Hardware-Aware Performance Optimizations

Systems for Machine Learning

Chapter 4

https://www.cse.iitb.ac.in/~mythili/sysml/

Mapping an ML Model to Hardware

- How to write code that uses hardware efficiently?
- **Compute placement**: how to map operations to hardware processing elements?
- **Memory management**: where to place data in the memory hierarchy, and what memory layout is optimal?
- **Data movement**: how to move data efficiently between memory and compute elements?

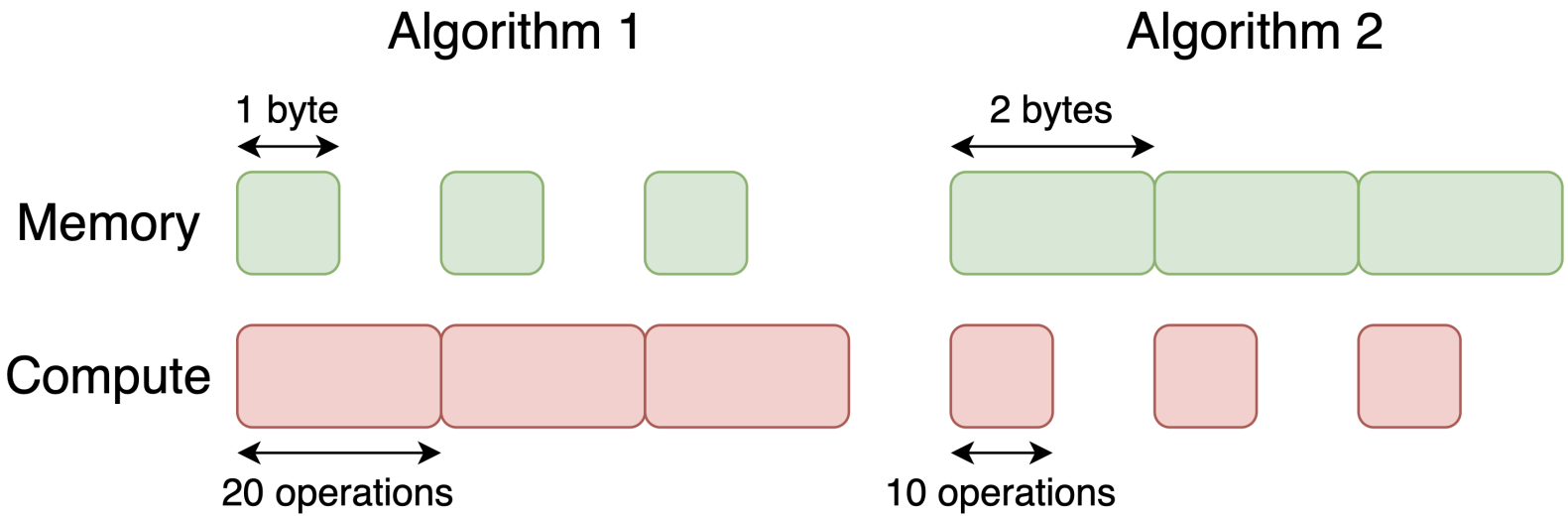
Understanding Performance

Compute vs. Memory

- Time taken by operation depends on time to fetch operands from memory, and time for actual compute
- If only one task can be done at a time, $T_{\text{total}} = T_{\text{compute}} + T_{\text{memory}}$
- If perfect overlap, $T_{\text{total}} = \max(T_{\text{compute}}, T_{\text{memory}})$
- If $T_{\text{compute}} > T_{\text{memory}}$, operation is **compute-bound**
- If $T_{\text{memory}} > T_{\text{compute}}$, operation is **memory-bound**

Compute vs. Memory

- Example: GPU performs 1000 operations per second, memory bandwidth 100 bytes per second
- Algorithm 1: 20 ops/byte, compute-bound
- Algorithm 2: 5 ops/byte, memory-bound



Arithmetic Intensity

- **Arithmetic Intensity** $= \frac{\#\text{ops}}{\#\text{bytes accessed}}$
- Property of the operation, not of the hardware
- For an operation to be compute bound:

$$T_{\text{compute}} > T_{\text{memory}}$$

$$\frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}} > \frac{\text{Communication Bytes}}{\text{Bandwidth Bytes/s}}$$

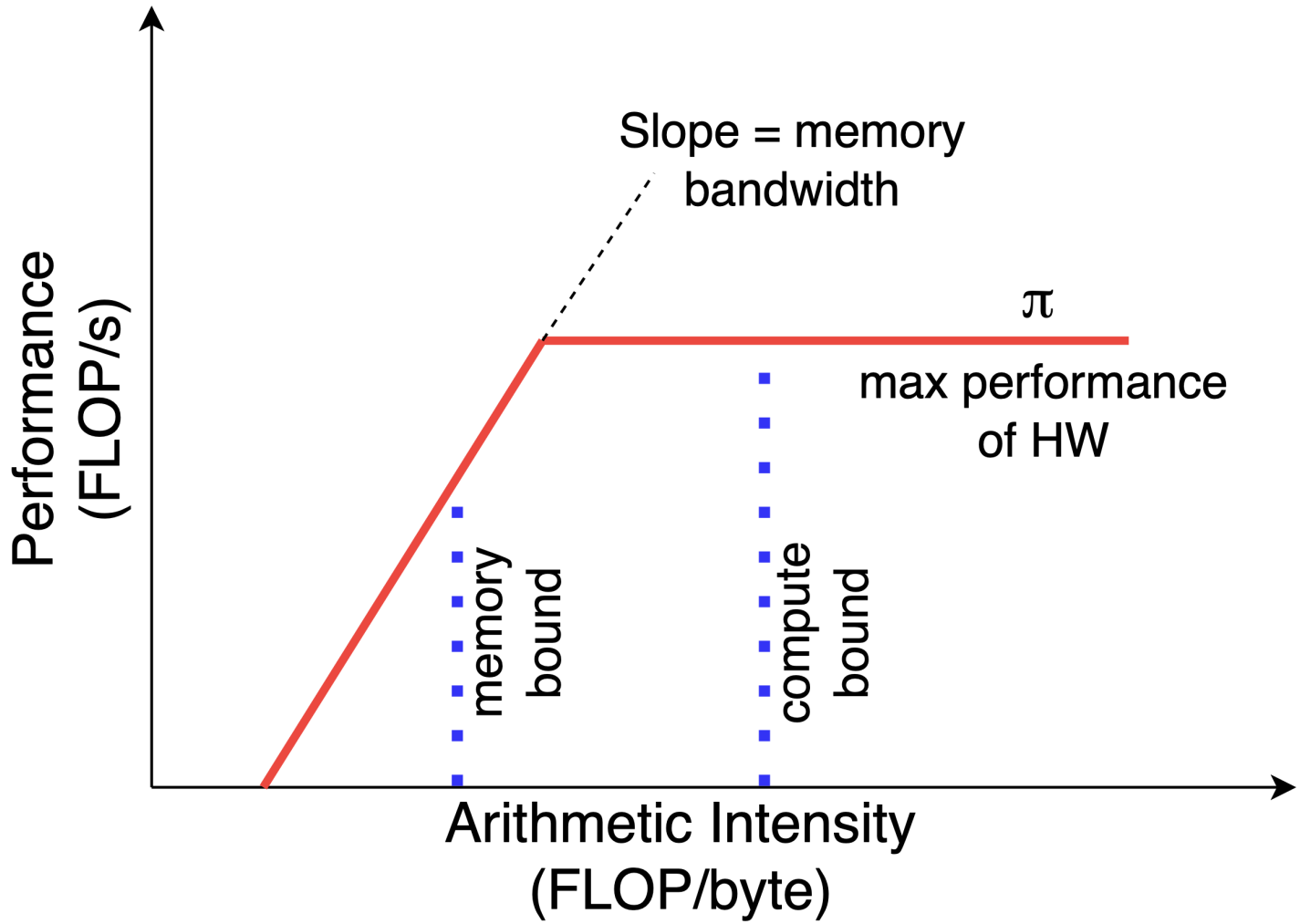
$$\frac{\text{Computation FLOPs}}{\text{Communication Bytes}} > \frac{\text{Accelerator FLOPs/s}}{\text{Bandwidth Bytes/s}}$$

