CS336 Language Modeling from Scratch

Lecture 6: GPU Kernels, CUDA, and Triton Programming

Stanford University, Spring 2025

Abstract

This lecture covers high-performance GPU programming for language models, focusing on writing efficient CUDA kernels and using Triton for kernel development. Key topics include benchmarking and profiling GPU code, understanding kernel fusion for performance optimization, implementing custom kernels in CUDA C++, leveraging Triton for Python-like kernel programming, and utilizing PyTorch's JIT compilation. The lecture demonstrates these concepts through practical examples including GLU activation functions and softmax operations, providing essential skills for Assignment 2's Flash Attention implementation.

Contents

1 Introduction and Course Context

1.1 Lecture Overview

This lecture focuses on the practical aspects of writing high-performance code for GPUs, particularly in the context of deep learning and language model training. The lecture serves as preparation for Assignment 2, which requires implementing Flash Attention 2 using Triton kernels.

Learning Objectives:

1. Master GPU benchmarking and profiling techniques
2. Understand kernel fusion and its performance benefits
3. Learn to write CUDA kernels in C++
4. Explore Triton for high-level kernel programming
5. Utilize PyTorch's JIT compilation for automatic optimization
6. Analyze performance at the PTX (assembly) level

1.2 Assignment 2 Context

Assignment 2 requires students to:

- Implement comprehensive profiling of GPU workloads
- Write custom Triton kernels for Flash Attention 2
- Optimize code for high performance on modern GPUs
- Apply parallelism concepts (covered in next lecture)

[Insert diagram showing the progression from PyTorch operations to optimized kernels]

2 GPU Architecture Review

2.1 Core GPU Components

Definition: Streaming Multiprocessor (SM)

The fundamental processing unit of a GPU containing multiple compute cores, shared memory, and scheduling logic.

Key Components:

- **Compute Units:** INT32 and FP32 processing cores within each SM
- **Memory Hierarchy:**
  - DRAM/Global Memory: Large capacity, high latency
  - L1 Cache: Fast access, limited capacity
  - Register File: Fastest access, thread-private storage
- **Thread Execution:** Large number of threads per SM for latency hiding

2.2 Execution Model

Hierarchical Structure:

1. **Grid:** Collection of thread blocks
2. **Thread Blocks:** Scheduled on single SM, can communicate via shared memory
3. **Threads:** Individual execution units within blocks

Key Point: Thread Block Communication

Thread blocks can communicate through shared memory (fast), while cross-block communication requires global memory (slow).

2.3 Warps and Performance Considerations

Definition: Warp

A group of 32 consecutive threads executed together with shared control flow.

Performance Implications:

- **Warp Scheduling:** Reduces control overhead by executing 32 threads simultaneously
- **Load Balancing:** Prefer thread block counts that divide evenly across SMs
- **Equal Work Distribution:** Ensure warps have balanced computational loads

2.4 Arithmetic Intensity

$$\text{Arithmetic Intensity} = \frac{\text{Floating Point Operations}}{\text{Bytes of Memory Movement}} \tag{1}$$

Performance Principle: High arithmetic intensity is crucial because compute scales much faster than memory bandwidth.

- **Matrix Multiplication:** Compute-bound when implemented efficiently
- **Element-wise Operations:** Generally memory-bound
- **Optimization Goal:** Reduce memory-bound operations through kernel fusion

3 Benchmarking GPU Code

3.1 Benchmarking Fundamentals

Key Point: Always Profile Before Optimizing

Use detailed profilers to identify actual bottlenecks rather than assumptions about performance issues.

