

Neptune: Advanced ML Operator Fusion for Locality and Parallelism on GPUs

YIFAN ZHAO, University of Illinois Urbana-Champaign, USA

EGAN JOHNSON, University of Illinois Urbana-Champaign, USA

PRASANTH CHATARASI, IBM Research, USA

VIKRAM S. ADVE, University of Illinois Urbana-Champaign, USA

SASA MISAILOVIC, University of Illinois Urbana-Champaign, USA

Operator fusion has become a key optimization for deep learning, which combines multiple deep learning operators to improve data reuse and reduce global memory transfers. However, existing tensor compilers struggle to fuse complex reduction computations involving loop-carried dependencies, such as attention mechanisms.

This paper introduces Neptune, a tensor compiler for advanced operator fusion for sequences of reduction operators. Neptune presents a new approach for advanced operator fusion, which intentionally breaks some existing dependencies and compensates by constructing algebraic correction expressions that allow the kernel to produce the correct result. Applying Neptune's advanced operator fusion to a plain attention operator generates operators equivalent to FlashAttention and FlashDecoding.

On ten attention-based benchmarks, Neptune, starting from a plain attention code and a high-level scheduling template, outperforms existing compilers like Triton, TVM, and FlexAttention, including Triton-based implementations of FlashAttention. Across four different GPU architectures from NVIDIA and AMD, Neptune-generated kernels have an average speedup of 1.35 \times over the next best alternative, with up to 2.65 \times speedup on Nvidia GPUs and up to 3.32 \times on AMD GPUs, demonstrating its effectiveness for deep learning workloads.

ACM Reference Format:

Yifan Zhao, Egan Johnson, Prasanth Chatarasi, Vikram S. Adve, and Sasa Misailovic. 2026. Neptune: Advanced ML Operator Fusion for Locality and Parallelism on GPUs. *Proc. ACM Program. Lang.* 10, PLDI, Article 220 (June 2026), 37 pages. <https://doi.org/10.1145/3808298>

1 INTRODUCTION

Finding high-performance implementations for modern deep learning models is essential for low-latency and low-cost model deployment. There is a large gap between high-level machine learning frameworks like Pytorch [31] and TensorFlow [1], which specify models in terms of mathematical tensor operators, and the high-performance kernels near the hardware, which must account for performance characteristics like memory hierarchies, complex loop tilings, and more.

Tensor kernel compilers are a promising way to translate groups of tensor operators into efficient device kernels. *Tile-based compilers* and languages, such as Triton [39], Pallas [4], and TileLang [44], operate with a low level of abstraction over hardware vendor languages. Programmers write tile-based programs where tiles (fixed-shape subtensors) are first-class objects. Tile languages provide an SIMD interface, and realize data-level parallelism on GPUs and accelerators. However, tile languages'

Authors' addresses: Yifan Zhao, University of Illinois Urbana-Champaign, USA, yifanz16@illinois.edu; Egan Johnson, University of Illinois Urbana-Champaign, USA, egancj2@illinois.edu; Prasanth Chatarasi, IBM Research, USA, prasanth@ibm.com; Vikram S. Adve, University of Illinois Urbana-Champaign, USA, vadve@illinois.edu; Sasa Misailovic, University of Illinois Urbana-Champaign, USA, misailo@illinois.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/6-ART220

<https://doi.org/10.1145/3808298>

low level of abstraction means programming in tile languages remains difficult, requiring both significant hardware and algorithmic optimization expertise. *Schedule-based compilers* such as Halide [33] and TVM [6] provide an alternative. They combine a mathematical definition of the program, with a recipe of transformation primitives (“schedule template”) to optimize the program. As the schedule template calls built-in transformations, the compiler automatically safeguards correctness and fidelity to the original program.

Presently, however, both tile and schedule compilers miss important transformations that are critical for performance, which forces programmers to escape the compiler pipeline and manually implement kernels for important operators in tile frameworks or in vendor languages such as CUDA and HIP. This approach may sometimes be effective, but requires significant expert effort and sacrifices the portability, flexibility, and maintainability of high-level frameworks.

Attention operators are one example where the lack of key optimizations in a compiler forces developers to provide handwritten kernels. Attention is a sequence of four operations that benefit greatly from operator fusion: matrix multiplication (matmul), element-wise division, softmax, and another matmul. Tile languages are unable to provide this fusion automatically, as it is above the scope of the tile optimizer and must be encoded by the developer in the input program. Schedule languages are at the right abstraction level for operator fusion, but existing compiler transformations are unable to fuse attention’s operators due to complex data dependencies surrounding multiple reduction loops. As a result, compilers and developers resort to manually developed, specialized fusion solutions such as FlashAttention [13, 14] and FlashDecoding [15], which struggle to generalize to diverse workloads of variant operators.

We identify two missing pieces preventing current tensor compilers from capturing optimizations equivalent to FlashAttention and FlashDecoding:

- Standard loop fusion in today’s tensor compilers misses many optimization opportunities because of complex, fusion-preventing dependencies. Programmers divert from the compiler pipeline to apply manual, ad-hoc operator fusion, and imitate the code generation capabilities of compilers with code templates like FlexAttention. This situation indicates that generalizing operator fusion is necessary and possible, but existing tensor compilers are not capable of it.
- Existing tensor optimization approaches do not suffice alone. Tile optimizers leave important high-level optimizations — such as operator fusion — to programmers. Schedule optimizers produce intractable search spaces, with long sequences of many fine-grained transformations, that overwhelm programmers and autotuners. Integration of schedule and tile optimization pipelines is desirable, but has been virtually non-existent.

Our Work: We present Neptune, a novel tensor kernel compiler that combines novel advanced operator fusion with a schedule optimizer and tile optimizer to provide full optimization in one pipeline. Neptune fuses multiple reduction operators with complex data dependencies that are beyond the reach of existing popular tensor compilers such as TVM [6], Triton [39], and OpenXLA [30]. Applying Neptune’s advanced operator fusion to a plain attention operator generates operators equivalent to FlashAttention and FlashDecoding. Neptune makes two key technical innovations:

- We present a novel paradigm for advanced operator fusion with sequences of reduction operators. This paradigm solves the issue of fusion-preventing data dependencies by 1) intentionally breaking some of those dependencies (“naive fusion”) and 2) constructing algebraic correction expressions (“repair terms”) that allow the kernel to produce the correct result by the end of its execution. In exchange for a small amount of computation and cache memory, the transformation reduces global memory transfers and improves data reuse.

We present two instantiations of this paradigm: Rolling Update fusion and Split-K fusion. Both

transformations are applicable and effective on attention-like operators. When applied to attention, rolling update fusion produces FlashAttention equivalents, suited for attention in prefill mode, and split-k fusion produces FlashDecoding equivalents for decoding attention. They are designed as primitives in schedule compilers and compose with other program transformations.

- We present a compilation pipeline that integrates scheduling and tile-based optimizations. It separates low-level optimizations, which the tile optimizer handles, and high-level optimizations scheduling primitives, keeping scheduling concise and simplifying the search space. Scheduling applies to a loop-scalar intermediate representation (IR) that consists of loop nests and scalar expressions, which is well-suited for expression-manipulating transformations such as rolling update. After scheduling, Neptune’s translation engine lowers sections of loop-scalar IR to a tile IR, which can be optimized by the tile optimizer for high performance on device.

These two innovations represent key building blocks for supporting — *natively within a compiler* — advanced algorithmic optimizations that produce high-performance tensor kernels. Like Halide and TVM, Neptune takes as input a mathematical program definition and optimization schedule, but keeps the primitives focused on high-level optimizations.

We evaluate Neptune on 10 different attention-based operators from the literature and several other operators, on multiple GPU architectures from NVIDIA and AMD. Our results show that Neptune-generated kernels have lower latency than other compiler-based frameworks, Triton [39], TVM [6], FlexAttention [18], and Mirage [47], including Triton-based implementations of FlashAttention. Out of 320 optimized configurations (combinations of operators, sequence lengths, and GPUs), Neptune generates the lowest-latency kernel in 284 cases. Neptune achieves a speedup over the next best, already highly optimized alternative of 1.35× (geomean of all cases), up to 2.65× on Nvidia GPUs and up to 3.32× on AMD GPUs. Further, in 101 out of 256 configurations, Neptune improves over the state-of-the-art kernels from CUTLASS [28] based library implementations.

Contributions: The paper makes several contributions:

- We present Neptune, a tensor compiler that supports advanced loop fusion algorithms for reduction operators, while leveraging the advantages of both schedule-based and tile-based GPU optimization of tensor programs.
- We present a novel paradigm for operator fusion that intentionally breaks data dependency and automatically derives algebraic repairs to obtain the correct result. Two instances, rolling update fusion and split-k fusion, are well-suited for optimizing attention-like operators.
- We present a translation from scheduled loop programs to tile programs that a standard tile optimizer can process, freeing schedules from the burden of low-level optimizations.
- We implemented Neptune on top of the Apache TVM schedule tensor compiler and the Triton tile tensor compiler. Neptune’s optimization approach is general and can be implemented over other similar tensor compilers. Neptune is available at <https://github.com/uiuc-arc/neptune>.
- We thoroughly evaluated Neptune against state-of-the-art baselines. Neptune produces kernels that are faster than existing compilers by 1.35× on average (geomean) and up to 3.32×, on a diverse set of attention variants and four GPU architectures from Nvidia and AMD.

2 MOTIVATION

Figure 1 shows an example program where data dependencies prevent traditional fusion. The input program (Figure 1a) runs on a 2-by-4 matrix `inp` and consists of three loop nests: a row-wise `max s_max`, an element-wise exponential `s_exp`, and a row-wise sum `s_sum`.

This program is similar to the softmax computation used in the attention operator [41]. A major challenge in computing attention is that it does not scale to large sequence lengths (L), because some intermediate tensors have size $L \times L$ and quadratic memory complexity. Operator fusion can

```

# s_max:
for i in range(2):
    for j in range(4): # loop_j
        xmax[i] = max(
            xmax[i], inp[i, j])
# s_exp:
for i, j in grid(2, 4):
    xexp[i, j] = exp(
        inp[i, j] - xmax[i])
# s_sum:
for i, j in grid(2, 4):
    xsum[i] += xexp[i, j]

```

```

for i in range(2):
    for j in range(4): # loop_j
        # s_max:
        xmax[i] = max(
            xmax[i], inp[i, j])
        # s_exp:
        xexp[i, j] = exp(
            inp[i, j] - xmax[i])
        # s_sum:
        xsum[i] += xexp[i, j]

```

```

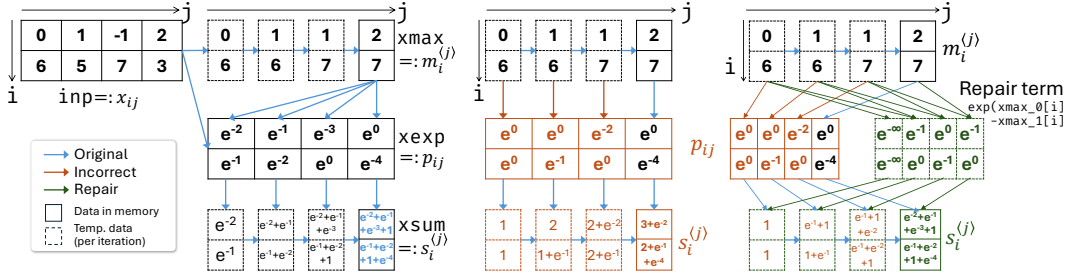
for i in range(2):
    xmax_0[i] = -inf
    for j in range(4): # loop_j
        # s_max:
        xmax_1[i] = max(
            xmax_0[i], inp[i, j])
        # s_sum:
        xsum[i] = (
            exp(xmax_0[i] - xmax_1[i])
            * xsum[i]
            + exp(inp[i, j] - xmax_1[i]))
        xmax_0[i] = xmax_1[i]

```

(a) The original program, consisting of 3 loop nests s_{\max} , s_{exp} , and s_{sum} .

(b) Naive fusion of the original program. s_{exp} and s_{sum} are fused under loop_j without considering the data dependencies.

(c) Correct fusion of Figure 1a changes the compute expressions of s_{sum} in addition to applying naive loop fusion.



(d) Materialized example of Figure 1a. Data dependencies and values of this program are shown in blue.

(e) Materialized example of Figure 1b; inp is omitted; incorrect results and dependencies in orange. Note, $s_i^{(3)} = s_i^{(3)}$.

(f) Materialized example of Figure 1c. Values of the repair term and repaired $xsum$ are shown in green.

Fig. 1. An example program requiring reduction fusion, incorrect result using naive loop fusion, and a correctly fused program, shown on the first row. The second row (“materialized example”) shows the values of each matrix in the programs when applied to a concrete inp matrix. x_{\max} and x_{sum} are each shown four times since their values vary over j iterations.

help avoid manifesting these tensors in memory, but classic fusion techniques do not apply to this program due to its complex data dependency. FlashAttention [14] is a specialized fusion solution for attention. It enables fusion using a manually derived and proven algebraic correction term, which is limited to attention. We now demonstrate how a correction term enables fusion for softmax.

Figure 1d shows a concrete execution of the program 1a on an input inp , where blue arrows mark data dependencies. We shorthand matrix entries $x_{ij} := \text{inp}[i, j]$ and $p_{ij} := \text{xexp}[i, j]$, and denote $s_i^{(j)}$ and $m_i^{(j)}$ as the values of $x_{\text{sum}}[i]$ and $x_{\max}[i]$ at iteration j . The program’s final result is $s_i^{(3)}$.

Fusing s_{sum} into s_{\max} , both of which are reduction loops, is an operator fusion that existing tensor compilers are not capable of. If we disregard loop-carried dependencies and naively fuse s_{exp} and s_{sum} into s_{\max} with existing loop fusion, we obtain an incorrect program shown in Figure 1b. The concrete execution in Figure 1e illustrates why it is incorrect: fusing s_{exp} under loop_j changes some s_{exp} iterations to read from x_{\max} too early. The brown arrows mark these incorrect data dependencies. For example, in the first iteration, p_{i0} reads from $m_i^{(0)}$ in the same iteration, instead of $m_i^{(3)}$ computed after 4 iterations. Thus, p_{i0} is incorrect and propagates to the final result $s_i^{(3)}$. We refer to this loop fusion without dependency checks as *naïve fusion*, and current tensor compilers would reject this fusion in practice.

However, it is possible to build on naïve fusion and produce a correct program if we *update*

the compute expressions of the fused loop nest `s_sum`. We can find new expressions for `s_sum` that not only compute the current iteration, but also *repair* results from the previous iteration as we move forward in the reduce dimension j . Figure 1c shows the correct fusion result based on this idea. This program multiplies a *repair term* $\exp(\text{xmax_0}[i] - \text{xmax_1}[i])$ with `xsum[i]`. Figure 1f shows the values of these repair terms and the repaired $s_i^{(j)}$ values. The repaired program is correct because $s_i^{(3)}$ is equal to $s_i^{(3)}$, even though $s_i^{(j)} \neq s_i^{(j)}$ for $j < 3$.

To illustrate how the repair term repairs `xsum[i]` at every j iteration, we write out $s_i^{(j)}$ for three iterations, as an expression of x_{ij} and $m_i^{(j)}$:

$$\begin{aligned}
 s_i^{(0)} &= \exp(x_{i0} - m_i^{(0)}); \\
 s_i^{(1)} &= \exp(m_i^{(0)} - m_i^{(1)}) \cdot s_i^{(0)} + \exp(x_{i1} - m_i^{(1)}) \\
 &= \exp(m_i^{(0)} - m_i^{(1)}) \cdot \exp(x_{i0} - m_i^{(0)}) + \exp(x_{i1} - m_i^{(1)}) \\
 &= \exp(x_{i0} - m_i^{(1)}) + \exp(x_{i1} - m_i^{(1)}) \\
 s_i^{(2)} &= \exp(m_i^{(1)} - m_i^{(2)}) \cdot (\exp(x_{i0} - m_i^{(1)}) + \exp(x_{i1} - m_i^{(1)})) + \exp(x_{i2} - m_i^{(2)}) \\
 &= \exp(x_{i0} - m_i^{(2)}) + \exp(x_{i1} - m_i^{(2)}) + \exp(x_{i2} - m_i^{(2)})
 \end{aligned} \tag{1}$$

We observe that the following behavior is key to a working repair term: at each iteration j , multiplying the previous iteration result $s_i^{(j-1)}$, which is an expression of $m_i^{(j-1)}$, and replaces all uses of $m_i^{(j-1)}$ with $m_i^{(j)}$. If $\langle j \rangle$ is a tag on the array access, then this repair term is a *tag updater*. We show in Section 4 that Neptune automatically finds a tag updater for a large class of programs.

This solution to operator fusion adds a small amount of computational overhead, as the repair term is evaluated once per iteration. It also adds memory storage overhead to record the previous iteration results (i.e. `xmax_0` and `xmax_1`) in GPU registers or shared memory. This memory overhead is low, as reduction output arrays are small compared to other arrays in the program, such as `xexp` and `inp`. Fusion benefits, including reduced global memory transfers and improved data reuse, often far outweigh the overhead, as evidenced by FlashAttention’s performance improvements.

