Modern GPU DSLs: Performance and Portability

Hayden Prairie **Bhrugu Bharathi** **Revant Mahajan**

Cody Wang **Dario Wisznewer** **Rishabh Chittaranjan**

Prince Modi
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
{hprairie, bbharathi, remahajan, cow012, dwisznewer, rchittaranjan, pmodi}@ucsd.edu

1 Introduction

LLM System efficiency is dependent on optimized attention/decoding kernels running on GPU clusters. Optimal kernel implementations, however, are difficult to realize. For one, kernel programmers require in-depth knowledge of the hardware features on GPU architectures they are developing the kernel for to maximize GPU utilization / performance; this design objective requires time spent to not only debug lower-level kernel issues but also hand-tune / design different configurations to potentially serve varying workloads. Additionally, newer GPU architectures are introduced yearly to provide specialized functional units for faster compute and I/O operations in AI workloads. This adds an additional complexity in implementing optimized GPU kernels: to make GPU kernels faster, kernel programmers need to adapt with the hardware market and redevelop previously hand-tuned kernels to take advantage of new features.

To ease kernel development, GPU Domain-specific Languages (DSLs) have been introduced to provide simpler/Pythonic interfaces which expose hardware features to developers via function calls, variables, and DSL-specific optimizations/characteristics (e.g., auto-tuning, compiler optimizations). With these interfaces, kernel developers can quickly develop kernels with adequate to optimal performance that run across different GPU devices easily. However, like GPU architectures, newer GPU DSLs are constantly added to the kernel development ecosystem, making GPU DSL selection another design choice. To understand GPU DSLs' effects on kernel development, this report examines the following questions through a small benchmark that runs RMSNorm and Flash Attention implemented with Helion, Triton, Gluon, and CuTe over the T4, RTX 4060 ti, A100, and RTX 5090 GPUs:

1. What is the the relationship between execution performance and implementation complexity across DSLs?
2. Is there a relationship between DSL performance and varying GPU architectures/devices?

This is the link to GitHub repository [1] which has the kernel implementation and evaluation suite. Branch *main_hayden* runs the evaluation suite while branch *main* holds the helion configurations for RMSNorm. Google drive [2] has the NCU Report and plots for analysis that is presented in this report.

[1] https://github.com/prince-modi/gpu-kernel-dev

[2] https://drive.google.com/drive/folders/1yHqis7ud6Ap05lklFCHH-1lP8TiFJ-UO

2 Background / Related Work

2.1 Performance, Productivity, and Portability

Performance, productivity, and portability can be framed in the following manner [10, 16]:

1. **Performance:** The percentage of utilization of the underlying hardware.

2. **Productivity:** The ease of using the DSL to quickly implement and test GPU kernels.

3. **Portability:** The performance disparity observed when running the same program across different GPU types.

With performance geared towards maximizing the utilization of underlying hardware, a GPU DSL is considered performant if it exposes many low-level GPU features as possible. With access to lower-level units, kernel programmers can apply their expertise to orchestrate functional units, leading to new kernel patterns which complement the architecture. As an example, with CuTe-DSL providing abstractions at a low-level like its CUTLASS C++ analog, Flash Attention-4 [15] orchestrates 5 warps in a multi-staged Flash Attention pipeline (data loading, matrix multiplication, softmax, data storing) to cooperatively compute attention scores on Blackwell-based GPUs, a 2x speedup over Flash Attention-3. This implementation not only breaks away from the conventional, producer-consumer warp-specialization pipelines, but also highlights the benefits of using low-level DSLs to leverage architectural features.

The Flash Attention-4 study additionally shows how aspects of productivity and performance intersect when working with DSLs. From a build/compilation point-of-view, just-in-time compilation-based DSLs enable kernel developers to quickly test and debug logic, enabling faster/easier development overall; specifically, [15] reports a 20-30x compilation speedup with the low-level CuTe DSL over the meta programming-based CUTLASS in Flash Attention-3. From a language-level perspective, though, easy development extends to language simplicity/complexity, which relates ramp-up time for a DSL to kernel performance: as shown in Figure 1, the more exposure a language provides to underlying hardware, the more complex the language is. This complexity eventually reduces developer productivity as programmers require more time to understand the language features as well as manually hand-tune the kernel to best utilize underlying GPUs. Highly productive languages like PyTorch and Helion abstract away lower-level details of the GPU, making logic / tensor manipulation the main concern behind the kernel; conversely, low-level languages like CuTe-DSL provide users complete control over GPU features [15], but this expressivity for performance comes at the cost of time as well as reduced programming clarity from longer programs and complex logic/synchronization. As a result, though faster build times can also come from low-level languages, this lengthy development time supports the general notion that low-level languages result in lower productivity.

While compilers ensure kernels from different DSLs are portable across GPUs, kernel programmers specifically look for performance portable kernels that exhibit high GPU utilization on any GPU architecture with one kernel implementation. The relationship between performance portability and productivity remains only partially characterized. Generally, low-level DSLs tend to be less performance portable than their high-level counterparts, as low-level DSL kernels utilize specific architecture features which may be incompatible with other GPUs; even if these kernels were compatible, GPU utilization rates may further vary (e.g., running Hopper kernel on Blackwell). Likewise, high-level DSLs like Helion and PyTorch respectively rely on auto-tuner- and compiler-guided optimizations to reorganize operations which leverage underlying architectures [9], adapting to different hardware with general programming. At the same time, though, compiler optimizations within relatively high-level DSLs may not be able to provide the best performance on other GPUs, making the productivity-performance portability relationship unclear; Triton, for example, made a fork to Gluon to provide lower-level capabilities which interface with features from newer architecture like Blackwell [14]. Additionally, alternatives methods for performance portability in lower-level DSLs are being developed recently [8], suggesting more developments for this DSL category.

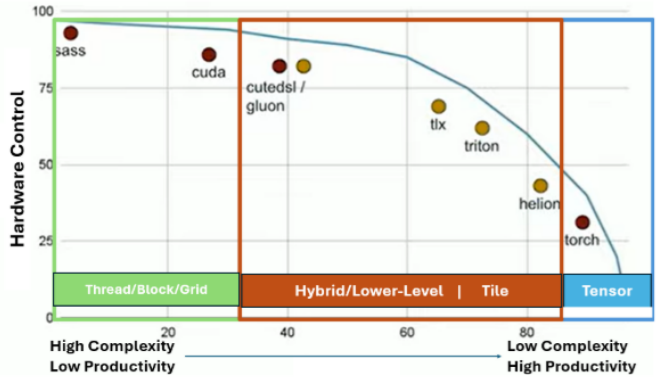

Figure 1: [7]'s productivity versus performance graph shows an inverse relationship between productivity and performance. The chart is further annotated to classify the parallel programming model each language uses.

