CS336 Language Modeling from Scratch

Lecture 5: GPU Architecture and Optimization Fundamentals

Stanford University, Spring 2025

Abstract

This lecture covers GPU architecture fundamentals and optimization techniques for language model training and inference. Key topics include GPU vs CPU design philosophies, memory hierarchy optimization, CUDA programming models, performance analysis techniques, and practical optimization strategies. The lecture prepares students for Assignment 2's Flash Attention implementation and provides essential foundations for understanding hardware-software co-design in deep learning.

Contents

1 Introduction and Course Context

1.1 Lecture Overview

This lecture focuses on understanding GPU architecture and optimization for language model training and inference, addressing the fundamental question of why GPUs exhibit mysterious performance patterns and how to write efficient code for them.

Key Point: GPU Performance Mystery

GPUs can show unpredictable wave-like performance patterns as matrix sizes change. Understanding the underlying architecture explains these behaviors and enables optimization.

Learning Objectives:

1. Demystify GPU architecture and CUDA programming
2. Understand why GPUs exhibit mysterious performance patterns
3. Learn to accelerate algorithms like Flash Attention
4. Develop intuition for hardware-efficient algorithm design

1.2 Assignment 2 Context

Algorithm: Assignment 2 Requirements

Students will implement Flash Attention 2 using Triton, requiring:

- Deep understanding of GPU memory hierarchy
- Profiling and benchmarking GPU code
- Kernel fusion and optimization principles
- Performance analysis at the PTX level

1.3 The Scaling Imperative

Historical Context: Single-thread CPU performance scaling ended around 2005 due to power and heat constraints, despite continued transistor density growth (Moore's Law without Dennard Scaling).

Modern Reality: Performance gains now come from parallel scaling rather than faster sequential execution, making GPU architecture understanding critical for deep learning.

[Insert diagram showing compute scaling trends: CPU vs GPU performance over time]

2 CPU vs GPU Design Philosophy

2.1 Fundamental Design Differences

Definition: [

CPU Optimization] CPUs optimize for latency by minimizing time to complete individual tasks through large control units, branch prediction, and complex instruction handling.

Aspect	**CPU**	**GPU**
Design Goal	Low Latency	High Throughput
Execution Model	Sequential	Massively Parallel
Core Count	4-32 cores	1000s of cores
Memory Focus	Large Caches	High Bandwidth
Control Logic	Complex	Simple
Thread Management	Heavy	Lightweight

Table 1: CPU vs GPU Design Comparison

Definition: [

GPU Optimization] GPUs optimize for throughput by maximizing total work completed per unit time through massive parallelism and lightweight thread management.

3 GPU Architecture Fundamentals

3.1 Hierarchical Structure

GPU Hierarchy:

1. **GPU:** Complete accelerator device
2. **SM (Streaming Multiprocessor):** Atomic control unit (128 SMs in A100)
3. **SP (Streaming Processor):** Individual compute elements within SMs
4. **Tensor Cores:** Specialized matrix multiplication units

[Insert diagram of GPU chip layout showing SM distribution and memory hierarchy]

3.2 Memory Hierarchy

Speed and Proximity Relationship:

- **Registers & L1/Shared Memory:** 20 clock cycles (inside SM)
- **L2 Cache:** 100 clock cycles (on-chip, adjacent to SMs)
- **Global Memory (HBM):** 200-300 clock cycles (off-chip)

Critical Insight: 10x speed difference between on-SM and global memory creates the fundamental optimization challenge.

4 CUDA Programming Model

4.1 Execution Granularity

Definition: [

CUDA Execution Model] GPU computation is organized in a three-level hierarchy:

- **Grid:** Collection of blocks
- **Block:** Group of threads assigned to one SM
- **Warp:** 32 consecutive threads executing in lockstep

SIMT (Single Instruction, Multiple Thread): All threads in a warp execute the same instruction on different data.

4.2 Memory Model

Logical Memory Types:

- **Registers:** Private to each thread, fastest access
- **Shared Memory:** Shared within a block, very fast
- **Global Memory:** Accessible by all threads, slowest
- **Constant Memory:** Read-only, cached for repeated access

Programming Principle: Minimize global memory access by maximizing use of shared memory and registers.

