DSC204A: Scaleable Data Systems Winter 2025

Guest Lecture: Enabling Efficient ML Algorithms via Kernels

Instructor: Umesh Bellur *Scribe: Sijin Lyu, Zhoutianning Pan, Tianhao Zhou*

1 Introduction

This lecture explores how GPU compute models and kernels drive efficient machine learning algorithms. Understanding these models and kernels will give us the tools to drive full-stack innovation in ML. Areas of classic systems ideas from databases and operating systems that boost efficiency broadly align with ideas introduced in this class.

2 Projects Overview

Three projects are discussed, each leveraging GPU kernels to improve ML efficiency:

- **Flash Attention.** A memory-efficient method to compute attention, analogous to a database join.
- **Thunder NLA.** Focuses on faster language model inference using a mega kernel, treating the GPU as a distributed system with independent compute units that communicate.
- **Chipmunk.** Accelerates diffusion models by addressing kernel scarcity and leveraging GPU hardware understanding.

3 Motivation

Recent developments in machine learning are extremely impressive. In terms of human-level text and code generation and understanding, machine learning models, such as GPT-4, DeepSeek, and LLaMA, are performing remarkably well. In terms of high-quality image generation, image-generating models such as Mid-Journey, DALL-E, and Flux are producing high quality images. The breakthroughs in sciences, like protein folding by AlphaFold, which received a Nobel Prize, and groundbreaking and emerging DNA foundation models, hold transformative potential for healthcare and drug development.

3.1 Scaling Transformers Efficiently on GPUs

Model sizes have scaled substantially, going from hundreds of millions of parameters in 2018 to hundreds of billions or trillions today. This scale provides new capabilities, human-like chat, code writing, a complicated understanding of text, etc. Training these models depends upon large GPU compute clusters, such as OpenAI's Stargate and several Middle Eastern investments in GPU infrastructure. NVIDIA's meteoric revenue growth demonstrates the importance of GPUs to machine learning overall.

Efficiency is increasingly linked to machine learning model quality, as more efficient models enable longer training or larger models within the same hardware budget. The Transformer architecture, central to recent ML breakthroughs, relies on two core operations: **Attention**, which scales quadratically with sequence length $\big(O(N^2)\big)$, meaning longer text or larger images significantly increase compute demands; and **MLP** (Multi-Layer Perceptron), which scales quadratically with model width $\big(O(d^2)\big)$, where doubling the width quadruples the compute required.

Challenge. How do we make these primitives more efficient?

4 Research Approach

4.1 ML Perspective

They aim to improve algorithmic efficiency by optimizing big-O scaling and developing data-efficient training methods. Additionally, we will explore novel loss functions or architectures to enhance model performance.

4.2 Systems Perspective

They aim to make algorithms hardware-aware for a variety of hardware devices including GPUs, TPUs, and ASICs. The general goal will be to use algorithmic best practices with hardware optimizations, such as tiling, scheduling, kernel fusion, and roofline analysis to enhance the performance.

4.3 Intersection

Through the development of efficient, hardware-aware primitives, they will put away the inputs of both algorithmic optimizations and hardware optimizations. They will target open-source models in different domains, such as DNA modeling, EEG analysis, video analysis, and fMRI, where available datasets are open-source as well, to develop high-quality foundations of open-source models that the entire community can access and build upon.

