Matrix Multiply with OpenMP and Cuda

James Canterbury and Tyler Collins

Project Overview

- Goal: Implement matrix multiply in OpenMP and Cuda
- Baseline Code:
  - naiveMult from program 2 (pictured below)
- All matrix multiply versions were compared to the naiveMult in program 2 for a consistent performance metric

```
void naiveMult(float * A, float * B, float * C, uint64_t M, uint64_t N, uint64_t L)
{
    float accum;
    uint64_t i,j,k;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            accum = 0;
            for (k = 0; k < L; k++)
            {
                accum += A[i*L+k]*B[k*N+j];
            }
            C[i*N+j] = accum;
        }
    }
}
```

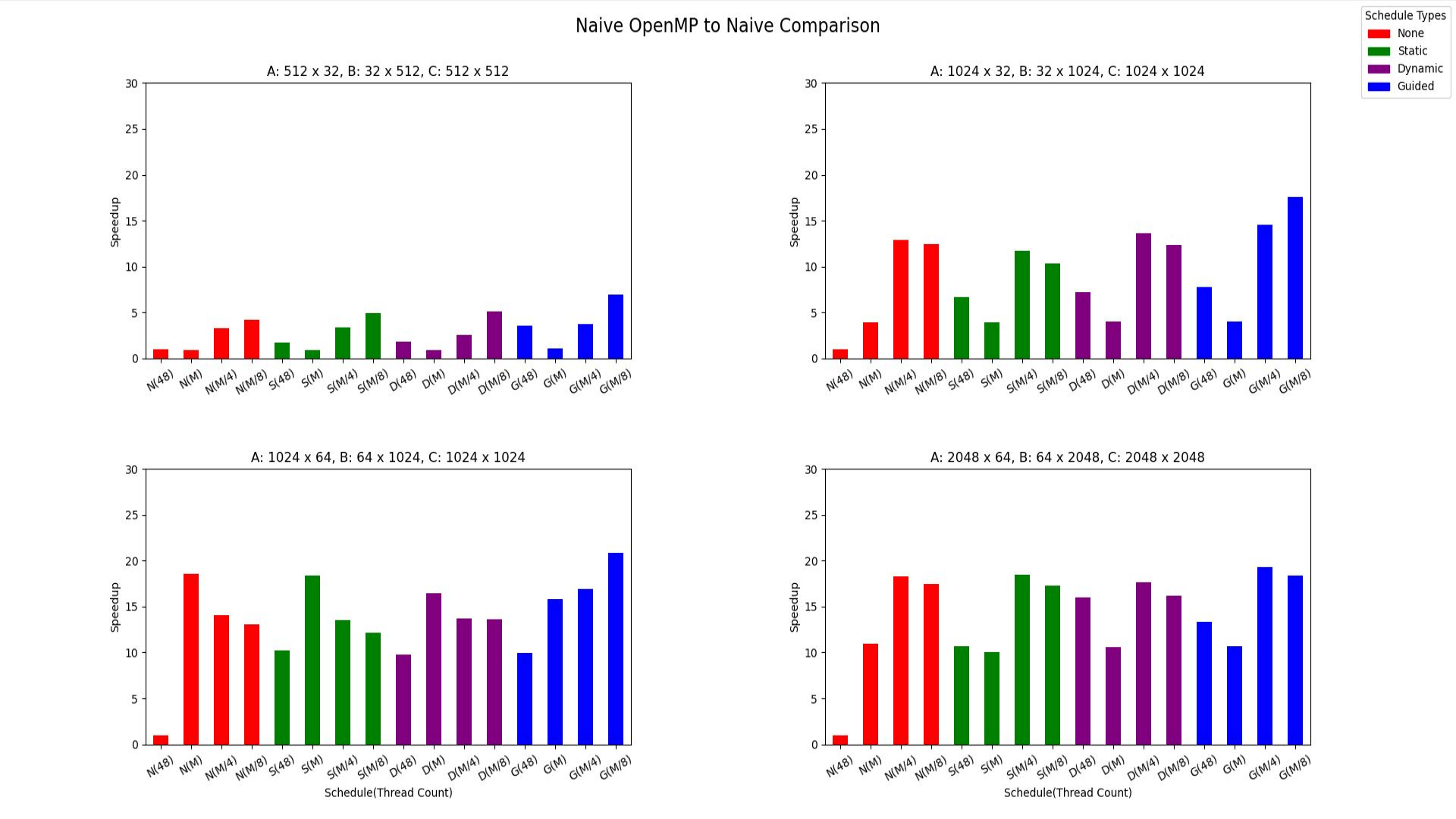
OpenMP Description

- OpenMP is a programming interface for shared-memory parallel programming.
- Code Implemented:
  - OpenMP Naive
  - OpenMP Transpose
- Matrices can be n x n or m x n
- Execution times are compared to the naiveMult from program 2

