FlashAttention on a Napkin: A Diagrammatic Approach to Deep Learning IO-Awareness

Anonymous authors
Paper under double-blind review

Abstract

Optimizing deep learning algorithms currently requires slow, manual derivation, potentially leaving much performance untapped. Methods like FlashAttention have achieved a ×6 performance improvement over native PyTorch by avoiding unnecessary data transfers, but required three iterations over three years. Automated compiled methods have consistently lagged behind. GPUs are limited by both transfers to processors and available compute, with transfer bandwidth having improved at a far slower pace. Already, transfer bandwidth accounts for 46% of GPU energy costs. This indicates the future of energy and capital-efficient algorithms relies on improved consideration of transfer costs (IO-awareness) and a systematic method for deriving optimized algorithms. In this paper, we present a diagrammatic approach to deep learning models which, with simple relabelings, derive optimal implementations and performance models that consider low-level memory. Diagrams generalize down the GPU hierarchy, providing a universal performance model for comparing hardware and quantization choices. Diagrams generate pseudocode, which reveals the application of hardware-specific features such as coalesced memory access, tensor core operations, and overlapped computation. We present attention algorithms for Ampere, which fits 13 warps per SM (FlashAttention fits 8), and for Hopper, which has improved overlapping and may achieve 1.32 PFLOPs.

1 Introduction

1.1 Background

To execute an operation, graphical processing units (GPUs) must move data from high-level DRAM to low-level compute cores. GPUs are as limited as much by GB/s of memory bandwidth as TFLOPs of available compute. However, AI models have passed the *memory wall*—algorithms are increasingly limited by bandwidth/transfer costs (Ootomo & Yokota, 2023; Ivanov et al., 2021; Gholami et al., 2024), as compute capability has improved far more quickly ×3/2y than DRAM bandwidth ×1.6/2y (Gholami et al., 2024). Furthermore, DRAM already accounts for 46% of total system power (Ghose et al., 2018). As memory becomes increasingly inefficient relative to compute, the importance of considering transfer costs—*IO-awareness* (Dao et al., 2022; Aggarwal & Vitter, 1988)—will become even more critical.

FlashAttention (Dao et al., 2022; Dao, 2023; Shah et al., 2024) is an IO-aware approach to attention that overcomes the memory wall. Attention (Vaswani et al., 2017) is central to generative models, including large language models (LLMs) (Jiang et al., 2024; Dubey et al., 2024) and image generation algorithms (Ho et al., 2020; Esser et al., 2024; Rombach et al., 2022; Podell et al., 2023). FlashAttention *fuses* the steps of attention. It computes all sequential steps on low-level memory, avoiding unnecessary intermediate data transfers. It achieves a ×6 increase in throughput compared to standard PyTorch, arguably making large contemporary models possible.

However, the conditions under which fusion is possible are not generally exploited. Simple cases like elementwise functions can be compiled into matrix multiplications (Li et al., 2020; Paszke et al., 2019; Sabne, 2020), but the bespoke techniques of FlashAttention required manual derivation and three iterations over three years to take full advantage of Hopper hardware (NVIDIA, 2022) features. *Triton* (Tillet et al., 2019) offers some compilation for hardware features but has lagged behind new FlashAttention algorithms (Dao, 2023; Shah et al., 2024). The current best technique for generating IO-aware algorithms that exploit hardware features remains slow, manual derivation.

Innovating new optimized algorithms is essential to efficient deployment of models. In addition to FlashAttention, methods like grouped query attention (Ainslie et al., 2023), KV-caching (Shazeer, 2019), and quantization (Frantar et al., 2023; Gholami et al., 2021) all reduce transfer costs while having minimal impact on the function we implement or the quality of model outputs. Much like fusion, the success of these approaches relies on understanding the compositional structure of algorithms so that similar but less costly algorithms can be executed. A systematic approach to innovating optimized algorithms will require a mechanism for understanding the compositional structure of algorithms along with a performance model which compares varying means of executing the same operation.

The hardware characteristics of GPUs have a significant impact on performance which varies depending on the target algorithm. When choosing between A100s, H100 SXM5s, or H100 PCIes (NVIDIA, 2022, p.39), we must consider the varying compute, bandwidth, intermediate hierarchies, architecture features, and low-level memory, for which we pay in environmental and economic resources. The application of these features is often non-obvious, FlashAttention-2 (Dao, 2023) was released while the hardware for FlashAttention-3 already existed (Shah et al., 2024), which achieved $\sim 75\%$ improvement in forward speed. Understanding the impact of GPU features is a necessary component of innovating optimized approaches and making full use of deployed resources.

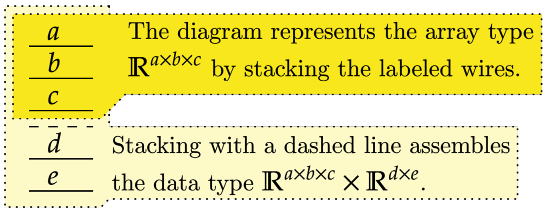
1.2 Contributions

This paper contributes a diagrammatic scheme for representing deep learning algorithms based off *Neural Circuit Diagrams* (Abbott, 2023) (Section 2) which can be used to quickly derive methods like FlashAttention along with a performance model for transfer costs which factors in lower-level cache size (Section 3). We then show how the performance model scales to a multi-level hierarchy, providing expressions that considers GPU hierarchy configurations and the memory sensitivity of algorithms (Section 4). Finally, we show how diagrams can be converted to pseudocode, which reveals the application of hardware-specific features like coalesced memory access, tensor-core operations, and overlapping operations (Section 5). To show the advantages of this approach, we present Ampere and Hopper attention algorithms with reduced low-level memory usage compared to FlashAttention.

2 Diagramming Deep Learning Algorithms

2.1 Diagramming Functions and Data Types

Diagrams have alternating columns of data types and functions. Data type columns are shown in Figure 1. Arrays such as $\mathbb{R}^{a\times b\times c}$ are represented by a wire for each axis labeled with the size. Data types may be tuples of arrays, such as $\mathbb{R}^{a\times b\times c} \times \mathbb{R}^{d\times e\times c}$, and are represented by placing a dashed line between constituent arrays.

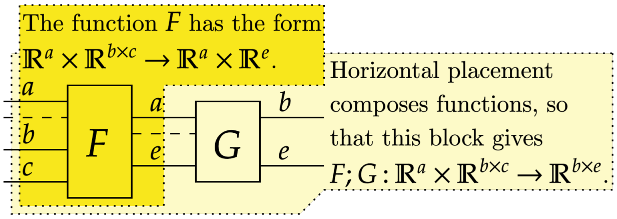


The diagram represents the array type $\mathbb{R}^{a\times b\times c}$ by stacking the labeled wires.

Stacking with a dashed line assembles the data type $\mathbb{R}^{a\times b\times c} \times \mathbb{R}^{d\times e}$.

Figure 1: We represent arrays, of forms such as $\mathbb{R}^{a\times b\times c}$, by labeling stacked wires in a column with a, b, and c. To represent data types that consist of lists of arrays, such as $\mathbb{R}^{a\times b\times c} \times \mathbb{R}^{d\times e}$, we place a dashed line between them.

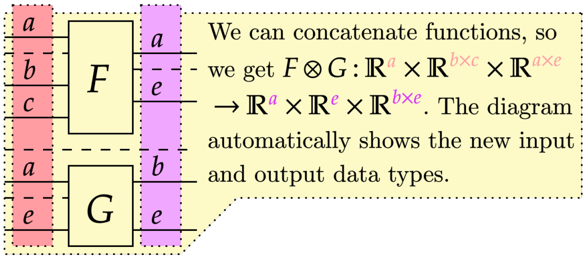
Functions between data types are represented by labeled boxes or pictograms with their input/output shapes to the left/right. Sequential execution (*composition*) of functions is shown by horizontal placement (Figure 2, creating a diagram with alternating columns representing data types and functions. Parallel execution (*concatenation*) of functions stacks them with a dashed line in between (Figure 3). A concatenated function takes concatenated inputs and provides concatenated outputs. The change in the input/output is reflected by the diagram.



The function F has the form $\mathbb{R}^a \times \mathbb{R}^{b\times c} \to \mathbb{R}^a \times \mathbb{R}^e$.

Horizontal placement composes functions, so that this block gives $F; G : \mathbb{R}^a \times \mathbb{R}^{b\times c} \to \mathbb{R}^{b\times e}$.

Figure 2: Functions are represented by labeled boxes or pictograms, which aid intuition. These representations can be horizontally composed, which represents sequential execution and yields a diagram with alternating data type and function columns. We represent composition by $F; G = G \circ F$.



We can concatenate functions, so we get $F \otimes G : \mathbb{R}^a \times \mathbb{R}^{b\times c} \times \mathbb{R}^{a\times e} \to \mathbb{R}^a \times \mathbb{R}^e \times \mathbb{R}^{b\times e}$. The diagram automatically shows the new input and output data types.

Figure 3: Functions can also be stacked with a separating dashed line, which concatenates their inputs and outputs. Concatenating tuples $\otimes$ is considered to be associative. For concatenated functions $F \otimes G$, if $F(x) = x'$ and $G(y) = y'$ then $(F \otimes G)(x \otimes y) = F(x) \otimes G(y)$.

We represent identity functions that leave inputs unchanged by extending the data type. This reflects that composition with identities leaves a function unchanged. Functions are stateless and are defined by how they map inputs to outputs. Therefore, we concatenate with identities to represent a function acting on some data but leaving the rest unchanged and available. With these tools, we can build compound diagrams such as Figure 4.

We have alternating columns of data types and functions. These columns are concatenated along dashed lines. Continued wires in function columns are identities.

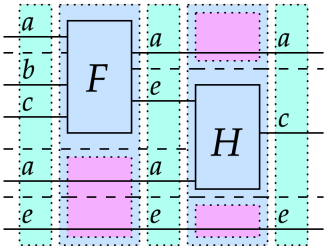


Figure 4: A compound diagram can be disassembled into columns representing alternating functions and data types. Stacked functions and data types can be further decomposed to find the core units concatenated to construct them. Identities are represented by continuing the representation of data types.

Functions can be mapped over an additional axis, represented by weaving the axis over the function's outputs and a choice of the inputs (Figure 5). This diagrammatic implementation naturally updates the input and output sizes for the mapped function. When an input segment is not weaved, its data is copied to evaluate each index along the outputs of the new axis. The axis can be weaved into any location of the target segments.

Weaving a function allows for complex mappings to be represented and avoids the ambiguity of typical expressions. We can weave primitives defined on items, such as multiplication, addition, and copying. We use weaving to represent the splitting and joining of shared axes, which overcomes the typical difficulties of expressing how arrays are partitioned and concatenated. We show this in Figure 6.

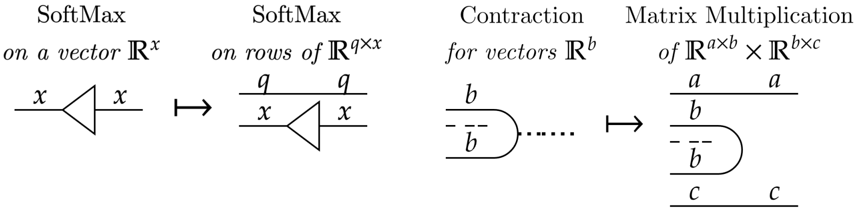

SoftMax
on a vector $\mathbb{R}^x$
x
x
$\mapsto$
SoftMax
on rows of $\mathbb{R}^{q \times x}$
q
q
x
x
Contraction
for vectors $\mathbb{R}^b$
b
b
$\mapsto$
Matrix Multiplication
of $\mathbb{R}^{a \times b} \times \mathbb{R}^{b \times c}$
a
a
b
b
c
c

Figure 5: A function can be weaved, which adds an axis to the outputs and some of the inputs. The function is mapped over this axis. When we weave the "item" $\mathbb{R}$ array represented by a thick dotted wire, we can remove it. Here, we provide a weaving for SoftMax, represented by a triangle, to have it act over each row of an array, and of linear contraction (*dot/inner product*), which provides matrix multiplication.

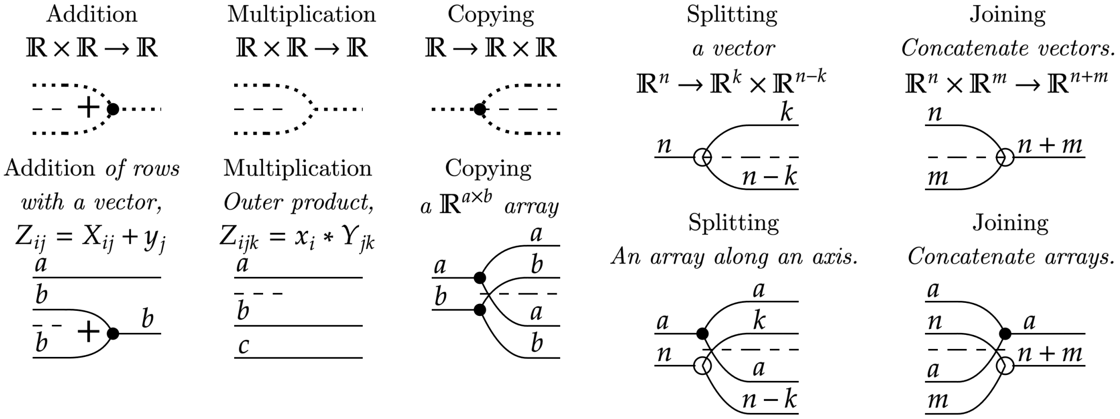
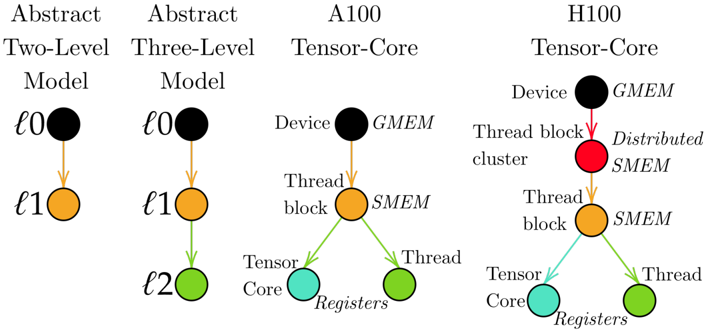


Figure 6: Weaving primitive functions let us express the addition of vectors to each row of an array, multiplication over a specific axis, and the copying of arrays. We can use weaving to split an array along slices of an axis or show the axes over which arrays are joined.