$$\textbf{Intensity(Operation)} > \textbf{Intensity(Accelerator)}$$

Arithmetic Intensity: Examples

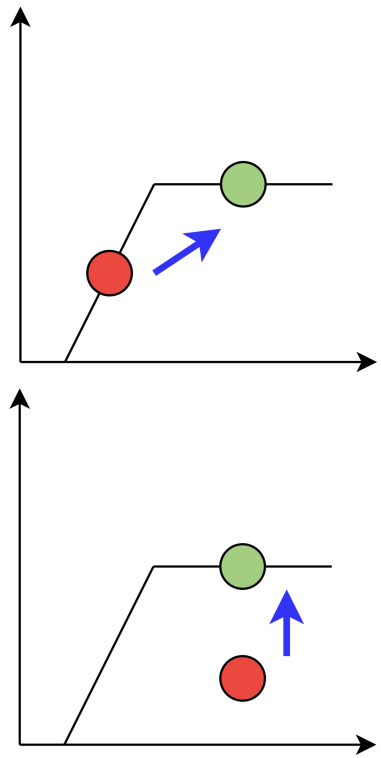
- **Matrix multiplication**: ops is $O(N^3)$, bytes accessed is $O(N^2)$, so arithmetic intensity is $O(N)$
- Matrix multiplication compute bound for large N, memory bound for small matrices
- **RELU**: single comparison for 4 bytes of data fetched, so arithmetic intensity is approx 0.25 ops/byte
- **Vector addition**: N adds for $2N$ data fetched
- Element-wise operations typically memory bound

Roofline Plot



Performance Optimization Techniques

- Which technique to use depends on where we start on roofline
- For **memory-bound** operations, increase the arithmetic intensity, e.g., tiling, kernel fusion, caching
- For **compute-bound** operations, optimize compute usage, e.g., batching, streams

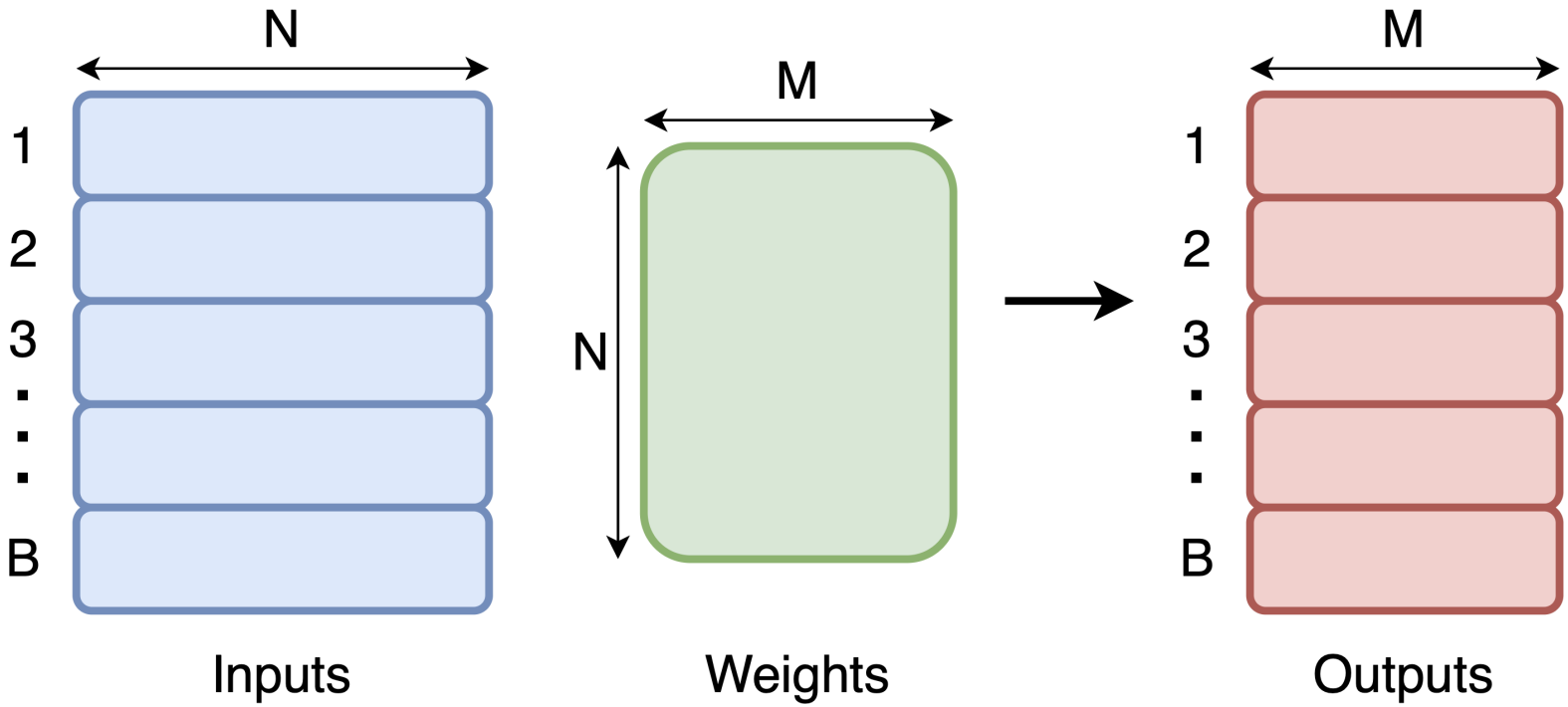


Performance Optimization Techniques

Batching

- **Without batching**: $\underbrace{[1 \times N]}_{\text{input}} \times \underbrace{[N \times M]}_{\text{weights}} = \underbrace{[1 \times M]}_{\text{output}}$
- **With batching**: $\underbrace{[B \times N]}_{\text{batched input}} \times \underbrace{[N \times M]}_{\text{weights}} = \underbrace{[B \times M]}_{\text{batched output}}$
- Reuse of weight matrices across inputs
- GPU has B times as much independent work
- Host-GPU data transfer, kernel launch amortized
- Matrix-matrix products can use GPU's tensor cores, whereas matrix-vector products cannot

Batching



Data Reuse

- Single output element in matmul requires entire row and column of input matrices to be fetched from memory
- Increase data reuse via **GPU caches**, e.g., keep one matrix stationary in shared memory
- Example: pin filter weights of CNN in shared memory, stream activations
- For $K \times K$ filter, reduce number of HBM accesses from $H_{out} \cdot W_{out} \cdot K^2$ (one fetch per output) to only K^2

