OpenMp Atax Parallelization

Problem's Introduction

Theme: OpenMp ATAX ($y = A^T(Ax)$) Parallelization

Problem: data copy, matrix accesses; single thread initial code.

Goal: reduce the time-to-solution and scale across multiple thread.

Metrics: time, speedup, efficiency; Clang VS GCC

Dataset:Different size matrices (N,M)

Our Contributes: metodi, profili, risultati comparativi

Problem's Preview

Definition: ATAX is a linear algebra kernel problem that computes: $y = A^T(Ax)$

Data Structures: matrix A (N×M), arrays x (M), tmp (N), y (M)

Step 1: tmp[i] = sum_j A[i][j] * x[j]
per ogni i = 0...N-1

Step 2: y[j] += sum_i A[i][j] * tmp[i]
per ogni j = 0...M-1

Complexity: O(N·M); Memoria: O(N·M)

Peculiarities: Multiple accesses on A, re-use of x and tmp; typically memory-bound on large datasets.

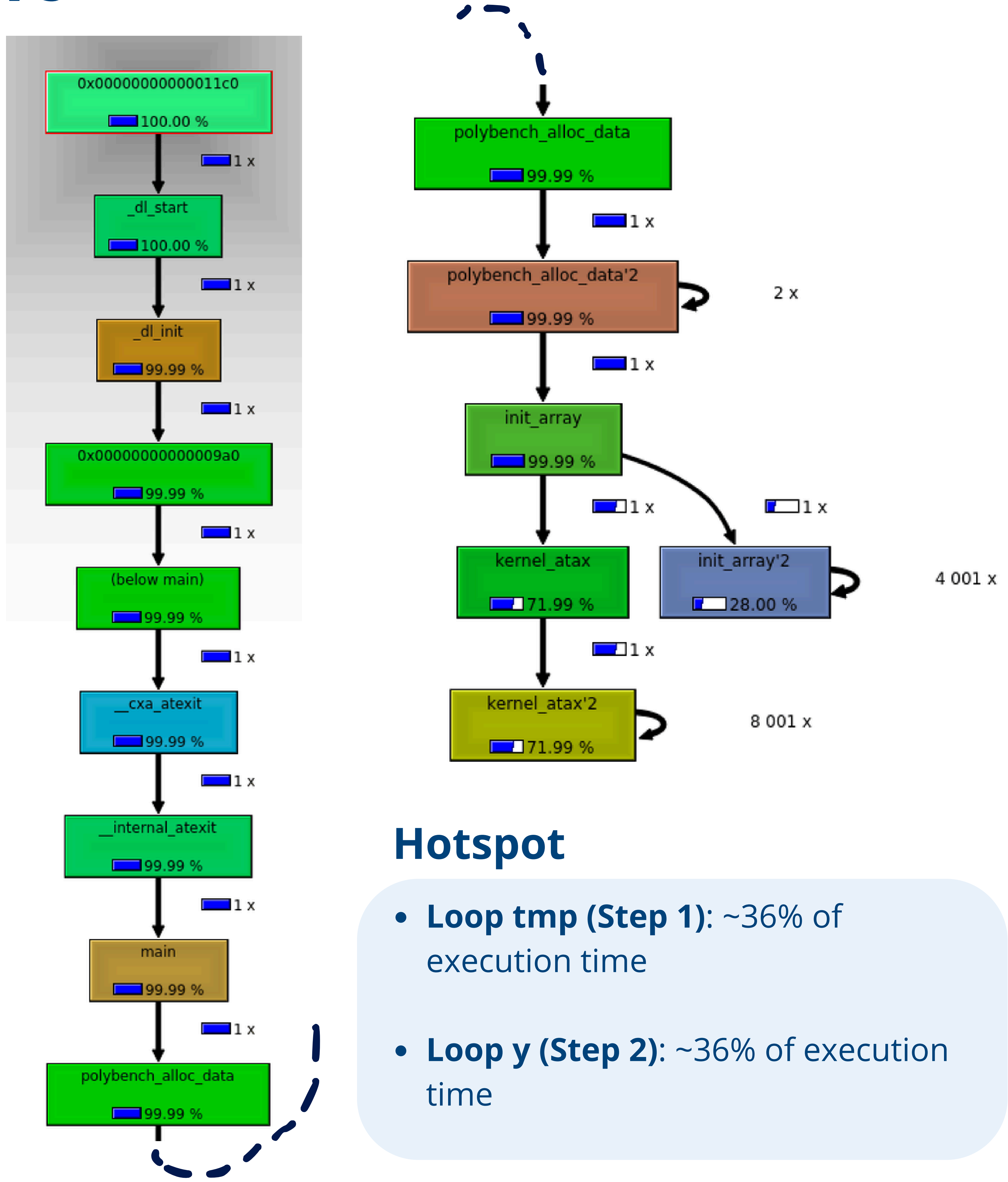
Why OpenMp and Atax call graph’s

Good Candidate

1. **Independent Loop**: ideal for “#pragma omp parallel for” directives.
2. **Reduce Operations**: assignment easily managed by privatization of variables.
3. **Strong Vectorization**: Synergy with simd
4. **Risks**: Data race on y and locality problems

Observations

- **Memory-Bound**: frequent cache and TLS misses when accessing to A matrix.
- **Low IPC**: Limited Instructions Per Cycle due to memory latency.
- **Target**: Dominant loop parallelization and better data locality

