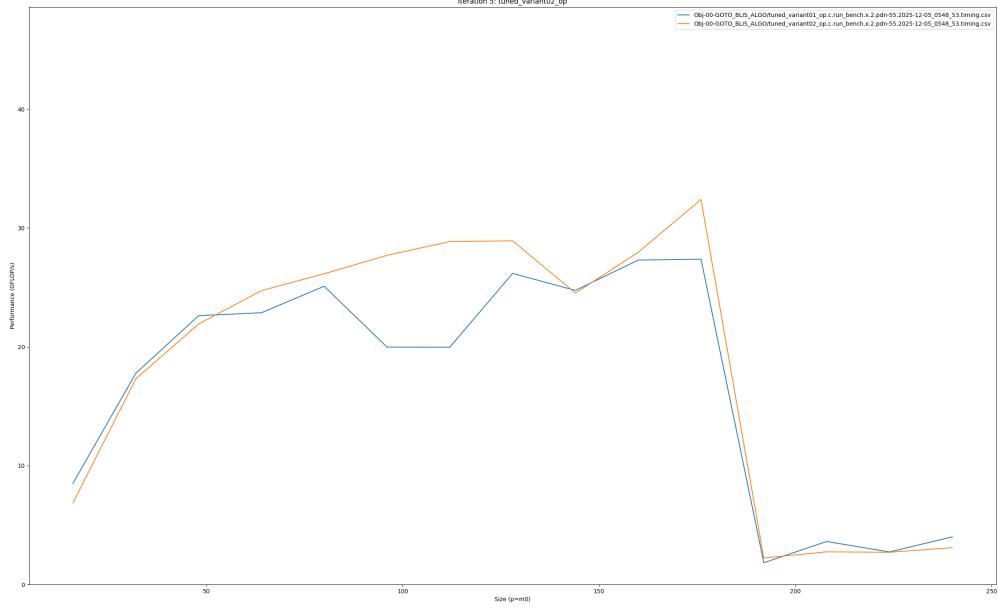


Figure 4: Performance comparison for Iteration 5

3. Explanation

The introduction of BLOCK_KC adds blocking in the k-dimension. This ensures panels of A ($m \times \text{BLOCK_KC}$) and B ($\text{BLOCK_KC} \times n$) fit in cache during the inner product accumulation, improving L2/L3 cache reuse. That being said, there is still 1 dimension left unblocked, hence we see a similar load-imbalance crash to that observed in iteration 3.

Iteration 6: tuned_variant03_op_block_mc.c

1. Code Differences

Adds BLOCK_MC (m-dimension blocking) to complete the 3-level cache hierarchy:

```
int n0_fringe_start = n0 - (n0%(BLOCK_NC));
int k0_fringe_start = k0 - (k0%(BLOCK_KC));
int m0_fringe_start = m0 - (m0%(BLOCK_MC));

// Steady State
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{
    for( int p0_o = 0; p0_o < k0_fringe_start; p0_o += BLOCK_KC )
```

```

{
for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
for( int i0_i = 0; i0_i < BLOCK_MC; ++i0_i )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
{
int j0 = j0_o + j0_i;
int i0 = i0_o + i0_i;
int p0 = p0_o + p0_i;
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
}

// Fringe for m0
for( int i0 = m0_fringe_start; i0 < m0; ++i0 )
for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
{
int j0 = j0_o + j0_i;
int p0 = p0_o + p0_i;
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
}
}
}
}

```

2. Performance Change

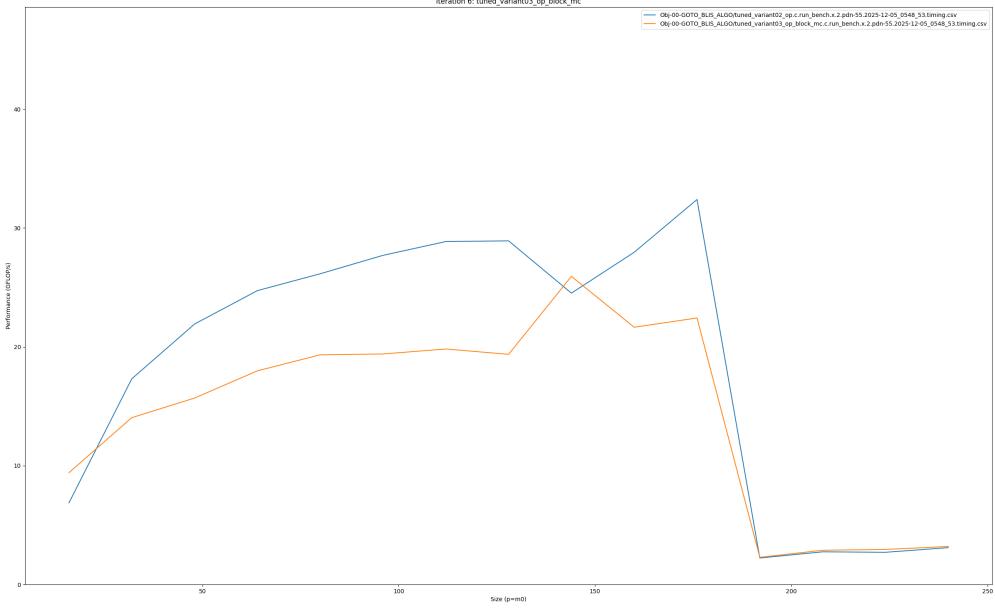


Figure 5: Performance comparison for Iteration 6

3. Explanation

The addition of BLOCK_MC adds blocking in the m-dimension (rows of C), creating a full 3-level cache hierarchy blocking structure (NC for L3, KC for L2, and MC for L1). This ensures the micro-panel of C (BLOCK_MC × BLOCK_NC) fits in L1 cache while blocks of A and B fit in L2/L3, reducing cache misses. Despite this completion, it seems there is an overall regression in performance in comparison to iteration 5, likely due to the extra MC loop increasing loop overhead.

Iteration 7: tuned_variant04_op_block_nr.c

1. Code Differences

Introduces micro-kernel blocking with BLOCK_NR for register-level optimization. The innermost j-loop is now blocked by BLOCK_NR:

```
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{
    for( int p0_o = 0; p0_o < k0_fringe_start; p0_o += BLOCK_KC )
    {
        for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
            for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
```

```

for( int i0_i = 0; i0_i < BLOCK_MC; ++i0_i )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i;
int p0 = p0_o + p0_i;
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
}
}
}

```

2. Performance Change

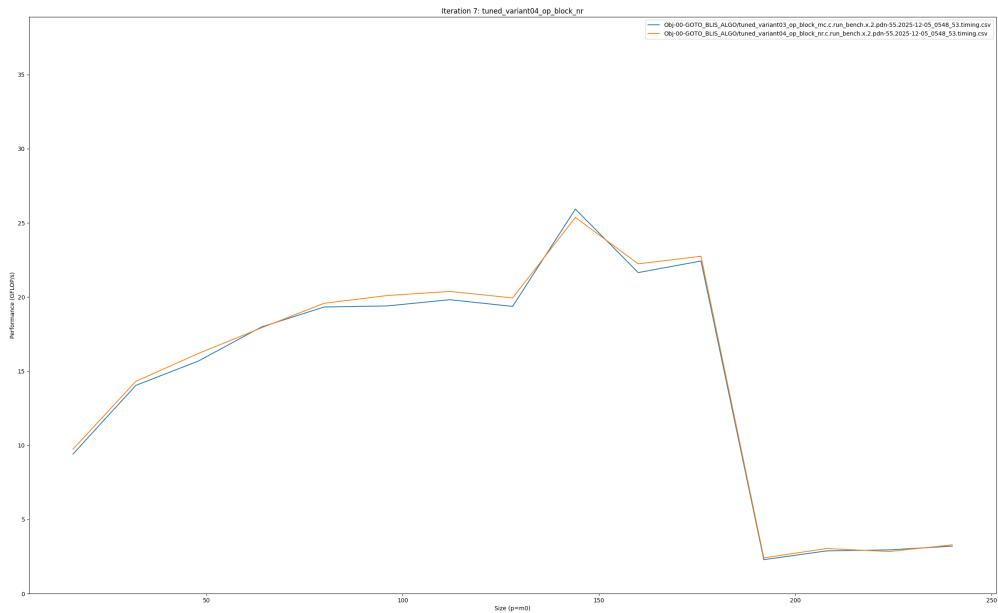


Figure 6: Performance comparison for Iteration 7

3. Explanation

BLOCK_NR's micro-kernel blocking for register-level optimization creates small register tiles ($MR \times NR$). These small blocks are kept in registers for the micro-kernel computation, enabling better register reuse. That being said, the improvement is only minimal given the lack of packing and a true micro-kernel. As such, loop overhead remains as B accesses are still strided, and register blocking alone cannot overcome cache limits.

