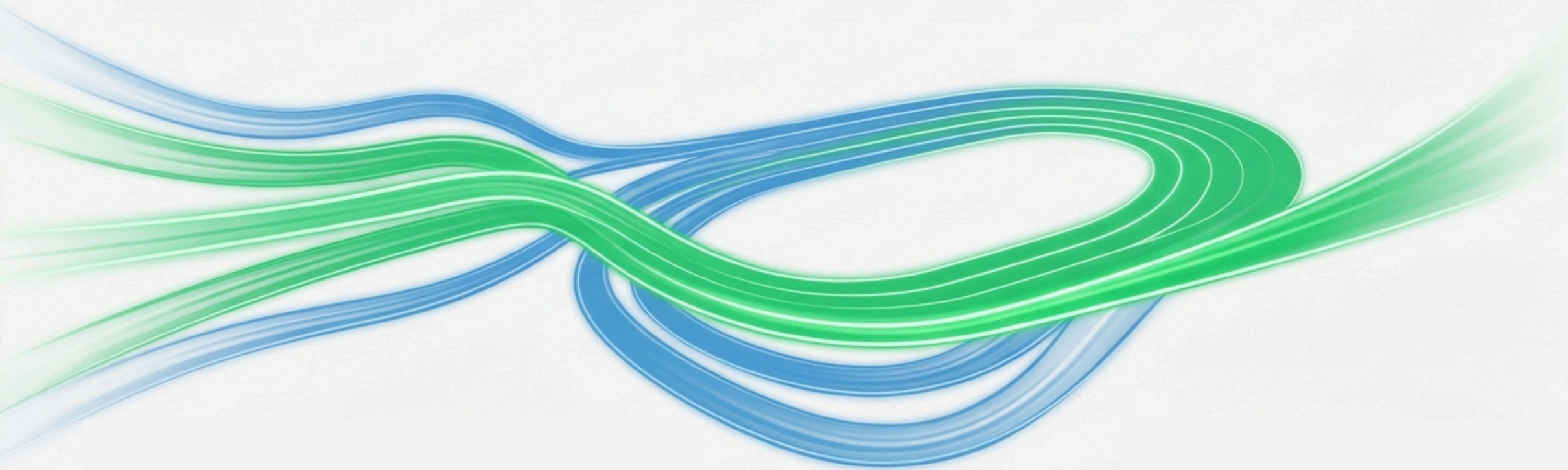
The Race Against Memory Latency

Benchmarking CUDA Optimizations for the ATAX Kernel



Manuel Testoni

Alessandro Chiarabini

Understanding the Opponent: The ATAX Kernel

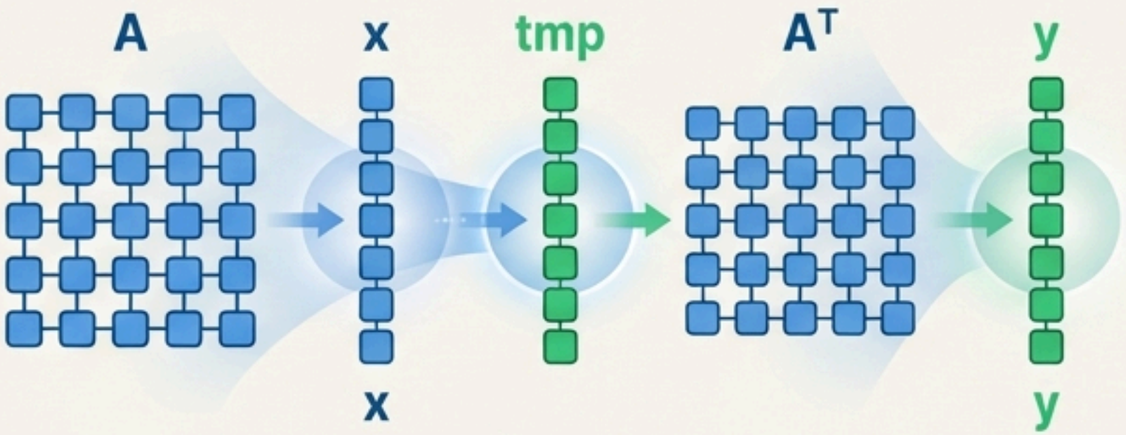
The WHAT

$$y = A^T(Ax)$$

Two-Step Process

tmp = A * x (Matrix-vector product)

y = A^T * tmp (Transposed matrix-vector product)