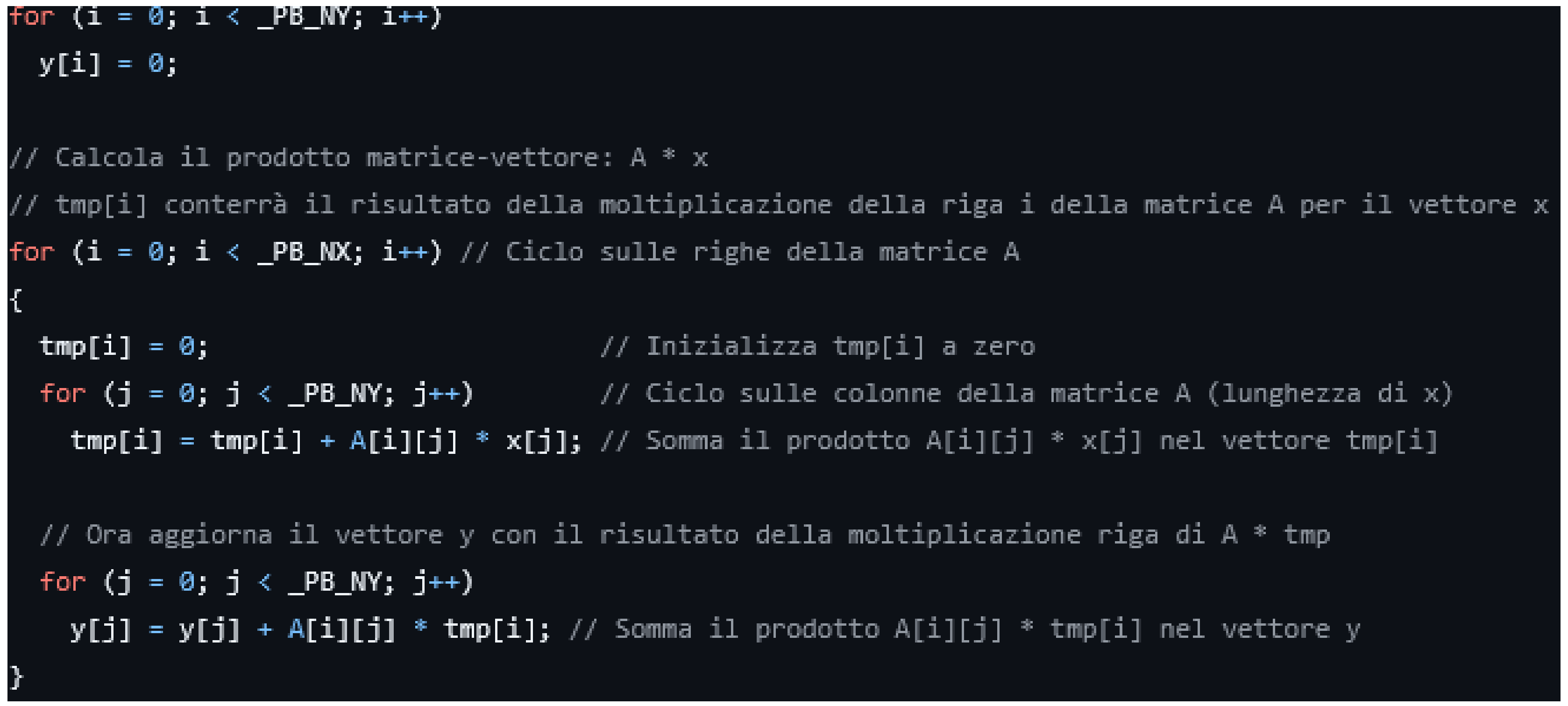


Hotspot

- **Loop tmp (Step 1)**: ~36% of execution time
- **Loop y (Step 2)**: ~36% of execution time

Sequential and base parallelism

V1- Sequential



```
for (i = 0; i < _PB_NY; i++)
  y[i] = 0;

// Calcola il prodotto matrice-vettore: A * x
// tmp[i] conterrà il risultato della moltiplicazione della riga i della matrice A per il vettore x
for (i = 0; i < _PB_NX; i++) // Ciclo sulle righe della matrice A
{
  tmp[i] = 0;                           // Inizializza tmp[i] a zero
  for (j = 0; j < _PB_NY; j++)          // Ciclo sulle colonne della matrice A (lunghezza di x)
    tmp[i] = tmp[i] + A[i][j] * x[j]; // Somma il prodotto A[i][j] * x[j] nel vettore tmp[i]

  // Ora aggiorna il vettore y con il risultato della moltiplicazione riga di A * tmp
  for (j = 0; j < _PB_NY; j++)
    y[j] = y[j] + A[i][j] * tmp[i]; // Somma il prodotto A[i][j] * tmp[i] nel vettore y
}
```

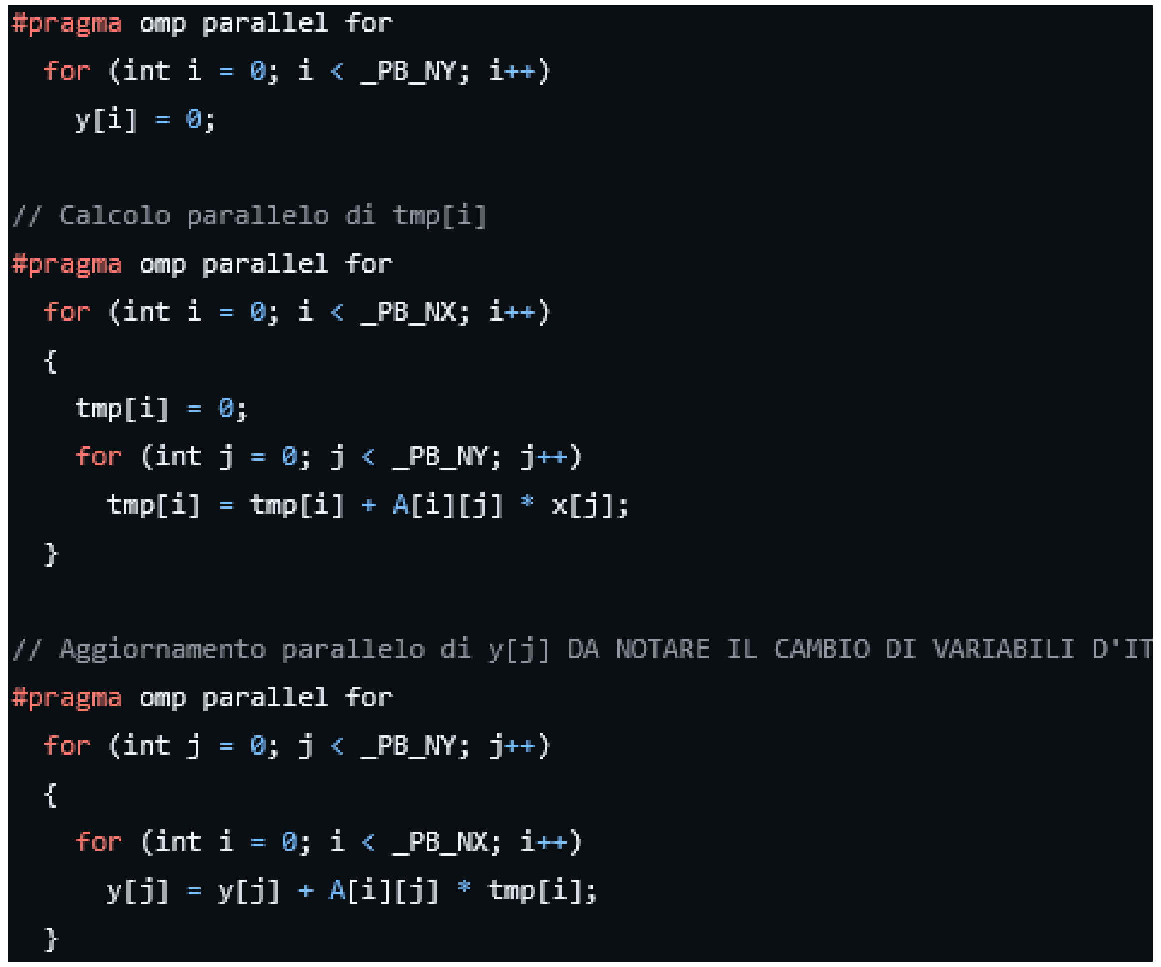
Problems:
1. **Absence of Parallelism:** Every for cycle is sequential

2. **Data dependency risk:** Parallelizing the accumulation phase without sync leads to data race on the shared vector

3. **Potential for spatial locality:** Any attempt to rearrange the loops results in Column-Major access → cache misses.

4.**Low Temporal Locality:** data are read into the cache and evicted before being reused.

