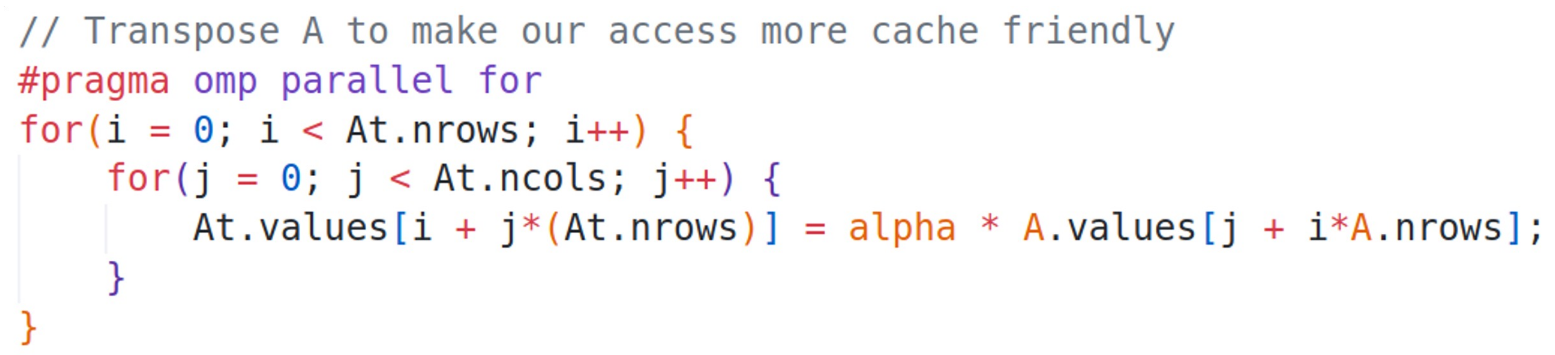
CSE 6230 - Code Walkthrough
Sooraj Karthik

Key Ideas

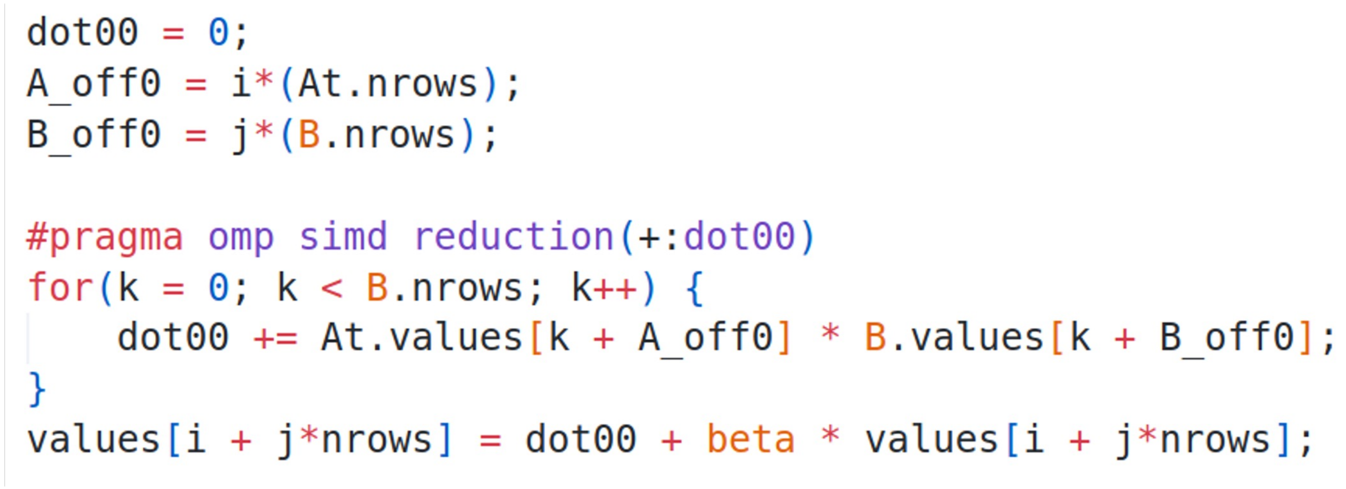
1. SIMD Instructions
2. Instruction Level Parallelism
3. Minimize memory operations
    a. Tiling
    b. Caching (CPU)
    c. Coalescing (GPU)
    d. Vectorized Loads (GPU)
    e. Load data once, use it multiple times

CPU - Transpose for Cache



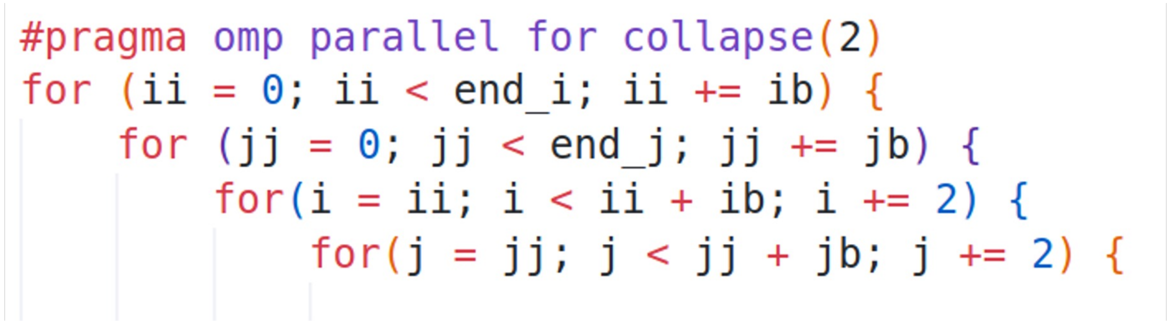
```
// Transpose A to make our access more cache friendly
#pragma omp parallel for
for(i = 0; i < At.nrows; i++) {
    for(j = 0; j < At.ncols; j++) {
        At.values[i + j*(At.nrows)] = alpha * A.values[j + i*A.nrows];
    }
}
```

CPU - Transpose for Cache



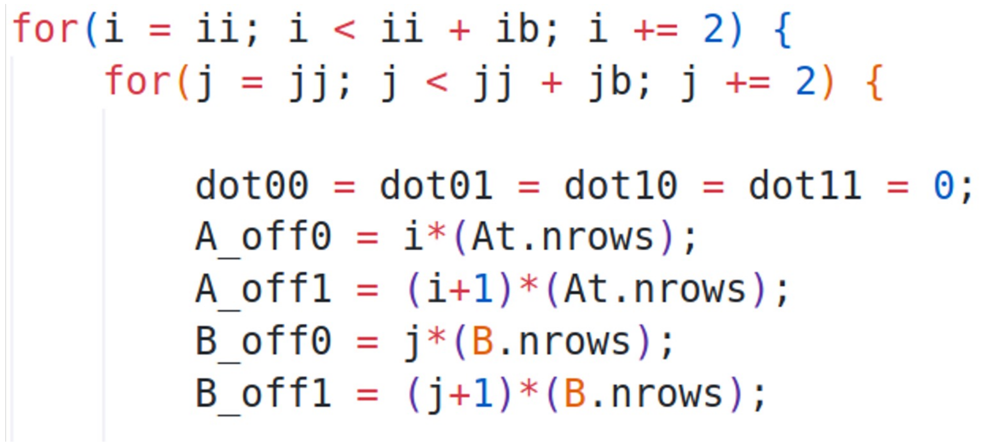
```
dot00 = 0;
A_off0 = i*(At.nrows);
B_off0 = j*(B.nrows);

#pragma omp simd reduction(+:dot00)
for(k = 0; k < B.nrows; k++) {
    dot00 += At.values[k + A_off0] * B.values[k + B_off0];
}
values[i + j*nrows] = dot00 + beta * values[i + j*nrows];
```

CPU - Tiling



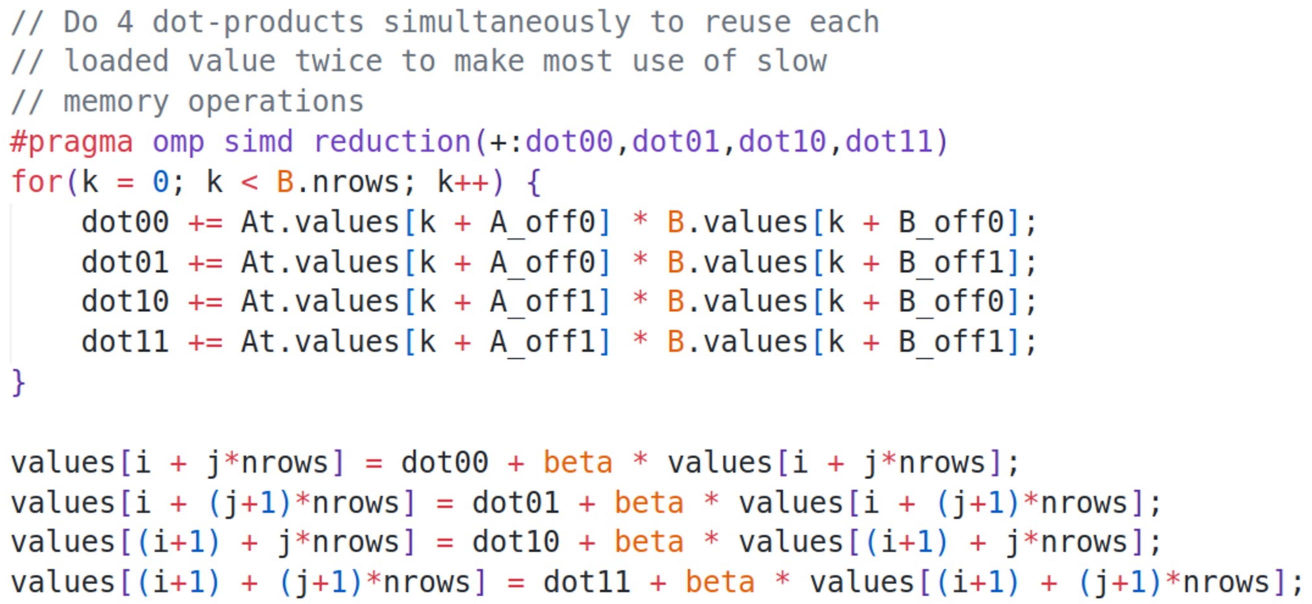
```
#pragma omp parallel for collapse(2)
for (ii = 0; ii < end_i; ii += ib) {
    for (jj = 0; jj < end_j; jj += jb) {
        for(i = ii; i < ii + ib; i += 2) {
            for(j = jj; j < jj + jb; j += 2) {
```

CPU - Reusing Loaded Memory



```
for(i = ii; i < ii + ib; i += 2) {
    for(j = jj; j < jj + jb; j += 2) {

        dot00 = dot01 = dot10 = dot11 = 0;
        A_off0 = i*(At.nrows);
        A_off1 = (i+1)*(At.nrows);
        B_off0 = j*(B.nrows);
        B_off1 = (j+1)*(B.nrows);
```

CPU - Reusing Loaded Memory



```
// Do 4 dot-products simultaneously to reuse each
// loaded value twice to make most use of slow
// memory operations
#pragma omp simd reduction(+:dot00,dot01,dot10,dot11)
for(k = 0; k < B.nrows; k++) {
    dot00 += At.values[k + A_off0] * B.values[k + B_off0];
    dot01 += At.values[k + A_off0] * B.values[k + B_off1];
    dot10 += At.values[k + A_off1] * B.values[k + B_off0];
    dot11 += At.values[k + A_off1] * B.values[k + B_off1];
}

values[i + j*nrows] = dot00 + beta * values[i + j*nrows];
values[i + (j+1)*nrows] = dot01 + beta * values[i + (j+1)*nrows];
values[(i+1) + j*nrows] = dot10 + beta * values[(i+1) + j*nrows];
values[(i+1) + (j+1)*nrows] = dot11 + beta * values[(i+1) + (j+1)*nrows];
```

CPU - Notes About Tiling

1. Why didn’t you tile over k?
    a. Too many boundary cases thanks to calculating a 2x2 dot product at once
    b. The branch statements added (like 4 if statements) actually slowed down the more than the tiling helped
2. Why not tile over the entire matrix?
    a. Notice we didn’t tile over the entire matrix if it's not a multiple of tile sizes
    b. Again needed to add a bunch of if statements and I didn’t want to write that much code
    c. Solution: add non-tiled code at the end to calculate the remainder of the matrix

CPU - Other Details (MPI)

1. Same implementation is used for MPI (just removed omp parallel for directives)
2. Support for 3D domain decomposition (x, y, z)
3. Scatter blocks of A and B
4. Only send blocks of C to the processors with z=0
    a. Save some communication time
    b. Only need add β^*C once

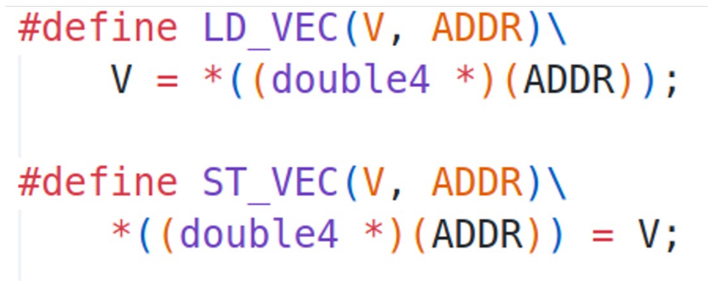
GPU - Optimizations Overview

1. Each tread calculates an 8x8 block of C
2. Use warp tiling
   a. Just a smart way of choosing which parts of the matrices each thread operates on
3. Use vectorized loads, stores, and computations (SIMD)
4. Prefetch memory from GMEM to SHMEM
5. Double buffered SHMEM to remove a synchronization inside dot-product loop

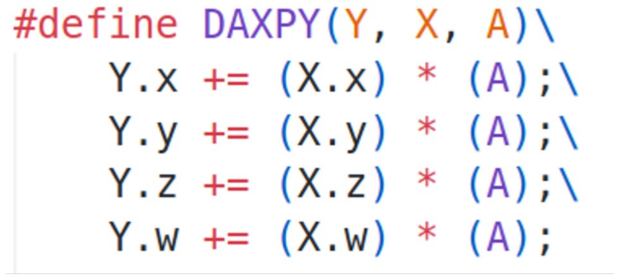
GPU - Algorithm Overview

1. Figure out which parts of matrices to operate on
2. Do initial fetch from global memory and write to shared memory
3. Load first vectors to operate on from shared memory
4. Tiled iteration over k with tile size = 8
    a. Prefetch data from global memory for next tile
    b. Computation for current tile
        i. Prefetch next vectors to operate on from shared memory
        ii. Do math
    c. Write prefetched data to shared memory (hopefully it will be done by the time we finish doing computation)
    d. Load next vectors to operate on from shared memory
5. Load C from gmem (no need for shmem since value is accessed by only one thread)
6. Calculate final values and store back into C

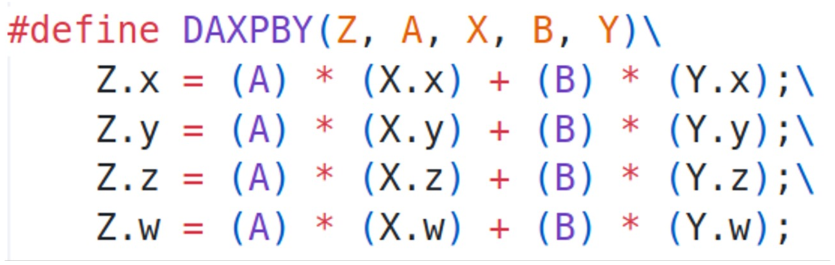
GPU - Vectorized Operations



```
#define LD_VEC(V, ADDR)\
    V = *((double4 *)(ADDR));

#define ST_VEC(V, ADDR)\
    *((double4 *)(ADDR)) = V;
```



```
#define DAXPY(Y, X, A)\
    Y.x += (X.x) * (A);\
    Y.y += (X.y) * (A);\
    Y.z += (X.z) * (A);\
    Y.w += (X.w) * (A);
```



```
#define DAXPBY(Z, A, X, B, Y)\
    Z.x = (A) * (X.x) + (B) * (Y.x);\
    Z.y = (A) * (X.y) + (B) * (Y.y);\
    Z.z = (A) * (X.z) + (B) * (Y.z);\
    Z.w = (A) * (X.w) + (B) * (Y.w);
```

GPU - Double Buffer Shared Memory

```
// 2x1024 so we can double buffer shmem so threads fetch
// can new data from A and B to shared before other
// threads are done with the old data. This reduces one
// sync from the loop.
__shared__ double shmem_A[2][1024];
__shared__ double shmem_B[2][1024];

// Pointers to the current shared memory buffer
double *ptr_shmem_A = (double*) shmem_A;
double *ptr_shmem_B = (double*) shmem_B;
```

GPU - Prefetching from Global Memory

```
for (int k_tile = 0; k_tile < K_tile_iters; k_tile++){

    // Shift position in A and B. Move A over 8 columns
    // and move B down 8 rows
    int inc = (k_tile + 1) % K_tile_iters;
    ptr_A = A + inc * M8;
    ptr_B = B + inc * 8;

    // Prefetch data from gmem while we do computations
    LD_VEC(pref_Av, &ptr_A(row_a,col_a))
    LD_VEC(pref_Bv, &ptr_B(row_b,col_b))

    // Calculate dot product for loaded vectors
```

GPU - Prefetch From Shared Memory

```
int next_k = (k + 1) & 7;
int vec_idx = k & 1;

LD_VEC(Av1[(k + 1) & 1], &ptr_shmem_A(row_c,   next_k))
LD_VEC(Av2[(k + 1) & 1], &ptr_shmem_A(row_c+4, next_k))
LD_VEC(Bv1[(k + 1) & 1], &ptr_shmem_B(col_c,   next_k))
LD_VEC(Bv2[(k + 1) & 1], &ptr_shmem_B(col_c+4, next_k))
```

