MATRIX MULTIPLICATION REPORT

Problem Statement

Implementation of Matrix Multiplication on single core CPU, multicore CPU, GPU and Tensor Cores within GPU. Benchmarking the program for different input sizes and data types (int 32, float, double)

Implementation of Matrix Multiplication on single core CPU

Introduction

Matrix multiplication is one of the most fundamental operations in computer science, machine learning, numerical computation, and scientific simulations.
However, the straightforward approach (naive multiplication) is often slow due to poor use of the memory hierarchy.

In this project, our goal is to understand how different versions of the same algorithm behave and why some are faster than others.

We compare them based on:

- How the algorithm works
- How well it uses memory (spatial and temporal locality)
- Their time and space complexity
- What performance optimizations can be applied
- How simply changing the order of loops can make huge differences in speed because of cache efficiency

What Problem Does This Solve?

The main challenge in matrix multiplication is not only the number of arithmetic operations—but **how efficiently data is accessed in memory**.

Modern CPUs are fast, but memory access is slow. So the real performance problem is how we read and reuse data, not just how much we compute.

In this project, we try to fix this by:

- **Improving cache locality**
- **Reducing cache misses**
- **Optimizing loop structure**
- **Rearranging data for sequential memory access**

Better locality → fewer cache misses → faster execution.

What Are Header Files?

Header files provide declarations of functions, classes, templates, and constants that facilitate modularity and code reuse.
They enable developers to incorporate standardized library functionalities or external interfaces into a program through the #include directive.

C++ Standard Library Header Files

1) #include <iostream>

Provides facilities for standard input and output streams, such as std::cin, std::cout, and std::cerr.
Essential for performing high-level console-based I/O operations.

2) #include <vector>

Introduces the std::vector container, a dynamic array supporting efficient random access and automatic memory management.
Widely used for flexible, resizable data storage in performance-critical applications.

3) #include <random>

Offers a comprehensive framework for random number generation, including engines and probability distributions.
Designed for simulations, statistical modeling, and reproducible scientific computations.

4) #include <chrono>

Provides high-precision time utilities, including clocks, durations, and time points.
Used extensively in benchmarking, performance evaluation, and real-time systems.

5) #include <algorithm>

Contains a wide set of generic algorithms such as sorting, searching, and transformations.
Optimized for use with standard containers and follows the principles of generic programming.

6) #include <cmath>

Includes mathematical functions for floating-point computations, such as trigonometric, exponential, and logarithmic operations.
Fundamental for numerical analysis and scientific computing.

7) #include <cstring>

Supplies functions for manipulating C-style character arrays, including copying, comparing, and searching operations.
Primarily used for low-level memory and string handling in legacy or performance-constrained environments.

createMatrix() Algorithm

Goal

Generate a matrix filled with random values in a deterministic and reproducible way.

How the Algorithm Works

- Allocates a 1D vector of size rows × cols.
- Uses **mt19937** (Mersenne Twister) for fast pseudorandom number generation.
- Generates values from **uniform distribution [-5, 5]**.
- Stores values in **row-major format**, meaning:
- A[i][j] = A[i * cols + j]

Pseudo Code

function createMatrix(rows, cols, seed):

allocate vector M of size rows * cols

initialize rng with seed

for each index i in M:

M[i] = random number in [-5, 5]

return M

How Matrix Multiplication Works

Given:

- A = m × n
- B = n × p
- C = m × p

Each output element:

C[i][j] = Σ over k (A[i][k] * B[k][j])

The implementation optimizes *how we access A and B*, which determines:

- CPU cache usage
- Runtime performance

- Memory bandwidth efficiency

Code Implementation

Naive Method (i → j → k)

The naive algorithm is the simplest and most direct method for matrix multiplication.It uses three nested loops to compute each element of the output matrix. It also serves as the baseline to compare improved algorithms.

Working

- For every row i and column j, calculate the dot product with all k.
- A's row access is sequential (good).
- B's column access is **non-sequential** and **cache-poor**.