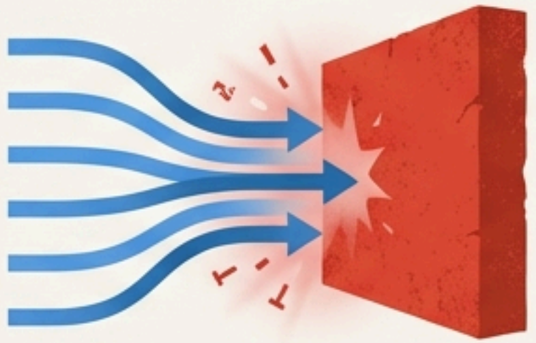
The WHY

The Bottleneck

Low operational intensity means performance is dominated by **global memory bandwidth**, not computation.

Key Challenge

Heavy global memory traffic creates a performance wall.



Conclusion

An ideal benchmark to test memory-access optimization techniques.

```cpp
/* Kernel: compute tmp[i] = sum_j A[i][j] * x[j]  */      You, 20 hours ago • Uncommitted changes
__global__ void kernel_tmp(const DATA_TYPE* A, const DATA_TYPE* x, DATA_TYPE* tmp, int nx, int ny) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < nx) {
    DATA_TYPE sum = 0.0;
    int row_off = i * ny;
    for (int j = 0; j < ny; ++j) {
      sum += A[row_off + j] * x[j];
    }
    tmp[i] = sum;
  }
}

/* Kernel: compute y[j] = sum_i A[i][j] * tmp[i]  (one thread per column j) */
__global__ void kernel_y(const DATA_TYPE* A, const DATA_TYPE* tmp, DATA_TYPE* y, int nx, int ny) {
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (j < ny) {
    DATA_TYPE sum = 0.0;
    for (int i = 0; i < nx; ++i) {
      sum += A[i * ny + j] * tmp[i];
    }
    y[j] = sum;
  }
}
```

```cpp
  /* Compute tmp */
  int block = BLOCK_SIZE;
  int grid_tmp = (nx + block - 1) / block;
  kernel_tmp<<<grid_tmp, block>>>(A_d, x_d, tmp_d, nx, ny);
  CUDA_CHECK(cudaGetLastError());

  /* Compute y */
  int grid_y = (ny + block - 1) / block;                //
  kernel_y<<<grid_y, block>>>(A_d, tmp_d, y_d, nx, ny); //
  CUDA_CHECK(cudaGetLastError());
```

We manged to divide the two main cycles in two different kernels. First one compute tmp array. Second one compute y, the final output we need to mark as completed our program.

This was our first implementation with no optimizations at all. Pretty fast due to GPU parallell computation.

The Contenders, Part 1: Memory Management Plays

Pinned Memory

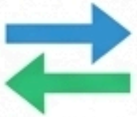
Strategy: Locks host memory to enable faster, asynchronous PCI-e transfers.

Trade-off: Higher allocation cost for improved throughput.

Unified Memory (UVM)

Strategy: Simplifies code with a single memory space for CPU/GPU.

Trade-off: Convenience can come at the cost of page-fault overhead, hurting performance on large datasets.



CUDA Streams

Strategy: Aims to overlap data transfers with kernel execution.

Trade-off: Benefit is limited in memory-bound kernels like ATAX where there's little compute to hide.

__constant__ Memory

Strategy: Uses a small, fast, read-only cache for data shared by all threads (like the vector x).