GPU - Calculate Dot Product

```
DAXPY(Cres[0],  Av1[vec_idx], Bv1[vec_idx].x)
DAXPY(Cres[1],  Av2[vec_idx], Bv1[vec_idx].x)
DAXPY(Cres[2],  Av1[vec_idx], Bv1[vec_idx].y)
DAXPY(Cres[3],  Av2[vec_idx], Bv1[vec_idx].y)
DAXPY(Cres[4],  Av1[vec_idx], Bv1[vec_idx].z)
DAXPY(Cres[5],  Av2[vec_idx], Bv1[vec_idx].z)
DAXPY(Cres[6],  Av1[vec_idx], Bv1[vec_idx].w)
DAXPY(Cres[7],  Av2[vec_idx], Bv1[vec_idx].w)

DAXPY(Cres[8],  Av1[vec_idx], Bv2[vec_idx].x)
DAXPY(Cres[9],  Av2[vec_idx], Bv2[vec_idx].x)
DAXPY(Cres[10], Av1[vec_idx], Bv2[vec_idx].y)
DAXPY(Cres[11], Av2[vec_idx], Bv2[vec_idx].y)
DAXPY(Cres[12], Av1[vec_idx], Bv2[vec_idx].z)
DAXPY(Cres[13], Av2[vec_idx], Bv2[vec_idx].z)
DAXPY(Cres[14], Av1[vec_idx], Bv2[vec_idx].w)
DAXPY(Cres[15], Av2[vec_idx], Bv2[vec_idx].w)
```

GPU - Write Prefetch to Shared Memory

```
// Swap back and front shmem buffers
int offset = ((k_tile + 1) & 1) << 10;
ptr_shmem_A = (double*)shmem_A + offset;
ptr_shmem_B = (double*)shmem_B + offset;

// Write data prefetched from gmem
((double4 *)ptr_shmem_A)[tx] = pref_Av;

// Stored like this to coalese shmem access
ptr_shmem_B(col_b, row_b)   = pref_Bv.x;
ptr_shmem_B(col_b, row_b+1) = pref_Bv.y;
ptr_shmem_B(col_b, row_b+2) = pref_Bv.z;
ptr_shmem_B(col_b, row_b+3) = pref_Bv.w;

__syncthreads();
```

GPU - Optimization Performance Gains

Large Impact:

1. Transposing B when packing into shmem (2x performance)
2. 8x8 computation per thread (9-10x performance)
    a. Making better use of the 64k registers per thread block
3. Vectorized computation (SIMD) + Warp Tiling (+60% performance)
    a. Access to same shmem address within a warp can be coalesced
4. Double Buffering (+10-15% performance)

Minimal Impact

1. Prefetching (So much extra code for barely 3% improvement 😞)
2. Vectorized load/store (~2% improvement)
    a. Should have used double2 instead since max memory transaction size is 128 bits

GPU - Other Details (Padding, MPI)

1. Need to pad input matrices to a multiple of 128 before calling kernel
    a. Slower to check boundaries on GPU rather than just padding on CPU side thanks to openmp
2. For MPI+Cuda DSRGEMM
    a. Copy-paste code change dot product operations
        i. Accumulating operator becomes min
            1. min is like 4x slower than addition but oh well :(
        ii. Multiplication becomes addition
    b. Repurpose domain decomposition code from MPI DGEMM
        i. Made scattering more efficient since we measured communication time

Questions? (I know that was a lot)

Optimizing DGEMM Implementations: OpenMP, MPI, and CUDA

Fan Qu

CONTENTS

Introduction

- DGEMM stands for Double-precision General Matrix Multiply, a core operation in linear algebra
- $C = \alpha \cdot AB + \beta \cdot C$

Introduction

- DGEMM is both computationally intensive and memory intensive.
- Floating-point arithmetic is fast, access to memory is slow.

Optimizing general matrix multiplication is mostly about optimizing memory accesses.

Single CPU Core

- When M=N=K=4096,
- Our DGEMM: 22.41 Gflops
- OpenBLAS: 67.57 Gflops

We are still far from optimal performance!
About 33% of OpenBLAS

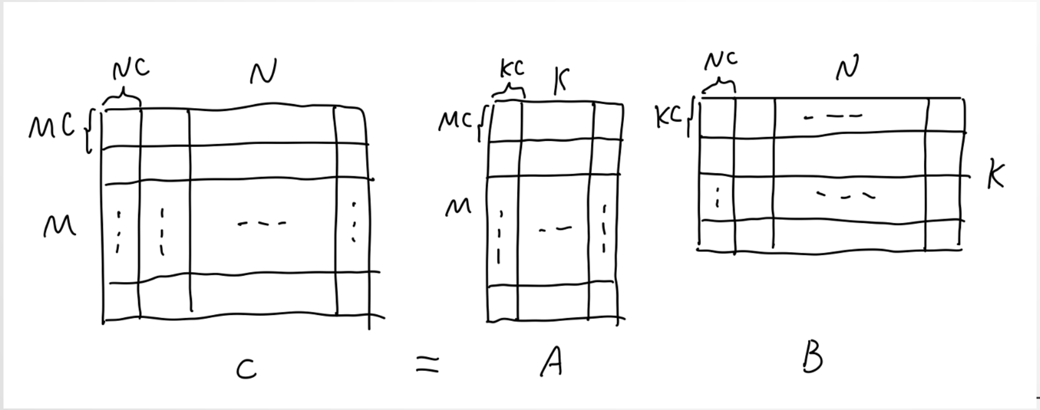
Single CPU Core

- Blocked Data
- Micro Kernel 4x4
- Packing Data
- SIMD

Blocked Data

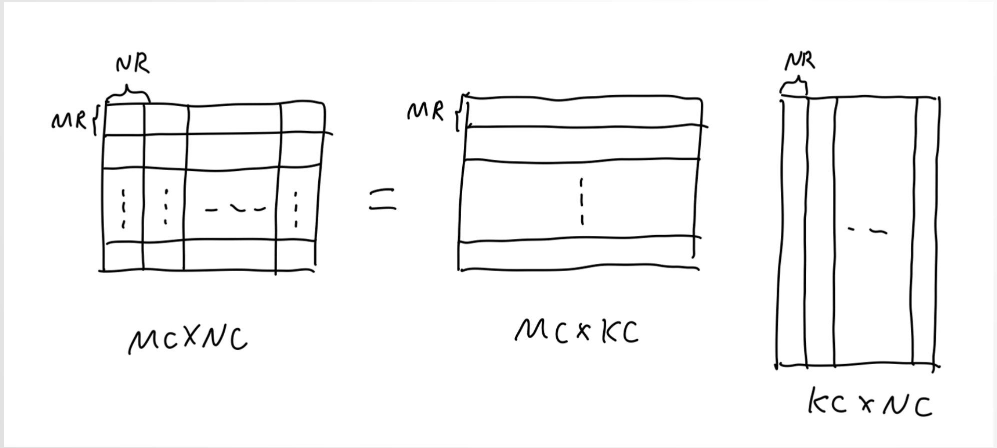
Use blocked matrices to increase cache hit rate.

However, we didn't spend a lot of time to investigate the best MC,NC,KC parameters for the cache with the CPU (Intel(R) Xeon(R) Gold 6226 CPU) we tested.



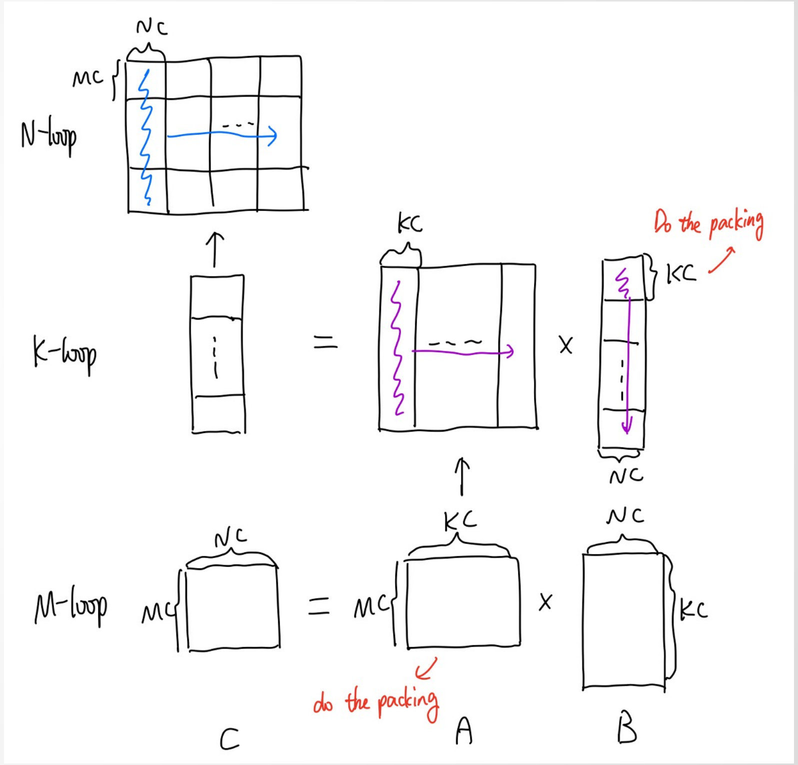
Micro Kernel

- Micro Kernel can be viewed as secondary level of blocking.
- We use 4x4 micro kernel, which is suitable to add SIMD instructions in the following optimization steps.



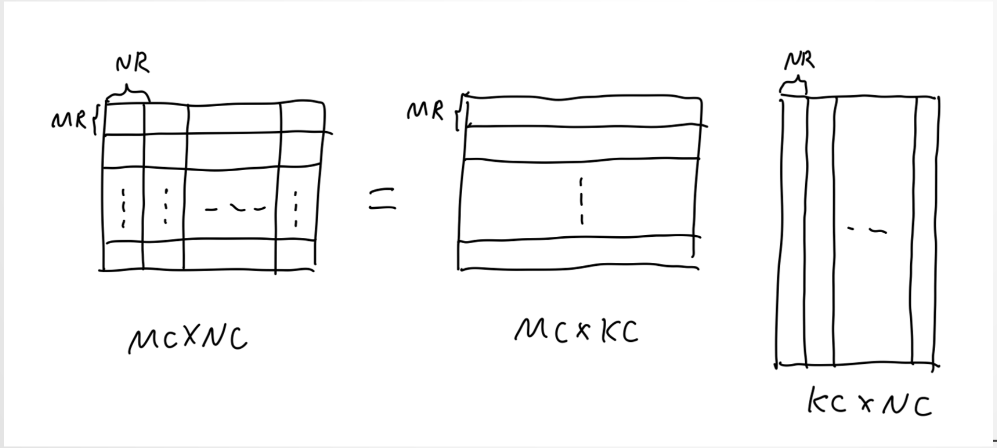
Packing Data

- Regardless of whether we choose a matrix representation with row- or column-major order, we always need access to non-contiguous memory.
- We can first pack the discontinuous elements in the matrix into continuous memory to improve the efficiency of subsequent multiple accesses.

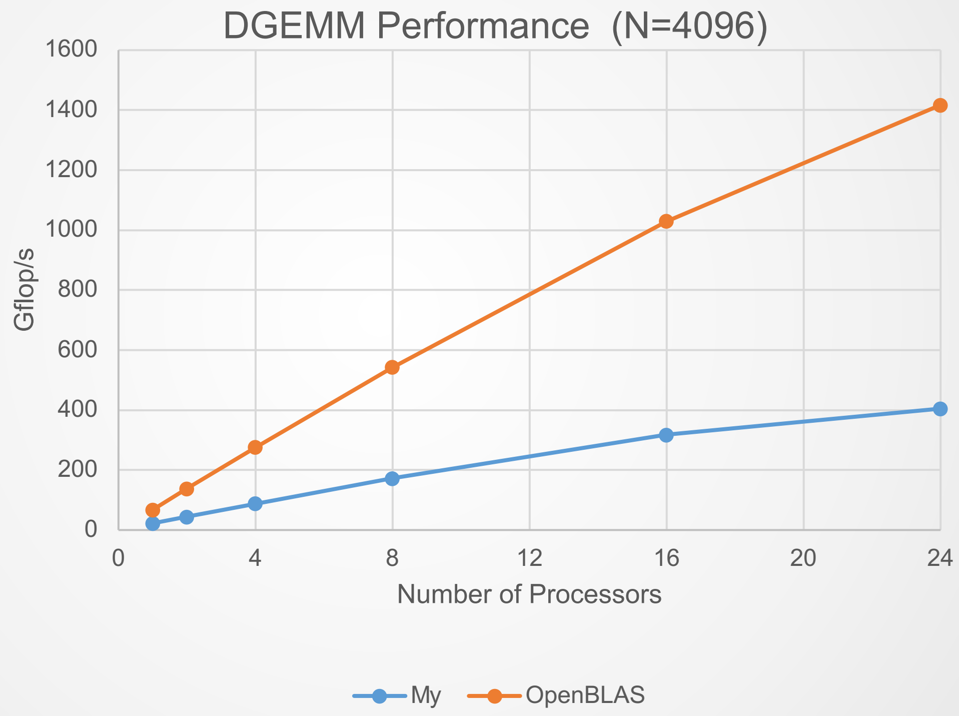


SIMD

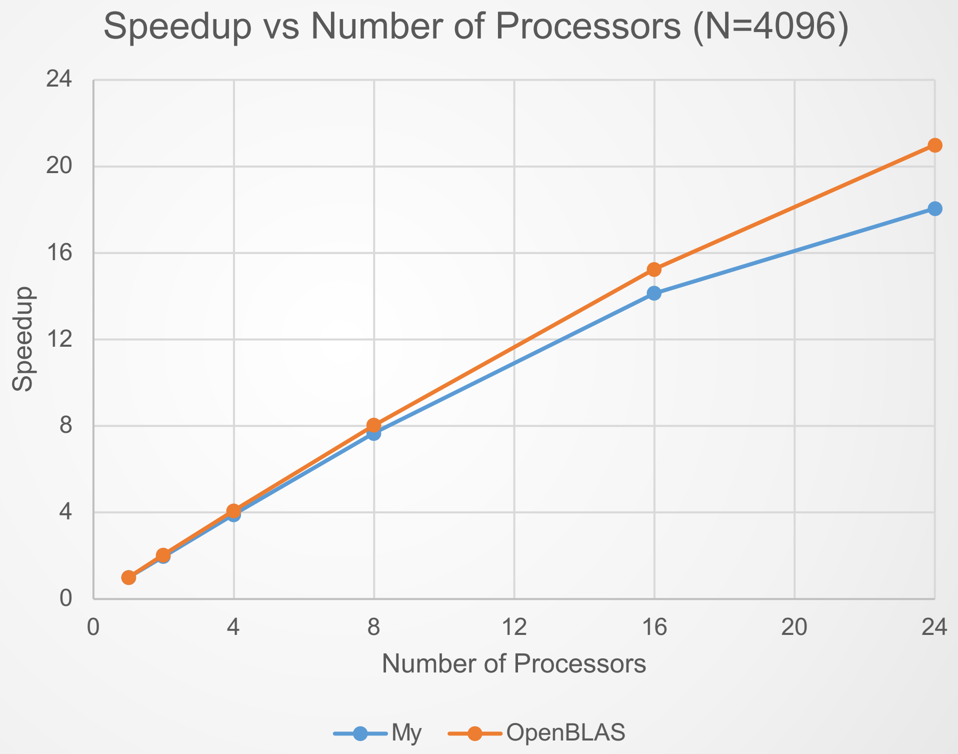
- Use AVX2 SIMD instructions to accelerate the computation
- We have not fully optimized the instruction generation and scheduling. In particular, we are currently using a disproportionate share of LOAD instructions.