3 SYSTEM OVERVIEW

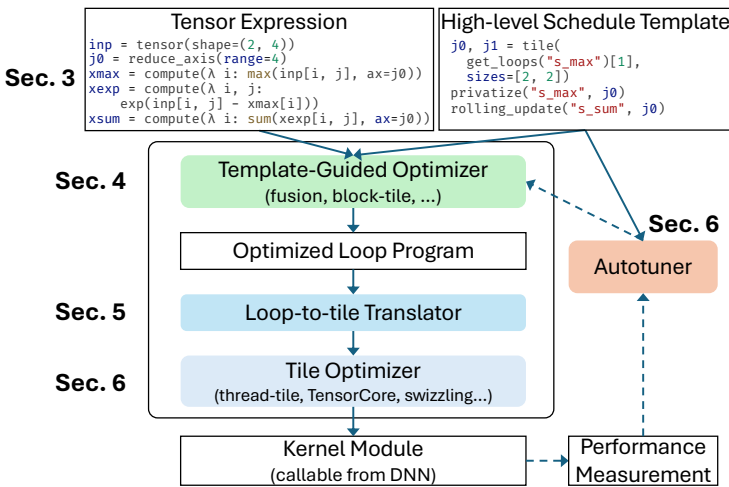


Fig. 2. The main components of Neptune.

Figure 2 summarizes the main components of Neptune. Neptune takes as input a program

written in tensor expressions and a schedule template for the program. The schedule template is a user-provided recipe with a list of *high-level optimizations* for the program. Neptune optimizes the input program with the following steps:

- The template-guided optimizer converts the tensor expression into a program in Neptune’s loop-scalar IR, and applies the transformations in the template on it. Neptune’s two novel advanced fusion algorithms (§4) are in this optimizer, and they compose with other standard transformations. This step outputs a loop program that has undergone high-level optimizations.
- The loop-to-tile translator converts the loop program into a program in Neptune’s tile IR. This translation algorithm (§5) is automatic and covers a majority of programs representable in the loop-scalar IR. The tile optimizer applies tile-specific transformations to the program (e.g., data movement, tile data layout, and HW-specific execution) and produces an executable kernel.
- Neptune’s autotuner (§6) mutates the schedule template and optimization parameters to find a better list of transformations that further improves kernel performance.

Input: Tensor Expression. Neptune’s input programs are written in a compact tensor expression language which translates straightforwardly into Neptune’s loop-scalar IR. Figure 2 (top left) shows the tensor expressions that produce the loop program in Figure 1a. Tensor expressions are written in TVM expression language here; other tensor expression languages such as JAX’s jaxpr language could be used in principle.

Input: Schedule Template. A schedule template is a list of high-level optimization steps the user writes to describe how to optimize the program. Each step calls a transformation primitive in Neptune with some parameters. The template-guided optimizer applies each step in the template to the program. We describe the optimizer and the transformation primitives available in Section 3. Figure 2 (top right) shows an example template that is applicable to the given program.

While schedule templates resemble the scheduling languages used in some tensor compilers, Neptune templates are concise. In a few lines of code, a user can express all the transformations needed to optimize a complex operator with multiple loop nests. Schedule templates only describe high-level optimizations that benefit from user direction, while Neptune delegates low-level optimizations to its tile optimizer without user intervention. An example of tensor expression and schedule for attention is in Appendix A.

3.1 Preliminaries and Notation

Loop Nest Notation. We use a compact notation for loop nests: $nest(\mathbf{i} : \mathbf{n})\{s\}$ to represent K perfectly nested loops with upper bounds $\mathbf{n} := (n_1, \dots, n_K)$ and body s . Neptune normalizes the lower bound of loops to 0 without loss of generality. $\mathbf{i} \in \mathbb{Z}^K$ is an iteration vector of the nest. The iteration domain \mathcal{D}^s is the set of values \mathbf{i} can take, determined by \mathbf{n} . We denote the previous iteration of \mathbf{i} by $prev(\mathbf{i})$ and the next iteration by $next(\mathbf{i})$.

Affine Access Function and Tags. In a tensor access $X[\phi(\mathbf{i})]$ under a loop nest (with iteration vector \mathbf{i}), ϕ is an *affine access function* that projects \mathbf{i} to tensor indices. The tensor access $xsum[\mathbf{i}]$ under loops i and j has an access function $\phi(i, j) = i$ (example from Figure 1a). When a tensor entry is updated multiple times, we add a *tag* on the access to mark the iteration we refer to: $X[\phi(\mathbf{i})]^{(j)}$ ($xsum[\mathbf{i}]^{(j)}$ for example). Tags reflect temporal data dependency on the same tensor entry. We have used tags in Figure 1.

Reduction in Neptune IR. A reduce loop nest is a loop nest that expresses a reduction operation. It consists of map loops and reduce loops, where the output is only indexed by the map loop variables, and each output value is updated as many times as the reduce loops execute. A reduce loop nest in Neptune’s loop-scalar IR has the form

$$rnest(\mathbf{i} : \mathbf{n}, \mathbf{j} : \mathbf{m}) \{ X[\phi(\mathbf{i})] = X[\phi(\mathbf{i})] \textcircled{f} expr(\mathbf{i}, \mathbf{j}) \} \quad (2)$$

The notation $rnest(i : n, j : m) \{ s \}$ distinguishes reduce loops from map loops. f is the reducer, an associative binary function, and $a \textcircled{f} b$ is equivalent to $f(a, b)$.

Recurrent and Explicit Forms of Reductions. Equation 2 is the *recurrent form* of a reduction, which iteratively updates the tensor X . We define the reduce operator \mathcal{R} to express the value of X compactly: $\mathcal{R}(f, 0 \leq j \leq j_0, g(j))$ applies f to fold over $g(j)$ from $j = 0$ to j_0 to produce a single value. For instance, $\mathcal{R}(+, 0 \leq j \leq 2, \text{xexp}[i, j])$ equals $\text{xexp}[i, 0] + \text{xexp}[i, 1] + \text{xexp}[i, 2]$. We refer to this \mathcal{R} -based expression as the *explicit form* for the value of X .

4 TEMPLATE-GUIDED OPTIMIZER

The template-guided optimizer translates the input tensor expression program into loop-scalar IR. This translation is a standard process, as the tensor expression language and the loop-scalar IR are both similar to those in many tensor compilers. The abstract syntax of this IR is shown in Figure 6 of the next section. Next, the template-guided optimizer applies the transformations in the template to the loop IR program. The template is a list of instructions that refer to transformation primitives in Neptune, including standard loop transforms, data layout transforms, and data placement transforms (e.g., caching in shared memory). Neptune also provides novel transformations that enable reduction fusion, described next.

4.1 Algebraic Correction Analysis for Reduction Fusion

Neptune is capable of fusing reduction operations. In Figure 1, we have manually fused the reductions `s_sum` and `s_max` and fixed the result with a repair term. Finding this repair term is the main challenge of reduction fusion. To enable generalized and reusable reduction fusion in a compiler, Neptune has a novel analysis that derives a repair term automatically from the program, which is the core behind its reduction fusion primitives. We give an intuition of how Neptune finds the repair term from the motivating example (§2) and show the formal generalization here.

Characterizing a Reduction. Neptune implements generalized fusion of a reduction loop (a consumer reduction) that depends on values produced by one or more other reduction loops (producer reductions). We consider consumer loops that decompose into a reducer function f performing reduction over terms and a function g that produces those terms:

$$f : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X} \text{ (associative);} \quad g : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}. g(r, c)$$

In $g(r, c)$, r is the result of the producers, and c is a collection of values not produced by a reduction (e.g., a program input). For simplicity, this section will provide intuition for a single producer reduction, which will be formalized for multiple producers in section 4.2.

Reduction Repair Function. We take inspiration from the repair term $\exp(m_i^{(j-1)} - m_i^{(j)})$ in the example, and generalize it to a *repair function*:

$$h : \mathcal{X} \times \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}. \quad h(t, r, r')$$

This function takes three arguments: t , an in-progress reduction value from the consumer reduction (`s_sum`), and r and r' , two reduction values from different iterations of the producer reduction (`s_max`). Like the repair term, we aim to find a specific h capable of taking a reduction value t computed with r , and repair it to a reduction value t' computed with r' . For a reduction characterized by f and g above, satisfying two conditions allows h to work as a repair function:

$$h(g(r, c), r, r') = g(r', c) \quad \forall r, r', c \in \mathcal{X} \quad (3)$$

$$h(x \textcircled{f} y, r, r') = h(x, r, r') \textcircled{f} h(y, r, r') \quad \forall x, y, r, r' \in \mathcal{X} \quad (4)$$

Namely, h replaces the r argument of g , and h distributes with the reducer f . Importantly, an h

satisfying condition 3 can be mechanically constructed if $g(r, c)$ is invertible in the argument c (called the inverse function g_c^{-1}):

$$h(t, r, r') = g(r', g_c^{-1}(r, t)) \quad (5)$$

Section 4.2 presents the full formal arguments that Equations 3-5 are correct, but intuitively they allow out-of-date r -values r_0 in a reduction to be replaced with new values r_1 . For example, consider a small two-term reduction $g(r_0, x) \textcircled{f} g(r_0, y)$ which transforms through h to $g(r_1, x) \textcircled{f} g(r_1, y)$:

$$\begin{aligned} h(g(r_0, x) \textcircled{f} g(r_0, y), r_0, r_1) &= h(g(r_0, x), r_0, r_1) \textcircled{f} h(g(r_0, y), r_0, r_1) && \text{Using Eq. 4} \\ &= g(r_1, x) \textcircled{f} g(r_1, y) && \text{Using Eq. 3} \end{aligned}$$

Consider again Equation 1. Our general formulation maps to this example with:

$$\begin{aligned} f(t_0, t_1) &= t_0 + t_1 && \text{Reducer Function} \\ g(m_i, x_{ij}) &= \exp(x_{ij} - m_i) && \text{Reduction Terms} \\ h(t, m_i^{(k)}, m_i^{(k+1)}) &= t \exp(m_i^{(k)} - m_i^{(k+1)}) && \text{Repair Term} \end{aligned}$$

The functions f and g are derived from the program, while h simplifies from Eq. 5. Algebraic correction analysis identified appropriate f , g , and h , and enables multiple operator fusion transformations in Neptune, described in sections 4.2 and 4.3.

4.2 Rolling Update

Rolling update is Neptune's novel transformation that fuses reduction loop nests. It applies to a reduce loop nest L_t (the consuming reduction/fusee) and a target loop l (the *rolling loop*) to fuse L_t under: `RollingUpdate(L_t, l)`. Producing reductions are not given as arguments and are discovered via data dependency analysis. Rolling update requires that the fusee L_t is a reduce loop nest, and that the rolling loop l does not enclose L_t .

Algorithm 1 outlines the full rolling-update algorithm. There are four major steps in rolling update, which we explain in the following paragraphs. When any step fails, such as precondition check, pattern matching, or repair function solving, rolling update returns the original program. A step-by-step example of applying rolling update to Figure 1a is in Appendix B.

Step 1: Dataflow Reorganization. Rolling update performs an analysis on the dataflow graph to find the producing reductions of L_t as a set \mathcal{L}_r (algorithm line 1). Then on each path from L_t that ends in \mathcal{L}_r (excluding both ends), rolling update inlines all the loop nests, similar to how we inlined `s_exp` in the example. These inlined loop nests are predecessors of L_t , so their computation eventually accumulates in L_t . After inlining, all of \mathcal{L}_r are immediate producers of L_t .

Step 2: Loop Transformation. Rolling update fuses \mathcal{L}_r , followed by L_t , under l using naive loop fusion (line 3), skipping loop nests that are already under l . This step ensures that all loop nests in \mathcal{L}_r share an outer loop nest with L_t . L_t is now in fused position and produces incorrect results, which need to be repaired.

Algorithm 1: ROLLINGUPDATE(L_t, l)

Input: L_t : the target loop nest to be fused. L_t holds a reference to the entire program.
Input: l : the loop to fuse L_t under ("rolling loop").
Output: The fused loop nest

- 1 $\mathcal{L}_r = \text{INLINEMAPRETURNREDUCE}(L_t)$;
- 2 **for** $L \in \mathcal{L}_r \cup \text{set}(L_t)$ **do**
- 3 $L = \text{NAIVELOOPFUSION}(L, l)$;
- 4 $(f, g) = \text{MATCHREDUCEPATTERN}(L_t, \mathcal{L}_r)$;
- 5 $h = \text{SOLVREPAIRFUNC}(g)$;
- 6 $\text{VALIDATEHCOMMUTATIVE}(h, f)$;
- 7 $M_x = \text{dict}()$;
- 8 **for** $L \in \mathcal{L}_r$ **do**
- 9 $(x_{\text{prev}}, x_{\text{curr}}) = \text{CACHEREDUCEPREVRESULT}(L)$;
- 10 $M_x[L] = (x_{\text{prev}}, x_{\text{curr}})$;
- 11 $L_t = \text{APPLYREPAIRTERM}(L_t, h, M_x)$;
- 12 **return** L_t

Step 3: Finding the Repair Function. Rolling update uses Equations 4 and 5 (§4.1) to find the repair function h . It first needs to understand the structure of the L_t loop nest. On algorithm line 4, it matches L_t to the structure of a reduction loop nest in Neptune’s IR. This step extracts f and g as expressions from the input program:

$$rnest(\mathbf{i} : \mathbf{n}, \mathbf{j} : \mathbf{m}) \{ X_t[\phi_t(\mathbf{i})] = X_t[\phi_t(\mathbf{i})] \textcircled{f} g(X_r[\phi_r(\mathbf{i})], C[\phi_c(\mathbf{i}, \mathbf{j})]) \} \quad (6)$$

This pattern matches a reduction in Neptune’s IR (Eq. 2), with additional structure $g(\dots)$ on the right of the reducer f . X_t is the output of L_t , X_r is the output of the producing reduction of L_t (which is $L_r \in \mathcal{L}_r$), and C is non-reduce input to L_t (not produced by a reduction, e.g., program input). We use one producing loop as an example for simplicity and will generalize it later.

Given f and g , rolling update solves Eq. 5 to find the repair function h (line 5), and proves if h satisfies Eq. 4 (line 6), both using a symbolic solver.

Step 4: Applying the Repair Function. Rolling update applies the repair function h to the body of L_t , performing the rewrite from Eq. 6 to the following (loop nest omitted):

$$X_t[\phi_t(\mathbf{i})] = h\left(X_t[\phi_t(\mathbf{i})], X_r[\phi_r(\mathbf{i})]^{(\text{prev}^{(j)})}, X_r[\phi_r(\mathbf{i})]^{(j)}\right) \textcircled{f} g(X_r[\phi_r(\mathbf{i})], C[\phi_c(\mathbf{i}, \mathbf{j})]) \quad (7)$$

This rewrite corresponds to the example in Figure 1c, where we applied the repair term on the left-hand side of the reducer (+) while not changing the right-hand side.

Eq. 7 requires both $X_r[\phi_r(\mathbf{i})]^{(\text{prev}^{(j)})}$ and $X_r[\phi_r(\mathbf{i})]^{(j)}$ (output of L_r at the previous and current iterations), but a reduction only keeps the output of the current iteration. Therefore, we need to transform the producing reduction to retain the results of two iterations. In the example (Figure 1c), `s_max` is transformed to produce `xmax_0` and `xmax_1`, one for each iteration. Lines 8-10 describe this process: rolling update transforms each $L_i \in \mathcal{L}_r$ to produce two output tensors. Finally, line 11 applies the above rewrite and concludes the rolling update algorithm.

Generalizing to Multiple Producing Reductions. Rolling update generalizes to a variety of dataflow patterns, including multiple producing reductions and non-reduce inputs for the fusee L_t , which we omitted in the repair function analysis for simplicity. Now, we discuss how to handle multiple producing reductions L_{r1}, \dots, L_{rN} and non-reduce inputs C_1, \dots, C_M .

