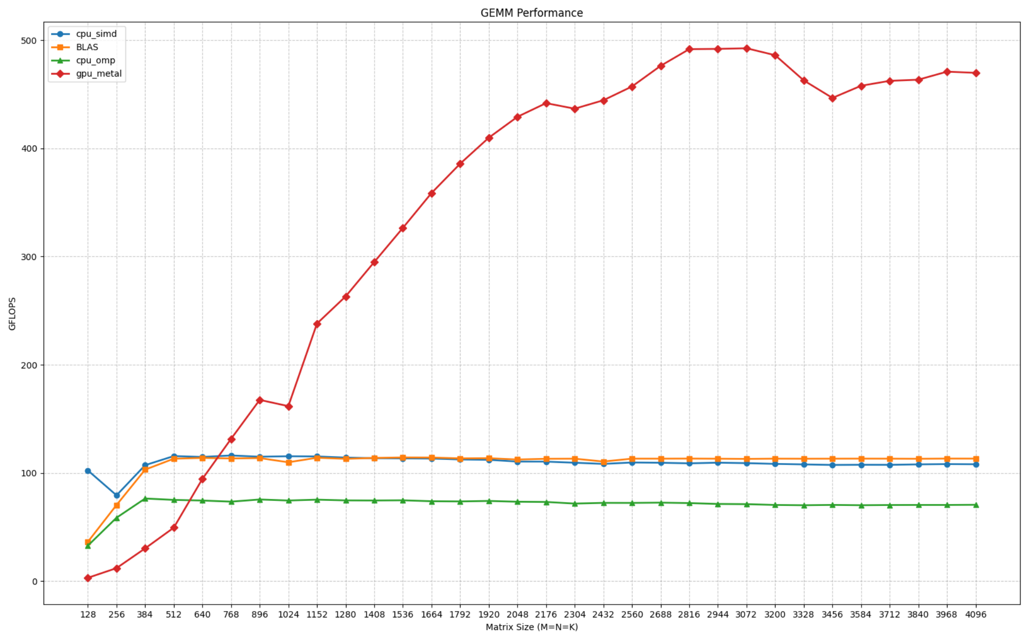
GEMM Optimization

- Diac Liu diacl@andrew.cmu.edu @diacccc
- Min Xiao minxiao@andrew.cmu.edu @Echo-minn
- Gaven Wang jiayuw3@andrew.cmu.edu @GavenWang1014