Schedule Types

- Static – Assigns each thread the set of loop iterations it needs to execute. Non-optimal if different iterations take different amounts of time.
- Dynamic – Assigns one iteration to each thread, when a thread finishes it's iteration it will be assigned an iteration that hasn't been executed yet. This method does include some overhead as threads must stop and receive a new value to use for the next iteration.
- Guided – This method is similar to dynamic scheduling except that the chunk size changes as the program runs. It begins with large chunks and adjusts to smaller chunk sizes if the workload is imbalanced.

Naive OpenMP to Naive Comparison



Bar Explanation

– 3 different scheduling types (static, dynamic, guided) and none for a base case.

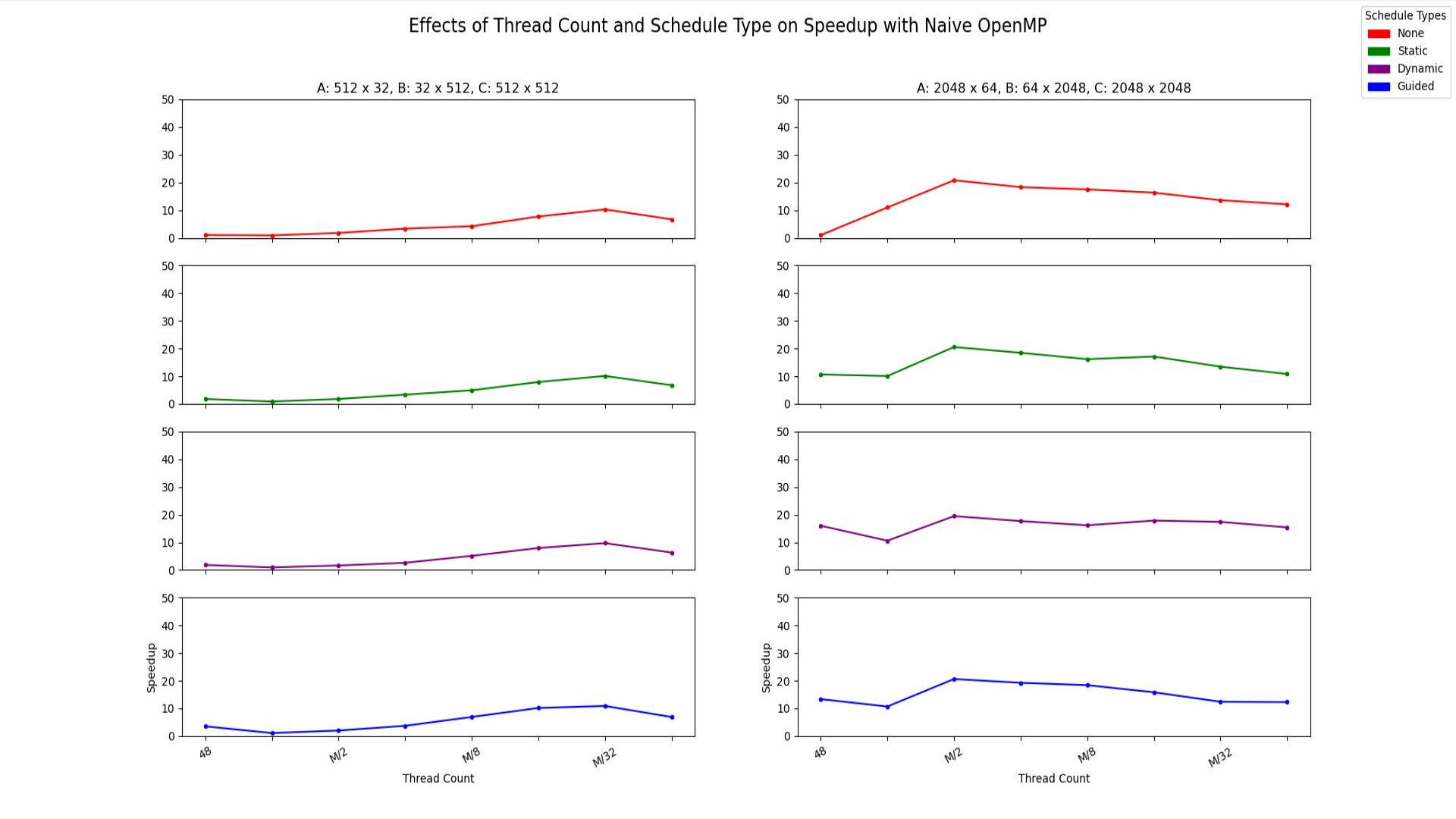
– 4 different thread counts for each scheduling type (default - 48, M, M/4, M/8). M being the most threads as thats 1 per row and M/8 being the least amount of threads besides the base case as thats 1 thread every 8 rows.