We make the following changes to the previous definitions in algebraic correction analysis (§4.1): g is a function of $\mathcal{X}^N \times \mathcal{X}^M \rightarrow \mathcal{X}$, which we denote as $g(\mathbf{r}, \mathbf{c})$. The repair function $h : \mathcal{X} \times \mathcal{X}^N \times \mathcal{X}^M \rightarrow \mathcal{X}$ now reads the previous and current iteration results from each of the N producing reductions: $h(t, \mathbf{r}, \mathbf{r}')$. We retain the conditions and solution for h from Section 4.1, which remains formally similar, with \mathbf{r} , \mathbf{r}' and \mathbf{c} becoming vectors:

$$h(g(\mathbf{r}, \mathbf{c}), \mathbf{r}, \mathbf{r}') = g(\mathbf{r}', \mathbf{c}) \implies h(t, \mathbf{r}, \mathbf{r}') = g(\mathbf{r}', g_c^{-1}(\mathbf{r}, t)) \quad \forall \mathbf{r}, \mathbf{r}' \in \mathcal{X}^N, \mathbf{c} \in \mathcal{X}^M \quad (8)$$

$$h(x \textcircled{f} y, \mathbf{r}, \mathbf{r}') = h(x, \mathbf{r}, \mathbf{r}') \textcircled{f} h(y, \mathbf{r}, \mathbf{r}') \quad \forall x, y \in \mathcal{X}, \mathbf{r}, \mathbf{r}' \in \mathcal{X}^N \quad (9)$$

Support for Non-Invertible Functions. Neptune can find a repair term for a class of compute patterns even when the g function is non-invertible. If g is not invertible against \mathbf{c} , making a change of variables $C := g_c(\mathbf{c})$ may help the solver find a solution:

$$\exists g_c : \mathcal{X}^M \rightarrow \mathcal{X}, g_0 : \mathcal{X}^N \times \mathcal{X} \rightarrow \mathcal{X} \quad \text{s.t.} \quad g(\mathbf{r}, \mathbf{c}) \equiv g_0(\mathbf{r}, g_c(\mathbf{c}))$$

If this change of variables produces a function $g_0(\mathbf{r}, C)$ that is invertible in C , Neptune can still find the repair function h . We denote the inverse function of $g_0(\mathbf{r}, C)$ as $g_0^{-1}(\mathbf{r}, t) = C$. In this case, the solution to the repair function h in Equation 5 still holds (replacing g with g_0):

$$h(t, \mathbf{r}, \mathbf{r}') = g_0(\mathbf{r}', g_0^{-1}(\mathbf{r}, t))$$

An example where this change of variables is useful is $g(\mathbf{r}, \mathbf{c}) = \exp(\mathbf{r} - \text{relu}^2(\mathbf{c}))$. This function is not invertible in \mathbf{c} because it applies the non-injective Squared ReLU function. Neptune’s solver

finds the variable substitution $C = \text{relu}^2(c)$, and solves the alternative problem $g(r, C) = \exp(r - C)$ which is invertible.

Correctness of the Repaired Program. A correct program is one where the L_t loop nest after rolling update produces the same result as in the original program.

THEOREM 4.1 (CORRECTNESS OF THE REPAIRED PROGRAM). *If rolling update succeeds and rewrites the reduce loop nest L_t to Eq. 7, the value of X_t in the repaired program is the same as in the original program after execution of L_t .*

We sketch a proof for this theorem here; the full proof of this theorem and the lemmas used are in Appendix D. The loop nest before fusion is shown as a program in Eq. 6, which iteratively updates X_t for iterations j from 0 to m . We convert this program into an expression for the value of X_t , expressed using the *explicit form* of a reduction:

$$X_t[\phi_t(\mathbf{i})]^{(m)} = \mathcal{R}(f, 0 \leq j \leq m, g(X_r[\phi_r(\mathbf{i})], C[\phi_c(\mathbf{i}, j)])) \quad (10)$$

To prove the correctness of rolling update, we show that the loop nest after fusion, which is given by Eq. 7, also corresponds to the same X_t value as Eq. 10. This calculation in the full proof uses the repairing property of h to simplify the expression, so we formally define it, which we refer to as the *tag-updating property*:

Definition 4.2. A function h *tag-updates* a reduction characterized by reducer f and function g if it satisfies the following condition:

$$\begin{aligned} & h\left(\mathcal{R}\left(f, 0 \leq j \leq j_0, g\left(X_r[\phi_r(\mathbf{i})]^{(j_f)}, C[\phi_c(\mathbf{i}, j')]\right)\right), X_r[\phi_r(\mathbf{i})]^{(j_f)}, X_r[\phi_r(\mathbf{i})]^{(j_t)}\right) \\ &= \mathcal{R}\left(f, 0 \leq j \leq j_0, g\left(X_r[\phi_r(\mathbf{i})]^{(j_t)}, C[\phi_c(\mathbf{i}, j')]\right)\right) \quad \forall j_0, j_f, j_t \in \mathcal{D}^s, j_f \preceq j_t \preceq j_0 \end{aligned}$$

This definition matches the intuition that h replaces out-of-date r -values (ones with tag j_f) in a reduction with new values (ones with tag j_t). Section 4.1 provided a solution for the repair function h (Equations 3-5). We check that the solution satisfies this definition of the tag-updating property with a lemma:

LEMMA 4.3. *If h satisfies Equations 3-5, then it tag-updates as Def. 4.2 requires.*

We prove Lemma 4.3 and Theorem 4.1 in Appendix D. Their proofs use the associative property of the reducer f , which is true for real (\mathbb{R}) inputs. In practice, rolling update applies to programs over floating-point inputs, for which f is not strictly associative. We empirically check the numerical accuracy of the transformed program (evaluation in Appendix F) and find it to be highly accurate.

Tradeoffs and Scope of Rolling Update. Rolling update implements reduction fusion. It extends operator fusion to a mix of elementwise and reduction operations, bringing the benefits of operator fusion: improving data reuse, reducing global memory transfers, and reducing peak memory usage. On the other hand, rolling update adds extra computation to the target loop nest L_t due to the repair term h . Moreover, since the new program maintains the result of last and current iterations, the rolling loop l becomes harder to parallelize due to more complex data flow. We describe another fusion approach in Section 4.3 that alleviates the second issue.

There exist cases where Neptune's solver truly fails to find a solution even with the change of variables technique. Such failures often indicate that fusion is inherently difficult between the operators. One example is the simultaneous computation of mean and variance known as the Welford algorithm [45]. For numerical stability, it is standard to compute the variance of data by first shifting the data by their mean: $s(\mathbf{x}) = \sum_i (x_i - m)^2 / (N - 1)$ where $m(\mathbf{x}) = \sum_i x_i / N$ and $\mathbf{x} \in \mathbb{R}^N$. The Welford algorithm provides a formula f to update the variance online for every incoming sample: $s_{i+1} = f(s_i, x_{i+1})$. Neptune's rolling update is applicable to fusing the mean and

variance computation and in principle can produce a fused one-pass algorithm similar to Welford’s algorithm, but the algebraic correction analysis fails to find a solution. Neptune’s solver currently does not take into consideration that the previous/current mean, the current variance, and the incoming sample are all related quantities, which can further simplify the compute expressions.

4.3 Split-K Update

Neptune is capable of another reduction fusion transformation, *split-k update*. While rolling update makes the rolling loop harder to parallelize, split-k update avoids this issue and aims to maximize parallelism. Split-k breaks dependencies of reductions so that *partial reductions* are computed in parallel in one loop nest, and a second loop nest combines and repairs the partial results simultaneously. Applying split-k update to attention produces kernels similar to FlashDecoding [15]. This transformation is named because it resembles the split-k strategy in matrix multiplication [9].

Split-k update derives from the same break-and-repair paradigm as rolling update, using the same analysis and repair function h . It also reuses dataflow reorganization (step one) of rolling update. The difference lies in the values applied to h : while rolling update uses h to tag-update by one iteration, split-k update uses h to replace local reduction results with global reduction results.

Figure 3 shows the result of applying split-k update to the motivating example in Figure 1a. An accompanying data dependency graph is shown in Figure 4. The fusee s_sum and the producing reduction s_max are each split into a local reduce loop nest (s_max_local , s_sum_local) and a global reduce loop nest (s_max_global , s_sum_global). The local nests compute partial reductions, and the global nests combine partial results. The repair function h is applied to the global nest of the fusee, s_sum_global . Unlike rolling update, the “rolling” loop $j0$ is now highly parallelizable.

Reduction Privatization. Splitting a reduction into a partial and a global reduction is a standard optimization technique called reduction privatization, which Neptune provides as a transformation primitive. It applies to a reduce nest L that runs K reduce iterations and creates two reduce nests: an L^{local} with K reduce iterations distributed to k threads, and an L^{global} with $\lceil K/k \rceil$ reduce iterations to combine the partial results. Privatization used in Figure 3 has $K = 4$ and $k = 2$.

Privatization requires a tile size k as input argument. Split-k uses a combined transformation of naive loop fusion followed by privatization: FuseAndPrivatize, as naive loop fusion produces a loop nest structure that privatization can infer the tile size k from.

Split-K Algorithm. Algorithm 2 outlines the split-k update algorithm. We explain this algorithm in comparison to rolling update:

- Split-k update applies dataflow reorganization of rolling update to get \mathcal{L}_r .
- On algorithm lines 2 to 5, split-k update uses FuseAndPrivatize to fuse all producing reductions of L_t under l , and tracks their local and global loop nests L^{local} and L^{global} .
- Line 6 replaces tensor reads in L_t such that it reads from the local nests of the producing reductions. Line 7 fuses and privatizes L_t under l .
- Line 8 reuses the repair function analysis of rolling update to find h from L_t^{local} .

Algorithm 2: SPLITKUPDATE(L_t, l)

Input: L_t : the target loop nest to be fused

Input: l : the target loop to fuse L_t under

Output: A pair of loop nests: the local and global versions of fused L_t

```

1  $\mathcal{L}_r = \text{INLINEMAPRETURNREDUCE}(L_t);$ 
2  $(M^{local}, M^{global}) = (\text{dict}(), \text{dict}());$ 
3 for  $L_r \in \mathcal{L}_r$  do
4    $(L_r^{local}, L_r^{global}) = \text{FUSEANDPRIVATIZE}(L_r, l);$ 
5    $(M^{local}[L_r], M^{global}[L_r]) = (L_r^{local}, L_r^{global});$ 
6  $L_t = \text{REPLACETENSORREADS}(L_t, M^{local});$ 
7  $(L_t^{local}, L_t^{global}) = \text{FUSEANDPRIVATIZE}(L_t, l);$ 
8  $h = \text{ROLLINGUPDATE SOLVER REPAIRFUNC}(L_t^{local}, \mathcal{L}_r);$ 
9  $L_t^{global} = \text{SPLITKAPPLYREPAIRTERM}(L_t^{global}, h, M^{local});$ 
10 return  $(L_t^{local}, L_t^{global});$ 

```

```

for i, j0 in grid(2, 2): # <- "rolling" loop j0
  for j1 in range(2): # s_max_local
    max_l[i, j0] = max(max_l[i, j0], inputs[i, j0 * 2 + j1])
  for j1 in range(2): # s_sum_local
    sum_l[i, j0] += exp(
      inputs[i, j0 * 2 + j1] - max_l[i, j0])
for i, j0 in grid(2, 2): # s_max_global
  max_g[i] = max(max_g[i], max_l[i, j0])
for i, j0 in grid(2, 2): # s_sum_global
  sum_g[i] += exp(
    max_l[i, j0] - max_g[i]) * sum_l[i, j0]

```

Fig. 3. The result of applying split-k update to Fig. 1a. Split-k update privatizes two reductions in the original program, creating a local and a global reduction for each.

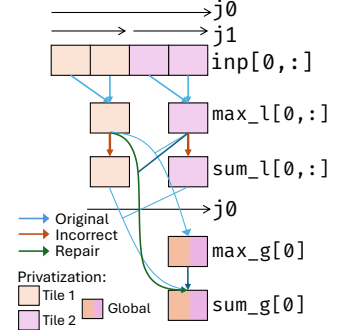


Fig. 4. Data dependency graph of split-k transformed program in Figure 3.

- Line 9 applies the repair function h to the global nest L_t^{global} .

Applying the Repair Function h (Split-K). After naive loop fusion and privatization, the local nest L_t^{local} of the fusee reads from the local versions of its producing reductions, which is incorrect. Split-k repairs the global nest L_t^{global} and does not change L_t^{local} , keeping it parallelizable. Another difference from rolling update is that split-k applies the repair function on the right-hand side of the reducer. The following *rewrite pattern* shows how split-k update applies the repair function:

$$\begin{aligned}
X_{t,g}[\phi_{t,g}(\mathbf{i})] &= X_{t,g}[\phi_{t,g}(\mathbf{i})] \circledast X_{t,l}[\phi_{t,l}(\mathbf{i})] \\
\Rightarrow X_{t,g}[\phi_{t,g}(\mathbf{i})] &= X_{t,g}[\phi_{t,g}(\mathbf{i})] \circledast h(X_{t,l}[\phi_{t,l}(\mathbf{i})], X_{r,l}[\phi_{r,l}(\mathbf{i})], X_{r,g}[\phi_{r,g}(\mathbf{i})])
\end{aligned} \quad (11)$$

This equation shows a “before \Rightarrow after” rewrite pattern. The first line is the body of the global nest before the rewrite, and the second line is after the rewrite. $X_{t,g}$, $X_{t,l}$, $X_{r,g}$ and $X_{r,l}$ are the output tensors of L_t^{global} , L_t^{local} , L_r^{global} and L_r^{local} respectively. Tags are omitted because every tensor access refers to the final result of that memory location. In the example (Figure 3), split-k modifies the global loop nest s_sum_global and the computation in the local s_sum_local is unchanged. The global loop nest s_sum_global now has the store statement $sum_g[i] = sum_g[i] + \underline{\exp(max_l[i, j0] - max_g[i])} * sum_l[i, j0]$, where the underlined term is the repair term.

Correctness of Split-K Update. The definition of transformation correctness remains the same as rolling update: split-k update produces a program where the reduce nest L_t^{global} produces the same tensor values as the original loop nest L_t before the fusion.

THEOREM 4.4 (CORRECTNESS OF SPLIT-K UPDATE). *If split-k update successfully creates L_t^{global} and applies the repair in Eq. 11, then $X_{t,g}$ in the repaired program after the execution of L_t^{global} equals X_t in the original program after the execution of L_t .*

We sketch a proof for this theorem here; the full proof of this theorem is in Appendix D.

We first describe the values of the program before and after the transformation. Before the transformation, the loop nest L_t produces X_t . The correctness proof of rolling update has shown the explicit expression for X_t in Eq. 10. We need to compare this expression of X_t to the expression of $X_{t,g}$ produced by the global nest L_t^{global} . Because L_t^{global} reads from the local reduction results $X_{t,l}$, we first write the expression for $X_{t,l}$ by writing down the body of L_t^{local} as a program:

$$(\text{for } j:) \quad X_{t,l}[\phi_{t,l}(\mathbf{i})] := X_{t,l}[\phi_{t,l}(\mathbf{i})] \circledast g(X_{r,l}[\phi_{r,l}(\mathbf{i})], C[\phi_c(\mathbf{i}, j)])$$

We denote the number of iterations of the local reduction as m^{local} , and express $X_{t,l}[\phi_{t,l}(\mathbf{i})]$ explicitly:

$$X_{t,l}[\phi_{t,l}(\mathbf{i})] = \mathcal{R}\left(f, 0 \leq j \leq m^{\text{local}}, g(X_{r,l}[\phi_{r,l}(\mathbf{i})], C[\phi_c(\mathbf{i}, j)])\right)$$

Combining $X_{t,l}[\phi_{t,l}(i)]$ and the body of L_t^{local} in Eq. 11, we simplify the expression using the repair function h , which is shown in the full proof. The simplified expression matches the before-transformation expression for X_t in Eq. 10.

Tradeoffs of Split-K Update. Split-k update provides different tradeoffs from rolling update: in addition to the benefits of operator fusion, it offers more parallelism through the use of privatization to create parallelizable local reductions. On the other hand, it requires extra space for the local reduction results, controlled by the tile size k . Applying split-k update to attention produces kernels similar to FlashDecoding [15], which is the state-of-the-art method for attention in decoding mode, as decoding tolerates more space overhead and benefits greatly from more parallelism.

4.4 Example Operators Amenable to Neptune Transformations

Attention. The attention operator is a sequence of four compute steps: matrix multiplication (matmul), element-wise score computation, softmax, and another matmul. Appendix C provides a detailed description of the attention computation. Neptune fuses all the compute steps of attention into a single loop nest by applying reduction fusion twice. The first fusion applies to the two reductions in softmax, similar to our motivating example in Figure 1. The second fusion fuses the second matmul into the softmax. Following the algebraic correction analysis (§4.1), Neptune identifies the element-wise computation $g(r, c) = \exp(c - r)$ and reduction function $f(x, y) = x + y$. Finally, Equation 5 produces the repair term $h(t, r, r') = t \exp(r' - r)$.

Performer. Performer [8] is an alternative operator to attention that approximates softmax with linear algebraic and element-wise computations. Performer consists of many small compute steps and exhibits lower overall computational complexity than attention. Neptune fuses the many compute steps of performer into two loop nests, applying reduction fusion four times. The compute pattern g in all cases and the h repair term in one case are different from attention. Appendix C describes these reduction fusion patterns in detail.

Numerically stable L^2 norm. Computing the L^2 norm of a vector by definition can overflow when the squared values are too large. It is more numerically stable to scale the entries in the vector first: $\|x\|_2 = m \sqrt{\sum_i (x_i/m)^2}$ where $m = \max_i |x_i|$. This numerically stable algorithm exists in widely used linear algebra libraries such as LAPACK [3]. Neptune fuses the second reduction (summation) into the first (maximum) so that the L^2 norm is computed in a single pass. Following the algebraic correction analysis (§4.1), it identifies the reduction function $f(x, y) = x + y$ and the element-wise computation $g(r, c) = (c/r)^2$. Using Equation 5, it finds the repair function $h(t, r, r') = t(r^2/r'^2)$.