2.2 GPU DSLs

This section presents a brief background on the various DSLs we utilize for benchmarking. They are presented in the following order: furthest away from the hardware with more abstractions to close to the hardware with more direct control.

2.2.1 Helion

Helion is a Python DSL that provides a tile-centric abstraction to implement kernels, which are then compiled down to auto-tuned Triton code. Developers write kernels using the user-friendly and familiar PyTorch operators, and the Helion compiler converts these down to Triton. While compiling down the Triton, Helion searches through various kernel configurations to find optimal parameters like block sizes, loop orders, and memory access patterns. This frees up the developers to focus on the algorithmic design and delegate hardware level optimization to Helion. Compared to Triton, Gluon, CuTe, developers can implement the same kernel with fewer lines of code and get high performance in general use cases. There are some cons, however: the auto-tuning can result in significant tuning latency per kernel. Additionally, Helion does not expose lower hardware level constructs, so achieving high-level performance might not be possible for some obscure kernel types where the configuration required to implement them is not within the implicit search space the Helion compiler uses.

2.2.2 Triton

Triton provides access to tile-level abstractions to write efficient GPU kernels in Python. This allows developers to achieve low level hardware performance parity with native CUDA kernels through high level Python syntax. By allowing the developers to write code processing tiles (multidimensional arrays) instead of managing individual threads, Triton bypasses a lot of boilerplate complexity inherent in traditional CUDA C++. At a high level, Triton compiler automates a lot of critical operations like memory coalescing, shared memory management, and intra SM scheduling. However, to maintain flexibility and broad applicability, Triton does not automatically schedule inter-SM operations or handle global synchronizations, leaving those to the user's discretion. To write efficient kernels, users are still required to have decent knowledge about parallel programming than standard PyTorch. However the barrier of entry is much lower to achieve high performance with significantly less code. Triton also has built-in auto-tuning that aids developers to achieve this high performance. Unlike Helion, the user is responsible for defining the search space within which the optimal value is found.

2.2.3 Gluon

Gluon is a layout-based Python DSL that is closer to hardware than Triton. The developer has to define the exact layout of the data as well as how it is accessed by threads. It exposes the underlying

compiler primitives that standard Triton is built upon. This is useful for developing high-performance kernels where developers need to bypass high-level abstractions and need access to lower-level primitives. Unlike standard Triton, which defers layout management and synchronization to the compiler, Gluon gives developers explicit control over memory layouts, thread-to-data mapping, and warp-level operations. By directly targeting the Triton-GPU Intermediate Representation (TTGIR) instead of the Triton IR (TTIR), Gluon allows users to implement highly specialized patterns that the standard Triton compiler might not automate optimally. Unlike Triton and Helion, developing on Gluon requires in-depth hardware knowledge and manual tuning. Often, the code developed using Gluon can be hardware-specific and would require more effort to support across hardware generations.

2.2.4 CuTe

The CuTe Pythonic DSL was developed, in part, to circumvent the large compilation times observed when using template heavy CuTe C++ code. CuTe Python achieves its massive compilation speedups by translating a light-weight Python AST into MLIR IR, avoiding the compilation overhead triggered when redundant C++ templates are instantiated. But perhaps more significantly, this Pythonic interface marries readability and maintainability with runtime optimization. In CuTe, `Layout` and `Tensor` types are first class objects, that impose no additional runtime overhead compared to hand-written indexing logic. Specifically, a `Tensor` object wraps an iterator and a `Layout`, which maps from an abstract coordinate space to tangible memory offsets. This flexibility forms the basis of CuTe's ability to use the same slicing, tiling, and partitioning logic at varying levels of hardware abstraction (i.e. global memory, shared memory, or register files). Developers can also specify matrix-multiply-accumulate (MMA) and copy "atoms", which determine how these instructions are assigned to threads and data. These features make CuTe distinct from other DSLs in that it is partition based at both the thread-level and data-level, while eliding manual and error-prone indexing arithmetic. However, this level of specificity comes with a tradeoff: portability. Atoms must be tailored to the hardware being used, and each new generation of hardware introduces new mechanics (e.g. Ampere's asynchronous copies or Hopper's thread-block clusters) that must be leveraged to approach the optimization frontier. Naturally, this imposes some additional cost on developers, who must stay abreast of new hardware capabilities and API changes to restructure kernels correspondingly, in pursuit of optimal performance.

2.3 Kernels

To benchmark different DSL, we implemented both RMSNorm (memory-bound) and FlashAttention (compute-bound). This allowed us to explore how different DSLs behave in memory vs compute bound settings.

2.3.1 RMSNorm

RMSNorm (Root Mean Square Layer Normalization) is a simplified variant of standard Layer Normalization that helps stabilize activations in deep neural networks. Unlike standard normalization, which does both mean centering and scaling, RMSNorm only scales the activations by the root mean square of the input vector. In terms of GPU kernel performance, the challenge of implementing RMSNorm in a DSL lies in the effectiveness of the memory orchestration. That is, how efficiently the compiler can coalesce global memory accesses and manage data movement into SRAM. Evaluating RMSNorm across various DSLs allows us to measure how much boilerplate memory logic is abstracted away versus how much manual control is required to saturate the memory bus.

$$y = \frac{x}{\text{RMS}(x)} \odot \gamma, \quad \text{where} \quad \text{RMS}(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 + \epsilon} \tag{1}$$

2.3.2 Flash Attention

The attention operation, central to modern language modeling, scales quadratically. As context lengths grew rapidly in the years following the release of the first generation of language models, many methods were introduced that partially sacrificed model quality in order to reduce this complexity

(e.g. [13]), by avoiding the computation of the full $N \times N$ matrix. These approaches reflected the prevailing view at the time, that quadratic FLOPs constituted the bottleneck impeding faster attention. However, this paradigm fundamentally shifted when [3] introduced their pioneering Flash Attention kernel. Flash Attention is derived from the realization that *materializing* the full $N \times N$ matrix triggers excessive HBM-SRAM traffic; that waiting on data from bandwidth-bound HBM is the bottleneck. Flash Attention bypasses this by tiling the attention computation, leveraging Q, K, and V block sizes that fit into SRAM. Then, to accommodate the softmax operation, which typically requires access to all elements in some target row, the authors introduce the online softmax, which operates block-wise while still ensuring numerical parity with standard softmax. Below is pseudo-code for the Flash Attention 2 kernel, which differs from its predecessor in that it loops over Q blocks, as opposed to K/V blocks. This change eliminates the inter-warp communication needed to synchronize in-progress online softmax states, and thus allows warps to operate independently.