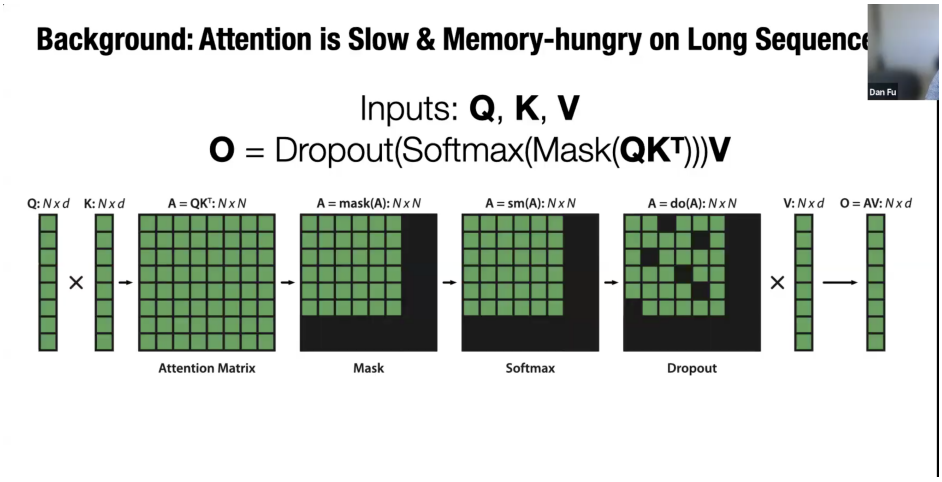
5 Project 1: Flash Attention

A fast, memory-efficient attention computation, analogous to a database join.

5.1 Attention Computation

- **Inputs.** Query (Q), Key (K), and Value (V) matrices, likened to join keys and row values in a relational database.
- **Process:**
  1. Compute the matrix product QK^T, yielding an $N \times N$ matrix (where N is sequence length; e.g., $N = 100{,}000$ for a long document, d is head dimension, e.g., 64 or 128).
  2. Apply optional masking.

3. Perform a row-wise Softmax.
4. Apply optional dropout.
5. Multiply the result by V to produce an $N \times d$ output matrix.



Problem in Frameworks. At every step, intermediate results are written to high-bandwidth memory (HBM), which mimics writing to disk in databases, creating a huge I/O bottleneck. This would be like materializing every intermediate for a join, resulting in a slow process and excessive memory consumption.

5.2 Flash Attention Solution

Attention is implemented within a single fused CUDA kernel, materializing only the final inputs and outputs to minimize intermediate memory usage. As with a block nested loop join in databases, block-wise tiling to partition our computation into tiles makes it possible to run the logic on a more manageable number of intermediate computations.

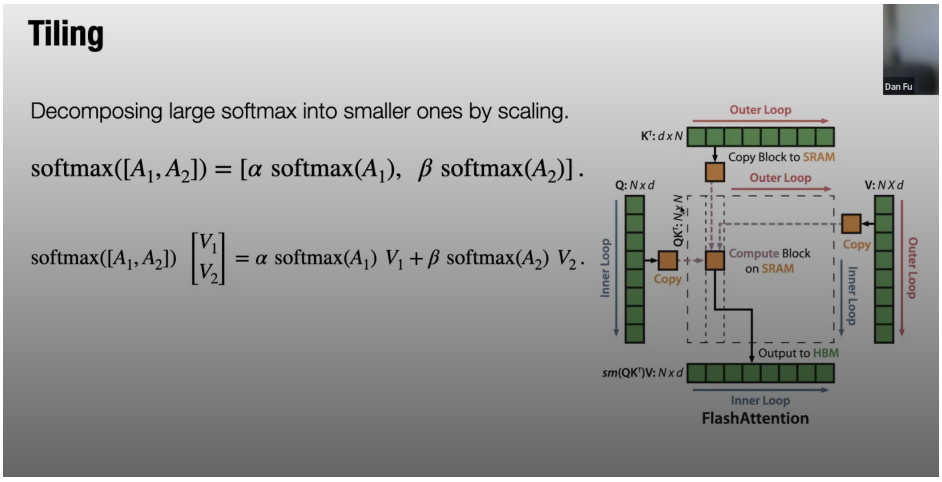
5.2.1 GPU Memory Hierarchy

- **HBM (High-Bandwidth Memory).** Throughput: 3.3–10 TB/s; large capacity but slower relative to compute speed.
- **SRAM (Shared Memory).** On Streaming Multiprocessors (SMs): throughput ≈ 20 TB/s; very fast but limited size.
- **Compute Units.**
  - Tensor Cores: specialized for matrix multiplications, achieving up to petaflops of throughput.
  - CUDA cores (FP32/FP64): general-purpose, but lower peak throughput.

Minimize HBM I/O and leverage Tensor Cores and shared memory for on-chip computations.

