

Portable Paged Attention: From Triton to Helion

Burkhard Ringlein
IBM Research
AI Native Systems

2025-10-21, PyTorch Conference North America, San Jose



PyTorch Mission: **Any model** on **any chip**....



...not to re-write models for new/different chips could help!

→ **performance portability** is key to **advance open source AI**

→ Helion can help! 

Agenda

- 1) Quick Introduction to Helion
- 2) Paged Attention in Helion: How to write, optimize,
...and the difference of Triton vs. Helion for kernel authors
- 3) Huge search spaces of Helions autotuner and how to navigate them
- 4) Performance results: Microbenchmarks & end-to-end vllm bench serve
- 5) Conclusion?

→ **Goal:** 1) Share the state of performance portability for critical kernels using Helion
...including highlight proven practices and sketch open questions

2) **get you excited about Helion!**



What is Helion?

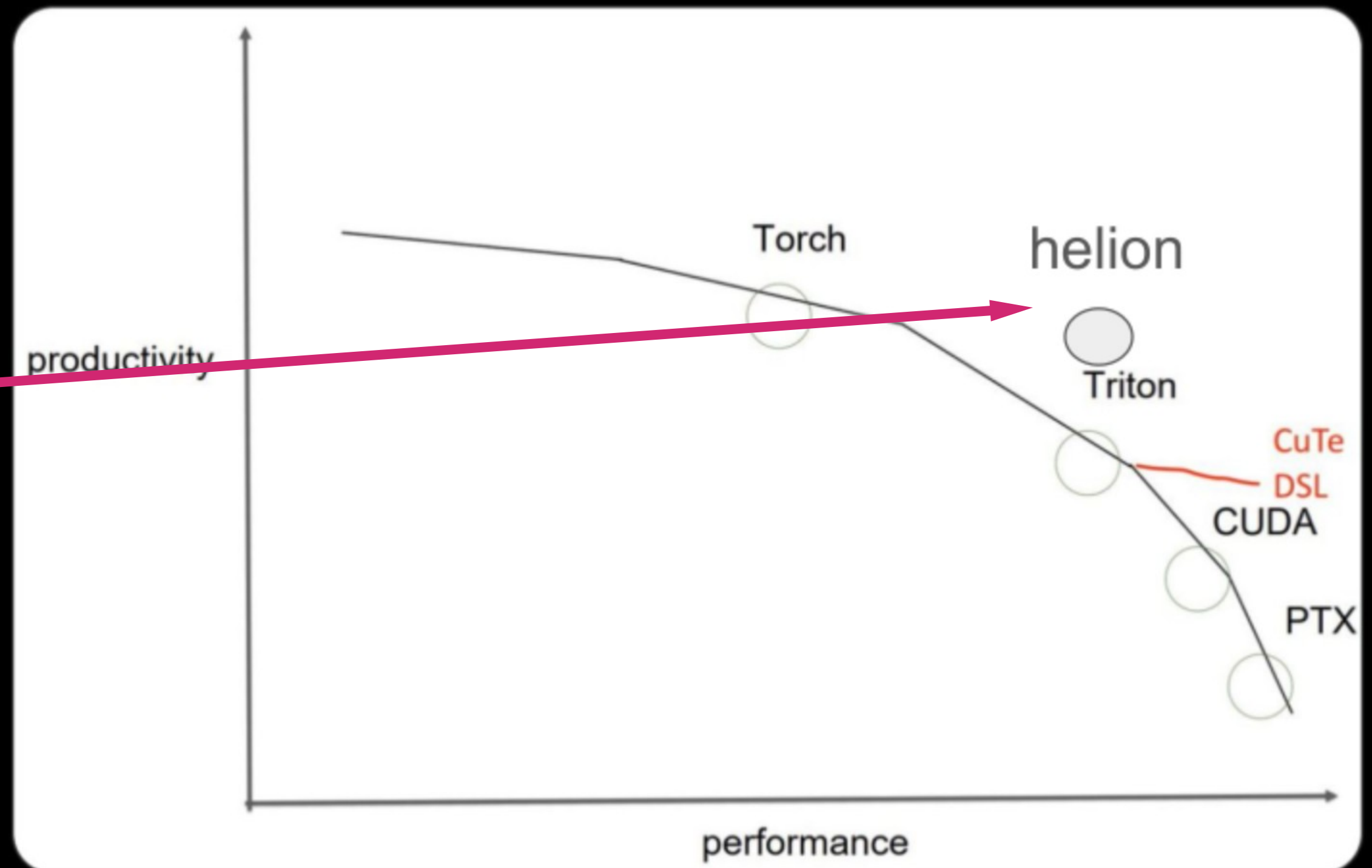
- PyTorch Foundation Project
- New kernel DSL based on Tiling
 - Lowers to Triton, Pallas, CuTe, Metal
- Comes with a powerful autotuner
- Aim: higher productivity and performance
- More control than torch.compile
- Better performance portability



