CS 402: HIGH PERFORMANCE COMPUTING

GPU Compute Architecture

A Comparative Analysis of Serial vs. Parallel Execution

Department of Computer Science
Advanced Systems Laboratory

Architectural Paradigm: CPU vs. GPU

Central Processing Unit (CPU)

Optimization Goal: Latency Minimization

- Designed for sequential logic and complex branching.
- Large caches to reduce memory access latency for single threads.
- Sophisticated control units (branch prediction, out-of-order execution).

Graphics Processing Unit (GPU)

Optimization Goal: Throughput Maximization

- Designed for massive data parallelism.
- Thousands of simpler, energy-efficient cores.
- Hides memory latency by interleaving thousands of active threads.

The CUDA Execution Hierarchy

To manage massive parallelism, CUDA abstracts execution into a three-level spatial hierarchy:

1. Grid

The entire problem domain. A Grid executes a single Kernel.

2. Block

A subdivision of the Grid. Threads within a Block can share memory.

3. Thread

The fundamental unit of execution. Each thread processes one data element.

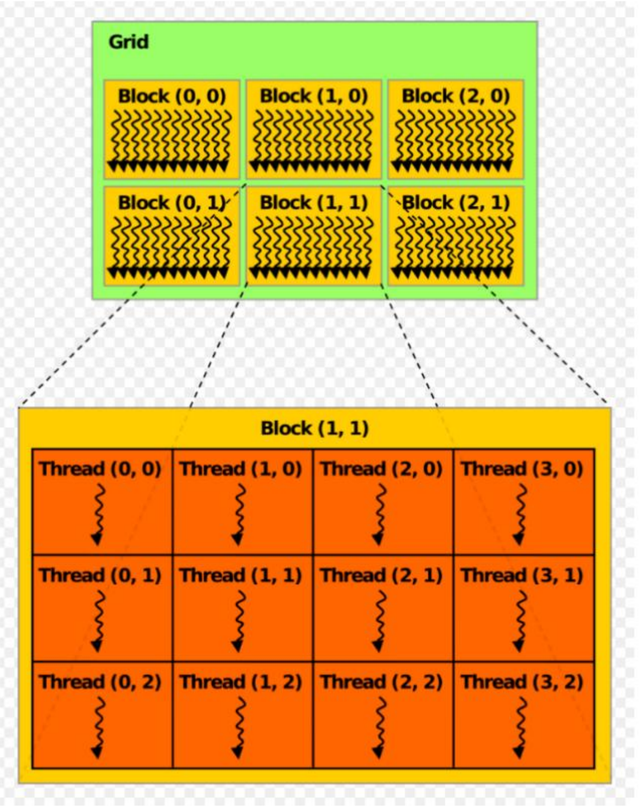


Figure 1: Spatial decomposition of a kernel execution.

Thread Addressing & Global ID

Coordinate Mapping

Threads require a unique identifier to determine which data element to process. CUDA provides built-in variables:

- `gridDim`: Total dimensions of the grid.
- `blockIdx`: Coordinate of the block within the grid.
- `blockDim`: Dimensions of the block.
- `threadIdx`: Coordinate of the thread within the block.

Global Index Formula

For a 1D layout, the unique global index i is calculated as:

$$i = (blockIdx.x \times blockDim.x) + threadIdx.x$$

This formula maps the hierarchical thread coordinate to a linear memory address.

Hardware Abstraction: Streaming Multiprocessors

Mapping Software to Silicon

The software hierarchy corresponds directly to physical hardware units on the GPU die.

- **Grid** maps to the **Entire GPU**.
- **Block** maps to a **Streaming Multiprocessor (SM)**.
- **Thread** maps to a **CUDA Core (SP)**.

Note: Once a Block is assigned to an SM, it resides there until completion. The SM manages the scheduling and execution of threads.