2.2 Representing Deep Learning Algorithms

We have so far expressed *functions* — maps between inputs and outputs — diagrammatically. Deep learning models employ *algorithms*, which implement a function but have resource usages and inputs/outputs located at different levels. We embed algorithms in a hierarchy. A hierarchy consists of levels connected with pipes (as in Figure 7), which allow for memory sharing with a family of cores located at the level below. The available algorithms are restricted to those provided at each level of the hierarchy.

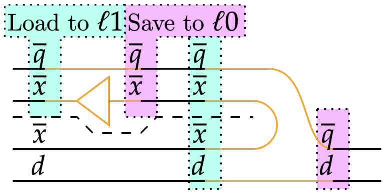
Figure 7: We can diagram hierarchies using a graph showing the available levels and their connections. For hierarchies representative of real GPUs, we note the layer abstraction and the physical memory it occupies in italics.



We use colors to represent levels, and color arrays to diagram where they are located. In this section, we use a two-level model where higher level $\ell 0$ arrays are colored black and lower level $\ell 1$ arrays are colored orange. This lets us diagram algorithms as in Figure 8.

We are interested in algorithms' total transfer cost H_ℓ and maximum memory usage per core M_ℓ (memory usage). These resource usages are defined per level and measured in number of values, and can be determined

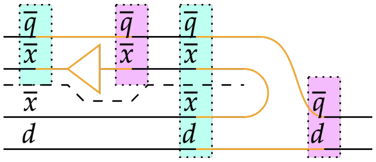
Figure 8: Here, we diagram an algorithm which takes a SoftMax over a $\overline{q} \times \overline{x}$ array and contracts it over a $\overline{x} \times \overline{d}$ array. We perform transfers to move data to lower levels for computation. This diagram shows sequential execution, concatenation, and weaving of algorithms diagrammatically.



from diagrams as in Figure 9. Total transfer costs are equal to the total size of data loaded to and saved from a level, equal to the sum of the size of arrays changing colors. Memory usage is lower bounded by the maximum size of data at a level for any column. We aim to minimize the total transfers while keeping memory usage below a hardware limit per level, $M_\ell^{\max}$.

Figure 9: The SoftMax-Contraction algorithm from Figure 8 can have its transfer cost derived from the total size of data changing colors and its memory usage lower bound determined by the maximum data present at the lower level at any point. We assume that $\overline{x} > d$.

					Cumulative
$M_{\ell 1}$			$\overline{q}\overline{x} + \overline{x}d$		
$H_{\ell 1}$	$\overline{q}\overline{x}$	$\overline{q}\overline{x}$	$\overline{q}\overline{x} + \overline{x}d$	$\overline{q}\overline{x}$	$4\overline{q}\overline{x} + \overline{x}d$

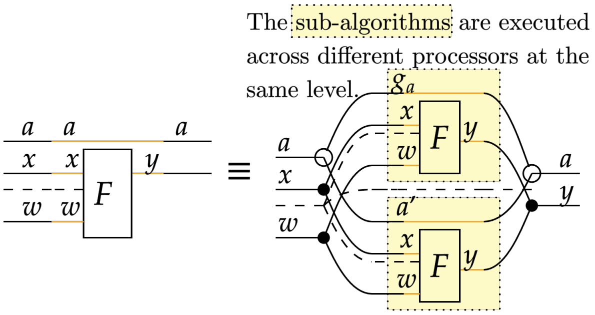


As diagrams show all data types, operations, and their arrangement, we can adapt our performance model to consider all aspects of an algorithm's performance. Using diagrams, we can approximate compute by taking the compute required to execute an algorithm multiplied by the size of axes it is weaved over. A k-size contraction requires $2k$ FLOPs; therefore, $m \times k$ by $k \times n$ matrix multiplication requires $2mkn$ FLOPs. In Appendix A.5.3, this is used to find the clock cycles required per column to overlap computation.

2.3 Group Partitioning

The first optimization strategy we introduce is group partitioning (*tiling*) a mapped algorithm (Figure 10). If an algorithm is weaved over an axis, then the axis can be split, the mapped function applied, and the axis rejoined to yield the same result. Each sub-algorithm acts on a batch of the data in a separate core with reduced low-level memory usage.

Figure 10: A weaved algorithm is functionally equivalent to sub-algorithms acting on partitions of the weaved axis. We use $\equiv$ to indicate functional equivalence, meaning algorithms map the same inputs to outputs but may have distinct resource consumption profiles, and therefore are not strictly equivalent. These partitions can be of any size, which we write as g_a. We can recursively expand the expression on the right until $a' \leq g_a$. The unweaved segment of the data must be loaded by each sub-algorithm.

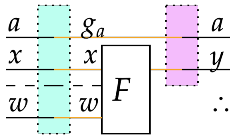

The sub-algorithms are executed across different processors at the same level.

We can diagram this strategy by labeling the weaved axis a with the target group size g_a while at the lower level as in Figure 11. The low-level memory usage M_ℓ for the diagram is then calculated using this group size, not the full size of the axis. Each sub-algorithm needs to load and save its batch input and output data. The per-group transfer cost $H_{\ell,g}$ is calculated using the g_a group size for the partitioned axis which is multiplied by $N_{\ell,g} = a/g_a$ batches to attain $H_\ell = N_{\ell,g} H_{\ell,g}$.

Non-grouped inputs are sent to all active cores at the lower level, meaning their transfer costs are multiplied by $N_{\ell,g}$ without reduced per-group transfer costs. Smaller group sizes g_a decrease memory usage but increase

$N_{\ell,g}$, increasing H_ℓ if there is an unweaved input. To reduce total transfer costs, we must find the maximum g_a value that does not exceed maximum memory usage $M_\ell^{\max}$.

$M_{\ell 1}$	$g_a x + w$	$g_a y$	*Cumulative*
$H_{\ell 1,g}$	$g_a x + w$	$g_a y$	$g_a(x+y)+w$

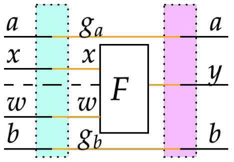


$$M_{\ell 1} \geqslant g_a x + w, g_a y$$
$$H_{\ell 1,g} = g_a(x+y)+w$$
$$N_{\ell 1,g} = a/g_a$$
$$\therefore H_{\ell 1} = ax + ay + aw/g_a$$

Figure 11: Group partitioning can be represented by relabeling an algorithm with the partition batch size (group size) at the lower level. The group size is used for memory usage and per-group transfer cost calculations for the lower level. Per-group transfer costs are then multiplied by the number of groups $N_{\ell,g} = a/g_a$ to give the overall transfer costs.

If multiple weaves are relabeled, the data is batched over each as in Figure 12. The total number of sub-algorithms is the product of the number of batches for each axis, $N_{\ell,g} = ab/g_a g_b$. The relabeled sub-algorithm represents the memory usage and per-group transfer costs of each sub-algorithm, using the group sizes g_a and g_b for resource usage calculations. We then multiply the transfer costs by $N_{\ell,g}$. We can use this to determine the optimal group sizes for an algorithm grouped over multiple axes, such as matrix multiplication (see Section 3.1), and to determine whether it is worth grouping a small axis or transferring its full size.

$M_{\ell 1}$	$g_a x + w g_b$	$g_a y g_b$	*Cumulative*
$H_{\ell 1,g}$	$g_a x + w g_b$	$g_a y g_b$	$g_a x + g_b w + g_a g_b y$



$$M_{\ell 1} \geqslant g_a x + g_b w, g_a g_b y$$
$$H_{\ell 1,g} = N_g(g_a x + g_b w + g_a g_b y)$$
$$N_{\ell 1,g} = ab/g_a g_b$$
$$\therefore H_{\ell 1} = abx\left(g_b^{-1} + g_a^{-1}\right) + aby$$

Figure 12: A function with multiple weaves can have a relabeling applied to each of its weaves. The relabeled sub-algorithm provides the memory usage and per-group transfer costs using the group sizes.

2.4 Stream Partition

Stream partitions (*recomputation*) exploit recursively decomposable polymorphic functions to feed data in batches while maintaining intermediate outputs on-chip, reducing low-level memory usage. Functions can be streamed if they are polymorphic for a specific axis (defined for that axis being of any size) and have an accumulator that can incorporate incoming data to recompute the maintained output, as shown in Figure 13. This allows for a recursive expansion (see Figure 14) that maintains minimum data on-chip at any point.

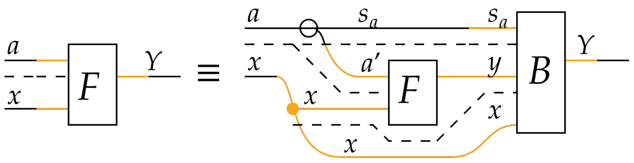


Figure 13: The condition for streaming requires that a function F be polymorphic along the axis a and can be decomposed in the manner above, requiring the existence of another polymorphic function B called the accumulator.

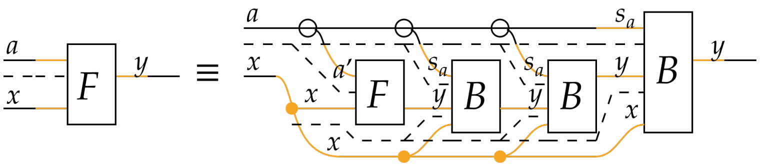


Figure 14: If the condition in Figure 13 is met, the function can be recursively decomposed until $a' \leq s_a$. This allows the function to be evaluated from batches of the input data.

If an axis originates from a transfer and is fed to a recursively decomposable polymorphic function, then it can be relabeled with the streaming batch size (stream size) s_b. This creates a representation of the sub-algorithm B which is repeatedly applied to process the data. We need to add the output size y in parentheses at the input to consider its contribution to memory usage. The memory usage of the algorithm is then determined using the stream size s_b instead of the full axis size b. As we eventually stream the entire axis, we use the full axis size b to evaluate transfer costs. Typically, we strictly benefit from limiting the stream size to 1 as this reduces memory usage while imposing no increase in transfer costs.

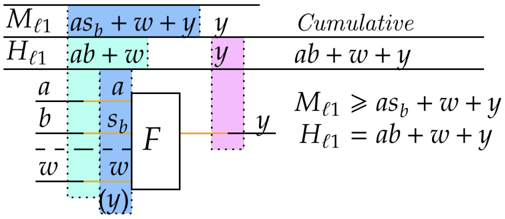


Figure 15: We can relabel a streamable axis with the batch size s_b as it is transferred. We are required to add the y output array shape at the input to the streamed algorithm. This lets the relabeled diagram derive the memory usage at the lower level using the stream size s_b. As all data along the axis is transferred to the chip, the full axis size b must be used for transfer costs.

Per the fusion theorems of Appendix A.1, streamable axes are resistant to modifications. The streamable axis may be a single axis of an array, and composing or weaving a streamable algorithm while maintaining this axis yields a streamable algorithm. This allows the stream labeling to be maintained for resource usage evaluation as in Figure 16. This allows the streamability of complex functions like attention to be derived from a streamable kernel. In Figure 17, we apply group partitioning to a mapped streamable algorithm. We use g_q for per-group transfer evaluations, and both g_q and s_b to evaluate memory usage.

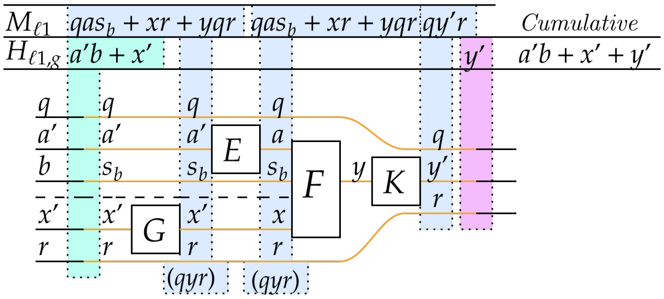


Figure 16: For a modified streamed algorithm, we can continue to use the stream batch size s_b for memory usage evaluations. As the function generates $q \times r$ distinct y values, it needs to maintain each on memory, resulting in $y \times q \times r$ maintained memory before and after the repeated E algorithm.

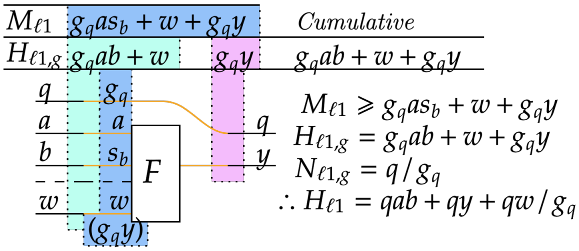
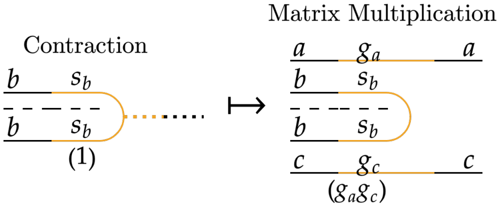

$M_{\ell 1}$
$g_q as_b + w + g_q y$
Cumulative
$H_{\ell 1, g}$
$g_q ab + w$
$g_q y$
$g_q ab + w + g_q y$
q
a
b
w
g_q
a
s_b
w
$(g_q y)$
F
q
y
$M_{\ell 1} \geqslant g_q as_b + w + g_q y$
$H_{\ell 1, g} = g_q ab + w + g_q y$
$N_{\ell 1, g} = q / g_q$
$\therefore H_{\ell 1} = qab + qy + qw / g_q$

Figure 17: We can apply multiple relabelings to an algorithm. This lets us find the per-group memory usage and transfer cost. As the function is mapped within each group, it needs to maintain g_q copies of the maintained y data, increasing its memory usage.