Dynamic scaling quantization on DNN operators. Low-precision training and inference often computes statistics (e.g. minimum and maximum) over the input and output tensors to derive scaling and quantize the tensors. Neptune can fuse many variants of this computation with the surrounding layers. One such example is computing RMSNorm: $y_i = x_i/\sqrt{s + \epsilon}$ where $s = \text{mean}(x^2)$, and then finding the per-row max of the output: $m = \max_i y_i$. The mean and max computations are two reductions that Neptune fuses. Neptune's algebraic correction analysis identifies:

$$g(r, c) = \frac{c}{\sqrt{r + \epsilon}}, \quad f(x, y) = \max(x, y), \quad \text{and} \quad h(t, r, r') = \frac{\sqrt{r + \epsilon}}{\sqrt{r' + \epsilon}} t$$

Notably, Neptune validates that h distributes with \max (condition 4), because $\sqrt{r + \epsilon}/\sqrt{r' + \epsilon}$ is always non-negative, and h monotonically increases with t .

5 TRANSLATING FROM LOOP-SCALAR IR TO TILE IR

Tile-based compilers such as Triton [39] and Pallas [4] excel in low-level optimizations to generate efficient device code optimized for local hardware features, such as tensor cores for matrix multiplication. To take advantage of tile compilers, Neptune translates loop-level programs operating

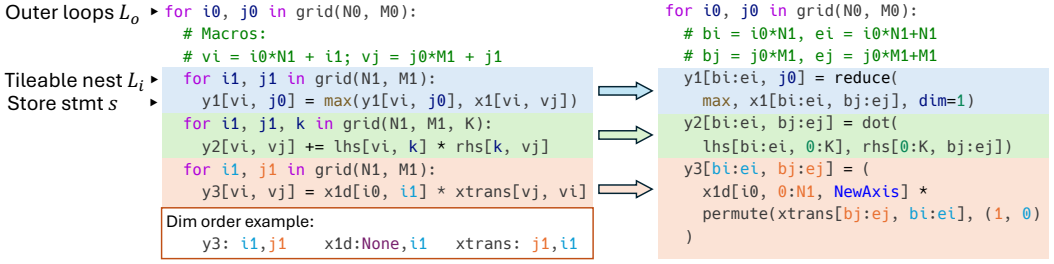


Fig. 5. An example translation from a loop-scalar program (left) to a tile program (right), with three tileable loop nests to translate. The bottom of the LHS program shows an example of dimension order reconciliation for the last loop nest. The LHS program uses shorthands v_i and v_j to express loop-tiled indices.

on scalar elements to tile-level programs in a process known as *tensorization*. While tensorization has been studied extensively [5, 46, 55], Neptune adds a novel *dimension reconciliation* step that generalizes the translation to a larger class of programs. Neptune translates loop programs to *HTile IR*, a novel, custom tile IR that we design to resemble existing tile languages, and generates code from *HTile IR* to different tile languages such as Triton.

Figures 6 and 7 show the abstract syntax of Neptune’s loop-scalar IR and *HTile IR*, respectively. Compared to the loop-scalar IR, *HTile IR* has the same loop and sequential statements, and replaces scalar expressions and store statements with tensor tile expressions and tile store statements. *HTile IR* also adds tile-specific operations like dimension operations (broadcasting and permutation), reductions, and dot products. These operations are selected because they receive wide support from tile compilers, which is an effect of the hardware features available on today’s GPUs.

Figure 5 shows a before-after example of a translated tile program. Algorithm 3 outlines the translation procedure with three main steps: tileable nest detection (algorithm lines 1-2), access region division (lines 3-4), and dimension order reconciliation (lines 5-6 in green). Dimension order reconciliation makes the translation more general than typical tensorization algorithms that rely on pattern matching to match tile computation to specific hardware features (e.g., detecting transposed input for TensorCore matmuls).

Tileable Nest Detection finds the tileable loop nest (line 2) for each store statement s by walking up the AST from s , stopping at the first dynamically sized loop or non-loop statement. This take-until procedure produces the tileable loop nest L_i of s . Then it continues to collect any outer loops enclosing L_i and returns them as L_o . For example, Figure 5 contains three tileable loop nests, the first being $L_i = (i1, j1)$ (with blue background), and $L_o = (i0, j0)$ are the outer loops for all three nests. Next, L_i will be translated into a single tile store statement.

Access Region Division (line 4) replaces each tensor access $id[expr*]$ in s with a tensor *tile* access. It converts $x1d[i0, i1]$, an access in the third nest of Figure 5, into $x1d[i0:i0+1, 0:N1]$. Each index is a *slice* with a start and an end index. Neptune uses existing tensorization algorithms (built on TVM [6]) to implement this step, finding regions of access under loop nests.

Algorithm 3: TRANSLATELOOPTOTYPELILE(s_0)

Input: s_0 : the statement to translate, in loop-scalar IR.

Output: The translated statement in *HTile IR*.

```

1 for  $s \in \text{VISITSCALARSTORESTMTS}(s_0)$  do
2    $(L_o, L_i) := \text{TRAVERSEUPFORTILEABLENEST}(s_0, s)$ ;
3   for  $e \in \text{VISITTENSORACCESEXPRS}(s)$  do
4      $e_t := \text{CREATETENSORTILEACCESS}(e, L_i)$ ;
5      $d := \text{EXTRACTDIMORDER}(e, L_i)$ ;
6      $e'_t := \text{RECONCILEDIMORDERTOLOOPORDER}(e_t, d, L_i)$ ;
7     REPLACE(in= $s$ , from= $e$ , to= $e'_t$ );
8    $s' := \text{MATCHCOMPUTEPATTERN}(s)$ ;
9   REPLACE( $s_0, L_o, s'$ );
10 return  $s_0$ 
  
```

```

    stmt := loop | store | seq_stmt
    seq_stmt := stmt stmt+
    iter_space := range(expr) | grid(expr+)
    loop := for id+ in iter_space: stmt
    store := id[expr*] = expr
    expr := lit | id | id[expr*] | op(expr*)
    op := + | - | exp | log | select | ...

```

Fig. 6. Neptune loop-scalar IR syntax.

```

    stmt := loop | storet | stmt*
    idx := expr | expr:expr | NewAxis
    storet := id[idx*] = exprt
    exprt := id[idx*] | op(exprt*)
           | permute(exprt, order=(i*))
           | reduce(op, exprt, dim=i)
           | dot(exprt, exprt, acc=exprt)

```

Fig. 7. Neptune HTile IR syntax. Non-terminals that are the same as in Figure 6 are omitted.

5.1 Dimension Order Reconciliation

The previous steps extract tile regions from store statements loop nests in the loop-scalar IR. A loop nest holds more information, however: the order of loops also naturally encodes how the dimensions of tensors are traversed. The third loop nest in Figure 5 highlights what is missing from just tile regions. Simply assembling the tiles from the Access Region Division step would produce $y3[bi:ei, bj:ej] = x1d[i0:i0+1, 0:N1] * xtrans[bj:ej, bi:ei]$, which is incorrect and leaves out the broadcast and permutation operations that the original loop nest expressed.

To preserve order information, we take inspiration from Einstein notation and *einops* expressions [35], which assigns a symbol per dimension and encodes the dimension orders in a string like $i, ji \rightarrow ij$. We use the loop variables in L_i as the set of symbols, and extract a dimension order d for each tensor access e . Examples on the bottom left of Figure 5 show: $y3[vi, vj]$ yields order $(i1, j1)$, $xtrans[bj, bi]$ yields $(j1, i1)$, and $x1d[i0, i1]$ yields $(None, i1)$. None is a placeholder for an index that does not depend on the iteration variables from loops in L_i .

Next, we match d to the order of loop variables in L_i , denoted as d_L . As we rearrange d , we apply a corresponding operation on the tile access e to obtain correct access e_t . For example, to bring tensor access $e = xtrans[bj:ej, bi:ei]$ (with $d = (j1, i1)$) to match loop order $(i1, j1)$, we apply a matching transpose $e_t = permute(xtrans[bj:ej, bi:ei], (1, 0))$. Any d can be rearranged to match d_L in three steps:

- **Dimension squeezing** removes a None from d and converts the slice on that dimension in e_t to a scalar index. The slice is guaranteed to have an extent of 1, because it does not use loops from L_i . For $d = (None, i1)$ and $e_t = x1d[i0:i0+1, 0:N1]$, we discard the None from d and convert e_t to $x1d[i0, 0:N1]$.
- **Permutation** arg-sorts d to d_L and produces a permutation order σ . Sorting is possible because we have removed None from d , and $d \subseteq d_L$. For $d = (j1, i1)$, we arg-sort d to get $\sigma = (1, 0)$ and apply $permute(e_t, order=(1, 0))$ to e_t .
- **Dimension unsqueezing** (dimension expansion) inserts a missing variable from d_L into d , and adds a NewAxis index on e_t , which creates a size-1 dimension. For $d = (i1,)$ and $e_t = x1d[i0, 0:N1]$, $j1$ is missing at the end of d , so we insert a NewAxis to e_t to get $x1d[i0, 0:N1, NewAxis]$, which brings the tile to 2-D of size $N_1 \times 1$.

On lines 5-7 in the algorithm, we repeat dimension reconciliation for each tensor access e in s , which produces a tile store statement that is faithful to the original loop nest. Lastly, the algorithm matches this tile store to the types of tile computation that HTile IR allows (dot product, reduction, element-wise; line 8), based on the number of reduction loops in the loop nest, and replaces the inner loop nest L_0 with this tile store s' .

5.2 Generality of the Translation Algorithm

The translation algorithm supports many programs representable in the loop-scalar IR. As HTile IR is designed to fit the capabilities of Neptune’s tile optimizer, it rejects a program if any tensor access uses an inner loop variable (from L_i) multiple times, such as `tensor[i1, i1]`, or uses any inner loop variable non-linearly, such as `tensor[i1 * i2]`. These usages are not supported by the tile optimizer and are rare in tensor programs used in machine learning.

6 IMPLEMENTATION

Neptune is built on top of TVM and Triton tensor compilers. We implement Neptune’s loop-scalar IR and HTile IR as extensions of TVM TensorIR. Neptune’s two inputs, tensor expression and schedule template, are based on TVM’s tensor expression (TE) and scheduling language respectively. Neptune extends TVM’s scheduling language by adding transformation primitives such as rolling update. Neptune uses Triton as its tile optimizer and code generator, and Neptune has a translator from HTile IR to Triton’s Python DSL. We use SymPy, a Python library for solving symbolic equations, to implement the analysis for the repair function h in rolling update and split-k update.

Neptune is implemented in 10K lines of C++ and Python code, with 3K lines for loop fusion transformations and 1.3K lines for loop-to-tile translation. Neptune adapts TVM MetaSchedule [36] as its autotuner, reusing its search algorithm. We register our transformations with MetaSchedule so that it knows how to work with Neptune schedules. MetaSchedule is a cost model-guided autotuner that repeats the following steps: produce 1024 schedules, predict their performance, and run 16 of them on the target hardware (called *empirical measurements*). Every prediction invokes the template-guided optimizer in Neptune, while every empirical evaluation fully generates the program, applying our loop-to-tile translation and Triton-based tile optimization.

7 EXPERIMENTAL METHODOLOGY

GPU platforms. We choose two *datacenter devices* Nvidia A100 [10] (SXM4 40 GB version) and AMD MI300; and two *desktop devices* Nvidia RTX A5000 [12] and Nvidia RTX 6000 Ada [11].

Attention-based Operator Experiments. We evaluate Neptune on 10 attention-based operators shown in Table 1. Each operator is extracted from a specific large language model (LLM) or vision language model (VLM) architecture, shown on the **Base Arch.** column. We profile operators in prefill (PF) mode: where the query sequence length s_q equals key/value length s_{kv} , and decoding (DC) mode, where $s_q = 1$. Variations on masking (Global, Causal, and Window) are indistinguishable in decoding, so we select the Causal variant out of the three. All operators run in inference mode at float16 (half) precision. We profile these operators at varying sequence lengths: $2^7 = 128$, $2^8 = 256$, ..., $2^{15} = 32768$, and a batch size of 1 unless otherwise specified. We refer to the triple of operator, input shape, and GPU as a **setup**, and we evaluate $10 \times 8 \times 4 = 320$ setups. We run Neptune’s autotuner for 128 empirical measurements per setup.

Non-Attention Operator Experiments. We evaluate Neptune on 3 non-attention operators: numerically stable L^2 norm, RMSNorm with dynamic scaling quantization, and performer, profiled

Table 1. LLM operators used in our evaluation. PF is short for prefill and DC for single-token decoding.

Operator	Description	Base Arch.
Global (PF)	Global (plain) attention	ViT-L/16
Causal (PF/DC)	Attn + causal mask	GPT3 6.7B
ALiBI (PF/DC)	Attn + ALiBi bias + causal mask	MPT 7B
GQA (PF/DC)	Attn + GQA heads + causal mask	LLama3 70B
SoftCap (PF/DC)	Attn + SoftCap bias + GQA heads + causal mask	Gemma2 27B
Window (PF)	Attn + windowed mask	Gemma2 27B

on Nvidia RTX 6000 Ada. For performer, we use the same base architecture as defined in the performer official implementation [24]: 8 attention heads, per-head dimension 64, and number of features 256. We maintain a batch size of 1 and vary the sequence length L from 512 to 32768. For L^2 norm and RMSNorm, we direct their reduction along the rows, and use a 2 dimensional input of shape $N \times M$. N varies from 32 to 256, while M varies from $2^{10} = 1024$ to $2^{17} = 131072$.

Baselines. We list our 10 baselines and underline the names we refer to them by in evaluation. Our baselines include 4 tensor compilers: Triton 3.2, FlexAttention (FlexAttn) 2.6.0, Mirage 0.2.2, and TVM 0.18. Triton requires a user to write the kernel in its frontend language, so we compare three mainstream implementations in Triton: OpenAI Triton [29], Tri-Dao Triton [16], and XFormers Triton [22]. Our baselines also include 4 optimized inference libraries: PyTorch 2.6.0 [31], cuDNN 9.11.0, Tri-Dao Cutlass 2.7.4 (Dao-AILab’s implementation) [16], and FlashInfer 0.2.4 [49]. Each implementation supports a subset of the setups we evaluate. For PyTorch, we use `torch.compile(backend='inductor')` to ensure high performance. We evaluate Mirage on A100, on the Global operator over a grid of $s_q \neq 1$ and s_{kv} , which is consistent with the evaluation in the Mirage paper. For non-attention operators, we compare Neptune kernels against PyTorch kernels optimized using `torch.compile(backend='inductor')`.

Profiling and Performance Reporting. For any kernel we profile, we run the kernel once to warm up and discard the result, and run 15 times to report the mean latency. We profile the kernels with Nsight Systems (nsys) on Nvidia GPUs and ROCprofiler (rocprofv3) on AMD GPU. Both profilers trace the program and measure the latency of every kernel in the program. We report **speedup** of Neptune over the best baseline: if Neptune’s kernel has mean latency t_0 and the baselines have t_1, \dots, t_n , then our speedup is $\min(t_1, \dots, t_n)/t_0$. We also report **relative performance** of a baseline relative to Neptune, which is t_0/t_i for the i -th baseline. To average these metrics across setups, we compute geometric means (geomeans).

8 EVALUATION

8.1 Neptune vs. Tensor Compilers

Overall Trends. Table 2 presents the average speedup of Neptune over the best compiler baseline where each cell is a geomean over the input sequence lengths. Compilers are a broad term; here we only include compilers that raise the abstraction level above hardware vendor languages, such as tile compilers. See §9 for a detailed explanation. Out of 320 setups we evaluated, Neptune achieves better or equal performance compared to all other compilers on 284 setups. Neptune shows an improvement over the baselines for all four GPU architectures, ranging from 1.15× to 1.85× and indicating the portability of our approach across GPU architectures. Across all setups, Neptune achieves 1.35× the performance of the best compiler baseline. All kernels for PF benchmarks use Rolling Update, and all kernels for DC benchmarks use Split-K Update.

Detailed Results for Setups. Figure 8 presents detailed performance for every setup for five operators, chosen for prevalence in existing LLMs. Each plot shows relative performance (y-axis) of other compilers normalized to Neptune for different sequence lengths (x-axis).

Table 2. Speedup of Neptune relative to the best compiler baseline, for all 10 operators on 4 GPUs.