Tri Dao ✓ @tri_dao · Jul 10, 2025

I really like Phil Tillet's framing of different tools having different tradeoffs in productivity and performance: torch compile, triton, CUDA, PTX. It's still early but CuTe-DSL and similar Python-based DSL might bend this curve. And soon we can probably get LLMs to generate these kernels!

Phil's talk here: [semianalysis.com/wp-content/upl...](https://semianalysis.com/wp-content/uploads/2025/07/Phil-Tillet-Talk-2025-07-10.pdf)



From: Helion: A High-level DSL for Kernel Authoring,
by Oguz Ulgen (Meta), PyTorchCon Europe 2026

Why Paged Attention ? (to study performance portability)

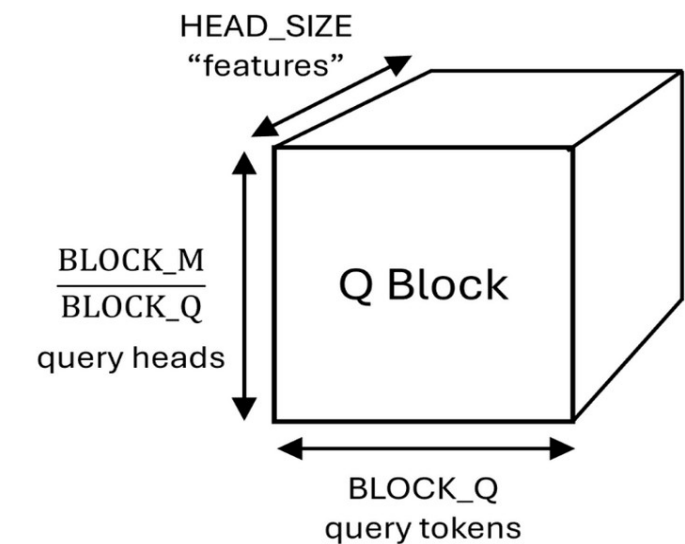
- Paged attention is the memory efficient version of attention, the core of transformers
- One of AI's most performance-critical kernel
 - Critical for (nearly) every LLM (Hybrid, MoE, Dense, ...)
- Relatively complex, many details need to be correct
 - e.g. the API with optimized inference servers like vLLM
 - Numerical stability
 - Large variety of workloads
 - Complex system integration: CUDA/HIP graph recording & replay
 - Roughly “400 million times more complex” than non-paged attention (as we will see)
- And, a rich body of well known optimizations exist
 - (FlexAttention, flash_attn2/3, FlashDecode++, ...)
 - So we can learn from them
 - ...and we can benchmark against meaningful baselines

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

How to write a high-performance portable paged attention in Helion... (1/N)

- Naive version (loop over batch, head, queries..) easy to write, however it leaves performance on the table due to tail waves
- We learned from our Triton Attention backend:
 - Batch size / num_requests should not be part of the launch grid
 - So focus the outer loop around “Q Blocks”

$$\frac{\text{BLOCK_M}}{\text{BLOCK_Q}} = \frac{\text{num_query_heads}}{\text{num_kv_heads}}$$