3 Examples

3.1 Matrix Multiplication

As contraction (dot product) is streamable (see Appendix A.3.1), we can use it as a kernel for deriving the streamability of matrix multiplication, its weaved form. This provides a diagram that supplies a performance model. We then optimize for the batch sizes to minimize total transfers given some maximum lower-level memory usage M. Unlike standard approaches (Gholami et al., 2024; Ootomo & Yokota, 2023), this performance model indicates that the transfer cost of $n = a = b = c$ matrix multiplication is cubic for $n \geq \sqrt{M}/2$.

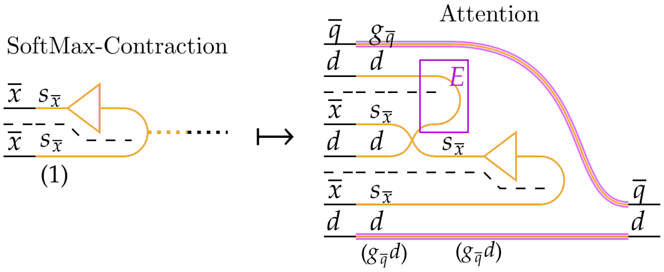


$$
\begin{aligned}
N_g &= \frac{ac}{g_a g_c} & H &= N_g H_g \\
H_g &= g_a b + b g_c + g_a g_c & &= \frac{ac}{g_a g_c}(g_a b + b g_c + g_a g_c) \\
M &\geqslant g_a g_c + g_a s_b + s_b g_c & &= 2abc\ g_a^{-1} + ac \\
\sqrt{M} &\geqslant g_c = g_a & &\geqslant 2abc\ M^{-0.5} + ac
\end{aligned}
$$

Figure 18: The dot product is a streamable function. Therefore, matrix multiplication, which is the weaved form of it, is also streamable and can be group partitioned.

3.2 Attention

We derive the streamability of attention from the fusion theorems. We begin with the fact that SoftMax-Contraction is streamable (Appendix A.4). Then, we can compose with a contraction over the queries as an E algorithm from Figure 16. This generates a streamable algorithm, which we weave with the $\overline{q}$ and d axes. This generates Figure 19. Correctness is ensured as the diagram gives the typical expression for attention, $O = \text{SoftMax}\left(Q \cdot K^T\right) \cdot V$, with axes clearly indicated. We can then label $\overline{q}$ to distribute the queries across processors, yielding the FlashAttention algorithm. Figure 19, then, can be seen as deriving and providing a performance model for FlashAttention.

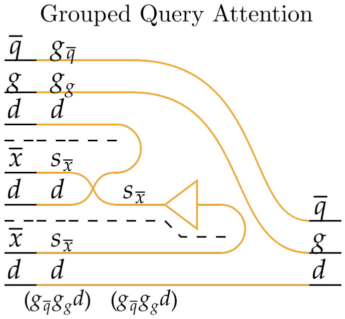


$$
\begin{aligned}
N_g &= \overline{q}/g_{\overline{q}} & H &= N_g H_g \\
H_g &= 2g_{\overline{q}}d + 2\overline{x}d & &= \frac{\overline{q}}{g_{\overline{q}}}(2g_{\overline{q}}d + 2\overline{x}d) \\
M &\geqslant 2g_{\overline{q}}d + 2s_{\overline{x}}d & &= 2\overline{q}d + 2\overline{x}d\overline{q}\ g_{\overline{q}}^{-1} \\
\therefore g_{\overline{q}} &\leqslant M/2d & &\geqslant 2\overline{q}d + 4\overline{x}\overline{q}d^2\ M^{-1}
\end{aligned}
$$

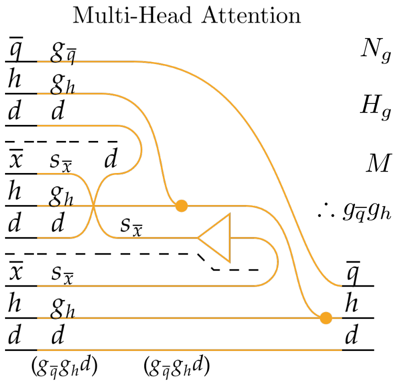
Figure 19: SoftMax followed by a contraction is streamable. We precompose with a contraction E weaved by s_x and provide weavings by $\overline{q}$ at the top and d at the bottom to construct attention.

We can apply a similar technique to find the transfer cost of grouped query attention (Ainslie et al., 2023) (Figure 20) and multi-head attention (Vaswani et al., 2017) (Figure 21). These use additional weaves, but their evaluation remains straightforward. This shows how diagrams can be used to both derive optimizations and experiment with modifications to the algorithm, motivating further innovation.

Figure 20: Grouped query attention has an additional weave accompanying the queries. This provides additional fidelity for the algorithm with minimal impact on total transfers.



$$
\begin{aligned}
N_g &= g\overline{q}/(g_{\overline{q}} g_g) & H &= N_g H_g \\
H_g &= 2g_{\overline{q}} g_g d + 2\overline{x}d & &= \frac{g\overline{q}}{g_{\overline{q}} g_g}(2g_{\overline{q}} g_g d + 2\overline{x}d) \\
M &\geqslant 2g_{\overline{q}} g_g d + 2s_{\overline{x}}d & &= 2g\overline{q}d + 2\overline{x}d\overline{q}/(g_{\overline{q}} g_g) \\
\therefore g_{\overline{q}} g_g &\leqslant M/2d & &\geqslant 2g\overline{q}d + 4\overline{x}\overline{q}d^2\ M^{-1}
\end{aligned}
$$



$$N_g = \overline{q}h/(g_{\overline{q}} g_h) \qquad H = N_g H_g$$

$$H_g = 2 g_{\overline{q}} g_h d + 2\overline{x}d \qquad = \frac{\overline{q}h}{g_{\overline{q}} g_h}(2 g_{\overline{q}} g_h d + 2\overline{x}d)$$

$$M \geqslant 2 g_{\overline{q}} g_h d + 2 s_{\overline{x}} d \qquad = 2\overline{q}hd + 2\overline{xq}hd/(g_{\overline{q}} g_g)$$

$$\therefore g_{\overline{q}} g_h \leqslant M/2d \qquad \geqslant 2\overline{q}hd + 4\overline{xq}hd^2 \; M^{-1}$$

Figure 21: Multi-head attention conducts multiple attention algorithms in parallel. It is represented by an additional h axis weaved over every operation. Notice how scaling h leads to a linear change in costs, while d leads to a quadratic change.

4 Analysis of Performance Models

Once a two-level model optimization of an algorithm is found, we can extend it to consider a multi-level hierarchy. Each lower level has tiles which fit into the level above (Figure 22), meaning the optimal strategy and performance model extend in a generalizable manner. We can create universal performance model for transfer costs which considers the impact of the GPU hierarchy and the transfer rate and memory caches at different levels. This allows us to make informed choices between different GPU architectures given their energy and capital costs, levels of quantization we employ, and the configuration of GPU hierarchies.

4.1 Optimal Transfers $H^*(\vec{a}, M)$

Applying the two-layer model to diagrams provides optimal transfer costs $H^*(\vec{a}, M)$ given some configuration of axis sizes $\vec{a}$ and lower-level memory M. So far, these expressions have a standard form given by the sum of power functions which solves for Equation 4 in Appendix A.2:

$$H^*(\vec{a}, M) = \sum_t \alpha_t(\vec{a}) \; M^{-\beta_t} \tag{1}$$

The index t iterates over terms, the coefficient $\alpha_t(\vec{a})$ is dependent on the axis sizes, and $\beta_t \geqslant 0$ is an exponent greater than zero as transfers necessarily decrease with increased memory size. The exponents β_t indicate the sensitivity of performance to memory size and indicate how data is distributed. For attention, the M^{-1} factor indicates the data is broadcast to all groups, while for matrix multiplication the $M^{-0.5}$ factor indicates square tile distribution.

4.2 Multi-Level Performance Models

An algorithm with multiple levels requires $H^*(\vec{a}, M_\ell)$ transfers for each. Even though data cannot be directly transferred from the highest to lowest levels, lower levels can utilize the data loaded and saved by intermediate levels. The execution of $H^*(\vec{a}, M_{\ell 1})$ intermediate level transfers makes data available for the lower level $\ell 2$ and accounts for saving it back. Each intermediate-level tile fits a larger number of low-level tiles, among which it distributes its data. This fitting process has a negligible error with large M. We assign a weighted transfer cost $\dot{H}_\ell^{-1}$ to each level. For the highest level, we assume $M_{\ell 0} \to \infty$ and $\dot{H}_{\ell 0}^{-1} = 0$, as data is already present. This means that the total weighted transfer cost of an algorithm can be expressed by:

$$H^* = \sum_\ell \dot{H}_\ell^{-1} \; H^*(\vec{a}, M_\ell) = \left(\sum_t \alpha(\vec{a}) \; \sum_\ell \dot{H}_\ell^{-1} \; M_\ell^{-\beta_t} \right) \tag{2}$$

Therefore, the relative performance of GPUs is determined not just by the raw transfer rates but also by the memory size of available levels and the specific algorithm being implemented. For attention, the key factor per level is $\sim \dot{H}^{-1}/M$. In Appendix A.5.3, we see that Hopper, by effectively doubling low-level memory, doubles the amount that K, V streams are shared. This is equivalent to an Ampere architecture with double the bandwidth, highlighting the importance of low-level memory and architecture features.

4.3 Quantization

Equation 2 and the two-level model consider transfers and storage limits in terms of number of values. However, GPUs are restricted by the number of bytes we can transfer and store. If we have q bytes per value, then the maximum number of values $M_\ell = M_\ell^{\text{Bytes}}/q$ and the transfer weight is $\dot{H}_\ell^{-1} = (\dot{H}_\ell^{\text{Bytes}}/q)^{-1}$. Substituting these expressions into the total transfer cost, we get:

$$\begin{aligned} H^{*\text{Bytes}} &= \sum_\ell (\dot{H}_\ell^{\text{Bytes}}/q)^{-1} \; H^*\left(\vec{a}, M_\ell^{\text{Bytes}}/q\right) \\ &= \left(\sum_t \alpha(\vec{a}) \; \sum_\ell (\dot{H}_\ell^{\text{Bytes}})^{-1} \; (M_\ell^{\text{Bytes}})^{-\beta} \; q^{1+\beta}\right) \end{aligned} \tag{3}$$

As $1 + \beta \geqslant 1$, total transfers are superlinear to the degree of quantization. Halving the quantization from $FP32$ to $FP16$ can accelerate attention by up to ×4, and improves large matrix multiplication by a factor of $2^{1.5} \approx 2.83$. This indicates that a generous use of quantization and specialized kernels is critical to high-throughput implementations of models.

4.4 Intermediate Caching

We can choose to store output data at lower levels, and save it up in chunks. This changes the level immediately above the lower level to a caching level, which we indicate by adding asterisks to its output data as in Figure 22. The size of this column is not memory restricted by the intermediate level which is only used to temporarily store data as it is sent up. However, the lower levels must remain active to store data, and this imposes the restriction that $N_{g,\ell 2}/N_{g,\ell 1} \leqslant N_{\ell 2}^{\max}$ which is a hardware limit. With an output restricted algorithm, this results in $H^*\left(\vec{a}, N_{\ell 2}^{\max} M_{\ell 2}\right)$ transfers being required for the intermediate level $\ell 1$, using the total lower level memory $N_{\ell 2}^{\max} M_{\ell 2}$ instead of its own hardware maximum memory $M_{\ell 1}^{\max}$. This is elaborated in Appendix A.2.1.

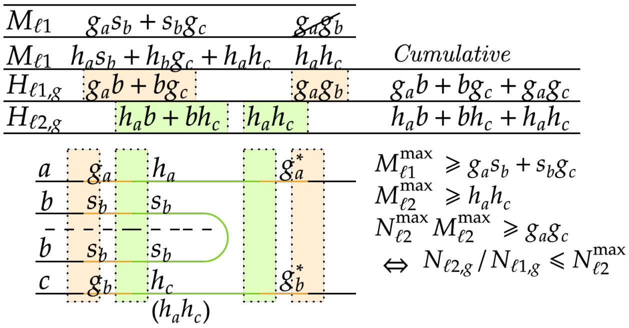
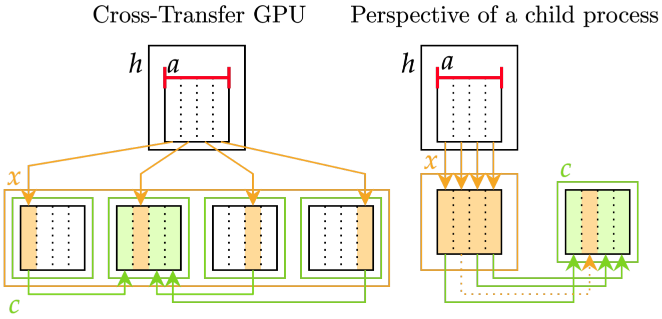

$M_{\ell 1}$
$g_a s_b + s_b g_c$
$\cancel{g_a g_b}$
$M_{\ell 1}$
$h_a s_b + h_b g_c + h_a h_c$
$h_a h_c$
Cumulative
$H_{\ell 1,g}$
$g_a b + b g_c$
$g_a g_b$
$g_a b + b g_c + g_a g_c$
$H_{\ell 2,g}$
$h_a b + b h_c$
$h_a h_c$
$h_a b + b h_c + h_a h_c$
a
g_a
h_a
g_a^*
b
s_b
s_b
b
s_b
s_b
c
g_b
h_c
g_b^*
$(h_a h_c)$
$M_{\ell 1}^{\max} \geqslant g_a s_b + s_b g_c$
$M_{\ell 2}^{\max} \geqslant h_a h_c$
$N_{\ell 2}^{\max} M_{\ell 2}^{\max} \geqslant g_a g_c$
$\Leftrightarrow \; N_{\ell 2,g} / N_{\ell 1,g} \leqslant N_{\ell 2}^{\max}$

Figure 22: With multiple levels in a hierarchy, we can maintain output or streamed data at the lowest levels and use the intermediate levels as a cache. The cache size does not contribute to the $M_{\ell 1}^{\max}$ restriction. Instead, we require that $N_{\ell 2,g}/N_{\ell 1,g} \leqslant N_{\ell 2}^{\max}$. If an algorithm is output restricted, we can implement this by the cache size being less than $M_\ell = M_{\ell 1} N_{\ell 2}^{\max}$.