V2- Parallel for

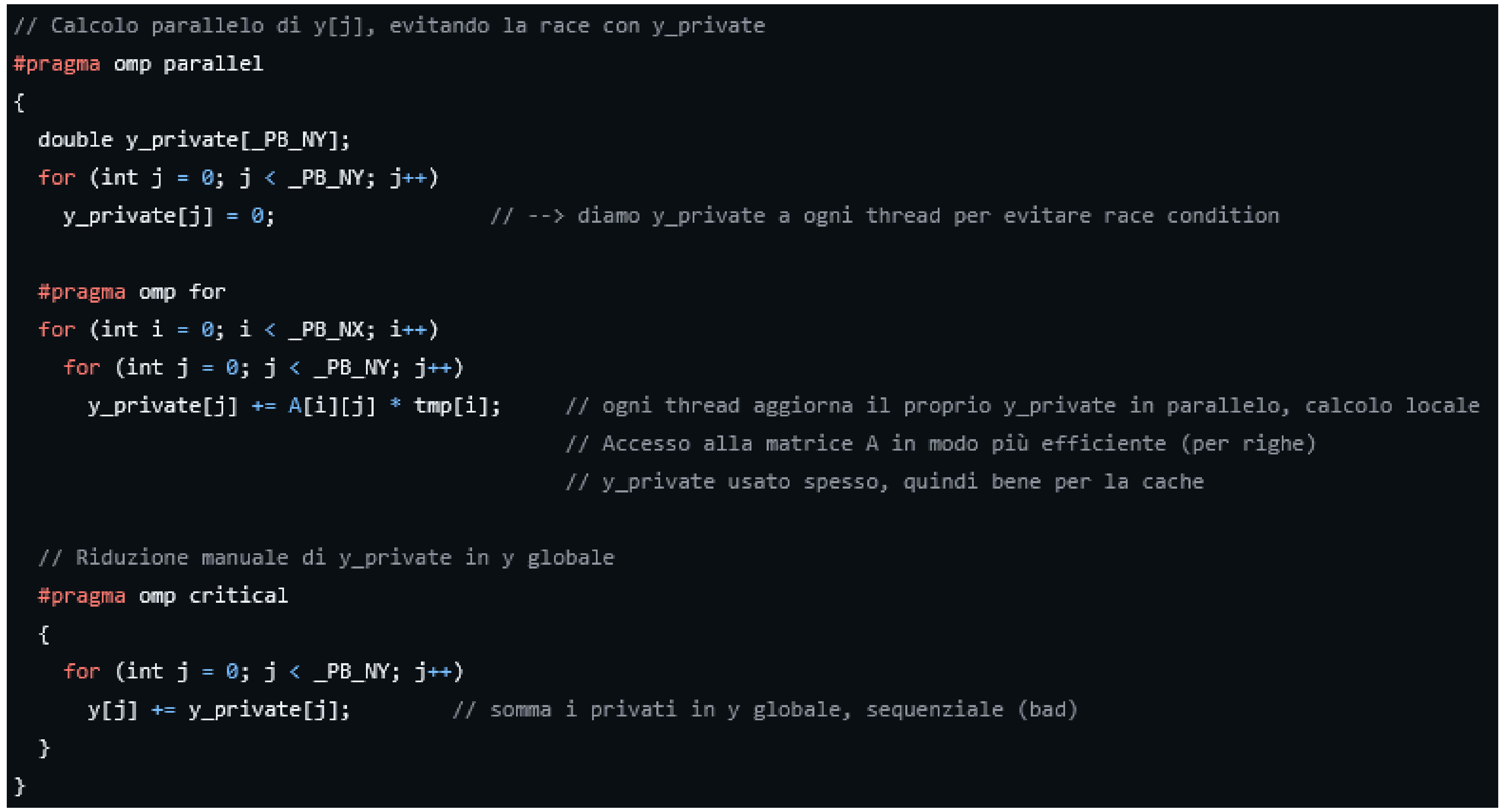


```
#pragma omp parallel for
  for (int i = 0; i < _PB_NY; i++)
    y[i] = 0;

// Calcolo parallelo di tmp[i]
#pragma omp parallel for
  for (int i = 0; i < _PB_NX; i++)
  {
    tmp[i] = 0;
    for (int j = 0; j < _PB_NY; j++)
      tmp[i] = tmp[i] + A[i][j] * x[j];
  }

// Aggiornamento parallelo di y[j] DA NOTARE IL CAMBIO DI VARIABILI D'IT
#pragma omp parallel for
  for (int j = 0; j < _PB_NY; j++)
  {
    for (int i = 0; i < _PB_NX; i++)
      y[j] = y[j] + A[i][j] * tmp[i];
  }
```

1. **Action Taken:** added *parallel for* in every cycle and inverted the loop order to avoid the data race. This inversion ensures that the outer loop is data indipendent (each thread writes to unique y[j] element)
2. **Critical problem:** The loop inversion forces access to the matrix A[i][j] by column. Since the matrix is allocated in Row-Major, this structure violates Spatial Locality.
3. **Consequences:** The kernel became 4-5 times slower than the seqeuential version due to multiple Cache Misses.

Optimized parallel and reduction

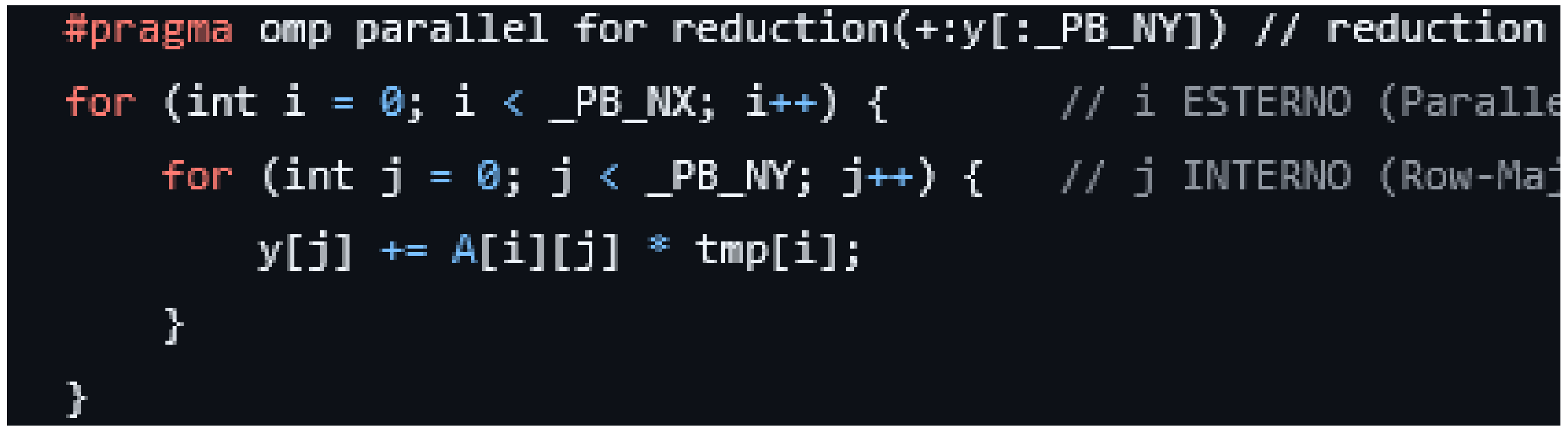
V3 - Parallel for optimized



```
// Calcolo parallelo di y[j], evitando la race con y_private
#pragma omp parallel
{
  double y_private[_PB_NY];
  for (int j = 0; j < _PB_NY; j++)
    y_private[j] = 0;                 // --> diamo y_private a ogni thread per evitare race condition

  #pragma omp for
  for (int i = 0; i < _PB_NX; i++)
    for (int j = 0; j < _PB_NY; j++)
      y_private[j] += A[i][j] * tmp[i];     // ogni thread aggiorna il proprio y_private in parallelo, calcolo locale
                                            // Accesso alla matrice A in modo più efficiente (per righe)
                                            // y_private usato spesso, quindi bene per la cache

  // Riduzione manuale di y_private in y globale
  #pragma omp critical
  {
    for (int j = 0; j < _PB_NY; j++)
      y[j] += y_private[j];         // somma i privati in y globale, sequenziale (bad)
  }
}
```

1. **Action Taken:** Everything like V2, but we reverted the loop order to the cache-friendly row-major structure.
2. **Critical problem:** Reverting the loop reintroduced the Data Race
3. **Solution:** Data Race was managed using a private array for every thread and a manual reduction protected by a *pragma omp critical*
4. **Consequences:** While correct and fast (1.93x faster than sequential), critical section creates an unscalable serial bottleneck. This affect the max speedup according to Amdahl's Law as the number of cores increases.

