

Lecture 10: GPUs, FlashAttention, and Inference

Prof. Noah Golowich

Today we will study efficient attention and inference, then analyze self-improvement through sharpening.

1 GPU Background

A *graphics processing unit* (GPU) performs many arithmetic operations in parallel. In NVIDIA GPUs, the arithmetic units are organized into *streaming multiprocessors* (SMs). Each SM has small, fast memories (“caches”), and all SMs share a larger main memory. An H100 SXM has 132 SMs [7]. Figures 1 and 2 show the chip and one SM.

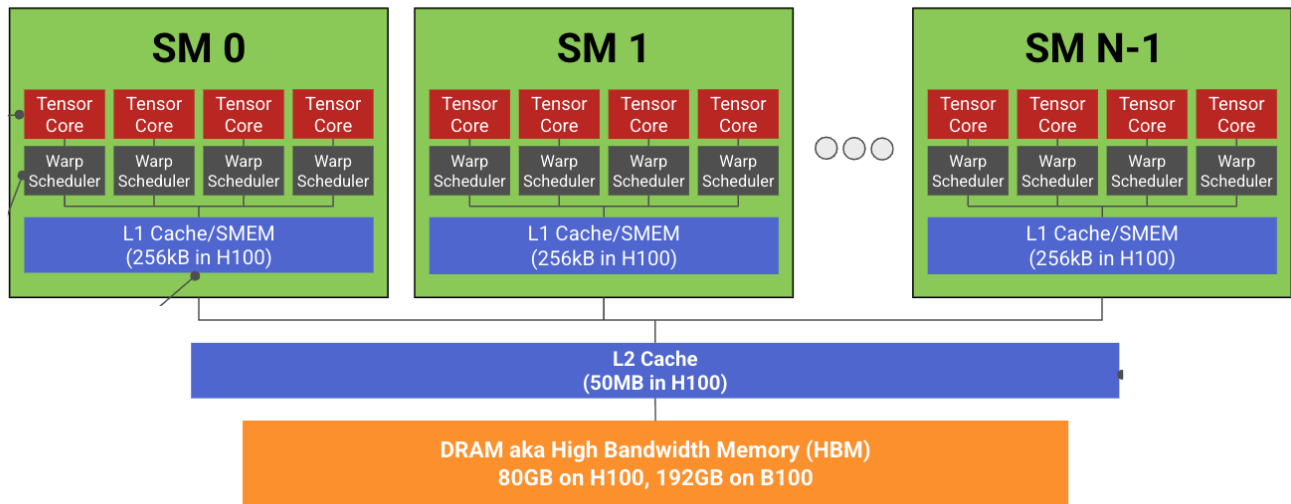


Figure 1: GPU compute and memory hierarchy [2].

Each H100 SM has four subpartitions, each with a warp scheduler, registers, and two kinds of arithmetic units:

- *CUDA cores* perform ordinary arithmetic, including pointwise operations and reductions. A warp scheduler issues instructions to groups of 32 threads called *warps*; threads in a warp execute a common instruction on different data.
- *Tensor cores* multiply small matrices and accumulate their products. They provide most of the GPU’s floating-point throughput, with higher throughput at lower precision. They accelerate the large matrix multiplications in transformers; other operations use the CUDA cores.

Memory. *SRAM* means *static random-access memory*: it retains bits while powered without the periodic refresh required by dynamic RAM (DRAM). It is fast but occupies more chip area per bit. GPU registers and on-chip caches use SRAM; HBM uses DRAM. The relevant memory hierarchy is:

- *Registers*: thread-local working storage in each SM subpartition. An H100 SM has 256 KiB in total; using many registers per thread limits how many threads can be resident at once.
- *SMEM / L1*: each H100 SM has a 256 KiB pool divided between programmer-managed shared memory and a hardware-managed L1 cache. Shared memory holds tiles of activations and matrix inputs that threads reuse.
- *L2*: a hardware-managed cache shared across SMs, about 50 MB on H100. It can serve repeated requests without accessing main memory.
- *HBM* (high-bandwidth memory): the large main memory, holding weights, activations, and, during training, gradients and optimizer state. An H100 SXM has 80 GB, with about 3.35 TB/s of memory bandwidth [6].