– Graph 1(top left): All scheduling types have the greatest speedups when using M/8 threads. Guided scheduling with M/8 threads shows greatest speedup.

– Graph 2(top right): All types other than guided show greatest speedup when using M/4 threads while guided shows greatest speedup when using M/8 threads. Guided scheduling with M/8 threads shows greatest overall speedup.

– Graph 3(bottom left): All types other than guided show greatest speedup when using M threads while guided shows greatest speedup when using M/8 threads. Guided scheduling with M/8 threads shows greatest overall speedup.

– Graph 4(bottom right): All types show highest speedup when using M/4 threads with guided having the highest overall.


Effects of Thread Count and Schedule Type on Speedup with Naive OpenMP
Schedule Types
None
Static
Dynamic
Guided
A: 512 x 32, B: 32 x 512, C: 512 x 512
A: 2048 x 64, B: 64 x 2048, C: 2048 x 2048
Speedup
Thread Count
48
M/2
M/8
M/32
0
10
20
30
40
50

Naive Line Explanation

– This graph compares speedups when using the same scheduling types as before but with 8 thread counts (default - 48, M, M/2, M/4, M/8, M/16, M/32, M/64) and only with the smallest and largest size of A, B, and C.

– The graphs on the left depict runs with small sizes for A, B, and C. These graphs show speedup to be greatest with a lower thread count, M/32.

– The graphs on the right depict runs with large sizes for A, B, and C. These graphs show speedup to be greatest with a higher thread count, M/2.