Operator \ GPU	GPU			
	6000Ada	A5000	A100	MI300
Global (PF)	1.03	1.05	1.10	1.84
Causal (PF)	1.06	1.06	1.05	1.29
GQA (PF)	1.02	1.05	1.06	1.32
Causal (DC)	1.06	1.08	1.11	3.26
GQA (DC)	1.45	1.37	2.21	1.55
ALiBi (PF)	1.56	1.61	1.71	2.02
ALiBi (DC)	1.28	1.27	2.65	3.32
SoftCap (PF)	1.08	1.07	1.03	1.42
SoftCap (DC)	1.08	0.95	1.86	2.55
Window (PF)	1.03	1.02	0.99	1.23
Average	1.15	1.14	1.38	1.85

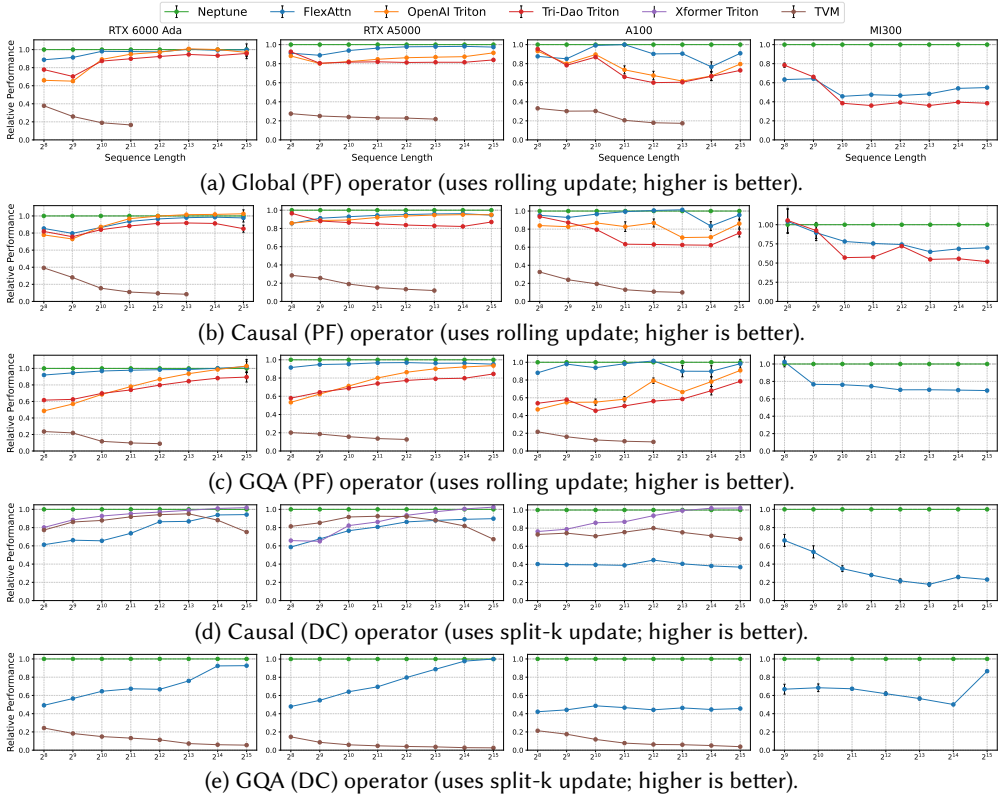


Fig. 8. Performance of Neptune kernels vs. kernels by other tensor compilers. Each y-axis shows relative performance of all compilers normalized to Neptune. The plots for the remaining benchmarks are in Appendix E.

On Nvidia GPUs, Neptune achieves the best performance in a majority of setups. TVM performance is low as its fusion legality check forbids reduction fusion, resulting in multiple kernels with increased memory transfer. Most tile-based implementations show better performance than TVM and improve their performance as the sequence length increases. For prefill (PF) operators, Neptune achieves higher performance for shorter sequence lengths. Neptune and FlexAttn benefit from autotuning and have higher performance than other baselines that rely on Triton heuristics.

For decoding (DC) operators, we evaluate XFormer Triton since the other two Triton kernels do not support them. TVM provides higher performance for Causal (DC) than for other operators, because Causal (DC) is a matrix-vector multiplication operator where fusion has limited effect on its memory-boundness. Neptune delivers consistently better performance on GQA (DC) because it uses masks to apply TensorCore to the dot product in the computation that would otherwise have too few input matrix rows for TensorCore. There is no single baseline that provides high performance like Neptune does across all operators and GPUs.

Only FlexAttn and Tri-Dao Triton (limited to Global and Causal PF) support AMD GPU. Neptune delivers consistently high performance compared to available baselines.

Programmability. Triton-based baselines require a user to write the kernel in their DSL, and FlexAttn uses multiple attention-specific templates developed by the framework authors. For instance, the Tri-Dao Triton kernel is 650 lines of code, while Neptune’s input is vanilla attention (38 lines) and schedule (28 lines) in total; see Appendix A for these inputs.

Mirage Superoptimizer. We discuss Mirage separately, as Mirage does not consistently find valid kernels for all input shapes and only supports variants of Global and Causal operators. Table

Table 3. Speedup of Neptune relative to Mirage on global attention, over a grid of s_q and s_{kv} where $s_q \leq s_{kv}$. Cells where $s_q > s_{kv}$ has an em-dash (–). Cells where Mirage fails to find a valid kernel has a cross (✗). The GPU is A100.

$s_q \backslash s_{kv}$	128	256	512	1024	2048
16	1.98	2.25	1.60	1.30	4.10
32	2.10	1.44	1.47	1.25	1.39
64	1.63	1.92	1.32	1.29	1.58
128	✗	1.83	2.40	1.63	1.85
256	–	2.04	3.35	4.40	6.40
512	–	–	3.64	6.01	✗
1024	–	–	–	6.71	✗
2048	–	–	–	–	✗

Table 4. Speedup of Neptune relative to the best manually optimized library baseline, for 8 operators on 4 GPUs. Each cell is an average over input sequence lengths.

Operator	GPU			
	6000Ada	A5000	A100	MI300
Global (PF)	0.96	0.95	0.84	0.93
Causal (PF)	0.97	0.89	0.81	0.68
GQA (PF)	0.93	0.85	0.80	0.64
Causal (DC)	0.99	0.98	0.99	5.32
GQA (DC)	1.09	1.17	1.24	2.14
ALiBi (PF)	1.65	1.36	1.24	0.98
ALiBi (DC)	1.07	1.12	1.17	5.71
Window (PF)	0.85	0.67	0.70	0.46
Average	1.04	0.98	0.95	1.36

3 shows the speedup of Neptune relative to Mirage. The columns and rows vary by s_{kv} and s_q respectively, where $s_q \leq s_{kv}$. Cells where $s_q > s_{kv}$ are filled with an em dash (–), while cells where Mirage fails to find a valid kernel have a cross (✗). The results show 2.21× lower latency (geomean) across all tested shapes, while Neptune provides deterministic correctness guarantees (unlike Mirage’s probabilistic), and we observed better numerical accuracy with Neptune.

8.2 Neptune vs. Manually Optimized Libraries

Table 4 shows the performance of Neptune kernels vs. kernels from manually optimized libraries. The y-axis shows average relative performance: each framework’s performance relative to Neptune, averaged over sequence lengths. The error bar depicts the range of relative performance over sequence lengths: if a framework does better on some shapes and worse on others, it will have a large error bar. Table 4 omits the SoftCap operator (both prefill and decoding), because existing baseline libraries do not support it. Therefore, this table shows $4 \times 8 \times 8 = 256$ setups.

Out of 256 setups, Neptune has better or equal performance compared to all libraries on 101 setups. On average of all setups, Neptune delivers geomean 1.07× the performance of the best library baseline. There is not a single library that consistently delivers the best performance or outperforms Neptune on all setups. For example, cuDNN has the highest performance on most short-sequence setups, and low performance for long sequences. In contrast, CUTLASS implementations have higher performance on longer sequences for many operators, such as Global (PF) and GQA (DC). Appendix E Figure 17 shows these trends over all input shapes.

The high performance of kernel libraries is a result of manual optimization: they show better performance on the more popular operators (Global, Causal and GQA PF) and GPUs (e.g. A100), and lower performance than Neptune otherwise. Neptune failed to optimize Window (PF) because its TVM-based loop analysis could not identify the proper condition for loop partitioning.

8.3 Scalability Test

Optimizations in Neptune involve various trade-offs that introduce a small amount of resource overhead, such as register and shared memory (SMEM) usage, for significant performance gains. To show that Neptune’s resource usage is acceptable, we stress-test Neptune on increasing batch sizes until we reach the limit of available GPU memory. Figure 9 shows the throughput (in TFLOPs/sec) of Neptune’s kernel and multiple baselines for the GQA (PF) operator on RTX A5000. Appendix E presents throughput results over all operators and GPUs. We use a sequence length of 8192 as it allows us to test more batch sizes up to 32. The throughput of all kernels decreases as input size increases, as longer workloads are more likely to trigger GPUs’ clock and power throttling. Neptune’s kernel remains close to the best baseline (slightly behind cuDNN) for all tested batch sizes.

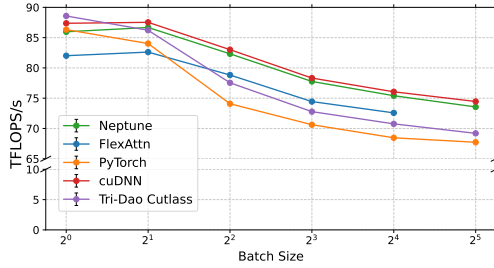


Fig. 9. Throughput of Neptune kernels and baselines over increasing batch sizes, evaluated on RTX A5000 for the GQA (PF) operator. The y-axis shows throughput in TFLOPS/sec, and the x-axis shows batch size.

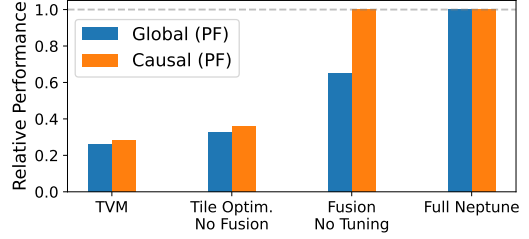


Fig. 10. Ablation studies of the Neptune kernel performance on sequence length 512 on RTX 6000 Ada. The x-axis labels mark the component of Neptune we keep and remove.

We also directly measure the number of registers, SMEM, and global memory used in Neptune’s kernels. On average, over all the Nvidia GPU setups, Neptune uses $0.68\times$ the registers of the baseline that uses the most registers, and 47% the SMEM of the baseline that uses the most SMEM. The number of registers and the amount of SMEM mainly depend on the operator and change little over input size. We did not observe that they have negative impact on speed.

8.4 Other Studies for Attention Models

Ablation Studies. To understand the impact of each component of Neptune, we perform ablation studies on the performance of Neptune’s kernels. We select the Global and Causal operators in prefill with sequence length 512 on RTX 6000 Ada. Figure 10 shows the results of this experiment. On the left, we start from baseline TVM with no Neptune components. As we move to the right, we first add Neptune’s tile optimizations, then Neptune’s fusion algorithms, and finally Neptune’s autotuner. For both operators, fusion is the most impactful optimization, and autotuning provides major improvement for Global and little for Causal. Overall, the effect of the first three steps is consistent over operators and sequence lengths.

Numerical Stability. We evaluate the numerical behavior of Neptune, Tri-Dao, and FlexAttention kernels on the 10 attention operators. Neptune produces kernels with competitive numerical precision. Compared to a FP64 unfused attention kernel baseline, Neptune kernels produce less or equal root mean square (RMS) error than both Tri-Dao and FlexAttention kernels on all prefill operators. Neptune also shows small RMS errors on decode operators. The full result is in Appendix F.

8.5 Performance of Neptune on Non-Attention Operators

Figure 11 presents the performance of Neptune compared to PyTorch for the L^2 norm, RMSNorm, and performer operators. The y-axis shows the relative performance of Neptune compared to PyTorch (higher is better). The x-axis shows the sequence length in the input tensor (log scale).

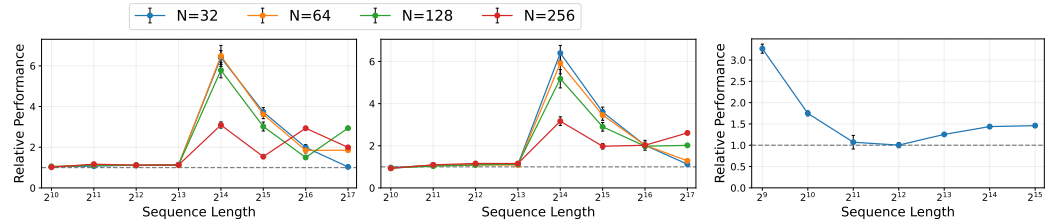


Fig. 11. Relative performance of Neptune kernels for L^2 norm (left), RMSNorm (middle), and performer (right), compared to PyTorch baseline. Higher is better.

For L^2 norm and RMSNorm, the sequence length is the number of columns M , and multiple lines

represent different numbers of rows N . These two operators show similar performance trends with a mode switch. On $M \leq 8192$, Neptune performs slightly better than PyTorch. Here both Neptune and PyTorch use a single kernel to run the computation. PyTorch produces a single kernel without deep fusion techniques like rolling update, because the input is small enough for a threadblock to communicate entire rows internally without going to global memory. When $M > 8192$, Neptune starts to significantly outperform PyTorch, as PyTorch now needs to use multiple kernels for the computation. By fusing reductions together into a single kernel, Neptune avoids two sources of overhead: kernel launch overhead and data movement overhead. Neptune's performance gain is the largest at $M = 16384$, and decreases as kernel launch overhead becomes less significant.

For Performer, Neptune significantly outperforms PyTorch on smaller sequences ($L \leq 1024$). The performance gain decreases as sequence length increases until $L = 2048$, and increases again afterwards. For all sequence lengths, Neptune fuses the entire performer operator into two kernels, while PyTorch uses 17 kernels for the computation. Compared to L^2 norm and RMSNorm, fusion is always beneficial for performer due to the large number of computation steps. Similar to L^2 norm and RMSNorm, the performance gain is larger for smaller sequences where kernel launch overhead dominates, and data movement overhead becomes more significant as sequence length increases.

9 RELATED WORK

Manual Tensor Operators. FlashAttention [14] details an attention-specific fusion algorithm with insights similar to Neptune's general rolling update, and FlashDecoding [15] proposes a fuse-and-split algorithm for decoding attention, similar to Neptune's split-k update. These works present transformed operators instead of a general means of transforming operators: a library [16] offers kernels for common attention variants, but specialized variants require manual implementation. Manually implemented operators lack the flexibility to apply to diverse computations, and may struggle to compose with automated compiler optimizations.

Tile-Based Compilers. Many tensor compilers and programming frameworks decompose tensor-level operations into operations over tensor tiles, which often map directly to the underlying hardware. Tile programming frameworks, such as Triton [39], Tilelang [44], and more [4, 28, 38], provide tiles as a core abstraction and map them to target devices like GPU TensorCores. Roller [57] recursively tiles kernels to match unknown accelerator architectures. Tile compilers automate much of low-level intra-tile optimization, but leave the burden of high-level optimization like block tiling and operator fusion to the programmer. As stand-alone systems, tile compilers function similarly to manual operators at a slightly higher level of abstraction, with similar limitations.

Scheduling Kernel Compilers. Halide [33], TVM [6], and other compilers [19, 56] separate the specification of a tensor operator from its implementation, called schedule. They search for schedules using autotuning or programmer input, but the number of steps required to reach a high-performance schedule can overwhelm programmers. Schedule primitives typically provide basic composable transformations, but not algebra-rewriting transformations like Neptune provides.

Auto-Schedulers and Autotuners. Tensor compilers use autotuners to search for high-performance tensor programs on specific hardware, such as [2, 23, 25] in Halide, and AutoTVM and Anso [54] in TVM. Felix [51] provides a novel formulation of tensor program autotuning as a continuous (gradient descent) search problem. Triton [39] provides an enumerative search while leaving the tuning knob definitions to the user. Autotuning is also used to explore performance-accuracy trade-offs of deep learning models [37, 52]. Scheduling compilers also integrate with auto-schedulers to define search spaces for autotuners, such as Anso [54] and MetaSchedule [36] in TVM. These auto-schedulers generate a list of candidate schedules for an autotuner to search from. As Neptune provides operator fusion as new schedule primitives, it can compose with auto-schedulers to further automate schedule creation.

PyTorch-Based Optimizations. FlexAttention [18] is a PyTorch framework that extends FlashAttention to allow customizing a mask and an element-wise score function. It improves coverage for attention variants but is still limited to attention, as it generates code from manual program templates. In parallel to our work, Flashlight [50] proposes semantic fusion to convert two adjacent tensor kernels to a single kernel with online reduction. This semantic fusion does not extend to more complex data flow like Neptune does, and the design of the Flashlight system is highly specific to PyTorch.

Graph- and Model-level Optimizations. Many tensor compilers like PyTorch [31], XLA/HLO [30], TVM [6], TASO [20], and others [1, 27, 33, 47] describe entire deep learning models as operator graphs and apply graph optimizations. Operator graphs hide the detailed description per operator, allowing optimizations to happen at a high level. They are distinct from kernel-level optimizations in kernel compilers, although they can co-exist in the same framework as a form of hierarchical optimization. Graph-level frameworks can provide operator fusion [7, 27] and algebraic rewriting [20, 21, 26, 33, 42, 48]. However, they do not support advanced operator fusion in Neptune or allow them to be expressed as rewrite rules. Operator graphs are not the ideal abstraction for Neptune’s operator fusion, which requires loop fusion and tile-level rewriting to occur in tandem.