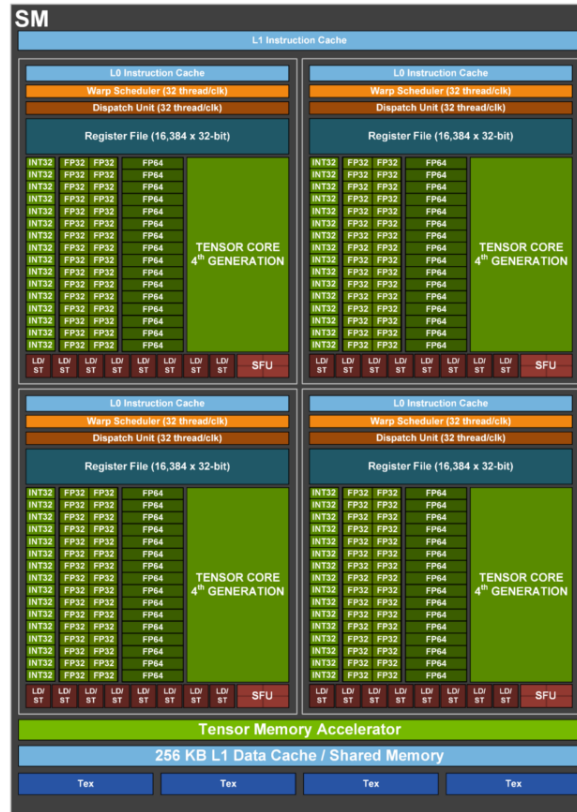


Figure 2: One H100 SM, with four subpartitions and a shared L1/SMEM pool [2].

Compute-bound and memory-bound operations. Computing floating-point operations (FLOPs) on resident data is much faster than repeatedly moving that data between HBM and SRAM. For example, H100 SXM has a peak dense FP16/BF16 Tensor Core throughput of roughly $9.9 \cdot 10^{14}$ FLOPs/s and a peak HBM bandwidth of $3.35 \cdot 10^{12}$ bytes/s [6, 7].¹ To keep the matrix units busy at these rates, we need roughly 300 FLOPs per byte transferred.

An operation is *compute-bound* when arithmetic determines its running time and memory transfer takes much less time; it is *memory-bound* when memory transfer determines its running time and

¹NVIDIA lists 1,979 TFLOPs/s for FP16/BF16 with structured sparsity, which doubles dense Tensor Core throughput; dividing by two gives approximately 990 TFLOPs/s. These are hardware peak rates, not measured application throughput.

arithmetic takes much less time [1]. Large matrix multiplications with substantial data reuse are often compute-bound. Pointwise operations and reductions, such as softmax, are often memory-bound.

Let $\mathcal{F}$ be the FLOP count and $\mathcal{D}$ the bytes transferred between HBM and the chip. Let C be the available compute rate in FLOPs/s for the operation's arithmetic and β the HBM bandwidth in bytes/s. Define the *arithmetic intensity*

$$I := \frac{\mathcal{F}}{\mathcal{D}} \quad (\text{FLOPs/byte}), \quad I_* := \frac{C}{\beta} \quad (\text{hardware balance}). \quad (1)$$

The idealized *roofline model* assumes that computation occurs at the same time as memory transfer:

$$t_{\text{comp}} = \frac{\mathcal{F}}{C}, \quad t_{\text{mem}} = \frac{\mathcal{D}}{\beta}, \quad t \approx \max\{t_{\text{comp}}, t_{\text{mem}}\}. \quad (2)$$

Thus $I > I_*$ predicts a compute bottleneck, while $I < I_*$ predicts a memory bottleneck. The maximum is an optimistic lower bound when C and β are peak rates; launch overheads and imperfect utilization can increase the time. We count a multiplication and an addition as two FLOPs.

2 Flash attention

FlashAttention [1] computes the usual attention output while avoiding the large intermediate attention matrix in HBM. A direct implementation shows where the memory traffic comes from.

2.1 Naive attention

We first review a naive implementation of attention. Consider one attention head, and let the sequence length be N and the query, key, and value dimensions all be d . Write

$$Q = [q_1, \dots, q_N], \quad K = [k_1, \dots, k_N], \quad V = [v_1, \dots, v_N] \in \mathbb{R}^{d \times N}.$$

The score matrix S , attention matrix A , and output Y are

$$S = \frac{K^\top Q}{\sqrt{d}}, \quad A_{ji} = \frac{e^{S_{ji}}}{\sum_{u=1}^N e^{S_{ui}}}, \quad Y = VA = [y_1, \dots, y_N] \in \mathbb{R}^{d \times N}. \quad (3)$$

Softmax acts on each *column*, and A_{ji} is the weight that query i assigns to key j .