Kernel Fusion

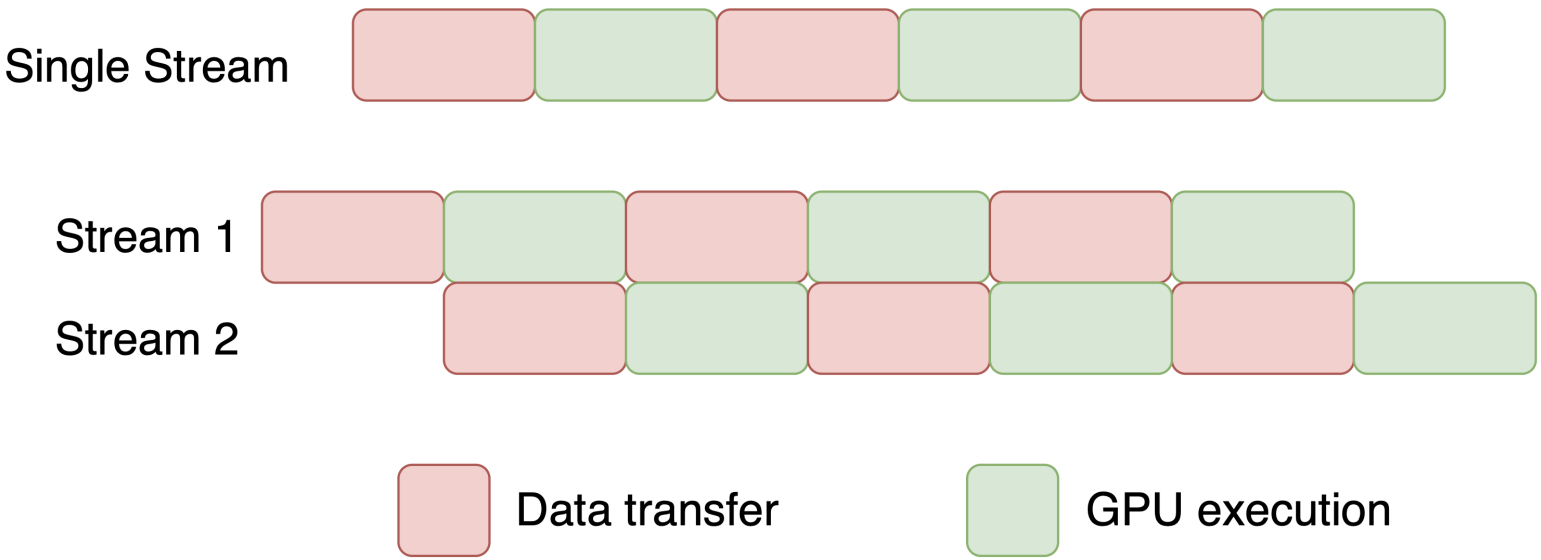
- Writing separate kernels for different operations requires more GPU memory reads and writes
- Consider linear layer followed by ReLU
  $Z = XW, \quad A = \text{ReLU}(Z)$
- If separate kernels, Z written back, read from HBM
- **Kernel fusion** combines both operations into single kernel, so intermediate Z never in slow memory

Improving GPU Compute Utilization

- Submit enough independent parallel work to GPU, via careful configuration of threads and blocks
- Optimize resource usage (e.g., shared memory, registers) to improve SM occupancy
- Optimize memory access patterns to avoid memory stalls
- Avoid warp divergence, wave quantization, etc.

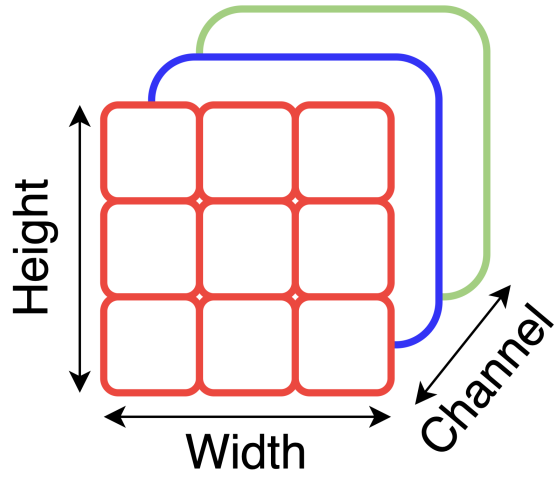
Improving GPU Compute Utilization

- Overlap host-device communication with compute, e.g., using asynchronous data transfers and **streams**
- Overlap network communication with compute, e.g., using prefetching



Memory Layout

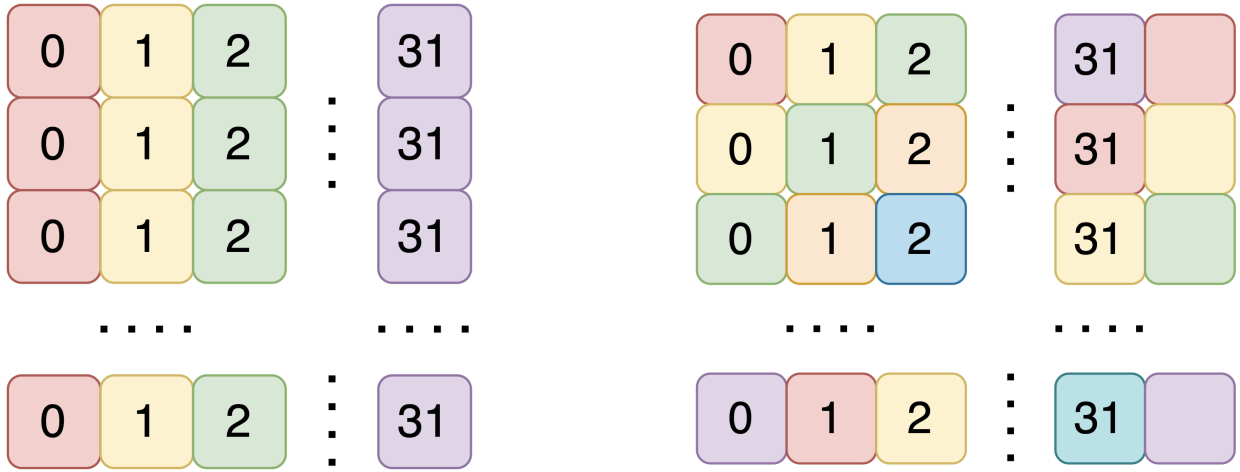
- Row-major layout or **NHWC**: image stored row-wise, with all channels together
- Channel-major layout or **NCHW**: image stored channel-wise, with rows/columns of channel together
- Memory layout impacts performance via coalesced memory accesses, so optimal layout depends on memory access pattern of computation



Shared Memory Bank Conflicts

- Shared memory divided into **32 banks** of 4 bytes each
- Each bank services one memory request per clock cycle, multiple requests serialized
- **Bank conflict** (two or more threads within a warp access different addresses in same bank) reduces throughput
- Design memory access patterns of threads in warp to avoid bank conflicts where possible

Shared Memory Bank Conflicts: Padding



(a) Without padding (b) With padding

Figure: Layout of the array on shared memory, different colours show different banks, and the numbers inside boxes denote warp numbers accessing the array element. Without padding, threads of the same warp access the same coloured bank, while with padding, threads of the same warp land on different banks.

Tiling

Tiling

- Breaking a large computation over large data into smaller sub-computations over small blocks of data, called **tiles**
- Fetch tile into **fast memory**, perform every computation that can be done using that data, and only then move on to the next tile
- Increases data reuse, reduces slow global memory accesses, increases compute efficiency

