# A Comprehensive Literature Review of High-Performance Matrix Multiplication: From Algorithmic Theory to Heterogeneous Cluster-Scale Implementation

Matrix multiplication is a fundamental building block in computational mathematics, scientific computing, graphics, and machine learning.[1] The naive algorithm, with its $O(n^3)$ asymptotic complexity [3], is often the first implementation a programmer learns, but it is far from the state-of-the-art. Achieving high performance in matrix multiplication is a canonical problem in High-Performance Computing (HPC), requiring a "full-stack" optimization approach. This involves co-designing across every layer of a modern compute system: from abstract mathematical algorithms and data structures, through compiler optimizations and cache-aware code, to single-node parallel programming on multi-core CPUs (OpenMP) and many-core GPUs (CUDA), and finally to cluster-scale parallelism using distributed-memory models (MPI).

This review synthesizes the extensive body of literature on this topic, tracing the optimization path from theoretical foundations to the complex, heterogeneous systems that power modern supercomputers.

## I. Algorithmic Foundations: From $O(n^3)$ to AI-Driven Discovery

The choice of algorithm provides the theoretical upper bound on performance. While the $O(n^3)$ baseline is ubiquitous, research has long focused on algorithms with lower asymptotic complexity, creating a divide between what is theoretically possible and what is practically useful.

## The $O(n^3)$ Baseline: A Foundation for Optimization

The standard, three-loop matrix multiplication algorithm (in its ijk form) is the baseline for all practical, high-performance libraries. While its $O(n^3)$ complexity is not asymptotically optimal, its operational simplicity and regular memory access patterns make it an ideal *target* for the hardware-level optimizations discussed in Section II. Its primary flaw, when implemented naively, is not its arithmetic cost but its catastrophic memory performance, incurring $O(n^3)$ cache misses. All practical $O(n^3)$ implementations are, therefore, *blocked* (or tiled) to improve cache locality.

## Sub-Cubic Algorithms: Strassen's Method ($O(n^{2.807})$)

In 1969, Volker Strassen demonstrated that the $O(n^3)$ complexity was not optimal.[3] His algorithm proved that a $2 \times 2$ matrix multiplication, which seemingly requires 8 multiplications and 4 additions, could be accomplished with only 7 multiplications and 18 additions/subtractions.[1]

When applied recursively, this substitution yields an algorithm with an asymptotic complexity of $O(n^{\log_2 7})$ or approximately $O(n^{2.807})$.[3]

### Practical Implementation of Strassen's Algorithm

For decades, Strassen's algorithm was one of the few sub-cubic algorithms considered practical for real-world use.[5] However, it is never implemented in its pure recursive form. Instead, practical implementations are *hybrid* [1]:

1. **Recursive Decomposition:** The algorithm recursively divides the matrices into four equal-sized sub-matrices.[1]
2. **Cutoff Criterion:** This recursion continues until the sub-matrix size $n$ reaches a "crossover point" or "cutoff criterion".[6]
3. **Baseline Kernel:** Below this threshold, the algorithm switches from Strassen's recursive calls to a highly-optimized, $O(n^3)$ DGEMM (Double-precision General Matrix Multiplication) routine, such as one provided by a vendor's BLAS (Basic Linear Algebra

Subprograms) library.[6]

The crossover point is hardware- and library-dependent. It is determined empirically by finding the matrix size where the overhead of Strassen's 18 additions and complex data management [1] becomes more expensive than the 8th multiplication in a standard DGEMM.[6] A 1996 study on an IBM RS/6000 found this crossover to be $m=176$.[6] More recent FPGA implementations report crossovers as low as $n=256$ [7], while other analyses suggest practical use for $n > 100$ [5] or even $n > 1,000$.[8]

Practical codes must also handle matrices that are not powers of two, often using a technique called "dynamic peeling with rank-one updates".[6] Furthermore, Strassen's algorithm has a higher memory footprint due to the need for temporary matrices to store the intermediate $M_1$ through $M_7$ results.[1]

### Numerical Stability

The long-held assumption that Strassen's algorithm is numerically unstable has been largely revised. While its error bounds are slightly weaker than the naive algorithm—it is not component-wise stable—it has been shown to be norm-wise stable and "stable enough" for many practical applications.[5]

## "Galactic" Algorithms: Theoretical Triumphs, Practical Impracticality

Following Strassen, a series of asymptotically faster algorithms were discovered, starting with the Coppersmith-Winograd algorithm, which achieved a complexity of $O(n^{2.375477})$ [9], and subsequent improvements.[10]

These algorithms are known as "galactic algorithms" because they are *never* used in practice.[9] Their implementation complexity is described as "enormous" [11], and the hidden constant factor in the $O$-notation is so large that the crossover point for any performance gain would be for matrices of "astronomical" or "impractical" size.[8] They remain purely theoretical achievements.

## The Modern Era: AlphaTensor and AI-Driven Discovery

The clear, decades-long dichotomy between the "practical" Strassen and the "theoretical" Coppersmith-Winograd has recently been disrupted by AI-driven algorithm discovery. DeepMind's AlphaTensor, an agent based on the AlphaZero reinforcement learning framework, reframed algorithm discovery as a single-player game.[2]

AlphaTensor's achievements are twofold:

1. **Discovering Novel Asymptotic Complexity:** For specific cases, such as $4 \times 4$ matrix multiplication over a finite field ($\mathbb{Z}_2$), AlphaTensor discovered an algorithm using 47 multiplications, outperforming Strassen's two-level application (which uses $7^2 = 49$). This yields a practical algorithm with complexity $O(n^{2.778})$.[15]
2. **Discovering Hardware-Specific Practical Algorithms:** This is the more profound contribution. AlphaTensor can be trained to optimize for *actual runtime on specific hardware*, such as an NVIDIA V100 GPU or a Google TPU.[14] In this mode, it discovered algorithms for standard arithmetic that, while having the *same asymptotic complexity* as Strassen ($O(n^{2.807})$), are practically 10-20% faster on those specific hardware targets.[14]

This represents a paradigm shift. These AI-found algorithms are often "denser" (containing more additions) than Strassen's, but are structured in a way that allows the target compiler (e.g., XLA) to perform operation fusion more efficiently.[14] The "optimal" algorithm is thus no longer a single, abstract mathematical formula, but a *hardware-dependent artifact*. While these algorithms are not yet standard replacements in libraries like cuBLAS or MKL [17], they prove that hardware-specific algorithm generation is a viable and superior path forward.

The following table summarizes the practical landscape of these algorithmic choices.

**Table 1: Comparison of Dense Matrix Multiplication Algorithms: Theory vs. Practice**

| Algorithm | Asymptotic Complexity | Practical Crossover Point | Key Limitation / Implementation Constraint |
|---|---|---|---|
| **Naive (Blocked)** | $O(n^3)$ | N/A (Baseline) | Memory-bound; performance is entirely dependent on cache blocking.[18] |

| | | | |
|---|---|---|---|
| **Strassen** | $O(n^{2.807})$ [3] | Hardware-dependent (e.g., $n \approx 100-2000$) [5, 6, 8] | Requires hybrid implementation, temporary memory overhead, and careful stability checks.[1] |
| **Coppersmith-Winograd** | $O(n^{2.37...})$ [9] | "Astronomical" / Not feasible [8, 11, 12] | Extreme implementation complexity; purely theoretical.[9] |
| **AlphaTensor (Hardware-Tuned)** | $O(n^{2.807})$ (or $O(n^{2.778})$) [15] | $n > 1$ (for small kernels) | Optimized for *specific hardware* (e.g., V100 GPU); may be "denser" and less human-readable.[14] |

# II. Single-Node CPU Optimization: Mastering the Memory Hierarchy

Given that even Strassen's algorithm relies on a highly-optimized $O(n^3)$ kernel [6], the vast majority of HPC research has focused on optimizing this $O(n^3)$ computation. On modern CPUs, matrix multiplication is almost always *memory-bound*. The time required to perform a floating-point operation is negligible compared to the time it takes to fetch the data from main memory (the "memory wall").[11]

Therefore, performance is not dictated by arithmetic speed, but by *data locality*—the ability to reuse data once it has been loaded into the CPU's fast cache memory.[18] The optimization of gemm on a CPU is a *fractal-like* problem: the same principle of "blocking" is applied recursively at every level of the memory hierarchy.