Trade-off: Only useful for small, read-only, uniformly accessed data.

Pinned Memory Allocation

```
/* Alloca pinned HOST */
CUDA_CHECK(cudaMallocHost((void**)&A_h, sizeA));
CUDA_CHECK(cudaMallocHost((void**)&x_h, sizex));
CUDA_CHECK(cudaMallocHost((void**)&y_h, sizey));
CUDA_CHECK(cudaMallocHost((void**)&tmp_h, sizetmp));

/* Inizializza dati in pinned memory */
for (int i = 0; i < ny; i++)
  x_h[i] = i;
for (int i = 0; i < nx; i++)
  for (int j = 0; j < ny; j++)
    A_h[i * ny + j] = 1.0;
```

Pinned Memory Deallocation

```
/* Free pinned HOST memory */
CUDA_CHECK(cudaFreeHost(A_h));
CUDA_CHECK(cudaFreeHost(x_h));
CUDA_CHECK(cudaFreeHost(y_h));
CUDA_CHECK(cudaFreeHost(tmp_h));
```

UVM Allocation

```
/* Allocate device memory */
CUDA_CHECK(cudaMallocManaged((void**)&A_d, sizeA));
CUDA_CHECK(cudaMallocManaged((void**)&x_d, sizex));
CUDA_CHECK(cudaMallocManaged((void**)&y_d, sizey));
CUDA_CHECK(cudaMallocManaged((void**)&tmp_d, sizetmp));
```

Constant Memory Allocation

```
/* Vettore di dimenesione massima = a constant memory
__constant__ DATA_TYPE x_d_const[8192];
```

Constant Memory Usage

```
/* Kernel: compute tmp[i] = sum_j A[i][j] * x[j] */
__global__ void kernel_tmp(const DATA_TYPE* A, DATA_TYPE* tmp, int nx, int ny) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < nx) {
    DATA_TYPE sum = 0.0;
    int row_off = i * ny;
    for (int j = 0; j < ny; ++j) {
      // Leggiamo da constant memory
      sum += A[row_off + j] * x_d_const[j];
    }
    tmp[i] = sum;
  }
}
```

Constant Memory Copy

```
/* Copiamo x nella costant memory invece che in quella global */
CUDA_CHECK(cudaMemcpyToSymbol(x_d_const, x_h, sizex));
```

Streams definition and creation

```
#define N_STREAMS 4
```

```
You, 18 hours ago • Adding stream versione, readme, exercise flag i…
int TILE = nx / N_STREAMS; // chunk size

/* Creiamo gli stream CUDA */
cudaStream_t streams[N_STREAMS];
for(int i = 0; i < N_STREAMS; i++) {
    CUDA_CHECK(cudaStreamCreate(&streams[i]));
}
```

When we use streams we have to ensure some things:

- **Overhead**: we have created 4 streams in order to not pay extra overhead
- **Pinned Memory**: Due to the fact that we are performing an asynchronously copy, we have to ensure we use pinned memory.
- **Synchronization**: While we perform simultaneous copy and execution, we must wait to execute the other kernel.
- **Offsets**: Our matrices are row major, this means that we had to calculate the right offset to access the right side of the matrices at every iteration.