Iteration 8: tuned_variant05_op_block_mr.c

1. Code Differences

Adds BLOCK_MR to complete the micro-kernel dimensions, creating a 16×16 register tile:

```
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{
for( int p0_o = 0; p0_o < k0_fringe_start; p0_o += BLOCK_KC )
{
for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
for( int i0_i = 0; i0_i < BLOCK_MC; i0_i += BLOCK_MR )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;
int p0 = p0_o + p0_i;
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_distributed[i0 * cs_C + j0 * rs_C] += A_ip*B_pj;
}
}
}
```

2. Performance Change

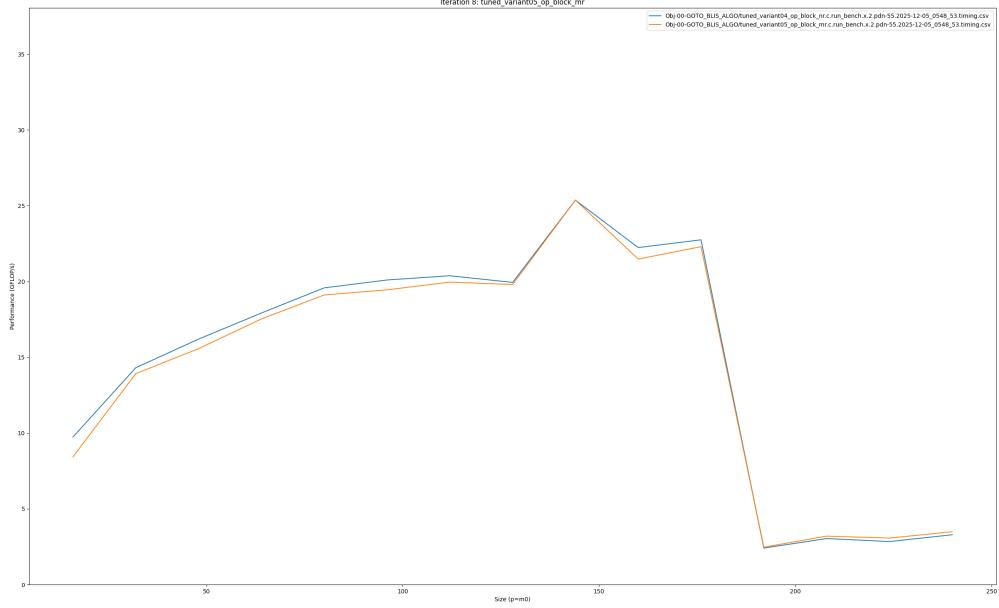


Figure 7: Performance comparison for Iteration 8

3. Explanation

BLOCK_MR completes the micro-kernel with m-dimension register blocking. The micro-kernel now operates on small MR×NR blocks that fit in registers and is now fully optimized for register tile computation. Despite this, performance overall is slightly regressed in comparison to iteration 7. This is likely, similarly, due to the fact that the code still lacks packing and a true fused micro-kernel.

Iteration 9: tuned_variant05_op_block_micro_kernel.c

1. Code Differences

Introduces a micro-kernel register tile (C_micro) to accumulate results in registers before writing back to memory:

```

for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
for( int i0_i = 0; i0_i < BLOCK_MC; i0_i += BLOCK_MR )
{
    // Small register tile for accumulation
    float C_micro[BLOCK_NR][BLOCK_MR];

    // Zero out C_micro
    for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )

```

```

for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
C_micro[j0_r][i0_r] = 0.0f;

// Rank-K update in register tile
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;
int p0 = p0_o + p0_i;
float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
C_micro[j0_r][i0_r] += A_ip*B_pj;
}

// Write back accumulated results to C
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;
C_distributed[i0 * cs_C + j0 * rs_C] += C_micro[j0_r][i0_r];
}
}

```

2. Performance Change

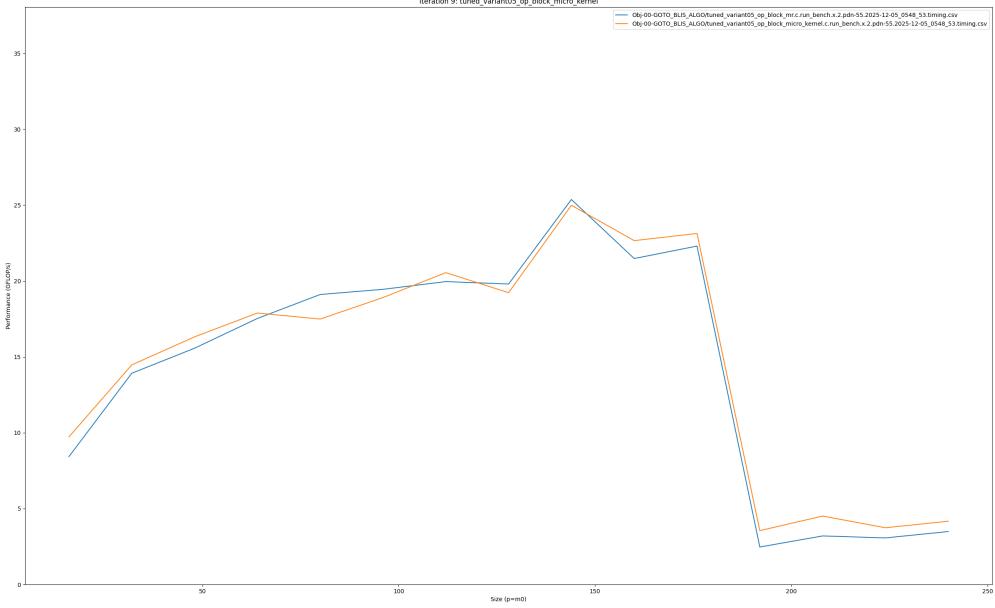


Figure 8: Performance comparison for Iteration 9

3. Explanation

Instead of writing to C_distributed every k iteration, results now accumulate in a small C_micro register tile and are written back only once per micro-tile. This reduces memory write traffic and bandwidth pressure, and improves register reuse across the rank-K update loop. Despite this, with a continued lack of packing, memory access patterns for A and B are still not optimal, so the gains are still minimal.

Iteration 10: tuned_variant06_op_pack_dlt_B.c

1. Code Differences

Introduces data packing for matrix B into a cache-friendly layout. Before the kernel loops, a BLOCK_KC x BLOCK_NC block of B is reorganized to match micro-kernel access order:

```
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{
    for( int p0_o = 0; p0_o < k0_fringe_start; p0_o += BLOCK_KC )
    {
        // Pack B block into contiguous cache-friendly layout
        // Reorder: B[p0][j0] -> B_dlt[j0_i_block][p0_i][j0_r]
```

```

float B_dlt[num_j0_i_blocks][BLOCK_KC][BLOCK_NR];
for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
{
int j0 = j0_o + j0_i + j0_r;
int p0 = p0_o + p0_i;
int j0_i_bid = j0_i/(BLOCK_NR);
B_dlt[j0_i_bid][p0_i][j0_r] =
B_distributed[p0 * cs_B + j0 * rs_B];
}

// Continues using B_dlt in kernel loops
// ... kernel computation using B_dlt ...
}
}

```

2. Performance Change



Figure 9: Performance comparison for Iteration 10

3. Explanation

Despite reorganizing B into a cache-friendly layout that should improve spatial locality and eliminate strided accesses, it seems performance overall regresses in comparison to iteration 9. This likely

occurs because the packing overhead is not being offset effectively at these block sizes due to the placement of the packing inside the KC loop and resulting redundancy.