We describe unmasked attention. (For causal attention, set $S_{ji} = -\infty$ for $j > i$, so each query attends only to itself and earlier positions.)

Algorithm 0 Naive attention (column-oriented form of [1, Algorithm 0])

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

- 1: Load K, Q by blocks, compute $S = K^\top Q / \sqrt{d}$, and write S to HBM.
 - 2: Read S , compute $A = \text{softmax}(S)$ column-wise, and write A to HBM.
 - 3: Load V, A by blocks, compute $Y = VA$, and write Y to HBM.
 - 4: **return** Y .
-

The two matrix multiplications take $\Theta(N^2d)$ FLOPs. But S and A each contain N^2 entries. Writing and rereading them requires $\Theta(N^2)$ HBM transfers and $\Theta(N^2)$ intermediate storage. Fusing softmax with the matrix multiplications would avoid these transfers, but an entire score matrix generally does not fit in SRAM.

2.2 Tiling and recomputation

FlashAttention uses two ideas:

- *Tiling*: load small blocks of queries, keys, and values into SRAM, and compute a block of scores there. Update the softmax normalization and output for each query, then discard the scores. Only the small running state must persist between blocks.
- *Recomputation*: during training, retain the output and softmax normalization statistics. Recreate the needed score and attention blocks in SRAM during the backward pass, instead of saving all N^2 entries. Recomputing these entries can take less time than reading them from HBM. The algorithm below is the tiled forward pass; recomputation is needed during the backward pass.

An online softmax update. For a fixed query i , suppose we have processed a set J of keys. Store

$$m_i = \max_{j \in J} S_{ji}, \quad \ell_i = \sum_{j \in J} e^{S_{ji} - m_i}, \quad y_i = \frac{\sum_{j \in J} e^{S_{ji} - m_i} v_j}{\ell_i}. \quad (4)$$

For each new block of keys that arrives online, compute its maximum $\tilde{m}_i$, exponential sum $\tilde{\ell}_i$, and unnormalized weighted value sum $\tilde{u}_i$. To combine old and new blocks, put

$$m'_i = \max\{m_i, \tilde{m}_i\}, \quad \ell'_i = e^{m_i - m'_i} \ell_i + e^{\tilde{m}_i - m'_i} \tilde{\ell}_i, \quad y'_i = \frac{e^{m_i - m'_i} \ell_i y_i + e^{\tilde{m}_i - m'_i} \tilde{u}_i}{\ell'_i}. \quad (5)$$

The factors $e^{m_i - m'_i}$ and $e^{\tilde{m}_i - m'_i}$ express both contributions relative to the new maximum m'_i , which maintains (4) without retaining earlier scores. Initialize $m_i = -\infty$, $\ell_i = 0$, $y_i = 0$; the old contribution in the first update is zero.

Block sizes and memory. Measure SRAM capacity M in *scalar words*, rather than bytes. Partition the token columns of Q into blocks Q_i of at most B_q columns and those of K, V into blocks K_j, V_j of at most B_k columns. We choose

$$B_k = \left\lceil \frac{M}{4d} \right\rceil, \quad B_q = \min\{B_k, d\}, \quad T_q = \left\lceil \frac{N}{B_q} \right\rceil, \quad T_k = \left\lceil \frac{N}{B_k} \right\rceil. \quad (6)$$

With these choices, $B_k = \Theta(M/d)$, and the live tiles occupy $O(M)$ words: K_j, V_j use $O(dB_k)$, Q_i, Y_i use $O(dB_q)$, and a score tile uses $O(B_k B_q)$.

Figure 3 shows one step: the key/value block j stays fixed while the algorithm moves through query blocks i .