Pseudo Code

Input: Matrix A (size m × n), Matrix B (size n × p)

Output: Matrix C (size m × p) where C = A × B

Step 1: Initialize result matrix C with all zeros

Step 2: FOR each row i from 0 to m:

FOR each column j from 0 to p:

Initialize sum = 0

FOR each index k from 0 to n:

sum = sum + A[i][k] × B[k][j]

Set C[i][j] = sum

Step 3: Return matrix C

Memory Locality

Access Pattern	Locality
A row access	Good (sequential)
B column access	**Bad** (strided, cache misses)

Interchanged Method (i → k → j)

The loop-interchanged method is simply a smarter rearrangement of the naive matrix multiplication loops.In the naive approach, we repeatedly fetch the same values from matrix A many times, which wastes time and cache space. In this improved version, we change the order of the loops so that once we pick an element from A, we immediately use it everywhere it is needed before moving on.

Working

- Swap inner loops so that each A[i][k] is reused for all j.
- Now B is accessed row-wise → **cache-friendly**.

Pseudo Code

Input: Matrix A (size m × n), Matrix B (size n × p)

Output: Matrix C (size m × p) where C = A × B

Step 1: Initialize result matrix C with all zeros

Step 2: FOR each row i from 0 to m:

FOR each index k from 0 to n:

Load value a = A[i][k]

FOR each column j from 0 to p:

C[i][j] = C[i][j] + a × B[k][j]

Step 3: Return matrix C

Memory Locality

Access Pattern	Locality
A row	Good
B row (instead of column)	**Very good**
C row updated sequentially	Excellent

This is significantly faster than naive.

Transpose-Based Method

One of the biggest problems in the naive multiplication method is that matrix B is accessed column by column, which is slow because memory is arranged row by row.
Here we turn the columns of B into rows by transposing it. After transposing, the multiplication becomes much more memory-friendly.

Working

1. Transpose B into Bt so row access replaces column access.
2. Multiply using the form:
3. C[i][j] = dot(A[i,*], Bt[j,*])

Pseudo Code

Input: Matrix A (size m × n), Matrix B (size n × p)

Output: Matrix C (size m × p) where C = A × B

Step 1: Compute transpose BT of matrix B (size p × n)

Step 2: Initialize result matrix C with all zeros

Step 3: FOR each row i from 0 to m:

FOR each column j from 0 to p:

Initialize sum = 0

FOR each index k from 0 to n:

sum = sum + A[i][k] × BT[j][k]

Set C[i][j] = sum

Step 4: Return matrix C

Memory Locality

Access Pattern	Locality
A row	Excellent
Bt row	Excellent (formerly B column)
C row writes	Excellent

This method changes the way we access matrix B so that we read it in straight, continuous rows instead of jumping around in memory. This makes the CPU access data faster and improves performance.

Special Optimizations in the Code

1.Pointer Aliasing Minimization

const T* arow = &A[i*n];

const T* brow = &B[k*p];

T* crow = &C[i*p];

This reduces repeated index computations.

2. Row-Major Data Access

All loops are arranged to ensure **sequential memory scanning**.

3. Transpose to Fix Poor Locality

Instead of accessing B column-wise, we precompute Bt:

Bt[j*n + k] = B[k*p + j];

4. Avoiding Dynamic Allocations Inside Loops

All vectors (C, Bt) are preallocated.

5.Improved Arithmetic Intensity

Interchanged & transpose methods minimize repeated multiplications and maximize data reuse

Blocked Matrix Multiplication Algorithm with Autotuning

Introduction

The blocked matrix multiplication algorithm and associated autotuning mechanism are described in this report. Although matrix multiplication is a basic computing process, the straightforward method is quite slow for big matrices. By using computer memory more intelligently, the blocked algorithm makes it significantly faster.

What Problem Does This Solve?

The basic approach of multiplying two huge matrices requires the computer to retrieve data from memory millions of times. When compared to calculations, memory access is slow. This is solved by the blocked algorithm, which speeds up the entire process by reusing data that has previously been loaded into the fast memory, also known as the cache.

Blocked Algorithm

Basic Concept

Instead of multiplying entire rows and columns at once, the blocked algorithm divides the matrices into smaller square pieces called **blocks**.