Algorithm 1 FlashAttention-2 Forward Pass

Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$, block sizes B_r, B_c
Ensure: Output $O \in \mathbb{R}^{N \times d}$

Init $O \leftarrow \mathbf{0}^{N \times d}$, $\ell \leftarrow \mathbf{0}^N$, $m \leftarrow -\infty \cdot \mathbf{1}^N$ ▷ output, normalizer, row-max
Split Q into $T_r = \lceil N/B_r \rceil$ blocks; K, V into $T_c = \lceil N/B_c \rceil$ blocks
for $i = 1$ to T_r **do** ▷ outer loop over Q; independent per warp
    Load Q_i to SRAM
    Init $O_i \leftarrow \mathbf{0}$, $\ell_i \leftarrow \mathbf{0}$, $m_i \leftarrow -\infty$ ▷ local accumulators
    for $j = 1$ to T_c **do** ▷ stream over K, V blocks
        Load K_j, V_j to SRAM
        $S_{ij} \leftarrow Q_i K_j^\top$ ▷ tile of attention scores
        $\tilde{m}_{ij} \leftarrow \text{rowmax}(S_{ij})$ ▷ tile-local max
        $\tilde{P}_{ij} \leftarrow \exp(S_{ij} - \tilde{m}_{ij})$ ▷ stable softmax numerator
        $\tilde{\ell}_{ij} \leftarrow \text{rowsum}(\tilde{P}_{ij})$ ▷ partial normalizer
        $m_i^{\text{new}} \leftarrow \max(m_i, \tilde{m}_{ij})$ ▷ update global row-max
        $\ell_i^{\text{new}} \leftarrow e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij}$ ▷ rescale and accumulate
        $O_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} \Big(\text{diag}(\ell_i)\, e^{m_i - m_i^{\text{new}}} O_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{P}_{ij} V_j \Big)$
        $m_i \leftarrow m_i^{\text{new}}$, $\ell_i \leftarrow \ell_i^{\text{new}}$
    end for
    Write O_i to HBM ▷ one write per Q block
end for
return O

2.4 Motivation

The rapid evolution of specialized AI hardware has introduced complex features like Tensor Cores and asynchronous memory DMA. While these innovations offer massive theoretical throughput for LLMs, harnessing that power traditionally requires writing low-level CUDA. This creates a significant bottleneck: manual optimization is incredibly time-consuming, requires deep architectural knowledge, and is notoriously difficult to debug. Existing DSLs promise to abstract these hardware details, yet there is little empirical data comparing the performance cost of these abstractions against the productivity gains for the developer. Understanding these trade-offs is essential for scaling future LLM systems.

3 Methodology

The following section describes our benchmarking and development process with respect to the shared Github repository. Most development work can be found under the `main_hayden` branch of the repository.

3.1 Kernel Design

To implement the RMSNorm and Flash Attention kernels, separate groups/individuals were responsible for specific DSLs. With the exception of the lowest-level DSL CuTe, to reduce variability among design/programming-style while aiming for generally optimal kernel performance, groups built upon examples from [12, 6, 4] when possible and tested their implementations on Google Colab's T4 and A100 GPUs. Though this suggests our implementations made with complex DSLs like Triton and (especially) Gluon may have less GPU utilization when executing on other GPU devices, the design decision allowed us to have relatively easy (and daily) access to GPUs. Further details on how the kernels were designed are shown in Table 4.

Of the listed kernels in Table 4, Helion's RMSNorm kernel and the CuTe kernels require extra compilation/preparation steps to obtain performant kernel configurations:

- Prior to running benchmarks, the top-level benchmark script run_from_top.sh runs generate_config.py scripts under the git repository's helion_kernels directory to run Helion's autotuner [5] against arbitrarily chosen small, medium, and large test cases (one of each), with the test cases being dependent on the kind of kernel being tested. RMSNorm's small, medium, and large test cases are of dimensions (row(M)-by-column(N)) 256x256, 1024x1024, and 8192x8192 (one 2D matrix and one 1D matrix were generated from these shapes - 1D used the column dimension), while the Flash Attention kernel was attempted to autotune on shapes (batch x heads x seqlen x headdim) 4x32x1024x64, 4x32x4096x64, and 4x32x16384x128 (3 matrices were generated from these shapes). Flash Attention autotuning was too expensive, causing the generate_config.py for Flash Attention to be manually commented out in local repositories during testing. The generated configurations are later profiled against one another for each test case encountered while running bench_driver.py, with the best configuration being used for benchmarking.

- CuTe DSL's just-in-time compilation is very fast, enabling its kernels to be recompiled for each test case it encounters; we opt for this testing choice to get competitive results.

The profiling/compilation steps are run prior to calling Triton's testing functions, ensuring compilation/profiling times do not influence results.

3.2 Benchmark Design and Environments

Benchmarks were executed using the run_from_top.sh script to test the kernels from Table 4 over the GPUs listed in Table 1. During each run, the script gathers one set of Nvidia nsys and ncu profiles using fixed test cases for each kernel as well as executes Triton-based benchmarks with varying-sized test cases for the default *triton.testing.do_bench* configuration of 25 warm-up iterations and 100 testing iterations. The Triton benchmarks generate memory and compute throughput graphs for the memory-bound operation RMSNorm and compute-bound operation of forward Flash Attention kernels respectively; each graph plots the throughput of each kernel, enabling comparisons and visualizations of DSL performance with known performance metrics. We ran our compiled/original kernels with the benchmark test cases listed in Table 2. Test cases were determined from examples [12, 4].

Table 1 outlines the GPUs used in our experiments; we vary architecture and product-type(consumer vs enterprise) to better understand the relationship between DSL- and Device/Hardware-type. The T4 and A100 GPUs are provided through Google Colab, while the consumer-grade GPUs RTX 5090 and RTX 4060 Ti are from our team members. Since each GPU (apart from those from Colab) has their own configurations, drivers, etc. some kernels were unable to run:

- For RMSNorm, CuTe could not run on T4, RTX 4060 Ti, and RTX 5090

- For Flash Attention, PyTorch's SDPA could not run on T4

Name	Architecture	Product Class
Tesla T4	Turing	Enterprise
RTX 4060 Ti	Ada	Consumer
A100 SXM4 40GB	Ampere	Enterprise
RTX 5090	Blackwell	Consumer

Table 1: GPUs included in testing. All GPUs are by NVIDIA.

