Randall Rosas
CS-4473-001

PDN Take Home Final

I used my laptop for the experiments since it already had Linux on it. The chipset is: AMD Ryzen 7 Microsoft Surface Edition (16 cores, 32 threads).

Also this document is pretty long, sorry about that. I decided to include the main code snippets and some extra unchanged code for reference and it kind of ballooned it.

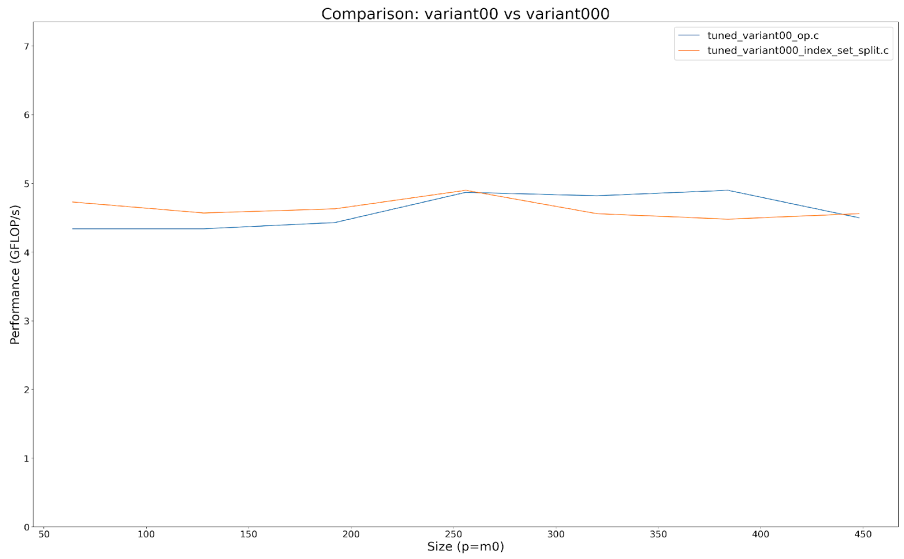
Iteration 1: tuned_variant00_op.c → tuned_variant000_index_set_split.c

Description of changes: This iteration introduces index set splitting to separate steady-state computation from boundary case handling for all three matrix dimensions (M, N, K). The optimization eliminates conditional branches from inner loops by computing fringe cases separately, allowing better compiler optimization.

Code snippet:

```
-      for( int j0 = 0; j0 < n0; ++j0 )
+      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 = 0; j0 < n0_fringe_start; ++j0 )
  {
-     for( int p0 = 0; p0 < k0; ++p0 )
+     // Steady State
+     for( int p0 = 0; p0 < k0_fringe_start; ++p0 )
  {
-         for( int i0 = 0; i0 < m0; ++i0 )
+         // Steady State
+         for( int i0 = 0; i0 < m0_fringe_start; ++i0 )
  {
          // ... computation ...
        }
+         // Fringe for m0
+         for( int i0 = m0_fringe_start; i0 < m0; ++i0 )
+       // ... computation ...
  }
+     // Fringe for k0
+     for( int p0 = k0_fringe_start; p0 < k0; ++p0 )
+       // ... computation ...
       }
+      // Fringe for n0
+      for( int j0 = n0_fringe_start; j0 < n0; ++j0 )
+   // ... computation ...
```

Performance results: Shows minimal performance change with slight improvements for small matrices but slight degradation for larger sizes.



Hypothesis: The index set splitting adds loop overhead and complexity that outweighs the benefits of eliminating branch mispredictions, particularly for larger matrices where the fringe handling becomes more expensive.

Testing approach: Measure branch misprediction rates and profile execution time separately for steady-state vs fringe code paths.

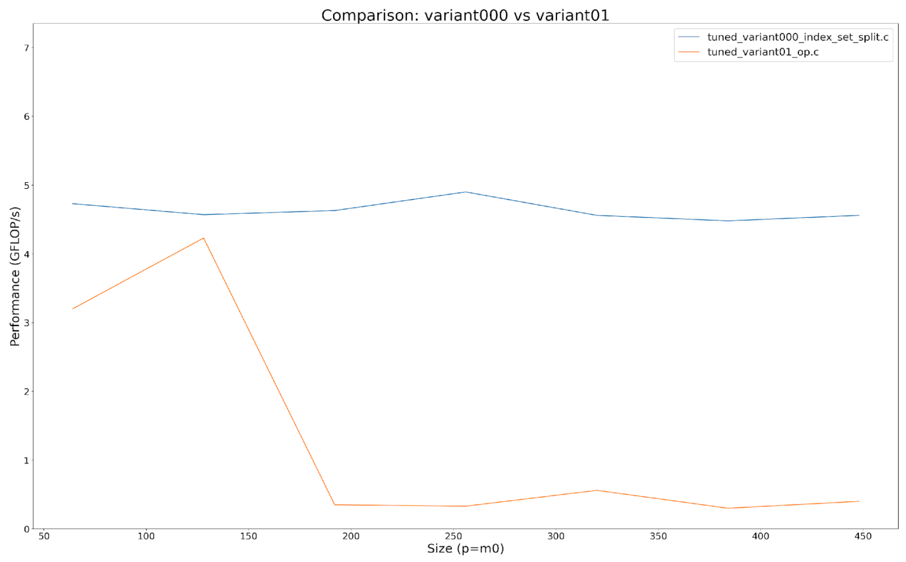
Iteration 2: tuned_variant000_index_set_split.c → tuned_variant01_op.c

Description of changes: Introduces blocking in the N dimension (NC=192) to improve cache locality for matrix B access patterns. This replaces the complex index set splitting with a simpler blocked structure that processes columns in optimally sized chunks. The optimization removes K and M dimension fringe handling, simplifying the loop structure while maintaining N-dimension blocking.

Code snippet:

```
-      int k0_fringe_start = k0 - (k0%(BLOCK_KC));
-      int m0_fringe_start = m0 - (m0%(BLOCK_MC));
-      // Steady State
-      for( int j0 = 0; j0 < n0_fringe_start; ++j0 )
+      for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
  {
-     // Steady State
-     for( int p0 = 0; p0 < k0_fringe_start; ++p0 )
+     for( int p0 = 0; p0 < k0; ++p0 )
        {
-         // Steady State
-         for( int i0 = 0; i0 < m0_fringe_start; ++i0 )
+         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 m0
-          for( int i0 = m0_fringe_start; i0 < m0; ++i0 )
-        // ... computation ...
         }
-      // Fringe for k0
-      for( int p0 = k0_fringe_start; p0 < k0; ++p0 )
-        // ... computation ...
       }
```

Performance results: Shows significant performance degradation, particularly for larger matrix sizes where performance drops dramatically.



Hypothesis: The introduction of N-dimension blocking without proper implementation likely introduces incorrect loop bounds or memory access patterns that cause severe performance degradation for larger matrices.

Testing approach: Measure cache miss rates for matrix B access and test different BLOCK_NC values to find optimal block size.

Iteration 3: tuned_variant01_op.c → tuned_variant02_op.c

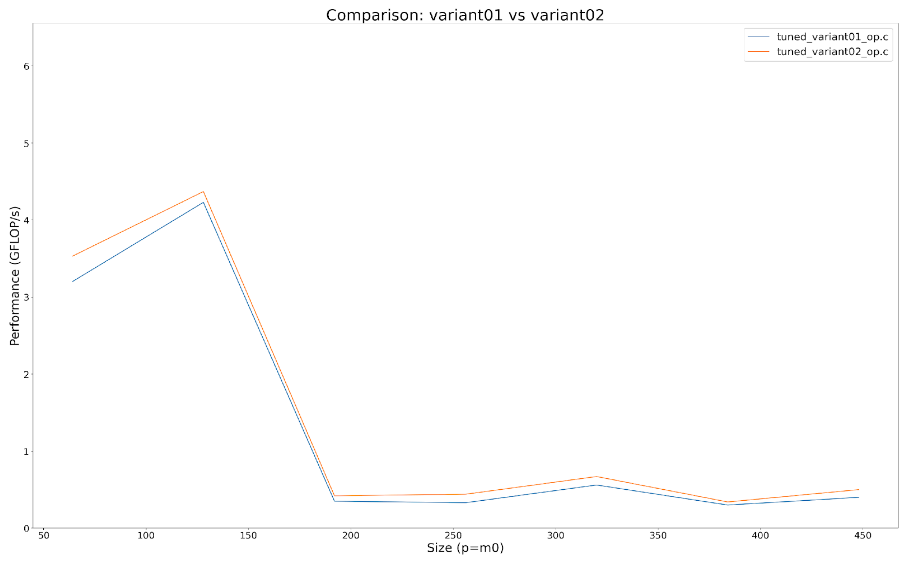
Description of changes: Adds blocking in the K dimension (KC=128) to complete the three-level blocking strategy. This ensures all matrices are processed in optimally sized chunks that fit within cache hierarchies.