## Cache-Aware Techniques: Tiling and Loop Interchange

The most fundamental cache optimization is **tiling** (also known as **blocking**).[18] The $O(n^3)$ computation is broken into a sequence of smaller $O(n^3)$ computations on "tiles" (or blocks) of the matrices that are small enough to fit into a specific level of the CPU cache (e.g., L1, L2, or L3).[20] By loading a tile of $A$ and a tile of $B$ into cache, they can be reused $B$ times (where $B$ is the tile dimension) to compute a tile of $C$. This exploits *temporal locality* and drastically reduces the number of slow main-memory accesses from $O(n^3)$ to $O(n^3 / B)$.[21]

This is often combined with **loop interchange**, a compiler transformation that reorders the i, j, k loops.[18] In a row-major language like C, the naive i, j, k order has poor *spatial locality* on matrix $B$, as it accesses its data column-wise, striding across memory. The i, k, j order, by contrast, is often superior, as the inner loop j runs contiguously over the rows of $C$ and $B$ (if $B$ were transposed) or $A$ and $C$ (in this configuration).[22]

## Data Layout: Row-Major, Column-Major, and Recursive

The physical layout of data in memory is critical. C/C++ use **row-major** order, where A[i][j+1] follows A[i][j] in memory.[24] Fortran, MATLAB, and traditional BLAS libraries use **column-major** order, where A[i+1][j] follows A[i][j].[25]

To achieve high performance, the loop order must match the data layout. In C, the (i, k, j) loop order is cache-friendly for A and C because they are accessed row-wise, maximizing spatial locality.[21] To avoid the cache-unfriendly access to $B$, high-performance libraries will often "pack" (transpose) matrix $B$ into a temporary, contiguous block of memory in an ideal format before computation begins.

More advanced are **recursive array layouts**, such as those based on Z-Morton (Peano) curves.[26] These layouts store matrix blocks in a recursive order, which naturally preserves locality for *all* levels of the cache hierarchy simultaneously.[26]

## Cache-Oblivious Algorithms

Recursive layouts are the practical implementation of **cache-oblivious algorithms**. These algorithms, often recursive in nature (like a naive Strassen's), are designed to achieve asymptotically optimal cache performance *without* any hard-coded, hardware-dependent parameters like cache size $Z$ or line length $L$.[28] A simple recursive gemm that divides $n$

by 2 at each step will automatically "tile" itself for every cache level in the hierarchy.[29]

In practice, however, these algorithms often have higher overhead from function calls and complex indexing than *cache-aware* (explicitly tiled) algorithms.[28] Their primary advantage emerges when the data size exceeds main memory and data must be "tiled" for the disk (out-of-core).[28]

## Register-Level Optimization: The "Inner Kernel"

The blocking principle extends all the way down to the CPU registers, which are the fastest, smallest level of the memory hierarchy.

- **Register Blocking:** The "inner kernel" (or "micro-kernel") is designed to load a small "micro-tile" (e.g., $4 \times 8$ or $8 \times 12$ elements) into the CPU's registers and perform the dot-product accumulations *entirely within registers*, minimizing L1 cache traffic.[30]
- **SIMD (Single Instruction, Multiple Data):** This micro-kernel is not written in standard C. It is implemented using **SIMD intrinsics** (or pure assembly).[33] These instructions, such as Intel **AVX (Advanced Vector Extensions)** [35] or Arm **SVE (Scalable Vector Extension)** [34], perform a single operation (like a multiplication) on a "vector" of 8, 16, or more floating-point numbers simultaneously.
- **FMA (Fused Multiply-Add):** The critical SIMD instruction is **FMA**, which computes $d = a \times b + c$ in a single clock cycle, effectively doubling the peak arithmetic throughput of the processor.[38]

## The BLAS Standard: Synthesizing All Techniques

Highly-optimized BLAS libraries are not single, monolithic functions. They are layered frameworks that synthesize all of these techniques.

- **GotoBLAS:** The "Anatomy of High-Performance Matrix Multiplication" paper [39] (describing the principles of GotoBLAS) reveals that this library is *not* cache-oblivious, but *hyper-aware*.[32] Its design is based on two key principles:
  1. **L2 Cache Targeting:** The primary blocking target is *not* the L1 cache, but the much larger **L2 cache**. A large $mc \times kc$ block of matrix $A$ is "packed" (copied and rearranged) into a contiguous L2 buffer.[32]
  2. TLB Awareness: This packing strategy is also designed to minimize misses in the TLB

(Translation Look-aside Buffer), a "hidden" cache for virtual-to-physical address mappings. By packing, a large panel of $A$ or $B$ fits within one or two TLB entries, avoiding costly page walks.32
The register-blocked SIMD micro-kernel (Section 6.1 of 32) then streams data from the L2 (packed $A$) and L1 (panels of $B$) caches to the registers.32

- **BLIS Framework:** The BLIS (BLAS-like Library Instantiation Software) framework is the modern, open-source spiritual successor to the *ideas* of GotoBLAS.[41] BLIS formalizes this layered approach. Its key innovation is isolating the entire architecture-specific, hand-tuned part of the library into *one* small, portable **"micro-kernel"**.[41] This micro-kernel is typically a register-blocked gemm (e.g., $8 \times 12$) written in SIMD intrinsics or assembly.[44] The rest of the entire BLAS library (all other functions, all other blocking) is then built *on top* of this micro-kernel in portable C.[42] To port BLIS to a new CPU, a developer only needs to write this *one* micro-kernel.

# III. On-Node Parallelism: Multi-Core CPUs with OpenMP

Once the gemm kernel is optimized for a *single* core, the next step is to parallelize it across *all* cores on a single compute node. This is typically done using OpenMP, a directive-based, shared-memory programming model. However, adding threads is not a simple path to $N \times$ speedup; it introduces new and complex data locality challenges.

## Parallelizing the Loops: OpenMP Directives

The standard approach is to use the #pragma omp parallel for directive on the *outermost* loop of the tiled gemm.[22] This divides the work (e.g., the "tiles" of the $C$ matrix) among the available OpenMP threads in a Single Program, Multiple Data (SPMD) fashion.[45]

It is critical *not* to parallelize the inner loops. If multiple threads were to compute partial sums for the *same* C[i][j] element, they would create a data race, requiring costly atomic operations or critical sections, which would serialize the computation and destroy performance.[23] By parallelizing only the outer loop, each thread works on disjoint portions of the output matrix, an "embarrassingly parallel" problem that requires no synchronization until the very end.

## Performance Pitfalls: Load Balancing and Loop Scheduling

In a standard gemm where all matrix tiles are of uniform size, the workload is perfectly balanced. The default schedule(static) (which divides loop iterations evenly among threads before the loop starts) is optimal.[48]

However, if the workload is *uneven*—for example, in a triangular matrix multiplication [132], or a sparse-matrix operation [49]—schedule(static) causes severe *load imbalance*. Threads assigned to the "easy" work finish early and sit idle while other threads complete the "hard" work.[48] In these cases, schedule(dynamic) or schedule(guided) must be used.[49] These directives assign work in smaller chunks at runtime, which balances the load but incurs higher scheduling overhead.[48]

## The Cache Coherence Problem: False Sharing

A more subtle performance bug in shared-memory programming is **false sharing**.[51] In a multi-core CPU, cache coherence is maintained at the "cache line" level (typically 64 bytes). If thread 0 on core 0 writes to x and thread 1 on core 1 writes to y, no conflict occurs. But if x and y are *different* variables that happen to be located on the *same* 64-byte cache line, the hardware's coherence protocol (e.g., MESI) treats the *entire line* as "shared and modified".[52] This forces the line to be shuttled back and forth between the cores, effectively serializing the writes and causing a massive performance drop.