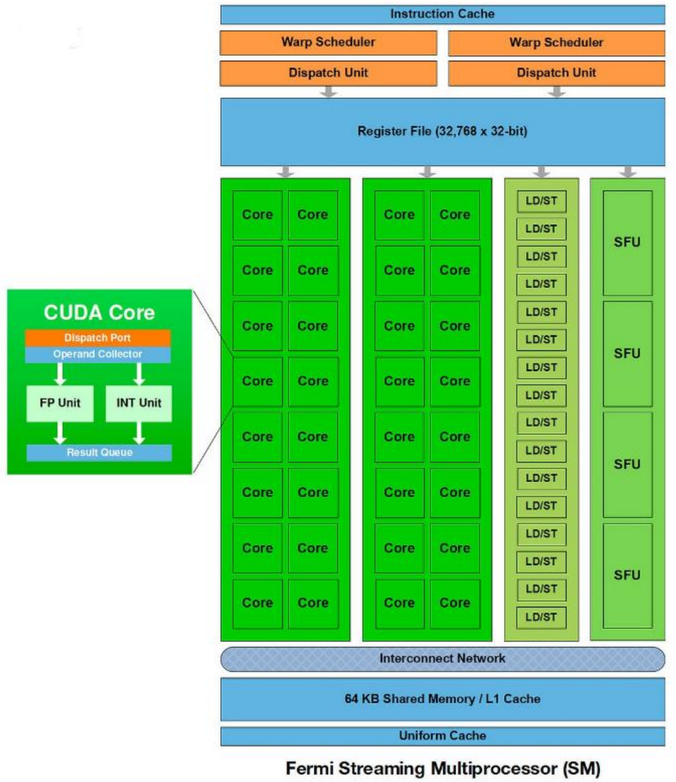
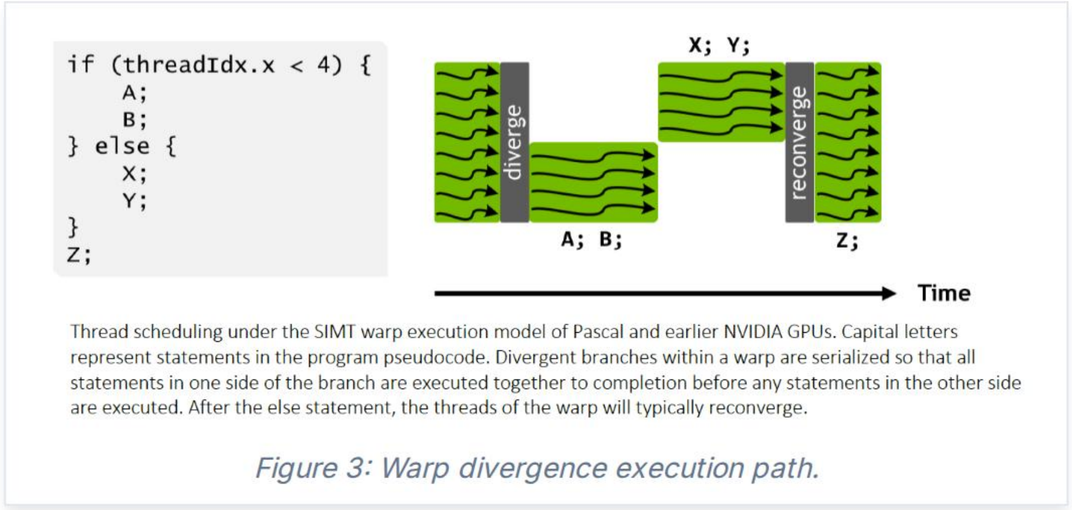


Figure 2: Architecture of a Streaming Multiprocessor.

SIMT Execution & The Warp



Thread scheduling under the SIMT warp execution model of Pascal and earlier NVIDIA GPUs. Capital letters represent statements in the program pseudocode. Divergent branches within a warp are serialized so that all statements in one side of the branch are executed together to completion before any statements in the other side are executed. After the else statement, the threads of the warp will typically reconverge.

Figure 3: Warp divergence execution path.

The Warp (32 Threads)

Threads are not scheduled individually. They are grouped into bundles of 32 called **Warps**.

Lockstep Execution

In SIMT (Single Instruction, Multiple Threads), all 32 threads execute the *same instruction* simultaneously.

Constraint: Divergence
If threads in a warp take different execution paths (e.g., if/else), execution is serialized, significantly reducing performance.

Memory Hierarchy Performance

Efficient GPU programming requires managing data movement between different memory tiers.

- **Global Memory (DRAM):**

  High Capacity, High Latency (Slow). Accessible by all.

- **Shared Memory (L1):**

  Low Capacity, Ultra-Low Latency. User-managed cache shared by a Block.

- **Registers:**

  Private to a thread. Zero latency.