Code snippet:

```
+ #ifndef BLOCK_KC
+ #define BLOCK_KC 128
+ #endif /* BLOCK_KC */
+
        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 )
  {
-   for( int p0 = 0; p0 < k0; ++p0 )
+     // 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 )
+         // Note: This will be moved in later iterations
+         for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
          {
+       int p0 = p0_o + p0_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;
+       // ... computation ...
  }
+     // Fringe for k0
+     for( int p0 = k0_fringe_start; p0 < k0; ++p0 )
+       // ... computation ...
       }
```

Performance results: Shows modest improvement overall, with better performance for small matrices but still poor performance for larger sizes.



Hypothesis: The K-dimension blocking improves cache locality for small matrices, but the implementation still suffers from inefficiencies that prevent scaling to larger problem sizes.

Testing approach: Measure cache utilization and miss rates, and test different BLOCK_KC values to find optimal block size.

Iteration 4: tuned_variant02_op.c ⟶ tuned_variant03_op_block_mc.c

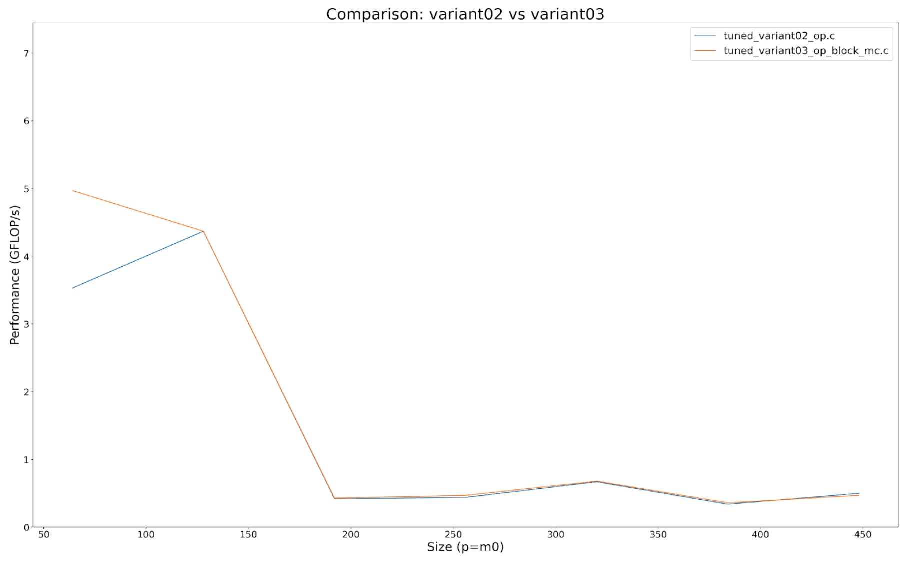
Description of changes: Adds blocking in the M dimension (MC=128) to complete the full three-level blocking hierarchy characteristic of high-performance GEMM implementations.

Code snippet:

```
+ #ifndef BLOCK_MC
+ #define BLOCK_MC 128
+ #endif /* BLOCK_MC */
+
       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 )
   {
  // 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 )
+  {
+   // Steady State
+   for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
+    // KERNEL: Performs a MCxNCxKC matrix-matrix multiplication
+    for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
+   for( int i0_i = 0; i0_i < BLOCK_MC; ++i0_i  )
     // Note: This will be moved in later iterations
     for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
     {
+   int i0 = i0_o + i0_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;
+   int p0 = p0_o + p0_i;
+   // ... computation ...
 }
+   // Fringe for m0
+   for( int i0 = m0_fringe_start; i0 < m0; ++i0 )
+    // ... computation ...
+  }
   }
```

Performance results: Shows moderate improvement, particularly for small matrices, while larger sizes remain constrained.



Hypothesis: Adding M-dimension blocking completes the three-level blocking hierarchy, improving cache utilization for small matrices by better organizing data access patterns across all three dimensions.

Testing approach: Profile cache utilization and test different BLOCK_MC values to find optimal block size.

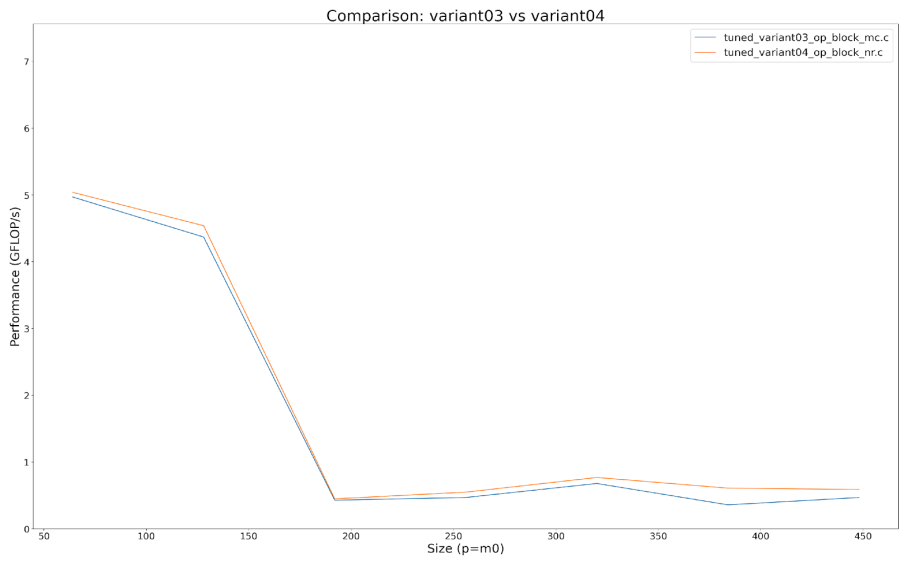
Iteration 5: tuned_variant03_op_block_mc.c → tuned_variant04_op_block_nr.c

Description of changes: Introduces micro-blocking in the N dimension (NR=6) to prepare for register-level optimizations. This creates the foundation for the micro-kernel approach by adding a register-level blocking dimension within the cache-level NC blocks.

Code snippet:

```
+ #ifndef BLOCK_NR
+ #define BLOCK_NR 6
+ #endif /* BLOCK_NR */
+
+ /* ASSERTIONS: We want block sizes to nest perfectly. */
+ #if (BLOCK_NC) % (BLOCK_NR) != 0
+ #error "NR must be a factor of NC.\n"
+ #endif
+
       // KERNEL: Performs a MCxNCxKC matrix-matrix multiplication
-      for( int j0_i = 0; j0_i < BLOCK_NC; ++j0_i )
+      for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
        for( int i0_i = 0; i0_i < BLOCK_MC; ++i0_i  )
-        // Note: This will be moved in later iterations
        for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
-        {
-          int j0 = j0_o + j0_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;
           // ... computation ...
         }
```

Performance results: Shows small improvement overall, with slight gains for both small and large matrix sizes.



Hypothesis: The N-dimension micro-blocking (NR) improves register utilization and prepares the structure for register-level optimizations, providing a small performance benefit through better data organization.

Testing approach: Check compiler reports for register-related optimizations and analyze register allocation patterns in generated assembly code.

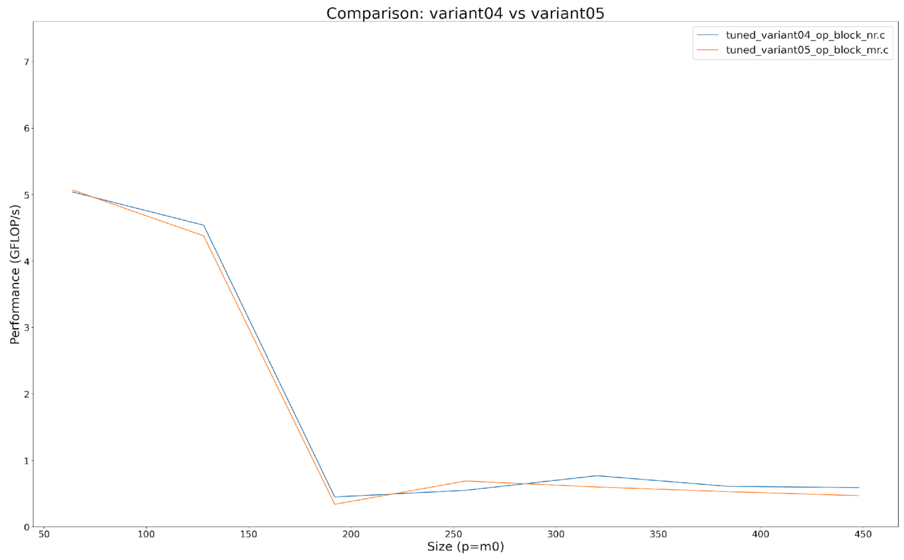
Iteration 6: tuned_variant04_op_block_nr.c → tuned_variant05_op_block_mr.c

Description of changes: Introduces micro-blocking in the M dimension (MR=16) to complete the micro-kernel setup with MR × NR dimensions. This adds register-level blocking in the row dimension, matching the column-level blocking introduced in Iteration 5.