Benefits of using blocks are:

- Small blocks fit into the computer's fast cache memory
- Once data is in cache, we can reuse it many times
- This reduces the number of slow memory accesses

How the Algorithm Works

The algorithm uses six nested loops organized in two groups:

Outer loops (blocking loops):

1. Loop through blocks of rows in matrix A (using variable `ii`)
2. Loop through blocks of columns in matrix A and rows in matrix B (using variable `kk`)
3. Loop through blocks of columns in matrix B (using variable `jj`)

Inner loops (computation loops):
4. Loop through individual rows within the current block (using variable `i`)
5. Loop through individual columns within the current block (using variable `k`)
6. Loop through individual elements to do the actual multiplication (using variable `j`)

Pseudocode

Algorithm: Blocked Matrix Multiplication

Input: Matrix A (size m × n), Matrix B (size n × p), Block size bs
Output: Matrix C (size m × p) where C = A × B

Step 1: Initialize result matrix C with all zeros

Step 2: FOR each block row (ii from 0 to m, jumping by bs):
FOR each block column/row (kk from 0 to n, jumping by bs):
FOR each block column (jj from 0 to p, jumping by bs):

```
Calculate block boundaries:
- i_max = minimum of (ii + bs, m)
- k_max = minimum of (kk + bs, n)
- j_max = minimum of (jj + bs, p)

FOR each row i from ii to i_max:
  FOR each column k from kk to k_max:
    Load value a = A[i][k]

    FOR each column j from jj to j_max:
      C[i][j] = C[i][j] + a × B[k][j]
```

Step 3: Return matrix C

Code Implementation

The code has **two variants** of the blocked algorithm:

1. Blocked Naive Algorithm

This version applies blocking to the naive (i-j-k) loop order:

template<typename T>
vector<T> matmul_blocked_naive(const vector<T>& A, const vector<T>& B,
int m, int n, int p, int bs = 128) {
vector<T> C(static_cast<size_t>(m) * p, 0);

// Outer loops - work on blocks
for (int ii = 0; ii < m; ii += bs)
for (int kk = 0; kk < n; kk += bs)
for (int jj = 0; jj < p; jj += bs) {

```
    // Handle edges when block doesn't fit perfectly
    int i_max = min(ii + bs, m);
    int k_max = min(kk + bs, n);
    int j_max = min(jj + bs, p);

    // Inner loops - naive order (i-k-j)
    for (int i = ii; i < i_max; ++i) {
      T* crow = &C[(size_t)i * p + jj];
      const T* arow = &A[(size_t)i * n];

      for (int k = kk; k < k_max; ++k) {
        T a = arow[k];
        const T* brow = &B[(size_t)k * p + jj];
        int J = j_max - jj;
```

```
        int j = 0;

        // Process 4 elements at once (unrolled loop)
        for (; j + 3 < J; j += 4) {
          crow[j]     += a * brow[j];
          crow[j + 1] += a * brow[j + 1];
          crow[j + 2] += a * brow[j + 2];
          crow[j + 3] += a * brow[j + 3];
        }

        // Process remaining elements
        for (; j < J; ++j)
          crow[j] += a * brow[j];
      }
    }
  }
```

return C;
}

2. Blocked Interchanged Algorithm

In this method we combine blocking with loop interchange for better performance:

template<typename T>
vector<T> matmul_blocked_interchange(const vector<T>& A, const vector<T>& B,
int m, int n, int p, int bs) {
vector<T> C(static_cast<size_t>(m) * p, 0);

// pointers for faster indexing
const T* Ap = A.data();
const T* Bp = B.data();
T* Cp = C.data();

// Outer loops - iterate over tiles
for (int ii = 0; ii < m; ii += bs) {
int i_max = min(ii + bs, m);
for (int kk = 0; kk < n; kk += bs) {
int k_max = min(kk + bs, n);
for (int jj = 0; jj < p; jj += bs) {
int j_max = min(jj + bs, p);

```
    // Inner loops - interchanged order (i-k-j)
    for (int i = ii; i < i_max; ++i) {
      T* crow = Cp + (size_t)i * p + jj;
      const T* arow = Ap + (size_t)i * n;

      for (int k = kk; k < k_max; ++k) {
        T a = arow[k];  // Load A[i][k] once
        const T* brow = Bp + (size_t)k * p + jj;
        int J = j_max - jj;
        int j = 0;

        // Unroll by 4 (reduces loop overhead)
        for (; j + 3 < J; j += 4) {
```

```
          crow[j]     += a * brow[j];
          crow[j + 1] += a * brow[j + 1];
          crow[j + 2] += a * brow[j + 2];
          crow[j + 3] += a * brow[j + 3];
        }
        for (; j < J; ++j)
          crow[j] += a * brow[j];
      }
    }
  }
}
```