Streams usage

```
/*ogni stream processa un chunk di righe di A in parallelo */
for (int s = 0; s < N_STREAMS; s++) {
    // Calcoliamo indici per questo chunk
    int row_start = s * TILE;
    int row_end = (s + 1) * TILE;
    int chunk_rows = row_end - row_start;
    size_t offset_A = row_start * ny;  // Inizio del chunk nella matrice A
    size_t chunk_size_A = chunk_rows * ny * sizeof(DATA_TYPE);  // Dimensione in byte

    /* ASYNC: Copia chunk di A da host a device */
    CUDA_CHECK(cudaMemcpyAsync(
        &A_d[offset_A],      // Destinazione GPU: posizione del chunk
        &A_h[offset_A],      // Sorgente HOST: posizione del chunk
        chunk_size_A,        // Dimensione del chunk in byte
        cudaMemcpyHostToDevice,
        streams[s]           // Stream dedicato a questo chunk
    ));
        You, 18 hours ago • Adding stream versione, readme, exercise flag i…
    /* chiamata kernel su un chunk: calcola tmp per queste righe  */
    int grid_tmp = (chunk_rows + BLOCK_SIZE - 1) / BLOCK_SIZE;
    kernel_tmp<<<grid_tmp, BLOCK_SIZE, 0, streams[s]>>>(
        &A_d[offset_A],      // Puntatore al chunk di A su GPU
        x_d,                 // x completo (serve a tutti i chunk)
        &tmp_d[row_start],   // Puntatore al chunk di tmp su GPU
        chunk_rows,          // Numero di righe in questo chunk
        ny                   // Numero di colonne (sempre ny)
    );
    CUDA_CHECK(cudaGetLastError());
}
```

Streams synchronization and distruction

```
// CI ASSICURIAMO CHE TUTTI GLI STREAMS ABBIANO FINITO TMP
for (int s = 0; s < N_STREAMS; s++) {
    CUDA_CHECK(cudaStreamSynchronize(streams[s]));
}
```

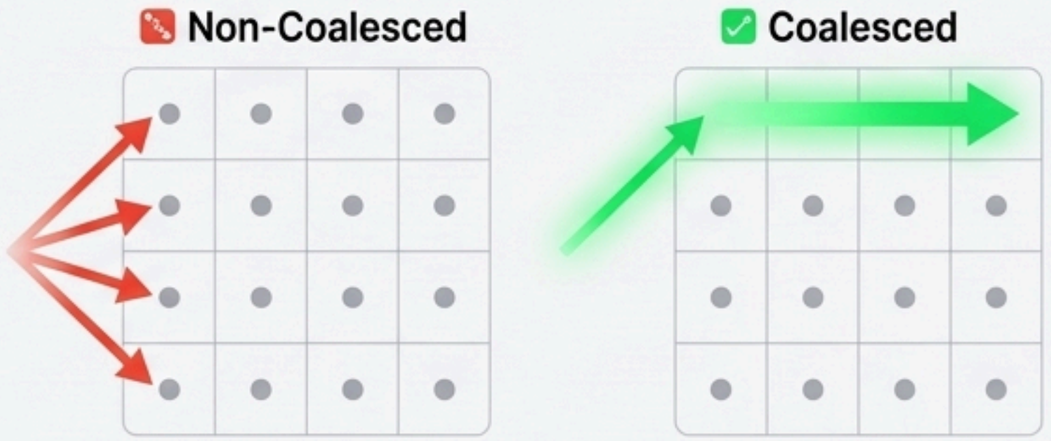
```
// distruggiamo tutti gli streams
for(int i = 0; i < N_STREAMS; i++) {
    CUDA_CHECK(cudaStreamDestroy(streams[i]));
}
```

The Contenders, Part 2: The Architectural Overhauls

Matrix Transposition

The Problem: Reading matrix `A` by columns in the second step ($A^T * tmp$) leads to non-coalesced memory access.