OpenMP

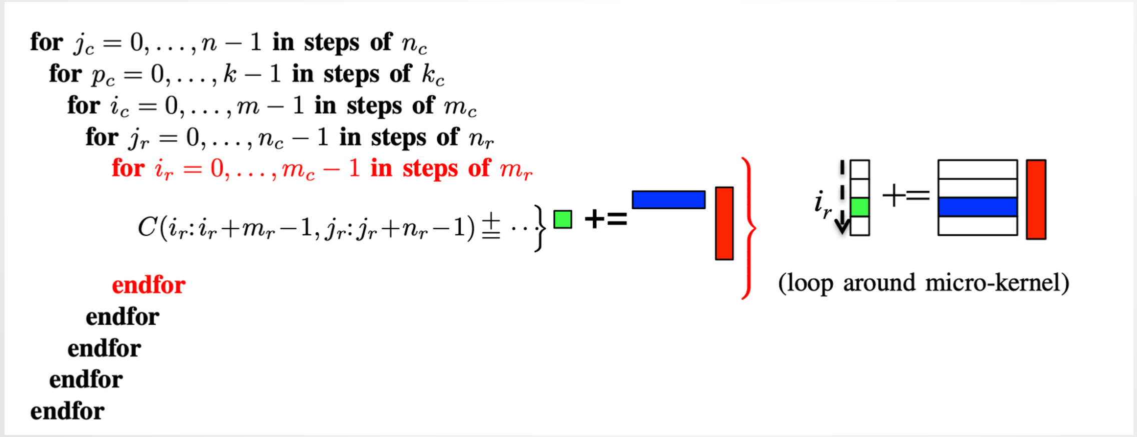

DGEMM Performance (N=4096)
Gflop/s
1600
1400
1200
1000
800
600
400
200
0
0
4
8
12
16
20
24
Number of Processors
My
OpenBLAS

OpenMP


Speedup vs Number of Processors (N=4096)
Speedup
Number of Processors
0
4
8
12
16
20
24
My
OpenBLAS

OpenMP

- We can choose to parallelize the processing at different levels of the loop.
- In the end we choose to parallelize at the k_c level. (which is m_c in our code)



```
for $j_c = 0, \ldots, n-1$ in steps of $n_c$
  for $p_c = 0, \ldots, k-1$ in steps of $k_c$
    for $i_c = 0, \ldots, m-1$ in steps of $m_c$
      for $j_r = 0, \ldots, n_c-1$ in steps of $n_r$
        for $i_r = 0, \ldots, m_c-1$ in steps of $m_r$
          $C(i_r{:}i_r+m_r-1, j_r{:}j_r+n_r-1) \mathrel{+}= \cdots$
        endfor
      endfor
    endfor
  endfor
endfor
```

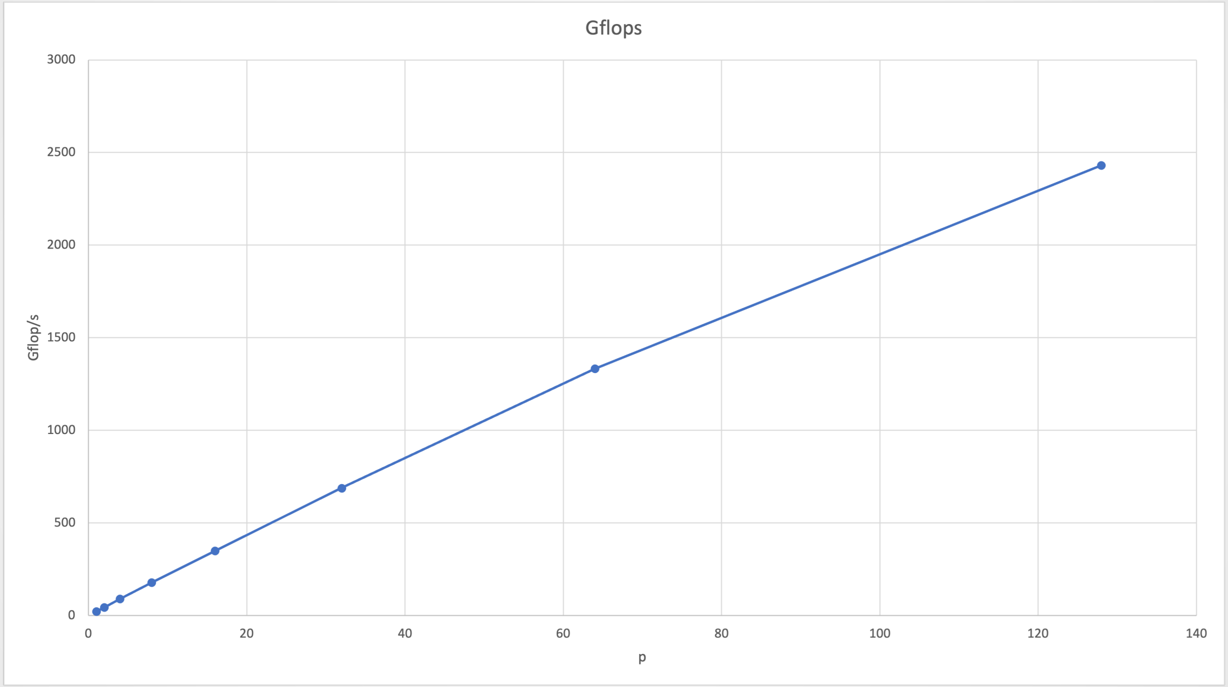
Figure from [1]

OpenMP

- We need to let every code has its private packing data
- The number of iterations of the parallel loop is M/MC. In order to ensure a higher degree of parallelism, MC cannot be too large; in order to maintain the effect of block, MC cannot be too small.

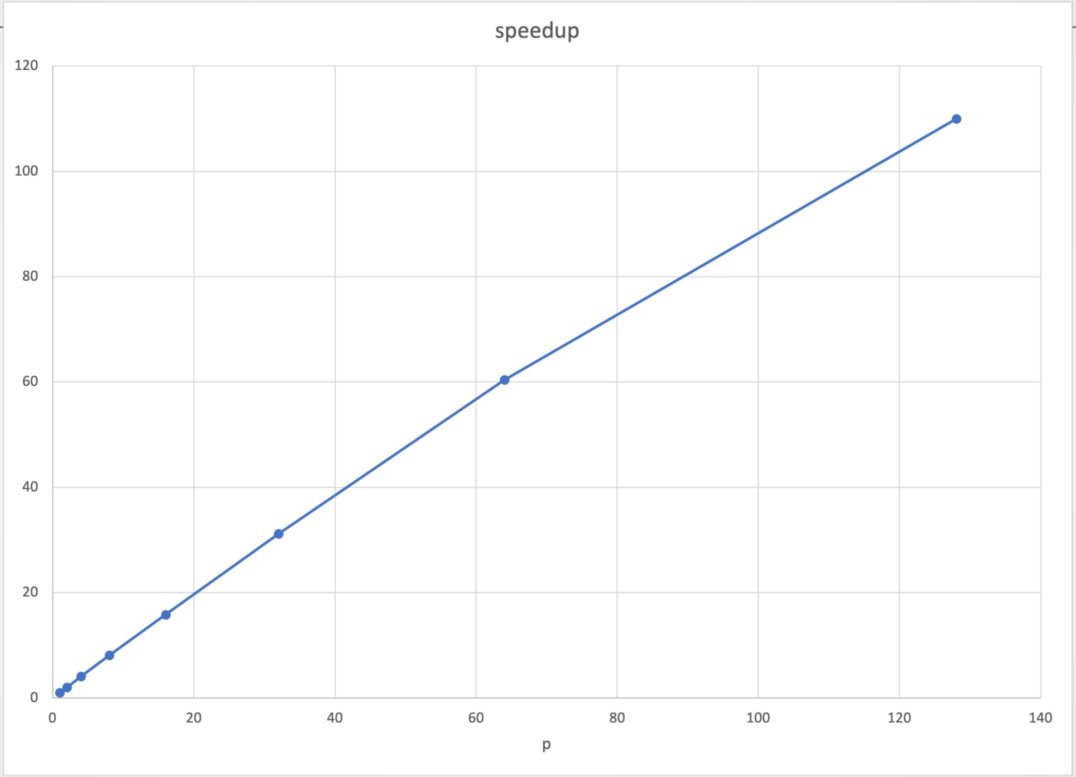
We choose $\mathrm{m_c} = \min(4\left\lceil\frac{M}{4\cdot nthreads}\right\rceil, 64)$

- MPI



Performance, M = 36864, N=18432, K=4608

MPI



Speedup, M = 36864, N=18432, K=4608

MPI

- We didn’t use the 3-d topology but use the simple split similar to the OpenMP version.
- We parallelize at the m_c level.
- Every processor holds (M/p, K) local_A, (K, N) local_B, (M/p, N) local_C

MPI

Advantages:

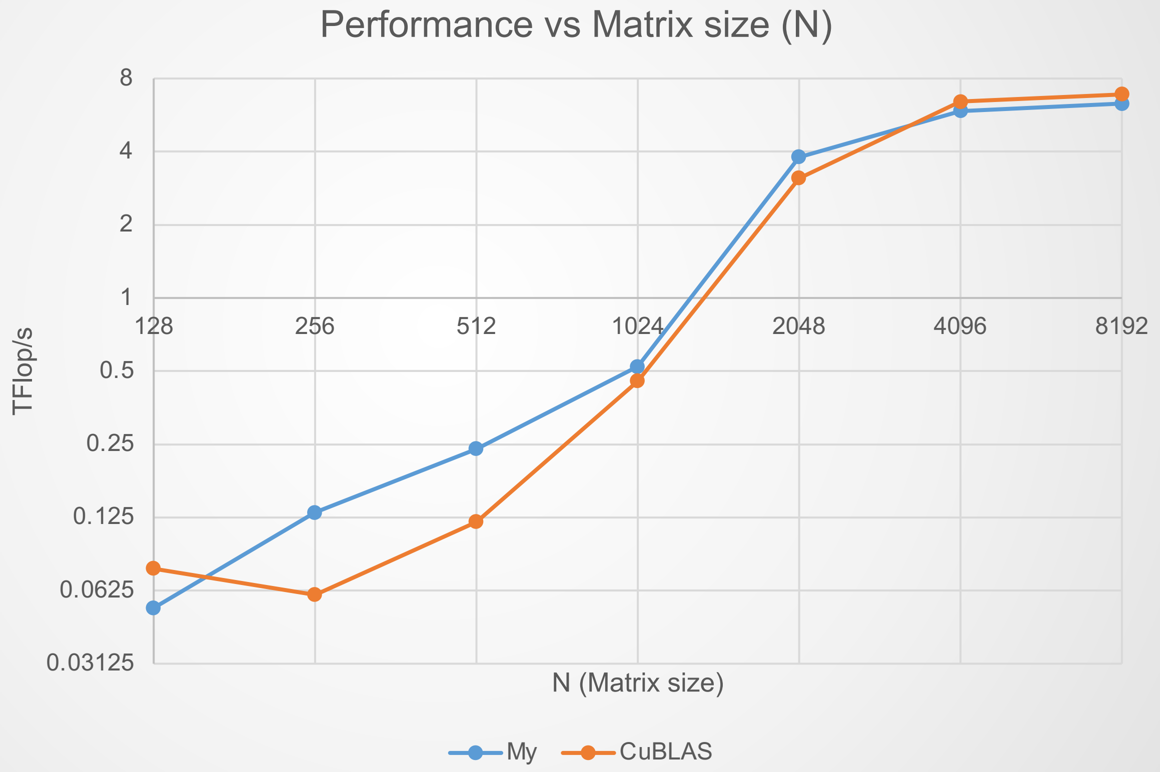
- Easy to implement
- No communication during computational kernel stage

Disadvantages:

- Cannot scale well when $M < p$.
- Memory usage is not optimal as we keep a full copy of B for every processor

CUDA

When M=N=K=4096, our performance is 5.89 Tflop/s, cuBLAS is 6.425 Tflop/s



CUDA

- Learn a lot from online resources [2, 3, 4, 5].

- Data Blocking
- Micro Kernel
- Vectorized Load
- Warp-level parallelism

CUDA

Vectorized Load

We can use double4 to utilize efficient load instruction

CUDA

Warp-level parallelism

- Different warps can execute in parallel on different warp schedulers, and concurrently on the same warp scheduler.
- Memory accesses to the same memory address in shared memory within the same warp can be coalesced

References

1. Smith, Tyler M., et al. "Anatomy of high-performance many-threaded matrix multiplication." *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014.
2. https://github.com/NVIDIA/cutlass/blob/master/media/docs/efficient_gemm.md
3. https://siboehm.com/articles/22/CUDA-MMM
4. https://github.com/yzhaiustc/Optimizing-SGEMM-on-NVIDIA-Turing-GPUs
5. Huang, Jianyu, Chenhan D. Yu, and Robert A. van de Geijn. "Implementing Strassen's algorithm with CUTLASS on NVIDIA Volta GPUs." *arXiv preprint arXiv:1808.07984* (2018).

Thanks

Assignment 3 Presentation

Peidi Song

Host <-> Device

- device_allocate_init
  - cuErrChk(cudaMalloc()) and cuErrChk(cudaMemcpy()) for all buffers
- device_free
  - cuErrChk(cudaFree()) for all buffers
- device_to_host
  - cuErrChk(cudaMemcpy()) for vals

Notice 1: No Tensor Core

- “Tensor Cores support double-precision floating point operations on devices with compute capability 8.0 and higher.”
- Compute capability for V100 is 7.0
- No Tensor Core can be used in this assignment

- https://docs.nvidia.com/cuda/archive/11.1.1/cuda-c-programming-guide/index.html#wmma-double
- Cuda check code: https://gist.github.com/f0k/0d6431e3faa60bffc788f8b4daa029b1

Kernel Code Skeleton

- Loop splitting: SSR~~S~~RS (S: Spatial; R: Reduce)

```
for {i0, j0} // bind to block
  for {i1, j1} // bind to thread/warp
    for {k0}
      load A, B: global -> shared
      for {k1}
        for {i2, j2} // bind to warp and sequential
          load A, B: shared -> local
          compute C: local
    store C: local -> global
```

kernel_params_init

- Define constants `tile_m`, `tile_n`, `tile_k`
  - To decide shared memory size for A and B
- Define constants `warp_size`, `bdx`, `bdy`
  - `warp_size` is always 32
  - `bdx` and `bdy` decide block size

```
block_size = dim3(bdx * warp_size, bdy);
grid_size = dim3(
    (C.nrows+tile_m*bdx-1)/(tile_m*bdx),
    (C.ncols+tile_n*bdy-1)/(tile_n*bdy)
);
shmem_size = (bdx*tile_m*tile_k + bdy*tile_n*tile_k) * sizeof(double);
```

Shared and Local Memory

- Define constants `warp_size_x*warp_size_y=warp_size`

```
// shared memory
extern __shared__ double shared_m[];
double* const a_shared = shared_m;
double* const b_shared = a_shared + bdx/warp_size*tile_m*tile_k;

// Local
double a_frag[tile_m/warp_size_x];
double b_frag;
double c_frag[tile_m*tile_n/warp_size];
```

Global -> Shared: Strided Load

- Load with stride `bd=bdx*bdy*warp_size`

```
for (uint32_t k0 = 0; k0 < K; k0 += tile_k){
    __syncthreads();
    for (uint32_t i = 0; i < bdx*tile_m*tile_k/bd; i++)
        a_shared[i*bd+tid] = A[(i*bd+tid)%(bdx*tile_m) + (i*bd+tid)/(bdx*tile_m)*M + ...];
    for (uint32_t i = 0; i < tile_k*bdy*tile_n/bd; i++)
        b_shared[i*bd+tid] = B[(i*bd+tid)%tile_k+(i*bd+tid)/tile_k*K + ...];
    __syncthreads();
    ... // Local load and compute
}
```

- Values of "…" can be calculated outside corresponding loop, i.e. ~constants

Local Load and Compute

- Smaller size GEMM

```
for (uint32_t k0 = 0; k0 < K; k0 += tile_k){
    __syncthreads();
    ... // Load from global to shared
    __syncthreads();
    for (uint32_t k = 0; k < tile_k; k++){
        for (uint32_t i = 0; i < tile_m/warp_size_x; i++)
            a_frag[i] = a_shared[i + k*bdx*tile_m + ...];
        for (uint32_t j = 0; j < tile_n/warp_size_y; j++){
            b_frag = b_shared[k + j*tile_k + ...];
            for (uint32_t i = 0; i < tile_m/warp_size_x; i++)
                    c_frag[i + j*tile_m/warp_size_x] += a_frag[i] * b_frag;
}}}
```

Store C back to Global

```
for (uint32_t j = 0; j < tile_n/warp_size_y; j++)
    for (uint32_t i = 0; i < tile_m/warp_size_x; i++)
        C[i + j*M + ...] = C[i + j*M + ...] * beta
        + c_frag[i + j*tile_m/warp_size_x] * alpha;
```

Notice 2: Indivision

- Tuned parameters for 4K*4K*4K size
  - `tile_m = tile_n = tile_k = 32`
  - `bdx = bdy = 2`
  - `warp_size_x = 16, warp_size_y = 2`
- This only fits matrix size larger than 64*64*64
- Solution: if statement? Slow! (about 40%)

```
const uint32_t idx = (bx*bdx*warp_size+tx) / warp_size;
const uint32_t idy = by*bdy+ty;
for (k0){
    __syncthreads(); ...; __syncthreads();
    if (idx*tile_m < M && idy*tile_n < N){...}
}
```

Notice 2: Indivision con’d

- “Big-Small” Algorithm (Almost 75%)
  - Called by lots of my high school classmates
  - Decision tree algorithm

```
if (M >= 64 && N >= 64 && K >= 64){
    // Codes without if-control
}
else{
    // Codes with if-control
}
```

- Same for kernel_params_init
- I also modified tuned parameters in codes with if-control for simplicity

Notice 3: Stride Load Optimization

- `bdx*tile_m` is a factor of `bd` in “big” part

`a_shared[i*bd+tid] = A[(~~i*bd+~~tid)%(bdx*tile_m) + (i*bd+tid)/(bdx*tile_m)*M + ...];`

- This eliminates one “%” operator per global load
- Same for `b_shared`
- Over 75% after optimization

Notice 4: No Double Buffer

- Double buffer eliminates one syncthreads per iteration

```
for (k0){
    ... // use buffer k0%2 for loading
    __syncthreads();
    ... // use buffer k0%2 for compute
}
```

- Don’t use double buffer, it speeds down
- Mainly because shared memory is already maximum, by using double buffer we have to half the parameters

Thanks

Accelerated, Distributed Memory Semi-Ring DGEMM

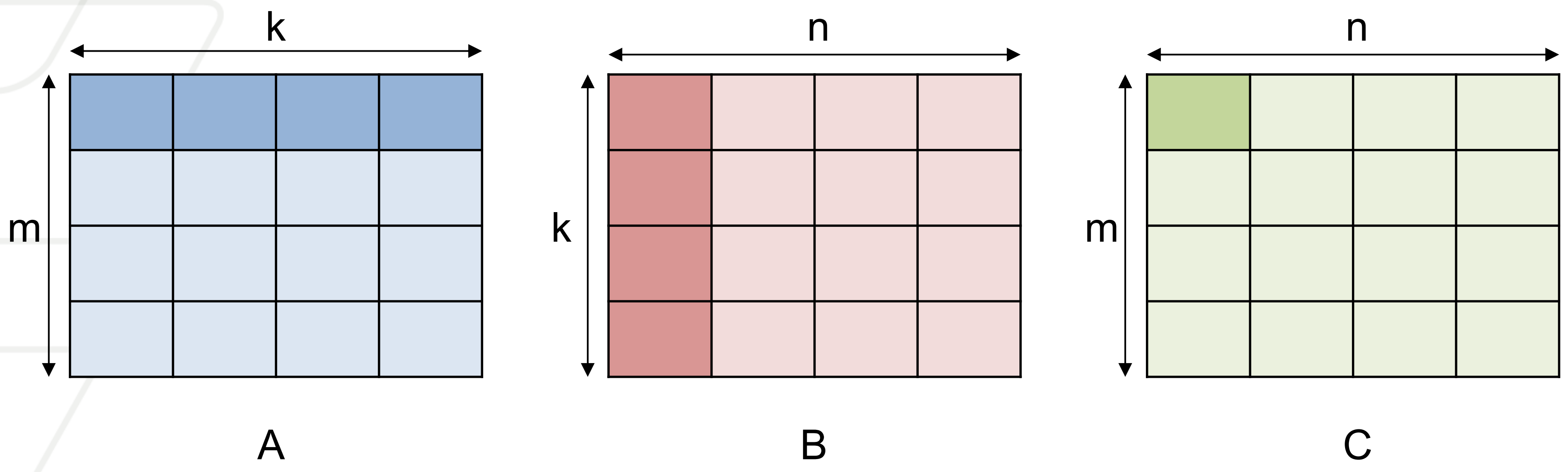
CSE 6230: High Performance Parallel Computing