Explicit declaration of tuneable blocksizes (since used across loops)

```
q_block_size = hl.register_block_size(1, q_block_padded_size)
num_pages_at_once = hl.register_block_size(1, 32)
```

```
# Host-side upper bound -> takes tuned tile size into account
num_q_blocks = t_query.size(0) // q_block_size + num_seqs
```

```
for q_blk_tile, tile_m in hl.tile(
    [num_q_blocks, num_query_heads],
    block_size=[1, num_queries_per_kv],
):
    gid = q_blk_tile.begin # is scalar
```

Outer loop becomes the launch grid

`hl.tile` → core construct of Helion

How to write a high-performance portable paged attention in Helion... (2/N)

Do attention calculation
(with online softmax)
(pytorch code, quite
similar to flash attention)

More detailed explanation and the
code can be found in the blog post
earlier this year:

<https://pytorch.org/blog/portable-paged-attention-in-helion/>

```
q_load_mask = adjusted_tile_q_index[:, None, None] < query_end
# (tile_q, tile_m, HEAD_SIZE)
q = hl.load(t_query,
            [adjusted_tile_q_index, query_head_offset, hl.arange(head_size)],
            extra_mask=q_load_mask,
        )

# adjust for causal mask
max_seq_prefix_len = context_len + q_block_offset + block_m_size + 1
max_seq_prefix_len = torch.minimum(max_seq_prefix_len, seq_len)
num_blocks = torch.ceil(max_seq_prefix_len / page_size)
for tile_n in hl.tile(num_blocks, block_size=num_pages_at_once):
    block_n_size = num_pages_at_once * page_size
    # explicit load due to wrong if tile_n is partial
    blk_idx = hl.load(t_block_tables, [block_table_offset +
                                       tile_n.begin + hl.arange(num_pages_at_once)],)
    blk_idx = blk_idx.view([num_pages_at_once]).to(torch.int64)

    # (tile_n, PAGE_SIZE, 1, HEAD_SIZE)
    k_load = t_key_cache[blk_idx, :, kv_head_idx, :]
    v_load ...

    S = hl.dot(q, k, out_dtype=torch.float32, acc=S) * scale
    ...
    acc = hl.dot(P.to(v.dtype), v, out_dtype=torch.float32, acc=acc)
    ...

# epilogue
acc = acc / L[:, None]
hl.store(t_output,
        [adjusted_tile_q_index, tile_m.index, hl.arange(head_size)],
        acc.view([q_block_size, num_queries_per_kv, head_size]),
        extra_mask=q_load_mask,
    )
```

Load queries & extracting
correct tokens from “flattened
varlen” batch

Loop over KV cache (tuned
block size in vllm **pages**)

Final normalization & store the
result

Difference Helion / Triton for kernel authors (1)

- No need to take care of masks...

Triton

```
dim_mask = tl.where(offsets_d < HEAD_SIZE, 1, 0).to(tl.int1)
...
seq_offset = j * TILE_SIZE + offsets_t
tile_mask = seq_offset < max_seq_prefix_len
...
k_offset = (
    physical_block_idx[None, :] * stride_k_cache_0
    + kv_head_idx * stride_k_cache_2
    + offsets_d[:, None] * stride_k_cache_3
    + (seq_offset % BLOCK_SIZE)[None, :] * stride_k_cache_1
)
# K : (HEAD_SIZE, TILE_SIZE)
K_load = tl.load(
    key_cache_ptr + k_offset,
    mask=dim_mask[:, None] & tile_mask[None, :],
    other=0.0,
)
```

Helion

```
k = t_key_cache[blk_idxs, :, kv_head_idx, :]
```

...nearly. Except, data dependent irregular accesses, as
e.g. vLLMs “flattened varlen batch layout”



```
q_load_mask = adjusted_tile_q_index[:, None, None] < query_end
# (tile_q, tile_m, HEAD_SIZE)
q = hl.load(
    t_query,
    [adjusted_tile_q_index, query_head_offset, hl.arange(head_size)],
    extra_mask=q_load_mask,
)
```

Difference Helion / Triton for kernel authors (2)

- No need to take care of pointer arithmetic / TMA loads etc.