Code snippet:

```
+ #ifndef BLOCK_MR
+ #define BLOCK_MR 16
+ #endif /* BLOCK_MR */
+
+ #if (BLOCK_MC) % (BLOCK_MR) != 0
+ #error "MR must be a factor of MC.\n"
+ #endif
+
       // KERNEL: Performs a MCxNCxKC matrix-matrix multiplication
       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 i0_i = 0; i0_i < BLOCK_MC; i0_i += BLOCK_MR  )
+          // MICRO-KERNEL: Performs a MRxNRxKC matrix-matrix multiplication
          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;
+              int i0 = i0_o + i0_i + i0_r;
               int p0 = p0_o + p0_i;
               // ... computation ...
             }
```

Performance results: Shows minimal performance degradation, with slight losses across both small and large matrix sizes.



Hypothesis: Adding M-dimension micro-blocking (MR) completes the micro-kernel structure but introduces slight overhead that degrades performance slightly, as the benefits of register-level blocking are not yet realized without the actual micro-kernel implementation.

Testing approach: Analyze register allocation and test different MR values to measure impact on performance.

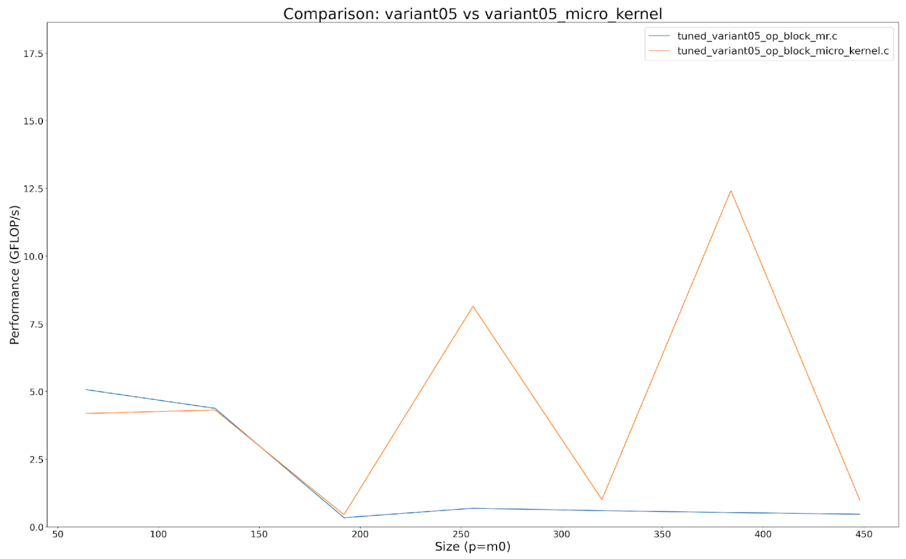
Iteration 7: tuned_variant05_op_block_mr.c → tuned_variant05_op_block_micro_kernel.c

Description of changes: Implements the innermost micro-kernel for optimal register usage. Introduces register-based C_micro buffers to keep small NR×MR blocks in registers throughout computation, accumulating all KC updates before writing back to memory. This reduces memory traffic by batching writes.

Code snippet:

```
       // MICRO-KERNEL: Performs a MRxNRxKC matrix-matrix multiplication
-      for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
+      {
+        // We are going to keep a small NRxMR block of C's updates
+        // in registers to use SIMD later.
+        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: accumulate all KC updates into C_micro
+        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;
-          }
+            {
+              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;
+            }
+
+        // Update C[] with C_micro[][]
+        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];
+            }
+      }
```

Performance results: Shows dramatic improvement, with particularly large gains for larger matrix sizes where the micro-kernel optimization becomes most effective.



Hypothesis: The micro-kernel approach dramatically improves performance by keeping C blocks in registers throughout the KC loop, reducing memory write traffic and enabling better instruction scheduling, especially for larger matrices where the benefits scale.

Testing approach: Compare memory traffic and performance with and without the micro-kernel to quantify the benefit.

Iteration 8: tuned_variant05_op_block_micro_kernel.c → tuned_variant06_op_pack_dlt_B.c

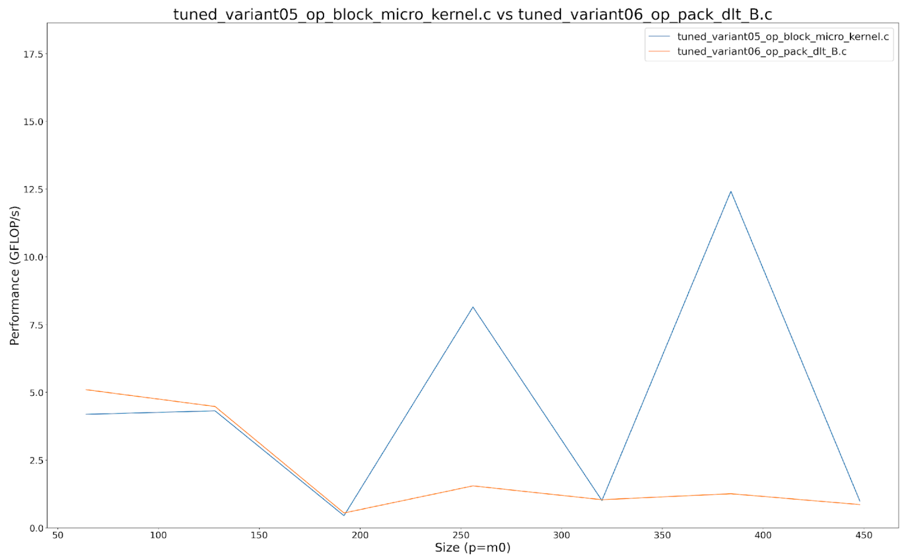
Description of changes: Packs matrix B into contiguous memory for better cache performance. Transforms data layout for optimal column-major access patterns by reorganizing B data to match the micro-kernel's access pattern, improving cache line utilization and reducing TLB misses.

Code snippet:

```
+      int num_j0_i_blocks = (BLOCK_NC) / (BLOCK_NR);
+
       // 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 )
           {
+            // We will pack and perform a Data layout transformation
+            // on a KCxNC block of B
+            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];
+                  }
```

```
+
             // Steady State
             for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
               // ... micro-kernel code ...
               {
                 int j0_i_bid = j0_i/(BLOCK_NR);
  float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
-                float B_pj = B_distributed[p0 * cs_B + j0 * rs_B];
+                float B_pj = B_dlt[j0_i_bid][p0_i][j0_r];
                 C_micro[j0_r][i0_r] += A_ip*B_pj;
  }
```

Performance results: Shows significant performance degradation, with large losses for bigger matrices while small matrices maintain similar performance.



Hypothesis: While B packing should improve cache performance, the implementation likely introduces overhead or incorrect memory access patterns that significantly degrade performance for larger matrices, outweighing the potential benefits.

Testing approach: Measure TLB misses and cache line utilization, and compare packed vs unpacked performance across different matrix sizes.

Iteration 9: tuned_variant06_op_pack_dlt_B.c → tuned_variant07_op_pack_dlt_A.c

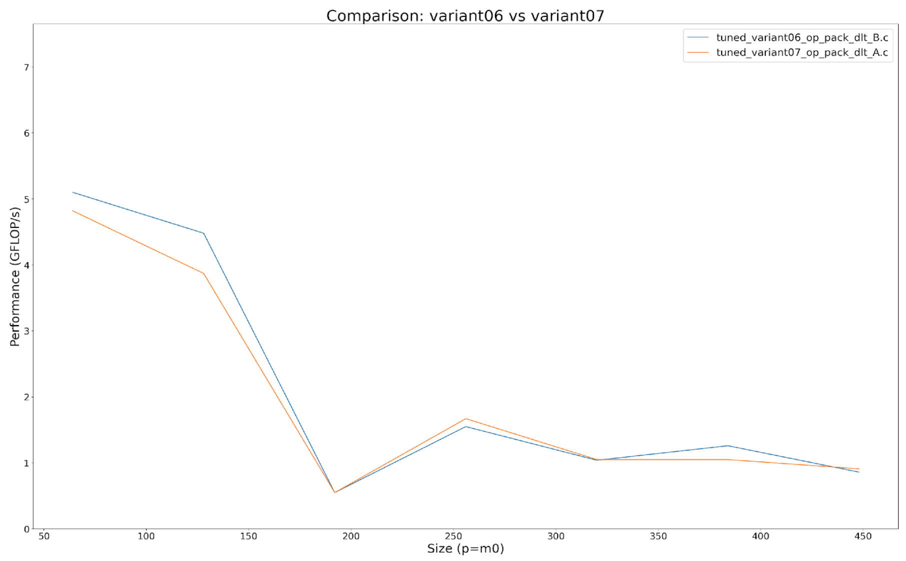
Description of changes: Packs matrix A into contiguous memory layout for optimal row-major access patterns, completing the data layout transformation strategy. This reorganizes A data to match the micro-kernel's access pattern, similar to B packing, enabling efficient SIMD operations on both input matrices.