Iteration 11: tuned_variant07_op_pack_dlt_A.c

1. Code Differences

Adds data packing for matrix A, complementing the B packing from iteration 10. A BLOCK_MC x BLOCK_KC block of A is reorganized inside the i0_o loop:

```
for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
{
    // Pack A block into cache-friendly layout
    // Reorder: A[i0][p0] -> A_dlt[i0_i_block][p0_i][i0_r]
    float A_dlt[num_i0_i_blocks][BLOCK_KC][BLOCK_MR];
    for( int i0_i = 0; i0_i < BLOCK_MC; i0_i += BLOCK_MR )
        for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
            for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
            {
                int i0 = i0_o + i0_i + i0_r;
                int p0 = p0_o + p0_i;
                int i0_i_bid = i0_i/(BLOCK_MR);
                A_dlt[i0_i_bid][p0_i][i0_r] =
                    A_distributed[i0 * cs_A + p0 * rs_A];
            }

    // Continues using A_dlt in kernel loops
    // ... kernel computation using A_dlt and B_dlt ...
}
```

2. Performance Change



Figure 10: Performance comparison for Iteration 11

3. Explanation

Despite packing both A and B into contiguous layouts, which should enable optimal sequential memory access, it seems performance overall regresses further in comparison to iteration 10. The regression likely persists due to the combined packing overhead now affecting both matrices without sufficient performance gains to offset the cost.

Iteration 12: tuned_variant08_op_zero_pack_remove_k_fringe.c

1. Code Differences

Eliminates k-dimension fringe handling by zero-padding packed buffers. The p0_o loop now iterates over full k0 dimension, and packing conditionally adds zeros for out-of-bounds elements:

```
for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
{
    int block_kc_remainder = min(BLOCK_KC, k0-p0_o);

    // Pack B with zero-padding for k-fringe
    float B_dlt[num_j0_i_blocks][BLOCK_KC][BLOCK_NR];
    for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
```

```

for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
{
int j0 = j0_o + j0_i + j0_r;
int p0 = p0_o + p0_i;
int j0_i_bid = j0_i/(BLOCK_NR);

// Zero-pad if beyond k dimension
if (p0 < k0 )
B_dlt[j0_i_bid][p0_i][j0_r] =
B_distributed[p0 * cs_B + j0 * rs_B];
else
B_dlt[j0_i_bid][p0_i][j0_r] = 0.0f;
}

// Continues similar zero-padding for A_dlt
// ... micro-kernel uses full BLOCK_KC without fringe checks
}

```

2. Performance Change

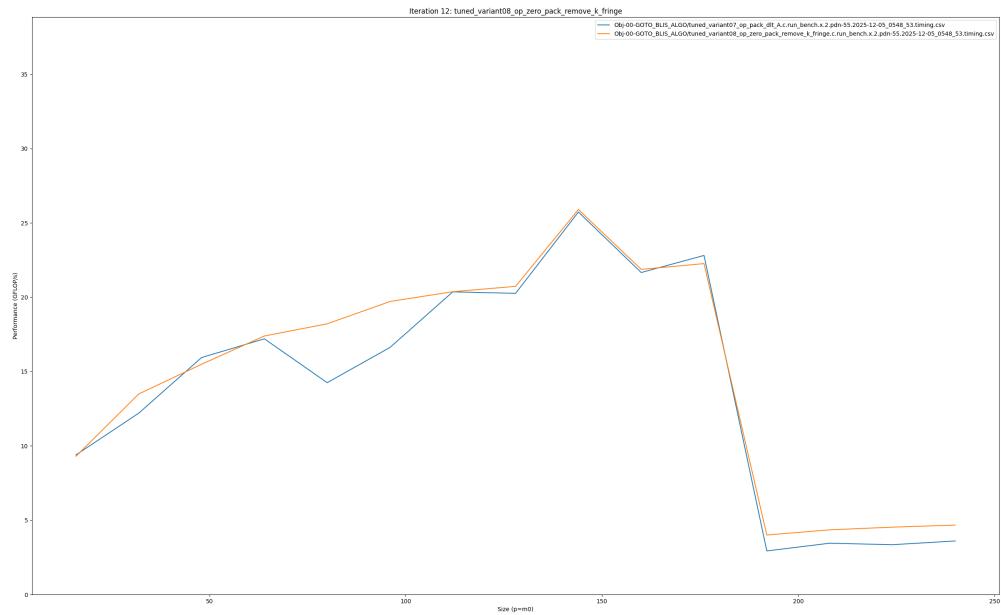


Figure 11: Performance comparison for Iteration 12

3. Explanation

Eliminating the k-dimension conditional branches reduces branching overhead in the micro-kernel, allowing the code to achieve more consistent performance across different matrix sizes. That being said, the improvement is small, likely due to the zero-padding during packing, which adds conditional checks that move some overhead from the micro-kernel to the packing phase.

Iteration 13: tuned_variant09_op_minimize_m_fringe.c

1. Code Differences

Minimizes m-dimension fringe by computing fringe at BLOCK_MR instead of BLOCK_MC, and uses zero-padding in A packing:

```
// Changed from m0 - (m0%(BLOCK_MC))
int m0_fringe_start = m0 - (m0%(BLOCK_MR));

for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
{
    int block_mc_remainder = min(BLOCK_MC, m0_fringe_start - i0_o);

    // Pack A with zero-padding for both k and m fringes
    float A_dlt[num_i0_blocks][BLOCK_KC][BLOCK_MR];
    for( int i0_i = 0; i0_i < BLOCK_MC; i0_i += BLOCK_MR )
        for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
            for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
            {
                int i0 = i0_o + i0_i + i0_r;
                int p0 = p0_o + p0_i;
                int i0_i_bid = i0_i/(BLOCK_MR);

                // Zero-pad if beyond k or m dimensions
                if (p0 < k0 & i0 < m0)
                    A_dlt[i0_i_bid][p0_i][i0_r] =
                        A_distributed[i0 * cs_A + p0 * rs_A];
                else
                    A_dlt[i0_i_bid][p0_i][i0_r] = 0.0f;
            }

    // Kernel loop uses block_mc_remainder
    for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
        for( int i0_i = 0; i0_i < block_mc_remainder; i0_i += BLOCK_MR )
            // ... micro-kernel computation
    }
}
```

2. Performance Change

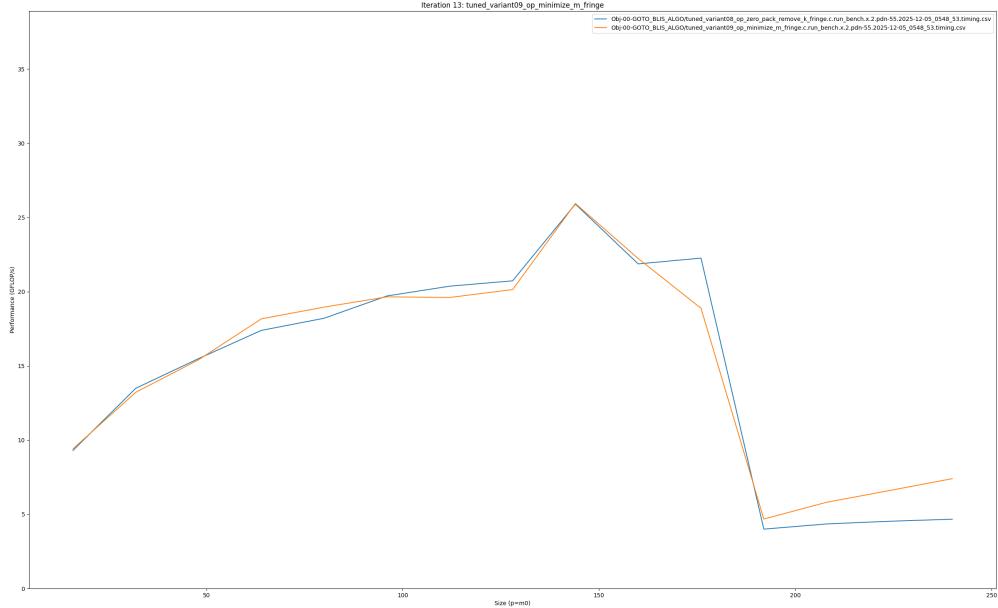


Figure 12: Performance comparison for Iteration 13

3. Explanation

Despite reducing m-fringe from BLOCK_MC to BLOCK_MR resolution, which should minimize edge case overhead, it seems performance overall regresses in comparison to iteration 12, likely due to the added " $i0 < m0$ " conditional in A packing, which introduces branching overhead that outweighs the gains from the fringe reduction.