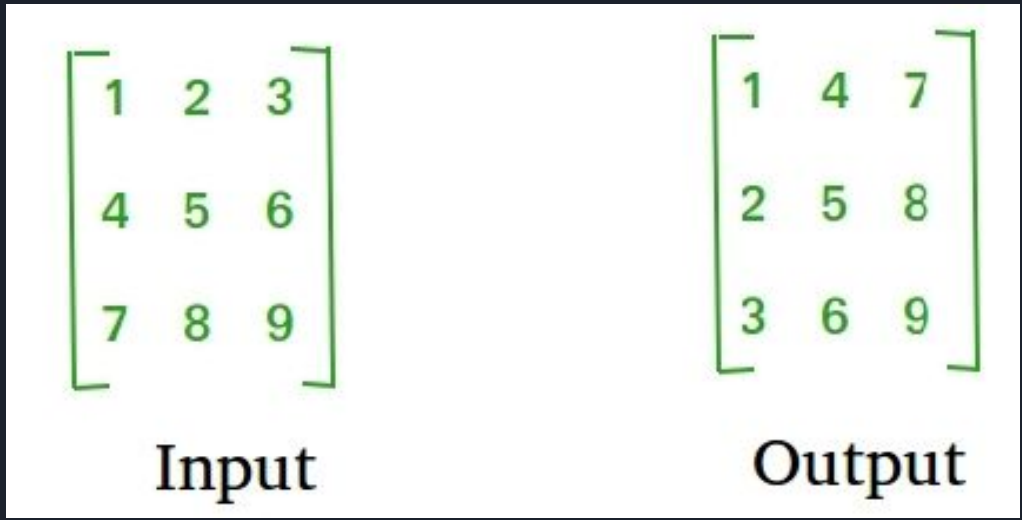
OpenMP Naive Code

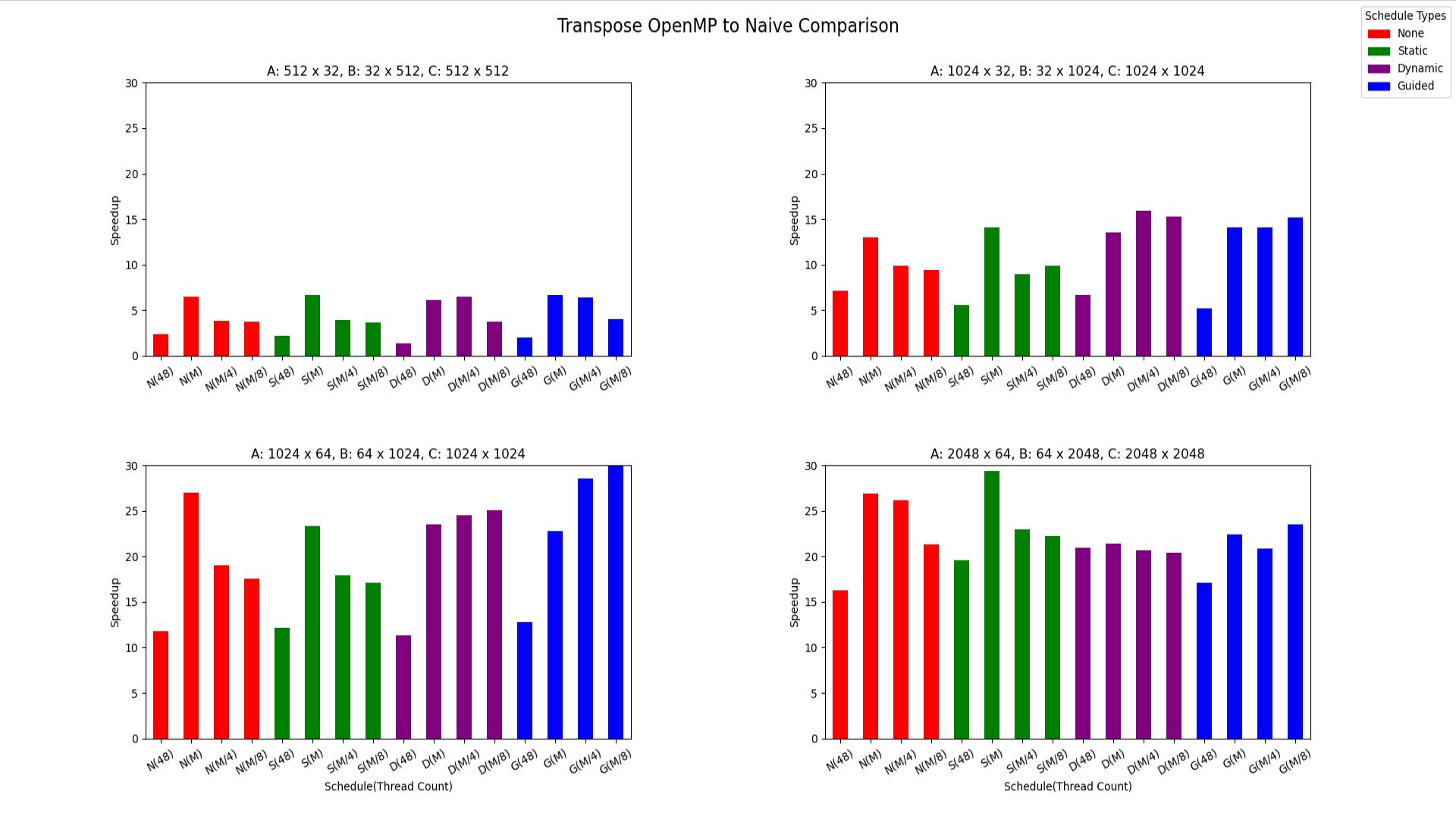
```
void openMPnaiveMult(float * A, float * B, float * C, uint64_t M, uint64_t N, uint64_t L)
{
    uint64_t i;

    auto naiveMultMP2 = [](uint64_t i, uint64_t j, float * A, float * B, float * C, uint64_t N, uint64_t L) {
        float accum = 0;
        uint64_t k;
        for (k = 0; k < L; k++)
        {
            accum += A[i*L+k]*B[k*N+j];
        }
        C[i*N+j] = accum;
    };

    auto naiveMultMP = [&naiveMultMP2](uint64_t i, float * A, float * B, float * C, uint64_t N, uint64_t L) {
        uint64_t j;
        for (j = 0; j < N; j++)
        {
            naiveMultMP2(i, j, A, B, C, N, L);
        }
    };

    #pragma omp parallel for schedule(guided) num_threads(M/8) //Modified this line for graphs
    for (i = 0; i < M; i++)
    {
        naiveMultMP(i, A, B, C, N, L);
    }
}
```

Transpose Description

– To transpose a matrix you have to flip the matrix over its diagonal. It switches the row and column indices of the matrix.