Triton

```
base = (  
    cache_ptr  
    + physical_block_idx_scalar * stride_cache_0  
    + kv_head_idx * stride_cache_2  
)  
desc = tl.make_tensor_descriptor(  
    base=base,  
    shape=(BLOCK_SIZE, HEAD_SIZE),  
    strides=(stride_cache_1, stride_cache_3),  
    block_shape=(TILE_SIZE, HEAD_SIZE_PADDED),  
)  
return desc.load([offset_in_block, 0])
```

Helion

```
k = t_key_cache[blk_idx, :, kv_head_idx, :]
```

+ autotuner config knob
(automatically discovered)

```
helion.Config(  
    ...  
    indexing=[..., 'tensor_descriptor', ... ],  
    ...
```

Triton→ Helion: Big difference in autotuning

- Triton and Helion both do autotuning, but...
- Triton does exhaustive search
- Helion....
 - autotuner has more degree of freedom
 - ...derives the search space!
→ No (wrong/biased) definition of programmers necessary
 - employs Bayesian Optimization, Genetic algorithms, pattern search
- still sometimes helpful to set boundaries...
 - but they are only at “data/application level”, not “hardware level”

```
q_block_size = hl.register_block_size(1, q_block_padded_size)
num_pages_at_once = hl.register_block_size(1, 32)
```

```
configs: list[triton.Config] = []
for block_q in (1, 2, 4, 8, 16):
    for tile_size in (16, 32, 64, 128, 256):
        for num_warps in (2, 4, 8):
            for num_stages in (1, 2, 3, 4):
                configs.append(
                    triton.Config(
                        {"BLOCK_Q": block_q, "TILE_SIZE": tile_size},
                        num_warps=num_warps,
                        num_stages=num_stages,
                    )
                )
```

Triton: Nested loops to define config list
→ easy to miss smth.

Helion: Config space automatically derived
(yes, you don't see any config definition here)

```
for tile_b, tile_m, tile_n in hl.tile([b, m, n]):
    acc = hl.zeros([tile_b, tile_m, tile_n], dtype=torch.float32)
    for tile_k in hl.tile(k):
        acc = torch.baddbmm(
            acc, A[tile_b, tile_m, tile_k], B[tile_b, tile_k, tile_n]
        )
    out[tile_b, tile_m, tile_n] = acc
```

How to write a high-performance portable paged attention in Helion... (3/N)

- Applying other well known tricks
(FlexAttention, flash_attn2/3, FlashDecode++)
 - Using exp2 instead of exp and changing the scale accordingly
 - Split the inner loop: Causal mask does not always need to be calculated
- → **193 LoC in total**
(naive version was 130)
- All optimizations combined:
Between 1.2x and 23x
speedup over naive version

Decorator:

```
@helion.kernel(  
    allow_warp_specialize=True,  
    static_shapes=False,  
    index_dtype=torch.int64,  
    autotune_baseline_fn=_triton_2d_attention,  
    autotune_effort="full",  
    autotune_ignore_errors=True,  
)  
def kernel_helion_unified_attention(  
    ...  
)
```



```
diag_start = context_len + q_block_offset + 1  
full_pages = (diag_start // (num_pages_at_once_full * page_size)) * num_pages_at_once_full  
full_pages = torch.minimum(full_pages, num_blocks)  
  
# — full tiles: no causal mask —————  
for tile_n in hl.tile(full_pages, block_size=num_pages_at_once_full):  
    ...  
  
# — boundary tiles: with causal mask applied —————  
for tile_n in hl.tile(  
    full_pages, num_blocks, block_size=num_pages_at_once  
):  
    ...
```