4.5 Cross-Transfer Levels

Our model can encompass levels that perform inter-core communication instead of providing shared memory by using modified weighted transfer weights. These cross-transfer levels encompass H100 thread block clusters (Luo et al., 2024), multi-GPU systems, and intra-warp communication. We set up h as a higher level, x as a cross-transfer level, and c as a child level composed of linked processors. Instead of sending $H^*(\vec{a}, M_c)$ data directly to children, we send $H^*(\vec{a}, M_c N_c^{\max})$ data to any of the interconnected children and cross-transfer the remaining data as in Figure 23. This results in a performance model with modified transfer weights and levels, adding a level x between h and c with transfer weight $\dot{H}_{h \to c}^{-1} - \dot{H}_{x \to c}^{-1}$ and memory $M_c N_c^{\max}$, and replacing the transfer weight of level c with $\dot{H}_{x \to c}^{-1}$. We outline this derivation in the Appendix A.2.2.

Figure 23: To send data to children, we directly transfer $H^*(\vec{a}, M_c N_c^{\max})$ distributed across the child processors, treating the cross-transfer level as having memory of size $M_c N_c^{\max}$. We then perform the remaining $H^*(\vec{a}, M_c) - H^*(\vec{a}, M_c N_c^{\max})$ transfers as fast cross-transfers.



5 Pseudocode and Hardware Optimizations

The abstract models we have provided so far hint at optimal algorithms and provide resource usage scaling laws at greater resolution than the theorems from FlashAttention (Dao et al., 2022). Shifting from an abstract model to applications requires considering specific batch-size configurations and expanding relabeled stream diagrams into looped diagrammatic pseudocode. We use abstract models as a guide to the ideal size of axes, and then impose that they should be integers divisible by certain sizes to take advantage of coalesced memory access and tensor cores. We can expand streamed diagrams into loops where all variables and the accumulator B are fully expressed, allowing for fine-tuned configuration of batch sizes and the exploitation of Hopper overlapped computation.

5.1 Coalesced Memory Transfer

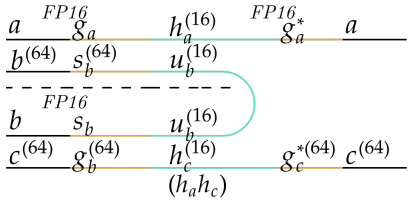
Between the device and SMEM memory, GPUs move data in chunks of $128B$ of consecutive memory. This is remarkably straightforward to represent with diagrams. Arrays represent how data is distributed across each stride, so the lowermost axis of an array represents consecutively stored memory. If we enforce that the lowermost axis is divisible by $128B/q$ when assembled in the device and transferred between GMEM and SMEM, then we can assure coalesced memory access. This may require that each thread block stream loads data for multiple lower-level streams. If using SMEM as a cache, there is usually plentiful memory available for larger streams.

A floating divisor in the superscript of an axis/batch size is used to indicate a value it is divisible by (see Figure 24). This is done at the point where the restriction is imposed and along the immediately weaved axis. Multiple divisors impose the least common multiple.

5.2 Tensor Core Operations

Tensor cores provide very fast matrix multiplication. Modern GPU tensor cores have far more FLOPs available than general-purpose register operations (NVIDIA, 2020; 2022). We can re-express matrix multiplications as tensor core operations. This requires saving to and loading from SMEM memory if data is initially at the register level. Tensor cores (`wmma`) can only manage data at certain sizes and quantizations (NVIDIA, 2024), which must be considered by diagrams. We can add a floating tag to indicate quantization. Matrix multiplications of larger sizes can be implemented by adding multiple smaller matrix multiplications, making divisibility by the available sizes the critical factor. We can enforce this restriction by placing superscripts for tensor core axes.

Figure 24: Multi-level matrix multiplication uses the SMEM level to cache data for lower-level tensor core operations. We enforce the divisibility restrictions for coalesced SMEM transfers and tensor cores using superscripts.



5.3 From Diagrams to Pseudocode

We can expand streamed algorithms into looped pseudocode forms where all variables are explicitly shown as in Figure 25. The columns of pseudocode diagrams provide the size of variables required in memory and the transfers/operations we need to apply. This allows us to pre-allocate memory to derive the exact memory usages, as well as per-group transfer and compute costs. Columns act like lines of code but more clearly express the details of axes and available optimization strategies than textual methods. As polymorphic streamed algorithms are defined for the stream axis being of any size, we can begin the algorithm with a head F taking an axis of size 0, initializing the maintained output to incorporate further information.

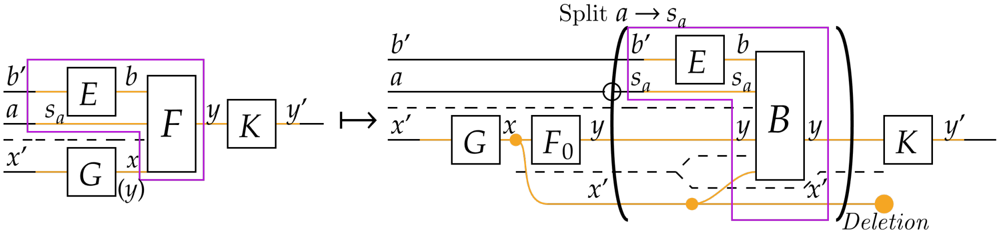


Figure 25: A streamed algorithm can be re-expressed with an explicit loop. The diagram illustrates how axes are partitioned, which variables are maintained, and what operations are applied iteratively.

We can use these diagrams to express the exact details of algorithms like Ampere attention in Figure 26. Pseudocode expansions allow us to transfer to and from fragmented memory to execute both tensor-core and general operations. This requires us to use exactly 32 times as many thread groups as tensor core groups. Furthermore, we can place dotted enclosed sub-loops. We add a circle to incoming or outgoing axes, which are iterated over the original size within the sub-loop. This constructs matrix-multiply add operations and lower-level substreams while accurately presenting the size of required variables, and imposes a divisor constraint.

As maintaining a diagram's shape ensures correctness, pseudocode expressions allow extensive tinkering. We present an Ampere Attention algorithm in Figure 26. Unlike FlashAttention-2 (Dao, 2023), we use registers to store and scale the maintained variable instead of tensor cores. We use smaller sub-loop stream sizes $s_{\overline{x}} \to u_{\overline{x}}^{(8)}$ and $d \to 16$ to reduce register memory usage. Extending this algorithm to grouped query attention and multi-head attention simply involves weaving in additional axes, as we did in Figures 20 and 21. In Appendix A.5.2, we derive the memory usage for our algorithm and show that it can fit 13 warps per thread block instead of FlashAttention-2's 4-8 warps.

The Ampere diagram hints at the techniques of FlashAttention-3 (Shah et al., 2024), which exploits hardware features of the H100 Hopper architecture. Hopper allows for warpgroups, enabling 128 thread groups per tensor core and storing some of their memory on SMEM. Hopper enables explicit asynchrony, allowing different warpgroups to simultaneously execute different operations. We allocate some warp groups to load data (producer-consumer asynchrony). Distinct processors on the SM execute different operations, so we can stagger different warpgroups to simultaneously execute green and blue columns. We can divide compute cost per column by operations per clock cycle to determine ideal overlapping. In Figure 27, we show our Hopper Attention. In Appendix A.5.3, we show the required memory and the staggering strategy indicated by the diagram. This algorithm can theoretically achieve close to the maximum 1.34 PFLOPs of H100 SXM5 compute.

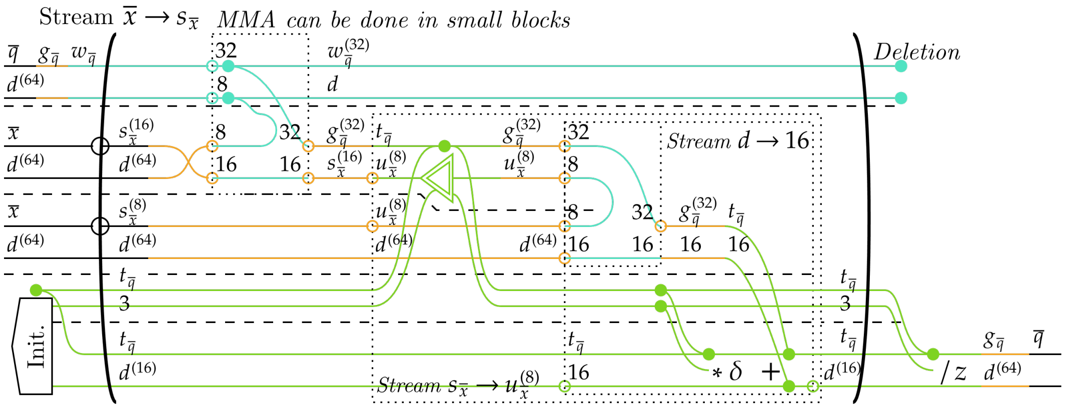

Stream $\overline{x} \to s_{\overline{x}}$
MMA can be done in small blocks
Deletion
Stream $d \to 16$
Stream $s_{\overline{x}} \to u_{\overline{x}}^{(8)}$
Init.
$\overline{q}$ $g_{\overline{q}}$ $w_{\overline{q}}$
$d^{(64)}$
$\overline{x}$
$s_{\overline{x}}^{(16)}$
$s_{\overline{x}}^{(8)}$
$w_{\overline{q}}^{(32)}$
$g_{\overline{q}}^{(32)}$
$t_{\overline{q}}$
$u_{\overline{x}}^{(8)}$
$d^{(16)}$
$*\ \delta$ $+$
$/z$

Figure 26: Ampere Attention shown using pseudocode. The diagram begins by distributing Q values to tensor cores, then each loop loads and computes a streamed portion of K and V. The sub-loop splits the streamed axes into smaller chunks. Certain operations (SMEM loads, constructed matrix multiply-add) impose divisor restrictions, which we place. We show required memory allocations in Appendix A.5.2.

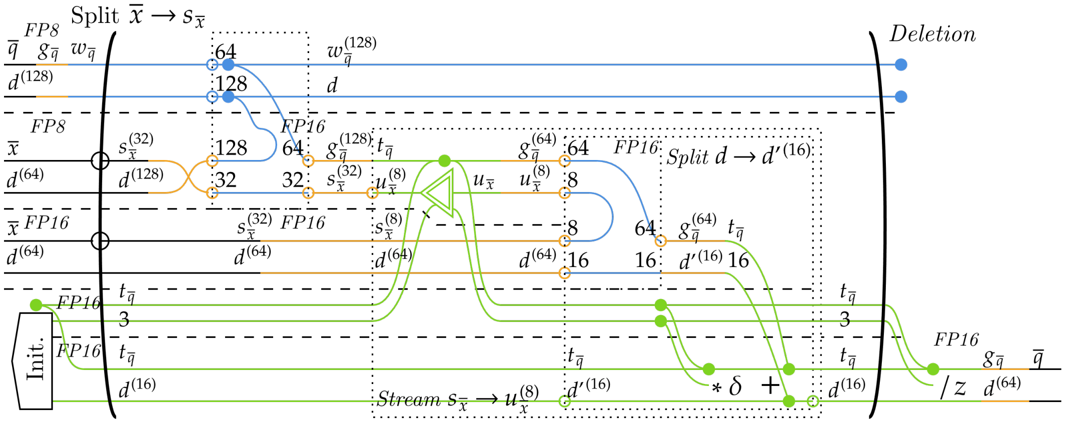

Split $\overline{x} \to s_{\overline{x}}$
FP8
FP16
Deletion
Split $d \to d'^{(16)}$
Stream $s_{\overline{x}} \to u_{\overline{x}}^{(8)}$
Init.
$\overline{q}$ $g_{\overline{q}}$ $w_{\overline{q}}$
$d^{(128)}$
$s_{\overline{x}}^{(32)}$
$w_{\overline{q}}^{(128)}$
$g_{\overline{q}}^{(128)}$
$g_{\overline{q}}^{(64)}$
$u_{\overline{x}}$
$d'^{(16)}$
$*\ \delta$ $+$
$/z$

Figure 27: Hopper attention incorporates larger tensor core groups and $FP8$ for specific operations. The algorithm is divided into producer and consumer workflows, as indicated by the separation between active green and blue columns, enabling staggered warp group operations.

6 Conclusions

In this work, we have used diagrams to derive, analyze, and fine-tune optimized deep learning algorithms. The complexities of generating optimized algorithms: tiling, streaming, and applying hardware features, are reduced to simple relabelings of diagrams. This vastly improves over existing manual derivations.

