Hybrid MPI + GPU Implementation of the SUMMA Algorithm

Arin Idhant Varun Satheesh

APMA 2822B – Brown University

Abstract

We present a hybrid distributed and GPU-accelerated implementation of the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [1]. Our approach combines MPI-based 2D process decomposition with GPU offloading using HIP to efficiently compute large dense matrix products. We evaluate performance across multiple matrix sizes and analyze scalability using a roofline performance model.

1 Introduction

Large-scale matrix multiplication can be slow, and naive approaches to GPU parallelization can suffer from becoming memory-bound as the size of the matrices increases.

A naive CPU implementation involves a triple-nested for loop, which yields an $\mathcal{O}(N^3)$ runtime. This scales very poorly for matrices of larger sizes. In our experimentation, for a square matrix multiplication of size N, we observed that the CPU implementation becomes impractically slow for $N > 1000$, making further evaluation infeasible within reasonable runtime constraints.

In this project, we explore strategies for parallelizing matrix multiplication with increasing levels of complexity to address both the computational and memory-bound challenges of performing this operation at scale. We use a naive CPU implementation as a baseline and accelerate computation using GPU kernels. We first employ a one-dimensional decomposition strategy and then extend this approach to a two-dimensional decomposition using the Scalable Universal Matrix Multiplication Algorithm (SUMMA). We use HIP for launching and executing GPU kernels, and MPI for the inter-process communication required by the SUMMA algorithm.

Our final solution shows how hybrid parallelism can be used to improve scalability and also improve the efficiency of the algorithm.

2 Background: The SUMMA Algorithm

2.1 Motivation

Consider the matrix product $C = A \times B$. A one-dimensional decomposition approach for this problem involves decomposing the matrix A into horizontal bands, distributing these bands across

the available processes, and computing row-wise dot products with the appropriate columns of B. In this approach, each process computes a subset of rows of C.

For each process to compute its assigned band of C, it must have access to the entire matrix B. This is because, for each row of A, the computation of the final product requires dot products with every column of B. As a result, a 1D decomposition necessitates broadcasting the full matrix B to all P processes.

This strategy incurs significant communication and memory overhead. The communication cost scales as $\mathcal{O}(N^2)$ per rank, since the full $N \times N$ matrix B must be communicated to every process. Additionally, memory quickly becomes a bottleneck, as each rank must store the entire matrix B of size $\mathcal{O}(N^2)$, along with its assigned N/P rows of matrix A. This approach effectively results in an all-gather of matrix B across many nodes, which limits scalability as the number of processes and matrix size increase.

2.2 2D Decomposition

The main idea behind SUMMA is that instead of decomposing matrix A into horizontal bands, we decompose both matrix A and matrix B into two-dimensional tiles. As a result, instead of having each process compute a horizontal band of C, each process computes a tile of C. A diagram illustrating this tiling distribution is shown in the figure below. Each process initially owns a tile of A, a tile of B, and is responsible for computing the corresponding tile of C.

This two-dimensional decomposition reduces both the communication and memory requirements per process. Rather than storing entire rows or columns of the input matrices, each process only stores local tiles whose dimensions scale with the square root of the total number of processes.

2.3 Broadcasting and Communication

The critical issue with this approach is that no process initially has all the data required to fully compute its tile of C. To compute a finalized entry of C, the full row of A and the full column of B must be available. As each process only stores local tiles, communication between processes becomes necessary.

Each process seeks to iteratively compute its final tile of the matrix C. Through synchronized communication, processes share their native tiles of A and B with other processes that require them at a specific iteration. The final result is computed over q iterations, where q is the number of tiles along one dimension of the matrix.

At each iteration k, one process broadcasts its tile $A_{i,k}$ across all processes that hold tiles in the same row. Similarly, one process broadcasts its tile $B_{k,j}$ across all processes that hold tiles in the same column. After these broadcasts, each process has the necessary tiles to compute the next partial contribution to its local tile of C.