}
return C;
}

Special Optimizations in the Code

1. Using Pointers

- Instead of accessing elements like `C[i][j]`, the code uses pointers (`T* crow`)
- This is faster because it avoids recalculating array positions repeatedly

2. Loading Data Once

- The value `A[i][k]` is loaded into variable `a` and reused multiple times
- This reduces memory reads significantly

3. Loop Unrolling

- The innermost loop processes 4 elements at once
- This reduces the overhead of loop control (checking conditions, incrementing counters)
- It also helps the processor work on multiple operations simultaneously

4. Boundary Checking

- The code handles cases where matrix dimensions aren't perfectly divisible by the block size
- Variables `i_max`, `k_max`, `j_max` ensure we don't go out of bounds

5. Raw Pointers (Blocked Interchanged Only)

- Uses `A.data()`, `B.data()`, `C.data()` to get raw pointers
- Provides even faster indexing compared to vector indexing
- Gives compiler more optimization opportunities

6. Two Optimization Levels

- **Blocked Naive**: Applies blocking to standard loop order
- **Blocked Interchanged**: Combines blocking + loop interchange for maximum performance

The Autotuning System

Why Autotuning is Needed?

The best block size depends on the specific computer hardware:

- Cache sizes (L1, L2, L3 caches)
- Cache line size
- Processor architecture

A block size that works well on one computer might be slow on another. Autotuning finds the best block size by testing different options.

Working of Autotuning

The autotuning system tests several block sizes and picks the fastest one.

Block sizes tested: 16, 32, 48, 64, 96, 128

Why these sizes?

- **16**: Minimum useful size, but might be too small
- **32, 48, 64**: Good middle ground, often optimal
- **96, 128**: Larger blocks, might be too big for cache

Autotuning Code

The code includes **separate autotuning** for both blocked algorithms:

Autotuning for Blocked Naive

int best_bs_naive = 128; // Start with default
{
cout << "Autotuning block size for matmul_blocked_naive...\n";
vector<int> cands = {16, 32, 48, 64, 96, 128};
double best_time = 1e308; // Very large number

// Try each block size
for (int bs : cands) {
auto t0 = HRClock::now();
auto Ctry = matmul_blocked_naive(A, B, m, n, p, bs);
auto t1 = HRClock::now();
double s = sec_between(t0, t1);

```
cout << " bs=" << bs << " -> " << s << " s\n";

// Remember the fastest one
if (s < best_time) {
  best_time = s;
  best_bs_naive = bs;
}

// Verify correctness
if (!almost_equal(C_naive, Ctry)) {
```

```
  cerr << "Autotune (naive): result mismatch at bs=" << bs << "\n";
  return 1;
}
```

}
cout << "Autotune (naive) picked bs = " << best_bs_naive << "\n";
}

Autotuning for Blocked Interchanged

int best_bs_interchange = 128; // Start with default
{
cout << "Autotuning block size for matmul_blocked_interchange...\n";
vector<int> cands = {16, 32, 48, 64, 96, 128};
double best_time = 1e308;

// Try each block size
for (int bs : cands) {
auto t0 = HRClock::now();
auto Ctry = matmul_blocked_interchange(A, B, m, n, p, bs);
auto t1 = HRClock::now();
double s = sec_between(t0, t1);

```
cout << " bs=" << bs << " -> " << s << " s\n";

// Remember the fastest one
if (s < best_time) {
  best_time = s;
  best_bs_interchange = bs;
}

// Verify correctness
if (!almost_equal(C_naive, Ctry)) {
  cerr << "Autotune (interchanged): result mismatch at bs=" << bs <<
"\n";
  return 1;
}
```

}
cout << "Autotune (interchanged) picked bs = " << best_bs_interchange << "\n";
}

Autotuning Algorithms

Algorithm 1: Autotuning for Blocked Naive

Input: Matrices A and B, List of candidate block sizes
Output: Optimal block size for blocked naive algorithm

Step 1: Create list of block sizes to test: [16, 32, 48, 64, 96, 128]
Step 2: Set best_time to infinity
Step 3: Set best_bs_naive to 128 (default)

Step 4: FOR each candidate block size:
Start timer
Run matmul_blocked_naive with this block size
Stop timer and record execution time

```
Verify result matches naive algorithm (correctness check)

IF execution time < best_time:
  best_time = execution time
  best_bs_naive = this block size

Print "bs=X -> Y seconds"
```