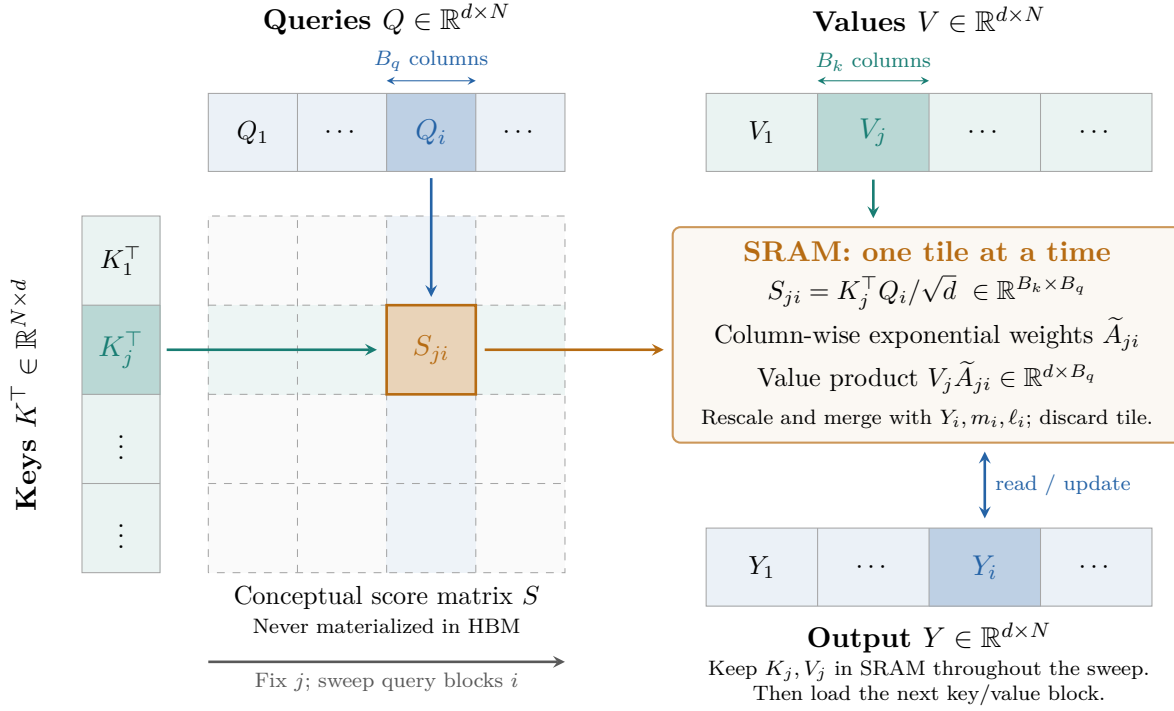


Figure 3: FlashAttention tiling with tokens stored as columns. The inputs and running output reside in HBM. Blue marks the current query/output block, green the key/value block, and orange the temporary score tile. The dashed score matrix shows the partition only; each tile is computed and discarded in SRAM.

For a matrix, `colmax` and `colsum` return vectors of column maxima and sums. Exponentials and products of vectors below act entrywise, $\odot$ denotes entrywise multiplication, $\mathbf{1}$ is an all-ones column vector of the required length, and `diag( $z$ )` has vector z on its diagonal.

Algorithm 1 FlashAttention forward pass (column-oriented form of [1, Algorithm 1])

Require: $Q, K, V \in \mathbb{R}^{d \times N}$ in HBM; SRAM workspace of order M words.

- 1: Choose block sizes and counts as in (6).
 - 2: In HBM initialize $Y = 0 \in \mathbb{R}^{d \times N}$, $\ell = 0 \in \mathbb{R}^N$, $m = -\infty \in \mathbb{R}^N$.
 - 3: **for** $j = 1, \dots, T_k$ **do**
 - 4: Load K_j, V_j into SRAM.
 - 5: **for** $i = 1, \dots, T_q$ **do**
 - 6: Load Q_i, Y_i, ℓ_i, m_i into SRAM.
 - 7: $S_{ji} \leftarrow K_j^\top Q_i / \sqrt{d}$.
 - 8: $\tilde{m} \leftarrow \text{colmax}(S_{ji})$; $\tilde{A} \leftarrow \exp(S_{ji} - \mathbf{1}\tilde{m}^\top)$; $\tilde{\ell} \leftarrow \text{colsum}(\tilde{A})$.
 - 9: $m' \leftarrow \max(m_i, \tilde{m})$.
 - 10: $\ell' \leftarrow e^{m_i - m'} \odot \ell_i + e^{\tilde{m} - m'} \odot \tilde{\ell}$.
 - 11: $Y_i \leftarrow (Y_i \text{diag}(e^{m_i - m'} \odot \ell_i) + V_j \tilde{A} \text{diag}(e^{\tilde{m} - m'})) \text{diag}(\ell')^{-1}$.
 - 12: Write $Y_i, \ell_i \leftarrow \ell'$, and $m_i \leftarrow m'$ to HBM.
 - 13: **end for**
 - 14: **end for**
 - 15: **return** Y (retain m, ℓ if a backward pass is needed).
-