Kernel	**Benchname**	**Cases**
RMSNorm	RMSNorm-With-Varying-N	rows $= 4096$; cols $= 256 \times 2^N$ with $N \in [2, 100)$
RMSNorm	RMSNorm-With-Varying-M	rows $= 256 \times 2^N$ with $N \in [2, 100)$; cols $= 4096$
FlashAttn	fused-attention-batch4-head32-d64	batch $= 4$, $N_{\text{Heads}} = 32$, head_dim $= 64$, $N_{\text{CTX}} = 2^N$ with $N \in [10, 15)$

Table 2: Benchmark configurations for RMSNorm and FlashAttention kernels.

4 Experiments

4.1 Case Study: Implementation Complexity versus Performance on A100

Original implementations of both kernels were written for Torch, Helion, and Triton. Given the lack of granularity in Torch, the Flash Attention backend for `scaled_dot_product_attention` was used as the Flash Attention-2 kernel. With Gluon and CuTe, original implementations of RMSNorm were written, and the original Flash Attention-2 implementation in CuTe [2] was used. As we were unable to write or find a Flash Attention-2 kernel for Gluon, its results are omitted.

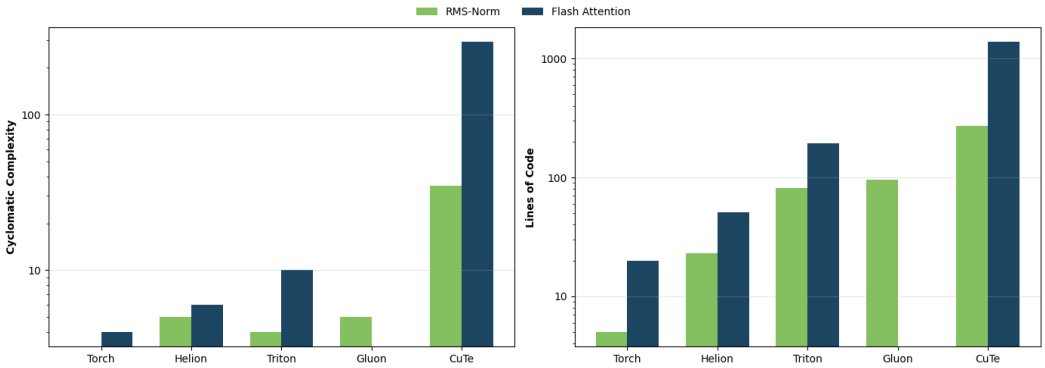

Figure 2: Implementation complexity for RMSNorm and Flash Attention-2 over the five surveyed DSLs. The left plot shows cyclomatic complexity, whereas the right plot shows lines of code. Note that the Torch implementation of RMSNorm has a cyclomatic complexity of zero. Data were collected by running scc [1] on the file containing each kernel implementation.

Figure 2 shows the implementation complexity of each kernel. Since code complexity can be highly subjective, we use cyclomatic complexity and lines of code as approximations. The plots of both metrics suggest that implementation complexity increases exponentially as abstraction decreases.

Figures 3 and 4 show the benchmark results on the A100. Overall, we see positive correlation between implementation complexity and performance, aligning with sentiments previously expressed by [7, 16]: the low-level DSL CuTe, which requires more specialization and hand-tuning effort, performs the best for memory- and compute-bound kernels as it provides finer-grained control over hardware features. However, there are are some exceptions to this trend. From the RMSNorm side, CuTe's memory throughput occasionally dips for row counts roughly within the range of 5000 to 12500, suggesting that more robust memory loading would be needed to reduce performance degradations from handling irregular or unaccounted-for shapes. Additionally, Helion's throughput exceeds that of Triton and Gluon for RMSNorm. In the plot for Flash Attention-2, CuTe's TFLOPS

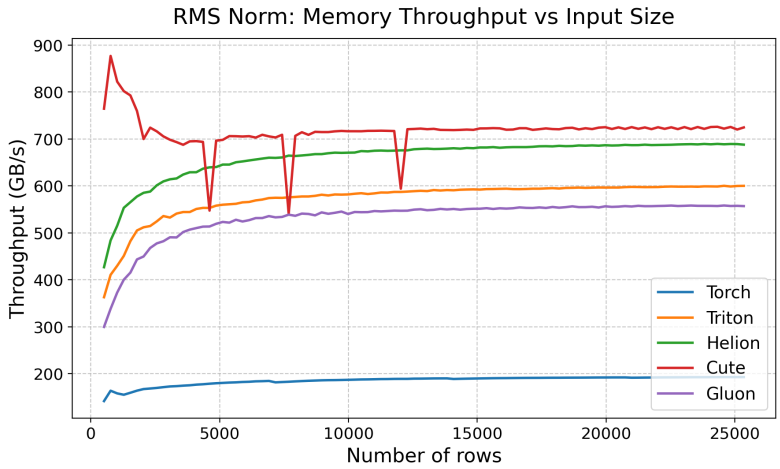

Figure 3: Throughput over input size (number of rows) for RMSNorm on the A100

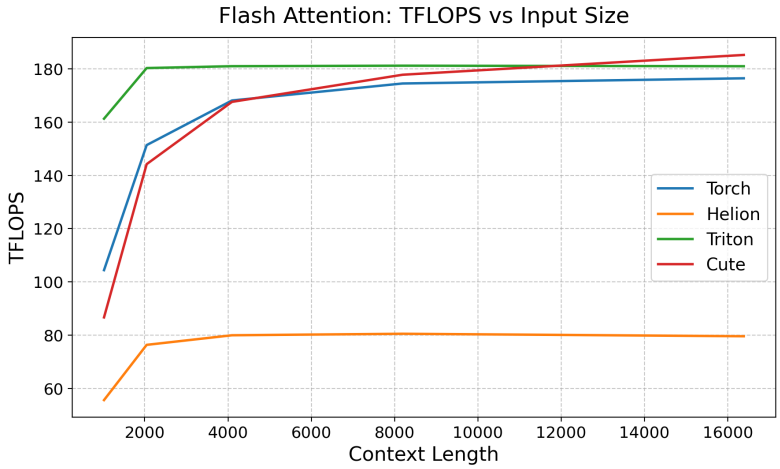

Figure 4: TFLOPS over context length for Flash Attention on the A100

only surpasses Triton for larger context lengths, and is lower than Torch for the smallest input sizes, suggesting that the extra development effort required to functionalize a CuTe kernel over a Triton one may not be mandatory for most “reasonable" problem sizes.

Relative to Triton and Gluon’s performances, Helion’s superior memory bandwidth is primarily due to its optimized management of the High Bandwidth Memory bottleneck. Figure 5 outlines the key difference between the Helion and Triton kernels in memory access. Specifically, Helion loads the input row into SRAM once for both summation and normalization, whereas the Triton and Gluon kernels require two full passes over the input data, effectively halving their theoretical bandwidth utilization. Unlike Triton and Gluon, Helion loads data tile-by-tile, 1 row for larger configurations and 8 rows for the smallest (256x256) configuration while appearing to load all the columns. Triton/Gluon makes up for low memory utilization per tile by spawning as many programs as there are workloads(e.g., Triton has one program per row for higher utilization).