Benchmarking vs Profiling:

- **Benchmarking:** Measures end-to-end execution time
- **Profiling:** Provides detailed breakdown of where time is spent

3.2 Proper Benchmarking Technique

Code: Benchmark Function Structure

```
def benchmark(fn, warmup=3, trials=10):
    # Warm-up phase
    for _ in range(warmup):
        fn()
        torch.cuda.synchronize()

    # Timing phase
    times = []
    for _ in range(trials):
        torch.cuda.synchronize()
        start = time.time()
        result = fn()
        torch.cuda.synchronize()
        end = time.time()
        times.append(end - start)

    return np.mean(times)
```

Critical Requirements:

1. **Warm-up Iterations:** First execution includes compilation overhead
   - Machine code compilation happens on first run
   - GPU initialization and memory allocation
   - Kernel loading and dispatch setup
2. **CUDA Synchronization:** Essential for accurate timing
   - GPU and CPU execute asynchronously
   - CPU dispatches kernels and continues execution
   - `torch.cuda.synchronize()` ensures GPU completion
3. **Multiple Trials:** Account for thermal and scheduling variations

Warning: Common Benchmarking Mistakes

Without proper synchronization, matrix multiplications may appear to complete "instantly" because you're only measuring kernel dispatch time, not execution time.

3.3 Matrix Multiplication Scaling Analysis

Empirical Results on H100 GPUs:

[Insert performance scaling chart showing matrix multiply times vs dimensions]

- **Small Matrices (1024×1024):** Constant overhead dominates
- **Large Matrices:** Super-linear scaling as expected ($O(n^3)$ complexity)
- **Overhead Sources:** CPU-GPU data transfer, kernel launch costs

3.4 MLP Benchmarking Example

Test Configuration:

- 256 dimensions, 4 layers, batch size 256
- Forward and backward passes
- Linear scaling with steps and layers confirmed

Results: Approximately 5 seconds per MLP execution, demonstrating linear scaling with both number of steps and number of layers.

4 GPU Profiling Techniques

4.1 PyTorch Built-in Profiler

Code: Basic Profiling Setup

```
def profile_function(fn, warmup=3):
    # Warm-up
    for _ in range(warmup):
        fn()
        torch.cuda.synchronize()

    # Profile
    with torch.profiler.profile(
        activities=[torch.profiler.ProfilerActivity.CPU,
                    torch.profiler.ProfilerActivity.CUDA]
    ) as prof:
        fn()
        torch.cuda.synchronize()

    print(prof.key_averages().table(sort_by="cuda_time_total"))
```

4.2 Profiling Results Analysis

Vector Addition Example:

- **High-level Call:** `aten::add` (PyTorch interface)
- **Kernel Dispatch:** `vectorized_elementwise_kernel`
- **Launch Overhead:** `cudaLaunchKernel`
- **Synchronization:** `cudaDeviceSynchronize`

Matrix Multiplication Example:

- **Interface:** `aten::mm`
- **Library:** cuBLAS (NVIDIA's optimized BLAS)
- **Kernel:** Specific cuBLAS kernel with tile size parameters
- **Size Dependency:** Different kernels for different matrix sizes

4.3 Advanced Profiling with NVIDIA Nsight

Nsight Systems Capabilities:

- **Timeline View:** CPU and GPU execution timelines
- **Kernel Analysis:** Individual kernel performance
- **Memory Tracking:** GPU memory allocation and usage
- **Annotations:** Custom markers for code regions

Code: Adding NVTX Annotations

```
import torch.profiler

# Annotate code regions
with torch.profiler.record_function("forward_pass"):
    output = model(input)

with torch.profiler.record_function("backward_pass"):
    loss.backward()
```

4.4 CPU-GPU Execution Model

Asynchronous Execution:

1. CPU dispatches CUDA kernels to GPU queue
2. CPU continues execution without waiting
3. GPU executes kernels from queue
4. Synchronization points force CPU to wait for GPU

Key Point: Print Statement Impact

Adding print statements can dramatically affect GPU utilization by forcing CPU-GPU synchronization, changing the execution pattern from asynchronous to synchronous.

