Task 1: Identify and Analyze a GPU-Accelerable Workload

Selected Workload: General Matrix Multiplication (GEMM)

Workload Selection

General Matrix Multiplication (GEMM) is selected as the target workload due to its high computational intensity and widespread use in scientific computing, deep learning, and numerical linear algebra. GEMM serves as a performance benchmark for modern CPUs and GPUs and is highly optimized in libraries such as BLAS and cuBLAS.

The operation is defined as:

$$C = A \times B$$

where:

$$A \in \mathbb{R}^{M \times K}, \quad B \in \mathbb{R}^{K \times N}, \quad C \in \mathbb{R}^{M \times N}$$

1.1 Operation Breakdown

Each output element is computed as a dot product between a row of matrix A and a column of matrix B:

$$C_{i,j} = \sum_{k=0}^{K-1} A_{i,k} \cdot B_{k,j}$$

Pseudocode

```
for i = 0 to M-1:
    for j = 0 to N-1:
        sum = 0
        for k = 0 to K-1:
            sum += A[i][k] * B[k][j]
        C[i][j] = sum
```

Input Sizes and Loop Bounds

Matrix A has dimensions $M \times K$, matrix B has dimensions $K \times N$, and the output matrix C has dimensions $M \times N$. The computation is expressed using three nested loops. The outer loops iterate over $i \in [0, M)$ and $j \in [0, N)$, while the inner loop iterates over $k \in [0, K)$.

Each inner-loop iteration performs one multiplication and one addition. Therefore, the total number of floating-point operations is:

$$\text{FLOPs} = 2 \times M \times N \times K$$

Independent Work Units (Parallelism)

Each output element $C_{i,j}$ is computed independently, resulting in $M \times N$ independent work units. There are no inter-element data dependencies, exposing a large two-dimensional parallelism space that maps naturally onto GPU thread blocks.

1.2 Compute vs Memory Analysis

Compute-bound vs Memory-bound Behavior

A naive GEMM implementation repeatedly loads elements of matrices A and B from global memory for each output computation. As a result, performance is limited by memory bandwidth, making the naive implementation **memory-bound**.

Optimized GEMM implementations employ tiling and shared memory to maximize data reuse. Each tile of A and B is loaded once and reused across multiple arithmetic operations. This significantly increases arithmetic intensity, shifting the workload to be **compute-bound** on modern GPUs.

Arithmetic Intensity

Arithmetic intensity (AI) is defined as:

$$\text{AI} = \frac{\text{Floating-Point Operations}}{\text{Global Memory Accesses}}$$

For GEMM:

$$\text{FLOPs} \approx 2MNK$$

- **Naive GEMM:** Low data reuse leads to $\text{AI} = O(1)$, resulting in memory-bound execution.
- **Optimized GEMM:** Tiling and reuse increase $\text{AI} \gg 1$, enabling compute-bound execution.

Shared Memory Reuse

- Tiles of matrix A are reused across multiple columns of C.
- Tiles of matrix B are reused across multiple rows of C.

Using shared memory reduces global memory traffic and improves cache efficiency, which is critical for achieving high performance on GPUs.

Parallelism and Scalability

Since each output element is independent, GEMM exhibits massive parallelism. GPUs can execute thousands of threads concurrently, allowing the workload to scale efficiently with increasing matrix sizes and hardware resources.

1.3 Expected Behavior on a GPU

- **Thread Mapping:** GPU threads are mapped to output elements or small tiles of C. Each thread computes one or more elements, ensuring uniform work distribution and minimal control divergence.
- **Scalability:** The large number of independent output elements enables high occupancy and efficient utilization of GPU cores. Tiled execution allows each thread to perform many fused multiply-add operations per memory load.
- **Potential Challenges:**

  - Warp divergence is minimal due to regular control flow.
  - Poor tiling strategies may reduce memory coalescing.
  - Small matrix sizes may lead to underutilization of GPU resources.
  - Synchronization overhead is required when using shared memory.

- **Expected Performance:** With proper tiling, coalesced memory access, and sufficient problem size, GEMM can achieve near-peak floating-point throughput on modern GPUs.