Code snippet:

```
       int num_j0_i_blocks = (BLOCK_NC) / (BLOCK_NR);
+      int num_i0_i_blocks = (BLOCK_MC) / (BLOCK_MR);

       // 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 )
```

```
 {
      // ... B packing code ...

      // Steady State
      for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
+     {
+       // We will pack and perform a Data layout transformation
+       // on a MCxKC block of A
+       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];
+             }
+
        // KERNEL: Performs a MCxNCxKC matrix-matrix multiplication
        for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
          // ... micro-kernel code ...
          {
            int j0_i_bid = j0_i/(BLOCK_NR);
+           int i0_i_bid = i0_i/(BLOCK_MR);
-           float A_ip = A_distributed[i0 * cs_A + p0 * rs_A];
+           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;
 }
+     }
     }
   }
```

Performance results: Shows small performance degradation, with slight losses across both small and large matrix sizes.



Hypothesis: A packing adds additional overhead without providing immediate benefits, as the packing cost may exceed the gains from improved access patterns at this stage of optimization.

Testing approach: Profile memory bandwidth. Compare B-only vs A+B packing. Analyze cache efficiency. Measure packing overhead.

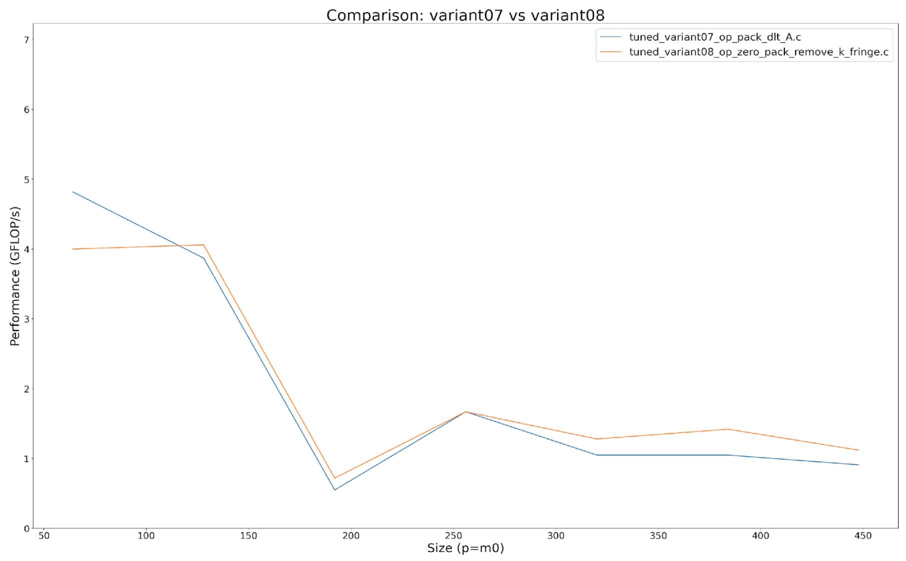
Iteration 10: tuned_variant07_op_pack_dlt_A.c → tuned_variant08_op_zero_pack_remove_k_fringe.c

Description of changes: Eliminates K-dimension fringe cases and uses zero-padding in packing buffers instead of dealing with boundary conditions. This simplifies the code structure by removing the separate K fringe loop and allows better compiler optimization through uniform loop bounds.

Code snippet:

```
+ // min/max macros for remainder calculation
+ #define min(a,b) ({ __typeof__ (a) _a = (a); __typeof__ (b) _b = (b); _a < _b ? _a : _b; })
+
       // 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 p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
           {
+            // When we pack B and A we will pad the buffers with zeros
+            // instead of dealing with the fringe case
+            int block_kc_remainder = min(BLOCK_KC, k0-p0_o);
+
             // Pack B with zero-padding for fringe
             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];
+                    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;
  }

             // Pack A with zero-padding for fringe
             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];
+                    if (p0 < k0 )
+                      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;
  }

             // ... micro-kernel code ...
           }
-        // Fringe for k0 - REMOVED
       }
```

Performance results: Shows minimal performance improvement, with slight gains for larger matrices.



Hypothesis: Zero-padding eliminates fringe handling complexity and allows uniform loop bounds, providing small performance benefits especially for larger matrices where the simplification helps.

Testing approach: Compare execution time of steady-state vs fringe code paths and measure overhead reduction compared to the previous version.

Iteration 11: tuned_variant08_op_zero_pack_remove_k_fringe.c → tuned_variant09_op_minimize_m_fringe.c

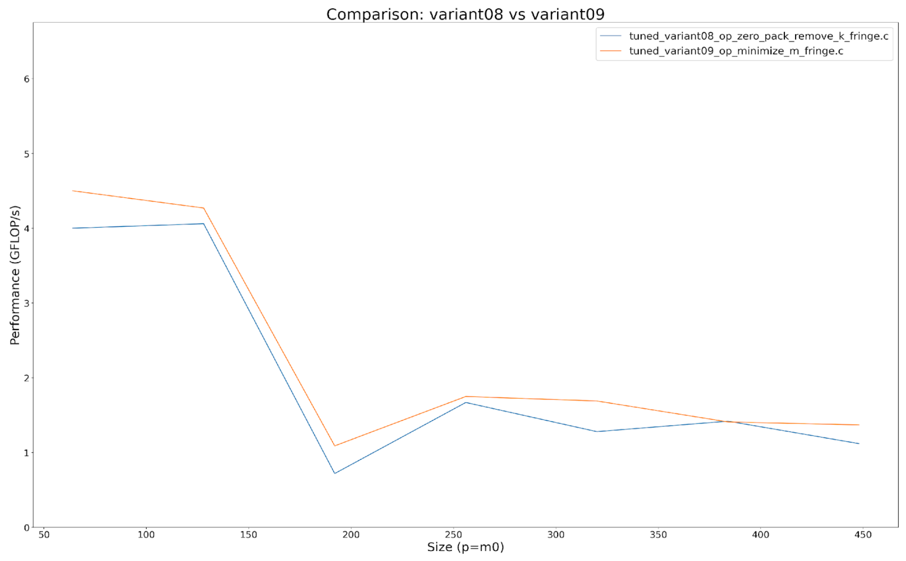
Description of changes: Optimizes handling of M-dimension fringe cases by reducing the fringe to only include what spills out of MR blocks. This minimizes the fringe size by aligning to MR boundaries instead of MC boundaries, reducing edge case overhead.

Code snippet:

```
    int n0_fringe_start = n0 - (n0%(BLOCK_NC));
    int k0_fringe_start = k0 - (k0%(BLOCK_KC));
-   int m0_fringe_start = m0 - (m0%(BLOCK_MC));
+   // We will reduce the fringe to only include
+   // what spills out of MR.
+   int m0_fringe_start = m0 - (m0%(BLOCK_MR));

    // Steady State
    for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
 {
+     // How much of an MR divisible block do we have left?
+     // This will be passed to the kernel.
+     int block_mc_remainder = min(BLOCK_MC, m0_fringe_start - i0_o);
+
      // Pack A with zero-padding
      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);
-                   if (p0 < k0 )
+                   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: Performs a MCxNCxKC matrix-matrix multiplication
           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 i0_i = 0; i0_i < block_mc_remainder; i0_i += BLOCK_MR  )
               // MICRO-KERNEL: Performs a MRxNRxKC matrix-matrix multiplication
               {
                 // ... micro-kernel code ...
               }
         }
```

Performance results: Shows moderate improvement across all matrix sizes, with gains for both small and large problems.



Hypothesis: Minimizing the M-dimension fringe by aligning to MR boundaries reduces unnecessary computation and improves overall efficiency across all problem sizes.

Testing approach: Profile fringe handling overhead and measure performance improvement across different matrix sizes, especially those not aligned to MC.