In a standard row-wise parallelization of gemm, this is *unlikely* to be a major bottleneck.[54] It can only occur at the single boundary where one thread's assigned chunk of C ends and the next thread's chunk begins. This minor effect can be eliminated entirely by aligning data structures to cache line boundaries [52] or ensuring the static work chunks are large multiples of the cache line size.[53]

## The NUMA Challenge: The *Real* Scalability Bottleneck

The most significant challenge in modern multi-core parallelization is the **NUMA**

**(Non-Uniform Memory Access)** architecture.[56] A modern server with two CPU sockets is a NUMA system. Each CPU (or "socket") has its *own* local memory controllers and attached RAM. Accessing this "local" memory is fast. However, a thread running on CPU 0 can also access the memory attached to CPU 1, but it must do so via a slower, cross-socket "remote" interconnect.[56]

An "agnostic" OpenMP scheduler, combined with a default OS memory policy, is a performance *disaster* on NUMA systems.[56] This can lead to scenarios where:

1. The main thread (running on CPU 0) allocates all matrices. By the "first-touch" policy, all data is allocated in the memory of NUMA Node 0.
2. The OpenMP runtime then spawns 16 threads, scheduling threads 8-15 to run on CPU 1.
3. Threads 8-15 will then spend their *entire* execution time performing *remote* memory accesses to Node 0, saturating the interconnect and running 30-100% slower than their counterparts on CPU 0.[57] This is a common reason why users report that their OpenMP gemm scales poorly or even *slows down* with more cores.[58]

The solution is to be **NUMA-aware**.[57] This involves two critical steps:

1. **Thread Affinity:** Threads must be "pinned" to specific cores (e.g., via GOMP_CPU_AFFINITY or kmp_affinity). This ensures that threads 0-7 *stay* on Node 0 and threads 8-15 *stay* on Node 1.
2. **Data Placement (First-Touch):** The data must be allocated on the correct NUMA node. This is achieved by ensuring that the *same thread* that will *compute* on a piece of data is the one to *initialize* it (the "first touch"). This is done by initializing the matrices *inside* the omp parallel region.

High-performance libraries like OpenBLAS [57] and custom implementations [57] automate this. They implement a two-level parallel design: first, work is divided at the NUMA-node level, and then threads are awakened *within* each node to perform the local initialization and computation.[57] This approach has been shown to reduce cross-die memory reads and writes by 24-62%, resulting in performance improvements of over 21%.[57]

# IV. Massive Parallelism: GPU Acceleration with CUDA

Graphics Processing Units (GPUs) offer orders of magnitude more parallelism than CPUs, with thousands of simple cores. To exploit this, the C-based CUDA programming model is used. The optimization journey for a CUDA gemm kernel involves mastering a different memory hierarchy (global, shared, register) and a unique execution model (warps, blocks, grids).

## From Naive to Optimized: A Step-by-Step Kernel Optimization

A typical gemm kernel evolves through several stages of optimization.[60]

- **Naive Kernel:** The simplest kernel assigns one thread to compute one element of the output matrix C[i][j].[60] This thread loops from k=0 to K-1, performing a dot product. This kernel is extremely slow because it is *memory-bound*.[61] It suffers from massive, redundant reads from slow **global memory** [62] and, critically, **uncoalesced memory access**.[63]

## Global Memory Coalescing: The First "Must-Do"

The CUDA execution model groups 32 threads into a "warp," which executes in lockstep (SIMT - Single Instruction, Multiple Thread).[64] The GPU memory system is designed to service memory requests for warps efficiently. If all 32 threads in a warp access 32 *consecutive* 32-bit (or 64-bit) words in global memory, the hardware "coalesces" these into one or two large memory transactions.[63]

The naive kernel's access to B[k][j] (column-wise) is the *opposite* of coalesced: consecutive threads access data separated by $N$ elements, resulting in 32 separate, slow transactions.[63] Fixing this access pattern (e.g., by transposing $B$, or more effectively, by using shared memory) is the first and most critical optimization, often yielding speedups of 6x or more.[60]

## Shared Memory Tiling: A Software-Managed L1 Cache

The next step is to eliminate redundant global memory access. This is achieved using **shared memory**—a small (e.g., 48-96 KB), on-chip scratchpad, programmable in CUDA (__shared__), that is orders of magnitude faster than global memory.[65] This is the CUDA equivalent of CPU cache-tiling.

The tiled algorithm works as follows [60]:

1. A "thread block" (e.g., $16 \times 16 = 256$ threads) is made responsible for computing one $16 \times 16$ tile of the $C$ matrix.

2. The 256 threads in the block *cooperatively* load one $16 \times k$ tile of $A$ and one $k \times 16$ tile of $B$ from slow global memory into two fast __shared__ memory arrays.
3. A barrier (__syncthreads()) is called to ensure all data is loaded before any thread proceeds.
4. The 256 threads then compute the tile-matrix-multiplication *entirely* from fast shared memory, reusing the loaded data $N$ times.
5. This process (load, sync, compute, sync) is looped over the $k$ dimension.

This technique dramatically reduces the global memory bandwidth requirement.[62] A pitfall to avoid is **shared memory bank conflicts**, where multiple threads in a warp access the same memory bank, serializing access.[65]

## Warp-Level and Register-Level Optimization

The fractal optimization principle applies again. The tiled shared-memory kernel can be further optimized by minimizing traffic *between* shared memory and the registers.

Instead of one thread computing one $C$ element, each *thread* is made responsible for a small $4 \times 4$ or $8 \times 8$ sub-tile of $C$.[60] The thread holds its $C$ accumulators in its private registers. In the inner loop, it loads the required values from *shared memory* into *registers*, performs the FMA operations, and repeats. This "register tiling" minimizes shared memory access, maximizes arithmetic intensity, and is the key to unlocking the GPU's computational power, often yielding another 5-10x speedup over the shared-memory kernel alone.[60]

## Exploiting Specialized Hardware: NVIDIA Tensor Cores

This leads to the optimization trend of moving from general parallel principles to specialized, hardware-first programming. Modern NVIDIA GPUs (Volta architecture and newer) include **Tensor Cores**: dedicated hardware units that compute a mixed-precision $D = A \times B + C$, where $A$, $B$, $C$, and $D$ are small, fixed-size matrices (e.g., $4 \times 4$ or $16 \times 16$).[69]

Their primary use is for deep learning, where they perform the multiplication using fast, low-precision **FP16 (half-precision)** inputs, while accumulating the result in **FP32 (single-precision)** to maintain accuracy.[69]

These cores cannot be used from standard C. They *must* be programmed using the **WMMA (Warp Matrix Multiply and Accumulate) API**.[69] This C++ API provides warp-level intrinsic functions (load_matrix_sync, mma_sync, store_matrix_sync) that map directly to the Tensor Core hardware operations.[69]

### The cuBLAS Library: An Optimized Dispatcher

The NVIDIA cuBLAS library is not a single kernel, but a massive, opaque, heuristic-driven *dispatcher*.[74] When a user calls cublasGemmEx(), the library inspects the matrix dimensions, data types, and the specific GPU architecture.[74]

- If the data type is FP16, and the matrix dimensions $M$, $N$, and $K$ are "just right" (e.g., multiples of 8 or 16), cuBLAS will automatically dispatch a highly-optimized **Tensor Core kernel** using WMMA.[71]
- If not, it will fall back to a "traditional" CUDA C++ kernel, one that has been hand-tuned using all the shared-memory and register-tiling techniques described above.[75]

NVIDIA's open-source **CUTLASS (CUDA Templates for Linear Algebra Subroutines)** library [69] provides a C++ template framework that shows how these complex, hierarchical, and hardware-specific kernels are constructed.

## V. Distributed-Memory Parallelism: Scaling Out with MPI

When matrices become so large that they cannot fit in the memory of a single node (even with GPUs), they must be "distributed" across the memory of multiple computers in a cluster. The Message Passing Interface (MPI) is the standard for managing this inter-node communication.

### Data Distribution Strategies

The first step in any distributed algorithm is to decide how to partition the matrices across the

$p$ available processors.

