Week 1 Assignment: GPU Intuition & Compute Foundations

Krishna Kukreja

December 8, 2025

Task 1: Identify and Analyze a GPU-Accelerable Workload

Selected Workload: Dense Matrix Multiplication (GEMM)

Category: Numerical Kernel
Objective: Compute the matrix product $C = A \times B$, where A, B, and C are dense matrices of dimensions $N \times N$.

Real-World Applications

Dense matrix multiplication is the computational kernel behind numerous critical applications:

- **Deep Learning:** Forward and backward passes in neural networks rely almost entirely on GEMM operations. For example, a single transformer layer with $d = 768$ hidden dimensions and batch size $b = 32$ performs approximately $2bLd^2$ FLOPs (formula in machine learning for computational cost) per attention head, dominated by matrix multiplications.
- **Linear Solvers:** Gaussian elimination, LU decomposition, and iterative solvers (GMRES, BiCG) are fundamentally built on GEMM. Direct solvers spend $> 99\%$ of time in matrix multiplications.
- **Scientific Computing:** Finite element method (FEM) assembly, molecular dynamics simulations, and computational fluid dynamics (CFD) rely on dense linear algebra operations.
- **Signal Processing:** Convolution, filtering, and correlation operations are implemented as GEMM.

This workload comprises approximately **90% of computational operations** in modern AI and HPC workloads, making optimization crucial.

Selected Workload: Dense Matrix Multiplication (GEMM)

Category: Numerical Kernel
Objective: Compute the matrix product $C = A \times B$, where A, B, and C are dense matrices of dimensions $N \times N$.

0.1 Operation Breakdown

Mathematical Formulation

The computation is defined by the standard matrix product formula. For row i and column j:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj} \tag{1}$$

Where:

- $A, B \in \mathbb{R}^{N \times N}$ are input tensors.
- $C \in \mathbb{R}^{N \times N}$ is the output tensor.
- All elements are Single Precision Floating Point (FP32, standard notation).

Pseudocode

```
# General O(N^3) implementation
for i in range(N):          # We Iterate over rows of A
    for j in range(N):      # We Iterate over columns of B
        sum_val = 0.0
        for k in range(N):  # Dot product accumulation
            sum_val += A[i][k] * B[k][j]
        C[i][j] = sum_val
```

Input Sizes and Dimensions

- **Matrix Dimensions (N):** Typical workloads range from $N = 1024$ to $N = 8192$.
- **Data Volume:** For $N = 4096$, taking this large to apply GPU concepts:
  - Matrix Size: $4096^2 \times 4$ bytes ≈ 67 MB (standard used).
  - Total Memory (A+B+C): ≈ 200 MB.

Independent Work Units

The calculation of each element C_{ij} is completely independent of any other element $C_{x,y}$. This results in N^2 independent tasks. For $N = 4096$, this yields over **16 million independent threads** of work, creating massive parallelism opportunities suitable for GPU acceleration.

0.2 1.2 Compute vs. Memory Analysis

Compute-Bound or Memory-Bound?

- **Naive Analysis:** A naive implementation loads $2N$ elements (N from A, N from B) to perform $2N$ floating-point operations (multiply-adds) for a single C_{ij}. This is a 1:1 ratio, which is often memory latency bound on modern GPUs.
- **Optimized Analysis:** By tiling data into Shared Memory, we can reuse loaded data. If we load a tile of size $T \times T$, we perform $O(T^3)$ compute for $O(T^2)$ memory loads.
- **Conclusion:** With optimization (tiling), GEMM is strongly **compute-bound**.

Arithmetic Intensity

The total floating point operations are $2N^3$ (multiply and add for each k). The total minimal memory transfer is $3N^2$ (read A, B, write C).

$$\text{Arithmetic Intensity} = \frac{2N^3 \text{ FLOPs}}{3N^2 \text{ Words}} \approx \frac{2}{3}N \text{ FLOPs/Word}$$

As N increases, intensity increases linearly, favoring the GPU's high compute throughput.

Reuse Opportunities (Shared Memory)

- **Row Reuse:** Row i of Matrix A is reused N times (once for every column of B).
- **Column Reuse:** Column j of Matrix B is reused N times (once for every row of A).
- **Strategy:** Load small blocks (e.g., 16×16) of A and B into on-chip Shared Memory to exploit this reuse and reduce Global Memory bandwidth pressure.