[Insert timeline diagram showing CPU ahead of GPU execution vs synchronized execution]

5 Kernel Fusion and Performance

5.1 The Kernel Fusion Concept

Definition: Kernel Fusion

Combining multiple operations into a single GPU kernel to reduce memory traffic between global memory and compute units.

Memory Traffic Problem: [Insert factory analogy diagram: multiple operations = multiple warehouse trips]

- **Naive Approach:** Each operation requires global memory read/write
- **Fused Approach:** Single kernel performs all operations in registers/shared memory
- **Performance Benefit:** Dramatically reduced memory bandwidth requirements

5.2 GELU Implementation Case Study

GELU Approximation Formula:

$$\text{GELU}(x) = 0.5 \times x \times \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}} \times (x + 0.044715 \times x^3)\right)\right) \tag{2}$$

Implementation Comparison:

1. **Manual (Unfused):** 8.1 milliseconds
   - Multiple separate operations: x^3, multiplication, tanh, etc.
   - Multiple kernel launches and memory round-trips
   - Poor performance due to memory bandwidth limitations
2. **PyTorch (Fused):** 1.1 milliseconds
   - Single optimized kernel
   - All operations performed in registers
   - 8× performance improvement

Profiling Evidence:

- **Manual Version:** Multiple `vectorized_elementwise_kernel` calls
- **Fused Version:** Single kernel execution consuming 100% GPU time

6 Writing CUDA Kernels

6.1 CUDA Programming Model

Kernel Function Structure:

- **Kernel:** GPU function executed by many threads
- **Host Function:** CPU wrapper that launches kernel
- **Grid/Block Organization:** Hierarchical thread organization

Code: CUDA Kernel Template

```
__global__ void kernel_function(float* input, float* output, int size) {
    // Calculate thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size) {
        // Perform computation
        output[idx] = computation(input[idx]);
    }
}

// Host launch function
void launch_kernel(torch::Tensor input, torch::Tensor output) {
    int threads_per_block = 256;
    int blocks = (input.numel() + threads_per_block - 1) / threads_per_block;

    kernel_function<<<blocks, threads_per_block>>>(
        input.data_ptr<float>(),
        output.data_ptr<float>(),
        input.numel()
    );
}
```

6.2 Thread Indexing and Grid Configuration

Index Calculation:

- **Block Index:** `blockIdx.x` (which block in grid)
- **Thread Index:** `threadIdx.x` (which thread in block)
- **Block Dimension:** `blockDim.x` (threads per block)
- **Global Index:** `blockIdx.x * blockDim.x + threadIdx.x`

Grid Configuration Considerations:

- **Block Size:** Typically 256 or 512 threads per block
- **Grid Size:** Enough blocks to cover all data elements
- **SM Utilization:** More blocks than SMs for load balancing

6.3 CUDA GELU Implementation

Code: Complete CUDA GELU Kernel

```
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void gelu_kernel(const float* x, float* out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size) {
        float val = x[idx];
        float val_cubed = val * val * val;
        float tanh_arg = 0.7978845608f * (val + 0.044715f * val_cubed);
        float tanh_val = tanhf(tanh_arg);
        out[idx] = 0.5f * val * (1.0f + tanh_val);
    }
}

torch::Tensor gelu_cuda(torch::Tensor x) {
    // Input validation
    TORCH_CHECK(x.is_cuda(), "Input must be CUDA tensor");
    TORCH_CHECK(x.is_contiguous(), "Input must be contiguous");

    auto output = torch::empty_like(x);

    const int threads = 256;
    const int blocks = (x.numel() + threads - 1) / threads;

    gelu_kernel<<<blocks, threads>>>(
        x.data_ptr<float>(),
        output.data_ptr<float>(),
        x.numel()
    );

    return output;
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("gelu", &gelu_cuda, "GELU activation (CUDA)");
}
```

6.4 Performance Results

CUDA Implementation Results:

- **CUDA Time:** 1.8 milliseconds
- **vs Manual:** 4.5× improvement (8.1 → 1.8 ms)
- **vs PyTorch:** 1.6× slower (1.1 vs 1.8 ms)
- **Single Kernel:** 100% GPU utilization

Performance Analysis:

- Successfully achieved kernel fusion benefits

- Close to PyTorch's optimized implementation
- Room for further optimization (register usage, memory coalescing)

6.5 Essential CUDA Debugging

Warning: CUDA Debugging Requirements

Always launch CUDA programs with `CUDA_LAUNCH_BLOCKING=1` environment variable for debugging. Without this, CUDA errors are reported asynchronously and may not provide useful error messages.

7 Triton Programming

7.1 Triton Overview

Definition: Triton

A Python-like domain-specific language for writing high-performance GPU kernels with automatic optimization and memory management.

Triton Advantages:

- **Python-like Syntax:** Easier to write and maintain than CUDA C++
- **Automatic Optimization:** Compiler handles memory coalescing and other optimizations
- **Block-based Programming:** Abstract away low-level thread management
- **Type Safety:** Better error checking than raw CUDA

7.2 Triton Programming Model

Core Concepts:

- **Program:** Executes on a block of threads
- **Pointers:** Block-level memory access patterns
- **Masks:** Handle boundary conditions automatically
- **Tiling:** Built-in support for blocked algorithms

Code: Triton GELU Implementation

```
import triton
import triton.language as tl

@triton.jit
def gelu_kernel(x_ptr, output_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    # Calculate block start
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    # Load data
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)

    # Compute GELU
    x_cubed = x * x * x
    tanh_arg = 0.7978845608 * (x + 0.044715 * x_cubed)
    tanh_val = tl.math.tanh(tanh_arg)
    output = 0.5 * x * (1.0 + tanh_val)

    # Store result
    tl.store(output_ptr + offsets, output, mask=mask)

def gelu_triton(x):
    output = torch.empty_like(x)
    n_elements = x.numel()

    # Choose block size
    BLOCK_SIZE = 1024
    grid = (triton.cdiv(n_elements, BLOCK_SIZE),)

    gelu_kernel[grid](x, output, n_elements, BLOCK_SIZE)
    return output
```

7.3 Triton Performance Characteristics

Performance Results:

- **Triton Time:** 1.848 milliseconds
- **vs CUDA:** Essentially identical performance
- **Development Time:** Significantly faster to implement
- **Maintainability:** Easier to understand and modify

When to Use Triton:

- **Element-wise Operations:** Excellent performance with simple syntax
- **Reduction Operations:** Built-in support for common patterns
- **Complex Algorithms:** Better abstraction for sophisticated operations
- **Rapid Prototyping:** Faster development cycle than CUDA

7.4 Triton Memory Management

Automatic Optimizations:

- **Memory Coalescing:** Automatic optimization of memory access patterns
- **Register Allocation:** Efficient use of register file
- **Shared Memory:** Automatic management of SM shared memory
- **Block Scheduling:** Optimal block size and grid configuration

8 PyTorch JIT Compilation

8.1 Torch Compile Overview

Definition: Torch Compile

PyTorch's just-in-time compiler that automatically optimizes Python code by generating fused kernels and applying hardware-specific optimizations.

Automatic Optimizations:

- **Kernel Fusion:** Combines multiple operations into single kernels
- **Matrix Multiply Optimization:** Automatically selects best GEMM kernels
- **Memory Layout:** Optimizes tensor layouts for hardware
- **Graph Optimization:** Eliminates redundant operations

8.2 Using Torch Compile

Code: Torch Compile Usage

```
# Original unoptimized function
def manual_gelu(x):
    return 0.5 * x * (1 + torch.tanh(
        0.7978845608 * (x + 0.044715 * x**3)
    ))

# Compile for optimization
compiled_gelu = torch.compile(manual_gelu)

# Use compiled version
x = torch.randn(1000000, device='cuda')
result = compiled_gelu(x)  # Automatically optimized
```