- **1D Decomposition:** The simplest approach is to slice the matrices by rows or columns.[77] For example, $A$ is sliced by rows, and $B$ is replicated on all nodes. This is simple to implement but not scalable, as it does not parallelize all dimensions and has a high memory cost for replicating $B$.[77]
- **2D Block-Cyclic:** This is the standard data layout used by **ScaLAPACK** (Scalable Linear Algebra PACKage).[79] The $p$ processors are viewed as a 2D grid, $P_r \times P_c$. The matrix is divided into blocks, which are then dealt out to the processor grid *cyclically* (like dealing cards). This complex layout solves two problems simultaneously [79]:
  1. **Load Balance:** For algorithms with an "active front" (like LU factorization), the cyclic distribution ensures all processors remain busy.[79]
  2. **Locality:** The blocking allows each processor to perform a local, cache-friendly gemm (using its single-node-optimized BLAS from Section II) on the data it owns.[79]

## Classical 2D Algorithms

- **Cannon's Algorithm:** A classic, elegant 2D algorithm designed for a $\sqrt{p} \times \sqrt{p}$ processor grid.[80] It works in two phases [80]:
  1. **Initial Shift:** Each $A(i,j)$ block is pre-shifted left by $i$ steps, and each $B(i,j)$ block is pre-shifted up by $j$ steps.
  2. Compute & Roll: In a loop $\sqrt{p}$ times:
     a. All $P(i,j)$ compute $C(i,j) \mathrel{+}= A(i,j) \times B(i,j)$.
     b. All $P(i,j)$ "roll" (circularly shift) their $A$ block one step left and their $B$ block one step up.
     The primary limitation of Cannon's algorithm is its rigidity: it assumes a square processor grid and does not gracefully handle rectangular matrices or processor grids.84
- The ScaLAPACK Standard: The SUMMA Algorithm (pdgemm)
  The Scalable Universal Matrix Multiplication Algorithm (SUMMA) is the state-of-the-art 2D algorithm and the foundation of ScaLAPACK's pdgemm.85
  SUMMA is simpler, more flexible, and more general than Cannon's.[84] It requires no initial data shifting. It iterates through the $k$ dimension in blocks (a "broadcast-multiply" approach) [85]:
  1. For $l = 0$ to $k-1$ (in blocks of size $k_b$):
     a. The processor column that owns the $l^{th}$ panel of $A$ broadcasts it horizontally along its processor row.
     b. The processor row that owns the $l^{th}$ panel of $B$ broadcasts it vertically along its processor column.
  2. All processors receive the two broadcasted panels and perform a local rank-$k_b$

update: $C_{\text{local}} \mathrel{+}= A_{\text{broadcasted}} \times B_{\text{broadcasted}}$.

This algorithm naturally handles rectangular matrices and processor grids and maps cleanly to the 2D block-cyclic data layout.[84]

### Communication-Avoiding Algorithms: 2.5D and 3D

In modern supercomputers, communication (data movement) is far more expensive than computation (flops).[88] The bandwidth cost of 2D algorithms (like SUMMA) is $W = \Omega(n^2 / \sqrt{p})$.[89] This cost is a bottleneck and can be asymptotically improved.

- **3D Algorithms:** These algorithms arrange the $p$ processors in a $p^{1/3} \times p^{1/3} \times p^{1/3}$ grid and replicate the data. This reduces the bandwidth cost to $W = \Omega(n^2 / p^{2/3})$, which is asymptotically optimal.[89]
- **2.5D Algorithms:** This is the practical, generalized version that *trades memory for communication*.[89]
    - **The Algorithm:** It uses $c$ copies of the input matrices, where $c$ is a tunable parameter ($1 \le c \le p^{1/3}$). The processors are arranged in a $\sqrt{p/c} \times \sqrt{p/c} \times c$ grid.[90]
    - **The Tradeoff:** By using $c$ times more memory, the algorithm reduces the total bandwidth cost by a factor of $c^{1/2}$ and the latency cost by a factor of $c^{3/2}$.[90]
    - This provides a "knob" to dial between the 2D (SUMMA, $c=1$) and 3D ($c=p^{1/3}$) algorithms, allowing an implementation to use all available node memory to reduce the primary bottleneck: communication. This has demonstrated speedups of up to 3x over 2D algorithms.[89]

This communication-avoiding principle also applies to Strassen's algorithm. It is less well-known, but Strassen *also* requires less communication than the naive $O(n^3)$ algorithm. Parallel Strassen implementations can communicate asymptotically less than *any* $O(n^3)$ algorithm, making them a powerful (though complex) choice for distributed-memory systems.[91]

## VI. Hybrid Parallelism and System-Level Integration

A modern supercomputer is a heterogeneous, hierarchical system. It is a *distributed-memory*

cluster (MPI) of *shared-memory*, multi-socket nodes (OpenMP/NUMA), which themselves contain specialized *co-processors* (CUDA/GPUs). High-performance applications must *mirror* this hardware structure in their software design, using a "hybrid" programming model.

## MPI + OpenMP: The CPU Cluster Standard

This is the dominant model for CPU-only clusters.[93]

- **MPI (Inter-node):** Handles message passing *between* compute nodes.[96] Typically, one MPI rank is started per *socket* or per *node*.
- **OpenMP (Intra-node):** Handles shared-memory parallelism *within* the node, using the NUMA-aware (Section III) and cache-blocked (Section II) strategies to manage all the cores on that node.[93]

This "MPI+OpenMP" model is more scalable than "flat MPI" (one MPI rank per core), as it drastically reduces the total number of MPI ranks, which in turn reduces the memory overhead and communication bottlenecks associated with MPI's internal data structures.[96]

## MPI + CUDA: The GPU Cluster Standard

This is the standard model for GPU-accelerated clusters.[97]

- **MPI (Inter-node):** Manages data movement between nodes.[97]
- **CUDA (Intra-node):** Each MPI rank (typically one per node) "owns" one or more GPUs on that node, launching CUDA kernels (from Section IV) to perform the local, compute-intensive portions of the algorithm.[99]

## Hiding Latency: Overlapping Communication and Computation

The *key* to distributed performance is to *hide* the cost of MPI communication.[102] This is achieved by overlapping communication with computation, typically using asynchronous operations and **CUDA Streams**.[103]

A CUDA stream is a sequence of operations that execute in order.[103] Operations in *different*

streams can run concurrently.[103] A high-performance application will use multiple streams to create a software pipeline [105]:

1. Launch a cudaMemcpyAsync() to download the *next* block of data (e.g., block[k+1]) from the CPU to the GPU on stream_H2D.
2. Launch the CUDA kernel (my_kernel) to compute on the *current* block (block[k]) on stream_compute.
3. Launch a cudaMemcpyAsync() to upload the *previous* result (result[k-1]) from the GPU to the CPU on stream_D2H.

While the GPU is busy with the kernel on stream_compute, the CPU and GPU's copy-engines are concurrently working on the data transfers. The CPU, free from managing the GPU, can then use non-blocking MPI (MPI_Isend, MPI_Irecv) to send result[k-1] and receive block[k+1].[102] Orchestrating this pipeline is complex, as a single synchronous call (like a blocking MPI_Wait or a non-streamed cudaMemcpy) can stall the entire pipeline.[107]

## Accelerating Communication: CUDA-Aware MPI

The pipeline described above is complex because it requires manual staging of data through host (CPU) memory.[109] A **CUDA-Aware** MPI library simplifies this and improves performance.[109]

- **Without CUDA-Aware MPI:** A transfer requires five explicit, slow steps: 1) cudaMemcpy(D2H) on sender, 2) MPI_Send(host_buf), 3) MPI_Recv(host_buf), 4) cudaMemcpy(H2D) on receiver.[109]
- **With CUDA-Aware MPI:** The programmer can pass a **GPU device pointer *directly* to the MPI call**: MPI_Send(gpu_buf,...). The MPI library itself, using CUDA's Unified Virtual Addressing (UVA), detects that the pointer is on the device and manages the transfer automatically.[109]

This enables the MPI library to use the most optimized data path. In the best-case scenario, it uses **GPUDirect RDMA**.[110] If the network interface card (e.g., InfiniBand HCA) and GPU support it, GPUDirect RDMA allows the network card to pull data *directly from the GPU's memory* (bypassing host RAM) and send it *directly to the remote GPU's memory* (bypassing the remote host's RAM).[110] This is the absolute lowest-latency, highest-bandwidth communication path available in a modern cluster.