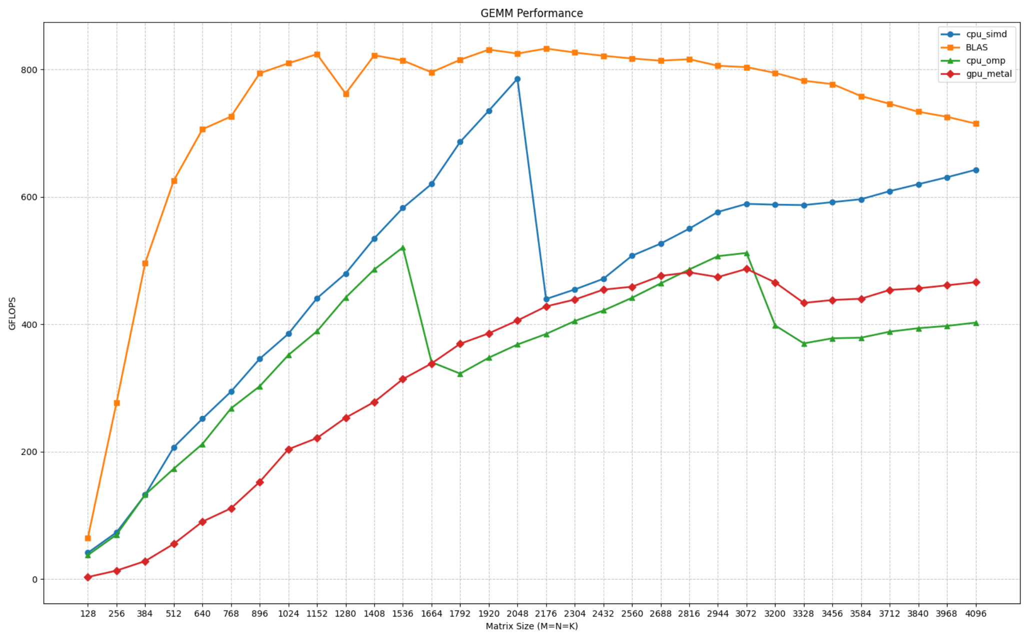
Experimental Results

Arm Neon

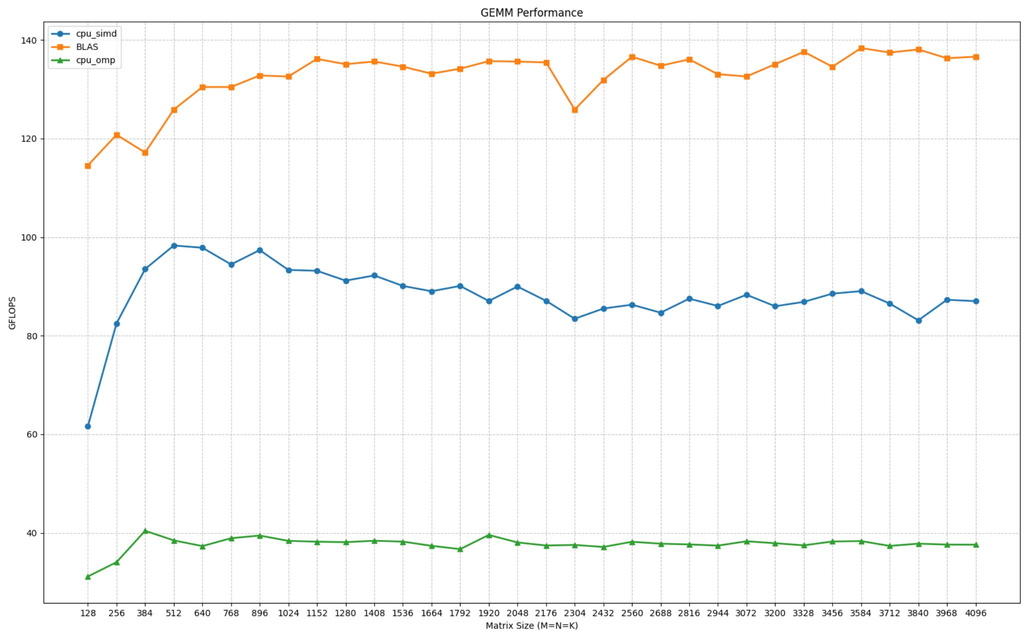
- Apple M4 pro
    - Single-thread



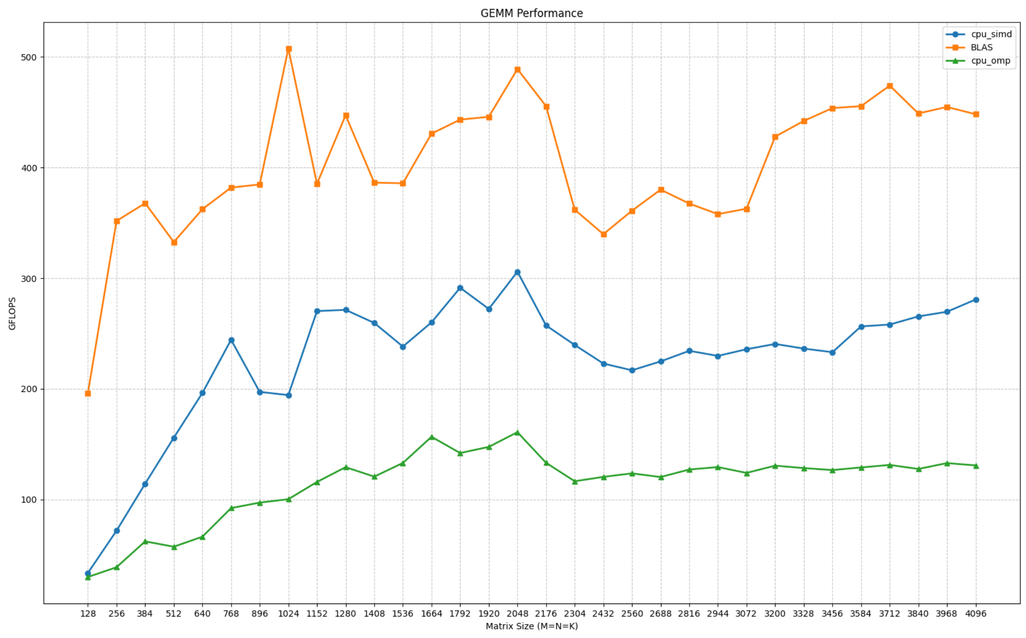
- 8-thread



- Intel i7
  - Single-thread



- 8-thread



Single-Core Step-by-Step Performance Comparison

Naive implementation

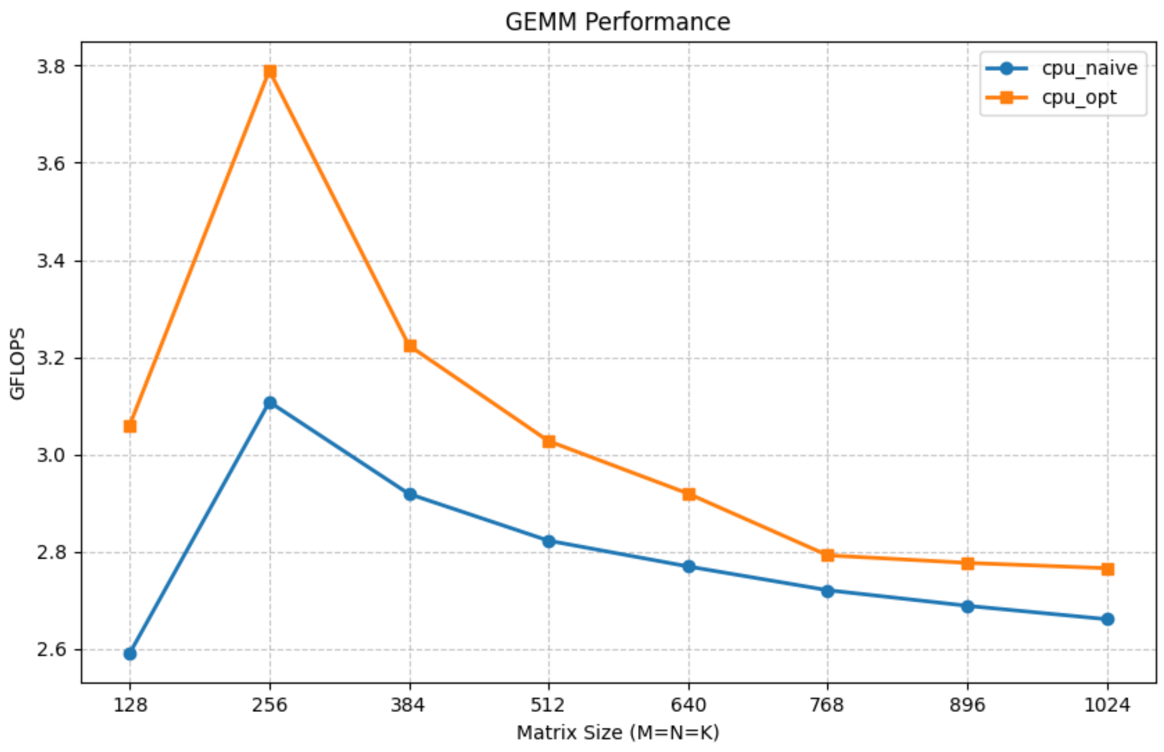
The baseline approach uses three nested loops that directly compute C[i,j] += A[i,k] * B[k,j], operating on one element at a time.

```
for (size_t i = 0; i < M; ++i)
{
    for (size_t j = 0; j < N; ++j)
    {
        for (size_t k = 0; k < K; ++k)
        {
            C.at(i, j) += alpha * A.at(i, k) * B.at(k, j);
        }
    }
}
```

Trick 1 - Register Reuse

Reduces memory access by accumulating the sum for each C[i,j] element in a register before writing back to memory. This improves cache utilization and reduces memory traffic.

```
float sum = 0.0f;
for (size_t k = 0; k < K; ++k)
{
    sum += alpha * A.at(i, k) * B.at(k, j);
}
C.at(i, j) += sum;
```