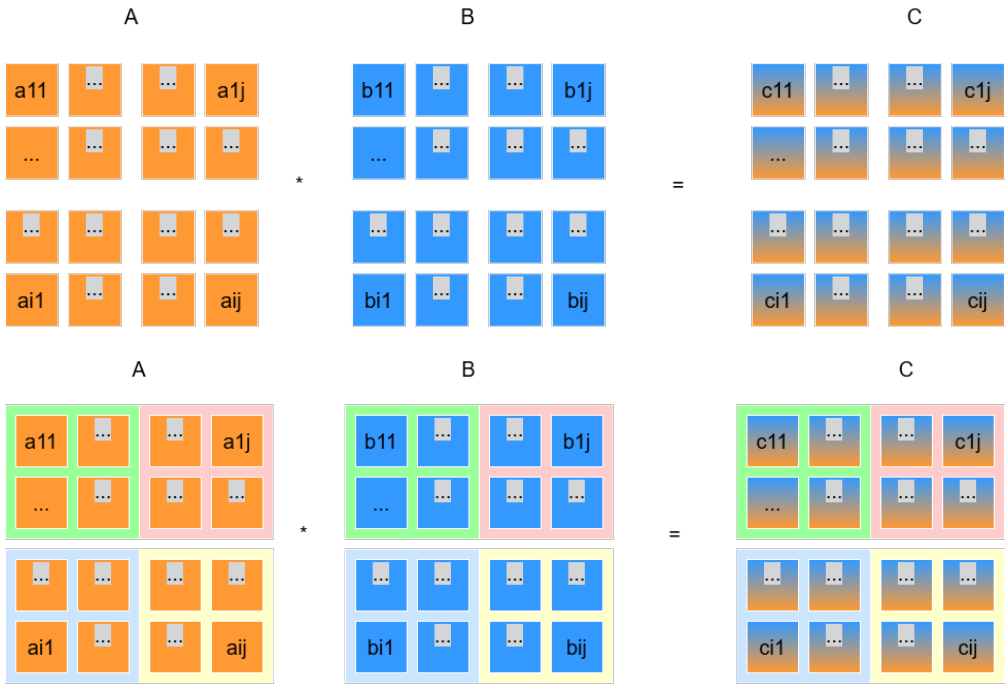

Figure 1: Process grid and tile distribution used in the SUMMA algorithm.

2.4 Iterative Calculation

At the core of SUMMA is an iterative accumulation of partial results. At each iteration, after receiving the appropriate tiles of A and B, each process computes a matrix product between these tiles and adds the resulting partial product to its local tile of C. In this way, each process incrementally builds its portion of the output matrix.

More formally, consider the computation of a tile $C_{i,j}$. This tile can be expressed as a sum over intermediate tile indices k:

$$C_{i,j} = \sum_{k=0}^{q-1} A_{i,k} \times B_{k,j},$$

where q is the number of tiles along each matrix dimension. At iteration k, the product $A_{i,k} \times B_{k,j}$ represents one term in the global row–column dot product. Each process computes this term locally and accumulates it into its tile $C_{i,j}$. Over all iterations, the local tile progressively accumulates all required partial sums, resulting in the final value of $C_{i,j}$.

We handle this part of the calculation using HIP Kernels to accelerate matrix multiplication on the GPU.

3 Parallel Design and MPI Implementation

We begin by initializing a two-dimensional Cartesian grid of $q \times q$ MPI processes, which serves as the logical process topology for the algorithm. Based on each process's position in this grid, the

input matrices A and B are decomposed into tiles and distributed across the ranks.

The computation then proceeds iteratively over q steps. At each iteration, processes broadcast their local tiles of A and B to other processes in the same row and column, respectively. Using the received tiles, each rank computes a partial matrix multiplication and accumulates the result into its local tile of C. After all iterations are complete, the local tiles of C are gathered back to rank 0 to assemble the final output matrix.