How to write a high-performance portable paged attention in Helion... (4/N)

- Compared to other kernels, we did NOT need to do
 - Merging masks and loads (`h1.load`, `h1.where`)
 - Hoisting constant masks outside the loop (GQA masks etc.)
 - Adding extra masks based on loop lengths (KV loop boundary)
 - Manually merge load and transpose
 - Reorder long query loops first → shorten tail waves
- These were done “gladly” by Helion automatically
- Other optimizations like reordering for caches (e.g. the order of the query tokens) are part of the autotuner search space (e.g. `12_groupings`)
 - So the compiler does NOT always apply them
 - But the autotuner can search them

Helion can't be thought of without autotuning

- It's not an “add-on”, it is integral part of the design
→ central for performance portability
- Helions autotuner has a lot degree of freedom
→ Search spaces are huge
 - Helion can analyze them with
`HELION_AUTOTUNE_LOG_SEARCH_SPACE_VERBOSE=1`
 - Which is the cartesian product of all dimensions
(→ not all of the combinations are “legal”)
- Paged attention has a ~400 million times larger search space than the non-paged attention example

Kernel (from helion/examples/)	Total search space
add	1.003e12
rms_norm_rwd	5.805e13
matmul	1.815e15
attention (“normal”)	3.111e15
paged_attention (ours)	1.296e24

(Numbers are for one specific shape, datatype, etc.)

Why is it so big?

- ~12 septillion configurations for one paged_attention kernel
 - Roughly as many water molecules as in a glass of water
 - Or the total count of space dust in the inner solar system
- Reason: “Helion can play a lot of tricks”
- Exact number varies slightly for batch shape, model definitions, datatype etc.

HELION_AUTOTUNE_LOG_SEARCH_SPACE_VERBOSE=1

Total search space size:

12,956,511,471,973,171,200,000,000

Per-feature exploration:

- indexing: 103/4096 options tested (2.5%)
- range_num_stages: 51/125 options tested (40.8%)
- range_unroll_factors: 52/125 options tested (41.6%)
- range_warp_specializes: 15/27 options tested (55.6%)
- maxnreg: 4/5 options tested (80.0%)
- block_sizes: 33/36 options tested (91.7%)
- range_flattens: 25/27 options tested (92.6%)
- loop_orders: 2/2 options tested (100.0%)
- flatten_loops: 1/1 options tested (100.0%)
- l2_groupings: 7/7 options tested (100.0%)
- reduction_loops: 1/1 options tested (100.0%)
- range_multi_buffers: 27/27 options tested (100.0%)
- static_ranges: 1/1 options tested (100.0%)
- num_warps: 6/6 options tested (100.0%)
- num_stages: 8/8 options tested (100.0%)
- atomic_indexing: 1/1 options tested (100.0%)
- host_tensor_descriptors: 2/2 options tested (100.0%)
- pid_type: 3/3 options tested (100.0%)
- num_sm_multiplier: 8/8 options tested (100.0%)