Here subscripts i on Y, ℓ, m denote blocks of queries, and S_{ji} denotes a whole score tile. All computations inside the inner loop take place in SRAM.

Theorem 1 (Correctness and storage [1, Theorem 1]). *For dense attention, Algorithm 1 returns $Y = V \text{softmax}(K^\top Q / \sqrt{d})$, where softmax is column-wise. It uses $O(N^2d)$ FLOPs and $O(N)$ additional HBM storage beyond the inputs and output, together with the SRAM workspace.*

Proof sketch. The update (5) preserves (4). After all keys have been processed, its normalized weighted sum is exactly (3). Each query–key pair participates in one dot product and one weighted value contribution, each costing $O(d)$ operations, giving $O(N^2d)$ in total. The only extra persistent arrays are the N maxima and N normalization sums; all score tiles are temporary SRAM data. $\square$

Theorem 2 (HBM transfers [1, Theorem 2]). *Let $d \leq M \leq Nd$. The naive implementation of attention makes $\Theta(Nd + N^2)$ HBM word accesses, while Algorithm 1 makes*

$$\Theta\left(\frac{N^2d^2}{M}\right) \quad (7)$$

HBM word accesses. (One access reads or writes one scalar word.)

Proof sketch. Naive attention reads and writes the N^2 scores and probabilities, in addition to the Nd -sized inputs and output. For FlashAttention, each key/value block is loaded once. There are $T_k = \Theta(Nd/M)$ such blocks, and each induces a pass over all queries, outputs, and normalization statistics, costing $\Theta(Nd)$ accesses. Thus the total is

$$\Theta(Nd + NdT_k) = \Theta(Nd + N^2d^2/M) = \Theta(N^2d^2/M),$$

where the last equality uses $M \leq Nd$. $\square$

3 Inference tricks

Caching and the two phases. *Inference* uses a trained model with fixed parameters to produce predictions or tokens. Autoregressive inference has two phases:

- *Prefill*: process the known prompt tokens in parallel, subject to the causal mask. At every layer, save their key and value vectors in a *KV cache*. The last prompt position’s logits give the distribution of the first generated token.
- *Generation* (or *decoding*): sample a token from the current logits, process that token using the existing KV cache, and append its new keys and values. The output logits then predict the next token. Ordinary decoding processes one new token per sequence per step.

Appending a token cannot change earlier tokens’ keys or values in a causal transformer. We can therefore reuse them (*KV-caching*). We use “generation” for the incremental phase and “inference” for both phases together.

Let B be the number of sequences in a batch, T the number of query tokens processed per sequence in one forward call, and S the number of key/value positions available to each sequence. For ordinary prefill, $S = T$ is the prompt length; for generation, $T = 1$ and S is the current context length, including the new token. Assume equal lengths within the batch. The following estimates use 16-bit values (two bytes per scalar) and omit non-matmul arithmetic.

3.1 Parameter matrix multiplications

The feedforward layers and attention projections multiply token activations by parameter matrices. Let $W \in \mathbb{R}^{F \times D}$ map width D to width F , and combine the BT input tokens into $X \in \mathbb{R}^{D \times BT}$. Computing WX takes

$$\mathcal{F}_{\text{lin}} = 2BTDF, \quad \mathcal{D}_{\text{lin}} = 2(BTD + DF + BTF). \quad (8)$$

The byte count reads X and W once and writes the output once. It assumes SRAM is large enough so as to avoid extra HBM loads. Therefore

$$t_{\text{comp}} = \frac{2BTDF}{C}, \quad t_{\text{mem}} = \frac{2(BTD + DF + BTF)}{\beta}, \quad I_{\text{lin}} = \frac{BTDF}{BTD + DF + BTF}. \quad (9)$$

If $BT \ll D, F$, parameter reads dominate the bytes, and

$$I_{\text{lin}} \approx BT, \quad \text{compute-bound when } BT \gtrsim I_*. \quad (10)$$

Here BT counts tokens, including those within the same prompt.