## Hybrid Scheduling Libraries: The MAGMA Framework

The **MAGMA (Matrix Algebra on GPU and Multi-core Architectures)** library [113] institutionalizes this hybrid philosophy *within a node*.[115] Instead of a "GPU-only" approach (which can be bottlenecked by serial portions of an algorithm), MAGMA uses a hybrid CPU-GPU scheduling algorithm.[115]

- **GPU:** Receives large, compute-bound, highly parallel tasks (like gemm on large blocks).[115]
- **CPU:** Receives small, serial, or latency-bound tasks (e.g., in LAPACK, the panel factorizations for LU or QR).[115]

MAGMA uses a "lookahead" technique with CUDA streams to *overlap* the CPU's serial work with the GPU's parallel work, effectively *hiding* the serial CPU bottleneck *behind* the parallel GPU computation.[115]

# VII. The Sparse Matrix Challenge: SpMV and SpGEMM Optimization

The optimizations for *dense* matrices (where all values are non-zero) are fundamentally *antithetical* to those for *sparse* matrices (where most values are zero). Dense optimization is about *exploiting static, regular structure*. Sparse optimization is about *managing dynamic, irregular structure* and data-dependent computation.

The primary challenges for sparse matrix computation are [117]:

1. **Irregular Memory Access:** Data is not contiguous. Accessing a non-zero element requires an *indirect* memory lookup (e.g., values[row_ptr[i]]), which defeats caches and global memory coalescing.[118]
2. **Load Imbalance:** The number of non-zeros (NNZ) per row can vary by orders of magnitude, causing massive load imbalance for parallel threads.[118]
3. **Unknown Output Size (for SpGEMM):** In $C = A \times B$, the number of nonzeros in $C$ (and their locations) is unknown in advance, making memory pre-allocation a major algorithmic hurdle.[121]

## Data Structures for Sparsity: The Optimization is the Layout

In sparse computing, the choice of data structure *is* the primary optimization.

- **COO (Coordinate):** Stores (row, col, val) triplets. Simple, but inefficient for computation.[122]
- **CSR (Compressed Sparse Row):** Stores values, column_indices, and row_pointers (an array of size $m+1$ pointing to the start of each row). This is the *de facto* standard for row-wise operations like SpMV ($y = Ax$).[119]
- **CSC (Compressed Sparse Column):** The transpose of CSR. Ideal for column-wise operations.[123]
- **ELL (Ellpack):** Stores a dense $N \times K$ array, where $K$ is the *maximum* NNZ in any row. This is very efficient for vector hardware (like GPUs) *if* all rows have a similar number of nonzeros. It is *extremely* wasteful if the NNZ-per-row varies wildly.[118]
- **HYB (Hybrid):** The practical, high-performance solution for GPUs.[118] It combines ELL and COO. It stores the "regular" part of the matrix (e.g., up to $K$ elements per row) in the fast ELL format, and all "irregular" remaining elements from long rows in the flexible COO format.[118]

## Parallel Sparse Matrix-Vector (SpMV) Multiplication

- **OpenMP (CPU):** Parallelizing the outer for loop (over rows) of a CSR-based SpMV is the standard approach.[125] However, due to the high likelihood of load imbalance, schedule(dynamic) or schedule(guided) is *required* to achieve good performance.[120]
- **CUDA (GPU):** Several kernels exist to combat irregularity [118]:
  1. **Scalar Kernel:** One thread per row. This is simple, but suffers from uncoalesced memory access.[118]
  2. **Vector Kernel:** One *warp* (32 threads) per row. This *solves* the coalescing problem (as the warp reads values and col_idx contiguously), but is very inefficient if most rows have fewer than 32 nonzeros.[118]
  3. **HYB Kernel:** The cuSPARSE library solution. It uses the HYB data format, running the fast, coalesced ELL kernel on the ELL portion and a robust COO-based kernel (which uses one thread per *non-zero*) on the COO portion.[118]

## Parallel Sparse Matrix-Matrix (SpGEMM) Multiplication

SpGEMM ($C = A \times B$) is significantly harder than SpMV.[121] The central problem is how to

accumulate the partial products that map to the same C[i,j] coordinate.

A common GPU-based approach is the **Three-Phase (ESC) Algorithm** [127]:

1. **Expansion:** In parallel, compute all the partial products (e.g., $A[i,k] \times B[k,j]$) and store them as a large, unsorted intermediate list of (row, col, val) triplets.
2. **Sorting:** Perform a massive, parallel sort (e.g., a radix sort) on this intermediate list, to group all triplets by their (i,j) coordinate.
3. **Contraction (Merge):** Perform a parallel "segmented reduction." For each unique (i,j) key (coordinate), sum (contract) all corresponding values to get the final C[i,j] element.

Load balancing is the dominant challenge.[121] Advanced methods like "TileSpGEMM" [129] break the sparse matrix into *fixed-size* tiles (even if they are mostly empty), which naturally alleviates the load imbalance by regularizing the problem.

# VIII. Synthesis and Future Trajectories

This literature review demonstrates that "matrix multiplication optimization" is not a single technique but a deep, hierarchical stack of optimizations that must be co-designed with the hardware.

## A Holistic View: The "Full Stack" Optimization

A world-class, cluster-scale gemm implementation, as found in modern HPC libraries, is a synthesis of every section of this report:

- **Algorithm (Sec I):** A hybrid algorithm, using Strassen's or an AlphaTensor-like algorithm for large $n$, which recursively calls a highly-optimized $O(n^3)$ kernel at its crossover point.[6]
- **CPU Core (Sec II):** The $O(n^3)$ kernel is a BLIS-like implementation with a hand-tuned SIMD/FMA micro-kernel, register blocking, and L2 cache packing.[30]
- **CPU Node (Sec III):** This kernel is parallelized using NUMA-aware OpenMP, with explicit thread affinity and a first-touch data placement policy.[42]
- **GPU Device (Sec IV):** On a GPU, the kernel is a CUTLASS-like, shared-memory and register-tiled CUDA implementation, which dispatches to a WMMA-based Tensor Core kernel if the hardware and data types are compatible.[60]
- **Cluster (Sec V):** The global problem is distributed using MPI with a 2D Block-Cyclic

layout, and the inter-node communication follows a 2.5D or SUMMA communication-avoiding algorithm.[85]

- **Scheduling (Sec VI):** The entire application is scheduled using a MAGMA-like hybrid model, using CUDA-aware MPI [109] and CUDA streams [103] to overlap communication (via GPUDirect RDMA [110]) with computation.
- **Matrix Type (Sec VII):** If the matrix is sparse, this entire, regular-data pipeline is discarded in favor of a specialized sparse stack (e.g., HYB format [118] and an ESC-based SpGEMM algorithm [127]).

## The Evolving Landscape: AI as the Performance Engineer

The most recent literature indicates a fundamental shift in *how* these optimizations are created. The process, which for 50 years required rare, expert human-level knowledge, is now becoming a target for AI-driven automation.

AlphaTensor was not just a mathematical discovery; it was a proof-of-concept for *hardware-aware code generation*.[14] It was explicitly rewarded for finding algorithms that performed well on specific hardware, even if they were less "elegant" to a human.[16]

This trend is accelerating. Recent work (2024-2025) describes "agentic frameworks" based on Large Language Models (LLMs) that are designed to *mimic the workflow of expert HPC engineers*.[130] These agents can generate novel GPU kernels, benchmark them, analyze the performance-profiler output, and iteratively refine the code, achieving speedups of 16x to 179x over human-written baselines.[130]

The future of high-performance matrix multiplication, and HPC at large, appears to be the full automation of the optimization stack. The role of the human expert is evolving from *writing* the perfect, hand-tuned micro-kernel to *designing the AI agent* that can *discover* it, thereby closing the hardware-software co-design loop that has defined the field for half a century.