Why **seeding** the autotune search space can help

Pattern Search

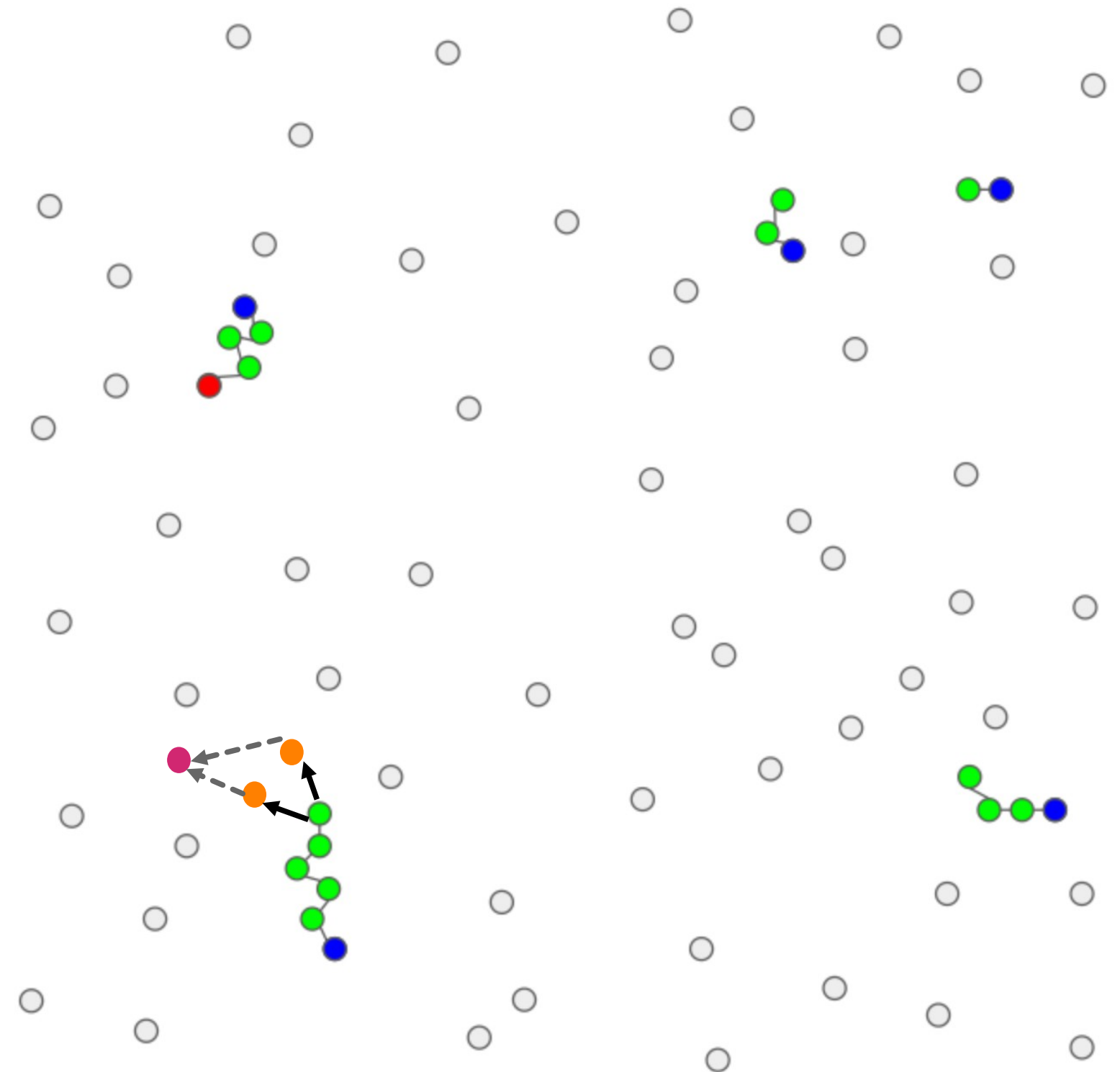
(also neighbor generation in TreeSearch)

- 1. Initial population of 100 random configs
- 2. Select top 5 fastest
- 3. Hill-climb to local minimums
- 4. Select fastest
- Bad performing neighbor, one permutation away (e.g. increasing block size **OR** increasing num_warps)
- Good performing neighbor, **two** permutations away (e.g. increasing block size **AND** increasing num_warps)