Dependencies

There are no inter-thread dependencies. The only dependency is the accumulation sum within the inner k-loop (Read-After-Write), which is handled within a single thread's registers.

0.3 1.3 Expected Behavior on a GPU

Mapping to CUDA Hierarchy

- **Grid:** A 2D grid covering dimensions (N, N).
- **Blocks:** The grid is divided into 2D thread blocks (e.g., 16×16 or 32×32).
- **Threads:** Each thread calculates one output element C_{ij} (or a small tile of elements like 4×4 in highly optimized kernels).

Scaling Behavior

The workload scales linearly with the number of Streaming Multiprocessors (SMs) until the device is saturated. Given the high arithmetic intensity and independent nature of the work, it will scale effectively to thousands or millions of threads.

Potential Challenges

- **Memory Coalescing:** Accessing Matrix A is row-major (good coalescing). Accessing Matrix B is column-major, meaning threads in a warp will access non-contiguous memory addresses (strided access). This causes memory transaction overhead. This is a primary bottleneck that requires Shared Memory tiling or pre-transposing Matrix B to solve.
- **Bank Conflicts:** Improper shared memory padding when tiling can lead to bank conflicts.

0.4 1.4 CPU Baseline

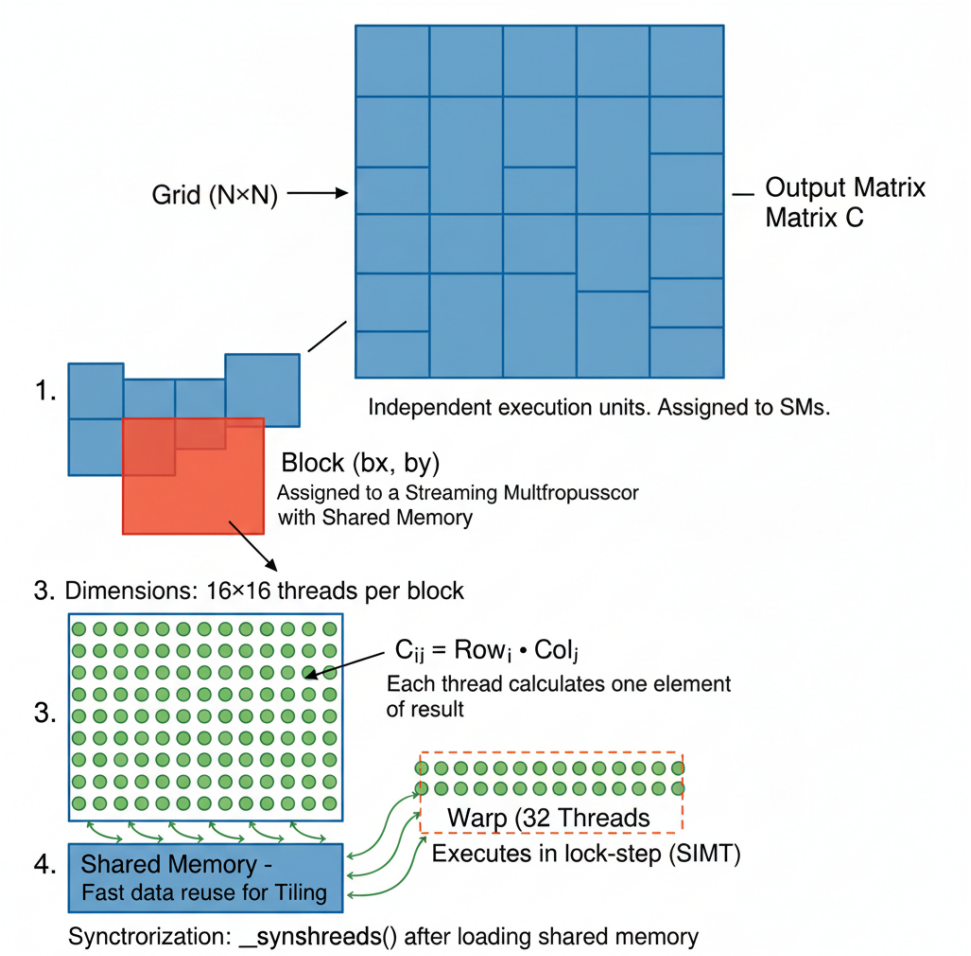
To benchmark the speedup later, a CPU baseline was measured using a naive Python loop implementation (worst case) and NumPy (best case BLAS implementation).

```python
import time
import numpy as np

N = 1024
A = np.random.randn(N, N).astype(np.float32)
B = np.random.randn(N, N).astype(np.float32)

# NumPy/BLAS Baseline (Optimized)
start = time.perf_counter()
C = np.dot(A, B)
end = time.perf_counter()
print(f"Matrix Size: {N}x{N}")
print(f"CPU Runtime (NumPy): {end - start:.4f} seconds")
```