### Works cited


1. Hybrid optimization technique for matrix chain multiplication using Strassen's algorithm - PMC - NIH, accessed November 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC12000801/
2. Discovering novel algorithms with AlphaTensor - Google DeepMind, accessed November 4, 2025, https://deepmind.google/discover/blog/discovering-novel-algorithms-with-alphatensor/
3. Strassen algorithm - Wikipedia, accessed November 4, 2025,

https://en.wikipedia.org/wiki/Strassen_algorithm
4. Why is the Strassen algorithm for matrix multiplication faster than the naive approach even though it does more addition and subtraction operations? - Reddit, accessed November 4, 2025, https://www.reddit.com/r/algorithms/comments/4z3aib/why_is_the_strassen_algorithm_for_matrix/
5. Computational complexity of matrix multiplication - Wikipedia, accessed November 4, 2025, https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication
6. (PDF) Implementation of Strassen's Algorithm for Matrix Multiplication, accessed November 4, 2025, https://www.researchgate.net/publication/2779622_Implementation_of_Strassen's_Algorithm_for_Matrix_Multiplication
7. Fast and Practical Strassen's Matrix Multiplication using FPGAs - arXiv, accessed November 4, 2025, https://arxiv.org/html/2406.02088v1
8. [R] Discovering Faster Matrix Multiplication Algorithms With Reinforcement Learning - Reddit, accessed November 4, 2025, https://www.reddit.com/r/MachineLearning/comments/xwfvlw/r_discovering_faster_matrix_multiplication/
9. What are the time complexities of the Coppersmith-Winograd algorithm and the Strassen algorithm? - Massed Compute, accessed November 4, 2025, https://massedcompute.com/faq-answers/?question=What%20are%20the%20time%20complexities%20of%20the%20Coppersmith-Winograd%20algorithm%20and%20the%20Strassen%20algorithm?
10. Multiplying matrices faster than Coppersmith-Winograd - ResearchGate, accessed November 4, 2025, https://www.researchgate.net/publication/262289301_Multiplying_matrices_faster_than_Coppersmith-Winograd
11. How fast can we *really* multiply matrices? - MathOverflow, accessed November 4, 2025, https://mathoverflow.net/questions/101531/how-fast-can-we-really-multiply-matrices
12. On AlphaTensor's new matrix multiplication algorithms - The ryg blog, accessed November 4, 2025, https://fgiesen.wordpress.com/2022/10/06/on-alphatensors-new-matrix-multiplication-algorithms/
13. A Framework for Practical Parallel Fast Matrix Multiplication : r/HPC - Reddit, accessed November 4, 2025, https://www.reddit.com/r/HPC/comments/2gnvsa/a_framework_for_practical_parallel_fast_matrix/
14. Paper: Discovering novel algorithms with AlphaTensor [Deepmind] - LessWrong, accessed November 4, 2025, https://www.lesswrong.com/posts/5Zfyktwgz3rvAvZyL/paper-discovering-novel-algorithms-with-alphatensor-deepmind
15. Discovering faster matrix multiplication algorithms with reinforcement learning -

PMC, accessed November 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC9534758/
16. The big claim turns out to be a little overstated. The claim: > AlphaTensor's al... | Hacker News, accessed November 4, 2025, https://news.ycombinator.com/item?id=33097086
17. [2405.17322] Evaluation of computational and energy performance in matrix multiplication algorithms on CPU and GPU using MKL, cuBLAS and SYCL - arXiv, accessed November 4, 2025, https://arxiv.org/abs/2405.17322
18. A data locality optimizing algorithm | Semantic Scholar, accessed November 4, 2025, https://www.semanticscholar.org/paper/A-data-locality-optimizing-algorithm-Wolf-Lam/f4dff66ba8f2338d118f379f2eff1410feb57ce6
19. A high-performance matrix-matrix multiplication methodology for CPU and GPU architectures, accessed November 4, 2025, https://shura.shu.ac.uk/18333/1/Kelefouras-HighPerformanceMatrix-MatrixMaultiplications%28AM%29.pdf
20. Defensive Loop Tiling for Shared Cache - Computer Science : University of Rochester, accessed November 4, 2025, https://www.cs.rochester.edu/u/cding/dpal/files/BaoD_CGO13.pdf
21. What are the pros and cons of row/column major ordering? - Julia Discourse, accessed November 4, 2025, https://discourse.julialang.org/t/what-are-the-pros-and-cons-of-row-column-major-ordering/110045
22. Optimizing matrix multiplication: cache + OpenMP - Mathieu GAILLARD, accessed November 4, 2025, http://www.mgaillard.fr/2020/08/29/matrix-multiplication-optimizing.html
23. Matrix Multiplication using OpenMP (C) - Collapsing all the loops - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/60360361/matrix-multiplication-using-openmp-c-collapsing-all-the-loops
24. Row Major vs Column Major Vectors and Matrices - Matrix Operations - Geometry, accessed November 4, 2025, https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/row-major-vs-column-major-vector.html
25. matrix - Row major versus Column major layout of matrices, accessed November 4, 2025, https://scicomp.stackexchange.com/questions/4796/row-major-versus-column-major-layout-of-matrices
26. Recursive Array Layouts and Fast Parallel Matrix Multiplication - Purdue Engineering, accessed November 4, 2025, https://engineering.purdue.edu/~mithuna/pubs/spaa99.pdf
27. Empirical Analysis of Cache-Oblivious Matrix Multiplication on Multicore Processor Systems, accessed November 4, 2025, https://www.researchgate.net/publication/351358601_Empirical_Analysis_of_Cache-Oblivious_Matrix_Multiplication_on_Multicore_Processor_Systems

28. Cache-oblivious algorithm - Wikipedia, accessed November 4, 2025, https://en.wikipedia.org/wiki/Cache-oblivious_algorithm
29. Cache-Oblivious Algorithms, accessed November 4, 2025, https://ocw.mit.edu/courses/6-895-theory-of-parallel-systems-sma-5509-fall-2003/6dc7de52dcf13b53cebf2fe10ae6752a_cach_oblvs_thsis.pdf
30. High-Performance Matrix Multiplication - GitHub Gist, accessed November 4, 2025, https://gist.github.com/nadavrot/5b35d44e8ba3dd718e595e40184d03f0?permalink_comment_id=2602745
31. Assignment 1: High Performance Matrix Multiplication on a CPU - Computer Science, accessed November 4, 2025, https://cseweb.ucsd.edu/classes/fa15/cse260-a/static/HW/A1/
32. Anatomy of High-Performance Matrix Multiplication - Texas ..., accessed November 4, 2025, https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf
33. Compilers, Hands-Off My Hands-On Optimizations - Carnegie Mellon University, accessed November 4, 2025, https://spiral.ece.cmu.edu/pub-spiral/pubfile/wpmvp16_260.pdf
34. Exploring Source-to-Source Compiler Transformation of OpenMP SIMD Constructs for Intel AVX and Arm SVE Vector Architectures - Stony Brook University, accessed November 4, 2025, https://www.stonybrook.edu/commcms/ookami/_pdf/PMAM2022_REXSIMDCompiler_Flynn.pdf
35. (PDF) Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor - ResearchGate, accessed November 4, 2025, https://www.researchgate.net/publication/338917279_Optimizing_matrix-matrix_multiplication_on_intel's_advanced_vector_extensions_multicore_processor
36. Acceleration of Particle Swarm Optimization with AVX Instructions - MDPI, accessed November 4, 2025, https://www.mdpi.com/2076-3417/13/2/734
37. Exploring Source-to-Source Compiler Transformation of OpenMP SIMD Constructs for Intel AVX and Arm SVE Vector Architectures, accessed November 4, 2025, https://par.nsf.gov/servlets/purl/10394960
38. Advanced Matrix Multiplication Optimization on Modern Multi-Core Processors - salykova, accessed November 4, 2025, https://salykova.github.io/matmul-cpu
39. Anatomy of High-Performance Matrix Multiplication - Texas Computer Science, accessed November 4, 2025, https://www.cs.utexas.edu/~flame/pubs/GotoTOMS.pdf
40. Anatomy of high-performance matrix multiplication Kazushige Goto, Robert A. van de Geijn ACM Transactions on Mathematical Software (TOMS), 2008 | Request PDF - ResearchGate, accessed November 4, 2025, https://www.researchgate.net/publication/255881938_Anatomy_of_high-performance_matrix_multiplication_Kazushige_Goto_Robert_A_van_de_Geijn_ACM_Transactions_on_Mathematical_Software_TOMS_2008
41. BLISlab: A Sandbox for Optimizing GEMM - arXiv, accessed November 4, 2025, https://arxiv.org/pdf/1609.00076