5 TPU Comparison

5.1 TPU Architecture Overview

TPUs share conceptual similarities with GPUs but are specialized for matrix operations:

- **Tensor Core:** Analogous to GPU SM
- **MXU (Matrix Multiply Unit):** Specialized for matrix operations only
- **Vector Unit:** For element-wise operations
- **Scalar Unit:** Control and CPU-like operations

Key Difference: TPUs are more specialized (matrix-multiply focused) while GPUs are more general-purpose parallel processors.

6 Performance Scaling Analysis

6.1 Component Scaling Trends

[Insert chart showing normalized performance scaling over GPU generations]

Scaling Rates (Normalized):

- **Compute (Matrix Multiply):** 100,000x improvement

- **Memory Bandwidth:** 100x improvement
- **Interconnect:** Modest improvement

Implication: Modern workloads are increasingly memory-bound rather than compute-bound, fundamentally changing optimization strategies.

6.2 The Roofline Model

$$\text{Performance} = \min\left(\text{Peak Compute}, \text{Memory BW} \times \text{Arithmetic Intensity}\right) \tag{1}$$

where Arithmetic Intensity = $\frac{\text{Operations}}{\text{Bytes Transferred}}$
Performance Regimes:

- **Memory-Bound:** Left side of roofline (low arithmetic intensity)
- **Compute-Bound:** Right side of roofline (high arithmetic intensity)

7 GPU Optimization Strategies

7.1 1. Avoiding Conditional Divergence

Problem: Conditional statements within warps cause serialization.
Example of Poor Performance:

```
if (threadIdx.x < 4) {
    // Some computation A
} else {
    // Some computation B
}
```

Execution: GPU must execute both branches sequentially, halving effective utilization.

7.2 2. Lower Precision Arithmetic

Memory Bandwidth Impact:
For element-wise ReLU operation on vector of size n:

- **FP32:** 8 bytes per operation (read + write)
- **FP16:** 4 bytes per operation
- **Effective Speedup:** 2x memory bandwidth improvement

Mixed Precision Strategy:

- **Inputs:** FP16 for memory efficiency
- **Accumulation:** FP32 for numerical stability
- **Special Functions:** BF16 for extended dynamic range

7.3 3. Operator Fusion

Definition: [

Kernel Fusion] Combining multiple operations into a single kernel to minimize memory traffic by keeping intermediate results in fast on-chip memory.

Example Fusion Opportunity:

```
# Naive approach (5 separate kernels)
sin_x = torch.sin(x)
cos_x = torch.cos(x)
sin2_x = sin_x ** 2
cos2_x = cos_x ** 2
result = sin2_x + cos2_x

# Fused approach (1 kernel)
result = torch.sin(x)**2 + torch.cos(x)**2
```

Tools: `torch.compile` automatically performs many fusion optimizations.

7.4 4. Recomputation

Trade-off: Exchange excess compute for scarce memory bandwidth.
Example Analysis:

- **Standard Backprop:** Store all activations (8 memory ops)
- **Recomputation:** Store only inputs (5 memory ops, extra compute)
- **Net Benefit:** 37.5% reduction in memory traffic when compute-bound

7.5 5. Memory Coalescing

Definition: [

Memory Coalescing] Hardware optimization where multiple memory requests from a warp are combined into fewer, larger transactions when accessing contiguous memory.

Burst Mode: DRAM returns multiple values (e.g., 4 elements) for each access request.
Access Pattern Impact:

- **Coalesced:** Warp threads access consecutive addresses
- **Uncoalesced:** Random access patterns require separate transactions
- **Performance Gap:** 4x bandwidth difference

[Insert diagram showing coalesced vs uncoalesced memory access patterns]

7.6 6. Tiling Strategies

Definition: [

Matrix Tiling] Decomposing large matrix operations into smaller submatrix operations that fit in fast shared memory, minimizing global memory access.

Tiling Algorithm for Matrix Multiplication:

1. Load submatrices (tiles) from global to shared memory
2. Perform all possible computations using loaded tiles
3. Write results and proceed to next tile
4. Repeat until complete

Memory Access Reduction:

$$\text{Global Memory Reads} = \frac{N^3}{T} + \text{overhead} \tag{2}$$

where N is matrix size and T is tile size.