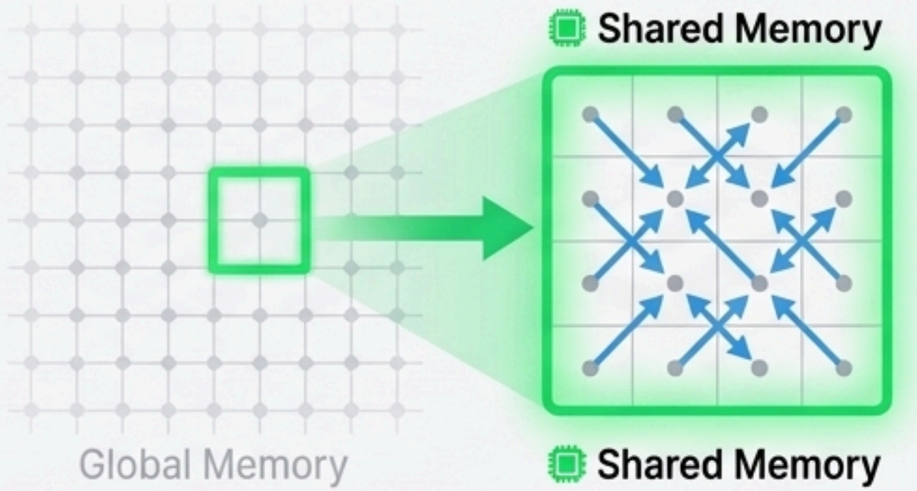
The Fix: Pre-transposing the matrix ensures all global memory reads are coalesced, dramatically reducing memory divergence.



Shared-Memory Tiling

The Problem: Threads repeatedly fetch data from slow global memory.

The Fix: Load a small 'tile' of the matrix into fast on-chip shared memory. Threads in a block can then reuse this data, minimizing global traffic.