42. 0 The BLIS Framework: Experiments in Portability - Texas Computer Science, accessed November 4, 2025, https://www.cs.utexas.edu/~flame/pubs/blis2_toms_rev3.pdf
43. LibShalom: Optimizing Small and Irregular-Shaped Matrix Multiplications on ARMv8 Multi-Cores - Zheng Wang, accessed November 4, 2025, https://zwang4.github.io/publications/sc21.pdf
44. Micro-Kernels for Portable and Efficient Matrix Multiplication in Deep Learning - Docta Complutense, accessed November 4, 2025, https://docta.ucm.es/bitstreams/d7d42148-4668-475f-950b-b17c4047d3df/download
45. PARALLEL COMPUTING OF MATRIX MULTIPLICATION IN OPEN MP SUPPORTED CODEBLOCKS - Mili Publications, accessed November 4, 2025, https://www.mililink.com/upload/article/1767042617aams_vol188_june_2019_a17_p775-787_hari_singh,_dinesh_chander_and_ravindara_bhatt.pdf
46. Matrix Multiplication: Optimizing the code from 6 hours to 1 sec | by Vipul Vaibhaw - Medium, accessed November 4, 2025, https://vaibhaw-vipul.medium.com/matrix-multiplication-optimizing-the-code-from-6-hours-to-1-sec-70889d33dcfa
47. A “Hands-on” Introduction to OpenMP*, accessed November 4, 2025, https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
48. Parallel Programming with OpenMP, accessed November 4, 2025, https://web.njit.edu/~shahriar/class_home/HPC/omp3.pdf
49. A Study on Load Imbalance in Parallel Hypermatrix Multiplication using OpenMP * - UPC, accessed November 4, 2025, https://people.ac.upc.edu/josepr/pubPapers/2005-09-13-PPAM05-HM-OpenMP-np.pdf
50. Low-overhead Loop Scheduling to Improve Performance of Scientific Applications - OpenMP, accessed November 4, 2025, https://www.openmp.org/wp-content/uploads/sc19-vivek-talk.pdf
51. 6.2 False Sharing And How To Avoid It, accessed November 4, 2025, https://docs.oracle.com/cd/E19205-01/819-5270/6n7c71veg/index.html
52. 6.7. Avoid False Sharing, accessed November 4, 2025, https://www.nic.uoregon.edu/~khuck/ts/acumem-report/manual_html/ch06s07.html
53. Avoiding False Sharing in OpenMP with arrays - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/19278435/avoiding-false-sharing-in-openmp-with-arrays
54. Parallelize a matrix with openmp to avoid false sharing - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/57921969/parallelize-a-matrix-with-openmp-to-avoid-false-sharing
55. How minimize "false sharing" in this array processing with OpenMP? - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/77882130/how-minimize-false-sharing-in-thi

s-array-processing-with-openmp
56. NUMA-Aware Multicore Matrix Multiplication | Parallel Processing Letters, accessed November 4, 2025, https://www.worldscientific.com/doi/full/10.1142/S0129626414500066
57. NUMA-Aware DGEMM Based on 64-Bit ARMv8 Multicore ... - MDPI, accessed November 4, 2025, https://www.mdpi.com/2079-9292/10/16/1984
58. Matrix multiplication poor efficiency on a 4 socket NUMA system - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/25055604/matrix-multiplication-poor-efficiency-on-a-4-socket-numa-system
59. NUMA-Aware Optimization of Sparse Matrix-Vector Multiplication on ARMv8-based Many-Core Architectures - Super Scientific Software Laboratory, accessed November 4, 2025, https://www.ssslab.cn/assets/papers/2020-yu-numa.pdf
60. How to Optimize a CUDA Matmul Kernel for cuBLAS-like ... - siboehm, accessed November 4, 2025, https://siboehm.com/articles/22/CUDA-MMM
61. Matrix Multiplication CUDA · Embedded Computer Architecture - GPU Assignment 2016-2017 - GitLab, accessed November 4, 2025, https://ecatue.gitlab.io/GPU2016/cookbook/matrix_multiplication_cuda/
62. Memory Coalescing and Tiled Matrix Multiplication - 0Mean1Sigma, accessed November 4, 2025, https://0mean1sigma.com/chapter-4-memory-coalescing-and-tiled-matrix-multiplication/
63. The CUDA Parallel Programming Model - 5. Memory Coalescing - Fang's Notebook, accessed November 4, 2025, https://nichijou.co/cuda5-coalesce/
64. Memory Coalescing Techniques, accessed November 4, 2025, http://homepages.math.uic.edu/~jan/mcs572f16/memory_coalescing.pdf
65. Mastering CUDA Matrix Multiplication: An Introduction to Shared Memory, Tile Memory Coalescing, and Bank Conflicts | by Dhanush | Medium, accessed November 4, 2025, https://medium.com/@dhanushg295/mastering-cuda-matrix-multiplication-an-introduction-to-shared-memory-tile-memory-coalescing-and-d7979499b9c5
66. High Performance Matrix Multiplication - arXiv, accessed November 4, 2025, https://arxiv.org/html/2509.04594v1
67. Performance Analysis of CUDA-based General Matrix Multiplication through Memory Coalescing and Grid-Level Parallelization - kth .diva, accessed November 4, 2025, https://kth.diva-portal.org/smash/get/diva2:1985710/FULLTEXT01.pdf
68. Matrix Multiplication in CUDA - Harshit Kumar, accessed November 4, 2025, https://kharshit.github.io/blog/2024/06/07/matrix-multiplication-cuda
69. NVIDIA Tensor Core Programmability, Performance & Precision - arXiv, accessed November 4, 2025, https://arxiv.org/pdf/1803.04014
70. Numerical behavior of NVIDIA tensor cores - PMC - NIH, accessed November 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC7959640/
71. Programming Tensor Cores in CUDA 9 | NVIDIA Technical Blog, accessed

November 4, 2025, https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/
72. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply, accessed November 4, 2025, https://www.cse.ust.hk/~weiwa/papers/yan-ipdps20.pdf
73. Matrix Multiplication Background User's Guide - NVIDIA Docs, accessed November 4, 2025, https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html
74. 1. Introduction — cuBLAS 13.0 documentation - NVIDIA Docs Hub, accessed November 4, 2025, https://docs.nvidia.com/cuda/cublas/
75. WMMA default cores - cuda - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/56968557/wmma-default-cores
76. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance - arXiv, accessed November 4, 2025, https://arxiv.org/pdf/2203.03341
77. Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication - People @EECS, accessed November 4, 2025, https://people.eecs.berkeley.edu/~yelick/papers/spdmmm16.pdf
78. Develop a parallel application that does multiply two Matrices C and MPI, accessed November 4, 2025, https://www.christianbaun.de/CGC1718/Skript/CloudPresentation_Shamima_Akhter.pdf
79. The Two-dimensional Block-Cyclic Distribution - NetLib.org, accessed November 4, 2025, https://www.netlib.org/scalapack/slug/node75.html
80. Cannon's algorithm: (a) Initial shifting of submatrices. (b)... - ResearchGate, accessed November 4, 2025, https://www.researchgate.net/figure/Cannons-algorithm-a-Initial-shifting-of-submatrices-b-Intermediate-result-of-first_fig3_299550775
81. Cannon's matrix multiplication algorithm with MPI. - GitHub, accessed November 4, 2025, https://github.com/canaknesil/cannons-algorithm-mpi
82. Matrix multiplication with mpi - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/13402727/matrix-multiplication-with-mpi
83. Scalability of Parallel Algorithms for Matrix Multiplication, accessed November 4, 2025, https://www3.nd.edu/~zxu2/acms60212-40212-S12/scalability-of-parallel-alg-for-matrix-multiplication.pdf
84. PARALLEL MATRIX MULTIPLICATION: A SYSTEMATIC JOURNEY 1. Introduction. This paper serves a number of purposes, accessed November 4, 2025, https://www.cs.utexas.edu/~flame/pubs/SUMMA2d3dTOMS.pdf
85. SUMMA: Scalable Universal Matrix Multiplication ... - NetLib.org, accessed November 4, 2025, https://www.netlib.org/lapack/lawnspdf/lawn96.pdf
86. SUMMA: Scalable Universal Matrix Multiplication Algorithm - CS@UCSB, accessed November 4, 2025, https://sites.cs.ucsb.edu/~gilbert/cs140/old/cs140Win2009/assignments/lawn96-SUMMA.pdf