Iteration 14: tuned_variant10_op_minimize_n_fringe.c

1. Code Differences

Minimizes n-dimension fringe by computing fringe at BLOCK_NR resolution instead of BLOCK_NC, and adds zero-padding for n-dimension boundary in B packing:

```
// Changed from n0 - (n0%(BLOCK_NC))
int n0_fringe_start = n0 - (n0%(BLOCK_NR));

for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{
    int block_nc_remainder = min(BLOCK_NC, n0_fringe_start-j0_o);
```

```

// Pack B with zero-padding for k and n fringes
float B_dlt[num_j0_i_blocks][BLOCK_KC][BLOCK_NR];
for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
{
int j0 = j0_o + j0_i + j0_r;
int p0 = p0_o + p0_i;
int j0_i_bid = j0_i/(BLOCK_NR);

// Zero-pad if beyond k or n dimensions
if( p0 < k0 & j0 < n0 )
B_dlt[j0_i_bid][p0_i][j0_r] =
B_distributed[p0 * cs_B + j0 * rs_B];
else
B_dlt[j0_i_bid][p0_i][j0_r] = 0.0f;
}

// Kernel loop uses block_nc_remainder
for( int j0_i = 0; j0_i < block_nc_remainder; j0_i += BLOCK_NR )
for( int i0_i = 0; i0_i < block_mc_remainder; i0_i += BLOCK_MR )
// ... micro-kernel computation
}

```

2. Performance Change

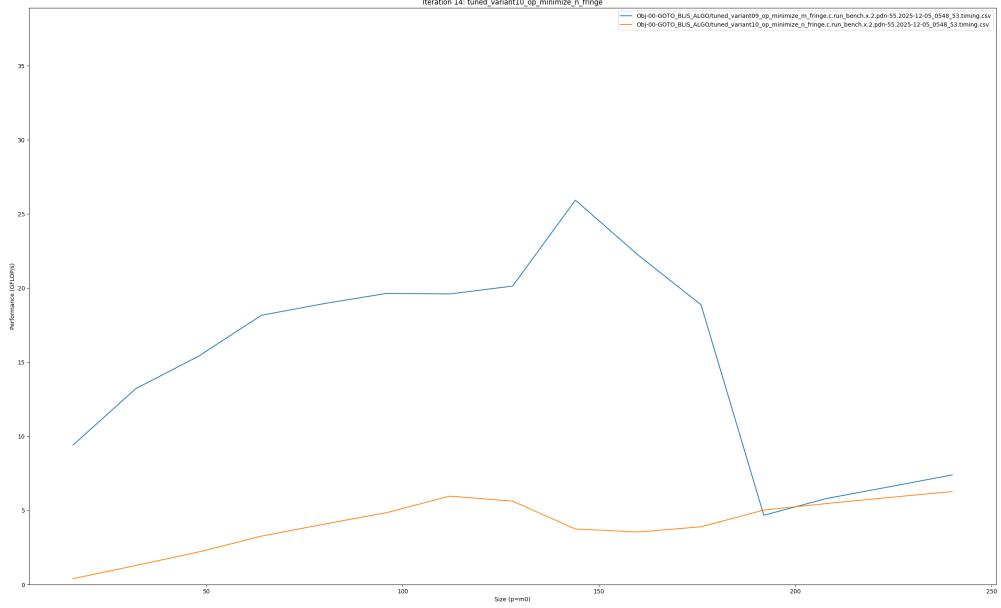


Figure 13: Performance comparison for Iteration 14

3. Explanation

In changing n0_fringe_start from BLOCK_NC to BLOCK_NR, the outer j0_o loop is caused to iterate BLOCK_NC times per outer loop, while the kernel loop j0_i now steps by BLOCK_NR and is bounded by block_nc_remainder. As such, the kernel must process incomplete BLOCK_NC tiles with variable iteration counts. In other words, this minimization reduces the n-fringe to only what spills beyond BLOCK_NR boundaries rather than BLOCK_NC, fundamentally breaking the micro-kernel scheduling and cache blocking strategy, thereby decreasing performance as reflected by the huge regression in performance in comparison to iteration 13.

Iteration 15: tuned_variant11_op_minimize_kc_zerowork.c

1. Code Differences

Eliminates zero-work in the k-dimension by using block_kc_remainder as the micro-kernel loop bound instead of the constant BLOCK_KC:

```
// Outer loop iteration
for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
{
    int block_kc_remainder = min(BLOCK_KC, k0-p0_o);
```

```

// Pack B and A with full BLOCK_KC allocation but only use valid data
float B_dlt[num_j0_i_blocks][BLOCK_KC][BLOCK_NR];
// ... packing loops iterate over BLOCK_KC ...

// Kernel loops
for( int j0_i = 0; j0_i < block_nc_remainder; j0_i += BLOCK_NR )
for( int i0_i = 0; i0_i < block_mc_remainder; i0_i += BLOCK_MR )
{
    // Micro-kernel now uses block_kc_remainder instead of BLOCK_KC
    for( int p0_i = 0; p0_i < block_kc_remainder; ++p0_i )
        for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
            for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
            {
                int j0 = j0_o + j0_i + j0_r;
                int i0 = i0_o + i0_i + i0_r;
                int p0 = p0_o + p0_i;

                int j0_i_bid = j0_i/(BLOCK_NR);
                int i0_i_bid = i0_i/(BLOCK_MR);

                float A_ip = A_dlt[i0_i_bid][p0_i][i0_r];
                float B_pj = B_dlt[j0_i_bid][p0_i][j0_r];

                C_micro[j0_r][i0_r] += A_ip*B_pj;
            }
        }
    }
}

```

2. Performance Change

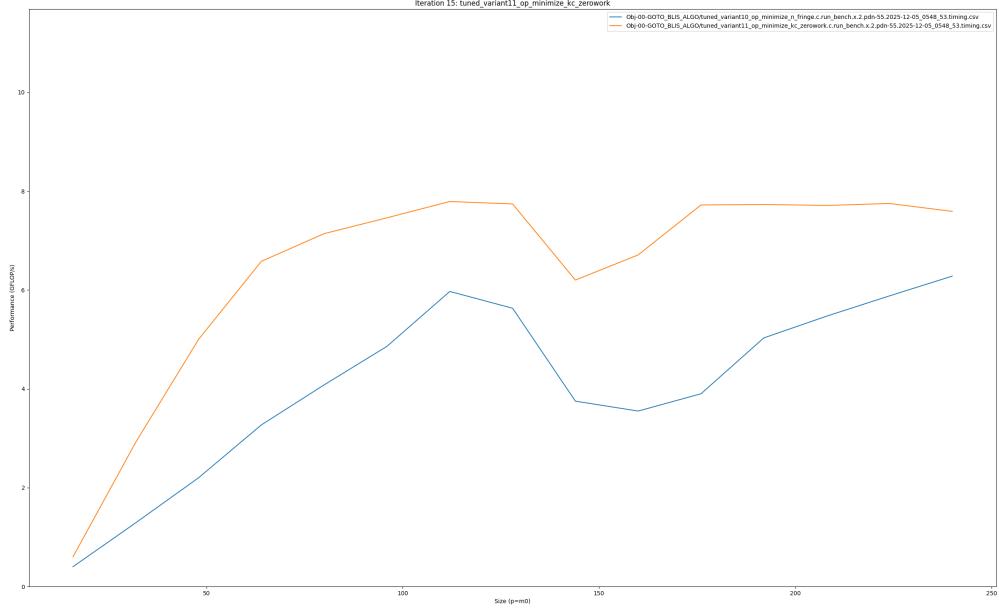


Figure 14: Performance comparison for Iteration 15

3. Explanation

Using block_kc_remainder in the micro-kernel innermost loop helps avoid redundant computation over zero-padded k-fringe elements, improving the efficiency of the loop. That being said, the runtime-variable k-loop bound still prevents compiler loop unrolling; however, the reduction in actual work creates a significant performance gain.