Assignment 4

Georgia Tech

Presented by: Parima Devanshu Mehta

Problem Description



$$C_{ij} = \min_{1 \le q \le k} A_{iq} + B_{qj} \quad \forall \quad 1 \le i \le m, 1 \le j \le n$$

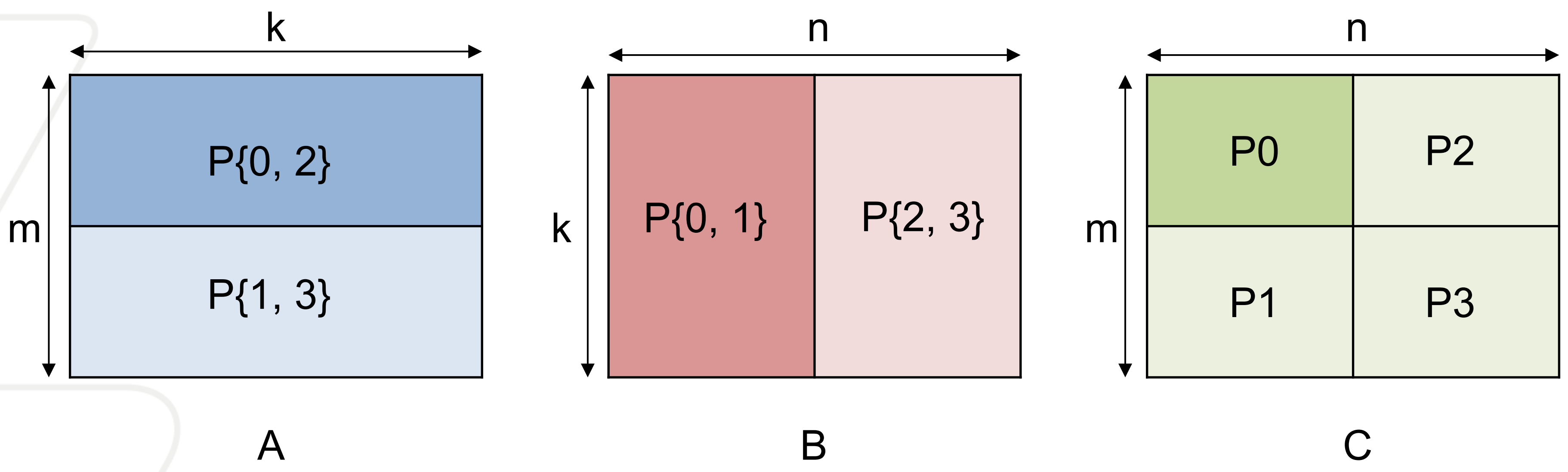
where, $A \in \mathbb{R}^{m \times k}$ $B \in \mathbb{R}^{k \times n}$ $C \in \mathbb{R}^{m \times n}$

Implement distributed-memory semi-ring DGEMM with:

- Multi-core parallelism per process using openMP
- GPU acceleration per process using CUDA

Domain Decomposition

- Adopted distribution strategy is 2D
- Each MPI process receives blocks of input A and B necessary to compute a block of output C



MPI DSRGEMM

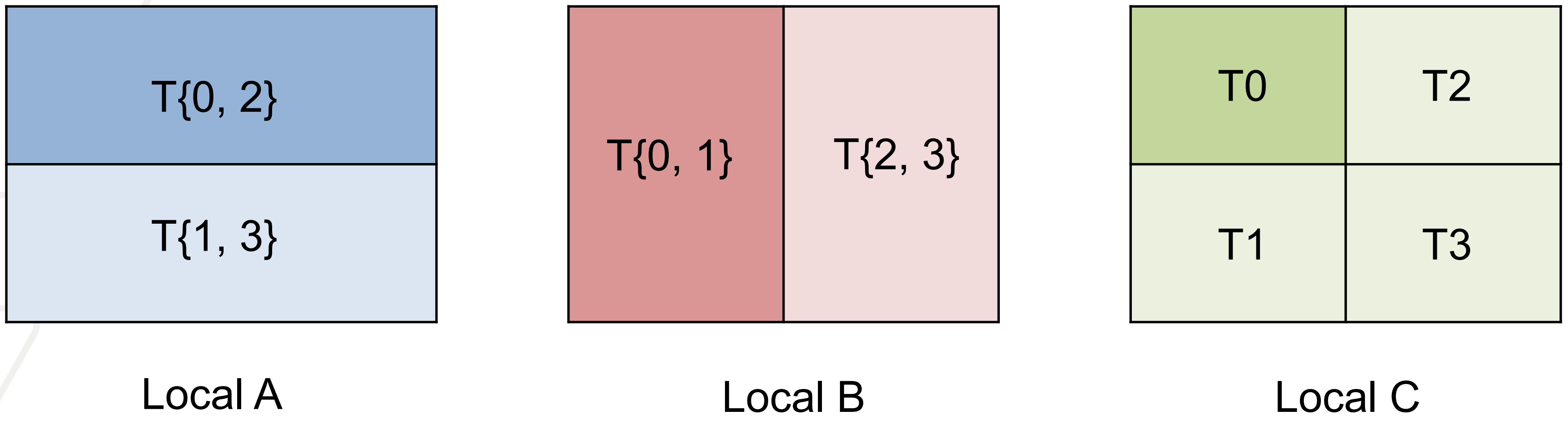
Multi-core Parallelism using openMP

- Each process spawns 8 threads to perform tiled semi-ring DGEMM on local matrices
- Each thread computes a subblock of output C

GPU Acceleration using CUDA

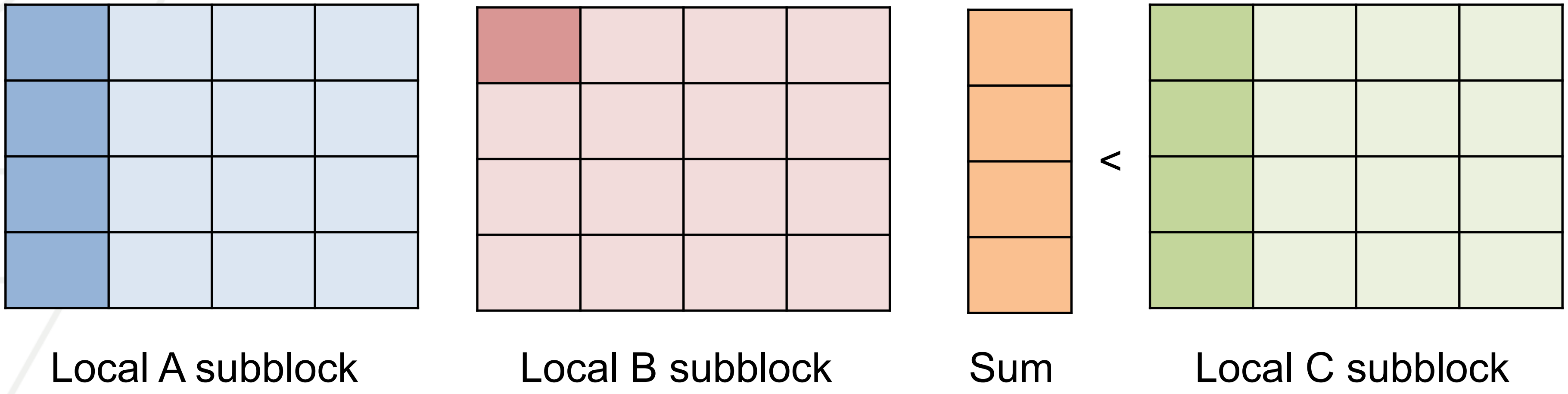
- Device memory is allocated for local matrices A, B, and C and copied from host to device per process
- Kernel is launched for DSRGEMM computation
- Local output is copied back from device memory to host memory, freeing the device memory

OpenMP DSRGEMM



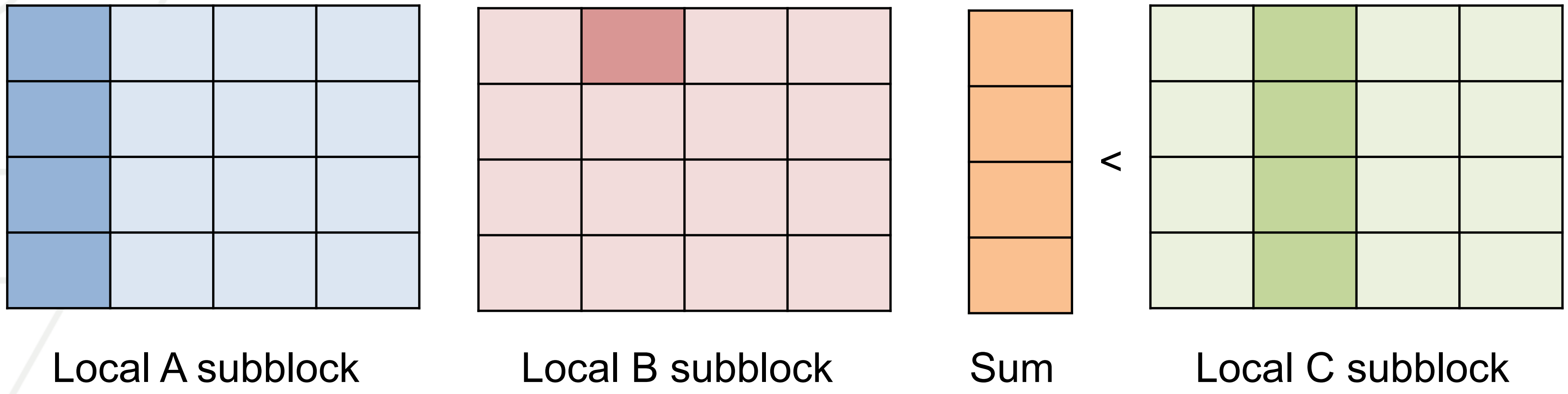
- Each process performs tiled semi-ring DSRGEMM
- Each thread computes a subblock of output C

OpenMP DSRGEMM



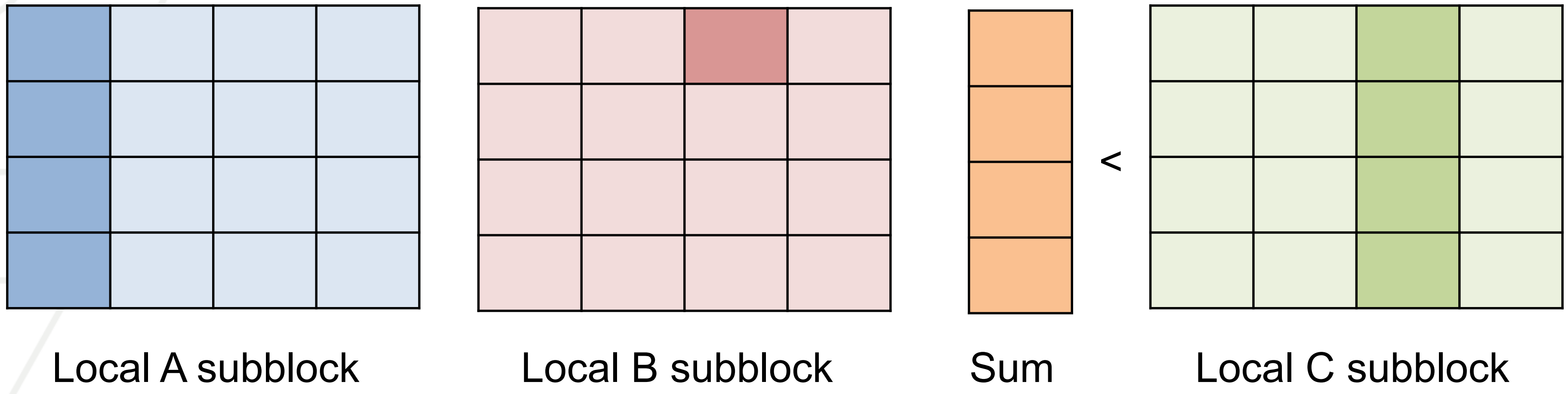
- Each column in A is added to a row in B to compute partial output in C
- OpenMP pragma SIMD used to improve performance

OpenMP DSRGEMM



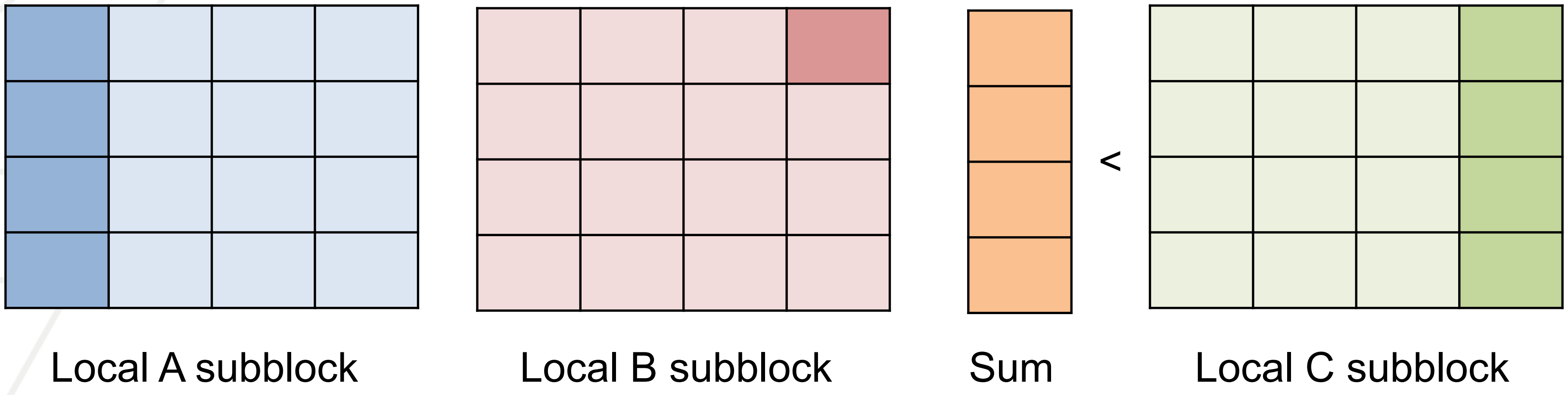
- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



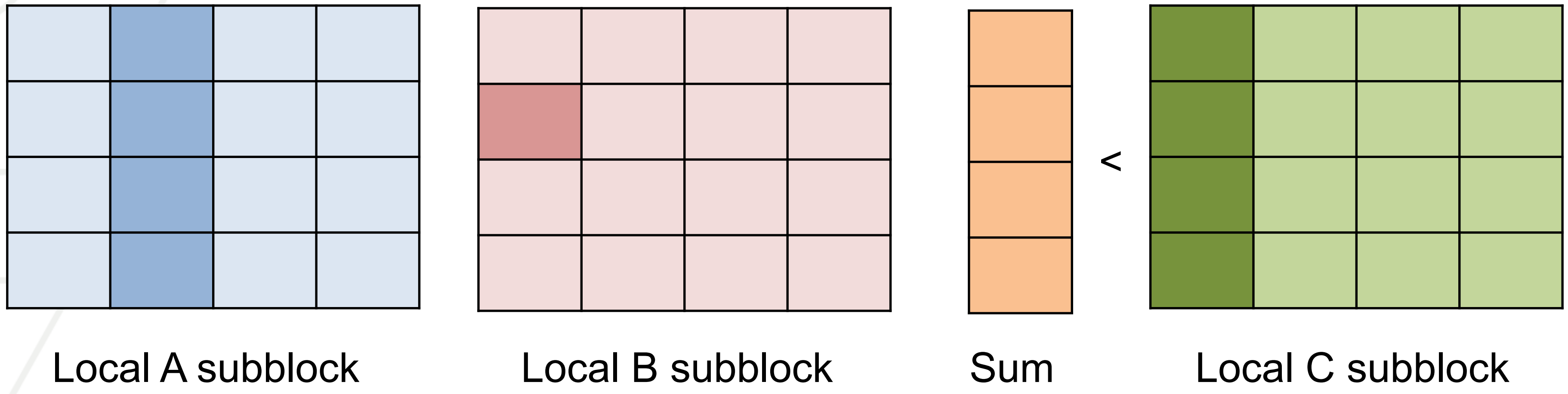
- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