87. Analysis of a Class of Parallel Matrix Multiplication Algorithms - Texas Computer Science, accessed November 4, 2025, https://www.cs.utexas.edu/~plapack/papers/ipps98/ipps98.html
88. Introduction to communication avoiding algorithms for direct methods of factorization in Linear Algebra - Inria, accessed November 4, 2025, https://who.rocq.inria.fr/Laura.Grigori/Papers/OverviewCA.pdf
89. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms - Berkeley EECS, accessed November 4, 2025, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-72.pdf
90. Communication-optimal parallel 2.5D matrix ... - UC Berkeley EECS, accessed November 4, 2025, http://eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-10.pdf
91. Communication-Avoiding Algorithms and Fast Matrix Multiplication - Microsoft Research, accessed November 4, 2025, https://www.microsoft.com/en-us/research/video/communication-avoiding-algorithms-and-fast-matrix-multiplication/
92. Communication-Avoiding Parallel Strassen: Implementation and Performance - CS.HUJI, accessed November 4, 2025, https://www.cs.huji.ac.il/~odedsc/papers/CAPS-impl.pdf
93. Hybrid MPI and OpenMP Parallel Programming, accessed November 4, 2025, https://www.openmp.org/wp-content/uploads/HybridPP_Slides.pdf
94. Hybrid MPI/OpenMP power-aware computing - SciSpace, accessed November 4, 2025, https://scispace.com/pdf/hybrid-mpi-openmp-power-aware-computing-4lkldddm9d0.pdf
95. Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming - arXiv, accessed November 4, 2025, https://arxiv.org/pdf/1101.0091
96. Performance Analysis of Matrix-Vector Multiplication in Hybrid (MPI + OpenMP) - International Journal of Computer Applications, accessed November 4, 2025, https://www.ijcaonline.org/volume22/number5/pxc3873561.pdf
97. Efficient large Pearson correlation matrix computing using hybrid MPI/CUDA, accessed November 4, 2025, https://www.researchgate.net/publication/252012001_Efficient_large_Pearson_correlation_matrix_computing_using_hybrid_MPICUDA
98. Matrix Multiplication using CUDA + MPI - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/5972033/matrix-multiplication-using-cuda-mpi
99. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters - ScholarWorks, accessed November 4, 2025, https://scholarworks.boisestate.edu/cgi/viewcontent.cgi?article=1004&context=mecheng_facpubs
100. CUDA Multi GPU matrix multiplication advice - NVIDIA Developer Forums, accessed November 4, 2025,

https://forums.developer.nvidia.com/t/cuda-multi-gpu-matrix-multiplication-advice/49448
101. GPU computing performance analysis on matrix multiplication - ResearchGate, accessed November 4, 2025, https://www.researchgate.net/publication/337979809_GPU_computing_performance_analysis_on_matrix_multiplication
102. Patterns for Overlapping Communication and Computation - Parallel Programming Laboratory, accessed November 4, 2025, http://charm.cs.illinois.edu/newPapers/09-30/paper.pdf
103. How to Overlap Data Transfers in CUDA C/C++ | NVIDIA Technical Blog, accessed November 4, 2025, https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/
104. CUDA Kernel Execution Overlap - Lei Mao's Log Book, accessed November 4, 2025, https://leimao.github.io/blog/CUDA-Kernel-Execution-Overlap/
105. Concurrency of one large kernel with many small kernels and memcopys (CUDA) - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/57061559/concurrency-of-one-large-kernel-with-many-small-kernels-and-memcopys-cuda
106. MPI overlapping communication and computation - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/72561085/mpi-overlapping-communication-and-computation
107. Overlapping computation with MPI communication - CUDA Programming and Performance, accessed November 4, 2025, https://forums.developer.nvidia.com/t/overlapping-computation-with-mpi-communication/62112
108. MVAPICH deadlocks on CUDA memory while kernel is running - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/42420146/mvapich-deadlocks-on-cuda-memory-while-kernel-is-running
109. An Introduction to CUDA-Aware MPI | NVIDIA Technical Blog, accessed November 4, 2025, https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/
110. 1. Overview — GPUDirect RDMA 13.0 documentation - NVIDIA Docs Hub, accessed November 4, 2025, https://docs.nvidia.com/cuda/gpudirect-rdma/
111. CUDA-Aware-MPI: Part 1: Understanding node topology and communication bandwidth, accessed November 4, 2025, https://blogs.fau.de/adityauj/2025/02/18/cuda-aware-mpi-part-1-understanding-node-topology-and-communication-bandwidth/
112. C-GDR: High-Performance Container-Aware GPUDirect MPI Communication Schemes on RDMA Networks, accessed November 4, 2025, https://par.nsf.gov/servlets/purl/10112761
113. MAGMA - NVIDIA Developer, accessed November 4, 2025, https://developer.nvidia.com/magma
114. MAGMA - Innovative Computing Laboratory - University of Tennessee,

Knoxville, accessed November 4, 2025, https://icl.utk.edu/magma/
115. MAGMA: Enabling exascale performance with ... - NetLib.org, accessed November 4, 2025, https://www.netlib.org/utk/people/JackDongarra/PAPERS/magma-enabled-2024.pdf
116. Accelerating Dense Linear Algebra for GPUs, Multicores and Hybrid Architectures: an Autotuned and Algorithmic Approach, accessed November 4, 2025, https://trace.tennessee.edu/cgi/viewcontent.cgi?article=1794&context=utk_gradthes
117. [2002.11273] A Systematic Survey of General Sparse Matrix-Matrix Multiplication - arXiv, accessed November 4, 2025, https://arxiv.org/abs/2002.11273
118. Efficient Sparse Matrix-Vector Multiplication on CUDA - NVIDIA, accessed November 4, 2025, https://www.nvidia.com/docs/io/66889/nvr-2008-004.pdf
119. A Systematic Literature Survey of Sparse Matrix-Vector Multiplication - arXiv, accessed November 4, 2025, https://arxiv.org/html/2404.06047v1
120. Performance Portability of an SpMV Kernel Across Scientific Computing and Data Science Applications - Sandia National Laboratories, accessed November 4, 2025, https://www.sandia.gov/app/uploads/sites/143/2021/10/daniel-dunlavy-2021-OIEIBeDu21.pdf
121. [1504.05022] A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors - arXiv, accessed November 4, 2025, https://arxiv.org/abs/1504.05022
122. Comparative Analysis of Sparse Matrix Algorithms For Information Retrieval - ir@Georgetown, accessed November 4, 2025, https://ir.cs.georgetown.edu/downloads/SCI-Journal-CameraReady-Goharian.pdf
123. Mastering Sparse Data Structures: Efficient Strategies for Handling Zero-Rich Datasets at Scale - Configr Technologies, accessed November 4, 2025, https://configr.medium.com/mastering-sparse-data-structures-efficient-strategies-for-handling-zero-rich-datasets-at-scale-0333f2ce24e0
124. Optimizing the Performance of Sparse Matrix-Vector Multiplication - UC Berkeley EECS, accessed November 4, 2025, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2000/CSD-00-1104.pdf
125. Autotuning Sparse Matrix-Vector Multiplication for Multicore - UC Berkeley EECS, accessed November 4, 2025, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-215.pdf
126. Optimizing Sparse Matrix-Vector Multiplications on GPUs - ResearchGate, accessed November 4, 2025, https://www.researchgate.net/publication/228345757_Optimizing_Sparse_Matrix-Vector_Multiplications_on_GPUs
127. Optimizing Sparse Matrix-Matrix Multiplication for the ... - Luke Olson, accessed November 4, 2025, https://lukeo.cs.illinois.edu/files/2015_BeDaOl_SPMM.pdf

128. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors - People @EECS, accessed November 4, 2025, https://people.eecs.berkeley.edu/~aydin/spgemm_parco2019.pdf
129. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs - Super Scientific Software Laboratory, accessed November 4, 2025, https://www.ssslab.cn/assets/papers/2022-niu-tilespgemm.pdf
130. STARK: Strategic Team of Agents for Refining Kernels - arXiv, accessed November 4, 2025, https://arxiv.org/html/2510.16996v1
131. [2506.09092] CUDA-LLM: LLMs Can Write Efficient CUDA Kernels - arXiv, accessed November 4, 2025, https://arxiv.org/abs/2506.09092
132. Parallelizing a 1D matrix multiplication using OpenMP - Stack Overflow, accessed November 4, 2025, https://stackoverflow.com/questions/67274057/parallelizing-a-1d-matrix-multiplication-using-openmp