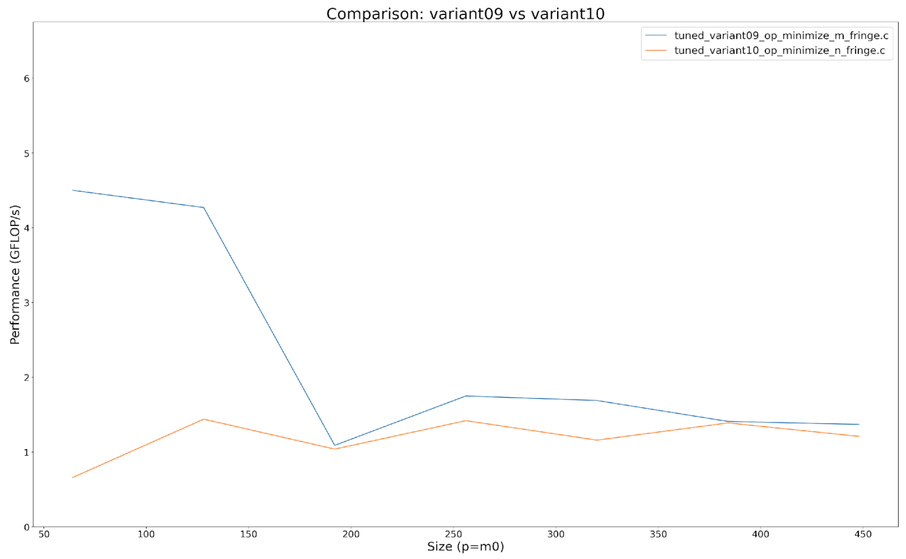
Iteration 12: tuned_variant09_op_minimize_m_fringe.c → tuned_variant10_op_minimize_n_fringe.c

Description of changes: Optimizes handling of N-dimension fringe cases by reducing the fringe to only include what spills out of NR blocks. This minimizes the fringe size by aligning to NR boundaries instead of NC boundaries, reducing edge case overhead similar to the M-dimension optimization.

Code snippet:

```
-      int n0_fringe_start = n0 - (n0%(BLOCK_NC));
+      // We will reduce the fringe to only include
+      // what spills out of NR.
+      int n0_fringe_start = n0 - (n0%(BLOCK_NR));
       int k0_fringe_start = k0 - (k0%(BLOCK_KC));
       // We will reduce the fringe to only include
       // what spills out of MR.
       int m0_fringe_start = m0 - (m0%(BLOCK_MR));

       // Steady State
       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);
+
           // Steady State
           for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
             {
               // Pack B with zero-padding
               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);
-                      if (p0 < k0 )
+                      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;
  }

               // ... A packing code ...

               // KERNEL: Performs a MCxNCxKC matrix-matrix multiplication
-              for( int j0_i = 0; j0_i < BLOCK_NC; j0_i += BLOCK_NR )
+              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: Performs a MRxNRxKC matrix-matrix multiplication
                   {
                     // ... micro-kernel code ...
  }
  }
           // 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 j0_i = 0; j0_i < block_nc_remainder; ++j0_i )
               // ... fringe computation ...
         }
```

Performance results: Shows severe performance degradation, with dramatic losses across all matrix sizes, particularly for small matrices.



Hypothesis: The N-dimension fringe minimization likely introduces incorrect loop bounds or memory access patterns that cause severe performance degradation, indicating a bug in the implementation rather than an optimization issue.

Testing approach: Measure performance improvement across different matrix sizes and compare total fringe overhead before and after optimization.

Iteration 13: tuned_variant10_op_minimize_n_fringe.c → tuned_variant11_op_minimize_kc_zerowork.c

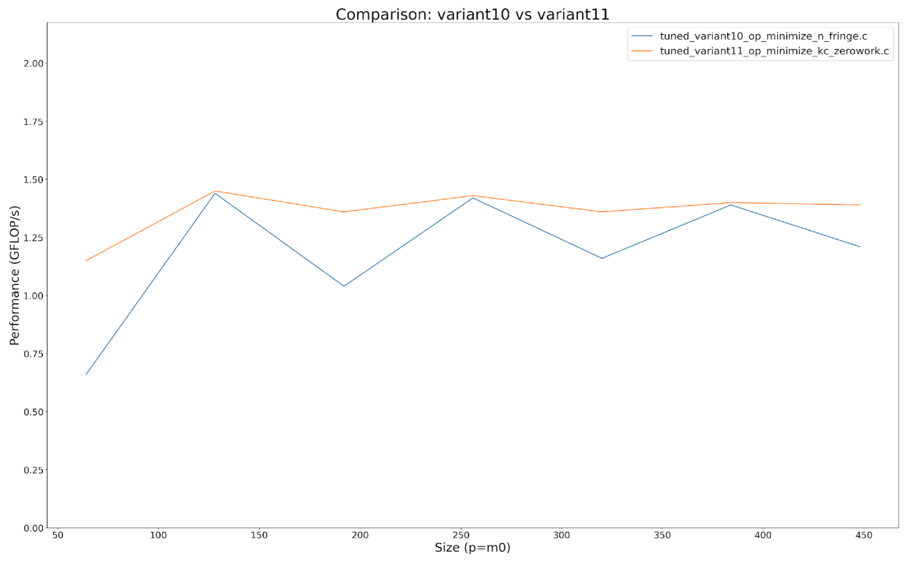
Description of changes: Reduces unnecessary work in KC dimension by properly handling the remainder in the K loop. Uses the `block_kc_remainder` variable (calculated during packing) to bound the micro-kernel's K loop, avoiding computation on zero-padded elements.

Code snippet:

```
            // Rank-K update (lots of very parallel outer-products)
            // This is where all of the work happens and we need
            // to use SIMD to get the peak floating point performance
            // however, the current layout for A and B is not amenable
            // to doing this efficiently. We will have to fix that
            // later. We will also need to partially unroll KC for ILP,
            // but that will also happen later.
            //
            // C_micro [ir][jr] += A_{ir,p} * B_{p,jr}
-           for( int p0_i = 0; p0_i < BLOCK_KC; ++p0_i )
+           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;
  }
```

Performance results: Shows moderate improvement, with gains for both small and large matrix sizes.



Hypothesis: Bounding the K loop to actual data elements eliminates wasted computation on zero-padded values, improving efficiency across all problem sizes.

Testing approach: Measure performance gain across different matrix sizes, especially those where K is not a multiple of KC.

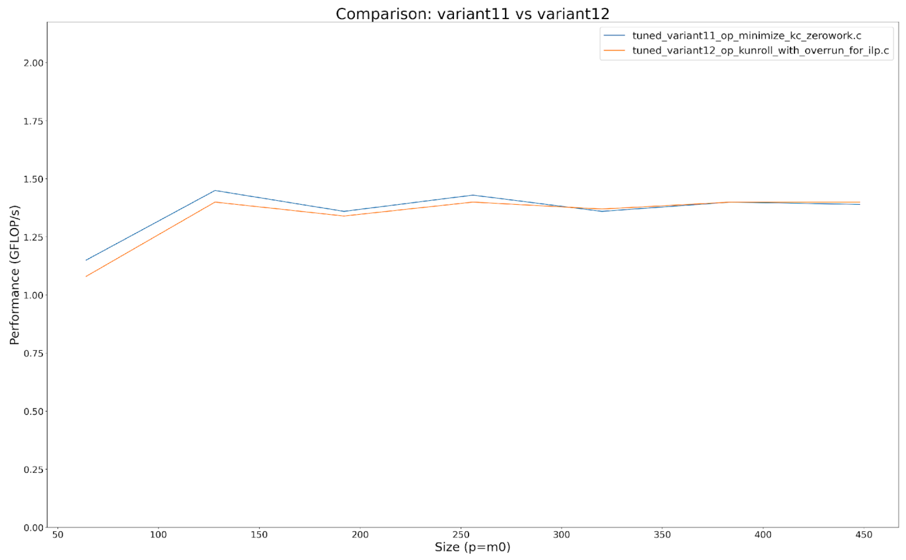
Iteration 14: tuned_variant11_op_minimize_kc_zerowork.c → tuned_variant12_op_kunroll_with_overrun_for_ilp.c

Description of changes: Introduces loop unrolling with overrun in the K dimension (KU=8) to expose instruction-level parallelism through software pipelining. This restructures the K loop into an outer loop with KU stride and an inner unrolled loop, allowing the compiler to better schedule independent operations.