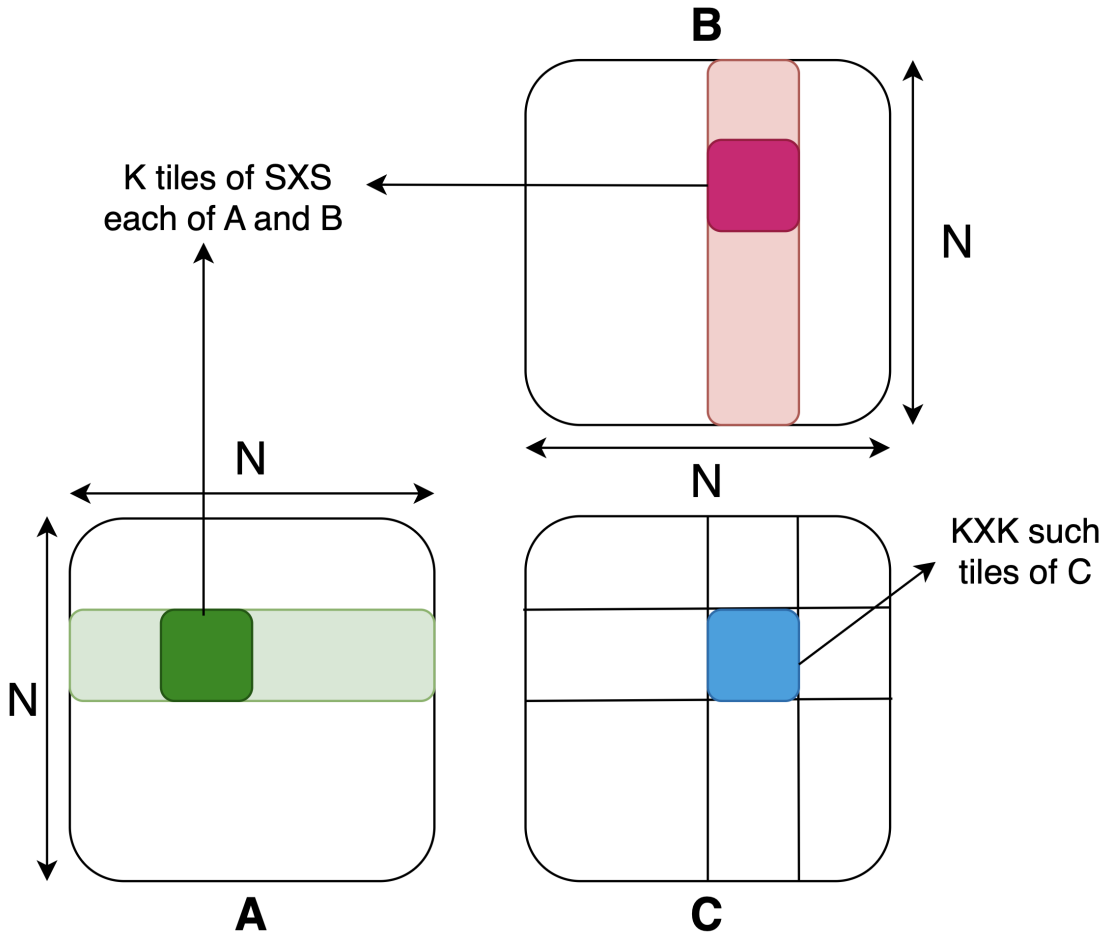
Matrix Multiplication without Tiling

- Multiplying two $N \times N$ matrices A and B to produce C, $C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$
- **#ops** $= N^2$ elements $\times N$ multiply-add ops $= N^3$
- **#memory accesses** to fetch A and B once from host to GPU $= 2 \times N^2$
- **#memory accesses** to fetch row of A and col of B for each elem of $C = N^2 \times 2 \times N = 2N^3$

Restructuring the Computation into Tiles

- Divide each matrix into tiles of size $S \times S$, there are $K = \frac{N}{S}$ tiles along each dimension
- C divided into a $K \times K$ grid of tiles
- Each tile of C computed by multiplying K tiles of A and B together, and adding partial results
- Fetch a tile each of A and B to shared memory, accumulate partial sum for tile of C, before moving to next tiles

Restructuring the Computation into Tiles



Restructuring the Computation into Tiles

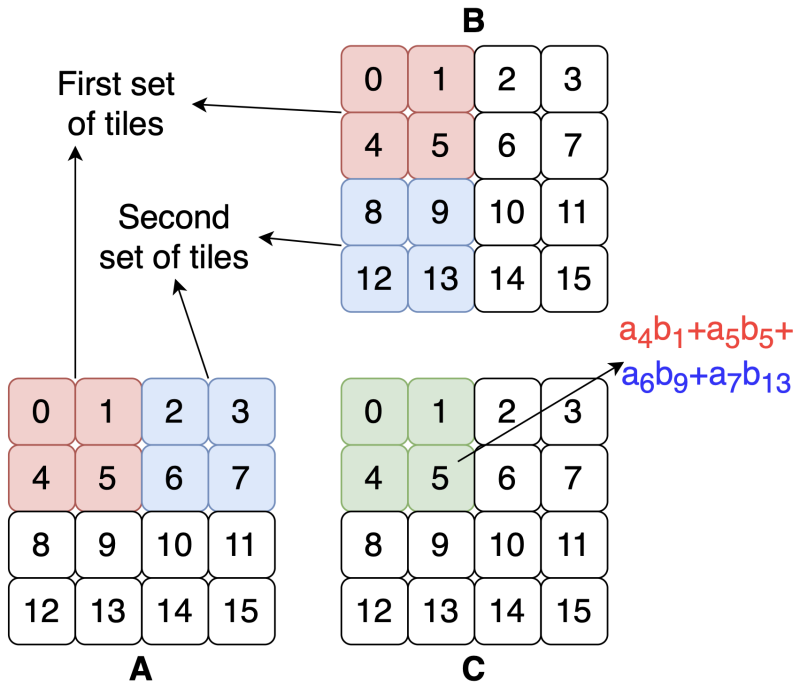


Figure: A 4×4 matrix multiply tiled into 2×2 blocks. A tile of C is accumulated from a small number of tile-by-tile products of A and B, rather than from full rows and columns.

Counting Operations and Memory Accesses With Tiling

- $K \times K$ tiles in C, and each requires accumulating K products of two $S \times S$ tiles
- **#ops** $= K^2 \times S^3 \times K = N^3$ (same as before)
- For each $K \times K$ tile of C, fetch K tiles of A and K tiles of B, each of S^2 elements
- **#memory accesses** $= K^2 \times 2 \times S^2 \times K = \frac{2N^3}{S}$
- Tiling has reduced the number of memory accesses by exactly a factor of S

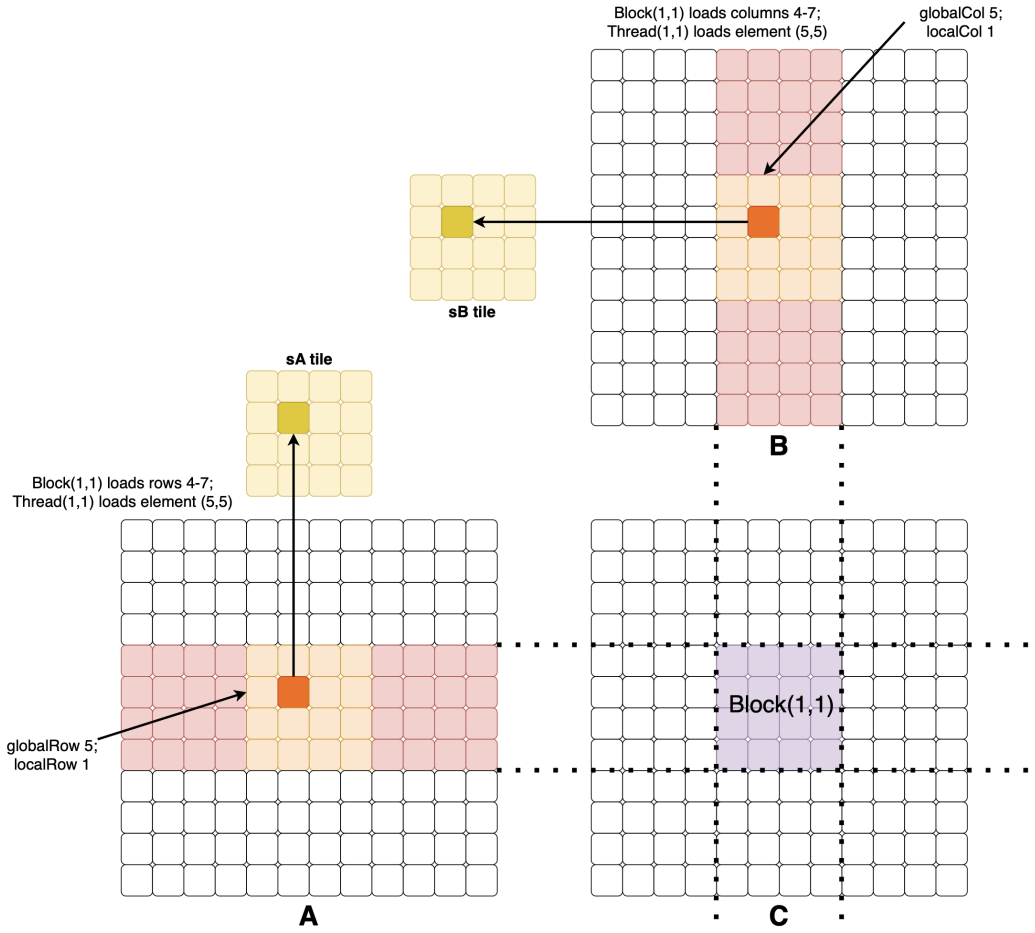
Choosing the Tile Size

- Ideally, make tile size S as large as possible, since bigger tiles mean more reuse
- **Cache capacity constraint**: tiles of A, B and the partial tile of C must all fit simultaneously in fast memory
- **SM occupancy constraint**: too large tiles may mean fewer blocks per SM and lower SM occupancy
- Maximize tile size subject to above constraints