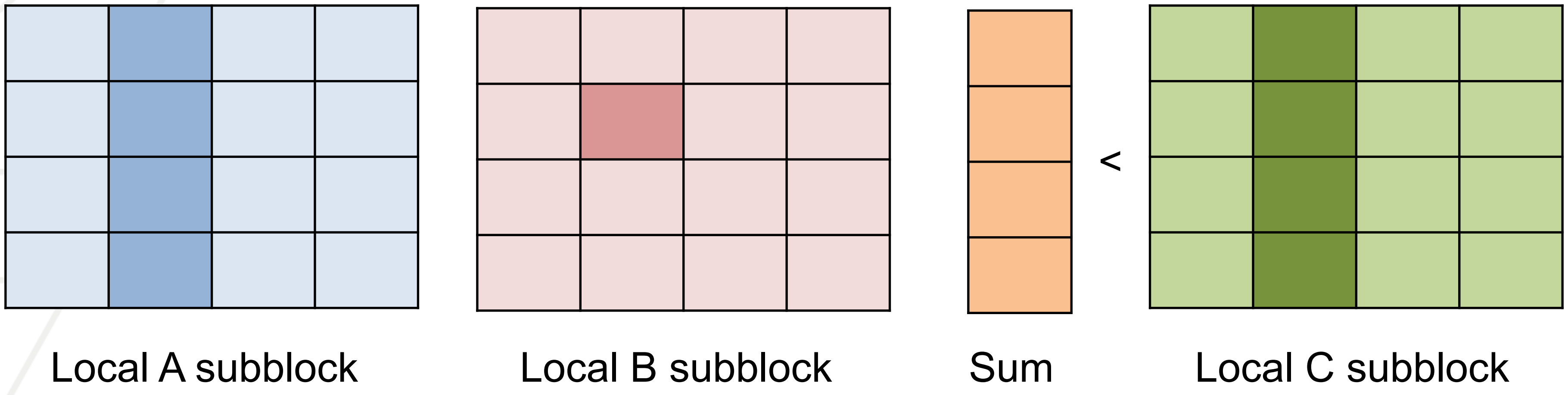
- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



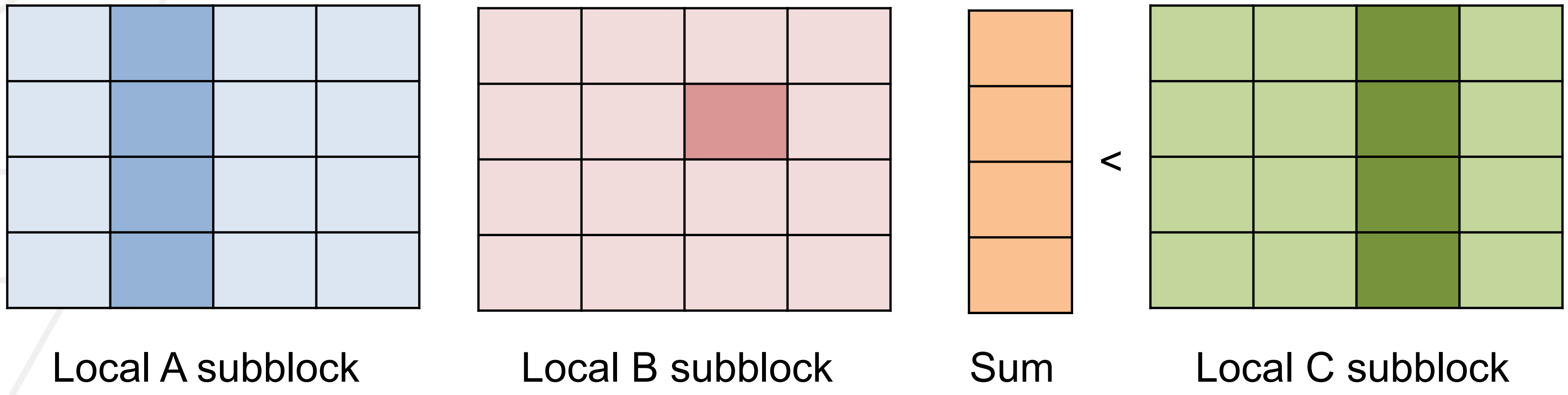
- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



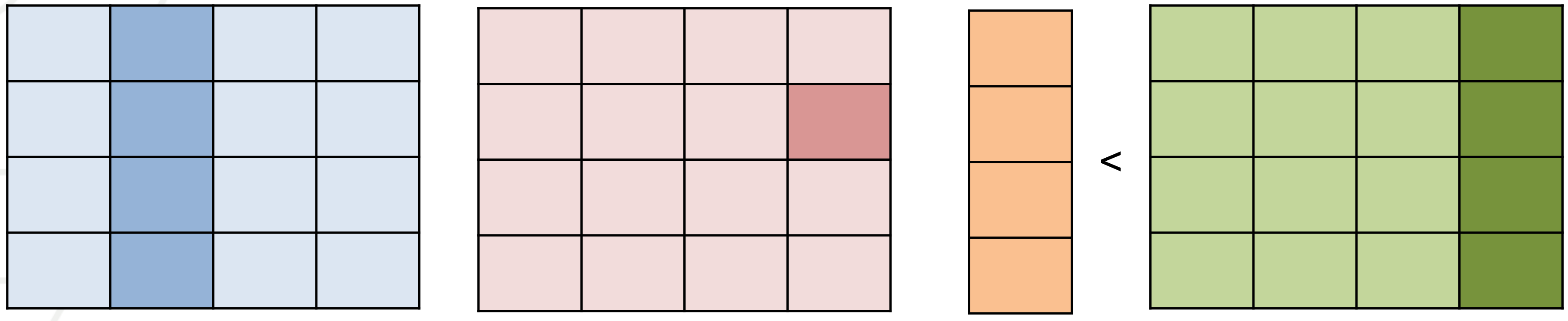
- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



Local A subblock

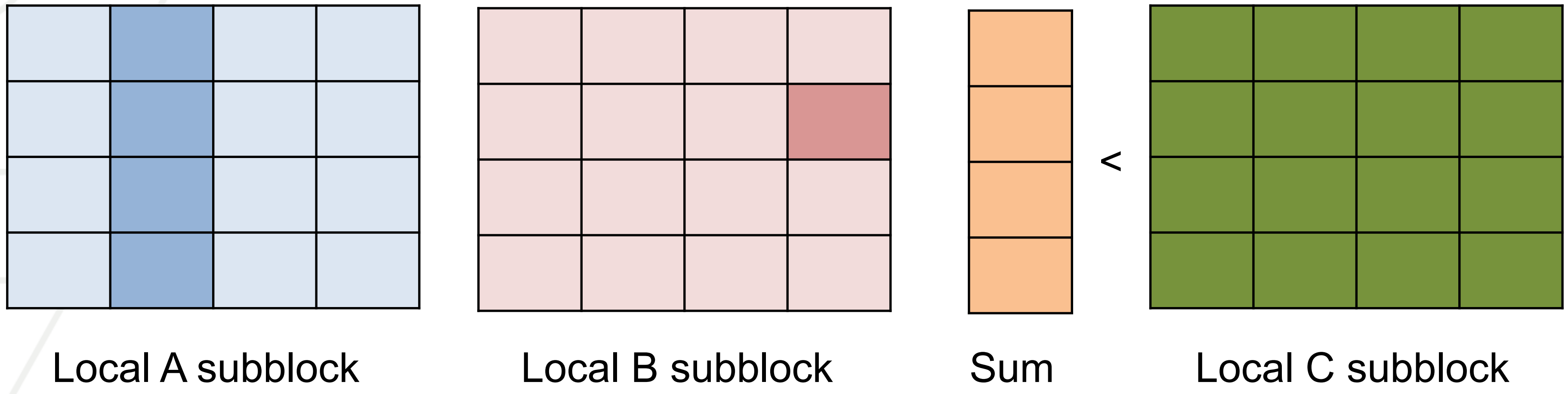
Local B subblock

Sum

Local C subblock

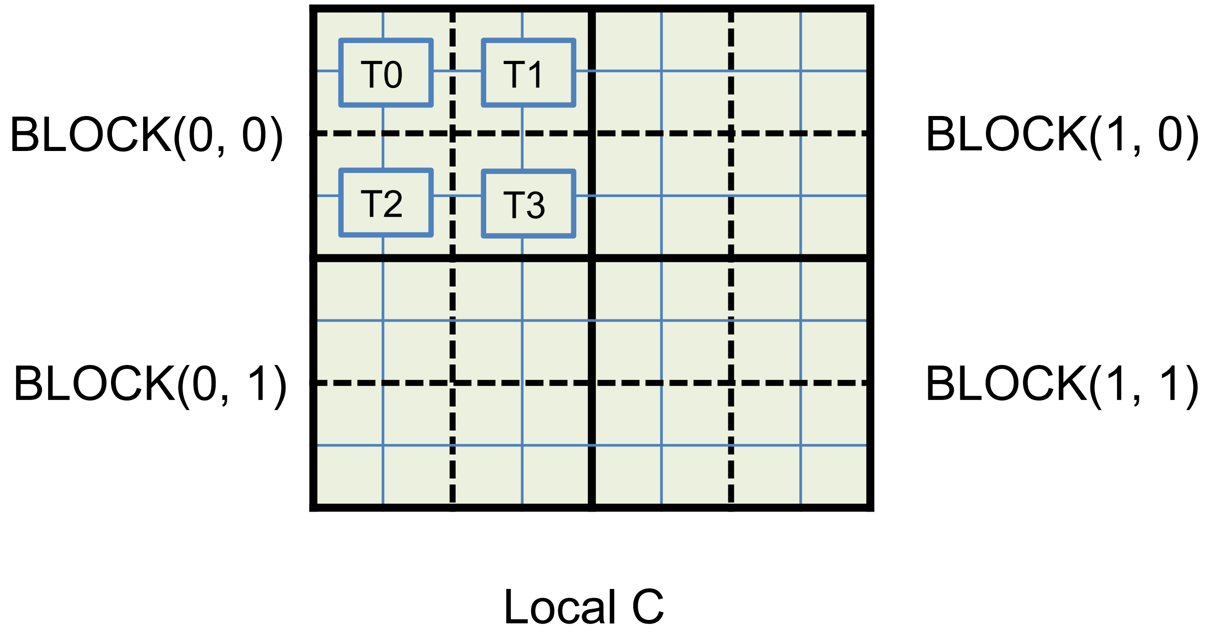
- Partial outputs in entire subblock of C are computed using a column of A and a row of B
- No elements in A or B are reloaded, allowing maximum data reuse

OpenMP DSRGEMM



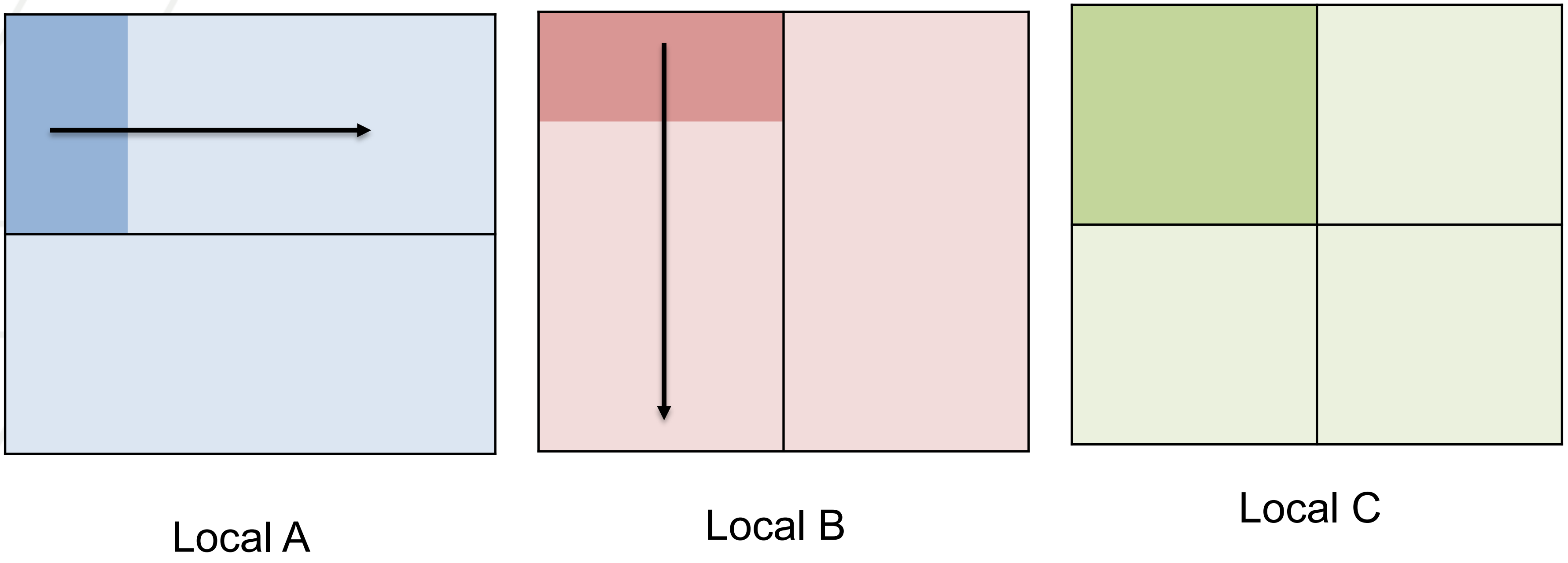
- Operating column-wise exploits spatial locality and column-major layout in memory
- Cache requirement reduced per core due to data reuse:
  L * W * 8 bytes (L = length of subblock, W = width of subblock)

CUDA DSRGEMM



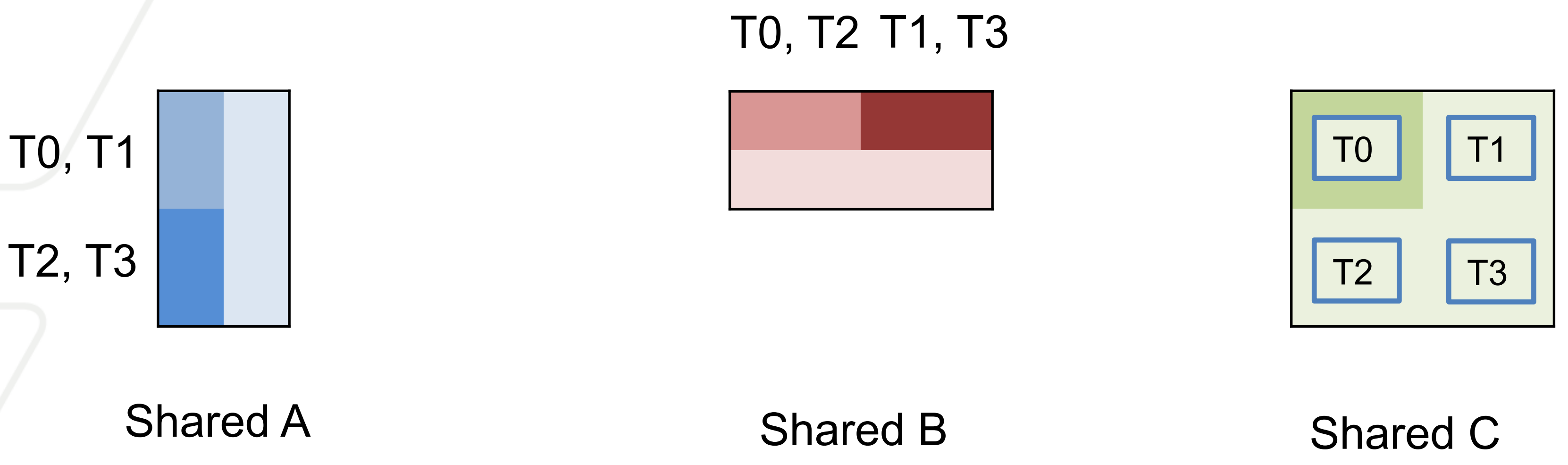
- Local matrix C is divided into a 2D grid of blocks of size TILE_M * TILE_N
- Each thread per block computes T_M * T_N outputs
- Each block contains (TILE_M * TILE_N / T_M * T_N) threads

CUDA DSRGEMM



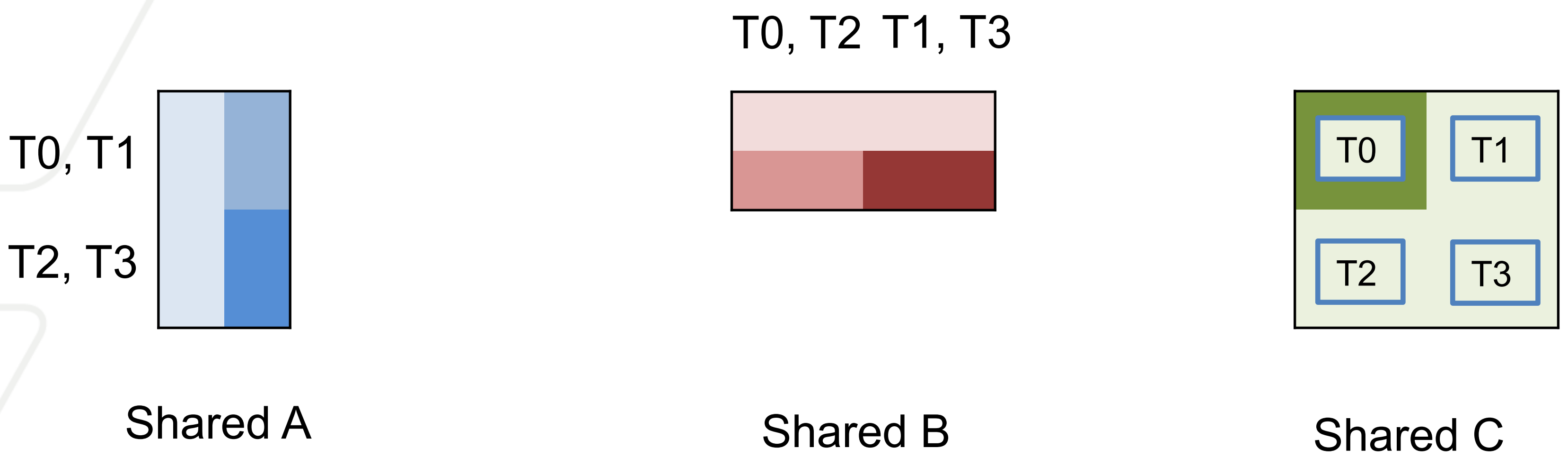
- Shared memory of size (TILE_M + TILE_N) * TILE_K dynamically allocated
- All threads in a block simultaneously load TILE_M * TILE_K block of local A and TILE_K * TILE_N block of local B in shared memory to compute partial outputs in local C
- Vectorized loads/stored can be used to improved performance