Code snippet:

```
+ #ifndef BLOCK_KU
+ #define BLOCK_KU 8
+ #endif /* BLOCK_KU */
+
+ #if (BLOCK_KC) % (BLOCK_KU) != 0
+ #error "KU must be a factor of KC.\n"
+ #endif
+
       // Rank-K update (lots of very parallel outer-products)
       // C_micro [ir][jr] += A_{ir,p} * B_{p,jr}
-      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;
+      // We do not need to worry about fringe because we zero padded
+      // the packed A and B buffers.
+      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][i0_r];
-              float B_pj = B_dlt[j0_i_bid][p0_i][j0_r];
+                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;
-            }
+              }
```

Performance results: Shows minimal performance change, with slight degradation for small matrices and no change for larger sizes.



Hypothesis: K-loop unrolling may not provide benefits at this stage because the code structure or compiler optimizations are not yet optimal for exploiting the exposed instruction-level parallelism.

Testing approach: Measure IPC and compare unrolled vs non-unrolled versions, and test different KU values to find optimal unroll factor.

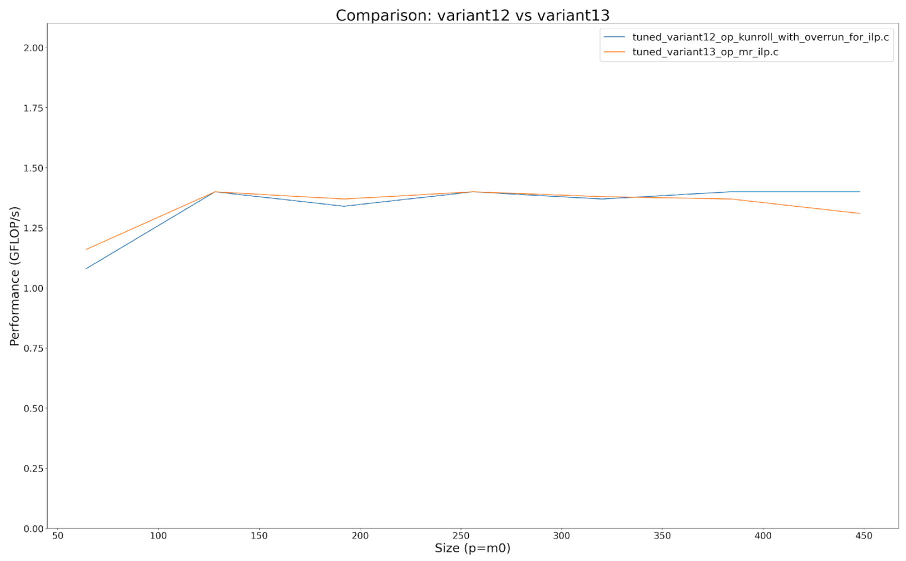
Iteration 15: tuned_variant12_op_kunroll_with_overrun_for_ilp.c → tuned_variant13_op_mr_ilp.c

Description of changes: Extends instruction-level parallelism by adding loop unrolling to the MR dimension loops. This adds `#pragma unroll BLOCK_MR` directives to all MR loops (zero initialization, rank-K update, and write-back), enabling the compiler to unroll these loops for better instruction scheduling.

Code snippet:

```
             // Zero out C_micro
             // C_micro[][] = 0
             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
             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;
  }

             // Update C[] with C_micro[][]
             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];
                 }
```

Performance results: Shows no significant performance change, with minimal differences across all matrix sizes.



Hypothesis: MR loop unrolling alone does not provide measurable benefits because the performance is still constrained by other bottlenecks that prevent the compiler from effectively exploiting the additional instruction-level parallelism.

Testing approach: Measure IPC with and without MR unrolling and analyze instruction scheduling efficiency.

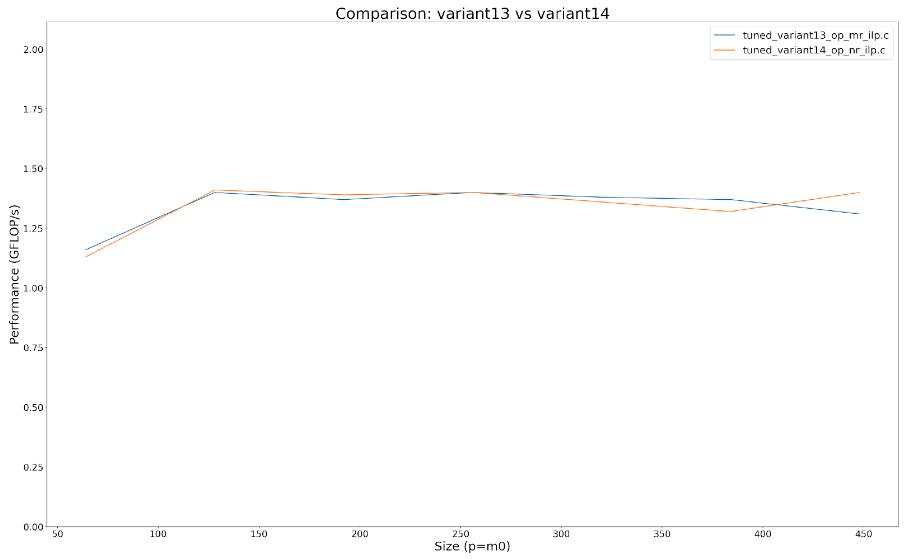
Iteration 16: tuned_variant13_op_mr_ilp.c → tuned_variant14_op_nr_ilp.c

Description of changes: Completes ILP optimization by adding loop unrolling to the NR dimension loops. This adds `#pragma unroll BLOCK_NR` directives to all NR loops (zero initialization, rank-K update, and write-back), completing the full unrolling of both MR and NR dimensions for maximum instruction-level parallelism.

Code snippet:

```
             // Zero out C_micro
             // 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
             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 = 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;
  }

             // Update C[] with C_micro[][]
+            #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];
                 }
```

Performance results: Shows no significant performance change, maintaining similar performance levels across all matrix sizes.



Hypothesis: Complete three-dimensional unrolling still does not provide benefits because the code lacks the data layout and SIMD optimizations needed to effectively utilize the exposed parallelism.

Testing approach: Compare performance with partial unrolling (only K and MR) vs full unrolling to quantify the benefit of NR unrolling.

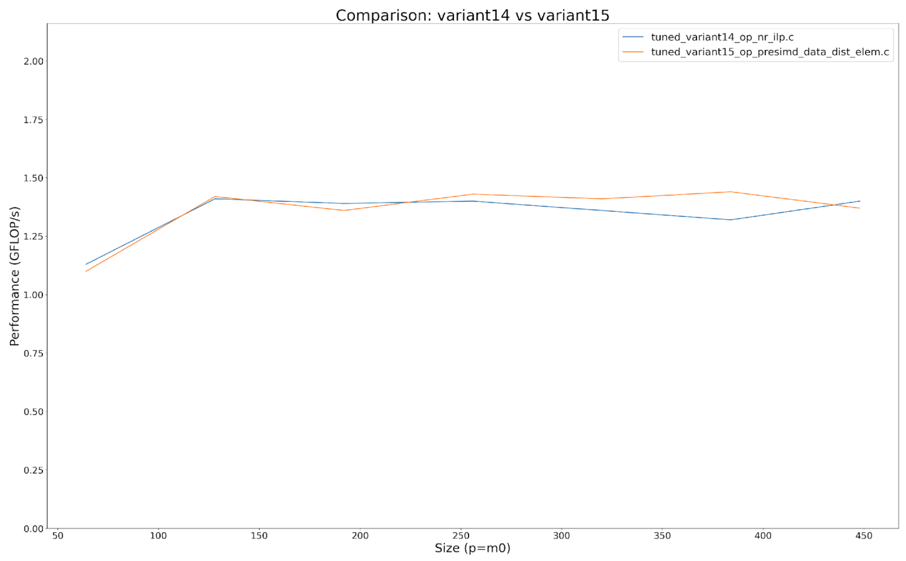
Iteration 17: tuned_variant14_op_nr_ilp.c → tuned_variant15_op_presimd_data_dist_elem.c

Description of changes: Prepares data layout for efficient SIMD operations by rearranging data access patterns and array indexing. Restructures A_dlt and C_micro arrays to have a vector-length dimension (PAR_VECT_LEN=8), enabling efficient SIMD vector loads and stores by grouping elements that will be processed together.