Iteration 16: tuned_variant12_op_kunroll_with_overrun_for_ilp.c

1. Code Differences

Introduces k-dimension loop unrolling with pragma directive to improve instruction-level parallelism. The k-loop is unrolled by BLOCK_KU iterations:

```
// Micro-kernel k-loop unrolling
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++ p0_u )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{ int j0 = j0_o + j0_i + j0_r;
```

```

int i0 = i0_o + i0_i + i0_r;
int p0 = p0_o + p0_i + p0_u;

int j0_i_bid = j0_i/(BLOCK_NR);
int i0_i_bid = i0_i/(BLOCK_MR);

float A_ip = A_dlt[i0_i_bid][p0_i+p0_u][i0_r];
float B_pj = B_dlt[j0_i_bid][p0_i+p0_u][j0_r];

C_micro[j0_r][i0_r] += A_ip*B_pj;
}

```

2. Performance Change

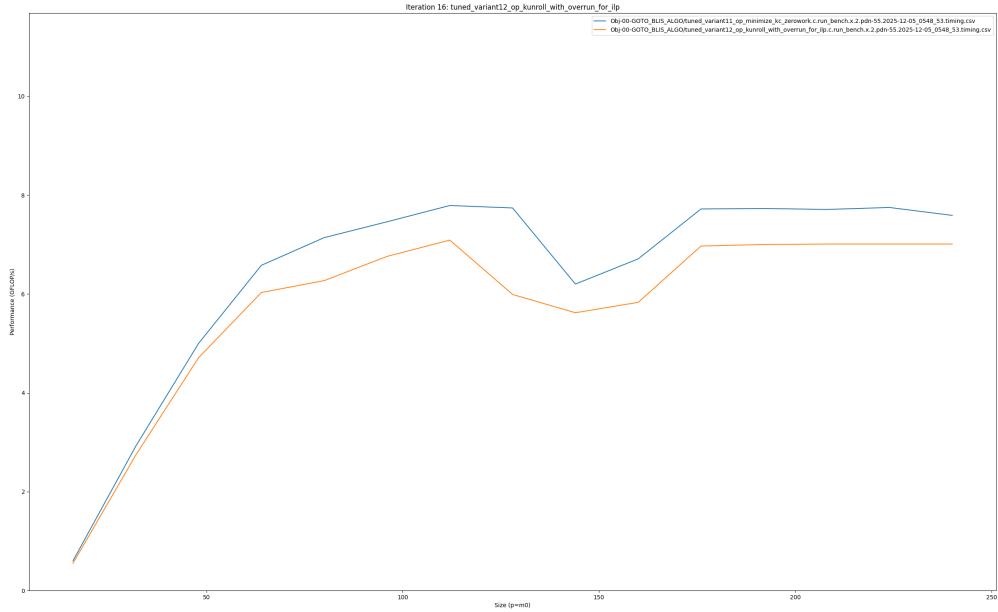


Figure 15: Performance comparison for Iteration 16

3. Explanation

This k-dimension unrolling exposes BLOCK_KU independent FMA operations per outer iteration. Despite the theoretical benefit of exposing more instruction-level parallelism this way, the performance seems to regress in comparison to iteration 15, indicating the unrolling is counterproductive at the current baseline state. This could be due to the pragma unroll directive creating BLOCK_KU copies of the inner p0_u loop body, increasing code size and instruction cache pressure, or the performance baseline potentially being CPU-limited by the broken fringe handling noted in iteration 14.

Iteration 17: tuned_variant13_op_mr_ilp.c

1. Code Differences

Unrolls the m-dimension (MR) loop in the micro-kernel by adding '#pragma unroll BLOCK_MR' directive:

```
// Initialize C_micro with MR-unrolled loop
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
C_micro[j0_r][i0_r] = 0.0f;
}

// Main computation with k and MR unrolling
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++p0_u )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;
int p0 = p0_o + p0_i + p0_u;

int j0_i_bid = j0_i/(BLOCK_NR);
int i0_i_bid = i0_i/(BLOCK_MR);

float A_ip = A_dlt[i0_i_bid][p0_i+p0_u][i0_r];
float B_pj = B_dlt[j0_i_bid][p0_i+p0_u][j0_r];

C_micro[j0_r][i0_r] += A_ip*B_pj;
}

// Write-back with MR unrolling
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;
C_distributed[i0 * cs_C + j0 * rs_C] += C_micro[j0_r][i0_r];
}
```

2. Performance Change

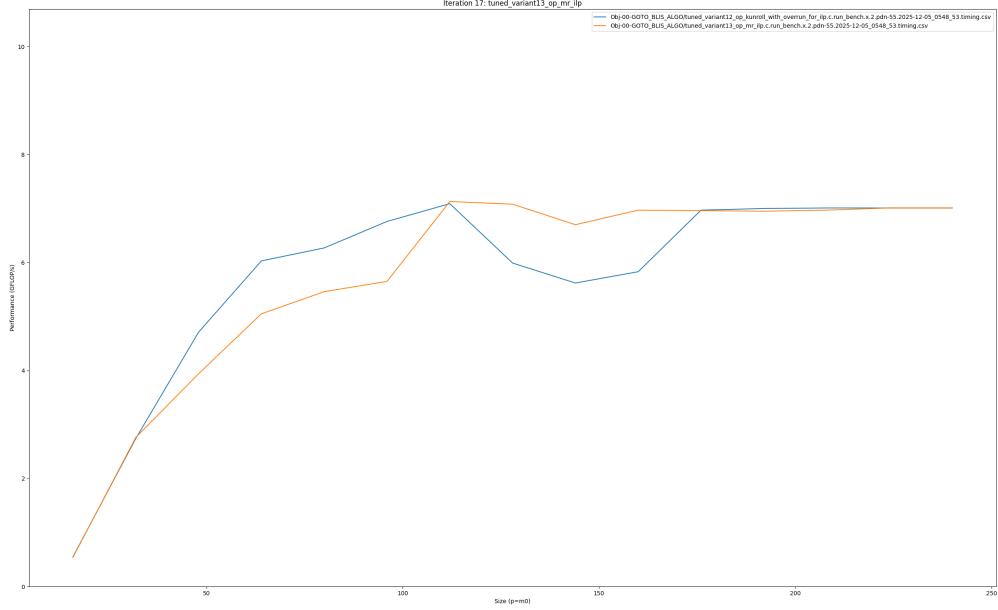


Figure 16: Performance comparison for Iteration 17

3. Explanation

The MR-dimension loop unrolling with ‘#pragma unroll BLOCK_MR’ exposes BLOCK_MR independent FMA operations per i0_r iteration, allowing the compiler to better schedule instructions and increase instruction-level parallelism. That being said, this might be increasing register pressure for smaller size matrices, causing degradation relative to iteration 16’s conservative k-only unrolling before surpassing and then converging with iteration 16 on bigger matrices.

Iteration 18: tuned_variant14_op_nr_ilp.c

1. Code Differences

Unrolls the NR dimension in addition to the MR unrolling from iteration 17. With both ‘#pragma unroll BLOCK_MR’ and ‘#pragma unroll BLOCK_NR’ applied, the entire $MR \times NR$ (16×6) micro-kernel computation is fully unrolled, exposing maximum instruction-level parallelism.

```
// Initialize C_micro[][] = 0
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
```

```

{
C_micro[j0_r][i0_r] = 0.0f;
}

// Rank-K update with BLOCK_KU and MR/NR unrolling
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++ p0_u )
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0_i_bid = j0_i/(BLOCK_NR);
int i0_i_bid = i0_i/(BLOCK_MR);

float A_ip = A_dlt[i0_i_bid][p0_i+p0_u][i0_r];
float B_pj = B_dlt[j0_i_bid][p0_i+p0_u][j0_r];

C_micro[j0_r][i0_r] += A_ip*B_pj;
}

// Write-back C_micro[][] to C_distributed[][]
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;

C_distributed[i0 * cs_C + j0 * rs_C] += C_micro[j0_r][i0_r];
}