- Transposing the second matrix before a matrix multiply provides better cache performance




Transpose OpenMP to Naive Comparison
Schedule Types
None
Static
Dynamic
Guided
A: 512 x 32, B: 32 x 512, C: 512 x 512
A: 1024 x 32, B: 32 x 1024, C: 1024 x 1024
A: 1024 x 64, B: 64 x 1024, C: 1024 x 1024
A: 2048 x 64, B: 64 x 2048, C: 2048 x 2048
Speedup
Schedule(Thread Count)
N(48)
N(M)
N(M/4)
N(M/8)
S(48)
S(M)
S(M/4)
S(M/8)
D(48)
D(M)
D(M/4)
D(M/8)
G(48)
G(M)
G(M/4)
G(M/8)

Transpose Bar Explanation

– Scheduling types: None, Static, Dynamic, Guided.

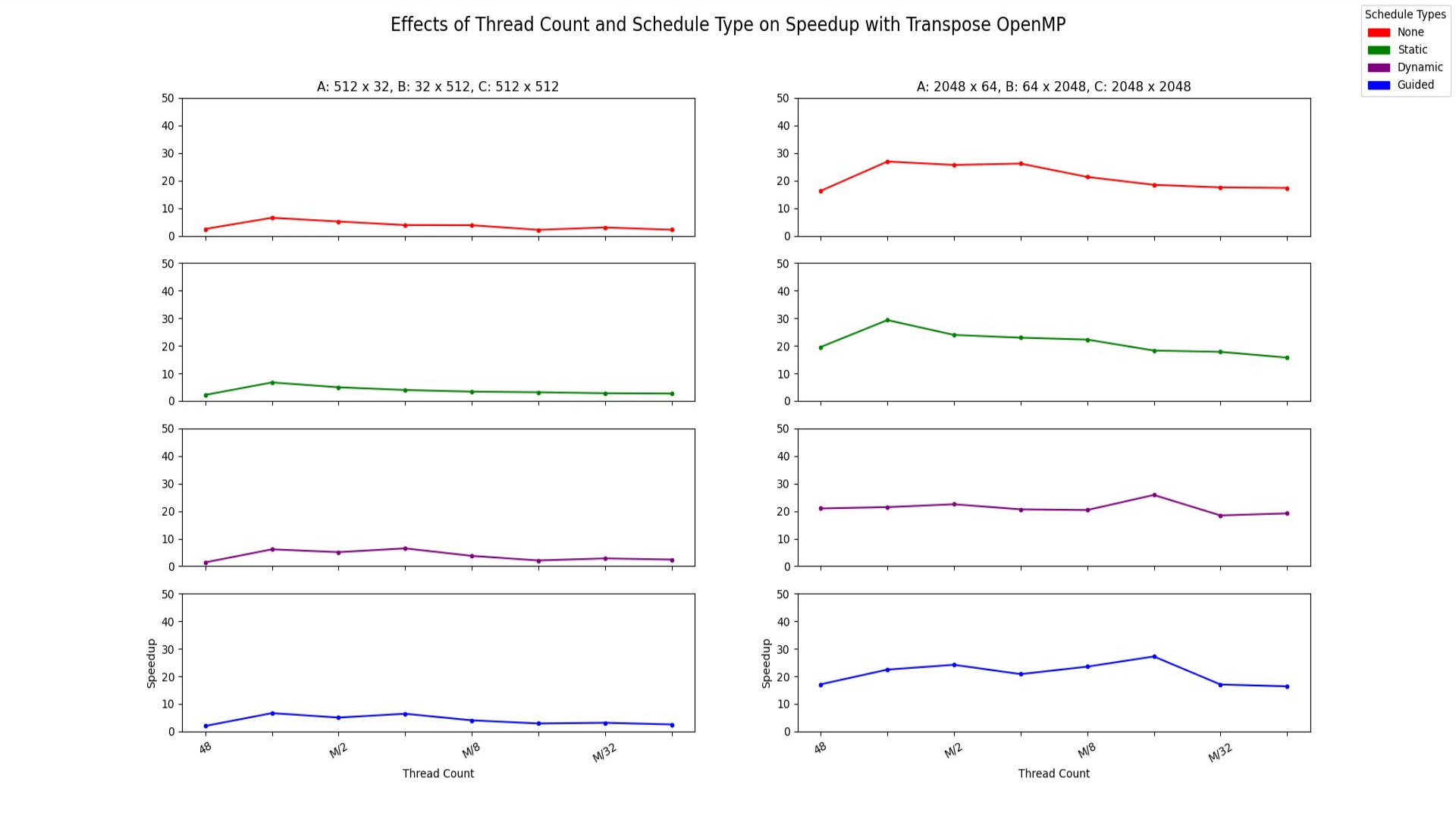
– Thread Counts: 48, M, M/4, M/8.