3.1 Process Grid Construction

To efficiently perform distributed matrix multiplication, we organize the $p = q^2$ MPI processes into a two-dimensional Cartesian grid of size $q \times q$. This allows processes to communicate efficiently along rows and columns during the SUMMA algorithm. The process grid is created using `MPI_Cart_create`:

```
int dims[2] = {q, q};
int periods[2] = {0, 0};
MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &cart_comm);
```

Each process can then query its coordinates $(\text{my\_row}, \text{my\_col})$ in the grid:

```
int coords[2];
MPI_Cart_coords(cart_comm, world_rank, 2, coords);
int my_row = coords[0], my_col = coords[1];
```

We then create row and column communicators using MPI_Cart_sub:

```
// create row and column communicators
int remain_row[2] = {0, 1};
int remain_col[2] = {1, 0};
MPI_Comm row_comm, col_comm;
MPI_Cart_sub(cart_comm, remain_row, &row_comm);
MPI_Cart_sub(cart_comm, remain_col, &col_comm);
```

This will later allow processes to broadcast to all processes in the same row and all processes in the same column in the process grid.

3.2 Data Distribution

Each process stores local sub-blocks of A, B, and C with dimensions $n_{\text{local}} = N/q$, $k_{\text{local}} = K/q$, and $m_{\text{local}} = M/q$:

```
std::vector<double> A_local(n_local*k_local);
std::vector<double> B_local(k_local*m_local);
std::vector<double> C_local(n_local*m_local, 0.0);
```

Rank 0 initializes the full matrices and distributes blocks to each process. We do so using MPI_Send:

```
MPI_Send(tmp.data(), tmp.size(), MPI_DOUBLE, dest_rank, tag, MPI_COMM_WORLD);
```

The other ranks in the code receive these tiles using MPI_Recv

```
MPI_Recv(A_local.data(), n_local*k_local, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
MPI_Recv(B_local.data(), k_local*m_local, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
```

This setup ensures each process starts SUMMA with its local sub-matrices and a zero-initialized C_{local} for accumulating results.

3.3 SUMMA Iteration and Communication Pattern

The SUMMA algorithm proceeds iteratively over q steps. At each step, processes broadcast the relevant panel of A across their row and the relevant panel of B across their column. Using the received panels, each process performs a local matrix multiplication and accumulates the result into C_{local}.

```
// buffers for broadcast
std::vector<double> A_panel(n_local*k_local);
std::vector<double> B_panel(k_local*m_local);

for (int k = 0; k < q; ++k) {
    if (my_col == k) A_panel = A_local;
    MPI_Bcast(A_panel.data(), ..., root_rank_in_row, row_comm);

    if (my_row == k) B_panel = B_local;
    MPI_Bcast(B_panel.data(), ..., root_rank_in_col, col_comm);

    local_gemm(A_panel.data(), B_panel.data(), C_local.data(),
               n_local, k_local, m_local);
}
```

This loop ensures that each process receives the correct tiles for the current iteration, performs the partial multiplication, and accumulates results into its local block of C. The local gemm is replaced with a GPU implementation in the following section to achieve a hybrid MPI + GPU parallelization.