→ Is not discoverable by the pattern search*

→ fix: seed specific combinations

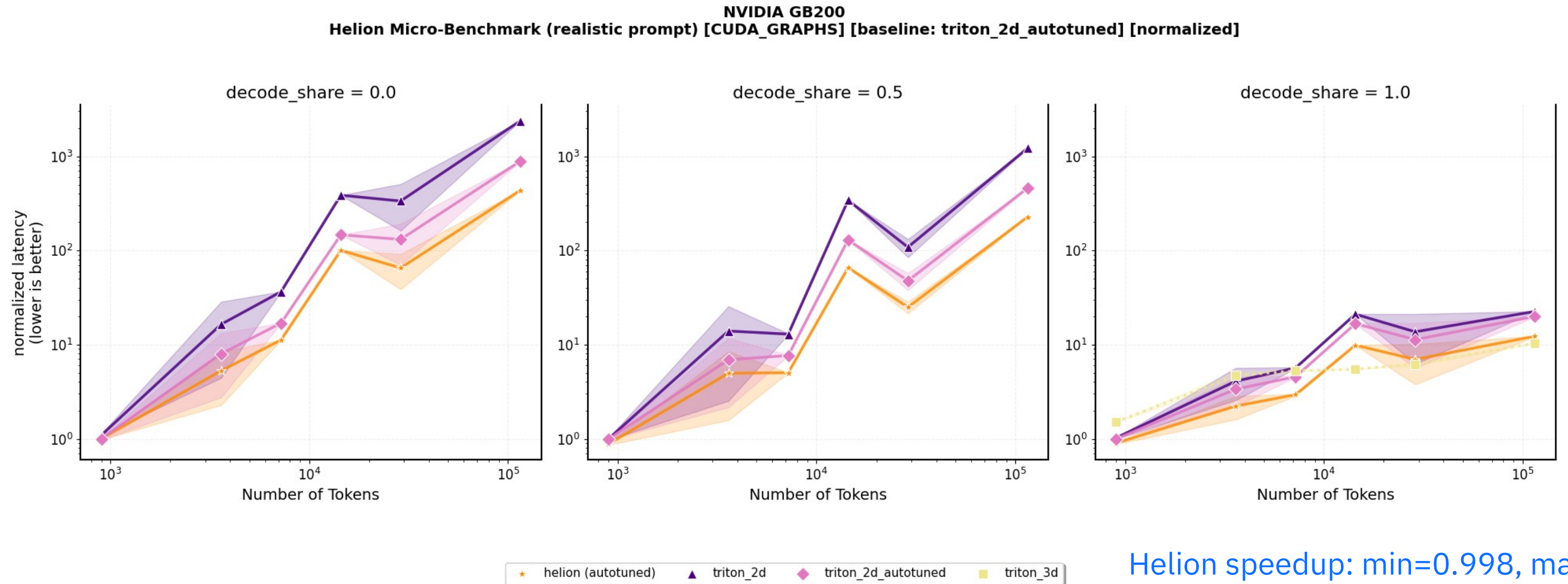
Speedup: up to 5.6x improvement



See also blog post: pytorch.org/blog/from-minutes-to-seconds-llm-guided-autotuning-for-helion-kernels/