The Arena: A Fair Fight Across All Scales

The Gauntlet (Datasets)

MINI

SMALL

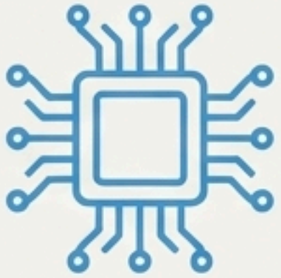
STANDARD

LARGE

EXTRALARGE

The Rules (Methodology)

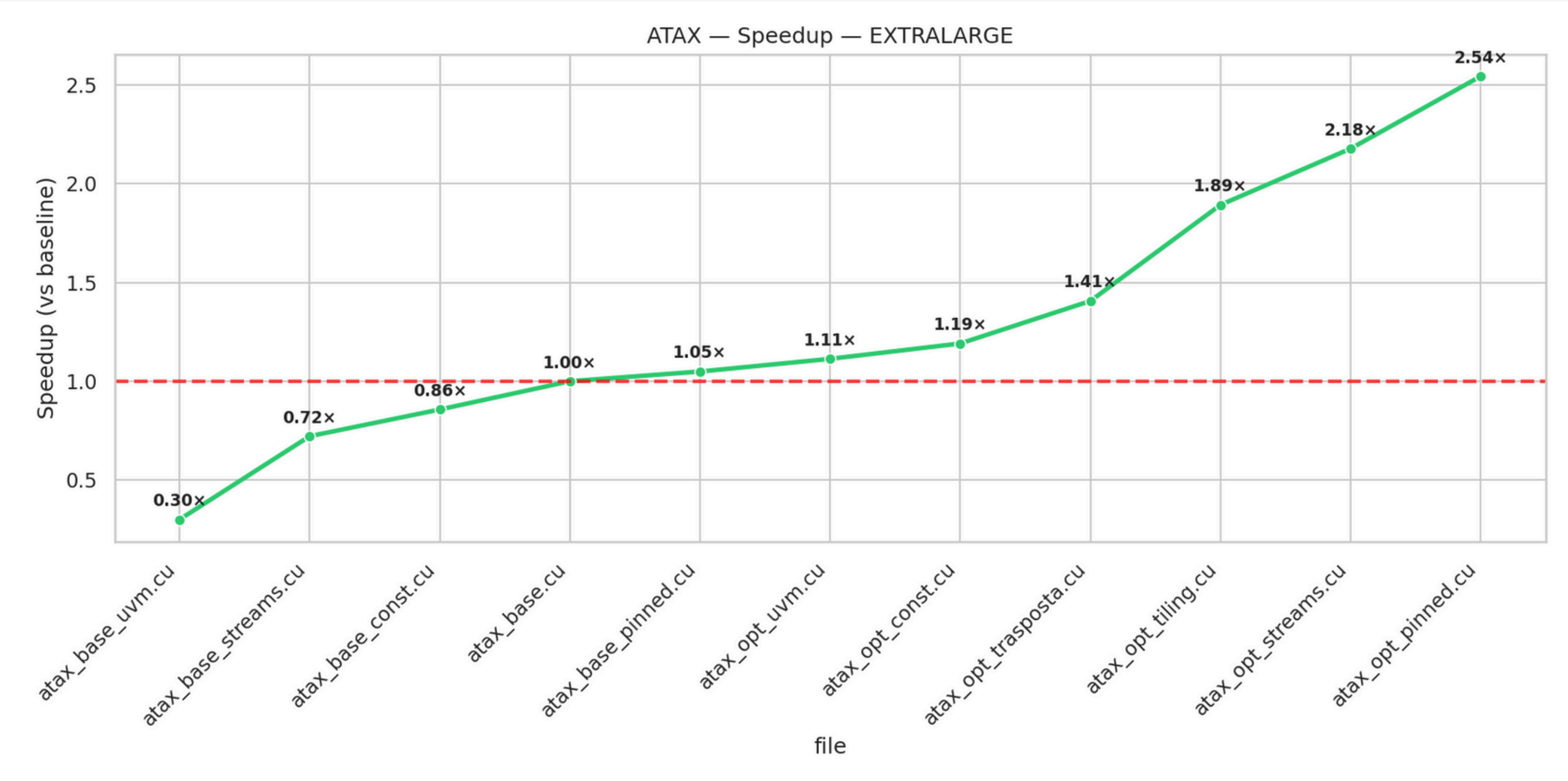
- Automation script runs each kernel 5 times per dataset.
- Average GPU execution time is measured.
- A warm-up run stabilizes GPU frequency.