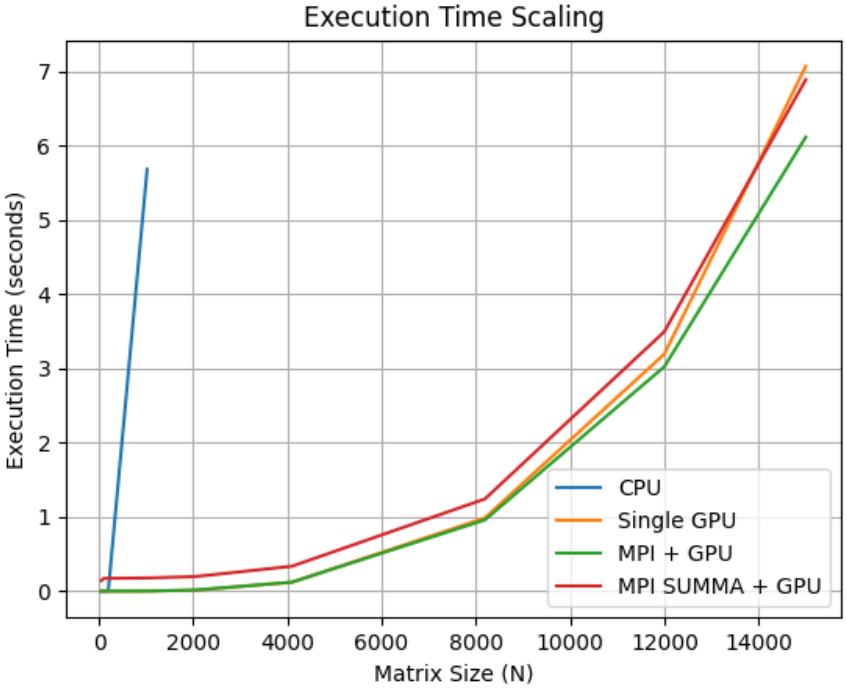

Figure 2: Execution time scaling with larger matrix sizes. (CPU is stopped early due to unreasonable run-times)