Optimal Tile Size Considerations:

- Must fit in shared memory
- Should align with memory burst boundaries
- Should divide matrix dimensions evenly
- GPU hardware prefers multiples of 32, 64, 128

8 Performance Mystery Decoded

8.1 Matrix Multiplication Performance Patterns

[Insert performance chart showing wavy patterns in GEMM throughput vs matrix size]

Performance Factors Explained:

1. **Roofline Behavior:** Small matrices are memory-bound, large matrices approach peak compute
2. **Tiling Alignment:** Performance drops when matrix dimensions are not multiples of tile sizes
    - Divisible by 32: Excellent performance (purple line)
    - Divisible by 16: Good performance
    - Divisible by 8: Moderate performance
    - Prime dimensions: Poor performance
3. **Memory Burst Alignment:** Additional performance drops when tiles don't align with DRAM burst boundaries
4. **SM Utilization:** Uneven workload distribution across SMs when tile counts don't divide evenly

Real-World Example: Increasing vocabulary size from 50,257 to 50,304 (nearest multiple of 64) yielded 25% speedup in nanoGPT.

9 Flash Attention Case Study

9.1 Attention Memory Challenge

Standard Attention Memory Complexity:

$$\text{Memory} = O(N^2) \text{ for sequence length } N \tag{3}$$

Problem: Intermediate attention matrices don't fit in GPU memory for long sequences.

9.2 Flash Attention Solution

Key Innovations:

1. **Tiling:** Compute attention in blocks that fit in shared memory
2. **Recomputation:** Recompute attention scores during backward pass
3. **Online Softmax:** Incremental softmax computation without storing full matrix

Memory Reduction:

$$\text{HBM Access} = O(N^2 d/M) \tag{4}$$

where d is head dimension and M is shared memory size.

Result: Sub-quadratic memory access while maintaining exact attention computation.

10 Optimization Best Practices

10.1 Algorithm Design Principles

1. **Maximize Arithmetic Intensity:** Reuse data loaded into fast memory
2. **Minimize Global Memory Access:** Use shared memory and registers aggressively
3. **Ensure Memory Coalescing:** Design access patterns for consecutive memory reads
4. **Choose Matrix Dimensions Wisely:** Prefer multiples of 32, 64, 128
5. **Leverage Mixed Precision:** Use appropriate numerical precision for each operation
6. **Enable Compiler Optimizations:** Use tools like `torch.compile`

10.2 Performance Debugging

Common Performance Issues:

- Uncoalesced memory access patterns
- Poor tile size choices
- Excessive global memory round-trips
- Warp divergence from conditionals
- Suboptimal matrix dimensions

Profiling Tools: NVIDIA Nsight, PyTorch Profiler, Triton profiling

11 Future Implications

11.1 Hardware Trends

Continuing Patterns:

- Compute scaling continues to outpace memory scaling
- Increasing importance of memory hierarchy optimization
- Growing specialization in matrix operations
- Enhanced support for lower precision arithmetic

11.2 Algorithm Design Evolution

Emerging Priorities:

- Memory-efficient algorithm variants
- Hardware-aware neural architecture design
- Automatic optimization through compilation
- Co-design of algorithms and hardware