This work also compels future research. The performance model and the hypothesized algorithm remain to be empirically validated. Mixture-of-expert models (Jiang et al., 2024) use immense resources and therefore optimizations are particularly impactful. Additional strategies can be formalized. Convolution and sliding window attention (Beltagy et al., 2020) reindex weavings (Abbott, 2023), which changes the amount of data accessed per group. We can use accumulators to congregate data processed on different cores, which is required to parallelize streamable algorithms weaved over a small axis.

Furthermore, diagrams conform to a category-theoretic description, which is not covered in this paper. However, a categorical perspective would allow back propagation (Fong et al., 2019; Cruttwell et al., 2021), compilation (Wilson, 2023), and error propagation (Perrone, 2022) to be understood. Formalizing the categorical aspects of this work would integrate it into existing research and provide a systematic framework for expressing, optimizing, back-propagating, and compiling deep learning models.

The performance model we provide considers both the characteristics of algorithms and the hardware they run on. This performance model can incorporate increasing fidelity, all the way down to the clock cycles per operation. This invites a systematic analysis of hardware design that relates requirements (transistor count, energy usage, chip area, production reliability) to functionalities (compute, available memory, bandwidth), which can be systematically conducted using categorical co-design (Zardini, 2023). This would allow us to use a shared mathematical framework for describing and optimizing deep learning algorithms and designing the hardware they run on.

References

Vincent Abbott. Robust diagrams for deep learning architectures: Applications and theory, 2023. URL https://vtabbott.io/honours-thesis.

Vincent Abbott. Neural Circuit Diagrams: Robust Diagrams for the Communication, Implementation, and Analysis of Deep Learning Architectures, 2023. URL https://openreview.net/forum?id=RyZB4qXEgt.

Alok Aggarwal and Jeffrey Vitter, S. The input/output complexity of sorting and related problems. 31 (9):1116–1127, 1988. ISSN 0001-0782. doi: 10.1145/48529.48535. URL https://dl.acm.org/doi/10.1145/48529.48535.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, et al. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL http://arxiv.org/abs/2305.13245.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL http://arxiv.org/abs/2004.05150. version: 2.

G. S. H. Cruttwell, Bruno Gavranović, Neil Ghani, et al. Categorical foundations of gradient-based learning, 2021. URL http://arxiv.org/abs/2103.01931.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning, 2023. URL http://arxiv.org/abs/2307.08691.

Tri Dao, Daniel Y. Fu, Stefano Ermon, et al. FlashAttention: Fast and memory-efficient exact attention with IO-awareness, 2022. URL http://arxiv.org/abs/2205.14135.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The llama 3 herd of models, 2024. URL http://arxiv.org/abs/2407.21783.

Patrick Esser, Sumith Kulal, Andreas Blattmann, et al. Scaling rectified flow transformers for high-resolution image synthesis, 2024. URL http://arxiv.org/abs/2403.03206.

Brendan Fong, David I. Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning, 2019. URL http://arxiv.org/abs/1711.10455.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers, 2023. URL http://arxiv.org/abs/2210.17323.

Amir Gholami, Sehoon Kim, Zhen Dong, et al. A survey of quantization methods for efficient neural network inference, 2021. URL http://arxiv.org/abs/2103.13630.

Amir Gholami, Zhewei Yao, Sehoon Kim, et al. AI and memory wall, 2024. URL http://arxiv.org/abs/2403.14123.

Saugata Ghose, Abdullah Giray Yağlıkçı, Raghav Gupta, et al. What your DRAM power models are not telling you: Lessons from a detailed experimental study, 2018. URL http://arxiv.org/abs/1807.05102.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, et al. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, et al. Data movement is all you need: A case study on optimizing transformers, 2021. URL http://arxiv.org/abs/2007.00072.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, et al. Mixtral of experts, 2024. URL http://arxiv.org/abs/2401.04088.

Mingzhen Li, Yi Liu, Xiaoyan Liu, et al. The deep learning compiler: A comprehensive survey, 2020. URL http://arxiv.org/abs/2002.03794.

Weile Luo, Ruibo Fan, Zeyu Li, et al. Benchmarking and dissecting the NVIDIA Hopper GPU architecture, 2024. URL http://arxiv.org/abs/2402.13499. version: 1.

NVIDIA. NVIDIA a100 tensor core GPU architecture overview, 2020. URL https://docs.nvidia.com/cuda/pdf/ptx_isa_8.5.pdf.

NVIDIA. NVIDIA h100 tensor core GPU architecture overview, 2022. URL https://resources.nvidia.com/en-us-tensor-core.

NVIDIA. PTX ISA 8.5, 2024. URL https://docs.nvidia.com/cuda/pdf/ptx_isa_8.5.pdf.

NVIDIA. CUDA C++ Programming Guide, 2024. URL https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Hiroyuki Ootomo and Rio Yokota. Reducing shared memory footprint to leverage high throughput on tensor cores and its flexible API extension library, 2023. URL http://arxiv.org/abs/2308.15152.

Adam Paszke, Sam Gross, Francisco Massa, et al. PyTorch: An imperative style, high-performance deep learning library, 2019. URL http://arxiv.org/abs/1912.01703.

Paolo Perrone. Markov categories and entropy, 2022. URL http://arxiv.org/abs/2212.11719.

Dustin Podell, Zion English, Kyle Lacey, et al. SDXL: Improving latent diffusion models for high-resolution image synthesis, 2023. URL http://arxiv.org/abs/2307.01952.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, et al. High-resolution image synthesis with latent diffusion models, 2022. URL http://arxiv.org/abs/2112.10752.

Amit Sabne. Xla: Compiling machine learning for peak performance. 2020. URL http://research.google/pubs/xla-compiling-machine-learning-for-peak-performance/.

Jay Shah, Ganesh Bikshandi, Ying Zhang, et al. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL http://arxiv.org/abs/2407.08608.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. URL http://arxiv.org/abs/1911.02150.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19. Association for Computing Machinery, 2019. ISBN 978-1-4503-6719-6. doi: 10.1145/3315508.3329973. URL https://dl.acm.org/doi/10.1145/3315508.3329973.

Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, et al. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Paul William Wilson. Category-theoretic data structures and algorithms for learning polynomial circuits, 2023. URL https://eprints.soton.ac.uk/483757/.

Gioele Zardini. Co-design of complex systems: From autonomy to future mobility systems, 2023. URL https://www.research-collection.ethz.ch/handle/20.500.11850/648075. Accepted: 2023-12-19T10:03:57Z.

A Appendix

A.1 Fusion Theorem

Theorem 1 (Fusion Theorem) *Composition and weaving of a streamable algorithm which does not remove the streamed axis yields a streamable algorithm.*

Streamable algorithms satisfies the form in Figure 28, allowing the recursive expansion of Figure 29. Streamability depends on polymorphism over the a axis, meaning the function/algorithm is defined for the axis being of any size, and the existence of an accumulator B, which allows the streamed input data to be split.

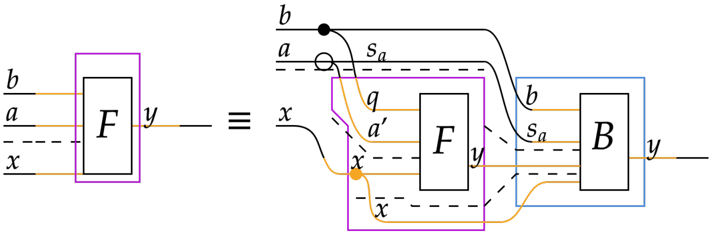


Figure 28: A streamable algorithm requires an accumulator B which allows the polymorphic streamable axis to be split. We can fuse the algorithm with a head and tail, which do not require additional loads and saves if their memory usage is sufficiently small.

This allows for recursive decomposition, reducing the size of the remainder a' axis to $a' \leq s_a$ as in Figure 29. Here, we add a head and a tail, which are not expanded but can be executed without an additional transfer and are hence fused. This requires their memory usage to be sufficiently small to not exceed hardware limitations. Composition on G or K simply replaces those algorithms with the composed form, yielding a modified head/tail for the streamable algorithm.

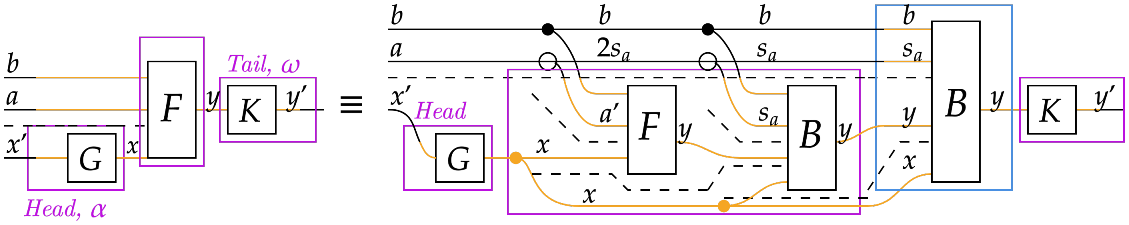
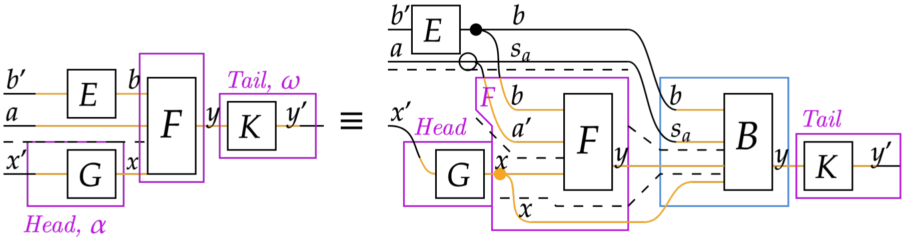
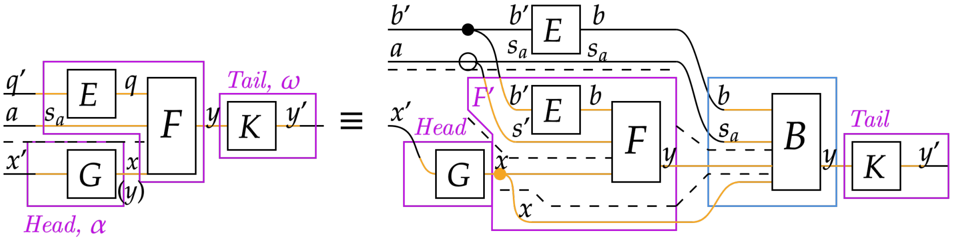
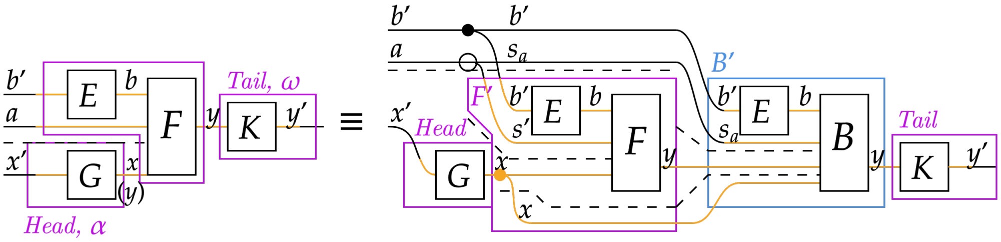


Figure 29: If Figure 28 is satisfied, then the algorithm can be recursively decomposed. This reduces the size of chunks until the on-chip memory is sufficiently small.

Composing by E on the b-axis with an algorithm weaved by the streamable s_a axis, we can exploit a partition copy (see Figure 10) to show that the composed F' has an accumulator B'.







Finally, we are required to show that weaving preserves streamability. This exploits a characteristic of mapping composed functions. Mapping a composed function over an additional axis is equivalent to composing the individual functions mapped over that axis. Therefore, we can show that a weaved B is an accumulator for a weaved F as in Figure 30.

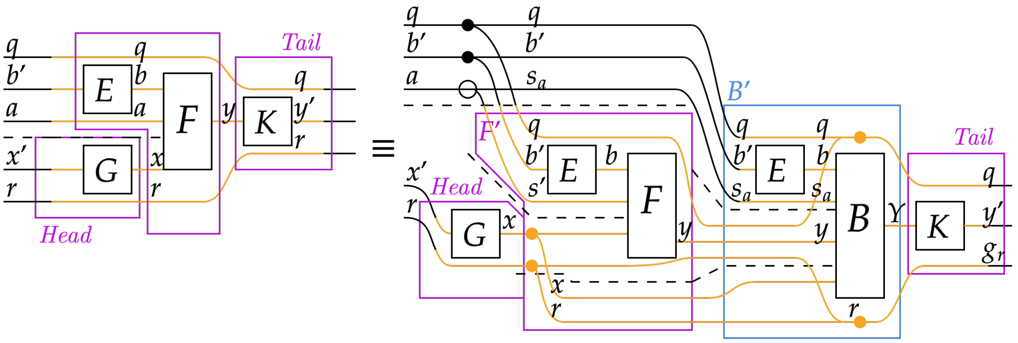


Figure 30: Weaving over F composed with E and G at its inputs is equivalent to weaving over the smaller F and B algorithms which, when composed, give F. This weaves need to trace over the inputs which they target.

We can combine all the above expressions into the single form of Figure 31, where we also apply group partitioning. This separates the mapped axis into groups of size g_q distributed across processors. We can iteratively apply these rules; an algorithm can be composed with an algorithm E weaved over the streamable axis. This generates a streamable algorithm, which can be weaved over one of the inputs introduced by E. This is used to construct streamable (flash) attention from a SoftMax-Contraction kernel.