Performance Results:

- **Compiled Time:** 1.3 milliseconds
- **vs Manual:** 6.2× improvement
- **vs PyTorch Native:** Slightly better (hardware-specific optimizations)
- **Development Effort:** Minimal - just add `@torch.compile` decorator

8.3 When Torch Compile Excels

Excellent Performance:

- **Simple Operator Fusion:** Element-wise operations
- **Matrix Multiply Optimization:** Automatic kernel selection
- **Known Shapes:** Static shape optimization
- **Standard Operations:** Well-understood operation patterns

Limitations:

- **Complex Algorithms:** Flash Attention requires custom implementation
- **Hardware-Specific Optimizations:** Advanced H100 features
- **Novel Patterns:** Operations not in compiler's knowledge base
- **Dynamic Shapes:** Performance may degrade with dynamic inputs

9 Low-Level Analysis: PTX Assembly

9.1 PTX Overview

Definition: PTX (Parallel Thread Execution)

NVIDIA's virtual assembly language that provides a hardware-independent representation of GPU programs before final compilation to machine code.

PTX Analysis Benefits:

- **Performance Understanding:** See actual operations performed
- **Register Usage:** Understand memory access patterns
- **Optimization Verification:** Confirm compiler optimizations
- **Debugging:** Identify performance bottlenecks

9.2 GELU PTX Analysis

Key PTX Observations:

- **Vectorization:** Each thread processes 4 values simultaneously
- **Register Usage:** Temporary values stored in registers (R38-R41)
- **Memory Access:** Efficient load/store patterns
- **Floating-Point Operations:** All computations in registers

Performance Implications:

- **High Register Utilization:** Minimizes memory access
- **Vectorized Operations:** Maximizes throughput per thread
- **Efficient Storage:** Optimal use of fastest memory tier

10 Advanced Topics and Future Directions

10.1 Softmax Implementation Comparison

Performance Comparison (Large Softmax):

Implementation	Time (ms)	Speedup
Manual (unfused)	8.1	1.0×
Torch Compile	1.3	6.2×
PyTorch Native	1.5	5.4×
Triton	1.9	4.3×
CUDA C++	1.8	4.5×

Key Insights:

- **Fusion Dominates:** Kernel fusion provides largest performance gains
- **Torch Compile Competitive:** Often matches or exceeds hand-written kernels
- **Implementation Complexity:** Diminishing returns for development effort

10.2 Development Strategy Recommendations

Key Point: Optimization Hierarchy

1. Start with torch.compile for automatic optimization
2. Use Triton for custom algorithms requiring kernel-level control
3. Resort to CUDA C++ only for hardware-specific optimizations
4. Always profile to validate performance assumptions

When to Write Custom Kernels:

- **Novel Algorithms:** Flash Attention, custom attention variants
- **Hardware Exploitation:** H100-specific features
- **Memory Pattern Optimization:** Complex tiling strategies
- **Research Applications:** Algorithm development and experimentation

10.3 Flash Attention Context

Why Flash Attention Needs Custom Kernels:

- **Complex Memory Patterns:** Sophisticated tiling strategies
- **Algorithmic Innovation:** Online softmax computation
- **Memory Hierarchy Exploitation:** Careful shared memory management
- **Numerical Stability:** Custom accumulation patterns

Assignment 2 Preparation:

- Master Triton programming paradigms

- Understand tiling and blocking strategies
- Practice profiling and optimization workflows
- Implement progressively complex reduction operations

11 Best Practices and Common Pitfalls

11.1 Performance Optimization Guidelines

Measurement and Analysis:

1. **Always Profile First:** Identify actual bottlenecks
2. **Use Proper Benchmarking:** Warm-up, synchronization, averaging
3. **Measure End-to-End:** Include all overheads in performance analysis
4. **Validate Assumptions:** Use profiling tools to confirm optimizations