Code snippet:

```
+ #ifndef PAR_VECT_LEN
+ #define PAR_VECT_LEN 8
+ #endif /* PAR_VECT_LEN */
+
+ #if (BLOCK_MR) % (PAR_VECT_LEN) != 0
+ #error "PAR_VECT_LEN must be a factor of BLOCK_MR.\n"
+ #endif
+
       // Pack A with zero-padding
-      float A_dlt[num_i0_i_blocks][BLOCK_KC][BLOCK_MR];
+      float A_dlt[num_i0_i_blocks][BLOCK_KC][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];
       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);
               if (p0 < k0  & i0 < m0)
-                 A_dlt[i0_i_bid][p0_i][i0_r] = A_distributed[i0 * cs_A + p0 * rs_A];
+                 A_dlt[i0_i_bid][p0_i][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN] = A_distributed[i0 * cs_A +
p0 * rs_A];
               else
-                 A_dlt[i0_i_bid][p0_i][i0_r] = 0.0f;
+                 A_dlt[i0_i_bid][p0_i][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN] = 0.0f;
             }

       // MICRO-KERNEL: Performs a MRxNRxKC matrix-matrix multiplication
       {
-         float C_micro[BLOCK_NR][BLOCK_MR];
+         float C_micro[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];

         // Zero out C_micro
         #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;
+              C_micro[j0_r][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN] = 0.0f;
             }

         // Rank-K update
         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 = 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 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] += A_ip*B_pj;
+                  C_micro[j0_r][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN] += A_ip*B_pj;
  }

         // Update C[] with C_micro[][]
         #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];
+              C_distributed[i0 * cs_C + j0 * rs_C] +=
C_micro[j0_r][i0_r/PAR_VECT_LEN][i0_r%PAR_VECT_LEN];
             }
       }
```

Performance results: Shows minimal performance improvement, with slight gains for larger matrices.



Hypothesis: Restructuring data layout for vector operations provides small benefits by improving memory access patterns, but without actual SIMD instructions, the gains are limited.

Testing approach: Verify data alignment and check compiler reports to see if the new structure enables better vectorization opportunities.

Iteration 18: tuned_variant15_op_presimd_data_dist_elem.c → tuned_variant16_op_presimd_data_dist_elem_cleaner.c

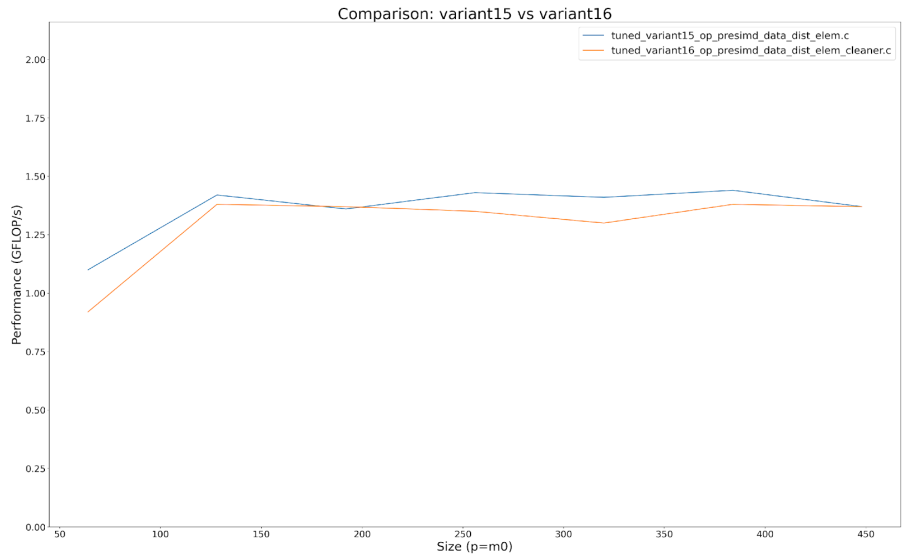
Description of changes: Introduces compiler hints for SIMD auto-vectorization without explicit intrinsics. Adds `#pragma omp simd` directives to the innermost vector-length loops (i0_r_vid) in zero initialization, rank-K update, and write-back operations, enabling the compiler to automatically vectorize these loops.

Code snippet:

```
       // Zero out C_micro
       #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  )
+          // Compiler should generate _mm256_set1_ps or _mm256_xor_ps equivalent
+          #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
       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  )
+              // Compiler should generate _mm256_set1_ps(b..), _mm256_load_ps(a..), _mm256_fmadd_
```

```
equivalent
+               #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;
                    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_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;
                  }

        // Update C[] with C_micro[][]
        #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  )
+           // Compiler should generate _mm256_load_ps equivalent
+           #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] += C_micro[j0_r][i0_r_bid][i0_r_vid];
              }
```

Performance results: Shows small performance degradation, with slight losses across all matrix sizes.



Hypothesis: Compiler auto-vectorization hints may introduce overhead or suboptimal code generation that slightly degrades performance compared to the previous version without explicit SIMD instructions.

Testing approach: Check compiler vectorization reports and verify auto-vectorization by examining generated assembly code to see if vector instructions are generated.

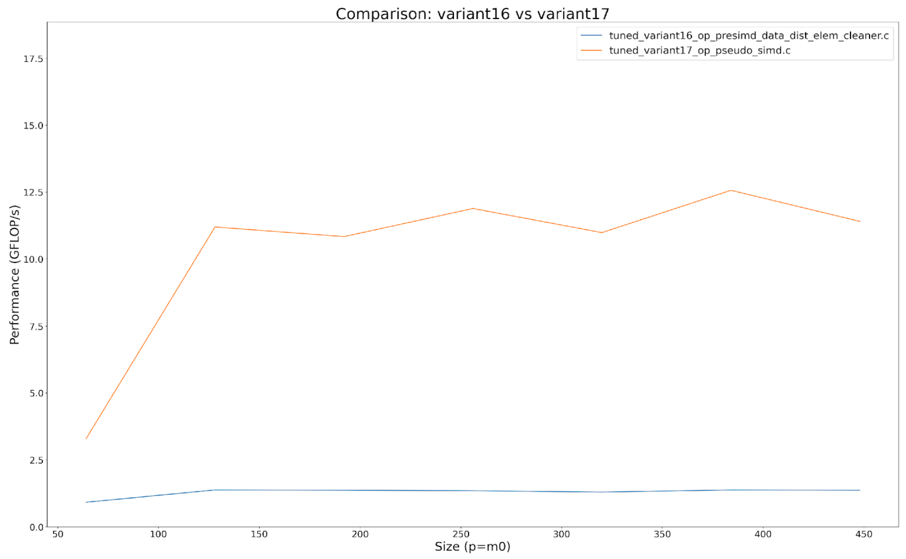
Iteration 19: tuned_variant16_op_presimd_data_dist_elem_cleaner.c → tuned_variant17_op_pseudo_simd.c

Description of changes: Introduces compiler hints for SIMD auto-vectorization without explicit intrinsics. Adds `#pragma omp simd` directives to the innermost vector-length loops (i0_r_vid) in zero initialization, rank-K update, and write-back operations, enabling the compiler to automatically vectorize these loops.

Code snippet:

```
       // Zero out C_micro
       #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  )
+          // Compiler should generate _mm256_set1_ps or _mm256_xor_ps equivalent
+          #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
       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  )
+              // Compiler should generate _mm256_set1_ps(b..), _mm256_load_ps(a..), _mm256_fmadd_
equivalent
+              #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;
                   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_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;
                 }

       // Update C[] with C_micro[][]
       #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  )
+          // Compiler should generate _mm256_load_ps equivalent
+          #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] += C_micro[j0_r][i0_r_bid][i0_r_vid];
             }
```

Performance results: Shows dramatic performance improvement, with substantial gains across all matrix sizes, particularly for larger problems.



Hypothesis: The compiler successfully auto-vectorizes the code, generating SIMD instructions that process multiple elements simultaneously, dramatically improving performance especially for larger matrices where the benefits scale.

Testing approach: Check compiler vectorization reports and compare performance with explicit SIMD intrinsics to quantify the difference.