V4- Reduction

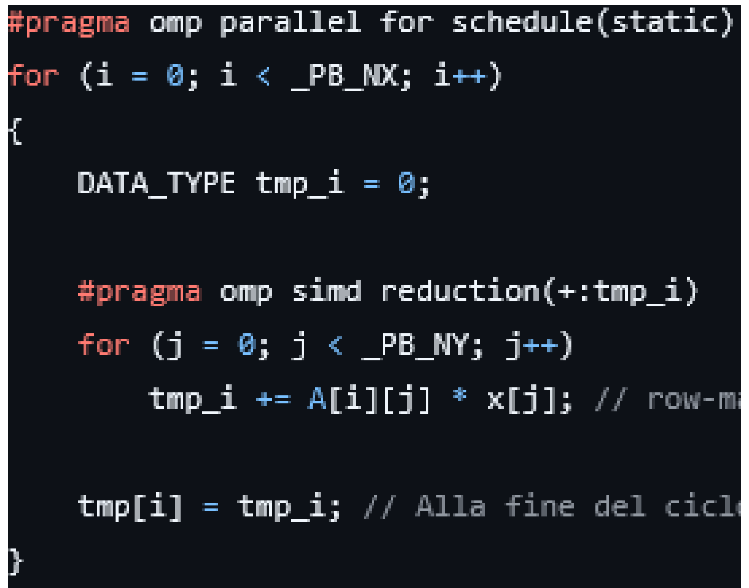


```
#pragma omp parallel for reduction(+:y[:_PB_NY]) // reduction
for (int i = 0; i < _PB_NX; i++) {        // i ESTERNO (Paralle
    for (int j = 0; j < _PB_NY; j++) {    // j INTERNO (Row-Maj
        y[j] += A[i][j] * tmp[i];
    }
}
```

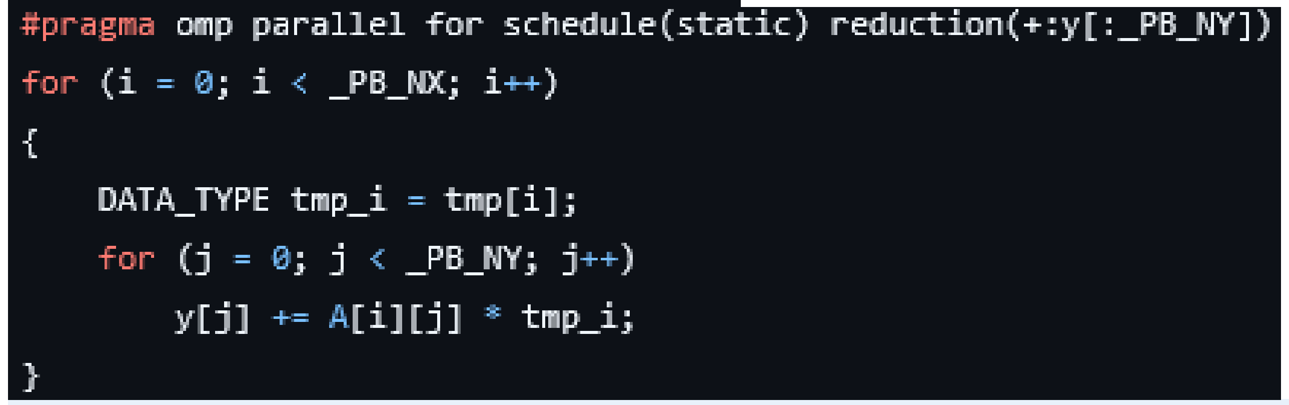
1. **Action Taken:** We replaced the inefficient manual reduction used in v3 with the specific OpenMP array *reduction(+:y[:_PB_NY])* clause.
2. **Solution:** The primary goal was to completely remove the serial bottleneck introduced by the critical section, thereby improving the scalability of the code under Amdahl's Law.
3. **Consequences:** This strategy maintains the crucial Row-Major access pattern (cache-friendly) while providing a correct and highly scalable solution. Kernel is now 1.94x faster than seq.

Two level parallelism and tiling

V5 - Two-level parallelism



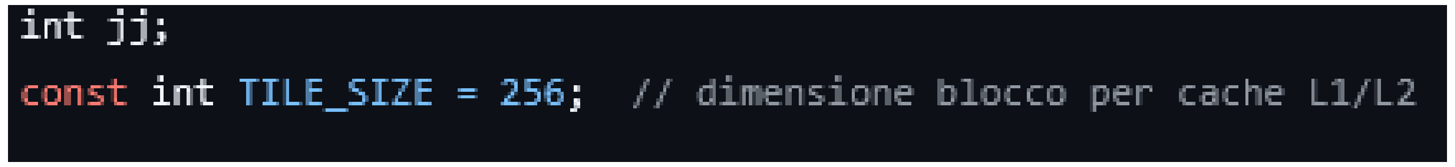
```
#pragma omp parallel for schedule(static)
for (i = 0; i < _PB_NX; i++)
{
    DATA_TYPE tmp_i = 0;

    #pragma omp simd reduction(+:tmp_i)
    for (j = 0; j < _PB_NY; j++)
        tmp_i += A[i][j] * x[j]; // row-m

    tmp[i] = tmp_i; // Alla fine del cicl
}
```



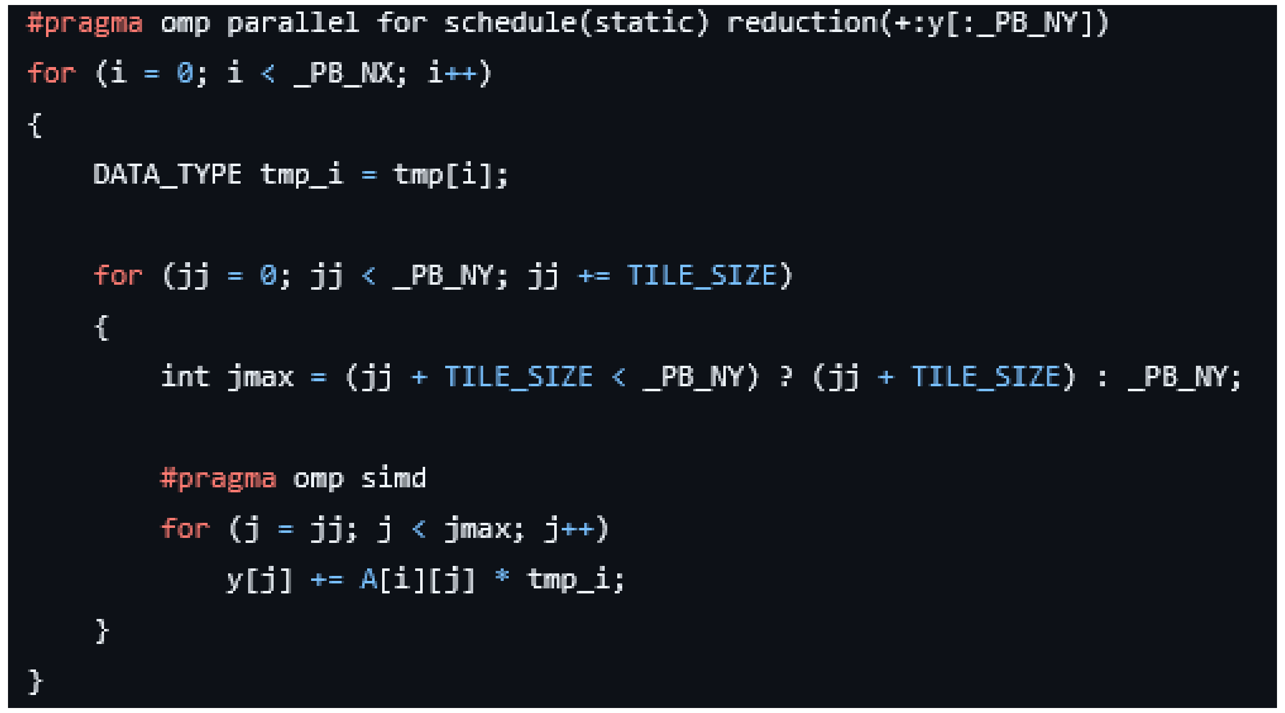
```
#pragma omp parallel for schedule(static) reduction(+:y[:_PB_NY])
for (i = 0; i < _PB_NX; i++)
{
    DATA_TYPE tmp_i = tmp[i];
    for (j = 0; j < _PB_NY; j++)
        y[j] += A[i][j] * tmp_i;
}
```