1.4 CPU Baseline

The CPU implementation of GEMM was benchmarked using `time.perf_counter()`, while the GPU version was executed using PyTorch with CUDA enabled. The measured runtime for the CPU was **0.01097 s**, whereas the GPU implementation achieved a runtime of **0.00190 s** (see Figures 1 and 2). This corresponds to an approximate **5.76× speedup** when using the GPU over the CPU for the same computation.

```
C:\Users\sunit>python
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import time
>>>
>>> # Matrix size
>>> N = 1024
>>>
>>> # CPU tensors
>>> A = torch.randn(N, N)
>>> B = torch.randn(N, N)
>>>
>>> # Warm-up
>>> _ = A @ B
>>>
>>> start = time.perf_counter()
>>> C = A @ B
>>> end = time.perf_counter()
>>>
>>> print("CPU time:", end - start)
CPU time: 0.010973899999953574
>>> exit()
```

Figure 1: CPU GEMM runtime measurement

```
(gpu) C:\Users\sunit>python
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import time
>>>
>>> # Matrix size
>>> N = 1024
>>>
>>> device = torch.device("cuda")
>>>
>>> A = torch.randn(N, N, device=device)
>>> B = torch.randn(N, N, device=device)
>>>
>>> # Warm-up
>>> _ = A @ B
>>> torch.cuda.synchronize()
>>>
>>> start = time.perf_counter()
>>> C = A @ B
>>> torch.cuda.synchronize()
>>> end = time.perf_counter()
>>>
>>> print("GPU time:", end - start)
GPU time: 0.0019047000000682601
>>>
```

Figure 2: GPU GEMM runtime measurement

Task 2: CUDA Execution Model Mapping Diagram

Problem Description

We consider a (1024×1024) matrix workload where each CUDA thread computes exactly one matrix element, as in element-wise operations or tiled GEMM. The objective is to efficiently map this iteration space onto the CUDA execution hierarchy of an NVIDIA GTX 1650 GPU.

The CUDA execution hierarchy is organized as:

$$\text{Grid} \rightarrow \text{Blocks} \rightarrow \text{Warps} \rightarrow \text{Threads}$$

Thread and Block Configuration

A two-dimensional thread block configuration is selected to improve spatial locality:

- Block dimensions: (16×16)
- Threads per block: 256
- Warp size: 32 threads

- Warps per block: $256/32 = 8$

The grid dimensions are:

$$\text{Grid.x} = 1024/16 = 64, \quad \text{Grid.y} = 1024/16 = 64$$

Thus, a total of $64 \times 64 = 4096$ thread blocks are launched.

Mapping Iteration Space to Threads

Each CUDA thread computes a single matrix element using:

```
row = blockIdx.y * blockDim.y + threadIdx.y;
col = blockIdx.x * blockDim.x + threadIdx.x;
```

Here, `blockIdx` selects the matrix tile and `threadIdx` identifies the thread's position within the tile, ensuring a one-to-one mapping between threads and matrix elements.

CUDA Execution Hierarchy Visualization

```
GRID (64 × 64 blocks)

 Block (16 × 16 threads = 256 threads)

    Warp 0  (32 threads)
    Warp 1  (32 threads)
    Warp 2  (32 threads)
    Warp 3  (32 threads)
    Warp 4  (32 threads)
    Warp 5  (32 threads)
    Warp 6  (32 threads)
    Warp 7  (32 threads)

 Each thread computes one matrix element
```

GTX 1650 Hardware Context

The NVIDIA GTX 1650 GPU (Turing architecture) provides:

- 14 Streaming Multiprocessors (SMs)
- 64 CUDA cores per SM
- 896 total CUDA cores
- Warp size: 32 threads

How Execution Occurs on the GPU

At runtime, thread blocks are scheduled onto available SMs. Each SM can host multiple blocks concurrently, depending on resource usage. Blocks are decomposed into warps, which are the fundamental scheduling unit and execute in a SIMT manner.

Each SM has 64 CUDA cores and can execute two warps (64 threads) simultaneously. When a warp stalls due to memory latency, another ready warp is scheduled, effectively hiding latency.

Synchronization Requirements

Synchronization is required only within a thread block. When shared memory is used, threads synchronize using:

```
__syncthreads();
```

This ensures all threads complete memory loads before computation begins. Synchronization across blocks is not possible within a kernel and is only achieved at kernel completion.

Shared Memory Optimization

To improve performance, shared memory is used to cache tiles of the matrices:

- Each block loads a (16×16) tile of matrix A
- Each block loads a (16×16) tile of matrix B
- Threads reuse these tiles across multiple computations

This approach significantly reduces global memory accesses and increases arithmetic intensity.