```

2. Performance Change

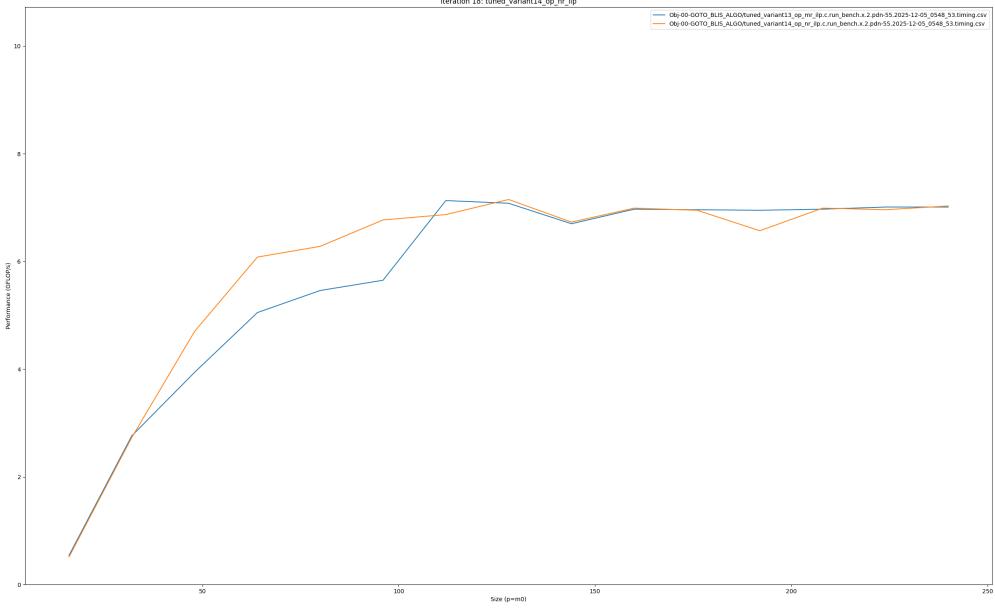


Figure 17: Performance comparison for Iteration 18

3. Explanation

The addition of NR-dimension unrolling with ‘#pragma unroll BLOCK_NR’ creates a fully unrolled MR×NR micro-kernel with no remaining loops, exposing BLOCK_MR × BLOCK_NR (16 × 6 or 96) independent FMA operations per rank-K iteration. This likely reduced any remaining branching overhead, overall improving performance.

Iteration 19: tuned_variant15_op_presimd_data_dist_elem.c

1. Code Differences

Instead of a flat 2D array ‘C_micro[BLOCK_NR][BLOCK_MR],’ this variant introduces a 3D structure ‘C_micro[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN]’ that groups BLOCK_MR elements into vector-sized chunks:

```
// C_micro with vectorized grouping (3D layout for SIMD)
float C_micro[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];

// Initialize with vector-aware indexing
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
```

```

#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
C_micro[j0_r][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN] = 0.0f;
}

// Rank-K update with vector-indexed accumulation
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++ p0_u )
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0_i_bid = j0_i/(BLOCK_NR);
int i0_i_bid = i0_i/(BLOCK_MR);

// A now vectorized along i0_r dimension
float A_ip = A_dlt[i0_i_bid][p0_i+p0_u][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN];
float B_pj = B_dlt[j0_i_bid][p0_i+p0_u][j0_r];

C_micro[j0_r][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN] += A_ip*B_pj;
}

// Write-back with vector-aware indexing
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll BLOCK_MR
for( int i0_r = 0; i0_r < BLOCK_MR; ++i0_r )
{
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;

C_distributed[i0 * cs_C + j0 * rs_C] += C_micro[j0_r][i0_r/PAR_VECT_LEN]
    [i0_r%PAR_VECT_LEN]
;
}

```

2. Performance Change

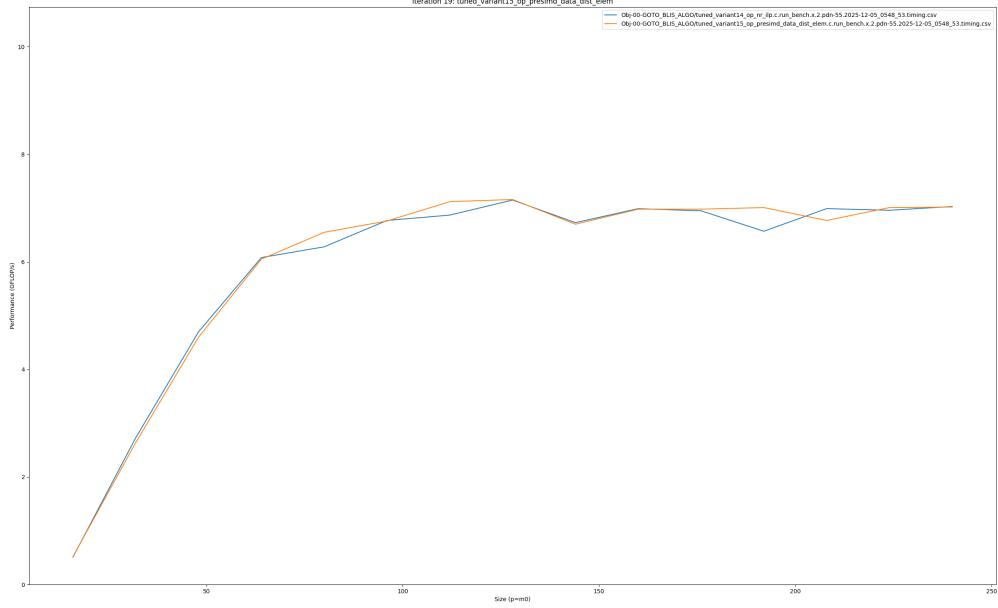


Figure 18: Performance comparison for Iteration 19

3. Explanation

The reorganization of C_micro from flat 2D to 3D is a pure data layout restructuring with no new computation changes or SIMD implementation. As such, performance remains essentially unchanged.

Iteration 20: tuned_variant16_op_presimd_data_dist_elem_cleaner.c

1. Code Differences

Cleans up the pre-SIMD data distribution by replacing implicit modulo/division indexing ('i0_r / PAR_VECT_LEN', 'i0_r % PAR_VECT_LEN') with explicit vector block index (i0_r_bid) and vector element index (i0_r_vid):

```
// Explicit vectorized indexing (cleaner version)
float C_micro[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];

// Initialize with explicit bid/vid loops
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
```

```

for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
C_micro[j0_r][i0_r_bid][i0_r_vid] = 0.0f;
}

// Rank-K update with explicit vector dimension unrolling
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++ p0_u )
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
int i0_r = i0_r_bid*(PAR_VECT_LEN) + i0_r_vid;
int j0_i_bid = j0_i/(BLOCK_NR);
int i0_i_bid = i0_i/(BLOCK_MR);

float A_ip = A_dlt[i0_i_bid][p0_i+p0_u][i0_r_bid][i0_r_vid];
float B_pj = B_dlt[j0_i_bid][p0_i+p0_u][j0_r];

C_micro[j0_r][i0_r_bid][i0_r_vid] += A_ip*B_pj;
}

// Write-back with explicit vector dimension unrolling
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
int i0_r = i0_r_bid*(PAR_VECT_LEN) + i0_r_vid;
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;

C_distributed[i0 * cs_C + j0 * rs_C] += C_micro[j0_r][i0_r_bid][i0_r_vid];
}

```

2. Performance Change

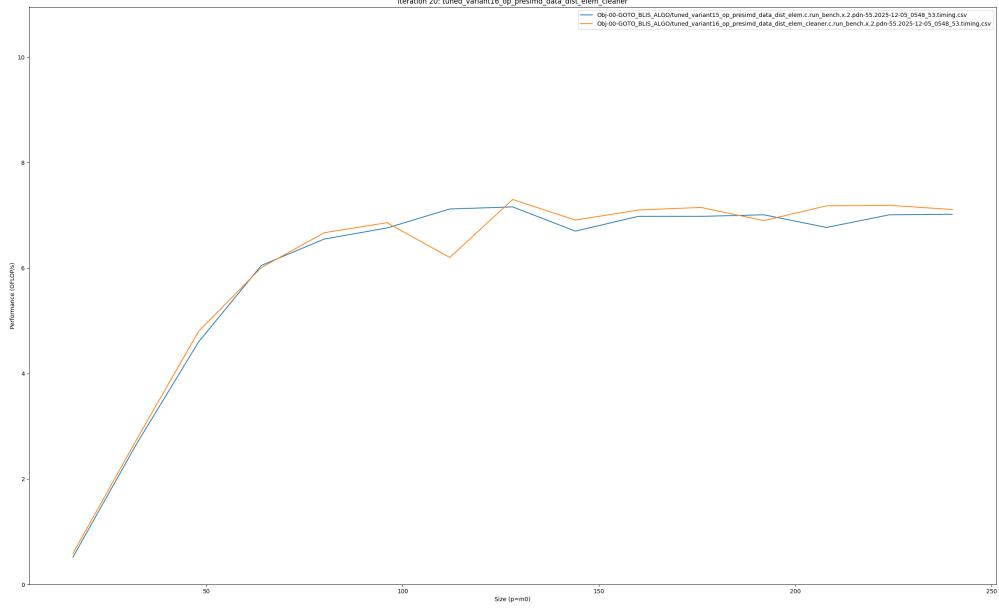


Figure 19: Performance comparison for Iteration 20

3. Explanation

The switch from implicit indexing to explicit indices eliminates expensive integer operations and enables the compiler to unroll the bid loop explicitly with ‘#pragma unroll ((BLOCK_MR) / (PAR_VECT_LEN))’, enabling better compiler optimization.