CUDA DSRGEMM



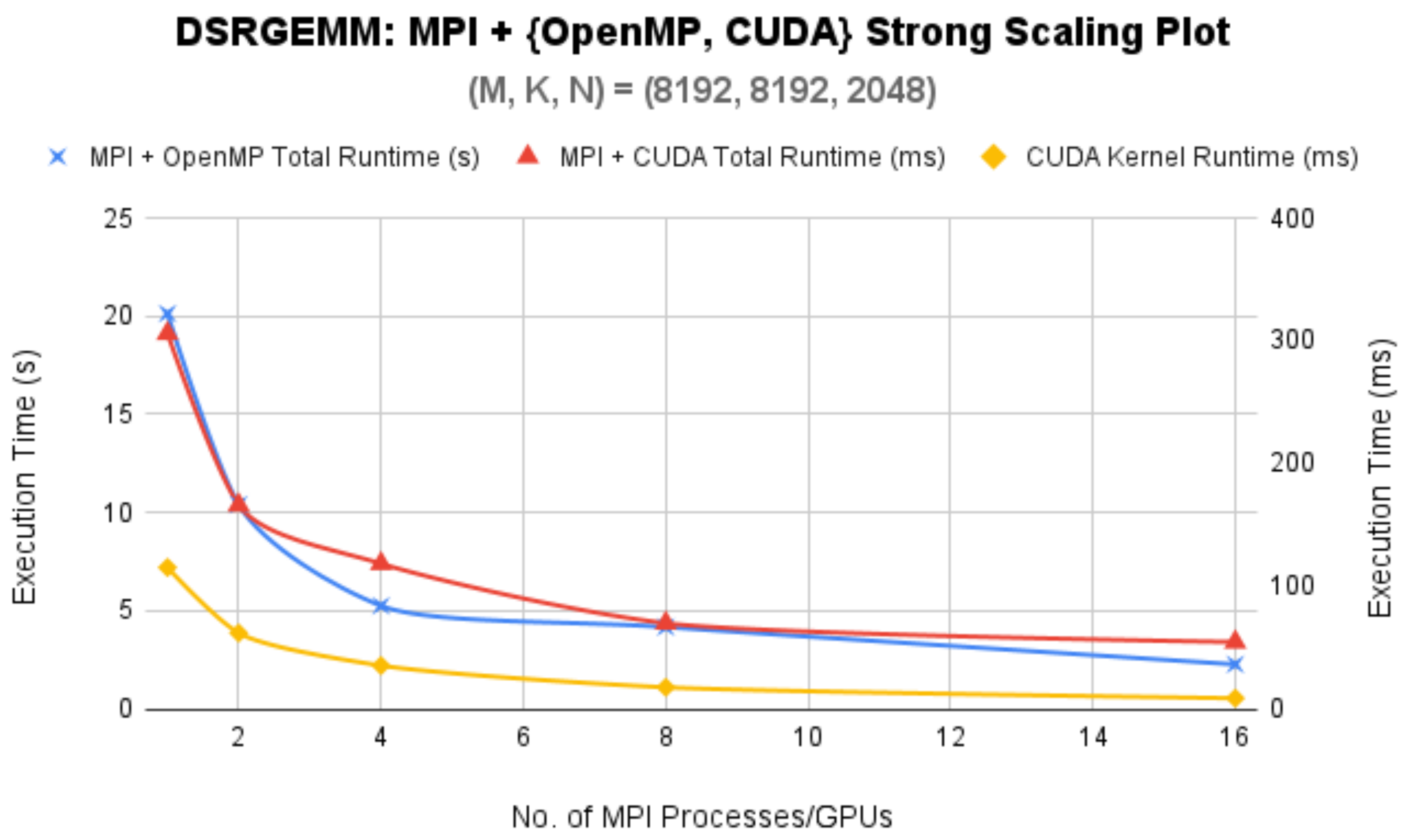
- Each thread uses register memory to load T_M elements from column of shared A and T_N elements from row of shared B to compute T_M * T_N outputs
- Output computation per thread is done using the algorithm described for openMP implementation to allow data reuse

CUDA DSRGEMM



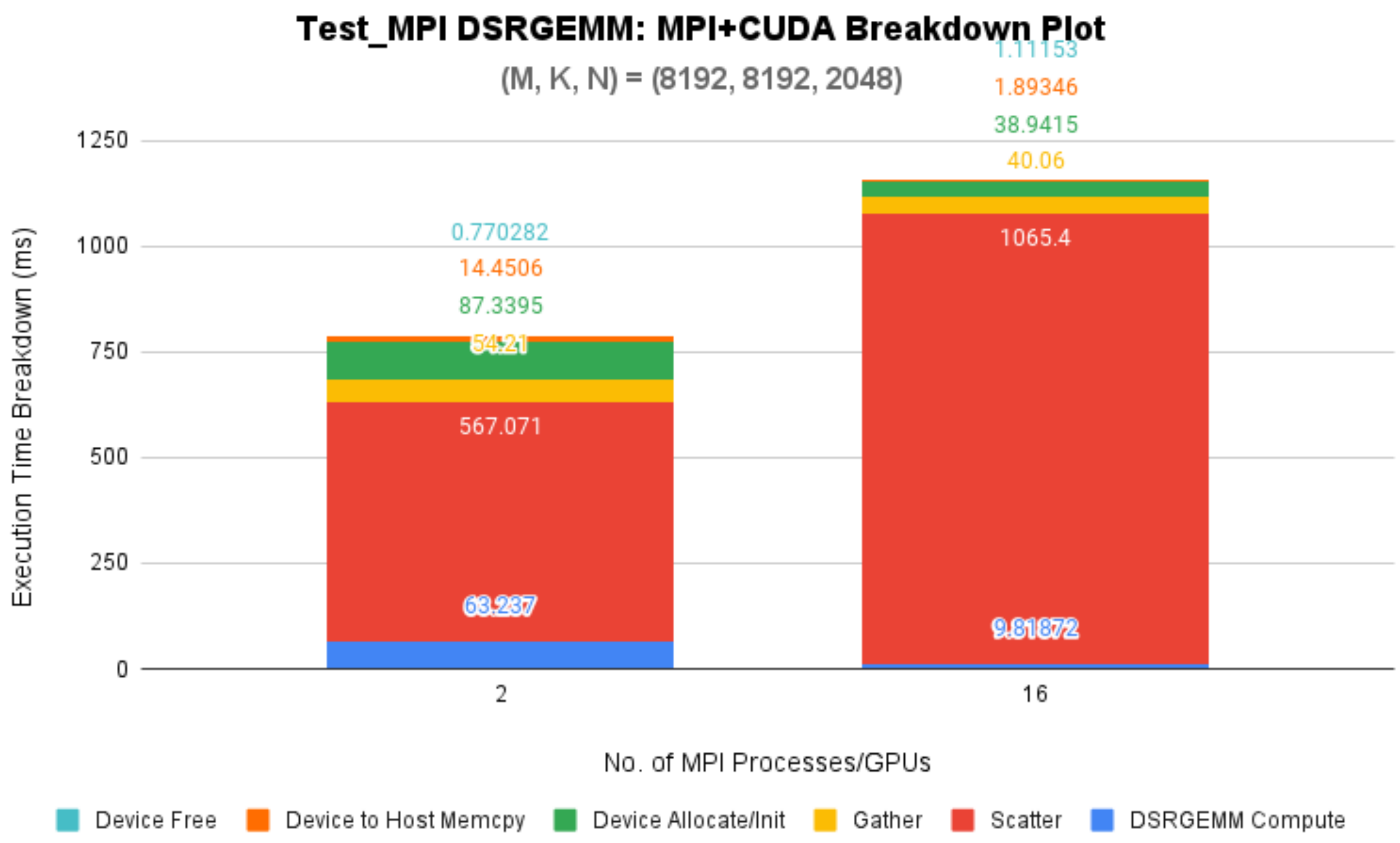
- Each thread uses register memory to load T_M elements from column of shared A and T_N elements from row of shared B to compute T_M * T_N outputs
- Output computation per thread is done using the algorithm described for openMP implementation to allow data reuse

Strong Scaling Plot

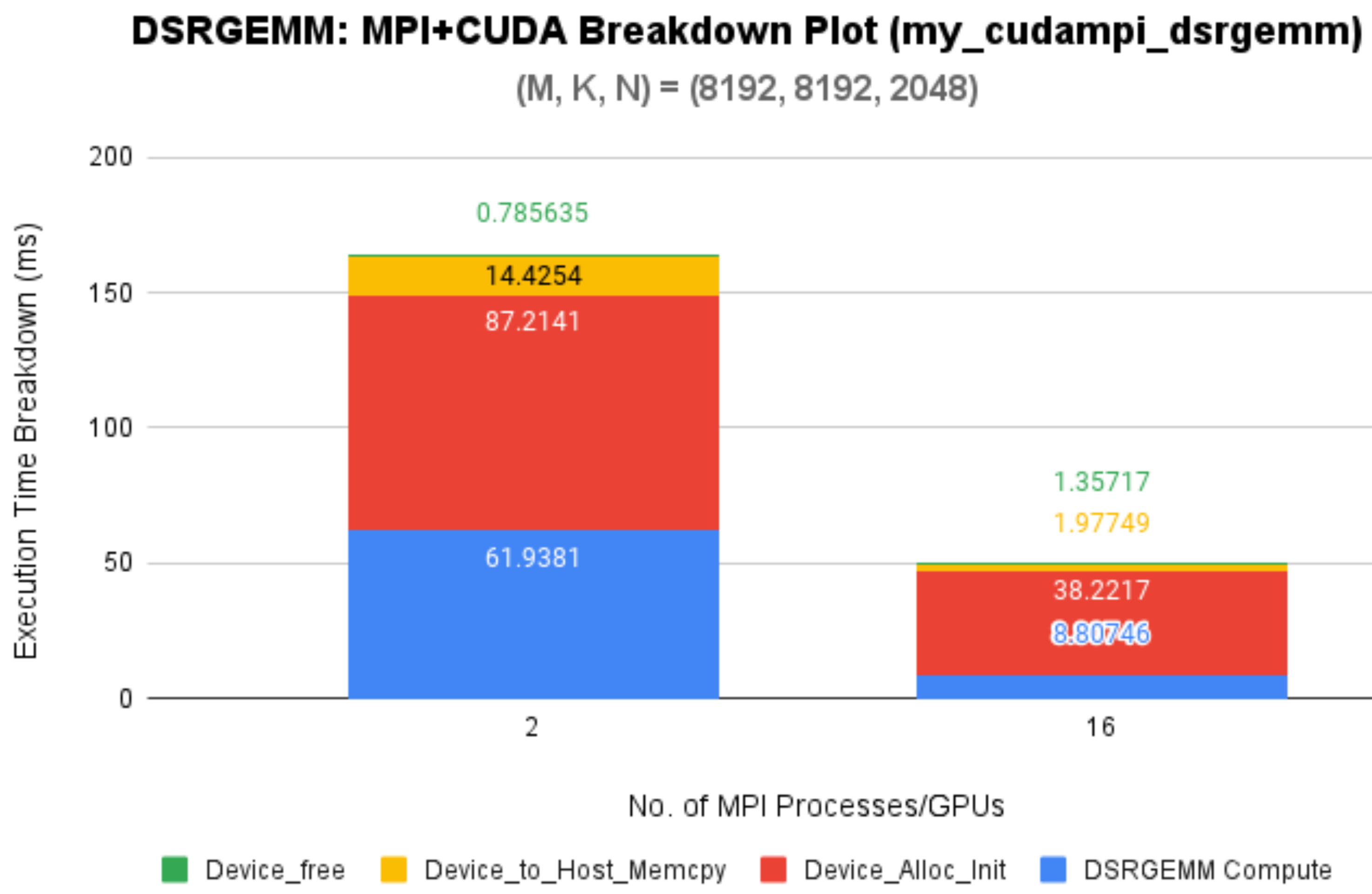


- Note:
  - MPI processes need to be mapped per node for maximum openMP thread utilization
  - CUDA DSRGEMM requires warmup to reduce CUDA API call overhead

