# Flash Attention: Fast and Memory-Efficient Attention

## Understanding GPU Memory Optimization

Akash Malhotra, PhD at LISN

November 11, 2025

# Outline

# What is a CUDA Kernel?

- A **CUDA kernel** is a function that runs on the GPU
- Written in CUDA C/C++ and executed by thousands of threads in parallel
- Each thread executes the same code but on different data (SIMT model)

## Key Characteristics

- **Small programs**: Designed for specific, focused operations
- **Massively parallel**: Can launch millions of threads
- **Kernel boundaries**: After execution, control returns to CPU

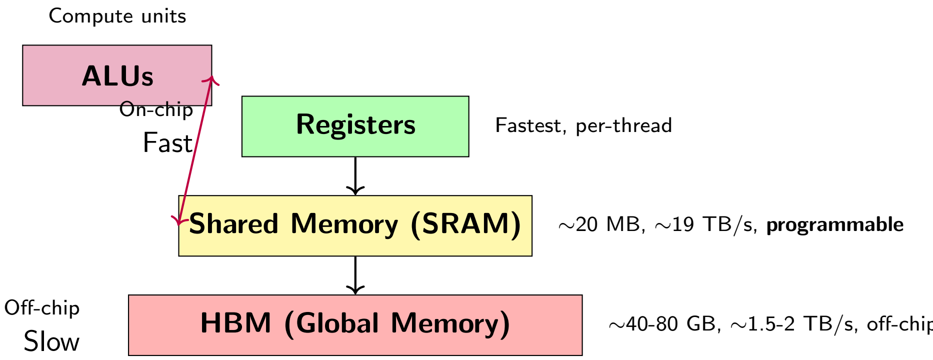
# Simple CUDA Kernel Example

```
// CUDA kernel definition
__global__ void vectorAdd(float* A, float* B, float* C, int
    N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] + B[idx];  // Each thread adds one
            element
    }
}

// Launch kernel from CPU (<<< >>> syntax)
vectorAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
```

Each thread computes one element of the result in parallel!

# GPU Memory Hierarchy




**Key Insight:** HBM is ∼10x slower than SRAM! When transferring data from HBM, ALUs sit idle waiting. Flash Attention keeps data in SRAM to keep ALUs busy.

*Note: L2 cache exists but is hardware-managed and transparent to programmers*

# Why is HBM Slower?

**Physical Reasons:**

1. **Size**: Larger memory $\rightarrow$ longer access time
2. **Location**: Off-chip $\rightarrow$ signal must travel further
3. **Latency**:
   - SRAM: $\sim$nanoseconds
   - HBM: $\sim$microseconds (1000x slower)

**The Problem**

When data is transferred HBM $\leftrightarrow$ SRAM, the ALUs (arithmetic units) sit **idle**, waiting for data.

This is called being **memory-bound**.

# Self-Attention Mechanism

**First:** Compute query, key, value matrices from input $X$:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

**Then:** Compute attention using $Q, K, V \in \mathbb{R}^{N \times d}$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V$$

## Step by Step

1. Compute $S = QK^T \in \mathbb{R}^{N \times N}$ (attention scores)
2. Compute $P = \text{softmax}(S) \in \mathbb{R}^{N \times N}$ (attention weights)
3. Compute $O = PV \in \mathbb{R}^{N \times d}$ (output)

**Memory Issue:** The attention matrices $S$ and $P$ are both $N \times N$ — for $N = 4096$, this is $\sim$67 MB each!

# Naive CUDA Implementation: Separate Kernels

**Kernels:** $Q = XW_Q, K = XW_K, V = XW_V$

↓

**Kernel:** Compute $S = QK^T$

↓

Write $S$ to HBM, then read back — $\mathcal{O}(N^2)$

↓

**Kernel:** Compute $P = \text{softmax}(S)$

↓

Write $P$ to HBM, then read back — $\mathcal{O}(N^2)$

↓

**Kernel:** Compute $O = PV$

**The "Swap Memory" Problem:** HBM acts like swap space — $S$ and $P$ are written to HBM, SRAM flushed, then read back for next kernel!