– Graph 1(top left): All scheduling types other than dynamic have highest speedups when using M threads while dynamic has highest speedup when using M/4 threads. All types have very similar speedups.

– Graph 2(top right): The base case None and static have highest speedups with M threads. Guided has the highest speedup when using M/8 threads and dynamic has the highest overall speedup when using M/4 threads.

– Graph 3(bottom left): The base case None and static have the highest speedups when using M threads while dynamic and guided have the highest speedups when using M/8 threads. Guided with M/8 threads has the highest overall speedup.

– Graph 4(bottom right): All graphs other than guided have the highest speedups when using M threads while guided has the highest speedup when using M/8 threads. Static with M threads has the highest overall speedup.


Effects of Thread Count and Schedule Type on Speedup with Transpose OpenMP
Schedule Types
None
Static
Dynamic
Guided
A: 512 x 32, B: 32 x 512, C: 512 x 512
A: 2048 x 64, B: 64 x 2048, C: 2048 x 2048
Speedup
Thread Count
48
M/2
M/8
M/32
0
10
20
30
40
50

Transpose Line Explanation

– Scheduling Types (None, Static, Dynamic, Guided)

– Thread Counts (default - 48, M, M/2, M/4, M/8, M/16, M/32, M/64)

– The 4 graphs on the left depict the same array sizes as in graph 1 in the last set of bar graphs while the 4 graphs on the right depict the same array sizes as in graph 4 in the last set of bar graphs.

– The graphs on the left which depict the smaller array sizes show the best speedups when using large thread counts, M.

– The graphs on the right all use larger array sizes. For no scheduling type and static scheduling type, speedup is greatest when using a large amount of threads, M. For the dynamic and guided scheduling types the speedup is greatest when using smaller amounts of threads, M/16.

OpenMP Transpose Code

```
void transposeAndMultMP(float * A, float * B, float * C, uint64_t M, uint64_t N, uint64_t L) {
    uint64_t i, k, j;
    float * Bt = new float[N*L];
    for (k = 0; k < L; k++)
        for (j = 0; j < N; j++)
            Bt[j*L+k] = B[k*N+j];

    auto transposeMP = [](uint64_t i, float * A, float * Bt, float * C, uint64_t N, uint64_t L) {
        uint64_t k, j;
        float accum;
        for (j = 0; j < N; j++)
        {
            accum = 0;
            for (k = 0; k < L; k++)
                accum += A[i*L+k]*Bt[j*L+k];
            C[i*N+j] = accum;
        }
    };

    #pragma omp parallel for schedule(guided) num_threads(M/8) //Modified this line for graphs
    for (i = 0; i < M; i++)
    {
        transposeMP(i, A, Bt, C, N, L);
    }
    delete Bt;
}
```

CUDA

- Allows for computing to be performed on NVIDIA GPUs
- Code Implemented:
  - Naive Kernel
  - Tiled Kernel
- Matrices are n x n
- Kernel execution times are compared to the naiveMult from program 2
- Kernel Launch Code

```
//Launch the appropriate kernel
if (which == NAIVE)
{
    dim3 block(TILE_SZ, TILE_SZ, 1);
    dim3 grid(ceil((float)width/TILE_SZ), ceil((float)width/TILE_SZ), 1);
    matrixMultNaiveKernel<<<grid, block>>>(gpuA, gpuB, gpuC, width);
} else {
    dim3 block(TILE_SZ, TILE_SZ, 1);
    dim3 grid(ceil((float)width/TILE_SZ), ceil((float)width/TILE_SZ), 1);
    matrixMultTiledKernel<<<grid, block>>>(gpuA, gpuB, gpuC, width);
}
```