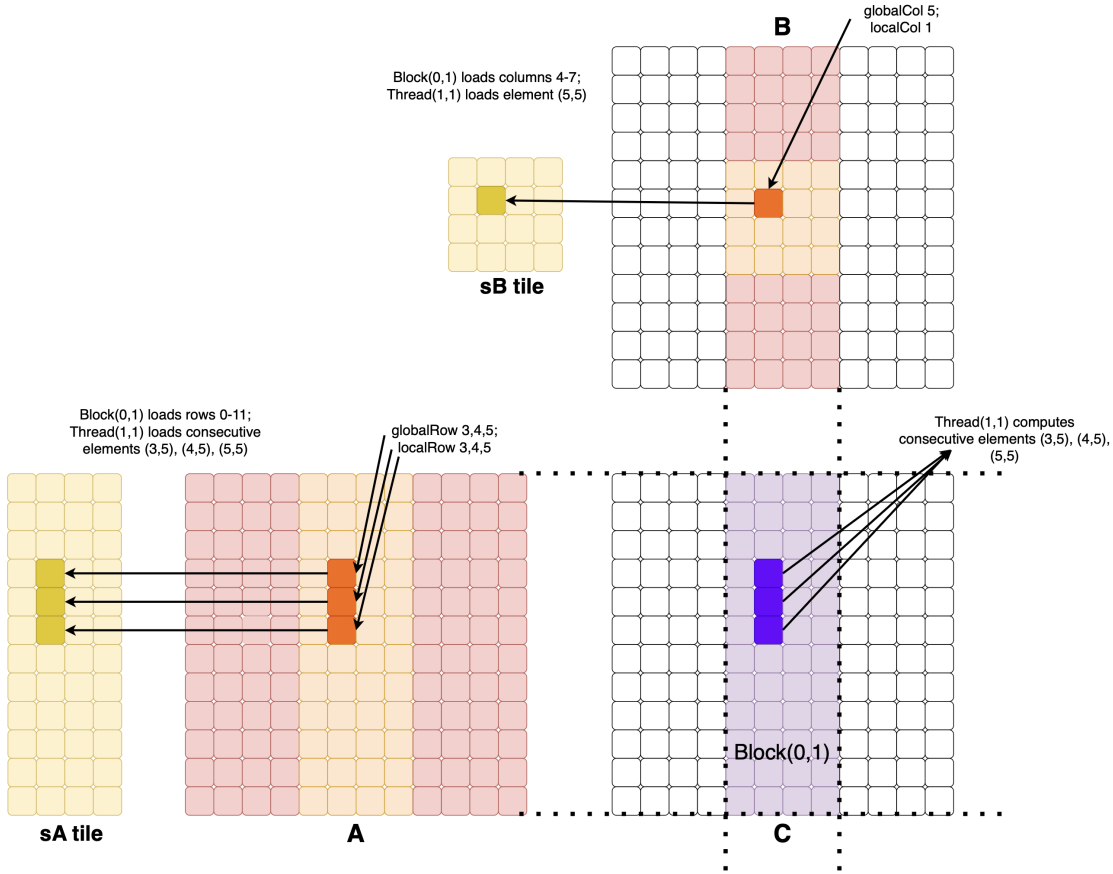
Tiling in CUDA

- Many ways to map tiling to threads in CUDA
- Example: launch a $K \times K$ grid of thread blocks, one for each tile of C, and every thread in block computes one element of the output C
- Threads in block cooperatively fetch tiles of A and B into shared memory, synchronize, accumulate output elements of C in parallel, and move on to next set of tiles
- Other variants possible too: threads can compute more than one element of C to reuse fetched inputs

Example: One Thread per Output Element



Example: Column-Contiguous Tiling



Flash Attention

Attention: Recap

- Attention: $S = QK^T, \quad P = \text{softmax}(S), \quad O = PV$
  - N is the sequence length, d is the head dimension
  - Q, K, V, O have shape $N \times d$
  - S and P have shape $N \times N$
- Complexity of attention grows quadratically with sequence length N
- FlashAttention designed to reduce HBM memory traffic, improve performance
- Causal mask ignored throughout the discussion

Standard Attention

Algorithm 1: Standard Attention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^T$, write $\mathbf{S}$ to HBM.

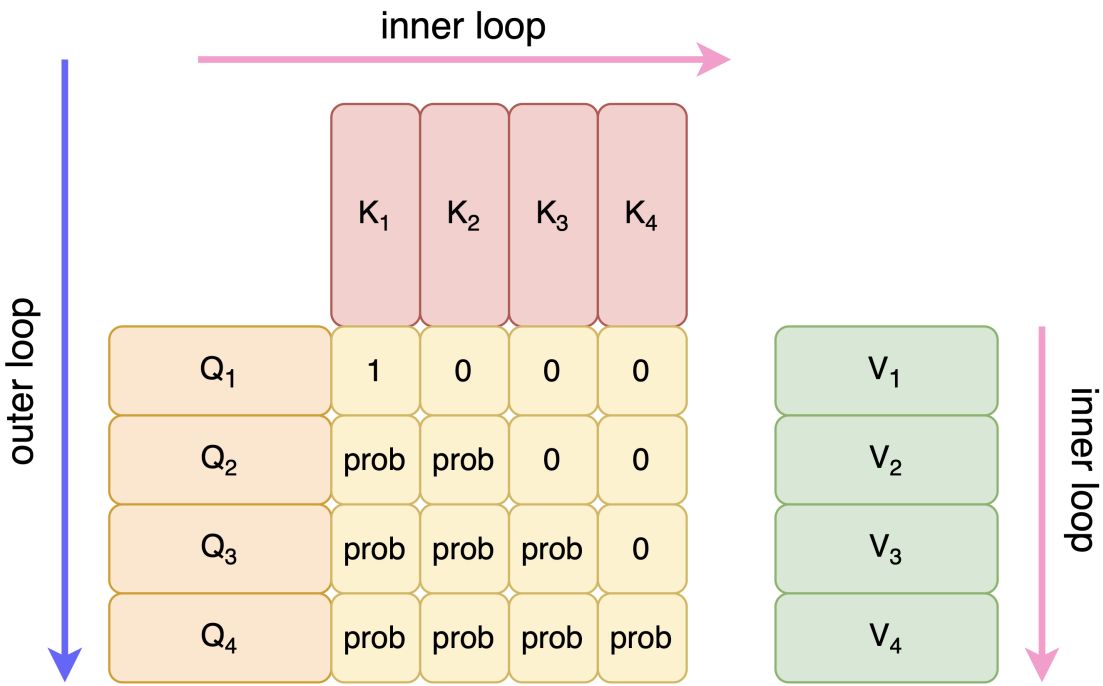
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.

3: Load $\mathbf{P}$ and $\mathbf{V}$ from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.

4: **return** $\mathbf{O}$.

Memory Accesses in Standard Attention

- One possible way of implementing standard attention
- Total number of memory accesses is $N \times \mathcal{O}(Nd) = \mathcal{O}(N^2 d)$



Why FlashAttention?