1. **Action Taken:** We introduced *pragma omp simd* directive to the inner loop of both Phase 1 (A · x) and Phase 2 (AT · tmp).
2. **Solution:** The objective was to achive the maximum possible speedup by introducing a second level od parallelism: SIMD (vectorization). This optimize the exec within each CPU core, complementing the multi-core parallelism achieved by *parallel for*.
3. **Consequences:** We enabled the processor to perform many operation simultaneously, Kernel is 1.98x faster, now the algorithm is computationally saturated and only memory-bound. K

V6- 2-level parallelism - tiling



```
int jj;
const int TILE_SIZE = 256;  // dimensione blocco per cache L1/L2
```

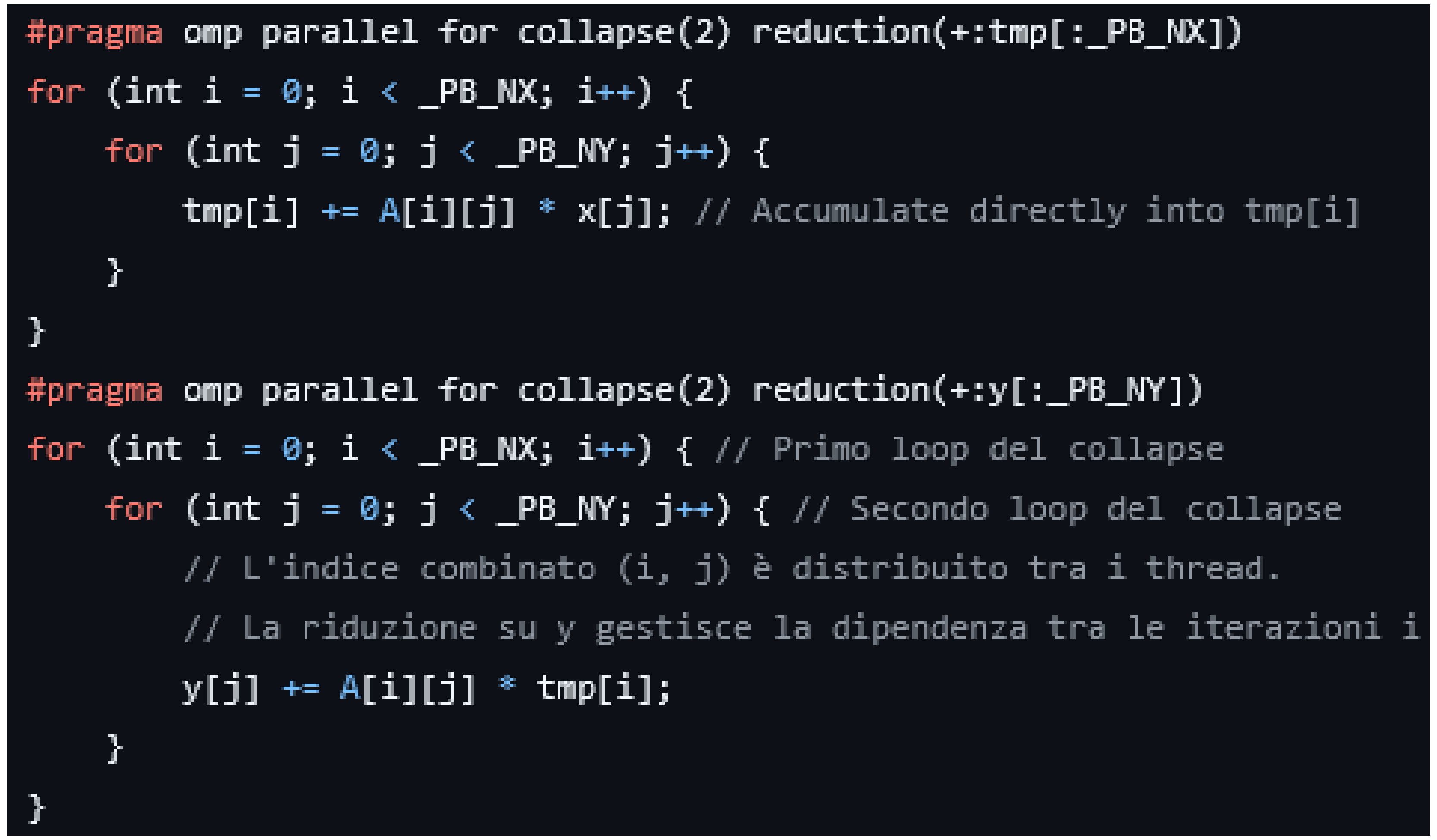


```
#pragma omp parallel for schedule(static) reduction(+:y[:_PB_NY])
for (i = 0; i < _PB_NX; i++)
{
    DATA_TYPE tmp_i = tmp[i];

    for (jj = 0; jj < _PB_NY; jj += TILE_SIZE)
    {
        int jmax = (jj + TILE_SIZE < _PB_NY) ? (jj + TILE_SIZE) : _PB_NY;

        #pragma omp simd
        for (j = jj; j < jmax; j++)
            y[j] += A[i][j] * tmp_i;
    }
}
```

1. **Action Taken:** We introduced loop tiling to the inner loop of phase 2. How did we choose tile_size? → near cache size.
2. **Solution:** The objective was to address the final bottleneck → memory bandwidth. By ensuring that blocks of the vector y remain in L1/L2 cache, we reduced the number of costly accessess to the main memory. So we maximized Temporal Locality.
3. **Consequences:** This version achieves the best speedup overall (2.01x). The minor performance over V5 confirms that the kernel is fully computationallly satured, now it is only a memory bandwidth matter.