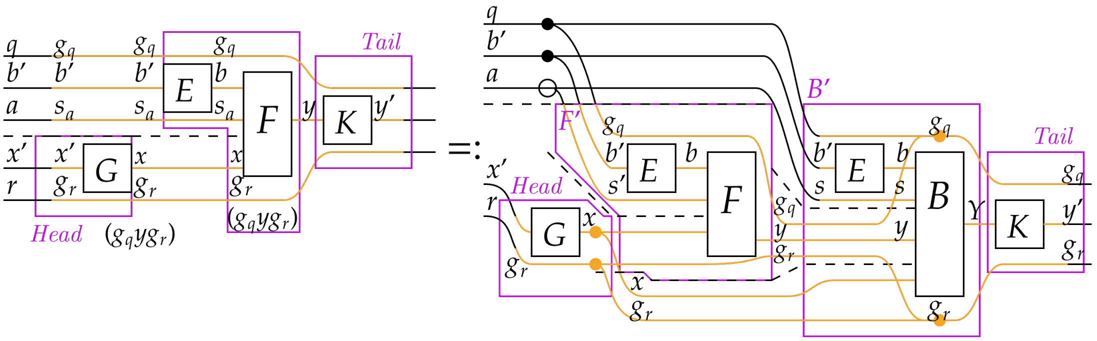


Figure 31: We can combine the streaming theorems into a single expression. Given that the algorithm F is streamable along the a/s_a axis, we can add modifications to generate a new streamable algorithm. We can group partition along the streamed axes, mapping groups of size $g_q \times g_r$ to different processors.

A.2 Multi-Level Performance Models

We define for a two-level model diagram the optimal transfers $H^*(\vec{a}, M)$, where $\vec{a}$ are the axis sizes and M is the maximum memory available at the lower level. We use $\vec{g}$ to indicate some configuration of group sizes. We are interested in memory usage in the limit of large M and $\vec{g}$, in which case the limiting factor is some $y\ \Pi_i\ g_i$ weaved by all grouped axes, with y typically being the output. For smaller M, we aim for a specific configuration as in Section 5. Two-level diagrams derived optimal transfers solve for Equation 4, where i iterates over the grouped axes:

$$H^*(\vec{a}, M) = \min_{\vec{g}}\ \frac{\Pi_i\ a_i}{\Pi_i\ g_i}\ H_g(\vec{a}, \vec{g}) \text{ given } M \geqslant y\ \Pi_i\ g_i \tag{4}$$

We extend this to multiple levels be assigning a memory M_ℓ to each level and a weighted transfer cost $\dot{H}_\ell^{-1}$, which represents bandwidth. As described in Section 4, this conforms to;

$$H^* = \sum_\ell\ \dot{H}_\ell^{-1}\ H^*(\vec{a}, M_\ell) \tag{5}$$

How additional levels are used, either as intermediate caches or cross-transfer levels, use the same general performance model of Equation 5 but with altered weighted transfer costs $\dot{H}_\ell^{-1}$ and per-level effective memory M_ℓ. Using $M_\ell^{\max}$ to indicate the maximum memory per level, $N_\ell^{\max}$ to indicate the maximum number of child nodes per higher level node, h for the immediately higher level and c for the immediately lower, and $\dot{H}_{\ell\to\ell'}^{-1}$ as the raw weighted transfer cost between ℓ and ℓ', we adapt our performance model to get;

$$\dot{H}_\ell^{-1} = \begin{cases} \dot{H}_{h\to c}^{-1} - \dot{H}_{\ell\to c}^{-1} & \ell \text{ is a cross-transfer level} \\ \dot{H}_{h\to\ell}^{-1} & \text{else} \end{cases}$$

$$M_\ell = \begin{cases} N_c^{\max} M_c & \ell \text{ is a cross-transfer or intermediate cache} \\ M_\ell^{\max} & \text{else} \end{cases}$$

Therefore, using cross-transfer or caching levels conforms to our standard performance models. This lets # act as a universal performance model for multi-level GPUs, and ensures that $\dot{H}_\ell^{-1}\ M_\ell^{-\beta}$ is the characteristic property for comparing various GPU architectures.

A.2.1 Intermediate Caching

Theorem 2 *An intermediate caching level $\ell 1$ for an output restricted algorithm with a number restriction $N_{\ell 2}^{\max} \geqslant N_{g,\ell 2}/N_{g,\ell 1}$ conforms to the standard performance model with $M_{\ell 1} = M_{\ell 2} N_{\ell 2}^{\max}$, requiring $H^*(\vec{a}, M_{\ell 2} N_{\ell 2}^{\max})$ total transfers.*

For an intermediate caching model which is output limited, we have the standard constraint for the lower level, $M_{\ell 2} \geqslant y\ \Pi_i\ g_{\ell 2,i}$, and the number restriction, $N_{\ell 2}^{\max} \geqslant N_{g,\ell 2}/N_{g,\ell 1} = \Pi_i\ g_{\ell 1,i}/\Pi_i\ g_{\ell 2,i}$. We aim to find the configuration of $\vec{g}_{\ell 1}$ and $\vec{g}_{\ell 2}$ which minimizes the number of transferred values. Assuming the algorithm is output limited, the effective size of the caching column is $y\ \Pi_i\ g_i^{\ell 1}$. This lets us express the restrictions as:

$$H_{\ell 1}^* = \min_{\vec{g}}\ \frac{\Pi_i\ a_i}{\Pi_i\ g_{\ell 1,i}}\ H_g(\vec{a}, \vec{g}_{\ell 1}) \qquad \text{given } N_{\ell 2}^{\max} \geqslant \Pi_i\ g_{\ell 1,i}/\Pi_i\ g_{\ell 2,i} \tag{6}$$

$$H_{\ell 2}^* = \min_{\vec{g}}\ \frac{\Pi_i\ a_i}{\Pi_i\ g_{\ell 2,i}}\ H_g(\vec{a}, \vec{g}_{\ell 2}) \qquad \text{given } M_{\ell 2} \geqslant y\ \Pi_i\ g_{\ell 2,i} \tag{7}$$

We can multiply the restriction of (6) by the restriction of (7) to get (8). Assuming that the inequalities are sufficiently close to equalities, we outline the new problem to solve:

$$\begin{aligned} H^*_{\ell 1} &= \min_{\vec{g}} \frac{\Pi_i\ a_i}{\Pi_i\ g_{\ell 1,i}}\ H_g(\vec{a}, \vec{g}_{\ell 1}) && \text{given } M_{\ell 2} N^{\max}_{\ell 2} \geqslant y\ \Pi_i\ g_{\ell 1.i} && (8) \\ H^*_{\ell 2} &= \min_{\vec{g}} \frac{\Pi_i\ a_i}{\Pi_i\ g_{\ell 2,i}}\ H_g(\vec{a}, \vec{g}_{\ell 2}) && \text{given } M_{\ell 2} \geqslant y\ \Pi_i\ g^{\ell 2}_i && (9) \end{aligned}$$

These restrictions correspond to Equation 4, so we can substitute in $H^*(\vec{a}, M_\ell)$ for both levels, but using $M_{\ell 2} N^{\max}_{\ell 2}$ for the intermediate level instead of its own maximum memory, $M^{\max}_{\ell 1}$. We therefore have;

$$\begin{aligned} H^*_{\ell 1} &= H^*(\vec{a}, M_{\ell 2} N^{\max}_{\ell 2}) \\ H^*_{\ell 2} &= H^*(\vec{a}, M_{\ell 2}) \end{aligned}$$

A caching level therefore conforms to our standard multi-level performance model derived from a two-level diagram, but with the intermediate level using total lower level memory instead of its own.

A.2.2 Cross-Transfer Level

Theorem 3 *Introducing a cross-transfer level x between a higher level h and a lower level c allows us to replace the weighted transfer cost of an output-limited algorithm;*

$$H^* = \ldots\ + \dot{H}^{-1}_{h \to c} H^*(\vec{a}, M_c) +\ \ldots$$

With,

$$H^* = \ldots\ +\ (\dot{H}^{-1}_{h \to c} - \dot{H}^{-1}_{x \to c})\ H^*(\vec{a}, N^{\max}_c M_c) + \dot{H}^{-1}_{x \to c} H^*(\vec{a}, M_c) +\ \ldots$$

Where $\dot{H}^{-1}_{h \to c}$ is the weighted transfer cost of sending data to children, and $\dot{H}^{-1}_{x \to c}$ is the weighted transfer cost of sending data between children.

The child level c requires a total of $H^*(\vec{a}, M_c)$ data transfers from the higher level, typically incurring a weighted transfer cost per value of $\dot{H}^{-1}_{h \to c}$. With a cross-transfer level x between h and c, we can send some data H^*_x to any of the children and cross-transfer the remaining at a weighted transfer cost of $\dot{H}^{-1}_{x \to c}$ per value. We need to derive the optimal configuration of $\vec{g}_x$ to minimize H^*_x. This configuration incurs a number restriction, as the child processors need to remain active. We therefore have,

$$\begin{aligned} H^*_x &= \min_{\vec{g}} \frac{\Pi_i\ a_i}{\Pi_i\ g_{x,i}}\ H_g(\vec{a}, \vec{g}_x) && \text{given } N^{\max}_c \geqslant \Pi_i\ g_{x,i} / \Pi_i\ g_{c,i} \\ H^*_c &= \min_{\vec{g}} \frac{\Pi_i\ a_i}{\Pi_i\ g_{c,i}}\ H_g(\vec{a}, \vec{g}_c) && \text{given } M_c \geqslant y\ \Pi_i\ g_{c,i} \end{aligned}$$

We perform a similar substitution to Section A.2.1, yielding a new set of restrictions;

$$\begin{aligned} H^*_x &= \min_{\vec{g}} \frac{\Pi_i\ a_i}{\Pi_i\ g_{x,i}}\ H_g(\vec{a}, \vec{g}_x) && \text{given } M_c N^{\max}_c \geqslant y\ \Pi_i\ g_{x,i} \\ H^*_c &= \min_{\vec{g}} \frac{\Pi_i\ a_i}{\Pi_i\ g_{c,i}}\ H_g(\vec{a}, \vec{g}_c) && \text{given } M_c \geqslant y\ \Pi_i\ g_{c,i} \end{aligned}$$

These restrictions conform to the two-level model optimal, with required transfers being $H^*_x = H^*(\vec{a}, M_c N^{\max}_c)$ and $H^*_c = H^*(\vec{a}, M_c)$. When considering transfers for the total weighted transfer cost calculation, we can subtract the required transfers to x from the transfers required to c. This is because data is transferred from the higher level to the cross-transfer level by sending it to children, so much of the data is already available. This results in the substituting the total weighted transfer costs (10) with (11):

$$\begin{aligned} H^* &= \ldots\ + \dot{H}^{-1}_{h \to c} H^*(\vec{a}, M_c) +\ \ldots && (10) \\ \mapsto H^* &= \ldots\ + \dot{H}^{-1}_{h \to c} H^*(\vec{a}, N^{\max}_c M_c) + \dot{H}^{-1}_{x \to c} \left(H^*(\vec{a}, M_c) - H^*(\vec{a}, N^{\max}_c M_c)\right) +\ \ldots && (11) \end{aligned}$$

We can rearrange (11) so that we have the standard format of each level having $H^*(\vec{a}, M_\ell)$ transfers by instead modifying the transfer cost:

$$H^* = \dots\ + \left(\dot{H}^{-1}_{h\to c} - \dot{H}^{-1}_{x\to c}\right) H^*(\vec{a}, N_c^{\max} M_c) + \dot{H}^{-1}_{x\to c}\ H^*(\vec{a}, M_c) +\ \dots \tag{12}$$

Therefore, a cross-transfer level conforms to the standard performance model. Instead of using the weighted transfer cost of sending data to the children $\dot{H}^{-1}_{h\to c}$ for the cross-transfer level, we set $\dot{H}^{-1}_x = \dot{H}^{-1}_{h\to c} - \dot{H}^{-1}_{x\to c}$ and we remap $\dot{H}_c = \dot{H}^{-1}_{x\to c}$. This lets us express (12) as the expression below, which conforms to the standard multi-level performance model of (5):

$$H^* = \dots\ + \dot{H}^{-1}_x\ H^*(\vec{a}, N_c^{\max} M_c) + \dot{H}^{-1}_c\ H^*(\vec{a}, M_c) +\ \dots$$

A.2.3 Additional Notes

In the case of a multi-GPU hierarchy, the interconnected topology is a cross-transfer level x which distributes data among child GPUs c at a weighted transfer cost of $\dot{H}^{-1}_{x\to c}$. If we assume data is already distributed across GPUs, then the number of GPU cross-transfers is $H^*(\vec{a}, M_c^{\max}) - H^*(\vec{a}, N_c^{\max} M_c^{\max})$. So far, we have considered the highest level to have unlimited memory and zero weighted transfer cost. We can model multi-GPU systems by the highest level (the multi-GPU level x) as having a memory equal to $N_c^{\max} M_c^{\max}$ and negative weighted transfer cost $-\dot{H}^{-1}_{x\to c}$, which assumes data is already distributed among GPUs. This provides a rough model, which can be refined by aligning the group sizes used in diagrams.

Often, we can configure the number of cross-transfer level children to alter the cross-transfer bandwidth. In (Luo et al., 2024), it was found that the bandwidth of H800 (*Chinese market Hopper GPUs*) SM-SM transfers varies from $3.27TB/s$ with a cluster size $N = 2$ to $2.65TB/s$ with a cluster size of $N = 4$, compared to $2.04TB/s$ of GMEM-SMEM bandwidth. This imposes a trade-off; smaller cluster sizes improve effective $\dot{H}_c$ but reduce the cross-transfer discount $H^*(\vec{a}, N_c^{\max} M_c^{\max})$. (Luo et al., 2024) notes that balancing this trade-off is an important direction for exploration. We can use our model to find the difference in weighted transfer costs with (11) and without (10) a cross-transfer level, providing an equation to optimize for N.