5.2.2 Implementation Outline

1. Load a block of Q and a block of K from HBM into SRAM .
2. Perform a block-wise matrix multiplication on Tensor Cores to compute partial QK^T.
3. Compute partial Softmax within the tile.
4. Update global scaling factors to merge blocks online .
5. Multiply the attention weights by the corresponding block of V in SRAM.
6. Accumulate the partial outputs in SRAM, then write the final $N \times d$ result back to HBM at the end.



5.2.3 Results on an A100 GPU

- **Speedup.** Achieves 2–4× speedup in attention computation compared to naïve implementations.
- **Memory Usage.** Scales linearly with sequence length N, instead of quadratically—dramatically reducing memory footprint.
- **Downstream Impact.**
  - 2–3.5× faster end-to-end training of GPT-2.
  - Enables longer context lengths (e.g., 32k or 100k tokens) rather than the previous 2k–4k token limit.

6 Project 2: Thunder MLA

6.1 Overview

Focuses on faster language model inference using mega kernels, treating the GPU as a distributed system with fine-grained control over compute units.

6.2 Problem: Language Model Decoding

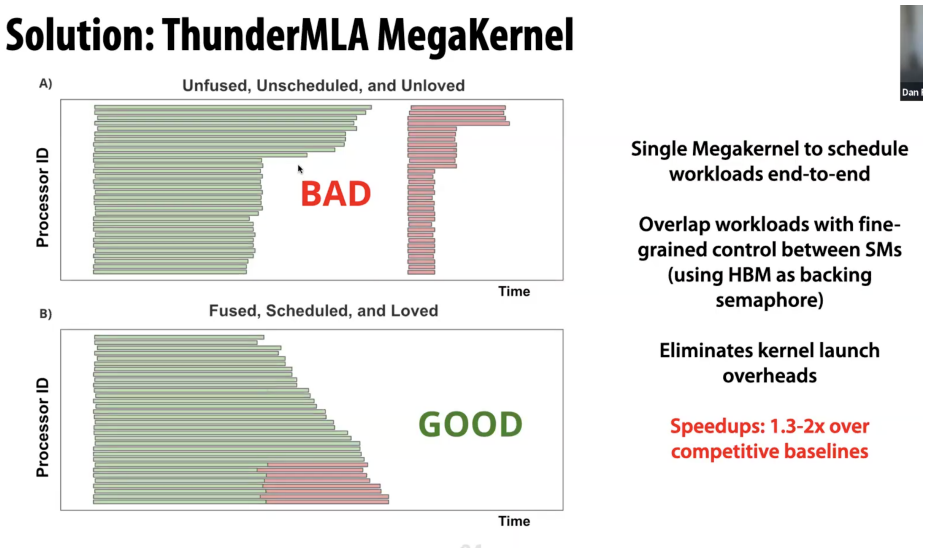
During language model inference like with ChatGPT, the model generates tokens one at a time with respect to a prompt: like completing a homework assignment. Each generated token (a query) is fed back into the model as input, processing a few previous tokens and the prompt, in a single "online" Softmax computation, allowing a new token to be generated. Each generated token output is just another query, limiting additional token generation while allowing all probable keys and values are indexed. The stepwise generation of tokens places a constraint on parallelism, consequently leading to underutilization of the parallel processing power of the GPU, particularly with the streaming multiprocessors or SMs, leading to less than optimal GPU utilization.