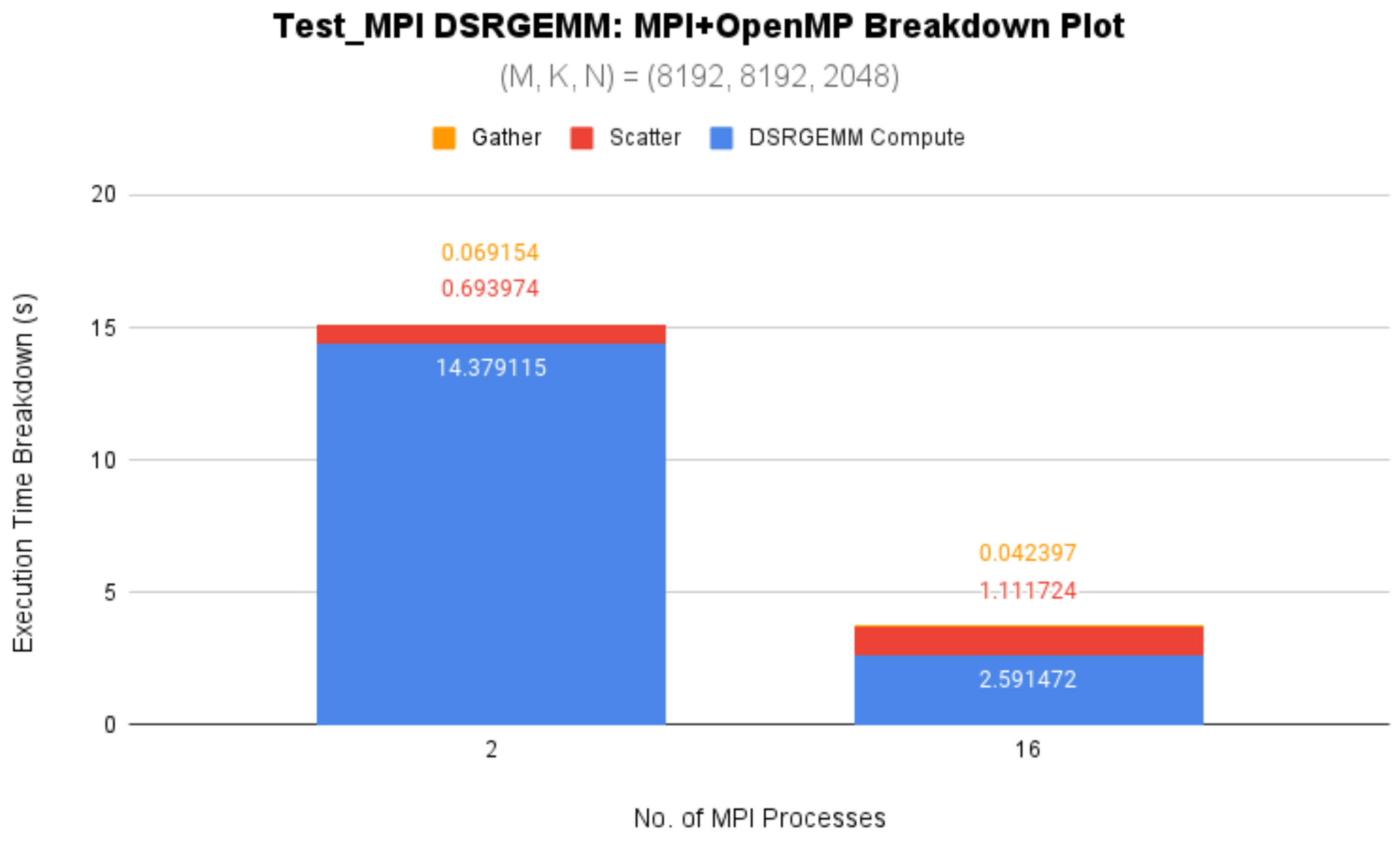
MPI + CUDA Breakdown Plot



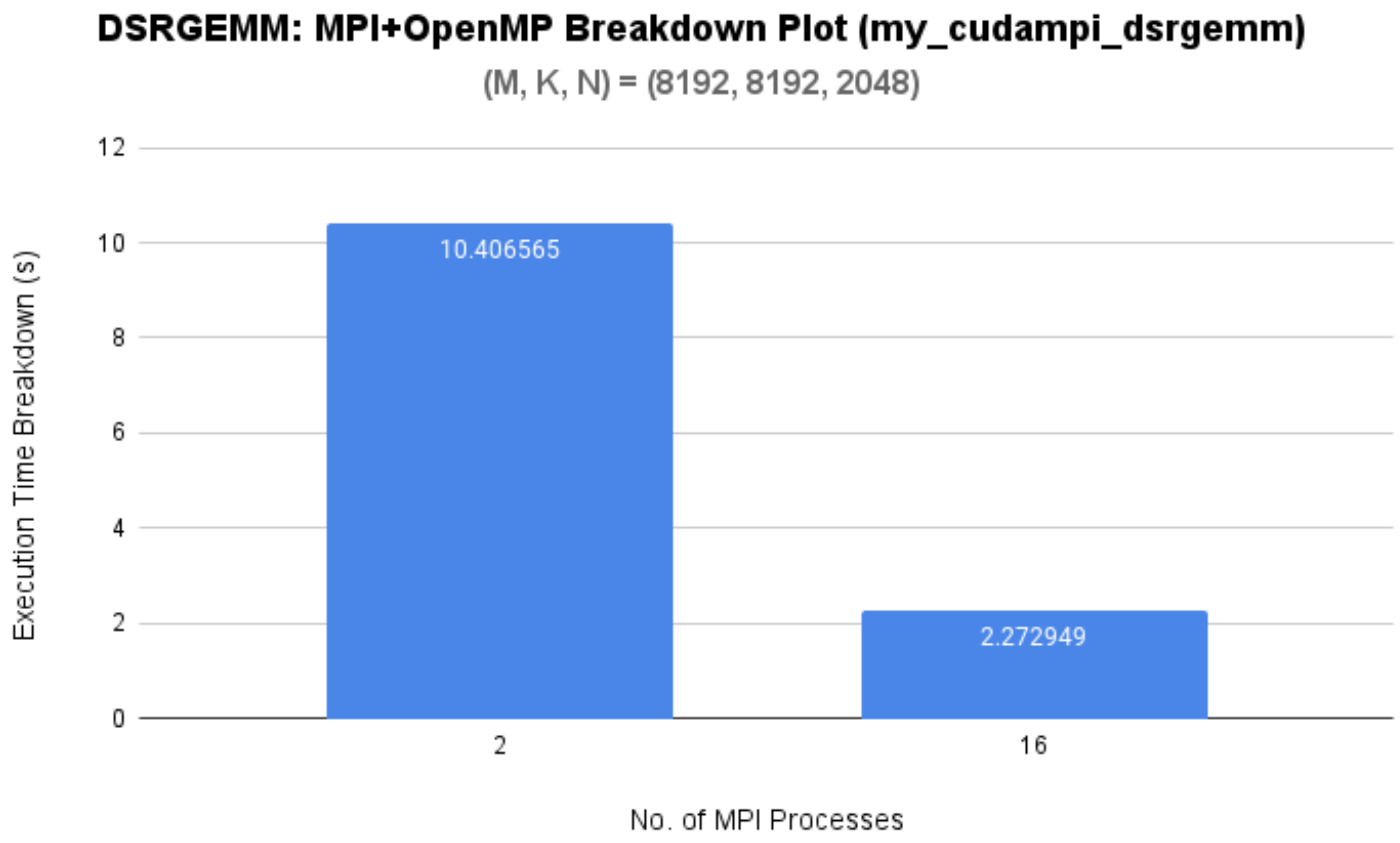
MPI + CUDA Breakdown Plot



MPI + OpenMP Breakdown Plot



MPI + OpenMP Breakdown Plot

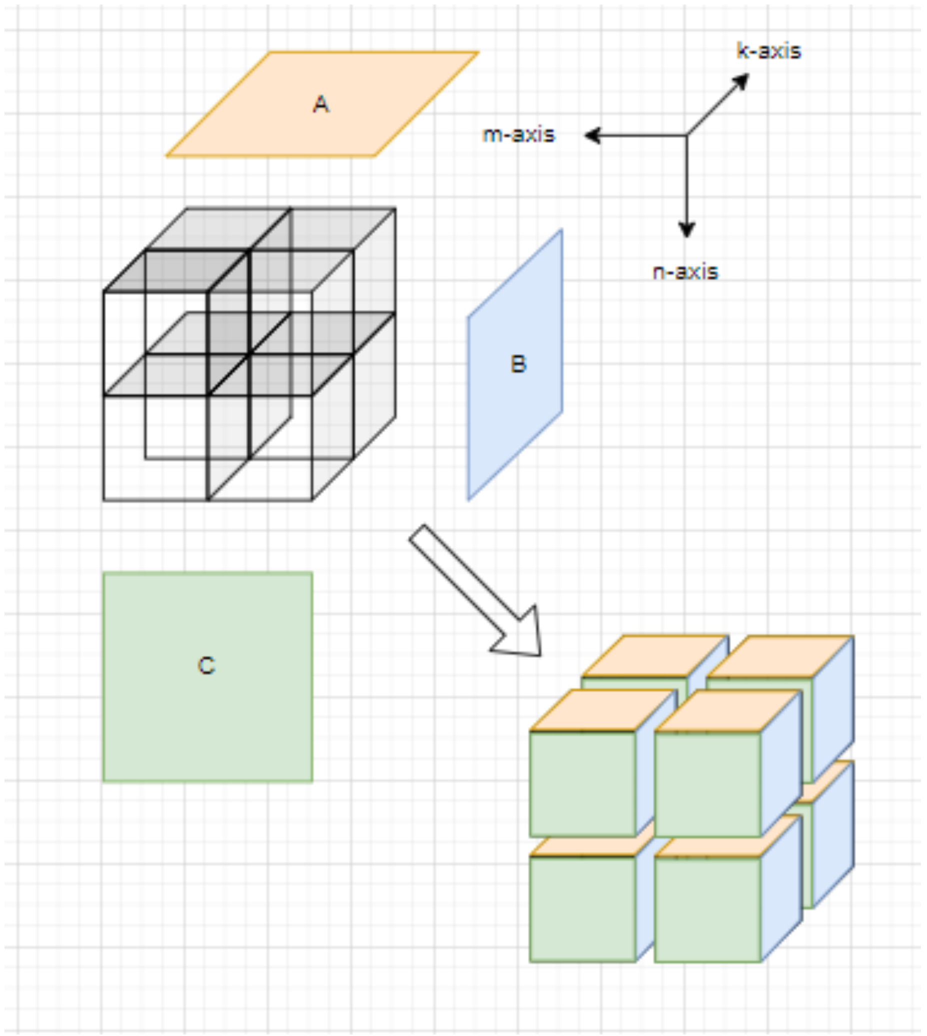


Thank you!

Techniques about HW4

Changhai Man, cman8@gatech.edu

Communications

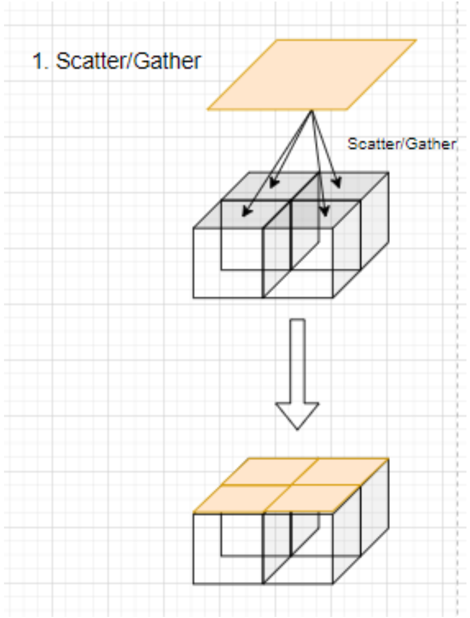


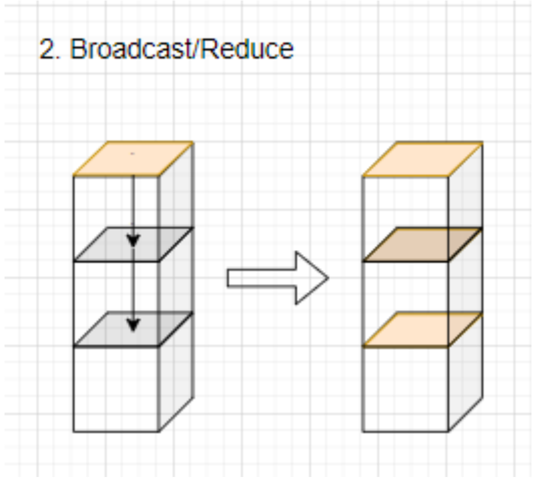
If we see the communications of Matmul in 3D grid, then:

- The input/output data is at the surface of the grid.
- For each grid, it is a computation node
- And to load/unload data to each node, it is like coloring the surface of each small cube

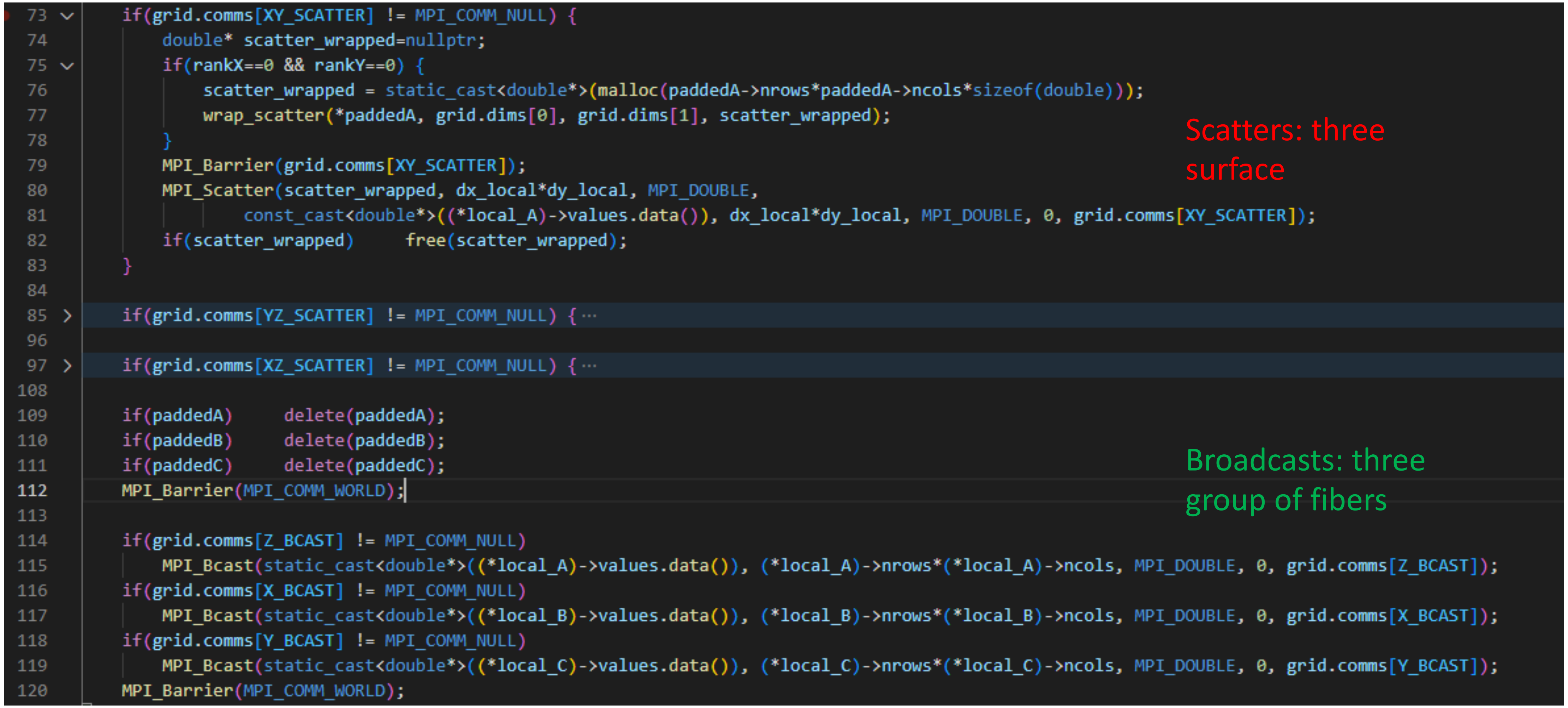
How to?

• Scatter/Gather:	m-n, m-k, n-k
• Broadcast/Reduce:	m, n, k





Communications



```
if(grid.comms[XY_SCATTER] != MPI_COMM_NULL) {
    double* scatter_wrapped=nullptr;
    if(rankX==0 && rankY==0) {
        scatter_wrapped = static_cast<double*>(malloc(paddedA->nrows*paddedA->ncols*sizeof(double)));
        wrap_scatter(*paddedA, grid.dims[0], grid.dims[1], scatter_wrapped);
    }
    MPI_Barrier(grid.comms[XY_SCATTER]);
    MPI_Scatter(scatter_wrapped, dx_local*dy_local, MPI_DOUBLE,
            const_cast<double*>((*local_A)->values.data()), dx_local*dy_local, MPI_DOUBLE, 0, grid.comms[XY_SCATTER]);
    if(scatter_wrapped)     free(scatter_wrapped);
}

if(grid.comms[YZ_SCATTER] != MPI_COMM_NULL) {⋯

if(grid.comms[XZ_SCATTER] != MPI_COMM_NULL) {⋯

if(paddedA)     delete(paddedA);
if(paddedB)     delete(paddedB);
if(paddedC)     delete(paddedC);
MPI_Barrier(MPI_COMM_WORLD);

if(grid.comms[Z_BCAST] != MPI_COMM_NULL)
    MPI_Bcast(static_cast<double*>((*local_A)->values.data()), (*local_A)->nrows*(*local_A)->ncols, MPI_DOUBLE, 0, grid.comms[Z_BCAST]);
if(grid.comms[X_BCAST] != MPI_COMM_NULL)
    MPI_Bcast(static_cast<double*>((*local_B)->values.data()), (*local_B)->nrows*(*local_B)->ncols, MPI_DOUBLE, 0, grid.comms[X_BCAST]);
if(grid.comms[Y_BCAST] != MPI_COMM_NULL)
    MPI_Bcast(static_cast<double*>((*local_C)->values.data()), (*local_C)->nrows*(*local_C)->ncols, MPI_DOUBLE, 0, grid.comms[Y_BCAST]);
MPI_Barrier(MPI_COMM_WORLD);
```

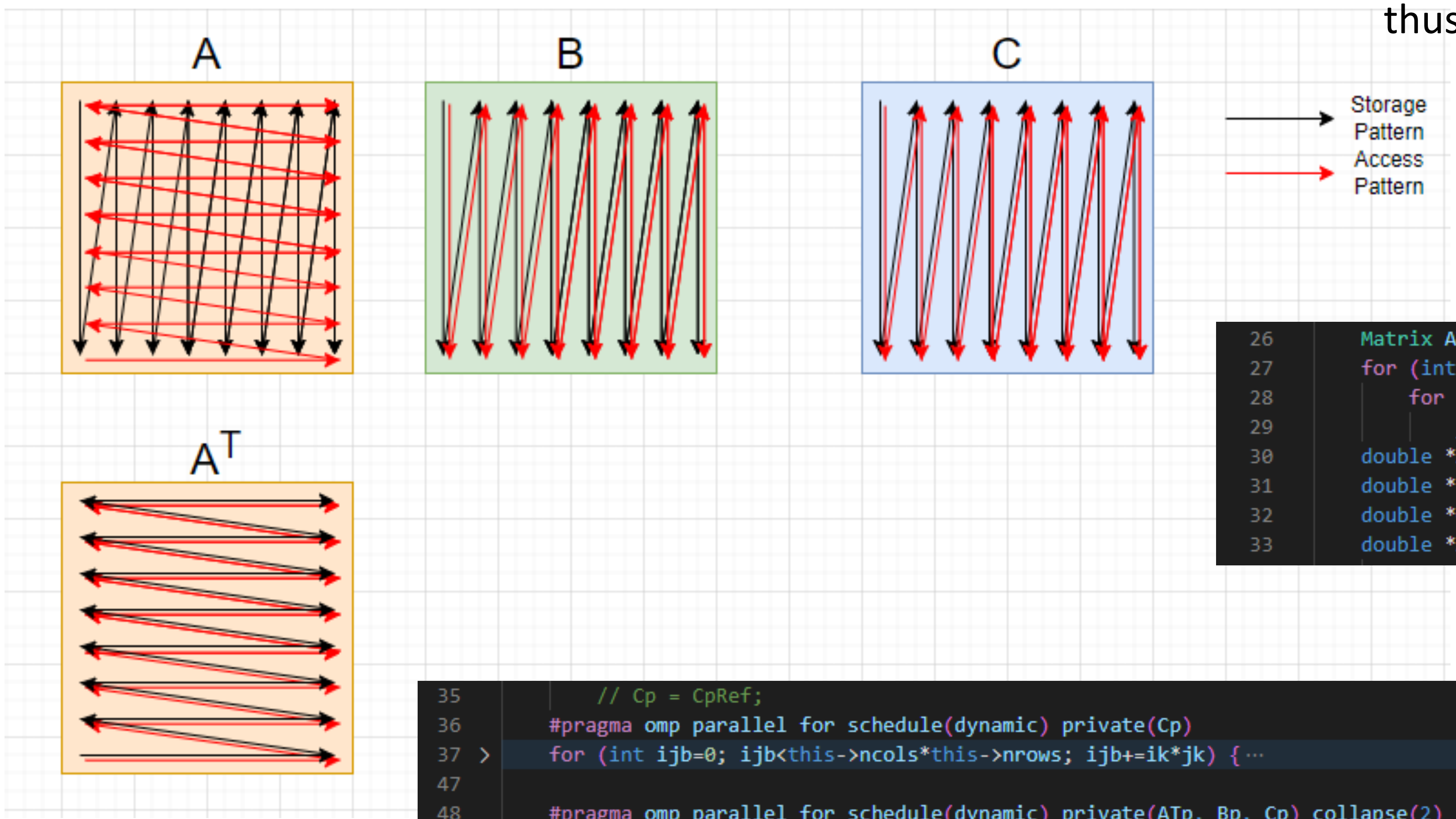
Scatters: three surface

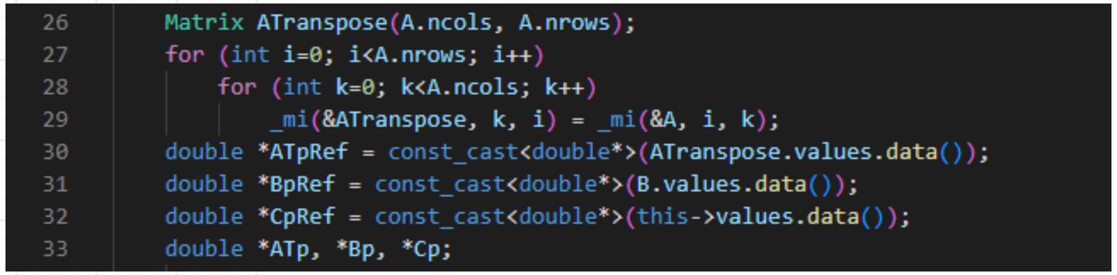
Broadcasts: three group of fibers

CPU version

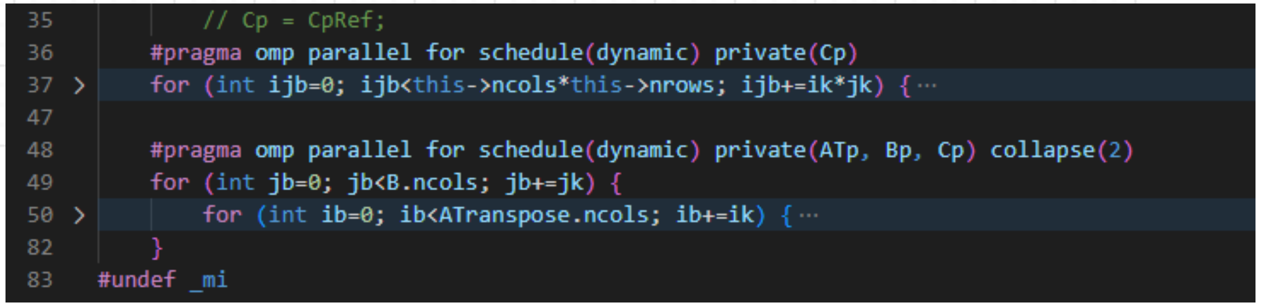
Cache Efficient Access

1. By transpose matrix A prior to the Matmul, the access pattern and storage pattern can better matched, thus increase the cache efficiency.
2. To make access of C efficient, the access pattern should match the storage pattern, thus the i-loop should be inside j-loop