In addition, Helion’s configuration files shows unique memory management/indexing patterns and parallelism details(num_stages, num_warps) which go beyond Triton/Gluon’s configurations. For example, Triton uses default load configuration without making modifications to eviction policy while Helion has auto-tuned these parameters. These subtle implementation differences illustrate how high-level DSLs are able to tune various parameters that are typically unapproachable or unknown for amateur kernel developers. However, the auto-tuning process can be prohibitively expensive for larger input sizes and more complex kernels. This suggests a hybrid approach is optimal: high-level

```
for off in tl.range(0, input_row_stride, BLOCK_SIZE):
   ...
   x = tl.load(X_row + cols, mask=mask, other=0.0).to(tl.float32)
   sum_sq += x * x
mean_sq = (tl.sum(sum_sq) / input_row_stride).to(tl.float32)
normfactor = 1.0 / tl.sqrt(mean_sq + eps)
for off in tl.range(0,input_row_stride,BLOCK_SIZE,num_stages = num_stages):
   ...
   x_block = tl.load(X_row + cols, mask = mask, other = 0.0).to(tl.float32)
   w_block = tl.load(w_ptr + cols, mask=mask,other = 0.0).to(tl.float32)
   tl.store(Y_row + cols, (x_block * w_block * normfactor).to(tl.float16), mask=mask)
```

```
for col_index_start in range(0, n_cols, BLOCK_SIZE):
    ...
    inp = gl.load(inp_ptrs, mask=mask, other=0.0)
    inp_sq_sum_per_block += inp * inp

rms = gl.sqrt(gl.sum(inp_sq_sum_per_block, axis=0) / n_cols + epsilon)

output_start_ptr = out_ptr + row_id * output_stride
for col_index_start in range(0, n_cols, BLOCK_SIZE):
    ...
    inp = gl.load(input_start_ptr + offsets, mask=mask, other=0.0)
    rms_norm = inp / rms * gamma

    gl.store(output_ptrs, rms_norm, mask=mask)
```

```
load = tl.load(x + (indices_0[:, None] * x_stride_0 + ...), mask_0[:, None] & mask_1[None, :], other=0)
v_0 = tl.cast(load, tl.float32)

v_1 = v_0 * v_0
mean_extra = tl.cast(tl.sum(v_1, 1), tl.float32)

v_5 = libdevice.rsqrt(v_4)

subscript = v_5[:, None]
v_6 = v_0 * subscript

tl.store(out + (...), v_10, mask_1[None, :])
```

Figure 5: RMSNorm memory access patterns in **Triton (top)**, **Gluon (middle)**, and **Helion-generated Triton (bottom)**. The Triton and Gluon implementations call `tl.load`/`gl.load` for variance calculation and normalization separately, resulting in redundant HBM access. The Helion-generated Triton implementation loads the input tensor into SRAM once and reuses the cached data, significantly improving overall memory throughput.

DSLs can be used to rapidly prototype in testing environments, then compiled to Triton code and further hand-tuned by expert kernel developers.

4.2 Question 2: DSL Performance Portability

4.2.1 Throughput Differences

As per Figure 6, memory throughput improves as architecture becomes newer based on the increasing y-axes scales from the left to right graphs in both RMSNorm, highlighting how a large factor in performance is the kind of GPU Device executing the kernel. Focusing on the performance differences between the T4 and A100, RMSNorm's memory throughput on the A100 is roughly 4+ times more than that of T4's: Pytorch, the slowest DSL, experienced a memory throughput increase from 40 GB/s on the T4 to 200 GB/s on the A100. Gluon, Triton, and Helion's RMSNorm implementations further display more pronounced memory throughput increases on the A100 when the number of loaded rows goes from approximately 500 to 5000; Triton and Gluon's throughput improvements across GPU devices further asserts the main notion that GPU device type plays a larger role in performance compared to programmer-developed optimizations / hand-tuning, which serve to maximize GPU utilization rather than be a main performance-driver.

Additionally, Figure 6 displays how Product Class noted in Table 1 is another performance factor for memory-bound operations: the RTX 4060 Ti and RTX 5090, though having relatively higher memory throughput ranges compared to T4 and A100 respectively, have clustered memory throughput due to limited shared memory. Compute-bound operations like the forward pass of Flash Attention

(bottom row graphs) share relatively similar curve shapes, indicating memory usage is potentially not a limiting factor for performance. Note that similar to the A100 analysis, Helion performs poorly for other device types due to the lack of auto-tuning (either manually disabled or with one configuration), with PyTorch's SDPA implementation outperforming it most likely due to utilizing optimized lower-level kernels.

Further analysis on Helion's performance portability to find hardware optimizations through its auto-tuner in the context of RMSNorm can be found in the Appendix B. Sections 4.3 and 4.4 examine the performance portability of all DSLs across GPU devices through NCU Reports gathered from the benchmarking platform.

4.3 Case Study: NCU Performance Deep Dive Comparing Triton and Gluon RMSNorm Kernels on A100 and T4

The NCU Reports on the T4 were gathered while Helion's auto-tuning picked from the 256x256 case (where max registers are set at 128) to expedite report gathering. As mentioned previously, auto-tuning is expensive. The A100's RMS plots were gathered when auto-tuning among three set configurations. We choose to ignore Helion's NCU plots due to this mismatch in configuration auto-tuning. However, one important point to note is that Helion's memory throughput on the A100 remains at/above 90% throughout the sweep of row counts during profiling: this aligns with our observations from Figure 6 in which Helion's performance was higher than Triton and Gluon's, and we expect a similar plot when running A100's benchmark configuration on T4 due to Helion's auto-tuner. It is interesting to further see that CuTe's proportion of memory throughput is slightly lower than that of Helion's, yet its overall memory throughput (GB/s) remains the highest; this suggests other factors contribute to this throughput (e.g., CuTe had the highest register allocation compared to other DSLs, but compute throughput was relatively low and its DRAM throughput remains the same as the overall memory throughput).

Apart from individual analyses of DSLs on A100, we direct our attention to Triton and Gluon, which do not require auto-tuning for its performance to observe their respective RMS kernels' performance portability. Specifically, Triton and Gluon's memory throughput percentages. T4's Triton, looking similar to Helion's A100 curve, remains near 90%, but its throughput drops below 85% across all cases. Likewise, T4's Gluon, which is below 87.5%, drops below 70% on the A100. As a result, even though Figure 6 indicates the GB/s of Triton and Gluon increase overall, Figure 7 shows RMS Kernels using Triton and Gluon are not performance portable since they do not maintain a similar memory throughput percentages across the T4 and A100. This suggests the Triton and Gluon kernels cannot remain general. Instead, specialization, to improve memory throughput using GPU-specific abstractions/operations, must happen for the kernels on A100.