Superoptimization Compilers. Some tensor compilers [20, 32, 40, 43] apply superoptimization, which transforms the given program in potentially semantic-breaking ways, and impose additional checks to select correct implementations. Most superoptimizers transform a single level of abstraction at a time (i.e. only graph rewrites or only instruction rewrites), unlike advanced fusions in Neptune which are expressed as a combination of loop and algebra transformations. As an exception, Mirage [47] is a hierarchical superoptimizer that jointly transforms the program at thread, tile, and operator levels. Mirage uses a probabilistic correctness test on integer inputs, producing programs that have (high) probability of being correct, contrasting with Neptune’s formal correctness. The test does not account for the numerical error on floating-point inputs, and Mirage has visible numerical instability in some cases such as attention. The search algorithm in Mirage can take minutes to hours, and may end up with no valid kernels or find kernels with unpredictable performance (as observed in experiments in §8).

Nondeterministic/Approximate Dependency Breaking. Some works apply dependency-breaking transformations, e.g., Hogwild [34] for distributed neural network training and Deiana et al. [17] for thread-level parallelization. These works leverage application-level tolerance to error or nondeterminism to break performance-limiting dependencies. Hogwild declines to repair broken dependencies, creating an approximate result, while Deiana et al. use runtime repair suitable for CPU workloads but infeasible for deterministic tensor workloads. Neptune differs from these works as it provides static transformations that recover equivalence (on real numbers) at compile time.

10 CONCLUSION

We presented Neptune, a novel tensor compiler for advanced ML operator fusion. Neptune shows, for the first time, that advanced fusion algorithms can be included in a standard compiler schedule, to transform plain attention into kernels like FlashAttention and FlashDecoding, reaching and exceeding state-of-the-art kernel performance. Our subsequent work Nautilus [53] shows the benefits of our transformations on new GPU architectures (Hopper and Blackwell) and automates scheduling. Neptune opens a new direction towards bringing such optimizations fully within the scope of optimizing tensor compilers. We anticipate that this integration will pave the way for automatic discovery of new efficient advanced ML operators on a broad range of emerging AI hardware platforms.

Acknowledgments

This research was supported in part by the NSF grant No. CCF-2217144 and the IBM-Illinois Discovery Accelerator Institute. This research used DeltaAI advanced computing and data resource, supported by the NSF (award OAC 2320345) and the State of Illinois.

REFERENCES

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
- [2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38 (2019). <https://api.semanticscholar.org/CorpusID:196834556>
- [3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.
- [4] The JAX Authors. 2024. Pallas: a JAX kernel language. <https://docs.jax.dev/en/latest/pallas/index.html>
<https://docs.jax.dev/en/latest/pallas/index.html>
- [5] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. *CoRR* abs/2006.12645 (2020). arXiv:2006.12645 <https://arxiv.org/abs/2006.12645>
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 <http://arxiv.org/abs/1802.04799>
- [7] Zhaodong Chen, Andrew Kerr, Richard Cai, Jack Kosaian, Haicheng Wu, Yufei Ding, and Yuan Xie. 2024. EVT: Accelerating Deep Learning Training with Epilogue Visitor Tree. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) (ASPLOS '24). 301–316. <https://doi.org/10.1145/3620666.3651369>
- [8] Krzysztof Marcin Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamás Szilárd, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. 2021. Rethinking Attention with Performers. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. <https://openreview.net/forum?id=Ua6zuk0WRH>
- [9] NVIDIA Corporation. 2021. NVIDIA cuSPARSELt. <https://docs.nvidia.com/cuda/cusparselt/types.html>.
- [10] NVIDIA Corporation. 2024. NVIDIA A10 Tensor Core GPU. <https://www.nvidia.com/en-us/data-center/a100/>.
- [11] NVIDIA Corporation. 2024. NVIDIA RTX 6000 Ada-generation Graphics Card. <https://www.nvidia.com/en-us/design-visualization/rtx-6000/>.
- [12] NVIDIA Corporation. 2024. NVIDIA RTX A5000 Graphics Card. <https://www.nvidia.com/en-us/design-visualization/rtx-a5000/>.
- [13] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*.
- [14] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.
- [15] Tri Dao, Grigory Sizov, Francisco Massa, and Daniel Haziza. 2023. Flash-decoding for long-context inference. <https://pytorch.org/blog/flash-decoding/>
- [16] Dao-AILab. 2023. FlashAttention. <https://github.com/Dao-AILab/flash-attention>.
- [17] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications. *SIGPLAN Not.* 53, 2 (March 2018), 432–447. <https://doi.org/10.1145/3296957.3173181>
- [18] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex Attention: A Programming Model for Generating Optimized Attention Kernels. arXiv:2412.05496 [cs.LG] <https://arxiv.org/abs/2412.05496>
- [19] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.
- [20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

- [21] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems* 1 (2019), 27–39.
- [22] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. 2022. xFormers: A modular and hackable Transformer modelling library. <https://github.com/facebookresearch/xformers>.
- [23] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* 37, 4, Article 139 (2018). <https://doi.org/10.1145/3197517.3201383>
- [24] lucidrains. 2026. performer-pytorch. <https://github.com/lucidrains/performer-pytorch>. Accessed: 2026-03-24.
- [25] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016). <https://doi.org/10.1145/2897824.2925952>
- [26] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoab Kamil. 2020. Verifying and improving Halide’s term rewriting system with program synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (Nov. 2020), 28 pages. <https://doi.org/10.1145/3428234>
- [27] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) (PLDI 2021)*. 883–898. <https://doi.org/10.1145/3453483.3454083>
- [28] NVIDIA. 2021. CUTLASS: CUDA Templates for Linear Algebra Subroutines. <https://github.com/NVIDIA/cutlass>.
- [29] OpenAI. 2024. Fused Attention – Triton Documentation. <https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html>.
- [30] OpenXLA. [n. d.]. XLA. <https://openxla.org/xla>. <https://openxla.org/xla>
- [31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
- [32] Jacques Arnaud Pienaar, Mangpo Phothilimthana, Max Willsey, Remy Wang, Sudip Roy, and Yichen Yang. 2021. Equality Saturation for Tensor Graph Superoptimization. In *MLSys*.
- [33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- [34] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger (Eds.), Vol. 24. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf
- [35] Alex Rogozhnikov. 2020. Einops: Clear and Versatile Tensor Manipulations for Deep Learning. <https://github.com/arogozhnikov/einops>. GitHub repository.
- [36] Junru Shao, Xiyu Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. In *Advances in Neural Information Processing Systems*, Vol. 35. https://proceedings.neurips.cc/paper_files/paper/2022/file/e894eafae43e68b4c8dfdacf742bcbf3-Paper-Conference.pdf
- [37] Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Akash Kothari, Ben Schreiber, Elizabeth Wang, Yasmin Sarita, Nathan Zhao, Keyur Joshi, Vikram S Adve, Sasa Misailovic, and Sarita V Adve. 2021. ApproxTuner: a compiler and runtime system for adaptive approximations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
- [38] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] <https://arxiv.org/abs/2410.20399>
- [39] Philippe Tillet, H. T. Kung, and David Cox. 2019. 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*. 10–19. <https://doi.org/10.1145/3315508.3329973>
- [40] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 267–284. <https://www.usenix.org/conference/osdi22/presentation/unger>

- [41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
- [42] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 37–54. <https://www.usenix.org/conference/osdi21/presentation/wang>
- [43] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 37–54. <https://www.usenix.org/conference/osdi21/presentation/wang>
- [44] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. 2025. TileLang: A Composable Tiled Programming Model for AI Systems. arXiv:2504.17577 [cs.LG] <https://arxiv.org/abs/2504.17577>
- [45] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4, 3 (1962), 419–420. <https://doi.org/10.1080/00401706.1962.10490022>
- [46] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 77–89. <https://doi.org/10.1109/CGO51591.2021.9370330>
- [47] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: A {Multi-Level} Superoptimizer for Tensor Programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 21–38.
- [48] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268.
- [49] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. arXiv preprint arXiv:2501.01005 (2025). <https://arxiv.org/abs/2501.01005>
- [50] Bozhi You, Irene Wang, Zelal Su Mustafaoglu, Abhinav Jangda, Angélica Moreira, Roshan Dathathri, Divya Mahajan, and Keshav Pingali. 2026. Flashlight: PyTorch Compiler Extensions to Accelerate Attention Variants. *Proceedings of Machine Learning and Systems* (2026).
- [51] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. 2024. Felix: Optimizing Tensor Programs with Gradient Descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. 367–381. <https://doi.org/10.1145/3620666.3651348>
- [52] Yifan Zhao, Hashim Sharif, Peter Pao-Huang, Vatsin Ninad Shah, Arun Narenthiran Sivakumar, Mateus Valverde Gasparino, Abdulrahman Mahmoud, Nathan Zhao, Sarita Adve, Girish Chowdhary, Sasa Misailovic, and Vikram Adve. 2023. ApproxCaliper: A Programmable Framework for Application-aware Neural Network Optimization. In *Proceedings of Machine Learning and Systems* 5.
- [53] Yifan Zhao, Yuchen Yang, Matei Budiu, and Sasa Misailovic. 2026. Nautilus: An Auto-Scheduling Tensor Compiler for Efficient Tiled GPU Kernels. arXiv:2604.14825 [cs.PL] <https://arxiv.org/abs/2604.14825>
- [54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. Article 49.
- [55] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (New York, New York) (ISCA '22)*. 874–887. <https://doi.org/10.1145/3470496.3527440>
- [56] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '20)*. 859–873. <https://doi.org/10.1145/3373376.3378508>
- [57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248. <https://www.usenix.org/conference/osdi22/presentation/zhu>

APPENDIX

A NEPTUNE INPUTS

Neptune takes as input a compute definition of the operator to optimize, and a schedule that describes how to optimize it.

Figure 12 shows one possible Neptune input, a flexible compute definition for attention. It allows customizing bias and mask conditions, and in 38 lines of code it covers all operators in Table 1 except GQA operators.

Figure 13 shows the schedule that pairs with the compute definition. In 28 lines of code, it applies rolling update to fuse the attention computation into a single loop nest, and produces many of the high-performance kernels in our evaluation.

```

1 def create_general_attention(
2     B: int, N: int, QS: int, KVS: int, H: int, mask_cond: Callable, score_mod: Callable
3 ):
4     q = placeholder((B, N, QS, H), "float16", name="q")
5     k = placeholder((B, N, KVS, H), "float16", name="k")
6     v = placeholder((B, N, KVS, H), "float16", name="v")
7     p = batch_matmul(q, k, trans_b=True, out_dtype="float32")
8     score = compute(
9         p.shape, lambda *ax: if_then_else(mask_cond(*ax), score_mod(p(*ax), *ax), float("-inf")),
10        name="score_mod")
11    j = reduce_axis((0, KVS), name="j")
12    s_max = compute(
13        (B, N, QS), lambda b, n, i: max(score(b, n, i, j), axis=j), name="softmax_maxelem")
14    s_exp = compute(
15        (B, N, QS, KVS), lambda b, n, i, j: exp(score(b, n, i, j) - s_max(b, n, i)),
16        name="softmax_exp")
17    s_expsum = compute(
18        (B, N, QS), lambda b, n, i: sum(s_exp(b, n, i, j), axis=j), name="softmax_expsum")
19    s_exp = compute(
20        p.shape, lambda *axes: s_exp(*axes).astype(q.dtype), name="softmax_exp_f16")
21    sv = batch_matmul(s_exp, v, trans_b=False, out_dtype="float32")
22    sv = compute(
23        sv.shape, lambda b, n, i, j: sv(b, n, i, j) / s_expsum(b, n, i), name="softmax_norm")
24    return compute(sv.shape, lambda *i: sv(*i).astype(q.dtype), "cast")

```

Fig. 12. Compute definition for the attention operator, which Neptune takes as input. This definition covers all operators in Table 1 except GQA operators.

```

1 def schedule_attn_with_rolling_update(sch: Schedule):
2     b0 = sch.get_block("batch_matmul_1")
3     b1 = sch.get_block("T_score_mod")
4     b2 = sch.get_block("T_softmax_maxelem")
5     b3 = sch.get_block("T_softmax_exp")
6     b4 = sch.get_block("T_softmax_exp_cast")
7     b5 = sch.get_block("T_batch_matmul_NN")
8     b6 = sch.get_block("T_softmax_expsum")
9     b7 = sch.get_block("T_softmax_norm")
10    b8 = sch.get_block("T_cast")
11    *axes, i, j, k = sch.get_loops(b0)
12    i0, j0 = sch.tile([i, j], [128, 32])
13    sch.compute_at(sch.cache_read(b0, 0, "shared"), i0)
14    sch.bind_block_idx([*axes, i0], ["blockIdx.x", "blockIdx.y", "blockIdx.z"])
15    sch.reverse_compute_at(b1, j0)
16    b2rf = sch.rolling_update(b2, j0, factor_axis=0)
17    b6rf = sch.rolling_update(b6, j0, factor_axis=0)
18    b5rf = sch.rolling_update(b5, j0, factor_axis=0)
19    sch.reverse_compute_at(b7, i0)
20    sch.reverse_compute_at(b8, i0)
21    for blk in [b0, b1, b2, b2rf, b3, b4, b5, b5rf, b6, b6rf, b7]:
22        sch.set_scope(blk, 0, "shared")
23    sch.split_scan_buffer(b2, j0, 0)
24    sch.decompose_reduction(b5, j0)
25    sch.decompose_reduction(b6, j0)

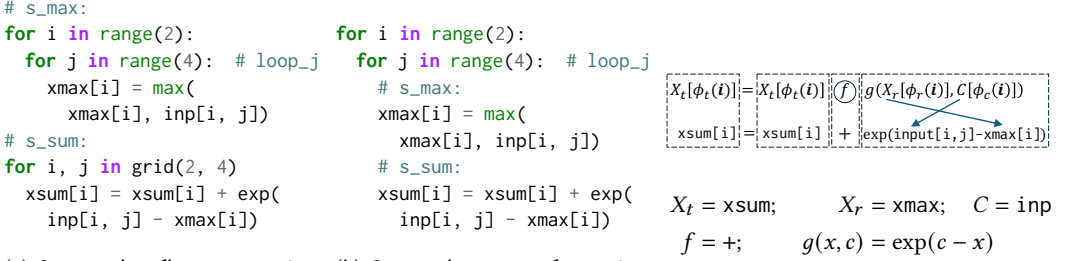
```

Fig. 13. Schedule for the attention operator, which Neptune takes as input. This schedule pairs with the compute definition in Figure 12.

B STEP-BY-STEP ALGORITHM APPLICATION ON MOTIVATIONAL EXAMPLE

Here we include the detailed steps of our rolling update algorithm for the example from Figure 1a.

B.1 Rolling Update



(a) Step 1: dataflow reorganization. This step inlines `s_exp` (into `s_sum`) and finds $\mathcal{L}_r = \{s_max\}$.
 (b) Step 2: loop transformation. Naive loop fusion fuses `s_max` and `s_max` under `loop_j`.
 (c) Step 3: pattern matching on the compute statement of $L_t = s_sum$.

$$y = g(x, c) = \exp(c - x) \Rightarrow c = g_c^{-1}(x, y) = x + \ln(y)$$

$$h(t, r, r') = g(r', r + \ln(t)) = \exp(r - r') \cdot t$$

Proving commutativity with f :

$$h(x+y, r, r') = \exp(r-r') \cdot (x+y) = h(x, r, r') + h(y, r, r')$$

(d) Step 4: finding the repair function h . Find the inverse of g , use Equation 5 to find h , and prove Equation 4 (commutativity with f) for h .

For `s_max`, cache its output `xmax` for the previous iteration:

$$xmax_0[i] := xmax[i]^{(j-1)}; \quad xmax_1[i] := xmax[i]^{(j)}$$

Apply $h(t, r, r')$ with $r = xmax_0[i]$ and $r' = xmax_1[i]$:

$$xsum[i] = \exp(xmax_0[i] - xmax_1[i]) \cdot xsum[i] + \exp(inp[i, j] - xmax_1[i])$$

(e) Step 5: applying the repair function.

Fig. 14. All intermediate results of rolling update, when applied to Figure 1a.

B.2 Privatization

Figure 15 presents privatization combined with rolling update in Neptune. On top of the rolling update output in Figure 1c, we privatize `s_max` and `s_sum` with a split size of 2. `xmax_1p` and `xsump` are the local arrays.

```

for i in range(2):
    xmax_0[i] = -inf
    for j1 in range(2):
        for j2 in range(2):
            # max_local
            xmax_1p[i, j1] = max(xmax_1p[i, j1], inp[i, j1 * 2 + j2])
            # max_global
            xmax_1[i] = max(xmax_1[i], xmax_1p[i, j1])
        for j2 in range(2):
            # sum_local
            xsump[i, j1] += exp(inp[i, j1 * 2 + j2] - xmax_1[i])
            xsum[i] = exp(xmax_0[i] - xmax_1[i]) * xsum[i] + xsump[i, j1]
        xmax_0[i] = xmax_1[i]

```

Fig. 15. Privatization combined with rolling update in Neptune.