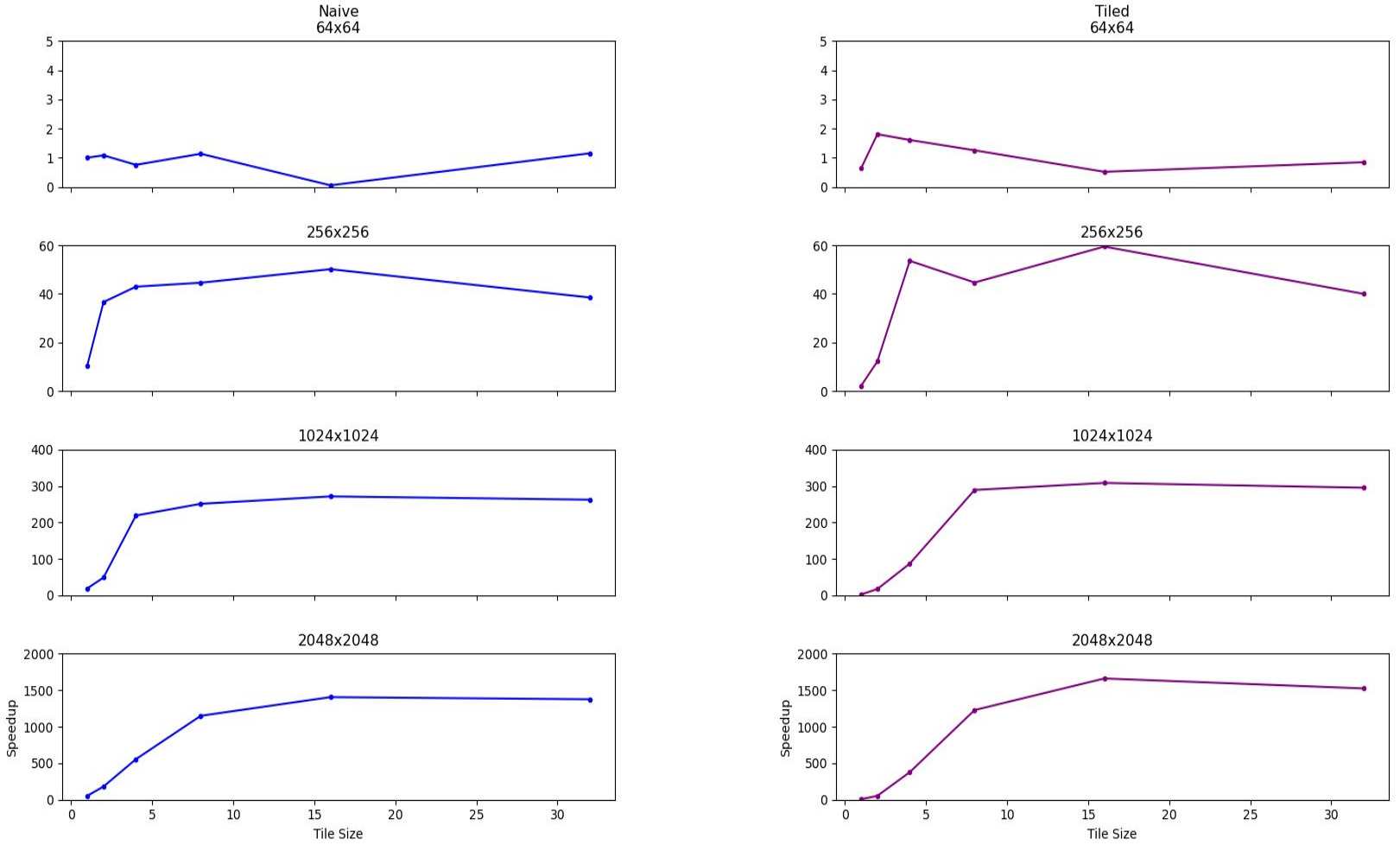
Naive Kernel Code

```
__global__
void matrixMultNaiveKernel(float * A, float * B, float * C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((row < width) && (col < width)) {
        float num = 0;
        for (int i = 0; i < width; ++i) {
            num += A[row * width + i] * B[i * width + col];
        }
        C[row*width+col] = num;
    }
}
```