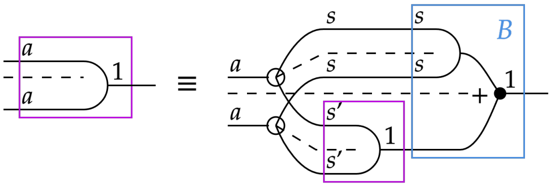
$$\begin{aligned}
\Delta H^* &= \left(\dots\ + \dot{H}^{-1}_{h\to c} H^*(\vec{a}, M_c) +\ \dots\right) \\
&\quad - \left(\dots\ + \left(\dot{H}^{-1}_{h\to c} - \dot{H}^{-1}_{x\to c}\right) H^*(\vec{a}, N\ M_c) + \dot{H}^{-1}_{x\to c}\ H^*(\vec{a}, M_c) +\ \dots\right) \\
&= \left(\dot{H}^{-1}_{h\to c} - \dot{H}^{-1}_{x\to c}\right) H^*(\vec{a}, M_c) - \left(\dot{H}^{-1}_{h\to c} - \dot{H}^{-1}_{x\to c}\right) H^*(\vec{a}, N\ M_c) \\
&= \left(\dot{H}^{-1}_{h\to c} - \dot{H}^{-1}_{x\to c}\right) \left(H^*(\vec{a}, M_c) - H^*(\vec{a}, N\ M_c)\right) \\
\Delta H^* &= \Delta \dot{H}^{-1}(N) \sum_t \alpha_t(\vec{a})\ M_c^{-\beta} \left(1 - N^{-\beta}\right)
\end{aligned}$$

A.3 Streamability

A.3.1 Contraction

The streamability of contraction (a dot product) requires that an accumulator exists of the form in Figure 28. Contraction for vectors $v, w \in \mathbb{R}^a$ is given by $v \cdot w = \Sigma_{i=0}^{n-1}\ v_i \cdot w_i$, which can be expressed as $v \cdot w = \Sigma_{i=0}^{s'-1}\ v_i \cdot w_i\ \underline{+\ \Sigma_{i=s'}^{a-1}\ v_i \cdot w_i}$, where the underlined portion is the accumulator. We diagrammatically show this in Figure 32, highlighting the accumulator in blue.

Figure 32: We can re-express contraction as an initial function followed by the accumulator. Any size can be chosen for s and s', and the expression can be recursively expanded until $s' \leq s_a$ for some target stream batch size s_a.



A.4 Fusion of SoftMax-Contraction

SoftMax-Contraction is streamable by maintaining a running maximum and sum on chip as auxiliary variables. We express this by streaming unscaled SoftMax-Contraction with the initialization of the auxiliary variables as the head and scaling by the sum as a tail as in Figure 33.

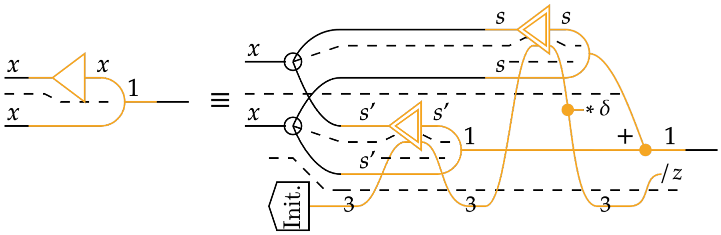


Figure 33: Streamable SoftMax-Contraction is implemented by accumulating the results of unscaled SoftMax-Contraction applied to segments, adjusting the baseline relative to the current maximum value. We apply the normalization by z as a tail for the expression.

We can derive streamable SoftMax-Contraction by taking recursively expanded Auxiliary SoftMax (Figure 34) and contracting its output. Recursively expanded Auxiliary SoftMax is not streamable as the memory usage increases with the input size. However, it terminates in a join, allowing it to be fused with a contraction.

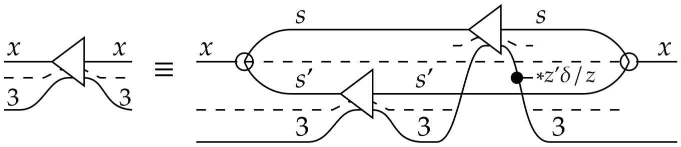


Figure 34: Auxiliary SoftMax (defined in Table 1), where we maintain auxiliary variables, can be recursively expanded.

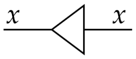
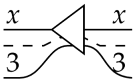
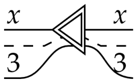
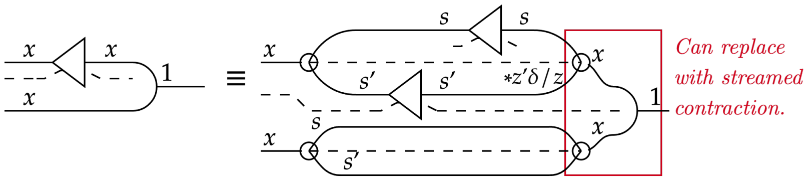
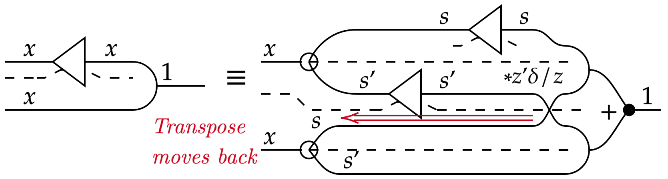
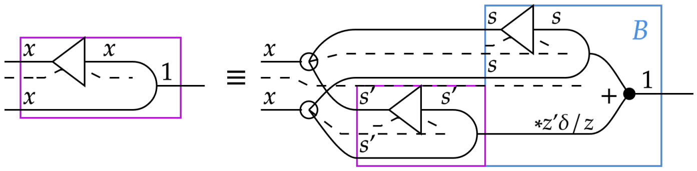
Base SoftMax	**Auxiliary SoftMax**	**Unscaled SoftMax**
		
SoftMax($\vec{x}$) : $\mu \leftarrow \max(\beta x_i)$ $s_i \leftarrow \exp(\beta x_i - \mu)$ $z \leftarrow \Sigma_i\ s_i$ $y_i \leftarrow s_i/z$ Return $\vec{y}$	Initialize() : Return $(-\infty, 0, 0)$ SoftMax$_0(\vec{x}, (\mu', \delta z', z'))$: $\mu \leftarrow \max(\beta x_i, \mu')$ $s_i \leftarrow \exp(\beta x_i - \mu)$ $\delta \leftarrow \exp(\mu' - \mu)$ $z \leftarrow \delta z' + \Sigma_i\ s_i$ $y_i \leftarrow s_i/z$ Return $\vec{y}, (\mu, \delta z', z)$ *Scaling (on prior values),* $\vec{y}'\ * = \vec{y}'\ \delta z'/z$	Initialize() : Return $(-\infty, 0, 0)$ SoftMax$_1(\vec{s}, (\mu', \delta', z'))$: $\mu \leftarrow \max(\beta x_i, \mu')$ $s_i \leftarrow \exp(\beta x_i - \mu)$ $\delta \leftarrow \exp(\mu' - \mu)$ $z \leftarrow \delta\ *\ z' + \Sigma_i\ s_i$ Return $\vec{s}, (\mu, \delta, z)$ *Scaling (on prior values),* $\vec{y}'\ * = \vec{y}'\ \delta$ *Scaling (at tail),* $\vec{y}\ * = \vec{y}/z$

Table 1: We provide diagrams and code for various forms of SoftMax. β is the inverse tempuruture parameter, and is set to $d^{-0.5}$.

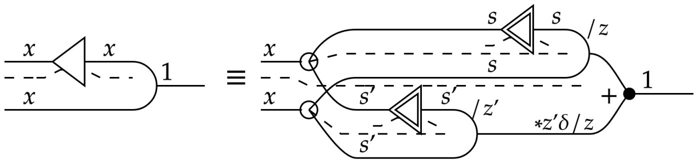
Fusing SoftMax with a contraction limits the output size. The join tail allows it to be fused with a contraction. This limits the size of the output, yielding a streamable algorithm. To streamline the derivation, we do not explicitly draw the updated maintained variables as in Figure 34. We can then apply rearrangements to recover a streamable form of the composed function.

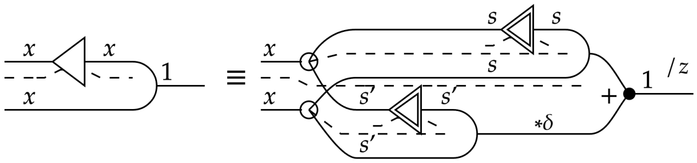


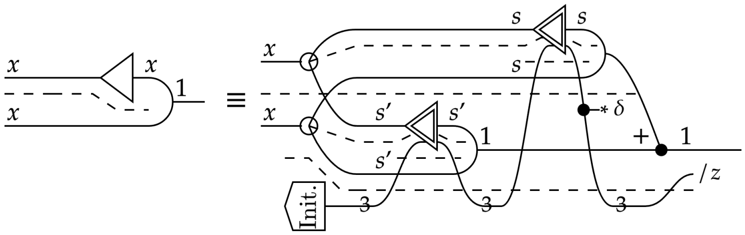




We can then replace SoftMax with unscaled SoftMax, which is done in FlashAttention-2. This lets us move the shared “$/z$” factor to the end of the expression, producing the numerically stable form we use. We forego drawing the auxiliary variables lines to streamline the derivation but add them later.







This analysis derives the streamability of SoftMax and its fusion with contraction as a standard procedure using diagrams. Diagrams, by showing the structure of constituent algorithms, act as algebraic tools for deriving fusion.

A.5 Configuration Tables

A.5.1 Justification for Register Row-Wise Multiplication-Addition

For our algorithms, we opt for scaling on registers instead of using tensor core diagonal matrices as FlashAttention does. This makes algorithms extremely memory efficient, as no *scale_o* (Shah et al., 2024, p.21) variable needs to be stored. We can accumulate in groups along d, leading to a small memory usage. However, these algorithms may suffer in performance. Even though we avoid the wasteful computations of a diagonal matrix, FP16 operations (312/989 TFLOPs at FP16 for A100/H100 (NVIDIA, 2022)) are much slower than tensor cores (78/134), and we require a multiply-add for each sub-loop. However, the compiler *might* decide to overlap tensor core operations on some warps with FP16 operations on others. This overlapping would be revealed by implementing the algorithm and assessing whether tensor core and FP16 multiply-add operations are both fully utilized.

A.5.2 Ampere Attention

To find an exact configuration for our version of Ampere Attention from Figure 26, we first label all the required variables and their sizes in Figure 35. Variables must be pre-allocated with a maximum size, and can encompass one array per column. This generates a configuration table (Table 2), which allows us to choose batch sizes. We configure the batch sizes in Table 2 to derive an implementation of the algorithm which conforms to hardware constraints. The variable d is typically configured to be 128, and it is reasonable to set the batch sizes to their smallest possible size. This results in $w_{\overline{q}} = 32$, $s_{\overline{x}} = 16$, $t_{\overline{q}} = 1$, $d' = 16$. Once we have found the total register and SMEM memory usage per warp, we can find the maximum number of warps per thread block (virtual SMEM). Each thread block can contain up to 256KB of register memory, 163KB/227KB of SMEM (A100/H100) (NVIDIA, 2024, p. 492), and 32 warps. Each thread is limited to 255 registers, containing 4B each (1KB total). We can only share SMEM data between virtual cores, so the number of warps determines our "discount" on distributed key/value loads.

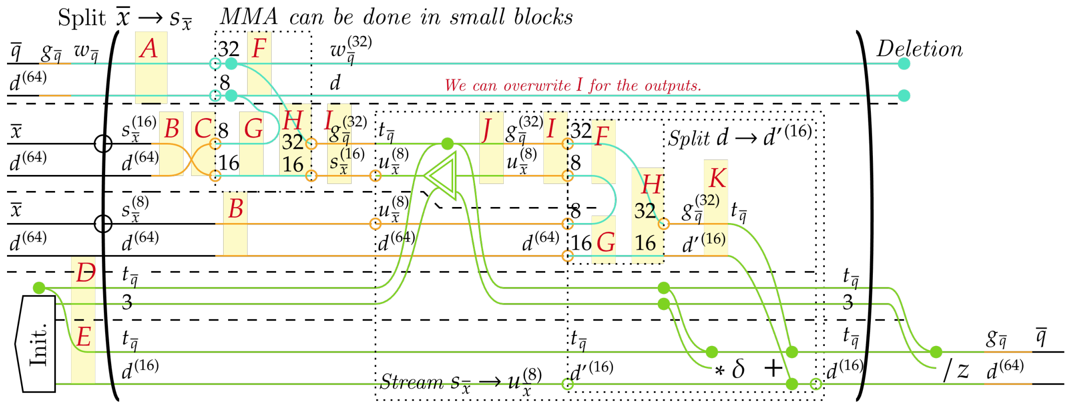

Split $\overline{x} \rightarrow s_{\overline{x}}$
MMA can be done in small blocks
$\overline{q}$ $g_{\overline{q}}$ $w_{\overline{q}}$
$d^{(64)}$
A
32 F
8
$w_{\overline{q}}^{(32)}$
d
We can overwrite I for the outputs.
Deletion
$\overline{x}$
$d^{(64)}$
$s_{\overline{x}}^{(16)}$ B C 8 G
$d^{(64)}$ 16
H 32 16
I $g_{\overline{q}}^{(32)}$ $s_{\overline{x}}^{(16)}$
$t_{\overline{q}}$ $u_{\overline{x}}^{(8)}$
J $g_{\overline{q}}^{(32)}$ I 32 F
$u_{\overline{x}}^{(8)}$ 8
Split $d \rightarrow d'^{(16)}$
H 32 16
K $g_{\overline{q}}^{(32)}$ $t_{\overline{q}}$
$d'^{(16)}$
$\overline{x}$
$d^{(64)}$
$s_{\overline{x}}^{(8)}$ B
$d^{(64)}$
$u_{\overline{x}}^{(8)}$
$d^{(64)}$
$d^{(64)}$ 8 16 G
D
$t_{\overline{q}}$
3
Init.
E
$t_{\overline{q}}$
$d^{(16)}$
$t_{\overline{q}}$
$d'^{(16)}$
Stream $s_{\overline{x}} \rightarrow u_{\overline{x}}^{(8)}$
$* \delta$ +
$d^{(16)}$
$t_{\overline{q}}$
3
$t_{\overline{q}}$
/ z
$g_{\overline{q}}$ $\overline{q}$
$d^{(64)}$

Figure 35: FlashAttention with required memory-allocated variables labeled. All arrays must be contained in a variable at its level in which it fits, and a variable cannot be used twice in a column.