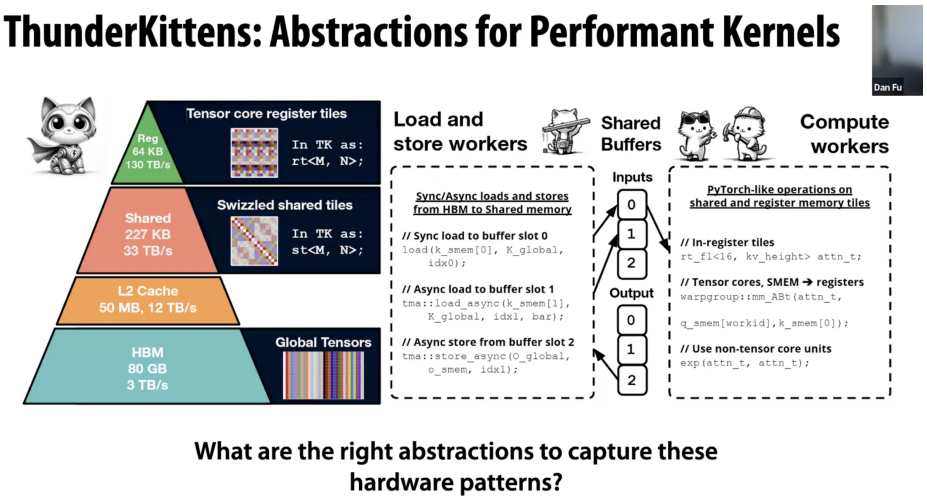
6.3 Initial Approach: Flash Decode

Flash Decode tackles language model inference by taking the prompt and splitting it into pieces that can be processed in parallel and then computing the keys and values at the same time. Flash Decode consists of two CUDA kernels: first a partial kernel, that computes the partial attention outputs, then a reduction kernel, to combine the partial outputs. The problem of using two kernels is that we can lose GPUS capability because of straggler effects where idle processors are waiting for the slowest partial to finish result in idle "white spaces" , and kernel launch overhead, where writing to memory, destroying the previous kernel, and launching the next kernel all require time to perform.

6.4 Thunder MLA Solution

- Fuse partial and reduction operations into a single "mega kernel," breaking NVIDIA's typical kernel programming paradigm.
- **Approach.**
  - Overlap operations by starting reductions as soon as each partial computation finishes, using fine-grained control over individual SMs.
  - Treat the GPU as a distributed system with independent workers (SMs) communicating via high-bandwidth memory (HBM).
- **Implementation.**
  1. Use a CUDA-based interpreter framework for scheduling tasks.
  2. Pre-compute durations of partial and reduction tasks based on input characteristics.
  3. Employ a simple bin-packing algorithm to assign tasks to SMs.
  4. Use semaphores in HBM to signal when a partial computation is complete, triggering the corresponding reduction step.
  5. Allocate shared memory once at kernel launch and dynamically reinterpret it for different tasks to minimize memory pressure.





7 Project 3: Chipmunk

Accelerates diffusion models by exploiting dynamic sparsity within the attention computation.

7.1 Problem: Diffusion Models

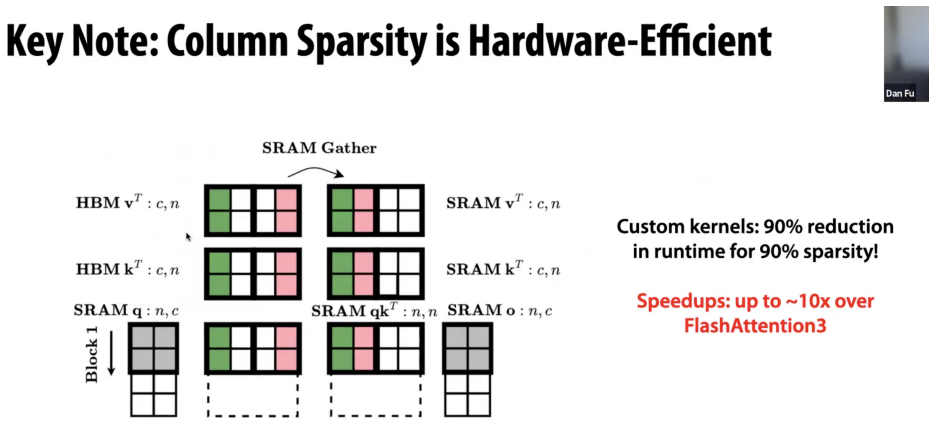
Diffusion models, such as *Flux*, generate high-quality images and videos by iteratively denoising a random noise vector using a Transformer. The process starts with a Gaussian noise latent vector X, where the Transformer predicts a change ΔX to reduce noise at each step, typically using Euler's method, and repeats for approximately 30–40 steps to produce the final sample. For example, a state-of-the-art video model like Runway takes around 17 minutes to generate a 5-second video on an H100 GPU, far from real-time. The challenge lies in the high computational cost, as each denoising step requires a full Transformer evaluation, including attention and MLP operations, repeated multiple times.