Prefill. A single long prompt supplies hundreds or thousands of tokens that reuse the same weights. Thus parameter multiplications can be compute-bound even with $B = 1$. Short prompts can be batched together to increase BT .

Generation. Now $T = 1$, so $I_{\text{lin}} \approx B$ in the weight-dominated regime. At small B , reading the weights takes longer than using them, so the multiplication is memory-bound. Increasing B amortizes each weight read over more tokens. Using the H100 SXM peak rates cited in Section 1 [6, 7], $I_* \approx 295$ FLOPs/byte, so the rough crossover is a few hundred tokens per call.

3.2 Attention

Consider one head of width d in each of the B sequences. Each sequence has $Q \in \mathbb{R}^{d \times T}$ and $K, V \in \mathbb{R}^{d \times S}$. In an ideal fused attention calculation (e.g., Flash Attention with $M \geq \max\{S, T\} \cdot d$), we read Q, K, V , compute both matrix products, and write $Y \in \mathbb{R}^{d \times T}$:

$$\mathcal{F}_{\text{attn}} = 2BSTd + 2BSTd = 4BSTd, \quad (11)$$

$$\mathcal{D}_{\text{attn}} = 2BTd + 4BSd + 2BTd = 4Bd(S + T). \quad (12)$$

There is no HBM write of an $S \times T$ attention matrix. Dividing gives

$$t_{\text{comp}} = \frac{4BSTd}{C}, \quad t_{\text{mem}} = \frac{4Bd(S + T)}{\beta}, \quad I_{\text{attn}} = \frac{ST}{S + T}. \quad (13)$$

The byte count in (12) is an optimistic model in which each input is read once. Finite SRAM can force repeated loads, as Theorem 2 shows. For dense square attention under that theorem's model, dividing $\Theta(N^2d)$ FLOPs by $\Theta(N^2d^2/M)$ word transfers gives intensity $\Theta(M/d)$ at fixed bytes per word. A longer prompt therefore provides more potential reuse, but the SRAM capacity limits how much of it we can exploit.

Prefill. Putting $S = T$ in (13) gives

$$I_{\text{attn}} = T/2.$$

Many queries reuse the same keys and values. In the rectangular idealization, compute becomes the bottleneck when $T \gtrsim 2I_*$, roughly 600 tokens with the cited H100 SXM peak rates [6, 7]. Causal masking changes this constant, and the actual crossover also depends on tile reuse and kernel efficiency.

Generation. Putting $T = 1$ and $S \gg 1$ gives

$$I_{\text{attn}} = \frac{S}{S+1} \approx 1 \quad \text{FLOP/byte.}$$

Ordinary attention during generation is therefore memory-bound. A larger batch does not help this ratio: each new sequence brings its own KV cache, adding bytes at the same rate as FLOPs. This differs from the shared weights in Section 3.1.

References

- [1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *NeurIPS*, 2022. <https://arxiv.org/abs/2205.14135>.
- [2] Jacob Austin, Swapnil Patil, Adam Paszke, and Reiner Pope. How to Think About GPUs. *How To Scale Your Model*, 2025. <https://jax-ml.github.io/scaling-book/gpus/>.
- [3] *How To Scale Your Model*. All About Rooflines. <https://jax-ml.github.io/scaling-book/roofline/>.
- [4] *How To Scale Your Model*. All About Transformer Inference. <https://jax-ml.github.io/scaling-book/inference/>.
- [5] Audrey Huang, Adam Block, Dylan J. Foster, Dhruv Rohatgi, Cyril Zhang, Max Simchowitz, Jordan T. Ash, and Akshay Krishnamurthy. Self-Improvement in Language Models: The Sharpening Mechanism. arXiv:2412.01951, 2024. <https://arxiv.org/abs/2412.01951>.
- [6] NVIDIA. NVIDIA H100 GPU: Product Specifications (H100 SXM). <https://www.nvidia.com/en-us/data-center/h100/>.
- [7] Michael Andersch et al. NVIDIA Hopper Architecture In-Depth. *NVIDIA Technical Blog*, March 22, 2022. <https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>.