Iteration 21: tuned_variant17_op_pseudo_simd.c

1. Code Differences

Introduces ‘#pragma omp simd’ directives on the innermost ‘i0_r_vid’ loop to guide the compiler to vectorize floating-point operations:

```
// Initialize with OpenMP SIMD pragma
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
#pragma omp simd
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
```

```

C_micro[j0_r][i0_r_bid][i0_r_vid] = 0.0f;
}

// Rank-K update with vectorized FMA
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++ p0_u )
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
#pragma omp simd
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
float A_ip = A_dlt[i0_i_bid][p0_i+p0_u][i0_r_bid][i0_r_vid];
float B_pj = B_dlt[j0_i_bid][p0_i+p0_u][j0_r];
C_micro[j0_r][i0_r_bid][i0_r_vid] += A_ip*B_pj;
}

// Write-back with SIMD
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
#pragma omp simd
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
int i0_r = i0_r_bid*(PAR_VECT_LEN) + i0_r_vid;
C_distributed[i0 * cs_C + j0 * rs_C] += C_micro[j0_r][i0_r_bid][i0_r_vid];
}

```

2. Performance Change

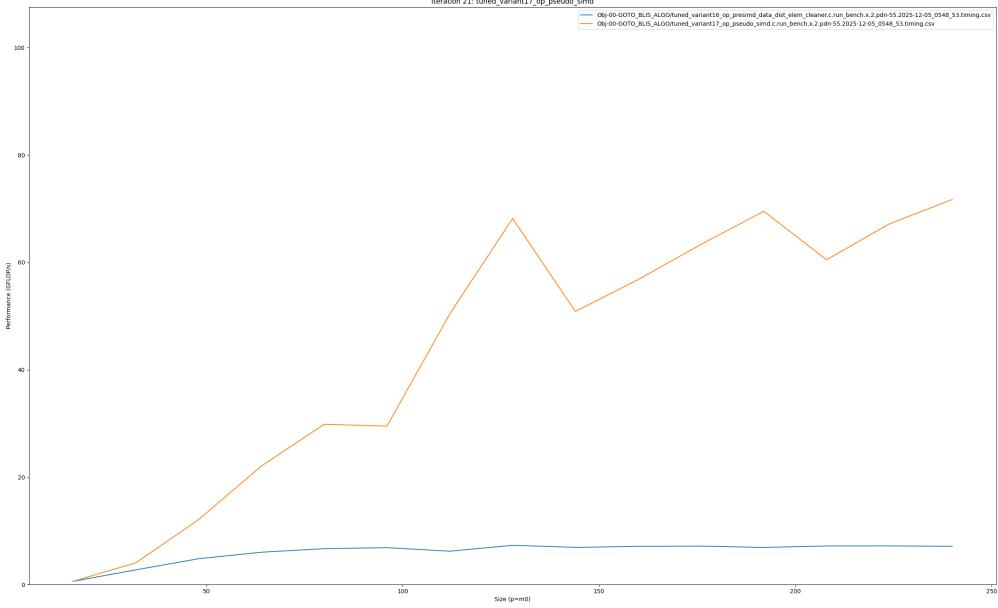


Figure 20: Performance comparison for Iteration 21

3. Explanation

The ‘#pragma omp simd’ pragma directive instructs the compiler to vectorize the ‘i0_r_vid’ loop, enabling AVX2 vector instructions to execute 4 parallel FMA operations per cycle instead of 1 scalar FMA. This, along with the clean data layout from iterations 19 and 20, enables efficient vector loads/stores, optimizing throughput.

Iteration 22: tuned_variant18_op_avx2_simd.c

1. Code Differences

Replaces pragma-driven vectorization with explicit AVX2 intrinsics. Instead of relying on the compiler to generate vector instructions from ‘#pragma omp simd’, now explicitly calls ‘_mm256_*’ functions for load, multiply, accumulate, and store operations. C_micro is now ‘__m256 C_micro_v[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN]’ (vector registers instead of scalar arrays):

```
// Vector accumulator array (4-wide AVX2)
__m256 C_micro_v[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN];

// Initialize with vector zero
#pragma unroll BLOCK_NR
```

```

for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
{
C_micro_v[j0_r][i0_r_bid] = _mm256_set1_ps(0.0f);
}

// Rank-K update with vector FMA
for( int p0_i = 0; p0_i < block_kc_remainder; p0_i += BLOCK_KU )
#pragma unroll BLOCK_KU
for( int p0_u = 0; p0_u < BLOCK_KU; ++ p0_u )
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
{
int j0_i_bid = j0_i/(BLOCK_NR);
int i0_i_bid = i0_i/(BLOCK_MR);

// Load 4 A elements, broadcast 1 B element
__m256 A_ip_v = _mm256_load_ps(&A_dlt[i0_i_bid][p0_i+p0_u][i0_r_bid][0]);
__m256 B_pj_v = _mm256_set1_ps(B_dlt[j0_i_bid][p0_i+p0_u][j0_r]);

// Vector FMA: C = A*B + C
C_micro_v[j0_r][i0_r_bid] = _mm256_fmadd_ps( A_ip_v, B_pj_v, C_micro_v[j0_r][i0_r_bid]);
}

// Write-back from vector to scalar C
#pragma unroll BLOCK_NR
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
#pragma unroll ((BLOCK_MR)/(PAR_VECT_LEN))
for( int i0_r_bid = 0; i0_r_bid < BLOCK_MR/(PAR_VECT_LEN); ++i0_r_bid )
{
float cur_row_sub_vect[PAR_VECT_LEN];
_mm256_storeu_ps(cur_row_sub_vect, C_micro_v[j0_r][i0_r_bid]);

#pragma omp simd
for( int i0_r_vid = 0; i0_r_vid < PAR_VECT_LEN; ++i0_r_vid )
{
int i0_r = i0_r_bid*(PAR_VECT_LEN) + i0_r_vid;
int j0 = j0_o + j0_i + j0_r;
int i0 = i0_o + i0_i + i0_r;

C_distributed[i0 * cs_C + j0 * rs_C] += cur_row_sub_vect[i0_r_vid];
}
}

```

2. Performance Change

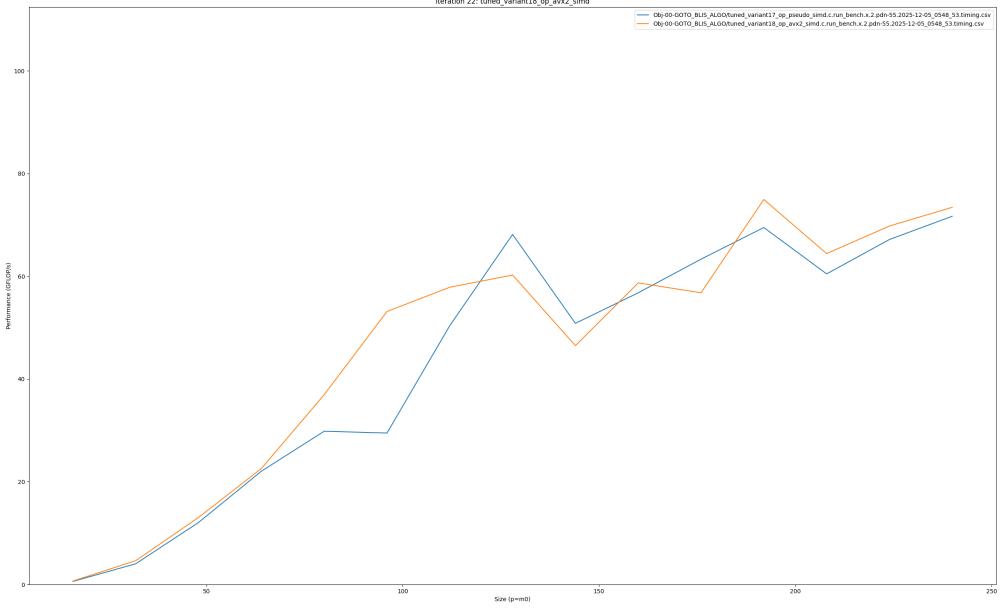


Figure 21: Performance comparison for Iteration 22

3. Explanation

SIMD's explicit AVX2 intrinsics ('`_mm256_load_ps`', '`_mm256_set1_ps`', '`_mm256_fmadd_ps`') enable 4 parallel FMAs per cycle, increasing computational throughput.