# Flash Attention: Key Ideas

**Main Goal**

**Reduce HBM access** by keeping computations in fast SRAM

1. **Kernel Fusion**: Combine all attention operations into a single kernel
2. **Tiling**: Divide $Q, K, V$ into small blocks that fit in SRAM
3. **Recomputation**: Use online softmax algorithm to avoid storing full attention matrix
4. **Streaming**: Process tiles incrementally, accumulating results

**Result:** Never materialize the full $N \times N$ attention matrix in HBM!

# Tiling Strategy: Matrix Multiplication Example

**Goal**

Compute $C = A \times B$ where $A, B$ are $4 \times 4$ matrices using $2 \times 2$ tiles

**Matrix A:**

$$\begin{array}{|c:c|} \hline A_{11} & A_{12} \\ \hdashline A_{21} & A_{22} \\ \hline \end{array}$$

**Matrix B:**

$$\begin{array}{|c:c|} \hline B_{11} & B_{12} \\ \hdashline B_{21} & B_{22} \\ \hline \end{array}$$

**Result C:**

$$\begin{array}{|c:c|} \hline C_{11} & C_{12} \\ \hdashline C_{21} & C_{22} \\ \hline \end{array}$$

**Computing tile $C_{11}$:**

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

**Key Point:** Each $2 \times 2$ tile fits in SRAM. Load tiles from HBM, compute in SRAM, accumulate!

# Flash Attention Algorithm

## One Fused Kernel with Tiling

**For each block $Q_i$:**

1. Load $Q_i$ into SRAM
2. Initialize output block $O_i = 0$, running statistics
3. **For each block $K_j, V_j$:**
   - Load $K_j, V_j$ into SRAM
   - Compute $S_{ij} = Q_i K_j^T$ (in SRAM)
   - Update online softmax statistics
   - Accumulate: $O_i \mathrel{+}= \text{softmax}(S_{ij}) V_j$
4. Write final $O_i$ to HBM

**Memory I/O:**

- Read $Q, K, V$ tiles once
- Write $O$ once
- **No intermediate writes!**

**Key Benefits:**

- Everything stays in SRAM
- ALUs stay busy
- Same $\mathcal{O}(N^2 d)$ FLOPs

# Naive vs Flash Attention: Side by Side

| **Aspect** | **Naive** | **Flash Attention** |
|---|---|---|
| # of Kernels | 3 separate | 1 fused |
| HBM Reads | $\mathcal{O}(N^2)$ | $\mathcal{O}(Nd)$ |
| HBM Writes | $\mathcal{O}(N^2)$ | $\mathcal{O}(Nd)$ |
| Store full $S, P$? | Yes | No |
| SRAM Usage | Flushed between kernels | Persistent |
| Bottleneck | Memory-bound | Compute-bound |
| Speedup | 1x (baseline) | **2-4x faster** |

### The Key Difference

Flash Attention does the **same amount of computation** but with **much less memory traffic**!

# The Online Softmax Trick

**Challenge:** Standard softmax requires all values before computing:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

**Solution:** Online/incremental softmax algorithm

- Compute softmax **incrementally** as new blocks arrive
- Maintain running max and sum of exponentials
- Update previous results when processing new blocks

This enables computing attention without storing the full $N \times N$ matrix!

*Details: "FlashAttention: Fast and Memory-Efficient Exact Attention" (Dao et al., 2022)*

# Results & Key Takeaways

**Performance:**

- **2-4x faster** than standard attention
- Greater speedup for longer sequences
- Memory: $\mathcal{O}(Nd)$ vs $\mathcal{O}(N^2)$
- Train with 4-8x longer contexts

**Impact:** Used in GPT-4, Claude, LLaMA, etc.

**Key Lessons:**

1. Memory hierarchy matters (SRAM vs HBM)
2. Memory-bound vs compute-bound
3. Kernel fusion + tiling keeps data in fast memory
4. Algorithmic innovation (online softmax)
5. Same math, better implementation

# Further Reading

- **Original Paper**: "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness"
  Dao, Fu, Ermon, Rudra, Ré (2022)
  `https://arxiv.org/abs/2205.14135`
- **FlashAttention-2**: Further optimizations
  Dao (2023)
  `https://arxiv.org/abs/2307.08691`
- **CUDA Programming Guide**:
  `https://docs.nvidia.com/cuda/`

**Questions?**