Collapse and off-loading (target)

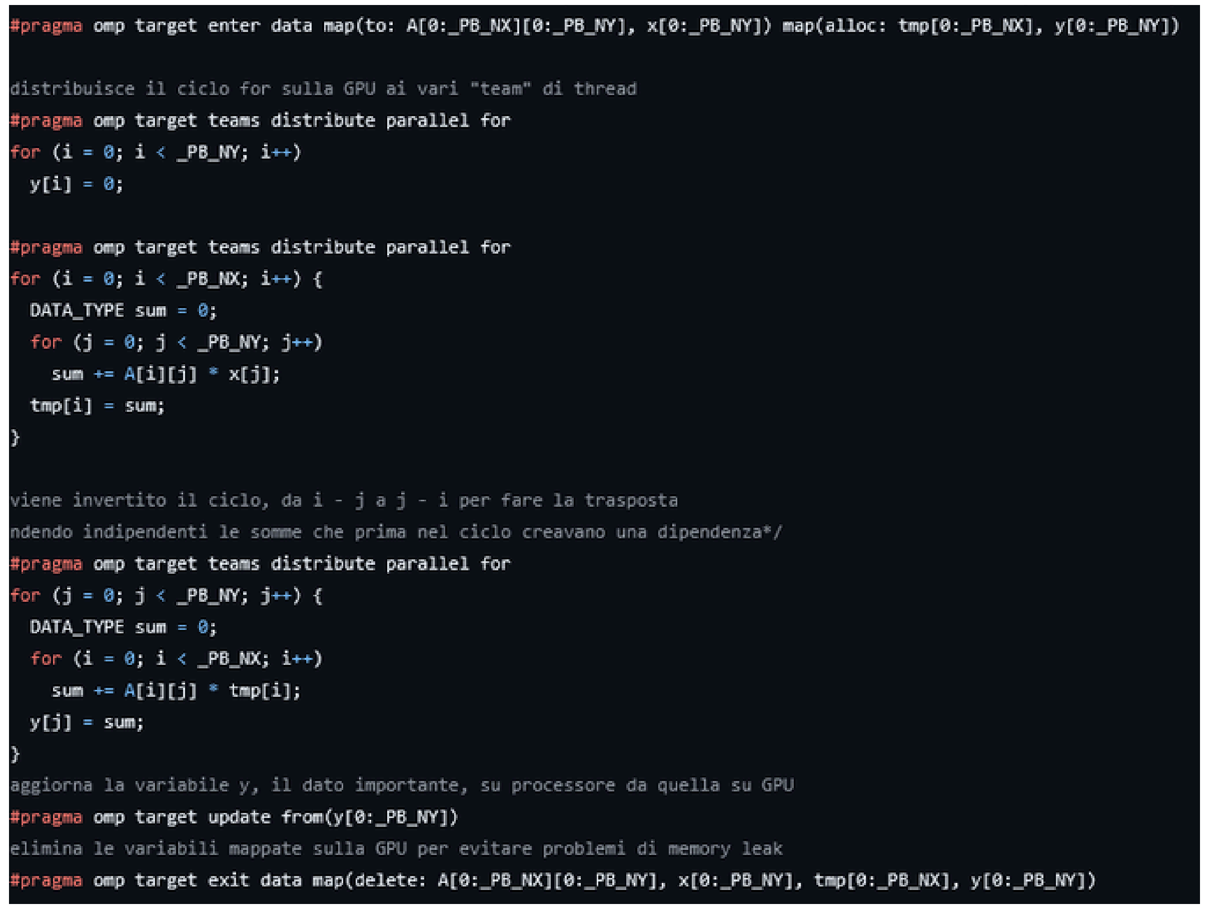
V7 - Collapse



```
#pragma omp parallel for collapse(2) reduction(+:tmp[:_PB_NX])
for (int i = 0; i < _PB_NX; i++) {
    for (int j = 0; j < _PB_NY; j++) {
        tmp[i] += A[i][j] * x[j]; // Accumulate directly into tmp[i]
    }
}
#pragma omp parallel for collapse(2) reduction(+:y[:_PB_NY])
for (int i = 0; i < _PB_NX; i++) { // Primo loop del collapse
    for (int j = 0; j < _PB_NY; j++) { // Secondo loop del collapse
        // L'indice combinato (i, j) è distribuito tra i thread.
        // La riduzione su y gestisce la dipendenza tra le iterazioni i
        y[j] += A[i][j] * tmp[i];
    }
}
```

1. **Action Taken:** We applied *pragma omp parallel for collpase(2)* directive to both phase 1 and phase 2. The objective was to increase the granularity of parallelism by fusing two nested loops.
2. **Critical Problem:** *Data race* → in tmp[i] on phase 1. *Overhead* → in mapping the combined index to memory locations. *Poor Locality* → collapse struggles to preserve the spatial locality (row-major), crucial for performance.
3. **Consequences:** Kernel is 1.80x faster than seq, but worse than the previous optimizations, because complexity and overhead outweighed any potential gain from finer-grained parallelism.