Tiled Code

- This code uses shared memory to reduce the amount of traffic to global memory
- Each block will declare sA and sB since the scope of shared memory is a single block
- All threads in a block need access to the shared data so syncthreads is called to ensure all the threads have the entire shared array of floats
- Once threads have the array of floats, the result is calculated and put into the C matrix

```
__global__
void matrixMultTiledKernel(float * A, float * B, float * C, int width) {
    __shared__ float sA[TILE_SZ][TILE_SZ];
    __shared__ float sB[TILE_SZ][TILE_SZ];
    int i;

    int bx = blockIdx.x;
    int tx = threadIdx.x;
    int by = blockIdx.y;
    int ty = threadIdx.y;
    int row = by * TILE_SZ + ty;
    int col = bx * TILE_SZ + tx;
    float num = 0;

    for (i = 0; i < ceil(width/(float)TILE_SZ); ++i) {
        if ((row < width) && (i * TILE_SZ + tx) < width)
            sA[ty][tx] = A[row * width + i * TILE_SZ + tx];
        if ((i * TILE_SZ + ty) < width && (col < width))
            sB[ty][tx] = B[((i * TILE_SZ) + ty) * width + col];

        __syncthreads();

        for (int j = 0; j < TILE_SZ; ++j) {
            num += sA[ty][j] * sB[j][tx];
        }
        __syncthreads();
    }
    if ((row < width) && (col < width))
        C[row * width + col] = num;
}
```


Effects of Tile Size and Matrix Size on Speedup with Naive Cuda and Tiled Cuda
Naive
64x64
Tiled
64x64
256x256
256x256
1024x1024
1024x1024
2048x2048
2048x2048
Speedup
Speedup
Tile Size
Tile Size

Graph Explanation(tileSz, blockSz)

- Matrices are 64 x 64, 256 x 256, 1024 x 1024, 2048 x 2048
- Tile Sizes are 1, 2, 4, 8, 16, 32
  - First data point is tile size 1, second data point is 2, …

Run Configuration	Fastest Tile Size	Speedup
Naive 64x64	Tile Size: 32	Speedup: 1.150844
Tiled 64x64	Tile Size: 2	Speedup: 1.805928
Naive 256x256	Tile Size: 16	Speedup: 50.152760
Tiled 256x256	Tile Size: 16	Speedup: 59.447556
Naive 1024x1024	Tile Size: 16	Speedup: 271.453247
Tiled 1024x1024	Tile Size: 16	Speedup: 308.405426
Naive 2048x2048	Tile Size: 16	Speedup: 1405.791748
Tiled 2048x2048	Tile Size: 16	Speedup: 1661.550049


Speedups with Final Versions of OpenMP and CUDA Methods
Matrix Size: 64 x 64
Speedup
0
1
2
3
4
5
naiveOMP
transposeOMP
naiveCUDA
tiledCUDA
Matrix Size: 256 x 256
Speedup
0
20
40
60
80
100
naiveOMP
transposeOMP
naiveCUDA
tiledCUDA
Matrix Size: 1024 x 1024
Speedup
0
100
200
300
400
naiveOMP
transposeOMP
naiveCUDA
tiledCUDA
Matrix Size: 2048 x 2048
Speedup
0
500
1000
1500
2000
naiveOMP
transposeOMP
naiveCUDA
tiledCUDA

Final Version Graph Explanation

- Block size = 16 and Tile Size = 16
- Matrix Size 64 x 64
  - naiveOMP Speedup: 1.23234
  - transposeOMP Speedup: 1.117732
  - naiveCuda Speedup: 0.053063
  - tiledCuda Speedup: 1.805928
- Matrix Size 256 x 256
  - naiveOMP Speedup: 13.4158
  - transposeOMP Speedup: 14.34752
  - naiveCuda Speedup: 50.152760
  - tiledCuda Speedup: 59.447556
- Matrix Size 1024 x 1024
  - naiveOMP Speedup: 15.90982
  - transposeOMP Speedup: 35.8776
  - naiveCuda Speedup: 271.453247
  - tiledCuda Speedup: 308.405426
- Matrix Size 2048 x 2048
  - naiveOMP Speedup: 16.41258
  - transposeOMP Speedup: 30.36528
  - naiveCuda Speedup: 1405.791748
  - tiledCuda Speedup: 1661.550049

Future Work / Conclusion

Future Work:

- Implement matrix multiply on MPI machine
- Implement M x N matrices on Cuda machine
- Implement blocking technique

Conclusion

- Cuda provides much better performance but isn't as easily accessible due to the requirement of an NVIDIA GPU
- OpenMP on the other hand is much more accessible but offers less performance increase than Cuda
- OpenMP provides greater speedup times with smaller matrix sizes while CUDA provides greater speedup times with larger matrix sizes.