The Environment (Hardware)

- All tests run on the same GPU with identical CUDA versions and build settings to ensure a level playing field.

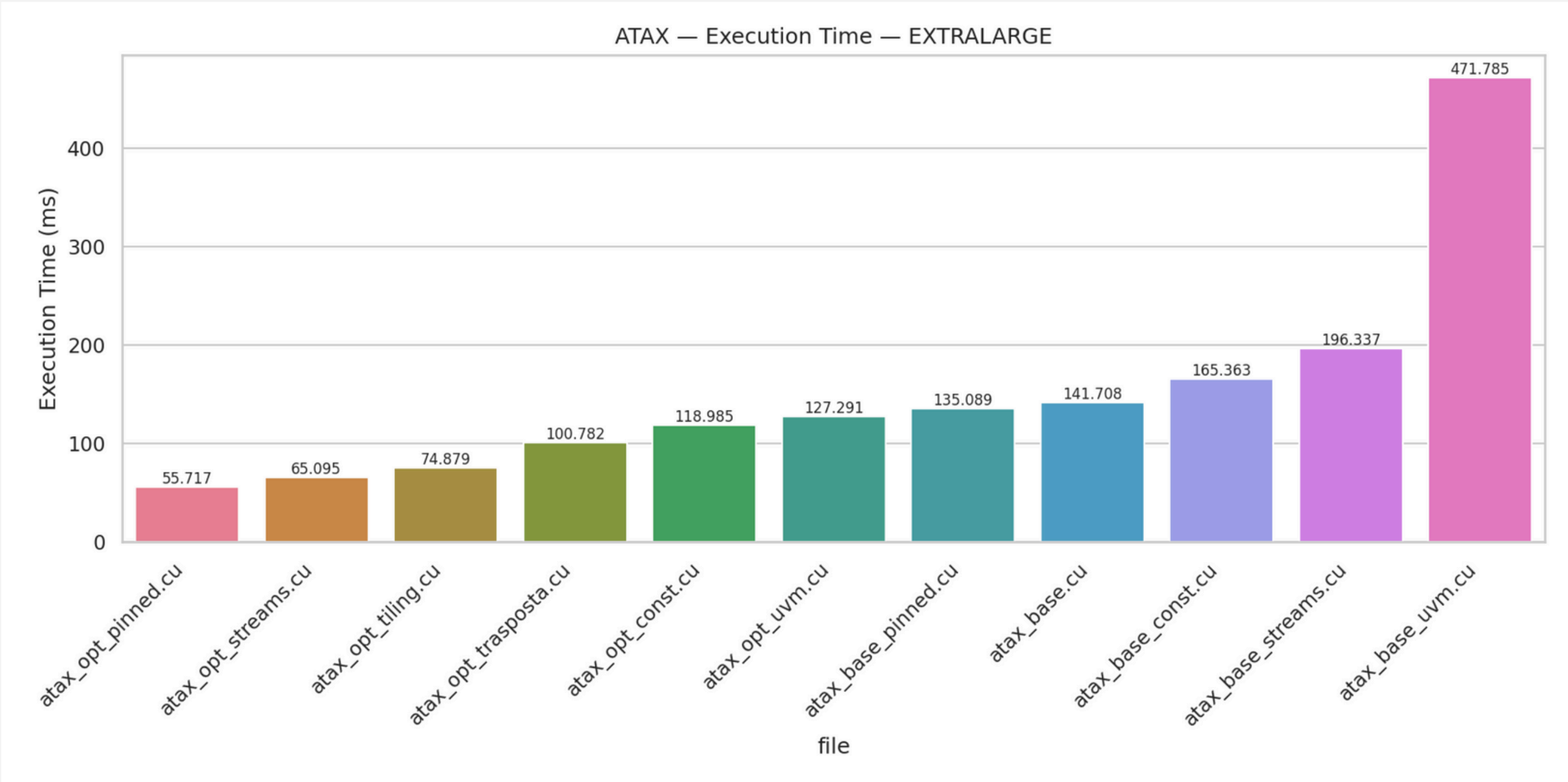
The Finale Showdown: Performance of the EXTRALARGE dataset



Memory management didn’t pay out when the memory access isn’t optimized

Pinned memory achieves a 2.54x speedup

The Finale Showdown: Performance of the EXTRALARGE dataset



Lower Is Better: pinned finishes in 55.17 ms

The champions

Pinned Memory (2.54x)

A strong supporting technique. Faster H2D/D2H transfers provide a significant boost, especially when memory access patterns are already good

CUDA Streams (2.18x)

A remarkably strong technique. By pipelining H2D/D2H copies with computation, CUDA streams can double throughput, especially for large datasets where overlap becomes fully effective.

Tiling - Global (1.89x)

Maximizing data reuse withing the SM minimized slow global memory traffic, leading to an higher effective bandwidth

The Underperformers

UVM (no OPT) (0.30x)

The convenience tax is too high, especially when the pattern access isn't optimized. For large dataset, page migration and faults caused a massive slowdown, making it over 3x slower than the baseline

CUDA Strams (no OPT) (0.72x)

Streams without optimized access patterns add more overhead than benefit. The kernel remains memory-bound, concurrency can't hide the bottleneck, and stream management costs make it ~0.72× slower than the baseline.

The ATAX Playbook: Lessons for Your Memory-Bound Kernels

1.

Prioritize Data Reuse Above All

Shared-memory tiling is not optional; it is the most powerful weapon against memory latency. If you can keep data on-chip, do it.

2.

Ensure Coalesced Access

Before complex optimizations, ensure your threads access global memory in contiguous blocks. Fixing this can provide a huge, foundational performance gain.

3.

Use the Right Tool for the Job

- **Pinned Memory**: A near-universal, easy win for improving data transfer speed.
- **UVM**: A great tool for rapid prototyping, but beware the performance penalty on large-scale problems.
- **Streams**: Only pursue if you have significant, independent compute that can truly overlap with memory transfers.

Thank You!