4.4 Case Study: NCU Performance Deep Dive Comparing Triton and Helion Flash Attention Kernels on A100 and T4

Here we present another case study of performance portability, centered around our Triton and Helion Flash Attention Kernels and spanning Colab's A100 and T4 GPUs. In particular, we used Nvidia's Nsight Compute tool to profile these kernels across the following context sizes, `CTX_Len` $\in [512, 1024, 2048, 4096, 6144, 8192, 12288, 16384, 20480, 24576]$, while using a head dimension of 64. We note that using a head dimension of 128 on T4 repeatedly triggered Out-of-Memory errors, due to a limited shared memory pool on the T4. We illustrate our findings of the Compute SM Throughput (%) and Memory Throughput (%) in Figures 8 and 9.

Before examining these profiling results, we expected to observe a trade-off between portability and performance: namely that, DSLs that granted their developers more granular access to hardware features would likely be less portable. In this case, the profiling results align with this hypothesis.

In Figure 8, we observe that Triton maintains an edge over Helion on the A100, across the majority of problem sizes, whereas its compute throughput peak drops on the T4 to less than 16%. This drop is directly downstream of Triton's tile-based programming model, which requires developers to select configurations based on where they will be applied. Here, the tile configurations selected to saturate A100 tensor cores, struggle to saturate less powerful T4 tensor cores, inadvertently making Helion the more performant DSL in that regime.

This hypothesis is substantiated in Figure 9, for a subtle reason. While maximum utilization across the memory hierarchy appears to trend similarly between devices, Triton's low compute throughput along with its relatively high memory throughput strongly suggests that Triton on T4 is memory bound. In other words, the T4 SM's are idle and waiting on data from HBM. This is sensible, because the tile size chosen to maximize arithmetic intensity on A100 tensor cores, is simply not amenable on the T4, which has less shared memory and registers and weaker tensor cores.

In contrast to Triton, Helion is moderately performant in both device regimes, because the choice of tile configuration is directly handled by the compiler. But, this does not imply that less granularity always yields good performance across devices; on A100, for example, the manually configured Triton kernel is better.

In general, these results support that because good performance is contingent on deep knowledge of one's target workload, DSLs that enable precisely tailored kernels are far more likely to yield good performance, compared to DSLs that produce generically tailored kernels, which may only be incidentally performant.

[Spike is due to Helion's selection of a sub-optimal config at that problem size <5000. And this is because minimal auto-tuning was applied to Helion, due to it took too long.]

5 Conclusion

In this work, we investigated the effect of GPU DSL selection on kernel development. With the large selection of DSLs available, understanding the tradeoffs between different DSLs can be challenging. To bridge this gap, we analyzed the differences in code complexity and performance of the kernels RMS Norm and Flash Attention-2 for various DSLs and devices. Our findings show that high-level DSLs allow for simpler and more accessible kernel development as well as better implementation portability, at the cost of lower performance on certain problem sizes. In contrast, Low-level DSLs achieve the best performance but require more developer time and knowledge to use. By quantifying the overhead of high-level abstractions against the productivity boost they enable, this study provides a framework for researchers to navigate the complex ecosystem of GPU programming models more effectively.

5.1 Limitations and Future Work

A major limitation of this work is that while we tried to formalize implementations across DSLs to the best of our ability, there are still minor kernel differences, which indirectly affect the results and case studies. Better formalizations of kernels across differing DSLs (e.g., between Triton and CuTile) could directly improve the achieved performance. Furthermore, to fully test out these major hypotheses and core questions would require extensive optimizations for each DSL on each piece of hardware, which was infeasible for the current scope of the project. We believe that both of these issues are viable avenues for future work. Furthermore, a deeper exploration of how hardware complexity intertwines with the hardware control and productivity trade-off would be an interesting future direction. Lastly, this analysis was restricted to NVIDIA-based GPUs, leaving a natural follow-up exploration of performance and portability across AMD GPUs, TPUs, and other forms of machine learning hardware.

6 AI Usage Acknowledgment

This project made limited use of large language models (LLMs) as assistive tools. Specifically, LLMs were used for: generating some figures and visualizations for slides and plots; proofreading and polishing original written content for clarity; and formatting LaTex layouts (e.g, image placement). All core ideas, analysis, and writing reflect the original work of the authors.

References

[1] Boyter, B. scc: v3.7.0, March 2026. URL `https://github.com/boyter/scc/releases/tag/v3.7.0`.

[2] Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

[3] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

[4] Dao-AILab. flash-attention, 2026. URL https://github.com/Dao-AILab/flash-attention/blob/main/flash_attn.

[5] Helion. Autotuning, 2026. URL https://helionlang.com/deployment_autotuning.html.

[6] Helion. Examples, 2026. URL https://helionlang.com/examples/index.html.

[7] MODE, G. Lecture 77: Domain specific languages for gpu kernels. https://www.youtube.com/watch?v=5qSN-R_E3w0, 2025.

[8] PyTorch. The future is tiled: Using cutile & tileir to write portable, high-performance gpu...- jared roesch. https://www.youtube.com/watch?v=UEdGJGz8Eyg, 2025.

[9] PyTorch Team at Meta. Helion: A high-level dsl for performant and portable ml kernels. https://pytorch.org/blog/helion/, October 2025. Accessed: 2026-03-17.

[10] Reguly, I. Z. and Mudalige, G. R. Productivity, performance, and portability for computational fluid dynamics applications. *Computers & Fluids*, 199:104425, 2020.

[11] Triton. flash-attention, 2026. URL https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html.

[12] Triton. Tutorials, 2026. URL https://triton-lang.org/main/getting-started/tutorials/.

[13] Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity, 2020. URL https://arxiv.org/abs/2006.04768.

[14] Yoshimi, B. Triton community meetup 20250709 130219 meeting recording. https://www.youtube.com/watch?v=5e1YKqsP8i8&t=1039s, 2025.

[15] Zadouri, T., Hoehnerbach, M., Shah, J., Liu, T., Thakkar, V., and Dao, T. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026. URL https://arxiv.org/abs/2603.05451.

[16] Zhang, L. Gluon: Explicit performance. https://www.lei.chat/posts/gluon-explicit-performance/, 2026.