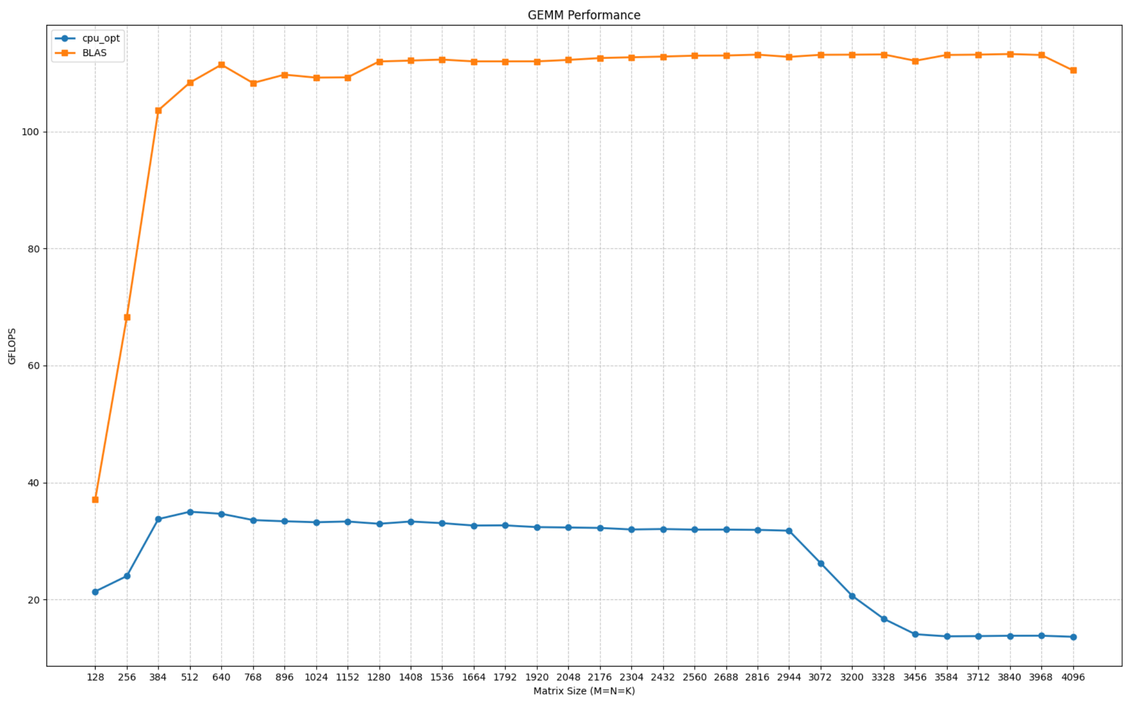
Trick 2 - Loop Unrolling & SIMD (4x4 Kernel)

Implements a 4x4 kernel using ARM Neon SIMD instructions to process multiple elements in parallel. This leverages vector registers to perform multiple operations simultaneously.

```
c0 = vdupq_n_f32(0);
c1 = vdupq_n_f32(0);
c2 = vdupq_n_f32(0);
c3 = vdupq_n_f32(0);

for (size_t k = 0; k < K; ++k)
{
   float32x4_t a = vmulq_f32(valpha, vld1q_f32(&A(i, k)));
   float32x4_t b0 = vld1q_dup_f32(b_ptr0++);
   float32x4_t b1 = vld1q_dup_f32(b_ptr1++);
   float32x4_t b2 = vld1q_dup_f32(b_ptr2++);
   float32x4_t b3 = vld1q_dup_f32(b_ptr3++);

   c0 = vfmaq_f32(c0, a, b0);
   c1 = vfmaq_f32(c1, a, b1);
   c2 = vfmaq_f32(c2, a, b2);
```

```
    c3 = vfmaq_f32(c3, a, b3);
}
vst1q_f32(&C(i, j), vaddq_f32(c0, vld1q_f32(&C(i, j))));
vst1q_f32(&C(i, j + 1), vaddq_f32(c1, vld1q_f32(&C(i, j + 1))));
vst1q_f32(&C(i, j + 2), vaddq_f32(c2, vld1q_f32(&C(i, j + 2))));
vst1q_f32(&C(i, j + 3), vaddq_f32(c3, vld1q_f32(&C(i, j + 3))));
```