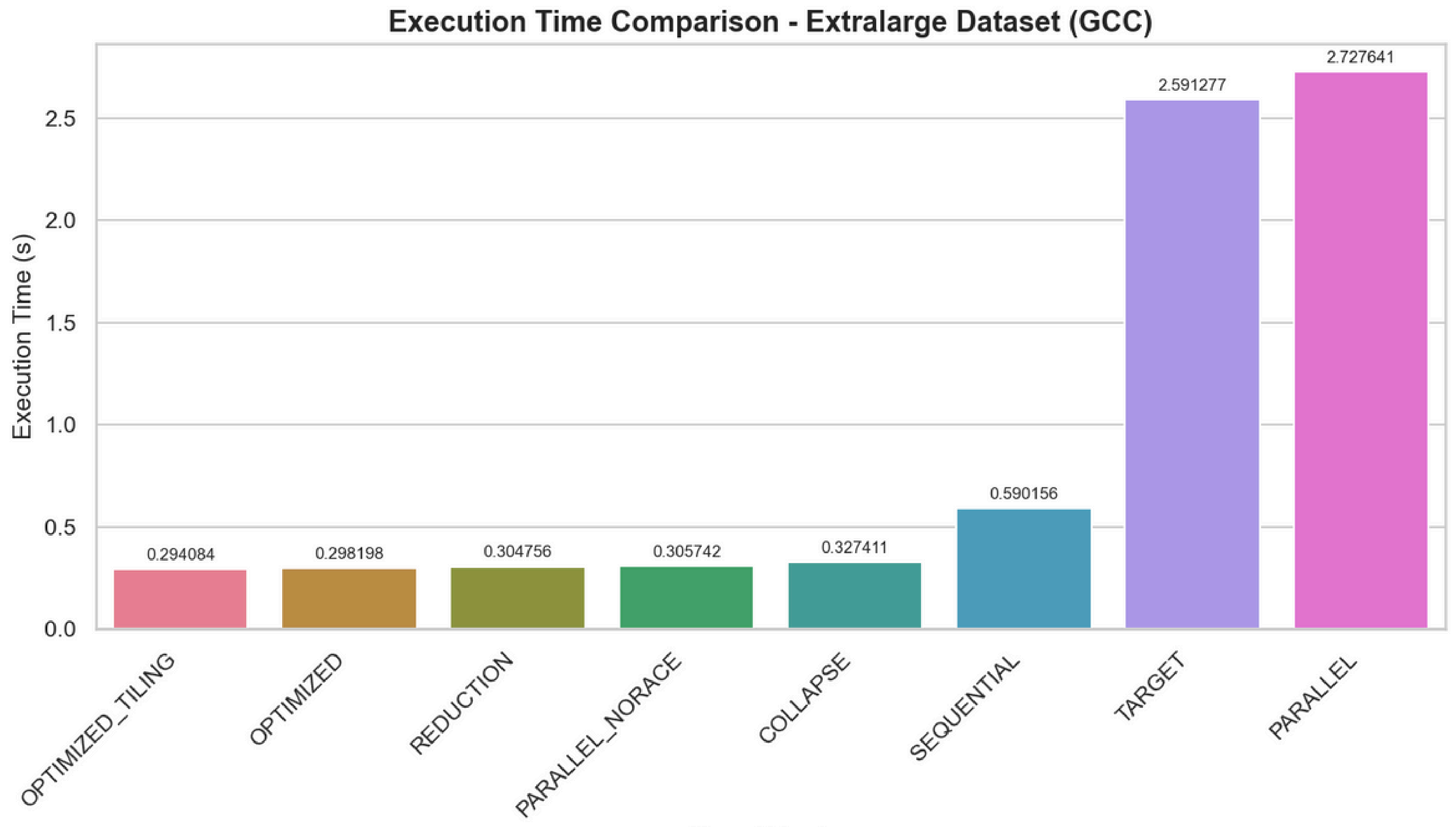
V8 - Target

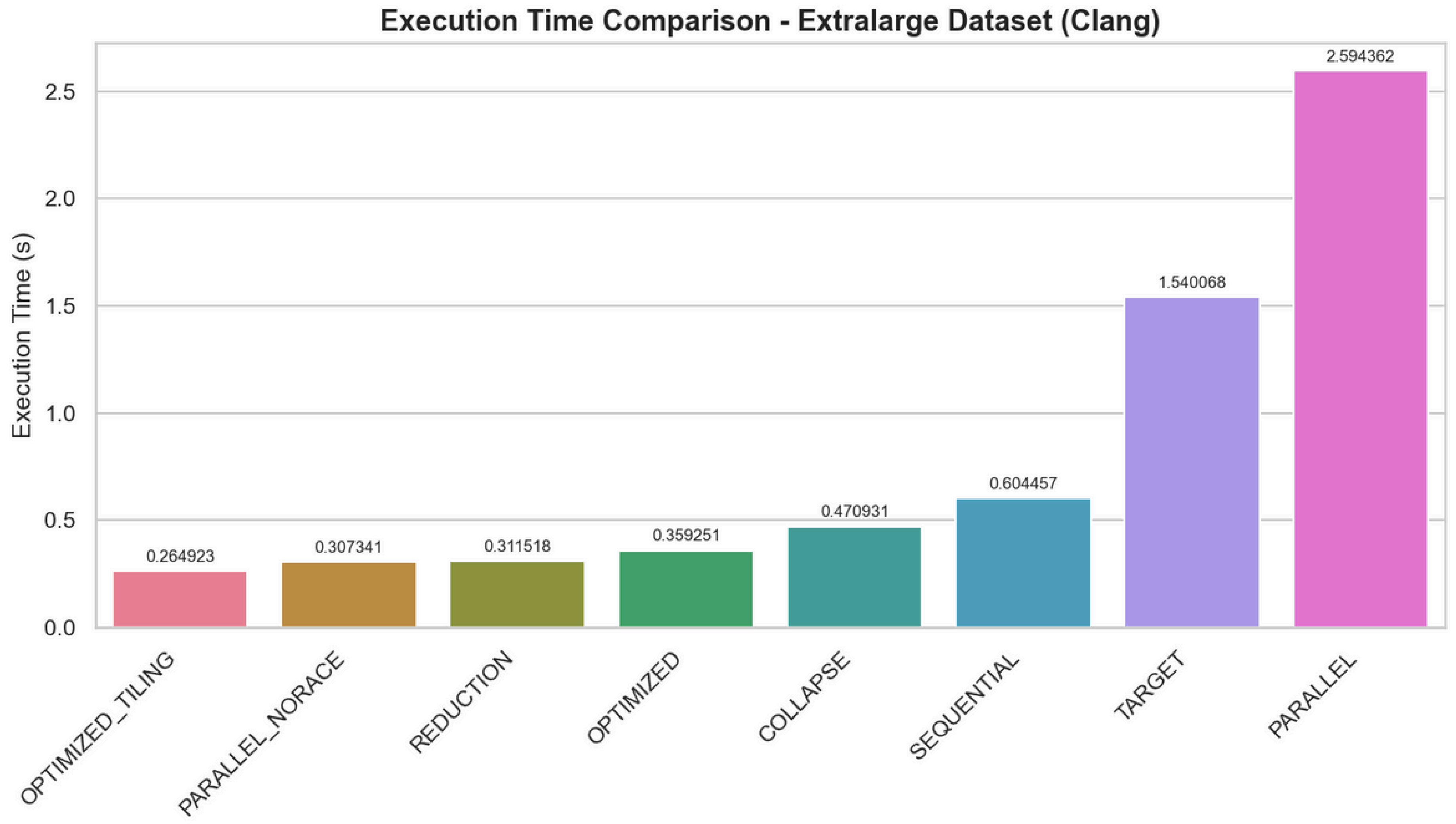


```
#pragma omp target enter data map(to: A[0:_PB_NX][0:_PB_NY], x[0:_PB_NY]) map(alloc: tmp[0:_PB_NX], y[0:_PB_NY])

distribuisce il ciclo for sulla GPU ai vari "team" di thread
#pragma omp target teams distribute parallel for
for (i = 0; i < _PB_NY; i++)
  y[i] = 0;

#pragma omp target teams distribute parallel for
for (i = 0; i < _PB_NX; i++) {
  DATA_TYPE sum = 0;
  for (j = 0; j < _PB_NY; j++)
    sum += A[i][j] * x[j];
  tmp[i] = sum;
}

viene invertito il ciclo, da i - j a j - i per fare la trasposta
ndendo indipendenti le somme che prima nel ciclo creavano una dipendenza*/
#pragma omp target teams distribute parallel for
for (j = 0; j < _PB_NY; j++) {
  DATA_TYPE sum = 0;
  for (i = 0; i < _PB_NX; i++)
    sum += A[i][j] * tmp[i];
  y[j] = sum;
}
aggiorna la variabile y, il dato importante, su processore da quella su GPU
#pragma omp target update from(y[0:_PB_NY])
elimina le variabili mappate sulla GPU per evitare problemi di memory leak
#pragma omp target exit data map(delete: A[0:_PB_NX][0:_PB_NY], x[0:_PB_NY], tmp[0:_PB_NX], y[0:_PB_NY])
```

1. **Action Taken:** The entire ATAX calculation was moved to a device (GPU) using OpenMP *target* directive, so *map, teams, distribute*…. The goal was to utilize the massive parallel capacity of a GPU, especially for a O($N^{2)}$) algorithm.
2. **Solution:** In Phase 2 (y=AT · tmp), the loop order was set to j,i to ensure Data Independence (essential for mass parallel execution). This choice, however, forced Column-Major access to the Row-Major matrix A.
3. **Consequences:** Column-major access results in uncoalesced memory access, so thread teams cannot read adjacent data simultaneously. This limits the kernel speed (0.39x).