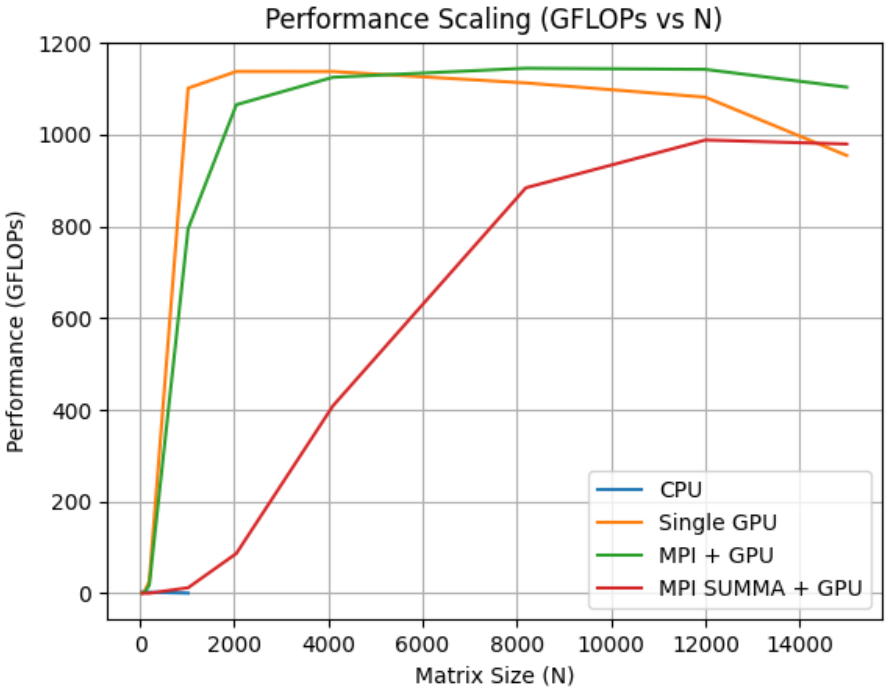

Figure 3: Comparing GFLops performance with increasing matrix sizes

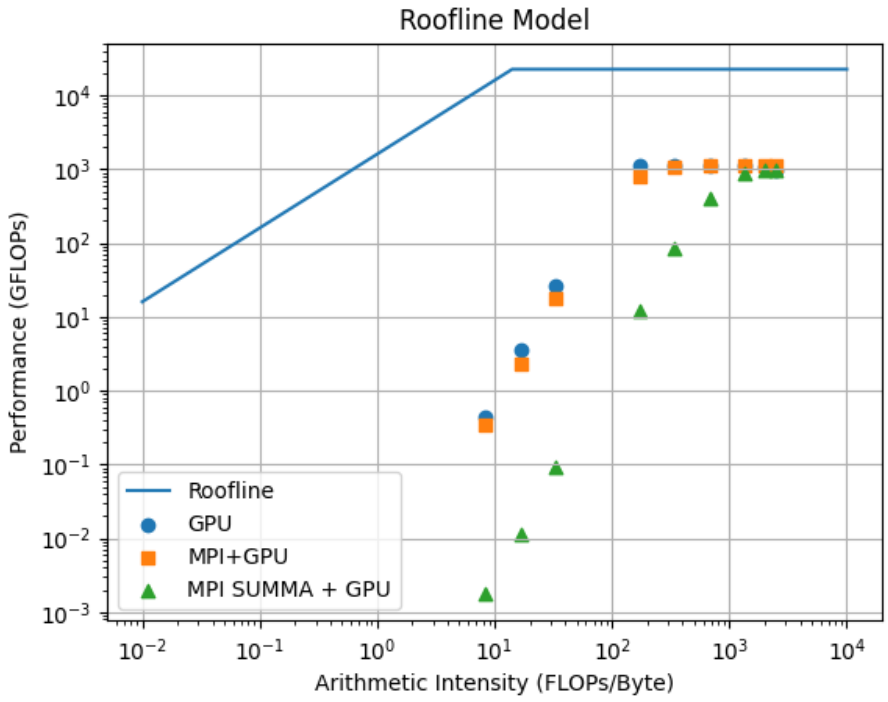

Figure 4: Roofline graph

4 GPU Acceleration

To accelerate the local matrix multiplications performed inside SUMMA, we offload the per-iteration update

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k}B_{k,j}$$

to the GPU using HIP. Conceptually, MPI handles the distributed data movement (broadcasting the required panels of A and B), and the GPU performs the compute-heavy local GEMM on each rank.

4.1 Rank-to-GPU Mapping

Each MPI rank selects a GPU device in a round-robin fashion to enable concurrent GPU use on multi-GPU nodes:

```
hipGetDeviceCount(&numDevices);
hipSetDevice(world_rank % numDevices);
```

This mapping preserves correctness for a single GPU and scales naturally when multiple GPUs are present.

4.2 Kernel Design and Launch Configuration

We use a baseline HIP kernel where each thread computes one element of C_{local}:

- Thread indices map to (row, col) of the local output tile.
- Each thread loops over the shared inner dimension k_{local} and accumulates a dot product.

We launch a 2D grid of 16×16 thread blocks:

```
dim3 threads(BLOCK, BLOCK);            // BLOCK = 16
dim3 grid((m_local + BLOCK - 1)/BLOCK,
          (n_local + BLOCK - 1)/BLOCK);
matmulKernel<<<grid, threads>>>(...);
```

This kernel is intentionally simple (no shared-memory tiling). While it does not maximize reuse of A and B values within a block, it provides a correct and portable baseline for integrating GPU compute into SUMMA.

4.3 Data Movement Strategy

At each SUMMA iteration k, ranks first obtain panels $A_{i,k}$ and $B_{k,j}$ via MPI broadcasts into host buffers. The GPU path then:

1. allocates device buffers for A, B, and C,
2. copies $A_{i,k}$, $B_{k,j}$, and the current C_{local} to the device,
3. launches the kernel and synchronizes,
4. copies the updated C_{local} back to the host.

This per-iteration allocation and host–device movement increases overhead, especially for smaller local tiles. However, it clearly isolates the benefit of GPU acceleration for the compute-dominant part of the algorithm, and it explains the behavior observed in the scaling plots: performance ramps up sharply once the local GEMM becomes large enough to amortize these fixed overheads.

4.4 GPU Acceleration in Distributed Settings

We evaluate three configurations: (i) single GPU GEMM, (ii) MPI + GPU (distributed with GPU compute), and (iii) MPI SUMMA + GPU (full 2D SUMMA communication pattern + GPU compute). The key difference between (ii) and (iii) is the communication structure: SUMMA introduces synchronized row/column broadcasts at every iteration, which increases latency sensitivity at smaller sizes and shifts some runs downward on the roofline plot. As N grows, the GPU compute increasingly dominates, and the SUMMA configuration approaches the performance of the other GPU-based approaches.