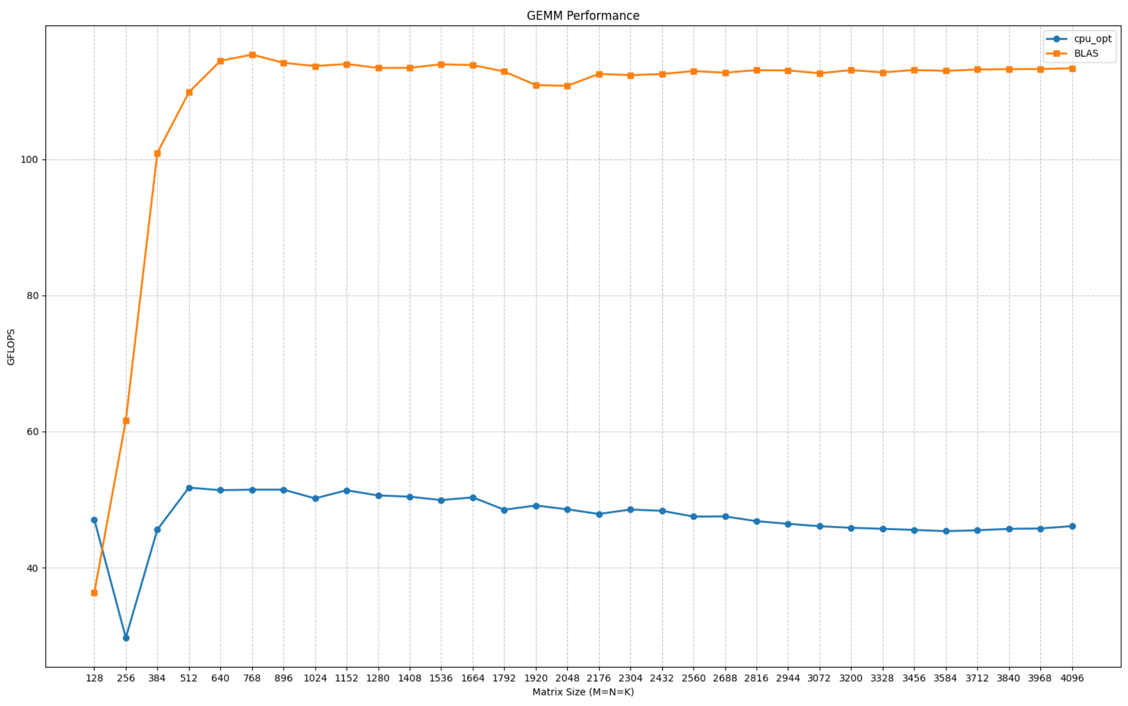
```
__m128 c0 = _mm_setzero_ps();
__m128 c1 = _mm_setzero_ps();
__m128 c2 = _mm_setzero_ps();
__m128 c3 = _mm_setzero_ps();
for (size_t k = 0; k < K; ++k)
{
    __m128 a = _mm_mul_ps(valpha, _mm_loadu_ps(&A(i, k)));
    __m128 b0 = _mm_set1_ps(B(k, j));
    __m128 b1 = _mm_set1_ps(B(k, j + 1));
    __m128 b2 = _mm_set1_ps(B(k, j + 2));
    __m128 b3 = _mm_set1_ps(B(k, j + 3));

    c0 = _mm_fmadd_ps(a, b0, c0);
    c1 = _mm_fmadd_ps(a, b1, c1);
    c2 = _mm_fmadd_ps(a, b2, c2);
    c3 = _mm_fmadd_ps(a, b3, c3);
}
_mm_storeu_ps(&C(i, j), _mm_add_ps(c0, _mm_loadu_ps(&C(i, j))));
_mm_storeu_ps(&C(i, j + 1), _mm_add_ps(c1, _mm_loadu_ps(&C(i, j + 1))));
_mm_storeu_ps(&C(i, j + 2), _mm_add_ps(c2, _mm_loadu_ps(&C(i, j + 2))));
_mm_storeu_ps(&C(i, j + 3), _mm_add_ps(c3, _mm_loadu_ps(&C(i, j + 3))));
```