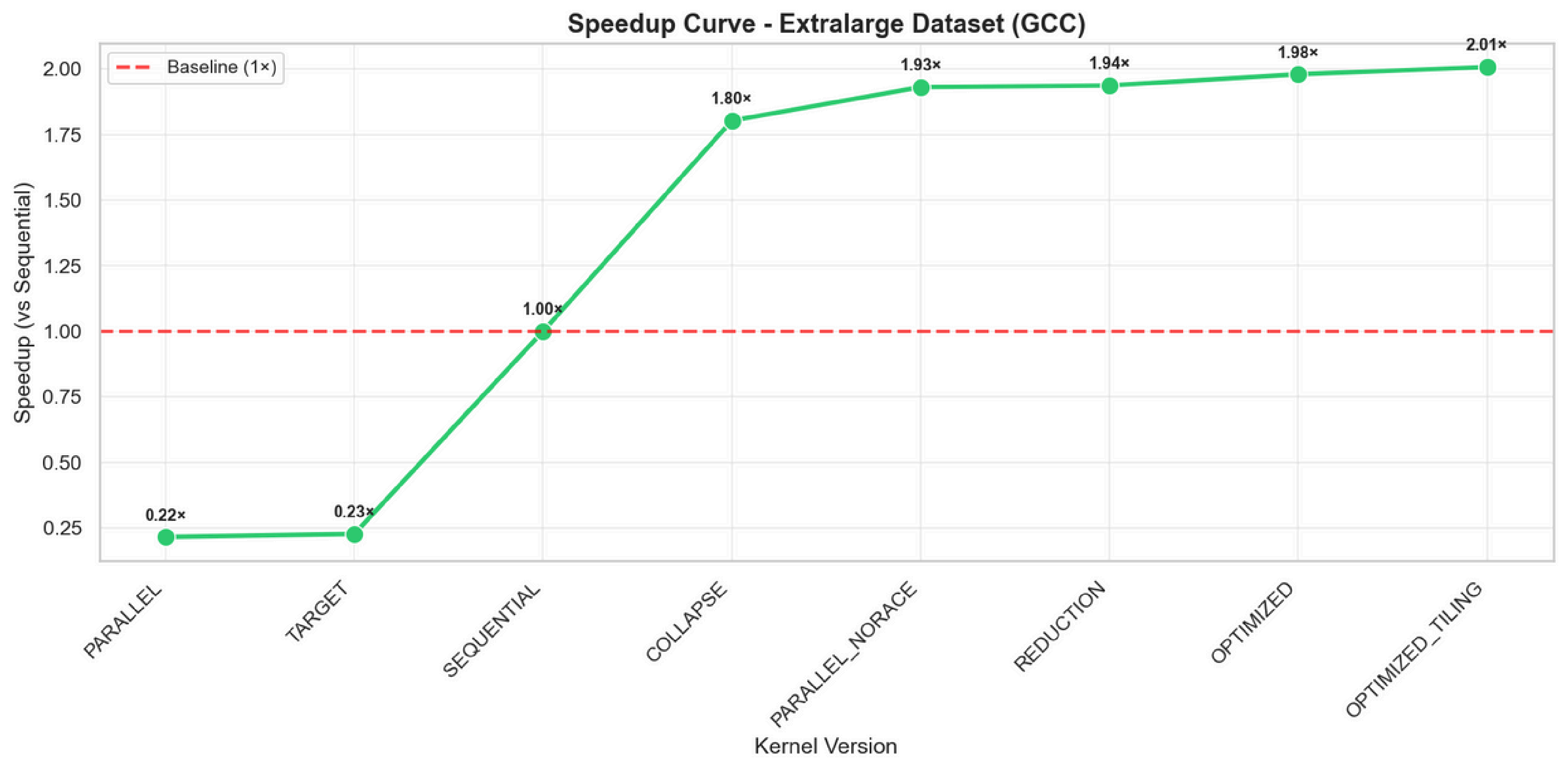
Results - GCC VS CLANG

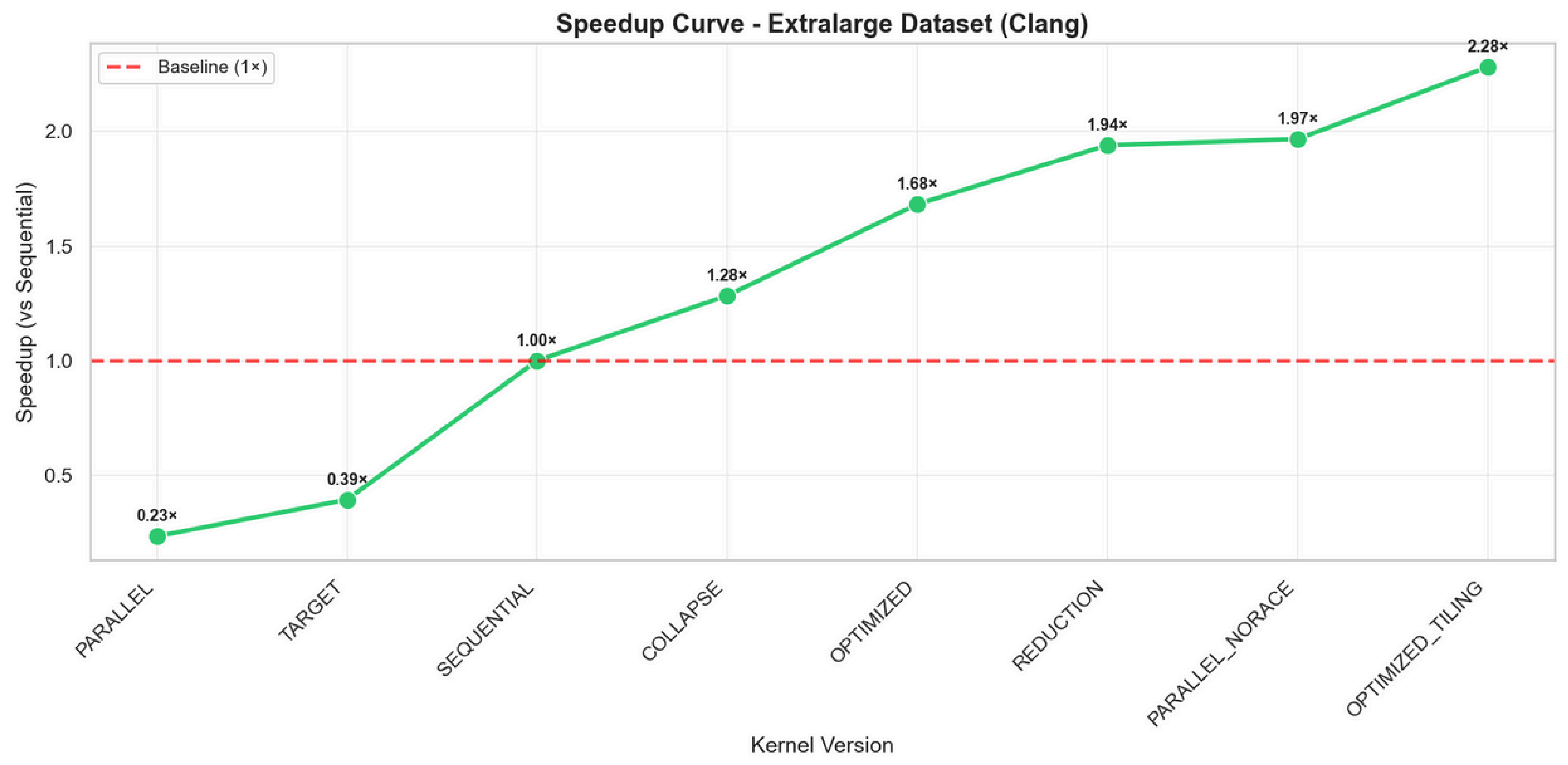




In order to be able to compile with the GPU at full potential we needed to compile with CLANG. So we took the times for the both of the compilers a compared them.
It has been generated an histogram that shows on Y axis the performance using second as metrics, and on X axis the various optimization tested.
Optimized_Tiling version keeps the one that succseds in term of performance.

Results - GCC VS CLANG





It has also been plotted a **speed-up curve** that takes “sequential” version as the start and from this, it shows us the improvement, in term of “times that a specific version is faster” , of each optimization performed.

Thank you!