A Kernel Implementation Details

Kernel (Filename)	Details
Triton-RMSNorm (`another_rmsnorm_with_loops.py`)	Hybridizes Triton's fused softmax and layer normalization examples to design a persistent kernel which calculates RMSNorm row-by-row using a fixed `BLOCK_SIZE` of 2048. The `BLOCK_SIZE` parameter can be tuned for better register utilization in the future. For simplicity as well as better GPU utilization, we opt to launch as many programs as there are rows to process.
Gluon RMS Norm	Similar to what we have for Triton's RMSNorm implementation. The block size used is 4096. The kernel is written to support cases where each input vector is more than the block size and for the cases where the number of input vectors is more than the number of programs.
CuTe RMSNorm - A100	This kernel is organized so that warps (or groups of warps) independently process one token row of length `hidden_dim`: each thread owns a contiguous and vectorized slice of this row, loads it as part of a coalesced transaction, contributes its local partial sum, and participates in a row reduction to calculate the final value. For most problem sizes, row reductions occur within a single warp, but for larger rows, multiple warps collaborate via a reduction buffer. Finally, latency overlap is achieved by staging data movement from GMEM to SMEM, with asynchronous copy atoms, before moving to registers.

Table 3: Summary of RMSNorm kernel implementations and optimization details.

Kernel (Filename)	Details
Helion-FlashAttn (`flash_attn_helion.py`)	Inspired by Helion's `attention.py` and DaoAI-Lab's `flash_attn_og.py` to implement a high-level kernel with softmax/numerical stability enhancements. While the Helion example suggests to set the auto-tuner's `static_shapes=True` to speed up auto-tuning, this causes the kernel to recompile each time it encounters a new shape (`hl.specialize` specifies a compile-time constant) during benchmarking. We initially opted to compile with `static_shapes=False` to obtain configurations which are robust against different shapes, but the auto-tuning effort was very large; eventually, we set `auto-tune_effort=None` in local repositories to expedite testing.
CuTe Flash Attention	This kernel is a 1:1 implementation of the Dao-AI-Lab SM80 kernel. It is organized so that warps cooperate on fixed-size attention blocks mapped directly to SM80 tensor-core fragments: each thread owns a structured fragment of the score accumulator and output accumulator, loads Q, K, and V through CuTe tiled-copy layouts as coalesced vector transactions, and participates in warp-level MMA, plus reductions and barriers to maintain the corresponding per-row softmax state. For most of the execution, performance comes from asynchronous copy atoms that stream Q, K, and V from GMEM to SMEM in staged groups and warp MMA instructions which consume SMEM tiles. Both mechanisms overlap compute with upcoming loads. In sum, CuTe is leveraged as a direct expression of Ampere scheduling primitives rather than as a high-level abstraction that hides them.
Triton Flash Attention	This kernel is based on the official Triton documentation's Flash Attention tutorial to implement a fused attention kernel with online softmax [11]. The kernel is split into an outer forward pass (attn_fwd) which handles per-batch and per-head offsets and launches one program per query block, and an inner loop (_attn_fwd_inner) which iterates over key-value blocks accumulating a running softmax output. A STAGE parameter controls causal vs. non-causal masking.

Table 4: Summary of FlashAttention kernel implementations and optimization details.

B Additional Analysis

B.1 RMSNorm: Triton Code Generation Differences for Helion

The generated Helion codes from A100 and T4 rely on similar logic in RMSNorm, in which only one tile load derives the sum of square values and normalized values. One difference between the two includes the block sizes being loaded for each configuration; though configuration sizes are the same, T4 bounds the number of rows per tile to 4 for the 256x256 and 1024x1024 cases, while A100's sets the number of rows per tile to 8 and 1 for the 256x256 and 1024x1024 cases. The 256x256 case can be reasoned out as differences in shared memory sizes, with the A100 having 164 KB per SM. The 1024x1024 case is slightly unique - further analysis must be done to reason out how the A100 and T4 weighed its system parameters. We can additionally see how Helion can tune program launching through the pid_type knob: T4's 256x256 case adheres to the *persistent_blocked* program policy which explicitly manages/ensures all SMs are fully utilized while remaining persistent(same program used throughout the execution lifetime); majority of the configurations though rely on the flat program configuration, which spawns as many programs as necessary - similar to the Triton/Gluon configurations. This explains why T4's 256x256 case includes another loop within its function body to virtualize program id's: one program can take up the virtual program id's and execute a tile belonging to that virtual program.

Apart from these differences, Helion's auto-tuner does not find any specialized hardware features to leverage at the Triton code level when running on T4 and A100; while this suggests further testing Helion on newer architectures to validate this claim, examples of unique hardware instructions/features may be found in assembly/ lower-level IRs. **Note:** The discussed configurations and generated code can be found under the *main* branch of the GitHub repository; the discussed T4 and A100 configurations can be found in rms-t4.tar and rms-a100.