C ADDITIONAL EXAMPLE OPERATORS AMENABLE TO NEPTUNE TRANSFORMATIONS

Attention definition. The Attention operator is a sequence of four compute steps: matrix multiplication (matmul), element-wise score computation (division), softmax, and another matmul. Attention typically use a numerically stable softmax implementation, subtracting the input by per-row maximum to prevent exponential overflow. Equation 12 shows the compute steps of Attention over 2D inputs $q, k, v \in \mathbb{R}^{L \times D}$. i and j are indices along the L (sequence length) dimension, and d is the index along the D (number of features) dimension.

$$\begin{aligned}
 p_{ij} &= \sum_d q_{id} v_{jd}; & p'_{ij} &= \frac{p_{ij}}{\sqrt{D}} && \text{matmul, division} \\
 m_i &= \max_j p'_{ij}; & e_{ij} &= \exp(p'_{ij} - m_i); & s_i &= \sum_j e_{ij} && \text{softmax (steps 1-3)} && (12) \\
 o_{id} &= \sum_j e_{ij} v_{jd}; & o'_{id} &= \frac{o_{id}}{s_i} && \text{matmul, softmax (step 4)}
 \end{aligned}$$

Performers. Performers [8] is an alternative transformer architecture that uses non-softmax computation to approximate softmax. The performer operator takes three inputs Q, K , and V with a sequence length (L) and a number of features (D) dimension, similar to attention. Performer applies a randomly sampled projection matrix of shape $M \times D$ along with certain normalization on Q and K to produce $Q', K' \in \mathbb{R}^{L \times M}$. Then performer computes $A = Q'K'V$ and applies additional normalization on A without needing to compute a softmax. These normalization steps provide multiple reductions that Neptune's reduction fusion applies to.

Neptune fuses performer into two loop nests, applying reduction fusion four times. Two fusions have computation pattern g_1 , and the other two have distinct computation patterns g_2 and g_3 :

$$\begin{aligned}
 g_1(r_1, c_1, c_2) &= \alpha \exp(c_1 - r_1 - \beta c_2); & h_1(t, r, r') &= t \exp(r' - r) \\
 g_2(r_1, c_1, c_2, c_3) &= \alpha c_3 \exp(c_1 - r_1 - \beta c_2); & h_2(t, r, r') &= t \exp(r' - r) \\
 g_3(r_1, r_2, c_1, c_2, c_3) &= \alpha \exp(c_1 - r_1 - \beta c_2 c_3) / r_2; & h_3(t, r_1, r'_1, r_2, r'_2) &= t(r_2 / r'_2) \exp(r_1 - r'_1)
 \end{aligned}$$

α and β are compile-time constants. The reduction function in all these cases is $f(x, y) = x + y$. g_1 , g_2 , and g_3 are all different from any softmax-based attention variants. The solution that Neptune finds is also given as h_1, h_2 , and h_3 . The last case g_3 is a fusion between one reduction and two producers, as the two reduction inputs r_1 and r_2 indicate, which Neptune is also capable of (§4.2, Eqns. 8 and 9).

Upstream optimizations can change the program that reaches Neptune fusion. One optimization is to delay the normalization on $A = Q'K'V$ to the end of the computation. This optimization can improve the performance and numerical stability of Neptune-fused kernel, as the normalization involves a division that is now delayed outside of the rolling loop and only applied once. When this optimization is applied, the fourth fusion (g_3) will have an identical compute pattern as the third one (g_2).

D PROOFS OF ROLLING UPDATE CORRECTNESS THEOREMS

We use these abbreviations for the following proofs

$$C[\phi_c(\mathbf{i}, \mathbf{j}')] =: C_{j'} \quad X_t[\phi_t(\mathbf{i})]^{(j)} =: X_t^{(j)} \quad X_r[\phi_r(\mathbf{i})]^{(j)} =: X_r^{(j)}.$$

essentially dropping the tensor access function and the map iteration vector \mathbf{i} , because all the proofs involve a single loop nest where \mathbf{i} is fixed.

PROOF OF LEMMA 4.3. We first show that the constructive solution of h , provided in Eq. 5, is a solution to condition Eq. 3. Since g_c^{-1} is the second-argument inverse of g , we have

$$g_c^{-1}(x, g(x, c)) = c$$

In Eq. 5, substitute $t = g(r, c)$ to get

$$h(g(r, c), r, r') = g(r', g_c^{-1}(r, g(r, c))) = g(r', c)$$

which is exactly Eq. 3. Next, given that h satisfies the conditions Eqns. 3 and 4, we prove that h satisfies Def. 4.2 which is an equality. The left-hand side of the equality is h applied to the reduce expression (rewritten using our abbreviations)

$$\mathcal{R}\left(f, 0 \preceq j' \preceq j_0, g\left(X_r^{(j_f)}, C_{j'}\right)\right) =: R_{j_0}^{(j_f)}$$

and the right-hand side is the updated reduce expression

$$\mathcal{R}\left(f, 0 \preceq j' \preceq j_0, g\left(X_r^{(j_t)}, C_{j'}\right)\right) =: R_{j_0}^{(j_t)}$$

The subscript j_0 in R_{j_0} indicates the upper bound of the reduce range. Therefore, we are to prove

$$\forall j_0, \mathbf{j}_f, \mathbf{j}_t \in \mathcal{D}^s, \mathbf{j}_f \preceq \mathbf{j}_t \preceq j_0 \quad h\left(R_{j_0}^{(j_f)}, X_r^{(j_f)}, X_r^{(j_t)}\right) = R_{j_0}^{(j_t)}$$

which we prove by induction over j_0 . We introduce j as the induction variable to go from 0 to j_0 , while all tags are fixed, and maintain the inductive hypothesis

$$\forall 0 \preceq j \preceq j_0 \quad h\left(R_j^{(j_f)}, X_r^{(j_f)}, X_r^{(j_t)}\right) = R_j^{(j_t)} \quad (13)$$

- Base case: when $j = 0$, $R_0^{(j_f)}$ and $R_0^{(j_t)}$ are each a single term, so we spell out these terms in the inductive hypothesis to get

$$h\left(g\left(X_r^{(j_f)}, C_0\right), X_r^{(j_f)}, X_r^{(j_t)}\right) = g\left(X_r^{(j_t)}, C_0\right)$$

which is true by applying Eq. 3.

- Inductive step: for any j where the inductive hypothesis Eq. 13 is true, we move on to $\text{next}(j)$ by adding one term to $R_j^{(j_f)}$:

$$R_{\text{next}(j)}^{(j_f)} = R_j^{(j_f)} \oplus g\left(X_r^{(j_f)}, C_{\text{next}(j)}\right)$$

Apply h to both sides, apply Eq. 4, then apply Eq. 3, and finally apply the inductive hypothesis

on the underlined term, to get

$$\begin{aligned}
& h\left(\underline{R_{\text{next}(j)}^{(j_f)}}, X_r^{(j_f)}, X_r^{(j_i)}\right) = h\left(R_j^{(j_f)} \textcircled{f} g\left(X_r^{(j_f)}, C_{\text{next}(j)}\right), X_r^{(j_f)}, X_r^{(j_i)}\right) \\
& = h\left(R_j^{(j_f)}, X_r^{(j_f)}, X_r^{(j_i)}\right) \textcircled{f} h\left(g\left(X_r^{(j_f)}, C_{\text{next}(j)}\right), X_r^{(j_f)}, X_r^{(j_i)}\right) \\
& = \underline{h\left(R_j^{(j_f)}, X_r^{(j_f)}, X_r^{(j_i)}\right)} \textcircled{f} g\left(X_r^{(j_i)}, C_{\text{next}(j)}\right) \\
& = R_j^{(j_i)} \textcircled{f} g\left(X_r^{(j_i)}, C_{\text{next}(j)}\right) = R_{\text{next}(j)}^{(j_i)}
\end{aligned}$$

Therefore, the inductive hypothesis is true for $\text{next}(j)$.

By induction, the theorem is true for all $j \preceq j_0$, and therefore true for j_0 . \square

PROOF OF THEOREM 4.1. The goal of this proof is to show that the result of rolling update transformed program matches that of the original program. We copy the explicit expression of X_t in the original program from Eq. 10, and apply our abbreviations:

$$X_t^{(m)} = \mathcal{R}\left(f, 0 \preceq j' \preceq m, g\left(X_r^{(m)}, C_{j'}\right)\right) \quad (14)$$

On the other hand, the repaired program Eq. 7 is (with tag added for clarity)

$$X_t^{(j)} = h\left(X_t^{(\text{prev}(j))}, X_r^{(\text{prev}(j))}, X_r^{(j)}\right) \textcircled{f} g\left(X_r^{(j)}, C_j\right)$$

and we assert it produces this value at iteration j :

$$X_t^{(j)} = \mathcal{R}\left(f, 0 \preceq j' \preceq j, g\left(X_r^{(j)}, C_{j'}\right)\right) =: R_j^{(j)} \quad (15)$$

If this is true, then we have $X_t^{(m)} = X_t^{(m)}$ by substituting m for j in Eq. 15 and comparing to Eq. 14. We prove this Eq. 15 by induction on j .

- Base case: when $j = 0$, the recurrent X_0 has iterated once:

$$X_t^{(0)} = g\left(X_r^{(0)}, C_0\right)$$

and the explicit $X_t^{(0)}$ has a single term:

$$R_0^{(0)} = g\left(X_r^{(0)}, C_0\right)$$

and they are equal, so the inductive hypothesis is true.

- Inductive step: for any j where the inductive hypothesis is true, the next iteration $\text{next}(j)$ updates the recurrent X_t once. We write down the updated value, apply the inductive hypothesis to convert to explicit form, then use the tag-updating property of h to get

$$\begin{aligned}
& X_t^{(\text{next}(j))} \\
& = h\left(X_t^{(j)}, X_r^{(j)}, X_r^{(\text{next}(j))}\right) \textcircled{f} g\left(X_r^{(\text{next}(j))}, C_{\text{next}(j)}\right) \\
& = h\left(R_j^{(j)}, X_r^{(j)}, X_r^{(\text{next}(j))}\right) \textcircled{f} g\left(X_r^{(\text{next}(j))}, C_{\text{next}(j)}\right) \\
& = R_j^{(\text{next}(j))} \textcircled{f} g\left(X_r^{(\text{next}(j))}, C_{\text{next}(j)}\right) \\
& = R_{\text{next}(j)}^{(\text{next}(j))}
\end{aligned}$$

Therefore, the inductive hypothesis is true for $\text{next}(j)$.

By induction, Eq. 15 is true for all $j \preceq \mathbf{m}$. Substituting \mathbf{m} for \mathbf{j} , we get

$$X_t^{(\mathbf{m})} = \mathcal{R} \left(f, 0 \preceq \mathbf{j}' \preceq \mathbf{m}, g \left(X_r^{(\mathbf{m})}, C_{\mathbf{j}'} \right) \right)$$

which matches Eq. 14, and therefore proves the theorem. \square

PROOF OF THEOREM 4.4. The goal of this proof is to show that the result of split-k update transformed program matches that of the original program. The expected expression of X_t in the original program is the same as in Eq. 14:

$$X_t^{(\mathbf{m})} = \mathcal{R} \left(f, 0 \preceq \mathbf{j} \preceq \mathbf{m}, g \left(X_r^{(\mathbf{m})}, C_{\mathbf{j}} \right) \right)$$

Privatization splits this reduction into a local reduction, that produces \mathbf{m}_0 partial results, each result reduced from \mathbf{m}_1 terms. Iterators for \mathbf{m}_0 and \mathbf{m}_1 combine to an iterator for \mathbf{m} similar to how loop tiling works:

$$\forall 0 \preceq \mathbf{j}_0 \preceq \mathbf{m}_0, 0 \preceq \mathbf{j}_1 \preceq \mathbf{m}_1 \quad \exists 0 \preceq \mathbf{j} \preceq \mathbf{m} \quad \mathbf{j}_0 \cdot \mathbf{m}_1 + \mathbf{j}_1 = \mathbf{j}$$

The result of the local reduction is defined by the local reduction that produces it:

$$X_{t,l}[\mathbf{j}_0] = X_{t,l}[\mathbf{j}_0] \textcircled{f} g \left(X_{r,l}[\mathbf{j}_0], C_{\mathbf{j}_0 \cdot \mathbf{m}_1 + \mathbf{j}_1} \right)$$

\mathbf{j}_0 shows up in the index of tensor $X_{t,l}$ because the local reduction produces \mathbf{j}_0 partial results. We convert this program into the following expression for $X_{t,l}$:

$$X_{t,l}[\mathbf{j}_0] = \mathcal{R} \left(f, 0 \preceq \mathbf{j}_1 \preceq \mathbf{m}_1, g \left(X_{r,l}[\mathbf{j}_0], C_{\mathbf{j}_0 \cdot \mathbf{m}_1 + \mathbf{j}_1} \right) \right)$$

The global reduction further combines these partial results into a single result over \mathbf{j}_0 iterations:

$$X_{t,g} = X_{t,g} \textcircled{f} h \left(X_{t,l}[\mathbf{j}_0], X_{r,l}[\mathbf{j}_0], X_{r,g} \right)$$

We substitute the expression of $X_{t,l}[\mathbf{j}_0]$ into this program to get

$$X_{t,g} = X_{t,g} \textcircled{f} h \left(\mathcal{R} \left(f, 0 \preceq \mathbf{j}_1 \preceq \mathbf{m}_1, g \left(X_{r,l}[\mathbf{j}_0], C_{\mathbf{j}_0 \cdot \mathbf{m}_1 + \mathbf{j}_1} \right) \right), X_{r,l}[\mathbf{j}_0], X_{r,g} \right)$$

Using the tag-updating property of h , the expression that we expanded from $X_{t,l}[\mathbf{j}_0]$ drops all $X_{r,l}[\mathbf{j}_0]$ and replaces them with $X_{r,g}$:

$$X_{t,g} = X_{t,g} \textcircled{f} \mathcal{R} \left(f, 0 \preceq \mathbf{j}_1 \preceq \mathbf{m}_1, g \left(X_{r,g}, C_{\mathbf{j}_0 \cdot \mathbf{m}_1 + \mathbf{j}_1} \right) \right)$$

which has the following explicit expression:

$$\begin{aligned} X_{t,g} &= \mathcal{R} \left(f, 0 \preceq \mathbf{j}_0 \preceq \mathbf{m}_0, \mathcal{R} \left(f, 0 \preceq \mathbf{j}_1 \preceq \mathbf{m}_1, g \left(X_{r,g}, C_{\mathbf{j}_0 \cdot \mathbf{m}_1 + \mathbf{j}_1} \right) \right) \right) \\ &= \mathcal{R} \left(f, 0 \preceq \mathbf{j} \preceq \mathbf{m}, g \left(X_{r,g}, C_{\mathbf{j}} \right) \right) \end{aligned}$$

which matches the expected expression of $X_{t,g}$ in the original program. \square

E ADDITIONAL EVALUATION

E.1 Compilation Statistics

We provide some statistics Neptune’s compilation pipeline when generating kernels we have evaluated in Section 8. For these evaluated operators, Neptune with 128 autotuning iterations takes 1.5 to 10 minutes to run on each setup. Triton-based tile optimization takes 50% to 90% of the time (depending on the input shape). Neptune’s template-guided optimization (rolling/split-k update) takes 5% to 10% of the time (does not depend on input shape), and the rest is TVM autotuning search algorithm.

E.2 Operator Evaluation

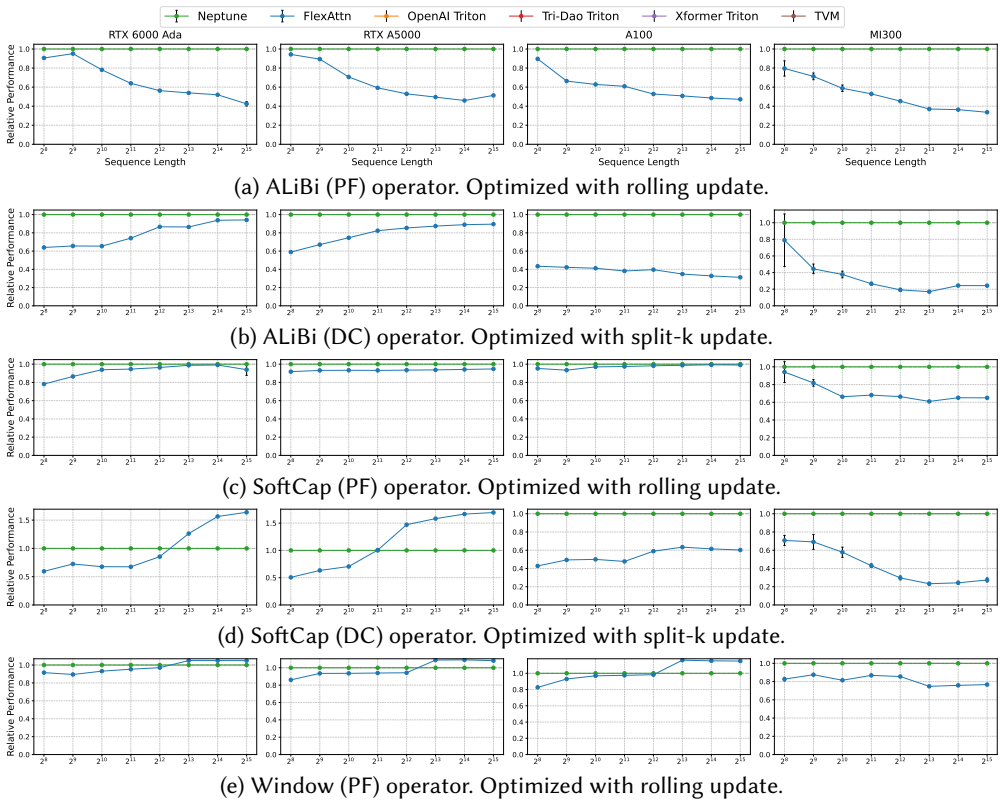


Fig. 16. Performance of Neptune vs. other tensor compilers on the five operators not shown in Figure 8.