Iteration 23: tuned_variant19_op_data_dist_1d_sharablemem.c

1. Code Differences

Introduces OpenMP shared-memory parallelism with 1D work distribution. The outer NC-loop is parallelized with '#pragma omp parallel for num_threads(PAR_COL_THREADS)', distributing column panels across threads:

```
// Shared-memory parallelism: 1D distribution over NC blocks
int num_j0_i_blocks = (BLOCK_NC) / (BLOCK_NR);
int num_i0_i_blocks = (BLOCK_MC) / (BLOCK_MR);

// Distribute NC-blocks across PAR_COL_THREADS threads
#pragma omp parallel for num_threads(PAR_COL_THREADS)
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
{
```

```

int block_nc_remainder = min(BLOCK_NC, n0_fringe_start-j0_o);

// Inner loops: pack B and perform A packing + micro-kernel
for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
{
int block_kc_remainder = min(BLOCK_KC, k0-p0_o);

// Pack B into cache-friendly layout (private to thread)
float B_dlt[num_j0_i_blocks][BLOCK_KC][BLOCK_NR];
for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
for( int j0_r = 0; j0_r < BLOCK_NR; ++j0_r )
{
// B packing with thread-private buffer
B_dlt[j0_i/BLOCK_NR][p0_i][j0_r] = B_distributed[...];
}

// Pack A and perform computation (still within thread)
for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
{
float A_dlt[num_i0_i_blocks][BLOCK_KC][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];
// A packing and micro-kernel computation...
}
}
}
}

```

2. Performance Change

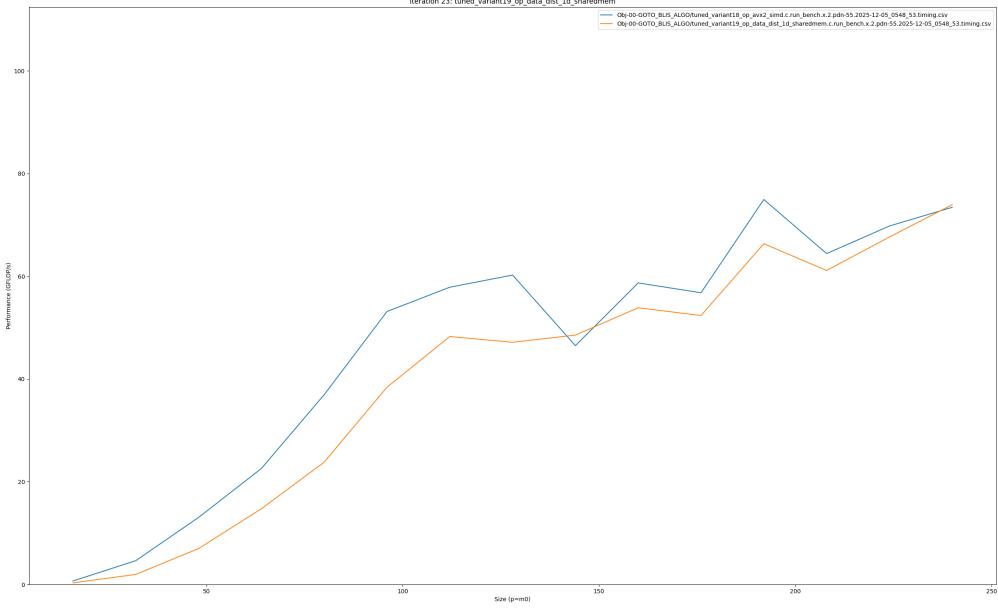


Figure 22: Performance comparison for Iteration 23

3. Explanation

Each thread processes independent $NC \times K \times M$ blocks to avoid data dependencies. Despite this, it seems performance overall regresses in comparison to iteration 22, likely due to 1D parallelism over the NC panels creating a load imbalance when $n0 < PAR_COL_THREADS \times BLOCK_NC$.

Iteration 24: tuned_variant20_op_2d_sharedmem.c

1. Code Differences

Extends 1D OpenMP parallelism to 2D work distribution with nested pragma directives. The outer loop distributes NC-blocks across $PAR_COL_THREADS$ threads, and the inner loop distributes MC-blocks across $PAR_ROW_THREADS$ threads:

```
// 2D shared-memory parallelism: NC x MC decomposition
int num_j0_i_blocks = (BLOCK_NC) / (BLOCK_NR);
int num_i0_i_blocks = (BLOCK_MC) / (BLOCK_MR);

// Outer level: distribute NC-blocks across column threads
#pragma omp parallel for num_threads(PAR_COL_THREADS)
for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
```

```

{
int block_nc_remainder = min(BLOCK_NC, n0_fringe_start-j0_o);

for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
{
// Pack B (shared within column thread)
float B_dlt[num_j0_i_blocks][BLOCK_KC][BLOCK_NR];

// Inner level: distribute MC-blocks across row threads
#pragma omp parallel for num_threads(PAR_ROW_THREADS)
for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
{
// Pack A (private to row thread)
float A_dlt[num_i0_i_blocks][BLOCK_KC][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];

// Micro-kernel computation with SIMD
// Each (row_thread, col_thread) pair computes independent MC x NC x KC block
}
}
}
}

```

2. Performance Change

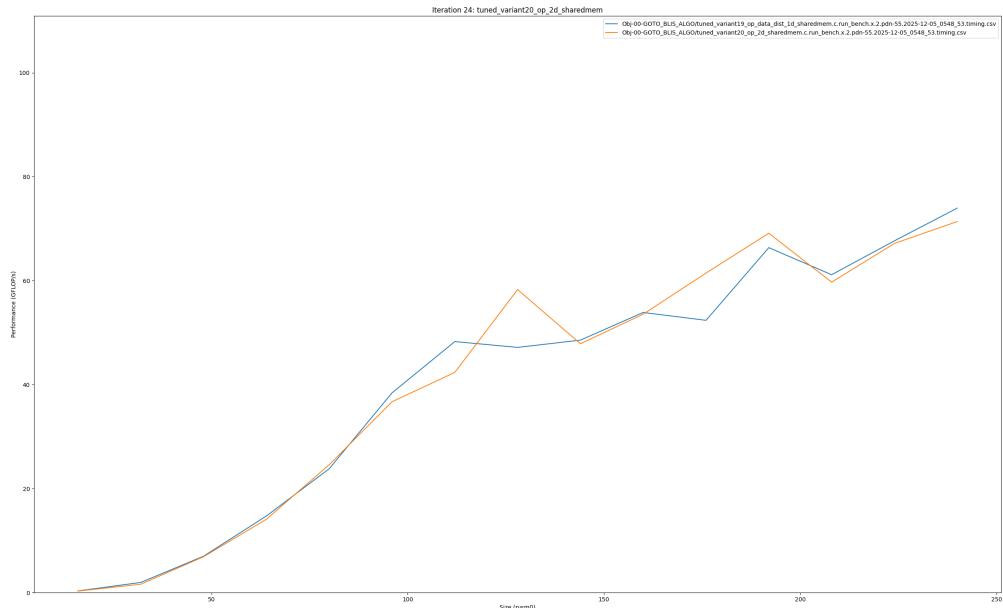


Figure 23: Performance comparison for Iteration 24

3. Explanation

Each thread now works on an $(MC \times NC \times KC)$ sub-block; the nested OpenMP pragmas distribute work across both N (columns) and M (rows) dimensions, improving load balancing and reducing thread idle time. Despite this, it seems there is only optimal alignment between 2D thread decomposition and cache architecture at mid-range problem sizes, while other sizes suffer from increased nested parallelism overhead without corresponding load balance benefits, demonstrating that 2D parallelism's sensitivity to the relationship between problem size, block dimensions, and thread count.