Development Process:

1. **Start Simple:** Begin with torch.compile
2. **Iterative Optimization:** Make incremental improvements
3. **Validate Correctness:** Test numerical accuracy at each step
4. **Document Performance:** Track improvements and regressions

11.2 Common Performance Mistakes

Warning: Benchmarking Errors

- Forgetting CUDA synchronization (measuring dispatch time, not execution time)
- Skipping warm-up iterations (including compilation overhead)
- Not accounting for thermal effects (single measurements)

Development Mistakes:

- **Premature Optimization:** Optimizing before profiling
- **Micro-Optimization Focus:** Ignoring algorithmic improvements
- **Platform Assumptions:** Optimizing for wrong hardware
- **Correctness Compromise:** Trading accuracy for speed inappropriately

11.3 Hardware-Aware Programming

Memory Hierarchy Considerations:

- **Global Memory:** Minimize access through fusion and tiling
- **Shared Memory:** Exploit for block-level data sharing
- **Registers:** Maximize reuse of frequently accessed data

- **Cache Efficiency:** Design access patterns for spatial locality

Compute Utilization:

- **Warp Efficiency:** Avoid divergent control flow
- **Occupancy:** Balance register usage with thread count
- **Instruction Mix:** Favor operations with hardware acceleration
- **Load Balancing:** Ensure even work distribution across SMs

12 Conclusion and Next Steps

12.1 Key Takeaways

1. **Profiling is Essential:** Always measure before optimizing
2. **Kernel Fusion Dominates:** Biggest performance gains come from reducing memory traffic
3. **Tool Hierarchy:** torch.compile → Triton → CUDA for increasing complexity
4. **Hardware Understanding:** Memory hierarchy knowledge enables effective optimization
5. **Iterative Development:** Build complexity gradually with validation at each step

12.2 Assignment 2 Preparation

Required Skills:

- Triton kernel programming
- GPU profiling and optimization
- Flash Attention algorithm understanding
- Memory hierarchy exploitation
- Performance validation techniques