Step 5: Print "Autotune (naive) picked bs = best_bs_naive"
Step 6: Return best_bs_naive as the optimal block size

Algorithm 2: Autotuning for Blocked Interchanged

Input: Matrices A and B, List of candidate block sizes
Output: Optimal block size for blocked interchanged algorithm

Step 1: Create list of block sizes to test: [16, 32, 48, 64, 96, 128]
Step 2: Set best_time to infinity
Step 3: Set best_bs_interchange to 128 (default)

Step 4: FOR each candidate block size:
Start timer
Run matmul_blocked_interchange with this block size
Stop timer and record execution time

```
Verify result matches naive algorithm (correctness check)

IF execution time < best_time:
  best_time = execution time
  best_bs_interchange = this block size

Print "bs=X -> Y seconds"
```

Step 5: Print "Autotune (interchanged) picked bs = best_bs_interchange"
Step 6: Return best_bs_interchange as the optimal block size

Performance Results

Based on the test results from the new code (2048×2048 matrices):

Overall Algorithm Comparison

Algorithm	Execution Time	Performance (GFLOPS)
Naive	67.1084 s	0.256002
Transpose-B	8.23425 s	2.09801
Loop-Interchanged	4.05504 s	4.23667
Blocked Naive (bs=64)	1.8354 s	**9.36007**
Blocked Interchanged (bs=48)	**1.69947 s**	**10.109**

Table 1: Performance comparison of all matrix multiplication algorithms

Autotuning Results for Blocked Naive

Block Size	Execution Time
16	4.21944 s
32	2.68331 s
48	1.95446 s
64	1.82055 s←OPTIMAL
96	1.89704 s
128	1.94693 s

Table 2: Autotuning results for Blocked Naive algorithm

Autotuning Results for Blocked Interchanged

Block Size	Execution Time
16	4.19837 s
32	2.51499 s
48	2.4046 s
64	1.83855 s
96	1.69815 s←OPTIMAL
128	1.73419 s

Table 3: Autotuning results for Blocked Interchanged algorithm

Key Findings

- **Naive Method**

  ->Time: 67.1 s, Performance: 0.256 GFLOPS
  ->Slowest method.
  ->Main reason: It accesses B column-wise, which causes lots of cache misses.
  ->It has very poor memory locality (CPU waits for the data, so it slows down).

- **Loop-Interchanged Method**

  ->Time: ~4–6 s, Performance: 3-4 GFLOPS
  ->Around 10× faster than naive. (A[i][k] reused many timesB accessed row-wise, not column-wise).
  ->Much better cache usage.

- **Transpose-Based Method**

  ->Time: ~8 s, Performance: 2-3 GFLOPS
  ->It is almost 15× faster than naive.
  ->Transposing B makes all memory access sequential.

- **Blocked Naive**

  ->(Block size 64 is optimal)
  ->Fastest execution time: 1.82055 seconds (during autotuning).
  ->Final benchmark result: 1.83544 s → 9.36007 GFLOPS.
  ->It is almost 36.6× faster than naive algorithm.

- **Blocked Interchanged**

  (Block size 96 is optimal)
  ->Fastest execution time: 1.69815 seconds (during autotuning).
  ->Final benchmark result: 1.69947 s → 10.109 GFLOPS.
  ->It is almost 39.5× faster than naive algorithm and is the fastest overall method.

- **Why different optimal block sizes?**
  ->Blocked Naive (bs=64): Naive order within blocks benefits from moderate-sized tiles.
  ->Blocked Interchanged (bs=96): Interchanged order has better cache reuse, works efficiently with slightly larger blocks.

- **Smaller blocks (16, 32) are slower because:**
  ->It spends too much time on loop overhead and there is not enough data reuse within each block.

- **Larger blocks (128) are slower because:**
  ->The Blocks may exceed L1/L2 cache capacity and there are more cache misses - data evicted before reuse.

- **Overall improvement over naive:**
  ->Naive algorithm: 67.1084 seconds, 0.256 GFLOPS,
  ->Best blocked algorithm: 1.69947 seconds, 10.109 GFLOPS.
  ->Speedup: 39.5× faster.
  ->Performance gain: 39.4× more GFLOPS,

- **Accuracy confirmed:** All matrix multiplication versions produced correct results (within tolerance).

Why These Techniques Work?