Trick 3 - Cache Blocking

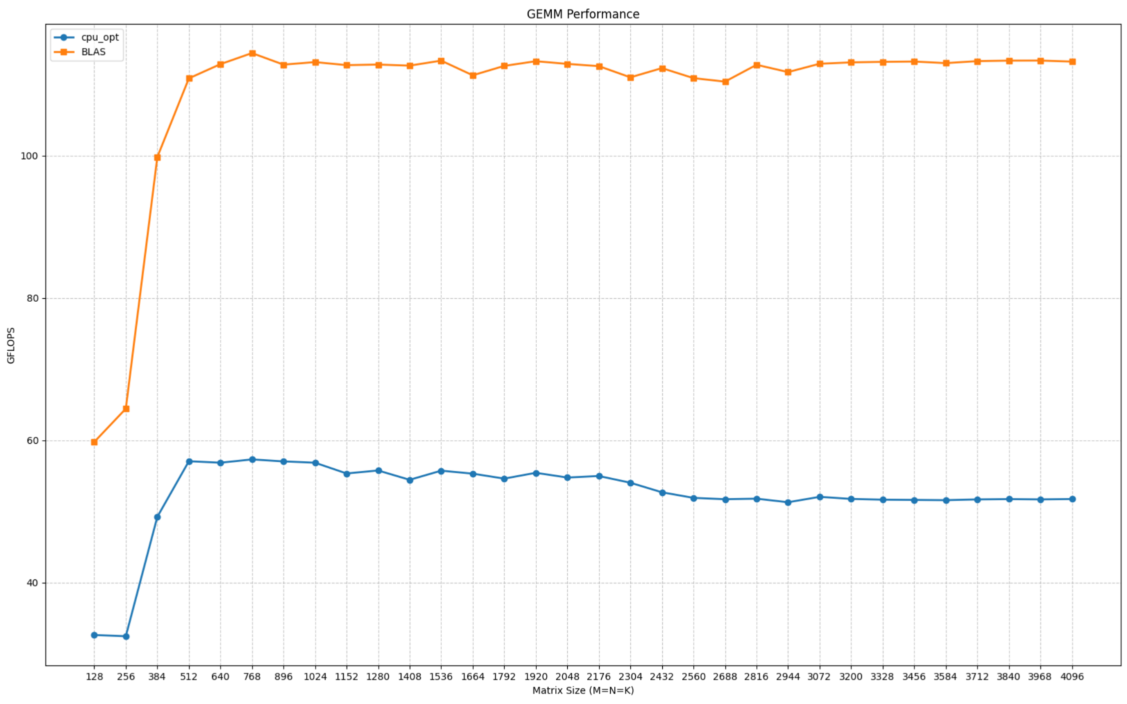
Divides the computation into smaller blocks that fit in cache, reducing cache misses by improving data locality. The code processes submatrices that fit entirely in cache before moving to the next block.

```
for (n_count = 0; n_count < N; n_count += N_BLOCKING)
{
    n_inc = (N - n_count > N_BLOCKING) ? N_BLOCKING : (N - n_count);
    for (k_count = 0; k_count < K; k_count += K_BLOCKING)
    {
        k_inc = (K - k_count > K_BLOCKING) ? K_BLOCKING : (K - k_count);
        for (m_count = 0; m_count < M; m_count += M_BLOCKING)
        {
            m_inc = (M - m_count > M_BLOCKING) ? M_BLOCKING : (M - m_count);

            macro_kernel_4x4_sgemm_neon(m_inc, n_inc, k_inc, alpha, &A.at(m_count,
k_count), A.ld(), &B.at(k_count, n_count), B.ld(), beta, &C.at(m_count, n_count), C.ld());
        }
    }
}
```



Trick 4 - Packing

Further optimizes memory access patterns by reorganizing data layout of both A and B for better cache utilization during computation.


GEMM Performance
cpu_opt
BLAS
GFLOPS
40
60
80
100
Matrix Size (M=N=K)
128
256
384
512
640
768
896
1024
1152
1280
1408
1536
1664
1792
1920
2048
2176
2304
2432
2560
2688
2816
2944
3072
3200
3328
3456
3584
3712
3840
3968
4096