7.2 Chipmunk Algorithm

- **Core Idea.** Recompute only the most impactful attention columns (e.g., top 5–10% contributing to ΔX) and reuse the others from a cached state.
- **Heuristic.** At each denoising step:
  1. Use a lightweight scoring function to identify the subset of query/key/value columns with largest impact.
  2. Recompute attention only for that subset; reuse the cached attention for the remaining columns.
- **Result.** Near-identical outputs to dense attention while skipping ≈90–95% of computations.

7.3 Systems Implementation: Column Sparsity

- **Sparsity Pattern.** Column-based sparsity in the QK^T computation:
  - Mark "large-impact" columns (green) for recomputation.
  - Mark "low-impact" columns (orange) to be loaded from a cached small-value representation.
- **Workflow.**
  1. Load relevant columns from HBM into SRAM.
  2. Perform an SRAM gather to group the selected (green) columns contiguously—enabling efficient Tensor Core usage on a smaller dense matrix.
  3. Keep the low-impact (orange) columns in a compressed representation in shared memory or on-chip buffer.
  4. Compute a dense matrix multiply on the gathered columns using Tensor Cores (maintaining ¿90% of GPU flops).
  5. Merge results with cached low-impact columns to form the full attention matrix.
- **Benefit.** Leverages Tensor Cores for the sparse subset, achieving high arithmetic intensity and reducing redundant work.
- **Performance.** Achieves up to 90% sparsity in attention while still using dense matrix multiply kernels, yielding a $\approx 10\times$ speedup over Flash Attention 3 in the attention kernel.



7.4 Results

- **Video Generation Speedup.** Up to $\approx 3.7\times$ faster on a 5-second video.
- **Output Quality.** Sparse-generated videos are visually almost indistinguishable from dense-generated ones; differences only become apparent under close inspection.

8 Q&A

Question 1: Handling Dynamic Shapes

Question: How do you handle fusion when the input has dynamic shapes—for example, variable sequence lengths? Won't that complicate things?

Answer: You simply treat a variable-length sequence as a loop in CUDA. In other words, you change the loop bounds to match the actual sequence length. If the sequence size doesn't divide evenly into your chosen thread-block dimensions, you handle the "leftover" elements by doing a smaller computation or padding with zeros. In practice, supporting different sequence lengths is straightforward: you just adjust the loop index and take care of any partial blocks at the end.

Question 2: Generalizing Fusion

Question: Is it possible to make this fusion technique more abstract, so that a compiler could automatically generate a fused kernel (not just for attention but for other operations)?

Answer: That turns out to be tricky. Although fusing simpler chains of operations (e.g., three matrix multiplies) has been automated for years, including Softmax in the middle introduces algebraic nuances that current compilers don't reliably handle. Even though the basic mathematical identities are known, attention implementations keep evolving, so you often have to re-derive the fusion logic anyway. In short, we'd expect compilers to automate this by now, but in practice they haven't caught up—figuring out exactly which algebraic transformations you need for attention fusion still requires manual effort.

Question 3: How about heterogeneous GPU architectures—say AMD GPUs or other accelerators? What impact will those have, and can these fusion techniques still apply?

Answer: We are actively exploring how to map our existing framework (e.g., Thunder MLA and Chipmunk) onto non-NVIDIA hardware. The same fine-grained control strategies can often be ported to AMD GPUs, TPUs, or even specialized ASICs (for example, Habana Gaudi or Graphcore IPU). In many cases, you can implement "flash attention" on TPU, although TPUs offer less low-level programmability than CUDA. Similarly, some ASICs (like those from Habana or other inference-oriented vendors) can execute fused kernels with their own low-level APIs. Ultimately, the core ideas—tiling, kernel fusion, and exploiting on-chip memory—translate across devices, but you must adapt to each platform's specific programming model and degree of hardware control.