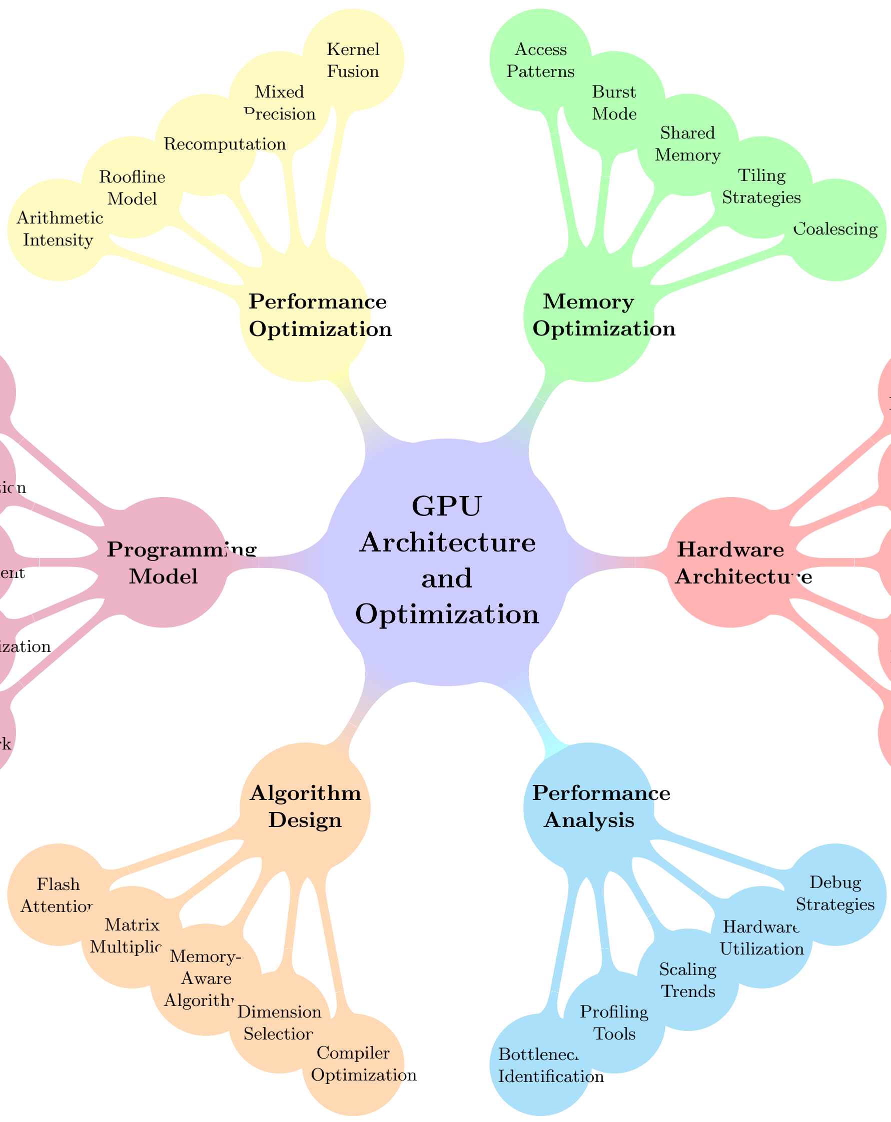
12 Conclusion

Understanding GPU architecture is essential for developing efficient language models. Key takeaways:

1. **Memory is the Bottleneck:** Modern GPUs are memory-bound, not compute-bound
2. **Hierarchy Exploitation:** Success requires effective use of memory hierarchy
3. **Pattern Awareness:** Memory access patterns dramatically affect performance
4. **Algorithmic Innovation:** Best performance comes from hardware-aware algorithm design
5. **Tool Utilization:** Modern compilers provide significant automatic optimization

The mysterious performance patterns in GPU computing become predictable when viewed through the lens of memory hierarchy, tiling strategies, and hardware constraints. This understanding enables the development of algorithms like Flash Attention that achieve dramatic performance improvements through clever memory management.

13 Visual Mind Map



13.1 Mind Map Structure Description

The mind map centers on "GPU Architecture and Optimization" with six main conceptual branches:

1. **Hardware Architecture (Red):** Foundation concepts including SM structure, memory hierarchy, tensor cores, SIMT execution model, and warp-based parallelism.
2. **Memory Optimization (Green):** Critical memory-related optimizations including coalescing techniques, tiling strategies, shared memory usage, burst mode understanding, and access pattern design.
3. **Performance Optimization (Yellow):** High-level optimization strategies including kernel fusion, mixed precision arithmetic, recomputation techniques, roofline model analysis, and arithmetic intensity optimization.
4. **Programming Model (Purple):** Practical programming aspects including CUDA kernel development, block organization, thread management, synchronization primitives, and modern frameworks like Triton.

5. **Algorithm Design (Orange):** Application-level considerations including Flash Attention implementation, matrix multiplication optimization, memory-aware algorithm development, dimension selection strategies, and compiler optimization utilization.
6. **Performance Analysis (Cyan):** Debugging and analysis techniques including bottleneck identification, profiling tool usage, hardware scaling trend understanding, utilization analysis, and systematic debugging approaches.

This structure reflects the lecture's progression from fundamental hardware concepts through practical optimization techniques to real-world algorithm implementation, emphasizing the critical relationship between understanding hardware constraints and achieving optimal performance in GPU computing.