- Tiling can be applied to matmuls inside attention: $\mathbf{S} = \mathbf{QK}^T$ and $\mathbf{O} = \mathbf{PV}$
- Obstacle is the softmax sandwiched between them
- Computing even one entry of P_{ij} requires entire row S_i
- Problem solved by online softmax

The Softmax Obstacle

- If we have three raw values X, Y, Z, the normalized value of Y is $P(Y) = \frac{Y}{X+Y+Z}$
- If in a streaming computation, we have only seen X and Y, the best estimate is $P(Y) = \frac{Y}{X+Y}$
- Once Z becomes available, we must correct the denominator, $P(Y) = P^{\text{old}}(Y)\frac{X+Y}{X+Y+Z}$
- Relevant to attention, because columns of K are seen one by one

Online Softmax

- Row-wise normalization constant $L_i = \sum_j e^{Q_i K_j^T}$
- Attention output $O_i = \sum_j P_{ij} V_j = \frac{\sum_j e^{Q_i K_j^T} V_j}{L_i}$
- Online softmax maintains running estimates of L_i and O_i and updates incrementally
  - $L_i^{\text{new}} = L_i^{\text{old}} + e^{Q_i K_j^T}$
  - $O_i^{\text{new}} = \frac{O_i^{\text{old}} L_i^{\text{old}} + e^{Q_i K_j^T} V_j}{L_i^{\text{new}}}$

Tracking a Running Maximum

- $M_i = \max_j \left(Q_i K_j^T\right)$ subtracted for numerical stability
- $P_{ij} = \frac{e^{Q_i K_j^T - M_i}}{L_i}$, where $L_i = \sum_j e^{Q_i K_j^T - M_i}$
- With online softmax:
  - $M_i^{\text{new}} = \max\left(M_i^{\text{old}}, Q_i K_j^T\right)$
  - $L_i^{\text{new}} = L_i^{\text{old}}\, e^{M_i^{\text{old}} - M_i^{\text{new}}} + e^{Q_i K_j^T - M_i^{\text{new}}}$
  - $O_i^{\text{new}} = \frac{O_i^{\text{old}} L_i^{\text{old}}\, e^{M_i^{\text{old}} - M_i^{\text{new}}} + e^{Q_i K_j^T - M_i^{\text{new}}} V_j}{L_i^{\text{new}}}$

FlashAttention

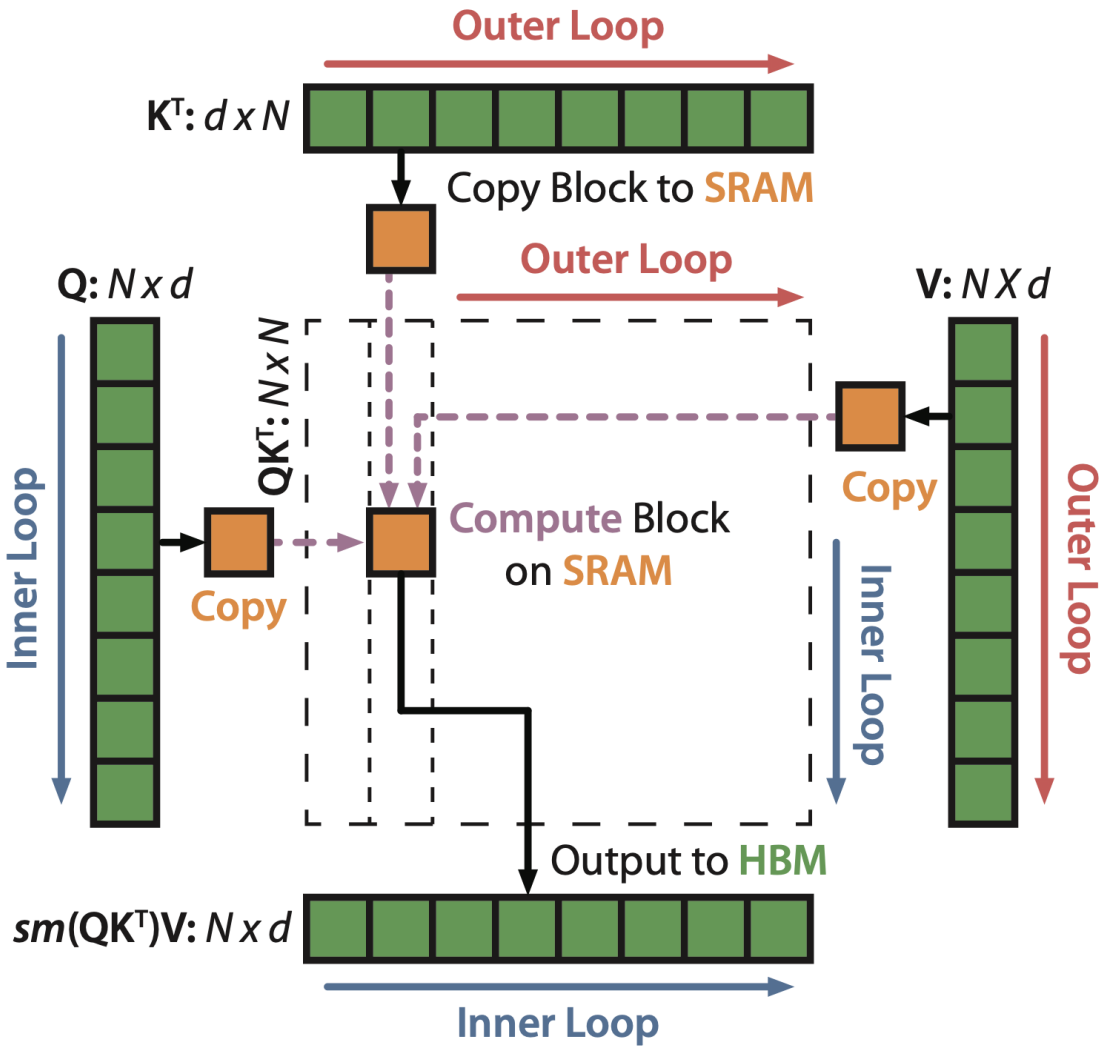


Image Credit: https://arxiv.org/pdf/2205.14135

Memory Accesses in FlashAttention

- Outer loop runs Nd/M times
- In the inner loop, for Q, $\mathcal{O}(Nd)$ memory accesses
- Total memory access cost of FlashAttention: $\frac{Nd}{M} \times \mathcal{O}(Nd) = \mathcal{O}\left(\frac{N^2d^2}{M}\right)$
- Ratio with standard attention is $\mathcal{O}\left(\frac{d}{M}\right)$
- In practice, $d \ll M$

Backward Pass of FlashAttention

- Given dO, we must produce dQ, dK, and dV
- Rather than storing S and P from the forward pass, recomputes tile-by-tile on-chip
- Save small extra information per query row from the forward pass, rather than the full S and P matrices
- Recomputation faster because of reduced HBM traffic

Later Versions of FlashAttention

- FlashAttention-2 swaps the loop order (outer loop over query, inner loop over key/value), defers division by ℓ_i to after the outer loop completes
- FlashAttention-3 targets the Hopper GPU generation
  - Assigns separate warps to fetching tiles from HBM and computing on tiles already in SRAM
  - Exploits Hopper's low-precision FP8 tensor cores

Model Optimizations

Pruning

- Pruning eliminates low magnitude weights
- Structured pruning: remove entire structural units of the model, e.g., neurons, layers, filters
- Unstructured pruning: remove individual weights
- Dynamic pruning: based on the input received