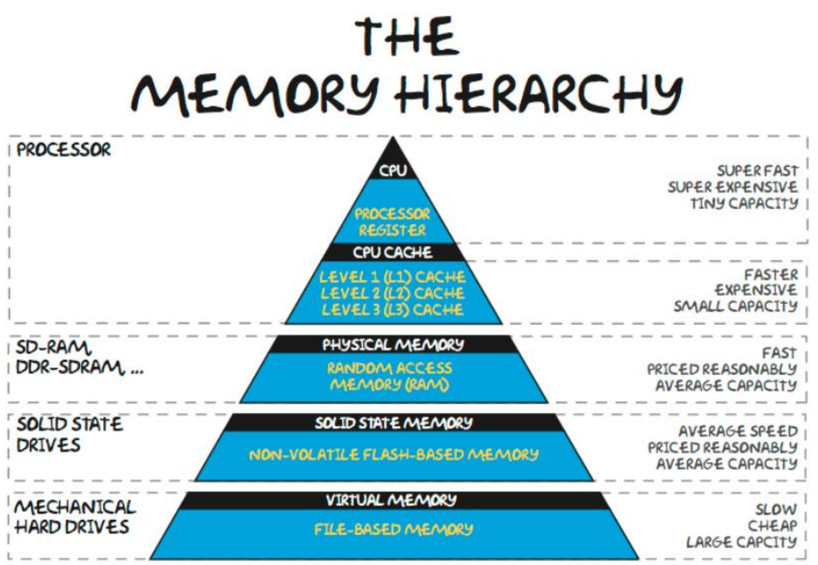


Figure 4: Memory Bandwidth vs Latency.

Implementation: Vector Addition Kernel

Kernel Logic

The following C++ CUDA code demonstrates the translation of the loop-based CPU approach to a parallel thread-based approach.

Note the absence of a loop; the grid provides the iteration implicitly.

```
__global__ void vectorAdd(float* A, float* B, float* C, int
N)
{
    // 1. Calculate Global Unique ID
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. Boundary Check (Guard)
    if (i < N) {
        // 3. Perform Computation
        C[i] = A[i] + B[i];
    }
}
```

Implementation: Vector Multiplication

Pattern Reuse

Because Vector Addition and Multiplication are both *element-wise* operations, they share the same memory access pattern.

This "Map" pattern is a fundamental building block of parallel algorithms.

```
__global__ void vectorMult(float* A, float* B, float* C, int
N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) {
        // Only the operator changes
        C[i] = A[i] * B[i];
    }
}
```

Data Structure: Linearizing 2D Matrices

Row-Major Layout

Computer memory is fundamentally linear (1D). To store a 2D matrix, rows are placed sequentially in memory.

Mapping Formula

$$index = row \times width + col$$

Both CPU (host) and GPU (device) functions must utilize this formula to map 2D coordinates `(x,y)` to a specific memory address.

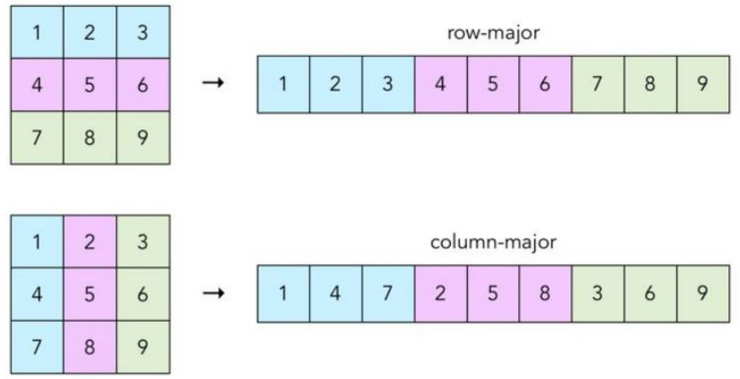


Figure 5: Row-major memory layout.

Matrix Addition: Sequential (CPU)

Algorithmic Complexity

The CPU implementation relies on nested iteration. The outer loop traverses rows, while the inner loop traverses columns.

Complexity: $O(N^2)$ for an $N \times N$ matrix.

```
void matAddCPU(float* A, float* B, float* C, int w, int h) {
    for (int r = 0; r < h; r++) {
        for (int c = 0; c < w; c++) {
            int idx = r * w + c;
            C[idx] = A[idx] + B[idx];
        }
    }
}
```

Matrix Addition: Parallel (GPU)

2D Grid Configuration

CUDA supports multi-dimensional blocks. By defining `dim3 block(16, 16)`, we can naturally map the thread grid to the matrix geometry.

The code calculates `col` (x) and `row` (y) independently before linearizing.

```
__global__ void matAddGPU(float* A, float* B, float* C, int
w, int h) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col < w && row < h) {
        int idx = row * w + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

Matrix Multiplication: CPU Bottleneck

Triple Loop Overhead

Matrix multiplication involves a dot product for every output element.

Complexity: $O(N^3)$.

This cubic complexity makes large matrix multiplication computationally prohibitive on serial processors.