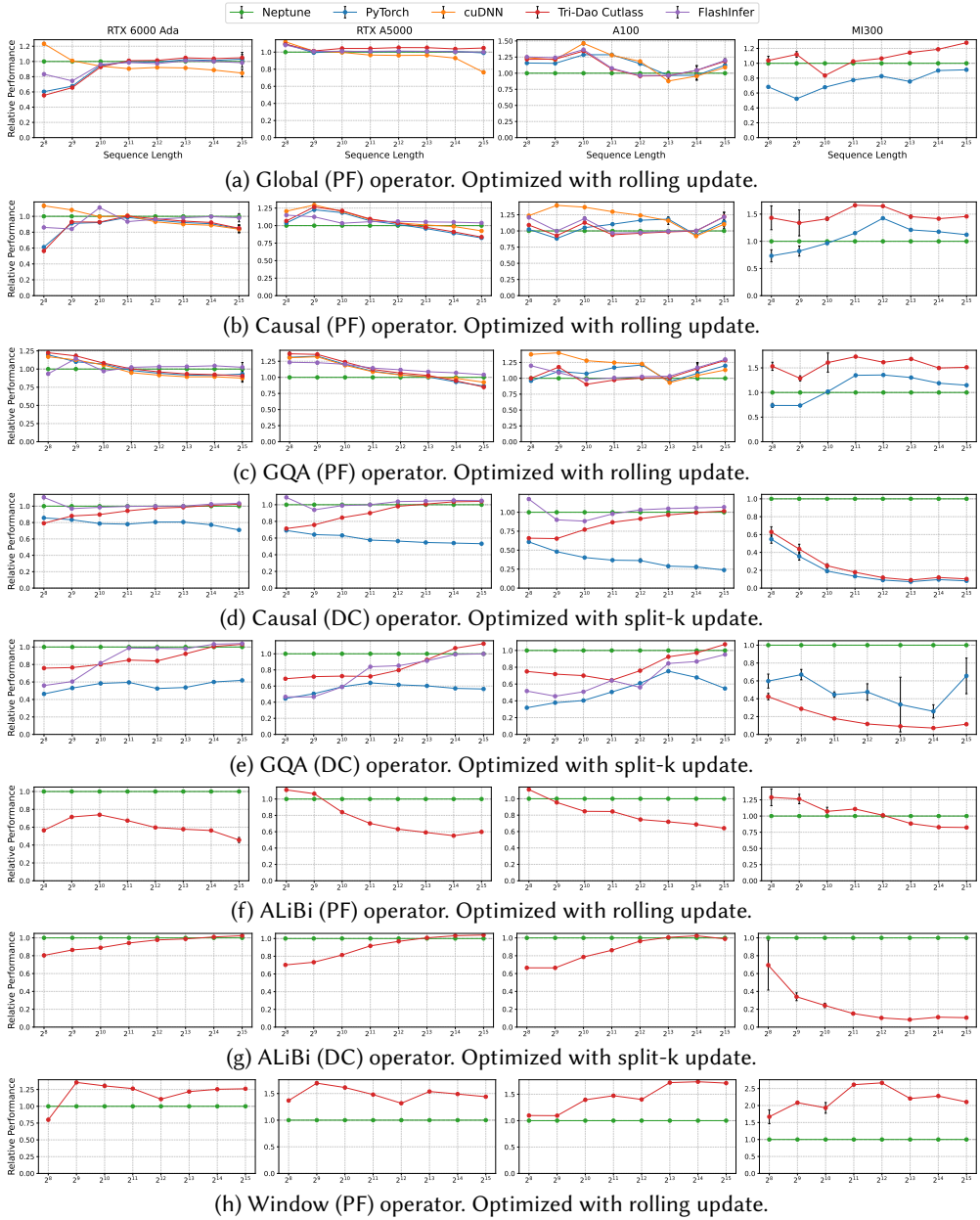


Fig. 17. Performance of Neptune vs. tensor libraries on 8 operators for which library baselines exist (we were unable to find the library implementations of SoftCap PF/DC).

E.3 Scalability Test

Figure 18 extends the scalability test of Figure 9 to all the 10 operators, 4 GPUs and all implementations used in our Evaluation section. We still use a sequence length of 8192 and batch sizes up to 32.

Figures 18a to 18f shows that the throughput of all prefill kernels decrease as input batch size increases. By closely inspecting profiling results, we find that larger workloads are more likely to trigger GPUs’ clock throttling. Figure 19 shows a profile in the Nvidia Nsight System profiler, with a kernel (flash_fwd_kernel) that runs for 281 milliseconds. The GPU compute clock (“GPC Clock Frequency”) starts to throttle within 50 milliseconds after the kernel starts.

In Figures 18g to 18j, the decoding kernels exhibit the opposite trend: throughput increases with batch size. Decoding kernels are much less compute-bound than prefill kernels to trigger clock throttling, and benefit from increasing parallelism at larger batch sizes.

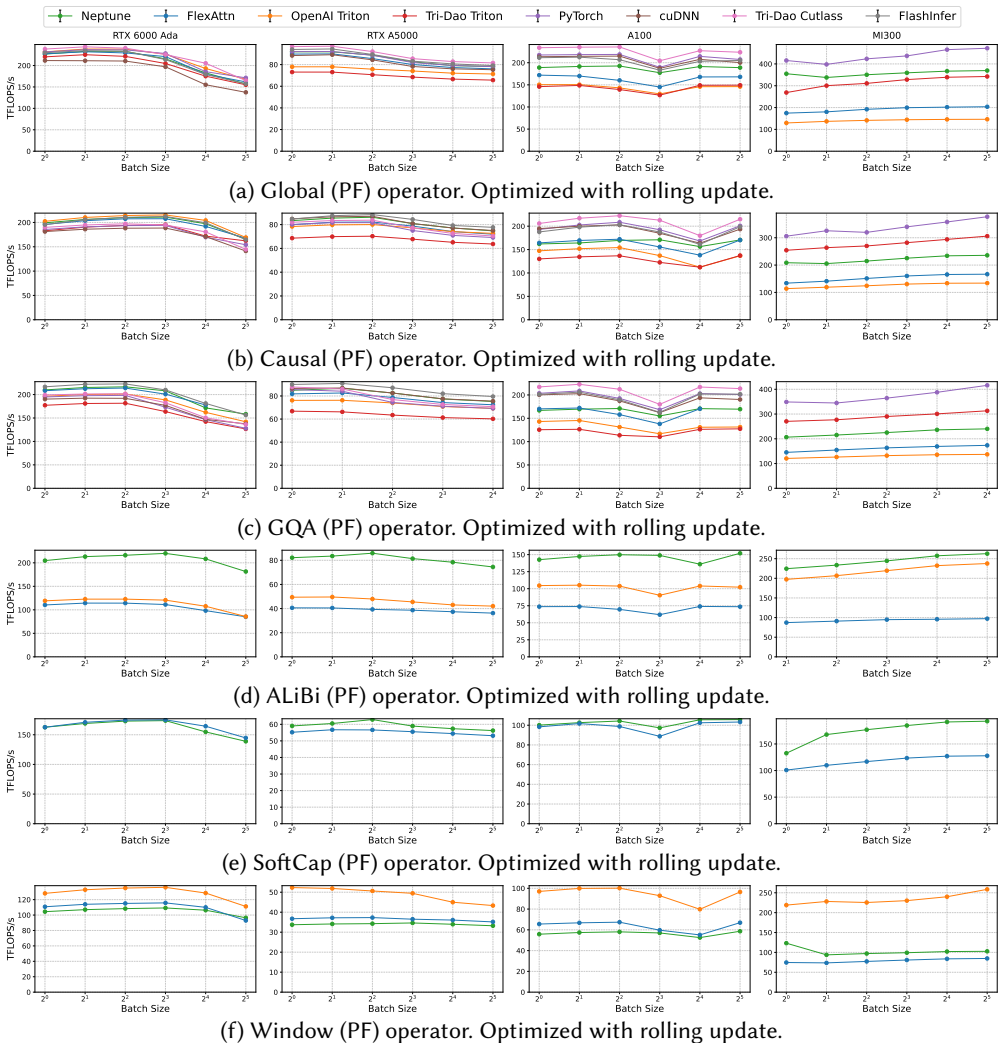


Fig. 18. Throughput of Neptune kernels and multiple baselines over increasing batch size, for all 10 operators on all 4 GPUs. The y-axis shows throughput in TFLOPS/sec, and the x-axis shows batch size.

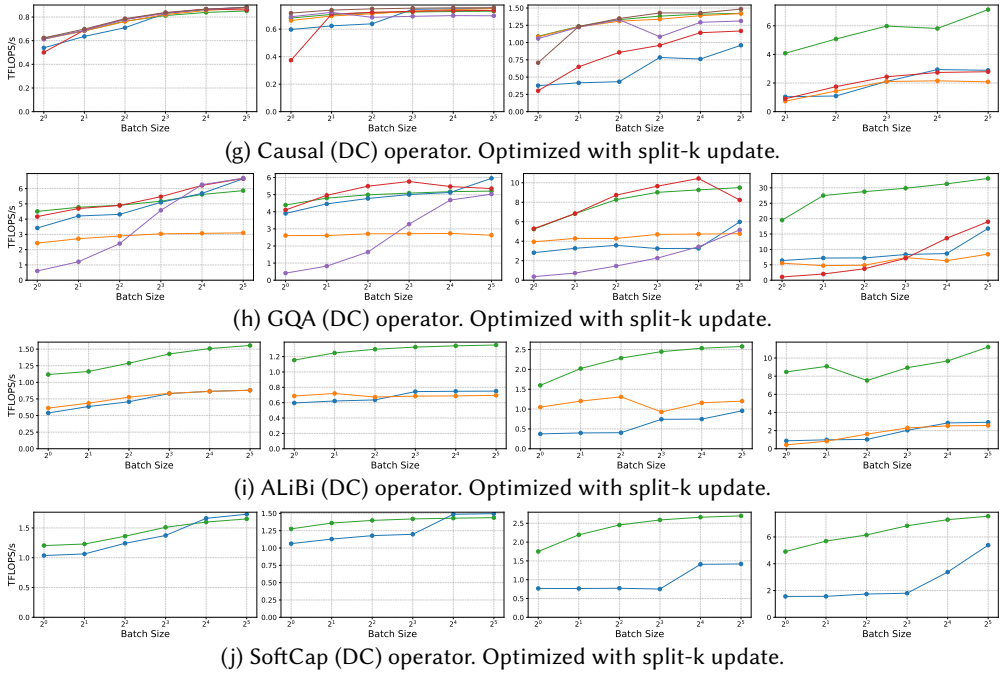


Fig. 18. (Continued) Throughput of Neptune kernels and multiple base- lines over increasing batch size, for all 10 operators on all 4 GPUs. The y-axis shows throughput in TFLOPS/sec, and the x-axis shows batch size.

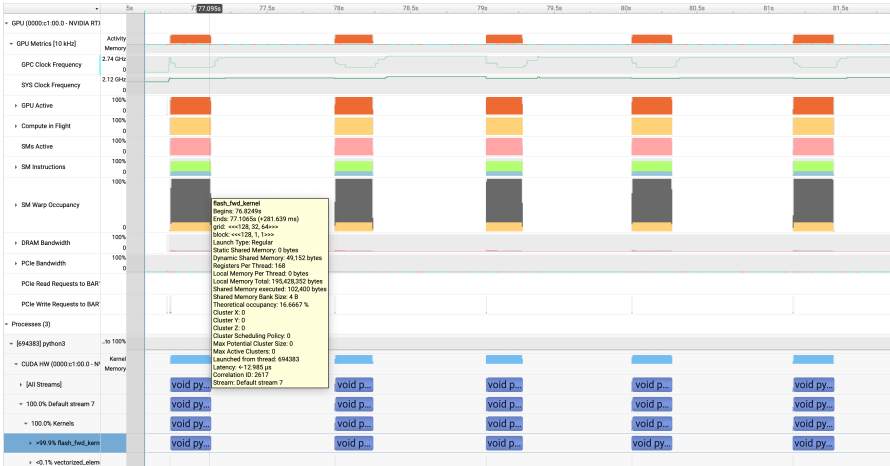


Fig. 19. Timeline view of a profile in the Nvidia Nsight System profiler. The bottom rows show when the kernels are executing, and the “GPC Clock Frequency” row near the top shows the frequency of the GPU graphic clock over time.

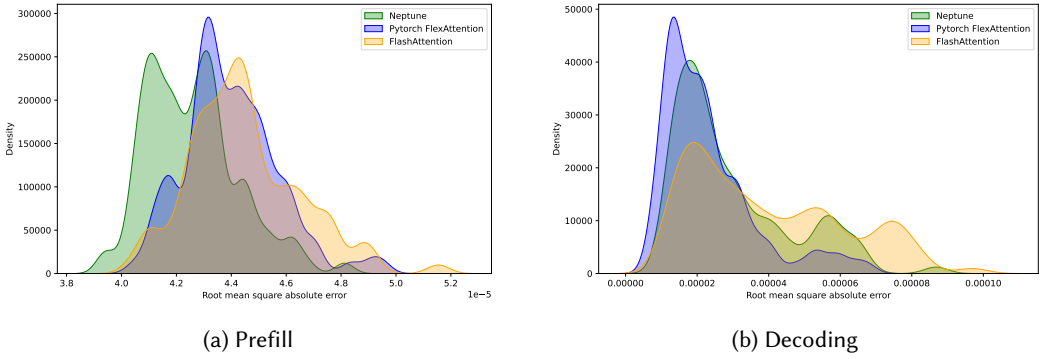


Fig. 20. Per-sample GQA RMSE for (a) prefill and (b) decoding phases.

F NUMERICAL STABILITY OF NEPTUNE KERNELS

Table 5. Numerical Error Summary (Absolute Error)

Variant	Neptune			Tri-Dao Attention			Flex Attention		
	RMS	90th %	99th %	RMS	90th %	99th %	RMS	90th %	99th %
PF Global	4.2e-05	1.5e-05	1.2e-04	4.2e-05	1.5e-05	1.2e-04	4.4e-05	1.5e-05	1.2e-04
PF Causal	4.1e-05	1.9e-05	1.2e-04	4.2e-05	2.5e-05	1.2e-04	4.2e-05	2.3e-05	1.2e-04
PF GQA	4.3e-05	2.3e-05	1.2e-04	4.5e-05	3.1e-05	1.2e-04	4.4e-05	2.7e-05	1.2e-04
PF ALiBi	4.3e-05	1.5e-05	1.2e-04	5.4e-05	3.1e-05	2.4e-04	4.3e-05	1.5e-05	1.2e-04
PF Softcap	2.4e-05	7.6e-06	3.1e-05	–	–	–	2.4e-05	7.6e-06	3.1e-05
PF Windowed	2.4e-05	7.6e-06	3.1e-05	2.5e-05	7.6e-06	6.1e-05	2.4e-05	7.6e-06	3.1e-05
DC Causal	3.9e-05	1.5e-05	1.2e-04	4.6e-05	1.5e-05	1.2e-04	3.0e-05	1.5e-05	1.2e-04
DC GQA	3.4e-05	1.5e-05	1.2e-04	4.5e-05	1.5e-05	1.2e-04	2.6e-05	7.6e-06	1.2e-04
DC ALiBi	4.3e-05	1.5e-05	1.2e-04	4.8e-05	1.5e-05	2.4e-04	4.1e-05	1.5e-05	1.2e-04
DC Softcap	1.4e-05	1.9e-06	3.1e-05	–	–	–	1.3e-05	9.5e-07	3.1e-05

Optimizations that rewrite or re-associate floating-point operations may alter the numerical properties of a kernel. We evaluate the numerical behavior of Neptune, Tri-Dao, and FlexAttention kernels on a variety of attention operators, and find Neptune to provide competitive numerical behavior. Using a set of common inputs, we evaluate kernel outputs against an FP64 unfused attention kernel baseline. Each kernel uses the same schedule and configuration as the performance evaluation. We sample realistic attention inputs by running a pretrained Qwen2.5 model on 100 sentences with sequence length 2048 from the WikiText dataset and recording the input tensors.

Table 5 presents the numerical error of all three frameworks, represented by the root mean square error (RMS) and the magnitudes of the 90th and 99th percentile errors. We also evaluate the distribution of per-sample (per-sentence) RMSE, with results for GQA prefill and decoding show in Figure 20. For all prefill benchmarks, Neptune kernels have lower error than the other two frameworks. For the decode benchmarks, Neptune has lower error than Tri-Dao Attention (when present) and comparable error to FlexAttention.

Received 2025-11-14; accepted 2026-04-03