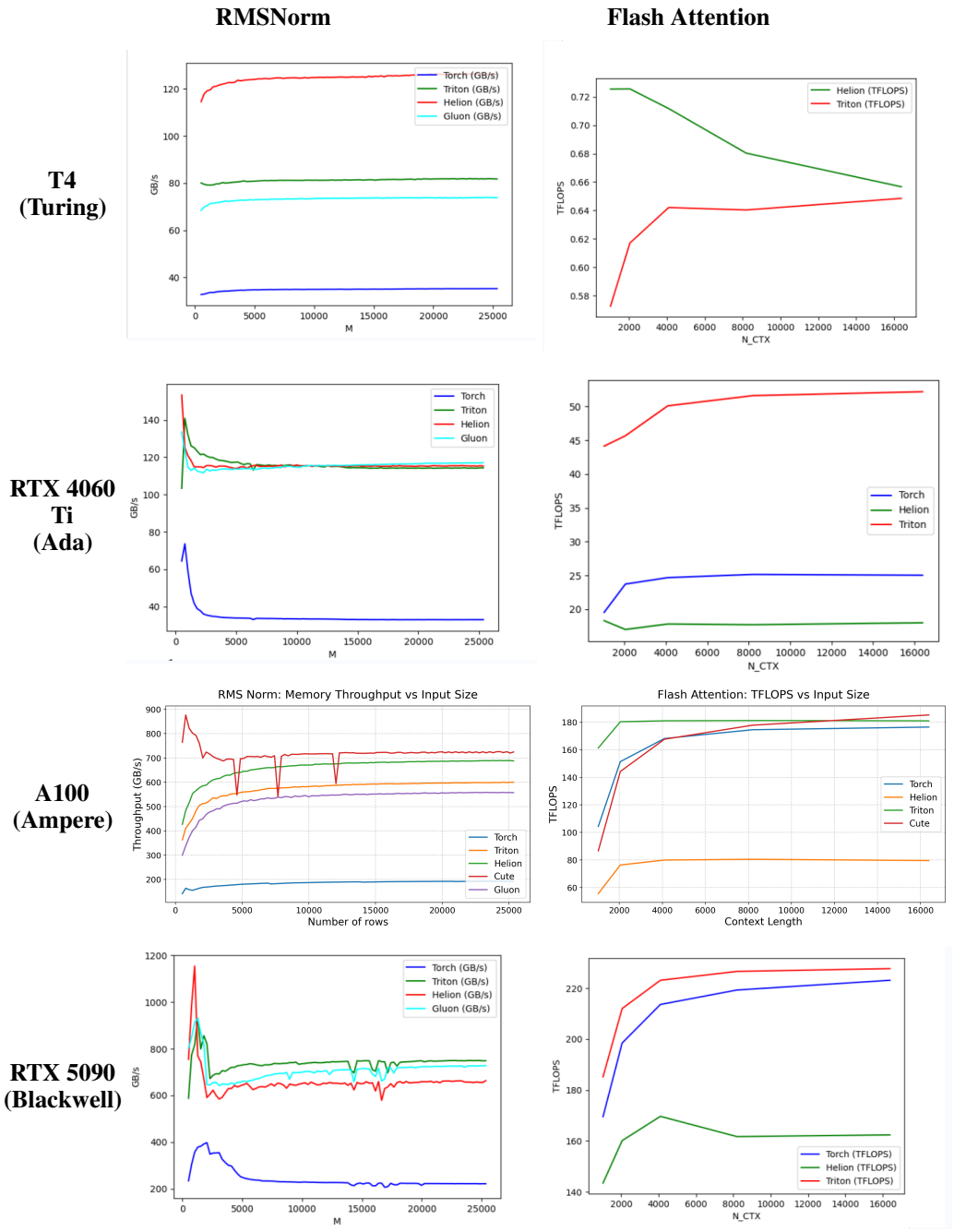

Figure 6: The memory and compute throughputs of RMSNorm and Flash Attention respectively across the A100 and other GPU devices; the RMSNorm graphs were generated with a varying number of rows and a fixed number of columns (4096) while the Flash Attention graphs were generated by varying the sequence length. The graphs are arranged by architecture recency with Turing being the oldest architecture and Blackwell being the newest architecture.

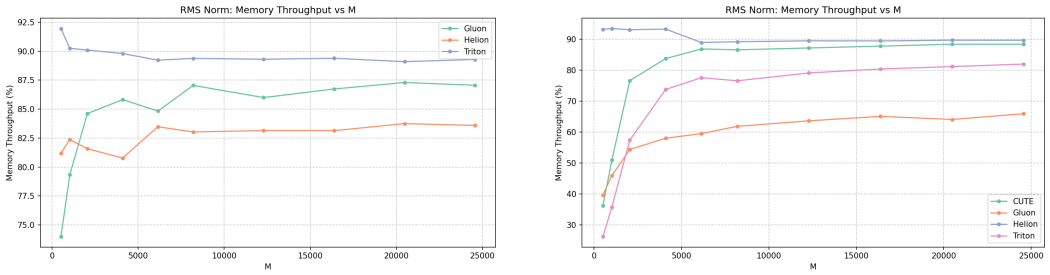

Figure 7: Memory throughput percentage as the number of rows increases as factors of 256 [512,24576] and the number of columns are fixed at 4096. The left figure is from the T4, while the right figure is from A100.

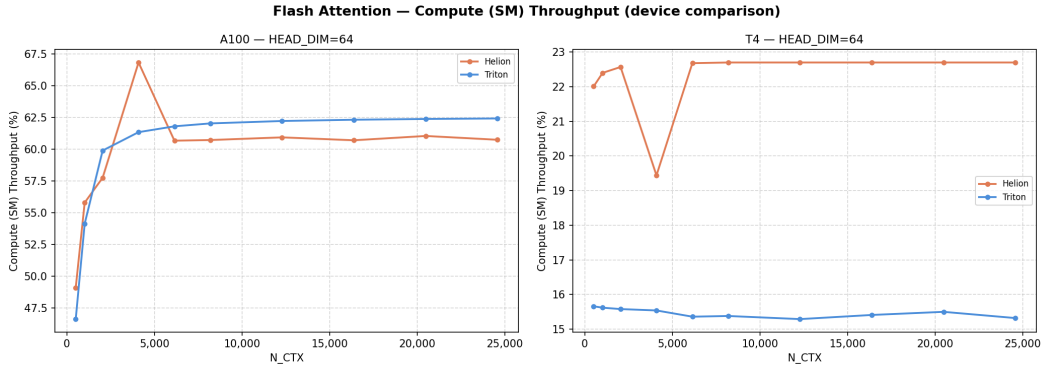

Figure 8: Compute Throughput comparison, as a percentage of the SM, between A100 (left) and T4 (right) for Helion and Triton. While Helion matches Triton's throughput on A100, it exceeds it on T4. Thus, Helion's compiler-selected config is robust across devices, unlike Triton's manually engineered config.

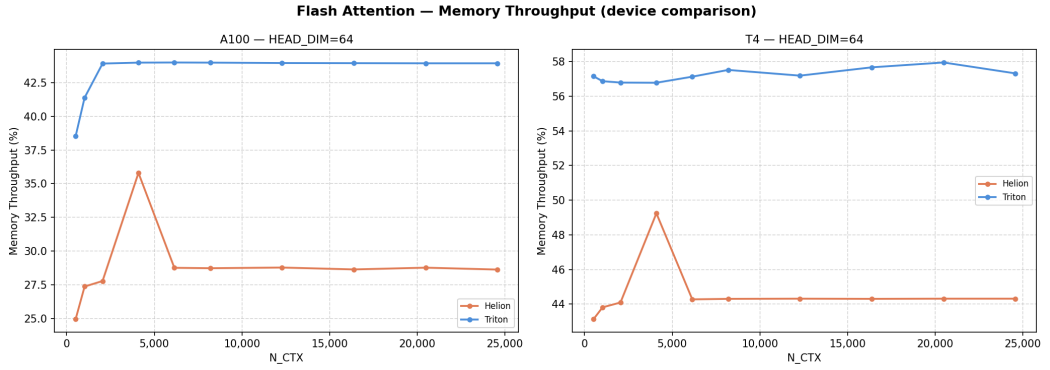

Figure 9: Memory Throughput comparison, as a percentage of the SM, between A100 (left) and T4 (right) for Helion and Triton. Both Triton and Helion maintain the same general trend across devices, with Triton achieving a consistently higher ceiling. In conjunction with Figure 8, Triton appears to be compute-bound on T4.