# 1. Compiler Flags & Vectorization Control

At the very top, you'll see these lines:

```
#if defined(DISABLE_VECTOR)
#pragma GCC optimize("no-tree-vectorize")
#elif defined(ENABLE_AVX2)
#pragma GCC optimize("Ofast")
#pragma GCC target("avx2","fma")
#endif
```

- **Purpose**: Let you pick at compile-time whether to turn off the compiler's auto-vectorization, enable AVX2+FMA optimizations, or just stick with the default optimizations.

- **How to use**:
  - `-DDISABLE_VECTOR` → completely disables SIMD/vector intrinsics
  - `-DENABLE_AVX2` → enables aggressive AVX2/FMA instructions
  - no flag → uses whatever optimizations your `-O3` (or other) setting gives you by default

# 2. Matrix Multiplication Functions

## 2.1 `naiveMultiply(…)`

```
void naiveMultiply(const vector<double>& A,
                   const vector<double>& B,
                   vector<double>& C,
                   int N) { … }
```

- **What it does**: Implements the straightforward triple-loop algorithm:
  1. Loop over each output row `i`
  2. Loop over each output column `j`
  3. Inner loop k accumulates `A[i][k] * B[k][j]` into `C[i][j]`

- **Why it matters**: This is easy to understand but suffers from poor cache locality—every read of `B[k][j]` may come from a distant memory location.

### 2.2 `tiledMultiply(…)`

```
void tiledMultiply(const vector<double>& A,
                   const vector<double>& B,
                   vector<double>& C,
                   int N,
                   int BS) { … }
```

- **What it does**: Breaks the matrices into small BS×BS blocks ("tiles") and multiplies block by block.
- **Key idea**: By working on one tile at a time, each small piece of A and B stays in cache long enough to be reused, drastically reducing memory traffic.
- **Structure**:
    1. Outer loops (`ii`, `kk`, `jj`) step through the matrix in increments of block size BS.
    2. Inner loops multiply one BS×BS submatrix of A by the corresponding submatrix of B and accumulate into C.

# 3. `main()` Function Breakdown

#### Argument Parsing

```
int N = 1000, BS = 32;
if (argc > 1) N = stoi(argv[1]);
if (argc > 2) BS = stoi(argv[2]);
```

1. **What it does**: Lets you override the default matrix size (`1000`) and block size (`32`) by passing two integers on the command line.

#### Header Output

```
cout << "Matrix size: " << N << " x " << N
     << ", Block size: " << BS << endl;
```

2. Prints what you're about to run, so you don't lose track when doing multiple benchmarks.

#### Vectorization Mode Printout

```
#if defined(DISABLE_VECTOR)
  cout << "Vectorization: DISABLED" << endl;
#elif defined(ENABLE_AVX2)
```

```
  cout << "Vectorization: ENABLED (AVX2/FMA)" << endl;
#else
  cout << "Vectorization: DEFAULT (auto-vectorization)" << endl;
#endif
```

3. Reminds you which optimization path you chose at compile time.

#### Matrix Allocation & Initialization

```
vector<double> A(N*N), B(N*N), C(N*N);
mt19937_64 rng(0);
uniform_real_distribution<double> dist(0.0, 1.0);
for (auto& x : A) x = dist(rng);
for (auto& x : B) x = dist(rng);
```

4. **Why a 1D `vector<double>`?** It guarantees contiguous storage (`&A[0]` is a valid pointer to all elements) and makes indexing with `i*N + j` trivial.
    - **Random fill** ensures you're multiplying "realistic" data each run, so timing is representative.

#### Timing & Running Naive Multiply

```
auto t1 = high_resolution_clock::now();
naiveMultiply(A, B, C, N);
auto t2 = high_resolution_clock::now();
```

5. Uses C++'s chrono library to get high-precision timestamps before and after the call, then reports the delta in seconds.

#### Timing & Running Tiled Multiply

```
fill(C.begin(), C.end(), 0.0);
auto t3 = high_resolution_clock::now();
tiledMultiply(A, B, C, N, BS);
auto t4 = high_resolution_clock::now();
```

6. Resets `C` to zero so your timing isn't "contaminated" by leftover results, then measures the cache-aware version.

**Final Output**

```
cout << "Naive multiplication took " << durNaive.count() << "
seconds" << endl;
cout << "Tiled multiplication took " << durTiled.count() << "
seconds" << endl;
```

7. Clearly shows you the performance gap (if any) between the two approaches.

# 4. Putting It All Together

1. **Compile** with your preferred flags (`-DDISABLE_VECTOR, -DENABLE_AVX2`, or none).

**Run** with optional size/block arguments, e.g.

```
./matmul 2048 64
```

2. **Observe** how the naive version compares to the tiled version—and how vectorization choices affect both.