```
void matMulCPU(float* A, float* B, float* C, int N) {
  for (int y = 0; y < N; y++) {
    for (int x = 0; x < N; x++) {
      float sum = 0;
      for (int k = 0; k < N; k++) {
         sum += A[y*N + k] * B[k*N + x];
      }
      C[y*N + x] = sum;
    }
  }
}
```

Matrix Multiplication: Naive GPU Approach

One Thread, One Pixel

We map each thread to one output element `C[row][col]`. This effectively parallelizes the two outer loops of the CPU implementation.

The Bottleneck:
Each thread must read an entire row of A and an entire column of B from slower global memory. This pattern has a low compute-to-memory-access ratio.

```
__global__ void matMulNaive(float* A, float* B, float* C,
int N) {
  // Calculate row and col indices
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  // Guard to prevent out-of-bounds
  if(row < N && col < N) {
    float sum = 0;
    // Standard Dot Product
    for(int k=0; k < N; k++) {
      sum += A[row*N + k] * B[k*N + col];
    }
    C[row*N + col] = sum;
  }
}
```

Optimization I: The Memory Bandwidth Crisis

Global vs. Shared Memory

The Naive Kernel is **Memory Bound**. It spends most cycles waiting for data from DRAM.

- **Global Memory:** Large but high latency (~hundreds of cycles).
- **Shared Memory:** Small on-chip cache. Extremely low latency (~few cycles).

To optimize, we must minimize Global Memory accesses by caching data in Shared Memory.

Memory Type	Scope	Latency	Size
Global (DRAM)	All Threads	High	~16-80 GB
Shared (SRAM)	**Block**	**Very Low**	**~48-100 KB**
Registers	Thread	Near Zero	~256 KB/SM

Optimization II: The Tiling Strategy

Cooperative Loading

Instead of threads reading individual values, threads in a Block cooperate to load a sub-matrix ("Tile") into Shared Memory.

The 4-Step Dance:

1. **Load:** All threads load one element from Global to Shared memory.
2. **Sync:** `__syncthreads()` ensures the Tile is fully loaded.
3. **Compute:** Perform dot product on the high-speed Shared Memory data.
4. **Sync:** `__syncthreads()` prevents overwriting data before others are done.

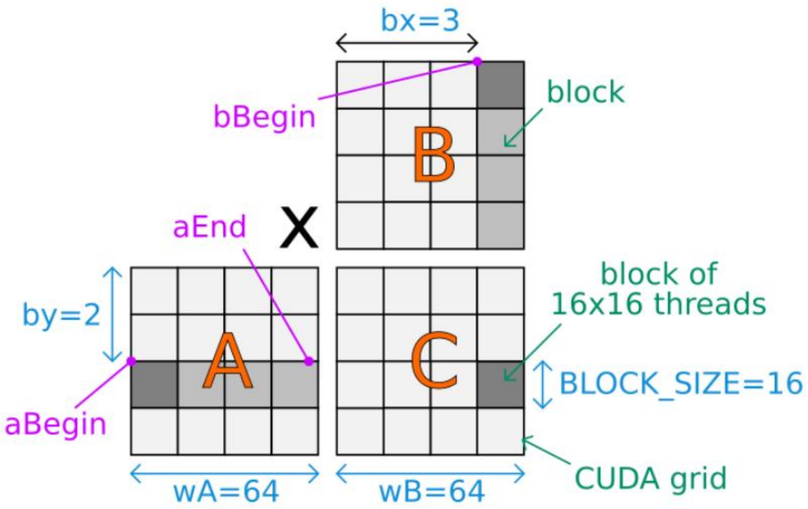


Figure 6: Tiled matrix multiplication conceptual view.

Optimization III: Tiled Implementation

```
__global__ void matMulTiled(float* A, float* B, float* C, int N) {
  // 1. Allocate Shared Memory Tiles
  __shared__ float tileA[16][16];
  __shared__ float tileB[16][16];

  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  float val = 0;

  // 2. Iterate over Tiles (phases)
  for (int ph = 0; ph < N/16; ++ph) {
     // 3. Cooperative Load: Global → Shared
     // Manually broken lines to prevent ugly wrapping
     tileA[ty][tx] = A[N * (by * 16 + ty)
                       + (ph * 16 + tx)];

     tileB[ty][tx] = B[N * (ph * 16 + ty)
                       + (bx * 16 + tx)];
     __syncthreads();

     // 5. Compute on Cached Data
     for (int k = 0; k < 16; k++)
       val += tileA[ty][k] * tileB[k][tx];
     __syncthreads();
  }

  // Store result
  C[N * (by * 16 + ty)
    + (bx * 16 + tx)] = val;
}
```

Key Mechanisms

- `__shared__`: Allocates memory that is physically on the SM, visible to all threads in the block.
- `__syncthreads()`: A hardware barrier. No thread proceeds past this point until *all* threads in the block have reached it. Essential for avoiding race conditions.
- `16`: The Tile Width. This determines the size of the shared memory buffer.