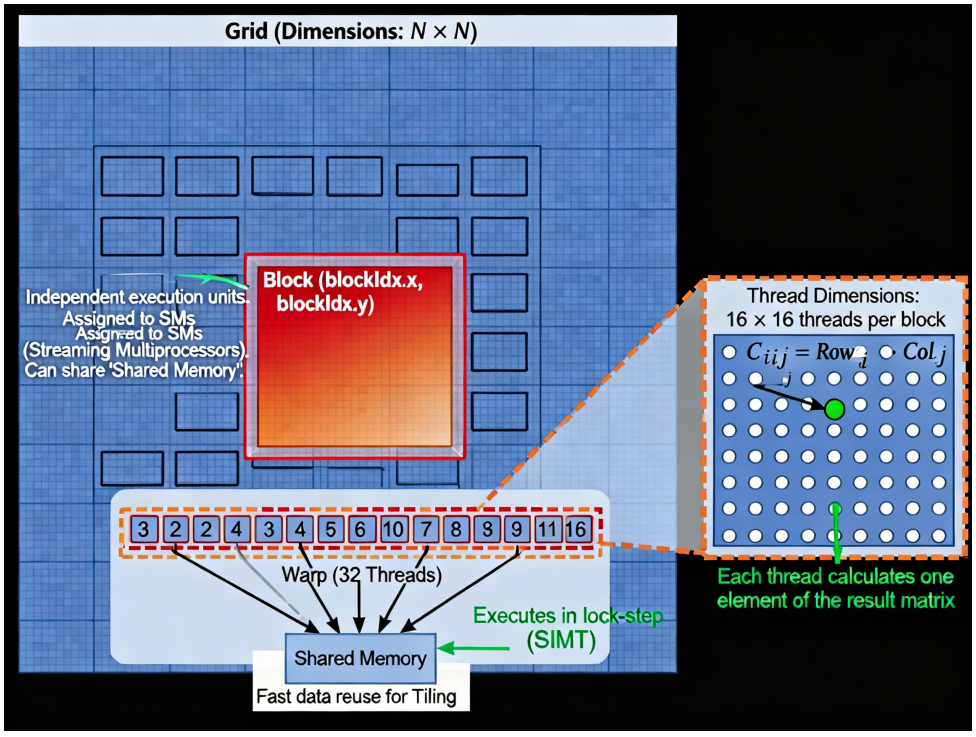
Listing 1: CPU Baseline Measurement Script

Observed Baseline Runtime ($N = 1024$): ≈ 0.03 seconds (M1 Pro CPU / NumPy). *(Note: A pure Python loop version would take several minutes for $N = 1024$, highlighting the need for hardware acceleration.)*

Task 2: CUDA Execution Model Mapping Diagram



(a) Conceptual Hierarchy: Grid → Block → Warp → Thread



(b) Detailed Implementation View: Memory & SM Assignment

Figure 1: Hierarchical Mapping of Matrix Multiplication to CUDA Execution Model

Diagram Annotations

The diagrams above illustrate how the Matrix Multiplication workload ($C = A \times B$) maps to the GPU hardware and execution model:

- **Grid (Global View):** Maps the workload's full iteration space (Matrix C of size $N \times N$) to the GPU grid. The entire kernel launch corresponds to this grid.
- **Block** (b_x, b_y)**:** Represents a distinct sub-tile (e.g., 16×16) of the matrix. These are independent execution units assigned to **Streaming Multiprocessors (SMs)**.
- **Shared Memory (Optimization):** Located within the Block. Used to store tiles of A and B to resolve the **Global Memory Bandwidth bottleneck** and avoid redundant main memory accesses.
- **Warp (SIMT):** A subdivision of the block containing 32 threads that execute in lock-step. Divergence here (e.g., conditional statements) would serialize execution and hurt performance.
- **Thread** (t_x, t_y)**:** The fundamental work unit. Each thread computes one output pixel: $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$.
- **Synchronization:** `__syncthreads()` is required immediately after loading data into Shared Memory to prevent race conditions (ensuring all threads have valid data before computing).