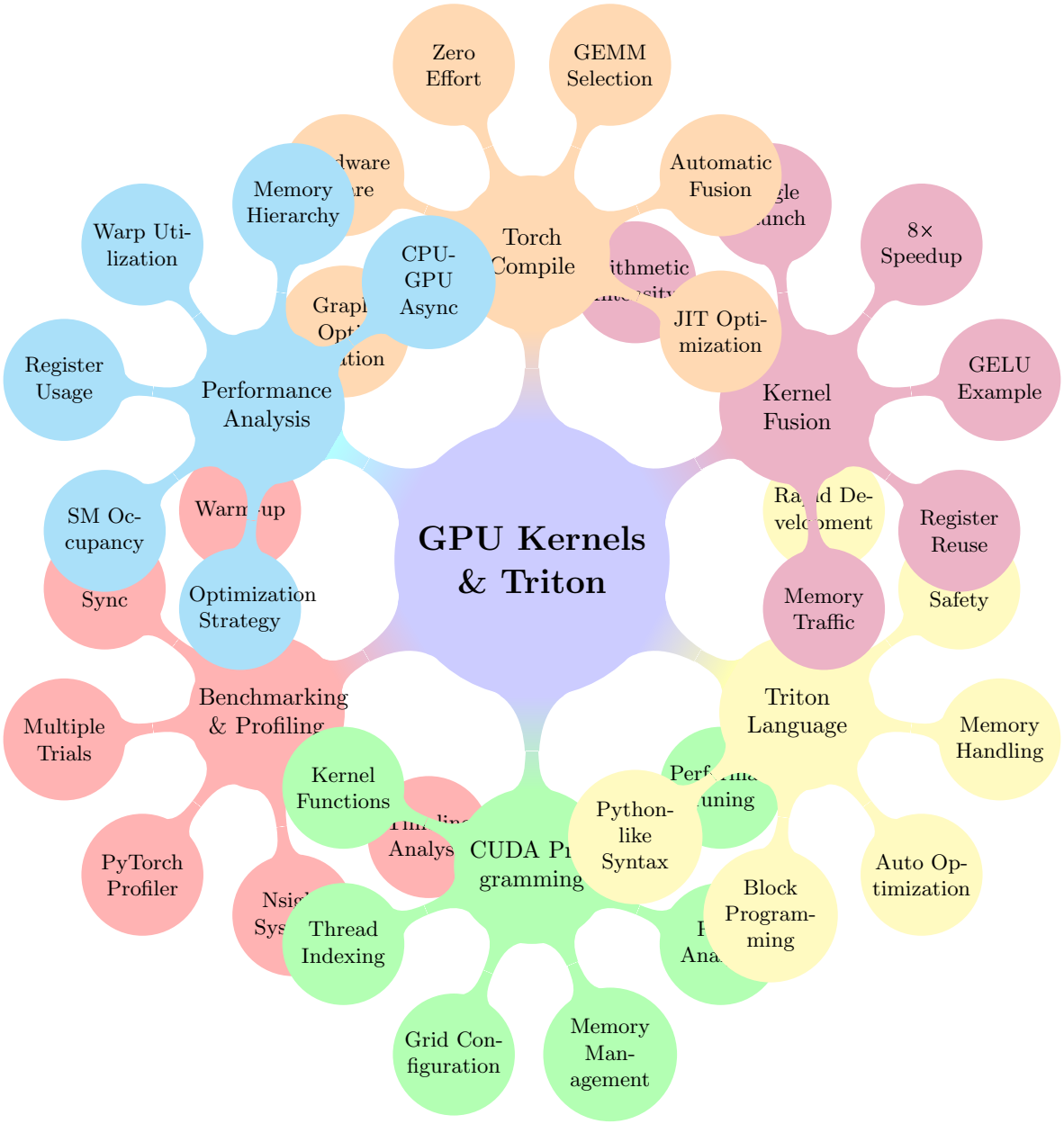
Next Lecture Preview:

- Parallelism and distributed training
- Data parallelism implementation
- Communication optimization
- Scaling strategies for large models

12.3 Further Reading

- **Triton Documentation:** Official programming guide and tutorials
- **CUDA Programming Guide:** Comprehensive GPU programming reference
- **Flash Attention Papers:** Algorithmic foundations and optimizations
- **NVIDIA Profiling Tools:** Nsight Systems and Compute documentation

13 Visual Mind Map



Mind Map Structure:
Central Concept: GPU Kernels & Triton Programming
Main Branches:

1. **Benchmarking & Profiling (Red):** Essential measurement techniques - Warm-up iterations, CUDA synchronization, multiple trials - PyTorch profiler for basic analysis - Nsight Systems for detailed timeline analysis

2. **CUDA Programming (Green):** Low-level GPU programming - Kernel functions and thread management - Grid configuration and memory handling - PTX assembly analysis for optimization

3. **Triton Language (Yellow):** High-level kernel programming - Python-like syntax with automatic optimizations - Block-based programming model - Rapid development with performance

4. **Kernel Fusion (Purple):** Core optimization strategy - Reducing memory traffic through operation combination - GELU implementation demonstrating 8× speedup - Maxi-

mizing arithmetic intensity

5. **Torch Compile (Orange):** Automatic optimization - JIT compilation with zero developer effort - Automatic kernel fusion and hardware optimization - Competitive performance with manual implementations

6. **Performance Analysis (Cyan):** Understanding GPU behavior - CPU-GPU asynchronous execution model - Memory hierarchy exploitation strategies - Warp utilization and occupancy optimization

This mind map illustrates the interconnected nature of GPU programming concepts, from measurement and analysis to implementation strategies and optimization techniques. Each branch represents a critical component of effective GPU kernel development for high-performance deep learning applications.