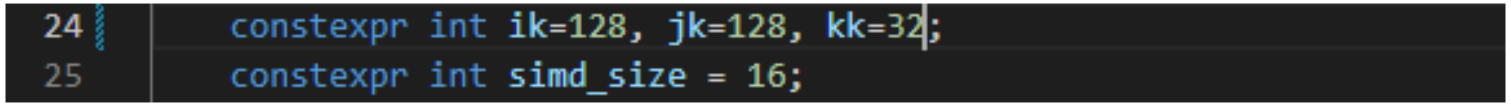


```
Matrix ATranspose(A.ncols, A.nrows);
for (int i=0; i<A.nrows; i++)
    for (int k=0; k<A.ncols; k++)
        _mi(&ATranspose, k, i) = _mi(&A, i, k);
double *ATpRef = const_cast<double*>(ATranspose.values.data());
double *BpRef = const_cast<double*>(B.values.data());
double *CpRef = const_cast<double*>(this->values.data());
double *ATp, *Bp, *Cp;
```



```
        // Cp = CpRef;
    #pragma omp parallel for schedule(dynamic) private(Cp)
    for (int ijb=0; ijb<this->ncols*this->nrows; ijb+=ik*jk) {⋯

    #pragma omp parallel for schedule(dynamic) private(ATp, Bp, Cp) collapse(2)
    for (int jb=0; jb<B.ncols; jb+=jk) {
        for (int ib=0; ib<ATranspose.ncols; ib+=ik) {⋯
    }
#undef _mi
```

CPU version



```
constexpr int ik=128, jk=128, kk=32;
constexpr int simd_size = 16;
```

Tiling

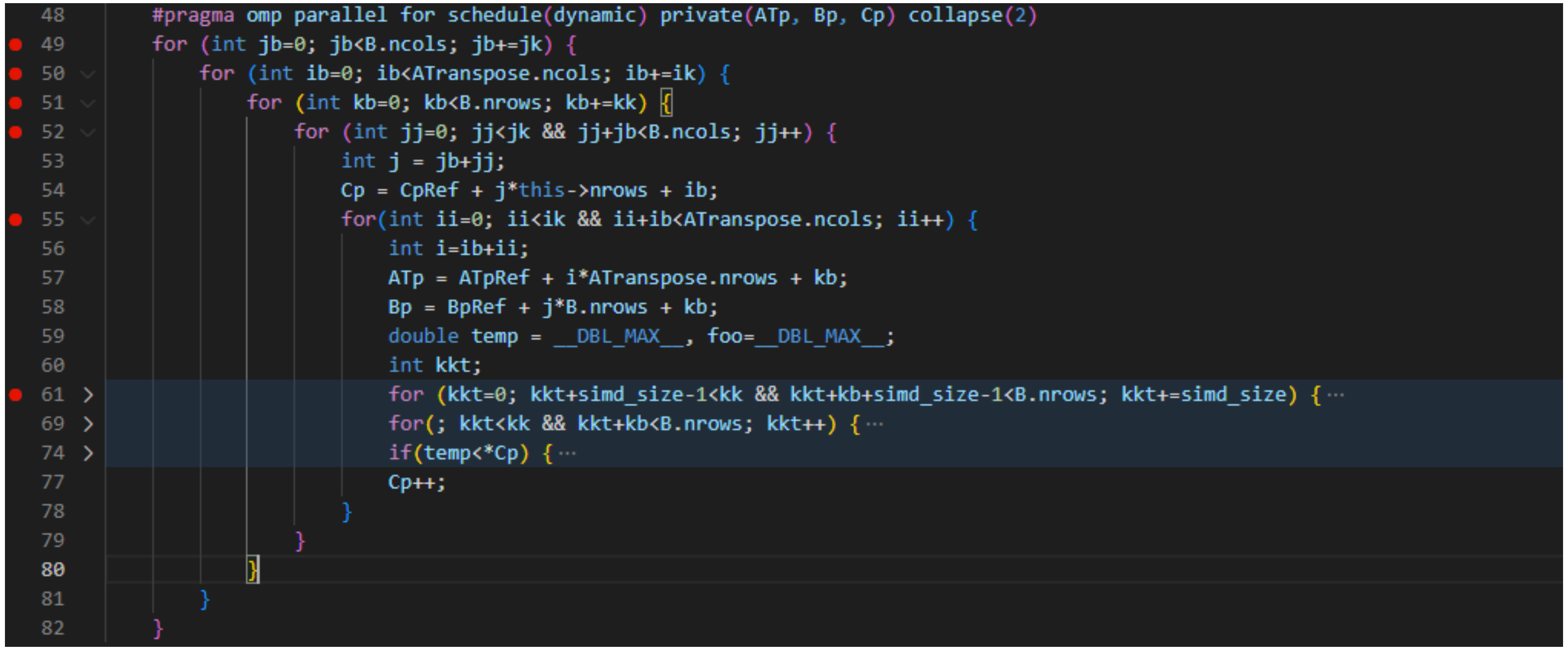
To fully utilize the cache of CPU, we want to make sure that for each run, the data size should be able to kept in the cache (instead of roundly kicked out by new data)

Loop order:
JBlock->iBlock->kBlock->j->i->k

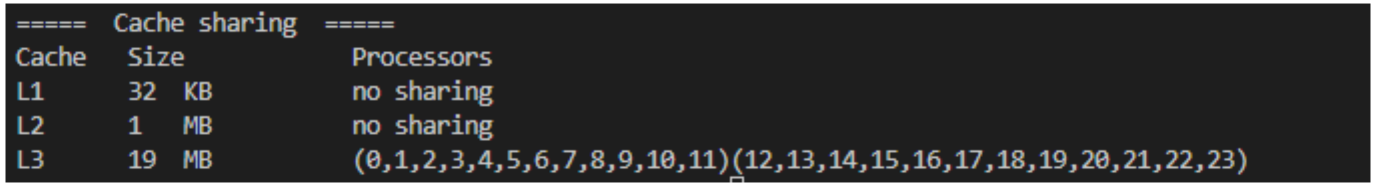
For each block:
(i, k) = 8*32*128=32KB matched L1d size

(j, k) = 32KB, (i, j) = 8*128*128=256KB.
(I, k) (j, k), (I, j) totally 320KB, matched L2 size

For maximum 12 cores per socket, total L3 size requirement is: 320 * 12 = 3.84 MB, fit L3 size



```
#pragma omp parallel for schedule(dynamic) private(ATp, Bp, Cp) collapse(2)
for (int jb=0; jb<B.ncols; jb+=jk) {
    for (int ib=0; ib<ATranspose.ncols; ib+=ik) {
        for (int kb=0; kb<B.nrows; kb+=kk) {
            for (int jj=0; jj<jk && jj+jb<B.ncols; jj++) {
                int j = jb+jj;
                Cp = CpRef + j*this->nrows + ib;
                for(int ii=0; ii<ik && ii+ib<ATranspose.ncols; ii++) {
                    int i=ib+ii;
                    ATp = ATpRef + i*ATranspose.nrows + kb;
                    Bp = BpRef + j*B.nrows + kb;
                    double temp = __DBL_MAX__, foo=__DBL_MAX__;
                    int kkt;
                    for (kkt=0; kkt+simd_size-1<kk && kkt+kb+simd_size-1<B.nrows; kkt+=simd_size) {⋯
                    for(; kkt<kk && kkt+kb<B.nrows; kkt++) {⋯
                    if(temp<*Cp) {⋯
                    Cp++;
                }
            }
        }
    }
}
```



```
=====  Cache sharing  =====
Cache   Size          Processors
L1      32  KB        no sharing
L2      1   MB        no sharing
L3      19  MB        (0,1,2,3,4,5,6,7,8,9,10,11)(12,13,14,15,16,17,18,19,20,21,22,23)
```

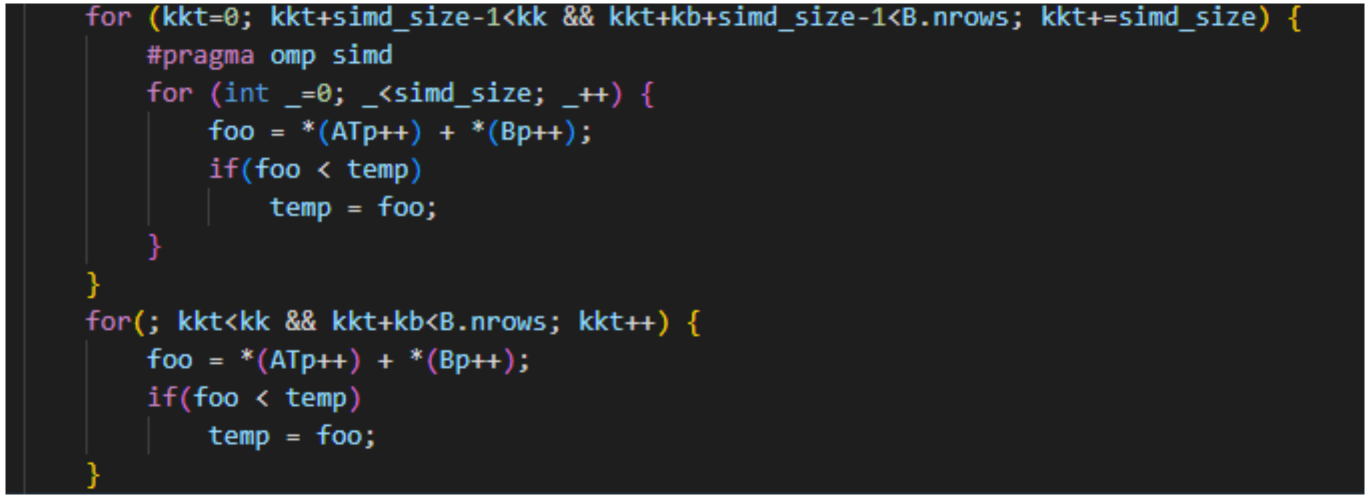
CPU version

Intel® 64 ‡	Yes
Instruction Set Extensions	Intel® SSE4.2, Intel® AVX, Intel® AVX2, Intel® AVX-512
# of AVX-512 FMA Units	2
Enhanced Intel SpeedStep® Technology	Yes
Intel® Volume Management Device (VMD)	Yes

Other tricks:

- Access using pointer:
  avoid time for calculating offset

- SIMD: Try to fit AVX512, SIMD=16



```
for (kkt=0; kkt+simd_size-1<kk && kkt+kb+simd_size-1<B.nrows; kkt+=simd_size) {
    #pragma omp simd
    for (int _=0; _<simd_size; _++) {
        foo = *(ATp++) + *(Bp++);
        if(foo < temp)
            temp = foo;
    }
}
for(; kkt<kk && kkt+kb<B.nrows; kkt++) {
    foo = *(ATp++) + *(Bp++);
    if(foo < temp)
        temp = foo;
}
```

GPU version

Actually, similar to CPU but different in two things：
- More dimensions for for-loops: either in time or space
- More tilling level as there are more complicated memory hierarchy

For-loop parallelism: space or time
Loop order:

JBlock->iBlock->kBlock->jThread->iThread->k->j->I

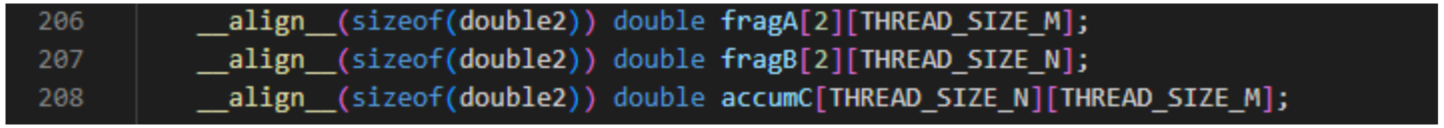
Where:

• (Jblock, Iblock) => BlockIdx(SM level)	Loop in Space
• Kblock	Loop in Time
• (Jthread, Ithread) => ThreadIdx(SP level)	Loop in Space
• k, j, i	Loop in Time

Tiling and storage:

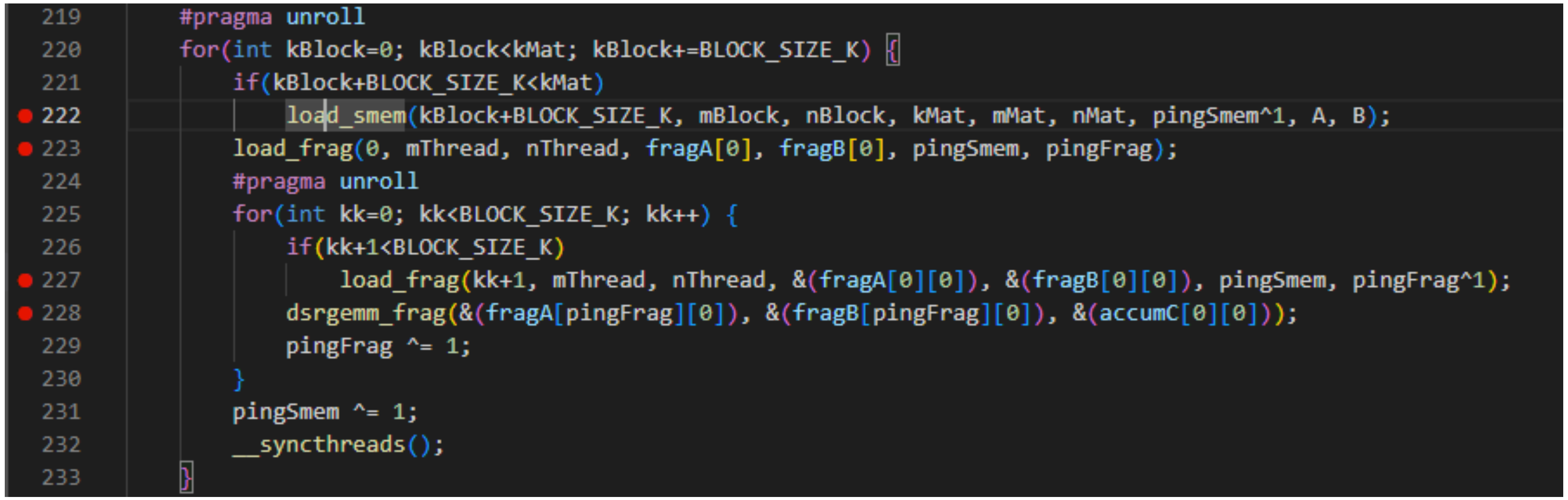
(Iblock, Kblock), (Jblock, Kblock), (Iblock, Jblock):	GDDR (Global Mem)
(Ithread, k), (Jthread, k):	Shared Mem
(Ithread, Jthread), (I, k), (J, k):	reg files

GPU version



```
    __align__(sizeof(double2)) double fragA[2][THREAD_SIZE_M];
    __align__(sizeof(double2)) double fragB[2][THREAD_SIZE_N];
    __align__(sizeof(double2)) double accumC[THREAD_SIZE_N][THREAD_SIZE_M];
```

Aligned reg files for each thread



```
#pragma unroll
for(int kBlock=0; kBlock<kMat; kBlock+=BLOCK_SIZE_K) {
    if(kBlock+BLOCK_SIZE_K<kMat)
        load_smem(kBlock+BLOCK_SIZE_K, mBlock, nBlock, kMat, mMat, nMat, pingSmem^1, A, B);
    load_frag(0, mThread, nThread, fragA[0], fragB[0], pingSmem, pingFrag);
    #pragma unroll
    for(int kk=0; kk<BLOCK_SIZE_K; kk++) {
        if(kk+1<BLOCK_SIZE_K)
            load_frag(kk+1, mThread, nThread, &(fragA[0][0]), &(fragB[0][0]), pingSmem, pingFrag^1);
        dsrgemm_frag(&(fragA[pingFrag][0]), &(fragB[pingFrag][0]), &(accumC[0][0]));
        pingFrag ^= 1;
    }
    pingSmem ^= 1;
    __syncthreads();
}
```

Shared mem and Frags (Register Files)