After performing the substitution of batch sizes in Table 2, we find the register and SMEM bytes per warp in Table 3. Using our strategy, we can theoretically fit up to 13 warps per thread block. We can see that SMEM is not an issue, nor is the limit of $1KB$ per thread (32KB/warp). This is significantly larger than the 4-8 warps per thread block given in the FlashAttention-2 paper, implying improved performance. The reasons for this are difficult to tell, as diagrams give much more detail about variable sizes than traditional pseudocode. A significant factor is likely to be avoiding *scale_o* and instead relying on register level row-wise multiplication. We also benefit from a small stream size and sub-loops which reuse variables. How this alternative strategy plays out remains to be tested, and may fail because of factors such as the small sub-stream size introducing excessive error.

	Variable	**Size**	**Location**
A	Queries	$w_{\overline{q}}^{(32)} \times d^{(64)}$	Registers
B	Keys/Values Stream	$s_{\overline{x}}^{(16)} \times d^{(64)}$	SMEM
C	Transposed Keys	$d^{(64)} \times s_{\overline{x}}^{(16)}$	SMEM
D	Auxiliary	$t_{\overline{q}} \times 3$	Registers
E	Output	$t_{\overline{q}} \times d$	Registers
F	Tensor Core Primary	32×8	Registers
G	Tensor Core Secondary	8×16	Registers
H	Tensor Core Accumulator	32×16	Registers
I	Transfer Cache	$g_{\overline{q}}^{(32)} \times s_{\overline{x}}^{(16)}$	SMEM
J	Processed Values	$t_{\overline{q}} \times u_{\overline{x}}^{(8)}$	Registers
K	Subloop Cache	$g_{\overline{q}}^{(32)} \times d'^{(16)}$	SMEM

Table 2: The configuration table for A100 attention lists all the required variables, their size, and location.

	Variable	**Constant per Thread Block**	**Values / Warp**	**Location**
A	Queries		4096	Registers
B	Keys/Values Stream	1024		SMEM
C	Transposed Keys	1024		SMEM
D	Auxiliary		96	Registers
E	Output		4096	Registers
F	Tensor Core Primary		256	Registers
G	Tensor Core Secondary		128	Registers
H	Tensor Core Accumulator		512	Registers
I	Transfer Cache		512	SMEM
J	Processed Values		256	Registers
K	Subloop Cache		512	SMEM
	Total	2048=*4096B*	1024=*2048B*	SMEM
			9440=*18,880B*	Register

	Maximum Bytes per Thread Block	**Maximum Number of Warps**
SMEM	163KB	79.5 (*max 32*)
Register	256KB	13.89 (*max 32*)

Table 3: We substitute in the minimum configuration for batch sizes, deriving the bytes of SMEM and register memory required per warp.

A.5.3 Hopper Attention

For our Hopper Attention, we again employ register register FP16 scaling operations. We will calculate the amount of registers and SMEM required per warp group of 128 threads, and must account for the different quantizations of data present. We identify the variables in Figure 36 and use the configuration Table 4 to find the maximum number of warpgroups per thread block. Furthermore, we can assess the clock cycle of each column to derive a strategy for asynchronous execution.

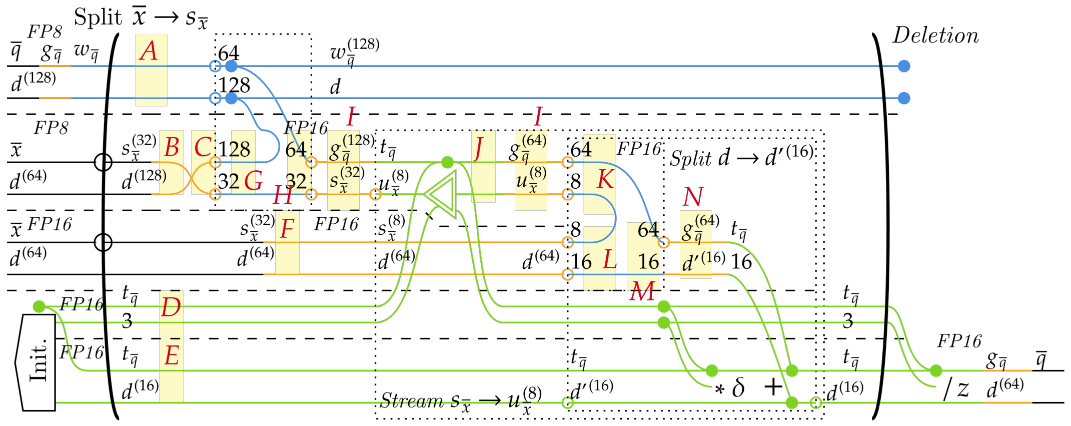


Figure 36: We perform a similar process to Figure 35 of identifying all required memory-allocated variables.

	Variable	**Size**	**Q.**	**Level**	*Per T.B.*	*Per W.G.*
A	Queries	$w_{\overline{q}}^{(128)} \times d^{(128)}$	FP8	SMEM		16384
B	Keys Stream	$s_{\overline{x}}^{(32)} \times d^{(128)}$	FP8	SMEM	4096	
C	Transposed Keys	$d^{(128)} \times s_{\overline{x}}^{(32)}$	FP8	SMEM	4096	
F	Values Stream	$s_{\overline{x}}^{(16)} \times d^{(128)}$	FP16	SMEM	2048	
D	Auxiliary	$t_{\overline{q}} \times 3$	FP16	Registers		384
E	Output	$t_{\overline{q}} \times d^{(128)}$	FP16	Registers		16384
G	FP8 Tensor Core Secondary	128×32	FP8	SMEM		4096
H	FP16 Tensor Core Accumulator	64×32	FP16	Registers*		2048
I	Transfer Cache	$g_{\overline{q}}^{(64)} \times s_{\overline{x}}^{(32)}$	FP16	SMEM		4096
J	Processed Values	$t_{\overline{q}} \times u_{\overline{x}}^{(8)}$	FP16	Registers		1024
K	FP16 Tensor Core Primary	64×8	FP16	SMEM		512
L	FP16 Tensor Core Secondary	8×16	FP16	SMEM		128
M	FP16 Tensor Core Accumulator	64×16	FP16	Register*		1024
N	Sub-loop Transfer Cache	$g_{\overline{q}}^{(64)} \times d'^{(16)}$	FP16	SMEM		2048
	Total (KB=1024B)			SMEM	12	33.25
	Total (KB)			Register		40.75 (*max 128*)

	Maximum Bytes per Thread Block	**Maximum Number of Warpgroups**
SMEM	228KB	6.50 (*max 8*)
Register	256KB	6.28 (*max 8*)

Table 4: We list the required variables and substitute their size. The total bytes needs to consider the quantization employed. * *Warpgroup tensor core operations require the accumulator to be in register memory.*

Referring to Table 4, we can run 6 consumer warpgroups per thread block. Collectively, this represents 24 producer warps per threadblock, double the amount for Ampere algorithm. From our model, we can understand why. Storing tensor core operations on SM effectively doubles the lower-level memory available, doubling the amount of processors which share a stream of keys and values. If we had simply assumed that lower-level memory M_ℓ is doubled without any further information, we would have estimated 26 warps which is 6.5 warp groups for the Hopper algorithm.

Note how in Figure 36 columns alternate between blue (tensor core) and green (register) operations. This indicates we can overlap computation. Each SM has cores dedicated to different operations (NVIDIA, 2022, p.21). These can run simultaneously, and Hopper supports user-configured asynchronous execution between warpgroups. We aim to overlap tensor core, general purpose FP16 operations, and special functions (the exponential). Furthermore, we can dedicate the remaining 2 warpgroups per thread block to be *producers*, asynchronously providing data from higher levels, which uses minimal registers.

We calculate the clock cycles for each column per thread, given in Table 5. We use the sub-loop sizes, as sub-loops can be exploited to overlap computation. Clock cycles are found by dividing TFLOPs by the number of SMs (132) and clock speed (1830MHz) (NVIDIA, 2022, p.39), or from the CUDA programming guide (NVIDIA, 2024).

	FP8 MatMul	**SoftMax**	**FP16 MatMul**	**FP16 Multiply-Add**
Operation	Tensor Core (*FP8*)	Special Function Unit (exponent)	Tensor Core (*FP16*)	Non-Tensor FP16
Ops / Thread	$2ds_{\overline{x}} = 8192$	$u_{\overline{x}} = 8$	$2u_{\overline{x}}d' = 256$	$2d' = 32$
Ops / SM Cycle	8192	16	4096	512
SM Cycles / Thread	1	0.5	1/16	1/16
Number per $u_{\overline{x}}$	n/a	1	$d/d' = 8$	$d/d' = 8$
SM Cycles / Thread *per* $u_{\overline{x}}$	n/a	0.5	0.5	0.5

Table 5: Ignoring the weaving over thread groups, we can find the operations for each column per thread from the diagram, Figure 36.

Each $s_{\overline{x}}$ stream requires an FP8 tensor-core operation followed by four SoftMax and FP16 stages. From Table 5, we see that each $u_{\overline{x}}$ sub-stream requires 0.5 clock cycles for SoftMax, FP16 tensor core, and thread-level scaling. Within warpgroups, we have two independent tiles of size 64 as indicated by $g_{\overline{x}}^{(64)}$. We split each warpgroup into two 64 thread groups and alternate FP16 tensor core operations on one with non-tensor FP16 operations on the other. Furthermore, we overlap SoftMax with FP16 operations on other warpgroups. In Figure 37, we represent overlapping between warpgroups.

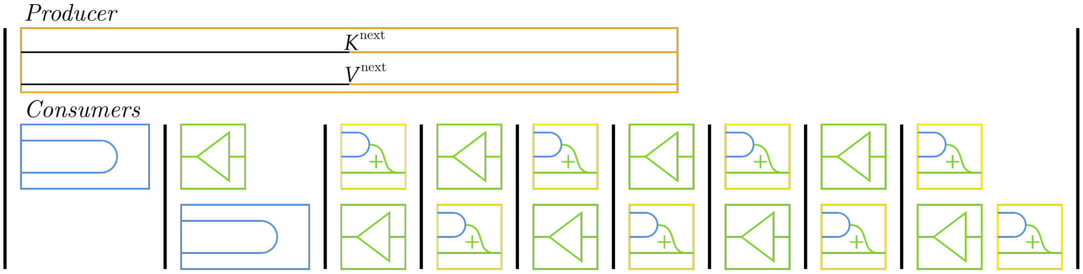


Figure 37: We can overlap the computation of different stages. Thick lines indicate wait blocks, syncing the processors.

In Figure 37, each column takes 0.5 clock cycles per active thread. By having two pipelines, half the warpgroups are active, and therefore each column requires 192 clock cycles. Overall, the algorithm takes 2304 clock cycles. At 1830MHz, this takes 1259ns. Across the SM, $2 \times 8192 g_{\overline{q}} = 1.26e7$ floating point operations are performed, giving 9.99 TFLOPs per core, or 1.32 PFLOPs on the 132 SMs of an H100 SXM5. Each SM loads $3s_{\overline{x}}d = 12KiB$ of data. To not be bandwidth bottlenecked, this requires 9.76 GB/s per SM, or 1.29 TB/s per GPU, which SXM5s can easily achieve. The 1.32 PFLOPs figure improves on FlashAttention-3's 1.2 PFLOPs, and is equal to the maximum throughput of tensor cores where compute is evenly distributed between FP8 and FP16 values.

However, the figure given in Table 5 is an underestimate of the clock cycles required for SoftMax. We require an additional exponential and various conversions between FP32 and FP16 for SoftMax, requiring more than

0.5 cycles per thread. We achieve full throughput if the tensor cores are continuously active, which is the case if tensor core operations bottleneck waits. We can use 64 × 64 by 64 × 32 FP8 tensor core operations, splitting the computation into four steps. After each $u_{\overline{x}}$ FP16 stage, we precompute a quarter of the next I variable. This means each $s_{\overline{x}}$ stream begins with the I input computed. This requires an additional I variable in SMEM, which does not reduce the number of warpgroups.

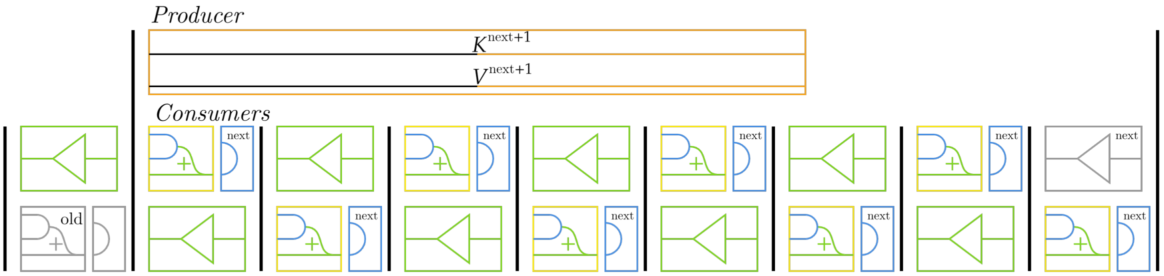


Figure 38: We can split the FP8 tensor core operation into four stages, which can shadow the execution of the slow SoftMax operation.