Potential Memory Bottlenecks

Without optimization:

- Frequent global memory accesses increase latency
- Redundant loads reduce effective bandwidth
- Poor memory coalescing increases memory transactions per warp

With shared memory tiling:

- Global memory accesses are coalesced
- Data reuse is maximized
- Memory bandwidth pressure is reduced

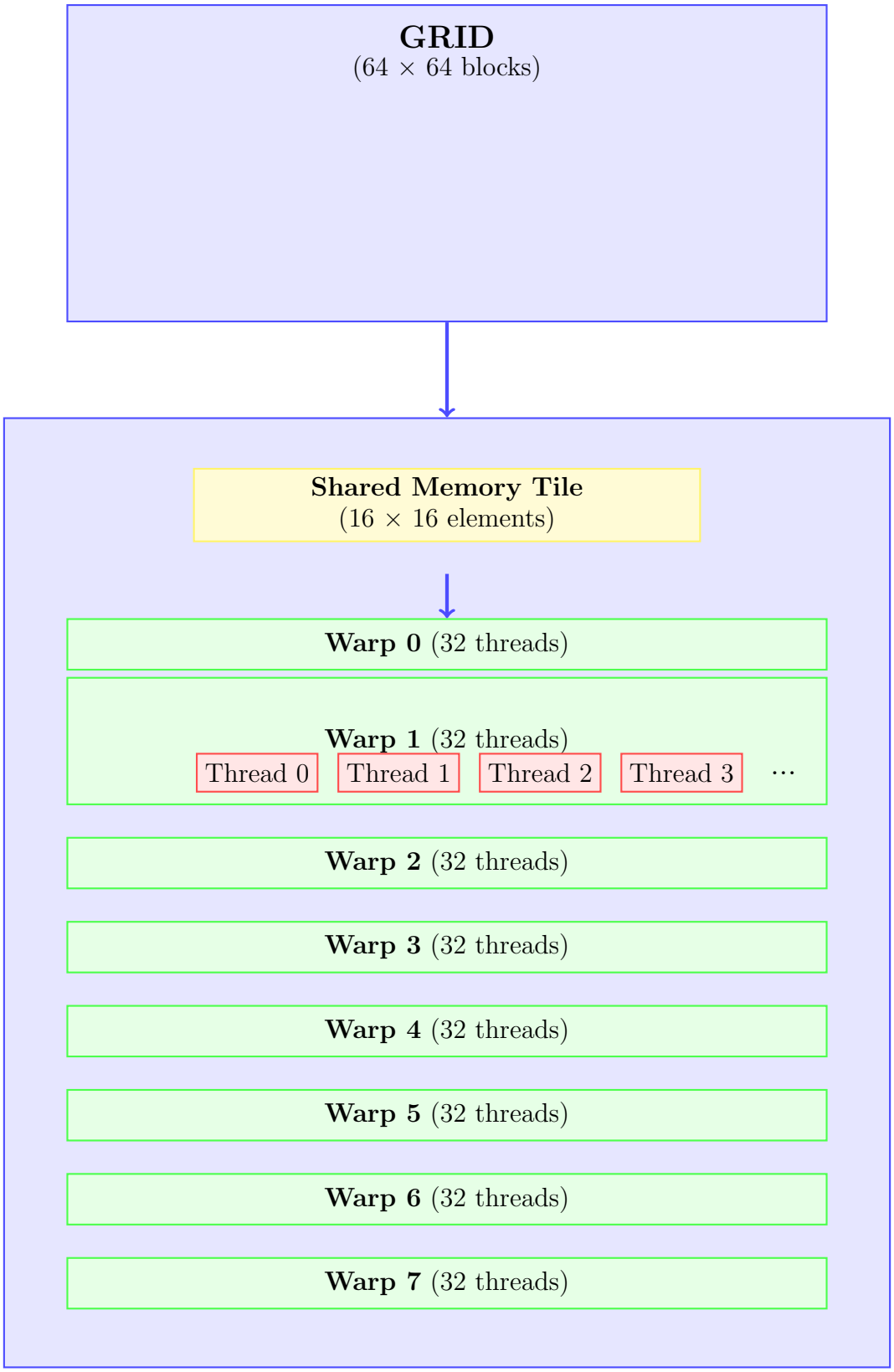
Warp-Level Behavior

All 32 threads in a warp execute the same instruction:

- Control flow is uniform
- Warp divergence is minimal
- Execution efficiency is high

Conclusion

A (1024×1024) matrix workload is mapped to a (64×64) grid of (16×16) thread blocks. Each block consists of 8 warps that are scheduled across the GTX 1650's SMs and executed in a SIMT fashion. By leveraging shared memory tiling and block-level synchronization, global memory traffic is minimized, enabling high throughput and efficient GPU utilization.



Hierarchy: Grid → Blocks → Warps → Threads
Total Threads per Block: 256 = 8 warps × 32 threads/warp