5 Performance Evaluation and Roofline Analysis

We evaluate our implementations using (1) a roofline model, (2) throughput scaling (GFLOPs vs. matrix size), and (3) execution time scaling. We compare four baselines: CPU, Single GPU, MPI + GPU, and MPI SUMMA + GPU.

5.1 Timing Methodology

We measure the end-to-end runtime of the core multiplication loop and report the maximum across ranks, since the slowest rank determines global progress. We synchronize before and after timing using `MPI_Barrier` and reduce with `MPI_MAX` to obtain a single global runtime.

5.2 Roofline Interpretation

Figure 4 plots achieved performance against arithmetic intensity. The diagonal segment corresponds to a bandwidth-limited regime, while the flat ceiling corresponds to a compute-limited regime.

The Single GPU and MPI+GPU points rise quickly with arithmetic intensity and cluster near $\sim 10^3$ GFLOP/s at high intensity, indicating that large problem sizes become compute-dominated on the GPU.

In contrast, MPI SUMMA + GPU starts substantially lower at small arithmetic intensity and gradually approaches the same high-performance cluster at larger intensities. This separation reflects additional overheads that are not present in single-GPU execution: repeated synchronized broadcasts (row/column collectives), per-iteration synchronization, and smaller effective compute per communication step at low N. As arithmetic intensity increases, the local GEMM work grows faster than the communication and synchronization costs, so MPI SUMMA + GPU transitions upward toward the compute roof.

5.3 Throughput Scaling (GFLOPs vs. N)

Figure 3 shows performance scaling with matrix size.

CPU: The CPU baseline remains near zero on this scale and becomes impractical beyond modest sizes, consistent with the $O(N^3)$ growth and limited single-node throughput.

Single GPU and MPI+GPU: Both ramp sharply and plateau around $\sim$ 1100–1150 GFLOP/s for mid-to-large N, indicating GPU saturation. MPI+GPU tracks single-GPU closely, showing that distributed overhead is relatively small once computation dominates.

MPI SUMMA + GPU: This curve increases more gradually. At smaller sizes it is significantly below the GPU plateau, which is consistent with the SUMMA loop performing q iterations of broadcasts and synchronization, where fixed per-iteration overhead dominates when tiles are small. As N increases, MPI SUMMA + GPU climbs and eventually approaches $\sim$ 1000 GFLOP/s,

demonstrating that the algorithm becomes compute-dominated and that SUMMA's 2D decomposition scales effectively for larger matrices.

5.4 Execution Time Scaling

Figure 2 plots runtime versus N.

Single GPU and MPI+GPU remain close across all large N, with MPI+GPU achieving the lowest execution time at the largest sizes shown. MPI SUMMA + GPU is consistently slower than MPI+GPU and single GPU, especially at smaller-to-mid sizes, reflecting the cost of repeated per-iteration broadcasts and synchronization. However, the gap narrows at larger N, where the dominant cost becomes the GPU GEMM itself.

5.5 Key Takeaways and Bottlenecks

Taken together, the figures support three main conclusions:

- GPU acceleration delivers the dominant performance improvement, reaching $\sim 10^3$ GFLOP/s at large N.
- The MPI+GPU configuration sustains near single-GPU throughput at large sizes, indicating communication overhead is well amortized.
- MPI SUMMA + GPU is overhead-limited at small sizes due to per-iteration collectives, but trends upward and approaches the compute-dominated regime as N increases.

These results also indicate clear optimization opportunities: reducing per-iteration overhead (persistent device buffers, keeping C_{local} on device across iterations, and overlapping broadcasts with device compute) and improving kernel arithmetic intensity (shared-memory tiling) would shift MPI SUMMA + GPU upward on the roofline and reduce its time scaling gap.

6 Limitations and Future Work

7 Conclusion

References

[1] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Department of Computer Sciences, University of Texas at Austin and Scalable Concurrent Programming Laboratory, California Institute of Technology, Austin, Texas and Pasadena, California, 2008. `https://www.netlib.org/lapack/lawnspdf/lawn96.pdf`.