Cache Memory

In Modern computers, we have a memory hierarchy:

- **L1 Cache**: Very fast, very small (about 32 KB)
- **L2 Cache**: Fast, small (about 256 KB)
- **L3 Cache**: Medium speed, larger (several MB)
- **Main Memory (RAM)**: Slow, very large (several GB)

Here, the blocked algorithm keeps data in the fast cache memory instead of repeatedly fetching it from slow RAM.

Data Reuse

When multiplying matrices, we need the same data multiple times. For example:

- Each row of matrix A is used to compute an entire row of the result
- Each column of matrix B is used to compute an entire column of the result

By working on small blocks, we load data into cache once and reuse it many times before moving to the next block.

Temporal and Spatial Locality in the Code

What is Temporal and Spatial Locality?

Temporal Locality: When you access a piece of data, you're likely to access it again soon. The idea is to keep recently-used data in cache.

Spatial Locality: When you access a piece of data, you're likely to access nearby data soon. The idea is to load chunks of nearby data together (cache lines).

Locality Analysis for Each Matrix

Matrix A - Strong Temporal Locality

- Each element `A[i][k]` is loaded once and reused for all elements in a row of matrix B
- In the code: `T a = arow[k];` loads the value once, then uses it multiple times in the inner j-loop
- The same row of A is accessed repeatedly within a block
- **Why this matters**: Loading `A[i][k]` once and reusing it reduces memory reads dramatically

Matrix B - Strong Spatial Locality

- Elements `B[k][j]`, `B[k][j+1]`, `B[k][j+2]`, `B[k][j+3]` are accessed sequentially
- In the code: `const T* brow = &B[(size_t)k * p + jj];` points to a row, then we move along it
- Sequential access means cache lines are used efficiently (one cache line holds multiple consecutive elements)
- **Why this matters**: When we load one element, nearby elements come "for free" in the same cache line

Matrix C - Both Temporal and Spatial Locality

- **Temporal locality**: Each element `C[i][j]` is updated multiple times (once for each k in the block)
- **Spatial locality**: Elements `C[i][j]`, `C[i][j+1]`, `C[i][j+2]`, `C[i][j+3]` are accessed sequentially
- In the code: `T* crow = &C[(size_t)i * p + jj];` then we update `crow[j]`, `crow[j+1]`, etc.
- **Why this matters**: C benefits from both types of locality - values stay in cache for multiple updates AND sequential access is efficient

How Blocking Enhances Locality?

The blocked algorithm amplifies these locality benefits:

1. **For Matrix A**: Within each block, the same rows are reused many times before moving to new rows. This keeps A-data in cache longer.
2. **For Matrix B**: Within each block, we access the same columns repeatedly. Small blocks mean B-data stays in cache across multiple uses.
3. **For Matrix C**: A small block of C stays in cache throughout the entire computation for that block. It gets updated many times without being evicted.

Example

When computing one block of C:

Matrix A block: Read row 0, reuse it many times → Read row 1, reuse it many times (TEMPORAL)

Matrix B block: Read elements 0,1,2,3 in sequence → Read elements 4,5,6,7 in sequence (SPATIAL)

Matrix C block: Update element [0][0] multiple times, update [0][1] multiple times (TEMPORAL + SPATIAL)

Without blocking, C[i][j] might be evicted from cache before its next update, losing temporal locality benefits.

Complexity Analysis

Time Complexity: $O(m \times n \times p)$

- Same as the basic algorithm
- Blocking doesn't reduce the number of operations
- It just makes each operation faster by improving memory access

Space Complexity: $O(m \times p)$

- Only need space for the result matrix C
- The algorithm works in-place on the input matrices

Cache Efficiency: Much better than basic algorithm

- Basic algorithm: Many cache misses
- Blocked algorithm: Few cache misses when block size is optimal

Conclusion

The real challenge in matrix multiplication isn't the math—it's how the computer reads and uses data. The naive method does the correct calculation, but it constantly jumps around in memory, which slows everything down. As we explored smarter techniques, we began to see how much performance can improve simply by organizing memory access more efficiently.

The blocked matrix multiplication algorithm with autotuning significantly improves the performance by optimizing memory usage.

- It splits large matrices into smaller blocks that can fit in the cache.
- It reuses cached data several times before fetching new data.
- It achieves a 26× speed increase compared to the basic algorithm.