Pruning

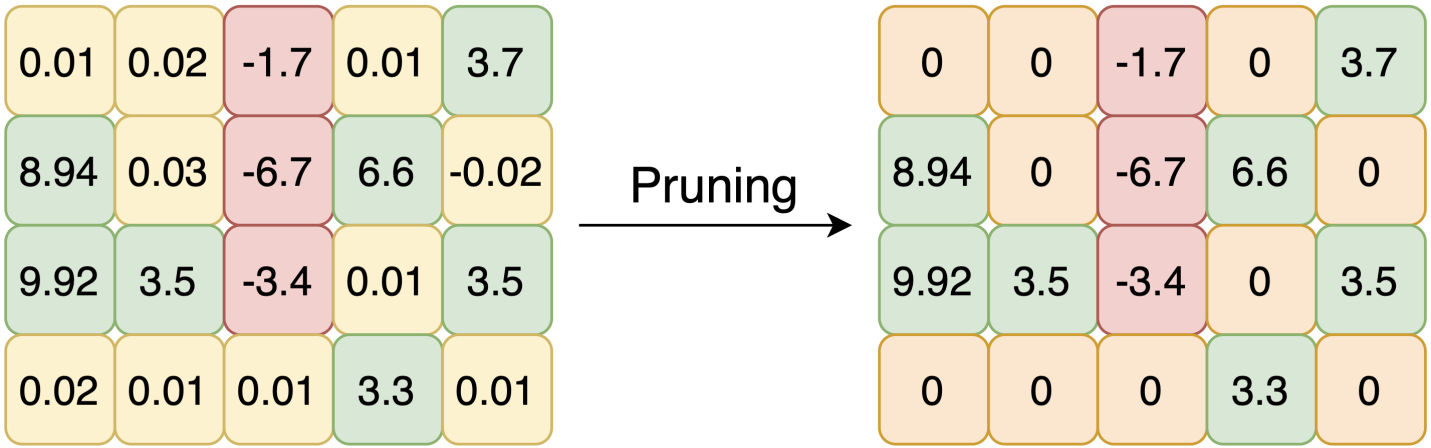
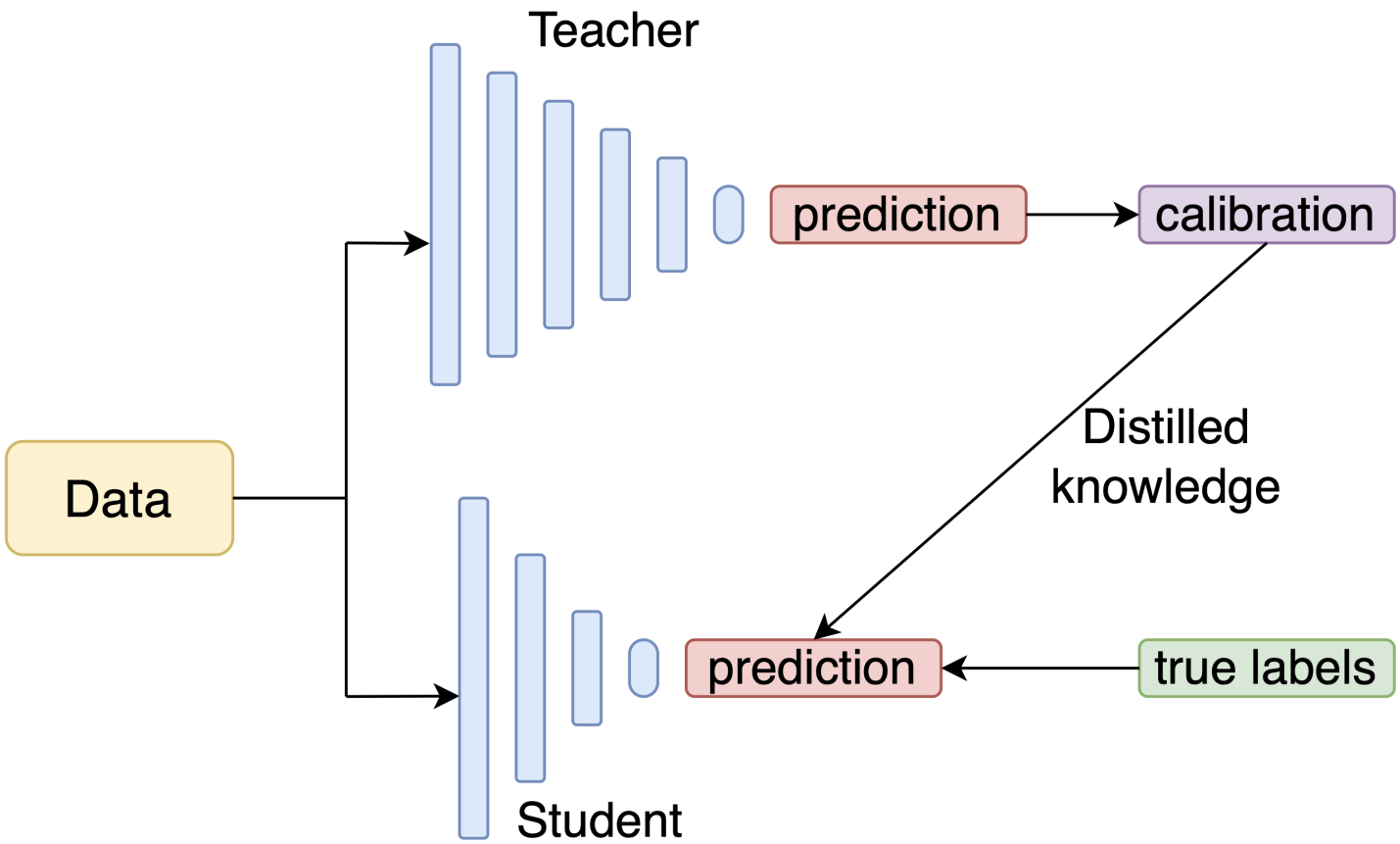


Figure: Unstructured Pruning

Knowledge Distillation

- Train a smaller "student" model to reproduce the behaviour of larger "teacher" model
- Teacher's output logits passed to student as soft labels
- Two objectives
  - Distillation loss measures difference between distributions of teacher and student
  - Conventional loss based on ground truth

Knowledge Distillation



Low-Rank Matrix Factorization

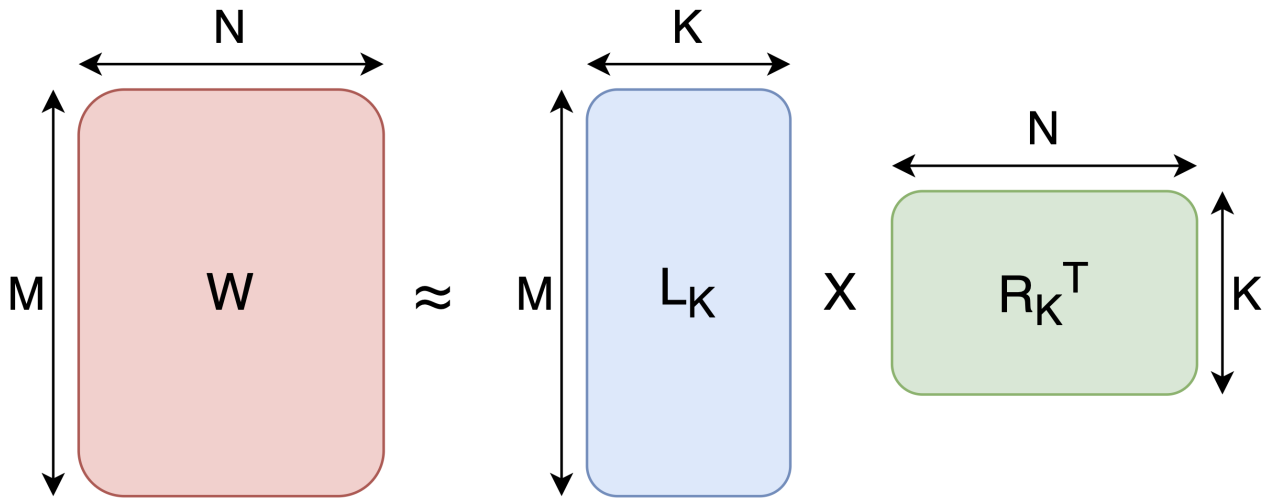
- Storing and multiplying weight matrix W of dimension $M \times N$ costs $O(MN)$ in both memory and computation
- Low-rank factorization:

$$W \approx L_K \times R_K^T,$$

- L_K has dimension $M \times K$, R_K^T has dimension $K \times N$
- K is much smaller than either M or N
- Storing and multiplying by these two smaller matrices costs only $O(MK + KN)$