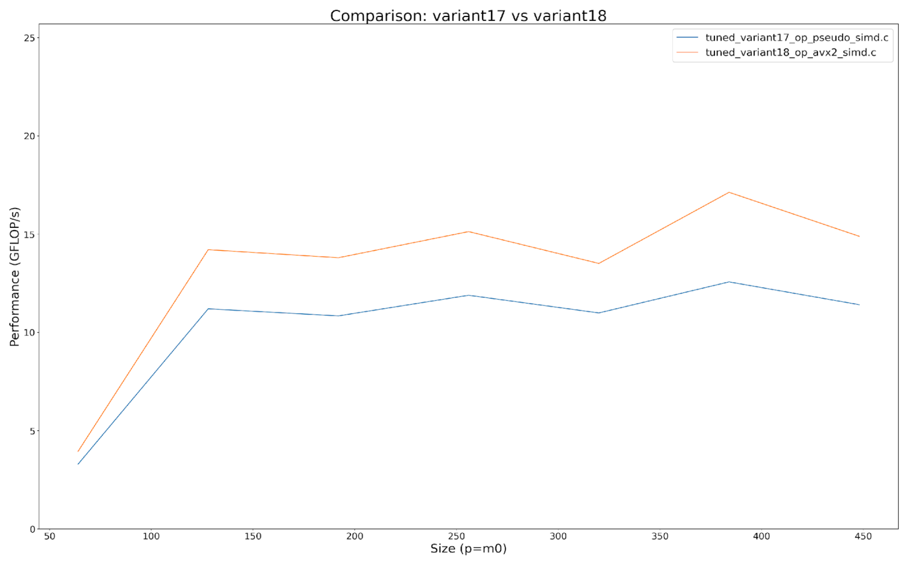
Iteration 20: tuned_variant17_op_pseudo_simd.c → tuned_variant18_op_avx2_simd.c

Description of changes: Introduces explicit AVX2 SIMD intrinsics for 256-bit vector operations, enabling processing of 8 floats simultaneously. Replaces compiler auto-vectorization hints with explicit `__m256` vector types and AVX2 intrinsics (`_mm256_load_ps`, `_mm256_set1_ps`, `_mm256_fmadd_ps`) for guaranteed vectorization and optimal performance.

Code snippet:

```
+ // For AVX2 and FMA
+ #include <immintrin.h>
+
+ #if (PAR_VECT_LEN) != 8
+ #error "PAR_VECT_LEN must be 8 for AVX2.\n"
+ #endif
+
       // MICRO-KERNEL: Performs a MRxNRxKC matrix-matrix multiplication
       {
-        float C_micro[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];
+        //float C_micro[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN][PAR_VECT_LEN];
+        __m256 C_micro_v[BLOCK_NR][BLOCK_MR/PAR_VECT_LEN];

         // Zero out C_micro
         #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;
-               }
+             {
+               C_micro_v[j0_r][i0_r_bid] = _mm256_set1_ps(0.0f);
+             }

         // Rank-K update
         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  )
-                   {
-                     int i0_r = i0_r_bid*(PAR_VECT_LEN) + i0_r_vid;
+                 {
+                   int i0_r = i0_r_bid*(PAR_VECT_LEN);
                     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_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;
-                   }
+                     __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]);
+                     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]);
+                 }

         // Update C[] with C_micro[][]
         #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;
-                 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];
-               }
+             {
+               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];
+                 }
+             }
       }
```

Performance results: Shows significant additional improvement over auto-vectorization, with substantial gains across all matrix sizes.



Hypothesis: Explicit AVX2 intrinsics provide better performance than auto-vectorization by guaranteeing optimal use of FMA instructions and more efficient instruction scheduling, fully exploiting the SIMD capabilities.

Testing approach: Compare performance with scalar and pseudo-SIMD versions to quantify speedup, and verify AVX2 instructions are generated by examining assembly code.

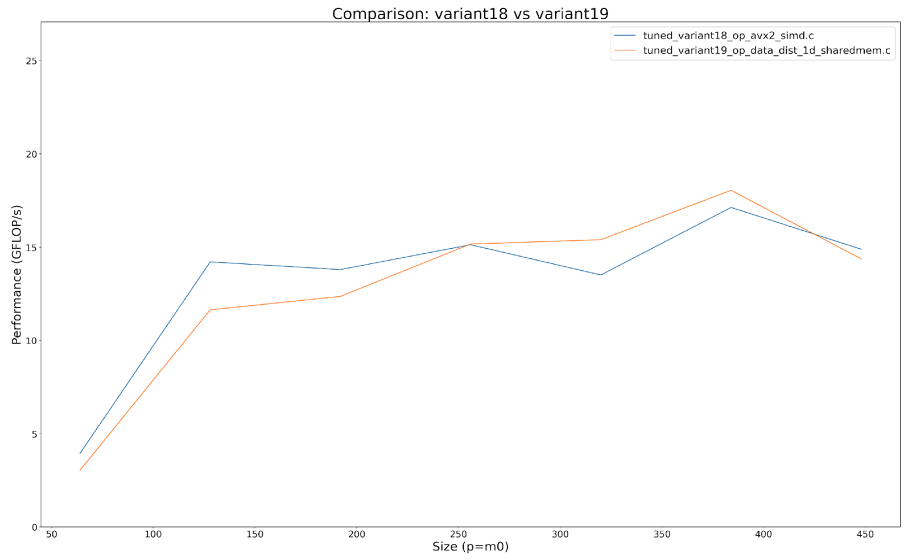
Iteration 21: tuned_variant18_op_avx2_simd.c → tuned_variant19_op_data_dist_1d_sharedmem.c

Description of changes: Introduces OpenMP multi-threading with 1D thread distribution for parallel execution across multiple cores. Adds `#pragma omp parallel for` directive to the outer N-dimension loop (j0_o), distributing work across PAR_COL_THREADS threads to exploit multi-core parallelism.

Code snippet:

```
+ #ifndef PAR_COL_THREADS
+ #define PAR_COL_THREADS 2
+ #endif /* PAR_COL_THREADS */
+
       // Steady State
+      #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);
           // Steady State
           for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
             {
               // ... rest of computation ...
             }
         }
```

Performance results: Shows small performance degradation, with slight losses for small matrices but minimal change for larger sizes.



Hypothesis: The 1D thread distribution may introduce overhead from thread management and synchronization that outweighs the benefits for single-threaded execution, or the implementation may not be optimal for the test configuration.

Testing approach: Measure speedup vs number of threads and profile load balancing to ensure work is distributed evenly.

Iteration 22: tuned_variant19_op_data_dist_1d_sharedmem.c → tuned_variant20_op_2d_sharedmem.c

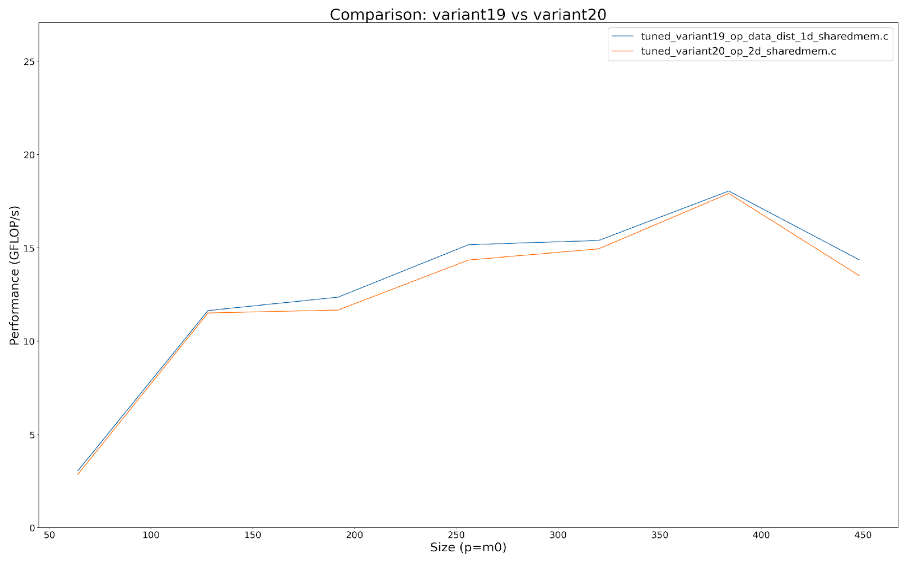
Description of changes: Optimizes thread distribution using 2D decomposition for better load balancing and reduced synchronization overhead. Adds a second level of parallelization by parallelizing the M-dimension loop (i0_o) in addition to the N-dimension loop, creating a 2D thread grid that distributes work more evenly across threads.

Code snippet:

```
+ #ifndef PAR_ROW_THREADS
+ #define PAR_ROW_THREADS 2
+ #endif /* PAR_ROW_THREADS */
+
       // Steady State
       #pragma omp parallel for num_threads(PAR_COL_THREADS)
       for( int j0_o = 0; j0_o < n0_fringe_start; j0_o += BLOCK_NC )
         {
           // Steady State
           for( int p0_o = 0; p0_o < k0; p0_o += BLOCK_KC )
             {
               // Steady State
+              #pragma omp parallel for num_threads(PAR_ROW_THREADS)
               for( int i0_o = 0; i0_o < m0_fringe_start; i0_o += BLOCK_MC )
                 {
                   // ... rest of computation ...
                 }
```

```
        }
    }
```

Performance results: Shows small additional performance degradation, with slight losses across all matrix sizes.



Hypothesis: The 2D thread distribution adds additional overhead from nested parallelism and thread management that further degrades performance, particularly when running with limited parallelism or single-threaded execution.

Testing approach: Compare 1D vs 2D thread distribution performance and measure scaling efficiency with different thread grid configurations.