Low-Rank Matrix Factorization

- Singular value decomposition used, top K singular values chosen
- Choosing K is a trade-off between accuracy and compression



Quantization

- Reduce numerical precision to trade-off accuracy in exchange for lower memory usage and faster arithmetic

Format	Total Bits	Sign	Exponent	Mantissa	Typical Use
FP32 (Single Precision)	32	1	8	23	Training (high precision)
FP16 (Half Precision)	16	1	5	10	Inference / mixed-precision training
BF16 (BFloat16)	16	1	8	7	Training (robust to large/small values)
TF32	19	1	8	10	Internal accelerator computation
FP8	8	1	4/5	3/2	Aggressive training/inference
INT8	8	–	–	–	Edge / resource-constrained inference

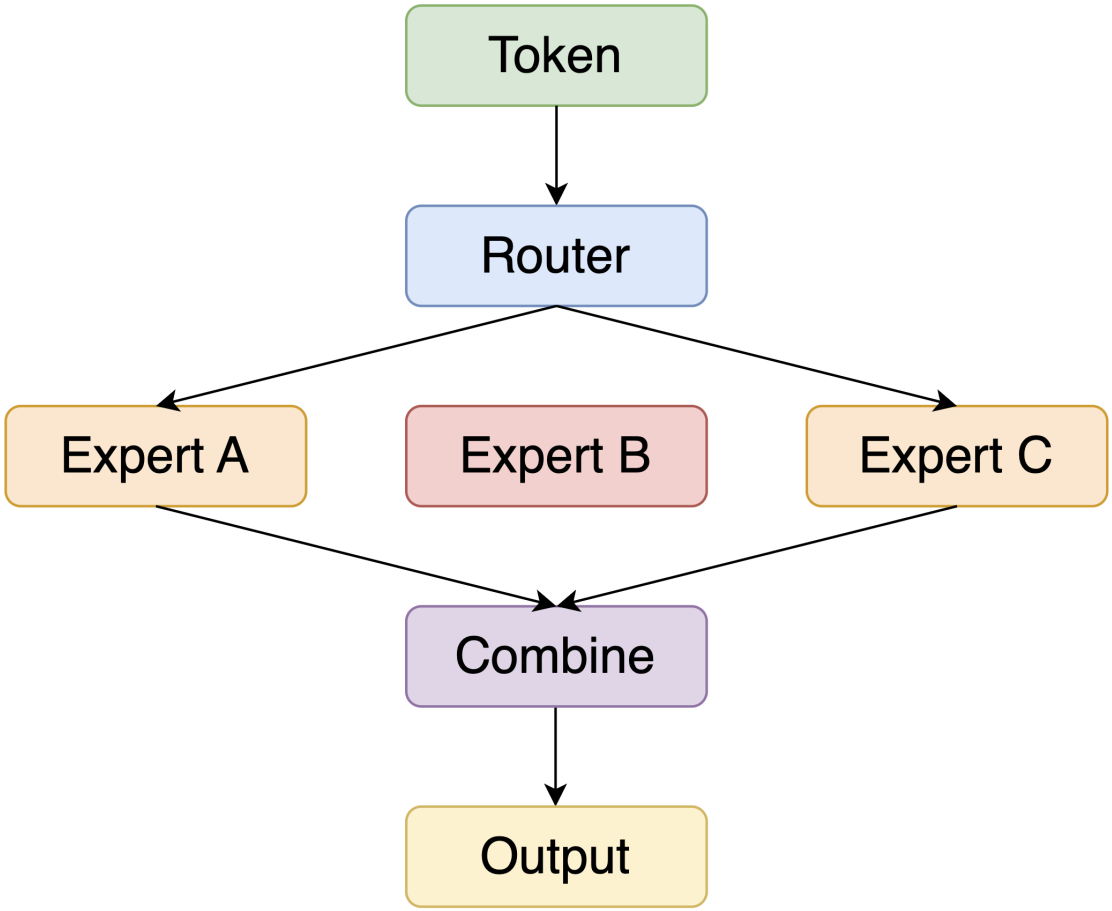
Quantization

- Post-training quantization: convert model weights after training to lower precision for inference
- Quantization-aware training: simulate effect of reduced precision throughout the training process itself
- Mixed precision training: forward/backward pass in lower precision, master copy of weights accumulated in high precision
- Calibration required to fit in smaller range of the new format

Mixture-of-Experts

- Single dense FFN replaced by smaller expert FFNs
- Small gating network (router) to decide which (one or top few) experts to send a token to
- Only fraction of parameters activated for each input
- Data-dependent decision of expert, may introduce imbalance across experts
- Routing inputs to experts and combining outputs introduces communication overheads

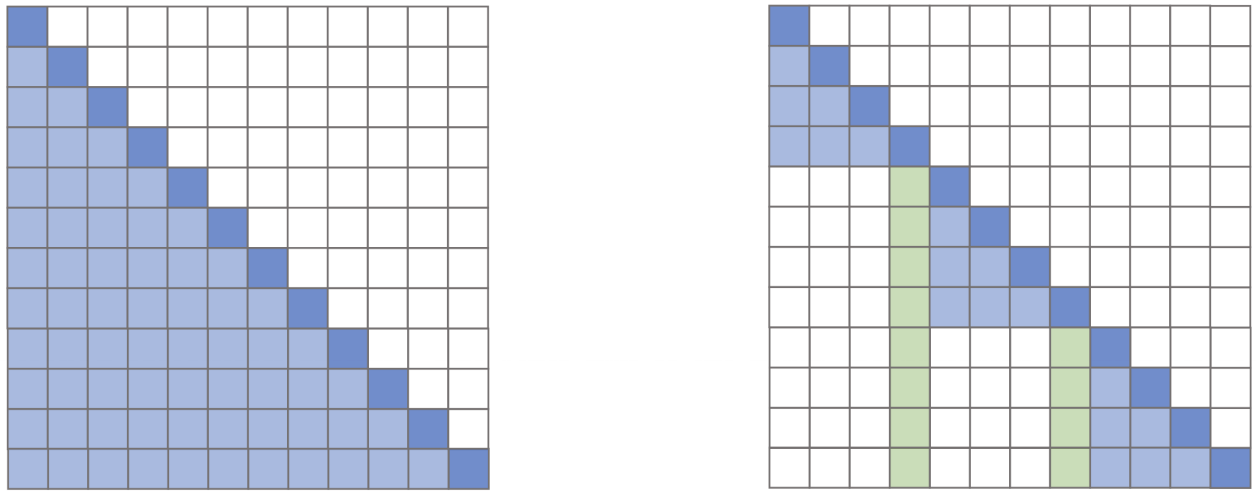
Mixture-of-Experts



Efficient Attention and Efficient Transformers

- Efficient attention variants and transformer architectures to avoid quadratic scaling issues of attention
- Example: Structured sparse attention pattern, attending to only a few tokens
- Example: low-rank methods project the keys and values to lower-dimensional subspace before attention
- Multi Query Attention (MQA) and Grouped Query Attention (GQA): multiple attention heads share keys, values

Efficient Attention and Efficient Transformers



(a) Transformer

(b) Sparse Transformer

Figure: Illustration of patterns of the attention matrix for dense self-attention in Transformers and sparse fixed attention in Sparse Transformers

Image Credit: `https://arxiv.org/pdf/2009.06732`

Parameter-Efficient Fine-Tuning

- Customizing a large model to a new specific task
- Low-Rank Adaptation (LoRA) – based on weight updates being low-rank matrices
  - $W = W_0 + \Delta W \Rightarrow W = W_0 + \frac{\alpha}{r} AB$
  - A of dimension $d \times r$ and B of dimension $r \times k$, where the rank r is chosen to be much smaller than d or k
- Multiple LoRA adapters, one for each task
- QLoRA stores W_0 in a low-precision format, while A and B in higher precision