*) the pattern search also evaluate pairs, but not along all dimensions/features

Micro-Benchmark: Paged Attention in Helion vs. Triton on B200



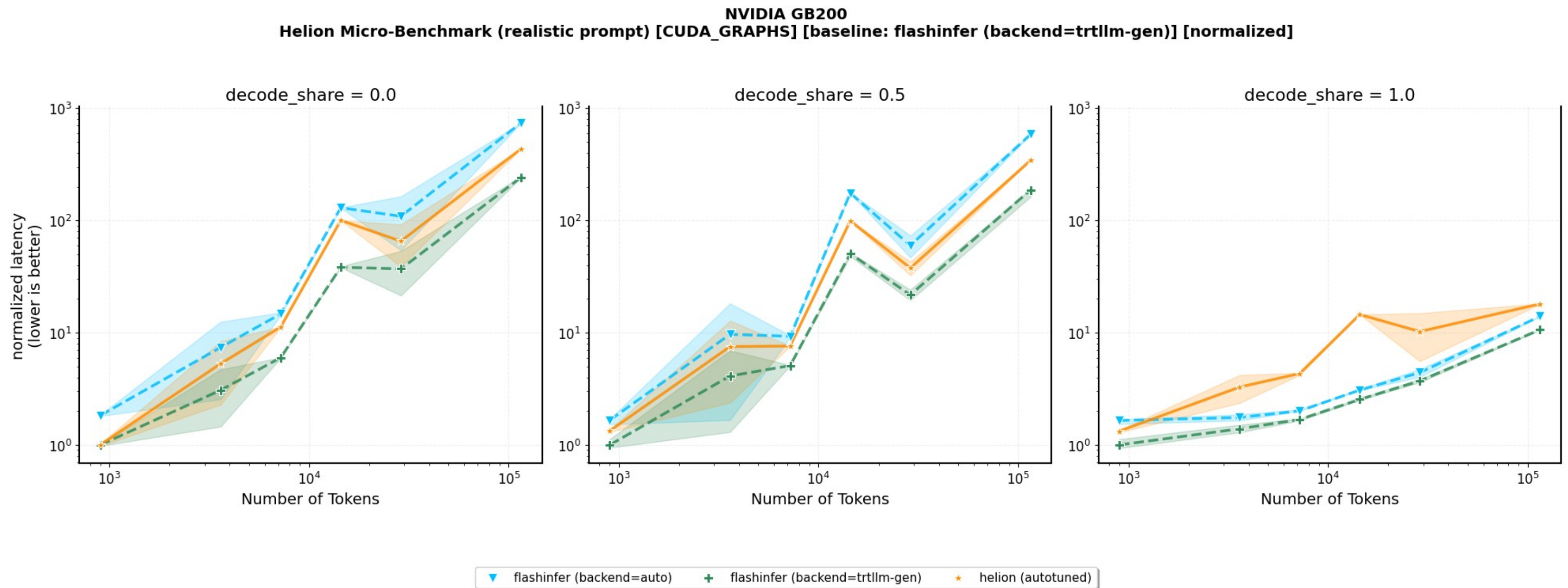
- Micro-benchmark mirroring vLLM setup, with varlen batches, using evaluation with cuda graphs → no software overhead (with a distribution of sequences that have on average 45% of the maximum sequence length)
- Triton autotuned done for comparison, “triton 2d/3d” are the kernels in upstream vllm (speedup autotuning Triton 2D up to 2.7x for this use case)
- Helion autotuning ~4 hours, Triton ~35min (search space is smaller)
- Helion has no parallel tiled softmax (like triton_3d) → nevertheless, comes close

How can Helion outperform Triton kernels if it is emitting Triton code?

- Helion specializes more (i.e. due to more autotuning)
- Helion generates highly different Triton code for different workloads
 - whereas Triton may generate different PTX files, but is the same Triton kernel
- E.g. on Blackwell, Helion can generate Triton code that could use more & more frequently tcgen05 features
 - for this, the Triton kernel would need to be re-written for every workload, GPU, etc.

PTX feature presence (fraction of kernels containing >= 1)		
feature	Triton (n=313)	Helion (n=24)
tcgen05.mma / .ld / .st / .alloc	43 (14%)	16 (67%)
mma.sync	270 (86%)	8 (33%)
ldmatrix	313 (100%)	24 (100%)
stmatrix	293 (94%)	8 (33%)
mbarrier	43 (14%)	16 (67%)
elect.sync	43 (14%)	16 (67%)
fence.proxy	43 (14%)	22 (92%)
cp.async.ca/cg	235 (75%)	22 (92%)
shfl.sync / redux.sync	313 (100%)	24 (100%)
ex2_approx	313 (100%)	24 (100%)
ld.global.L1::evict_*	0 (0%)	22 (92%)
cp.async.bulk_tensor (TMA)	0 (0%)	0 (0%)
setmaxnreg (warp spec)	0 (0%)	0 (0%)
st.local (register spill)	0 (0%)	0 (0%)

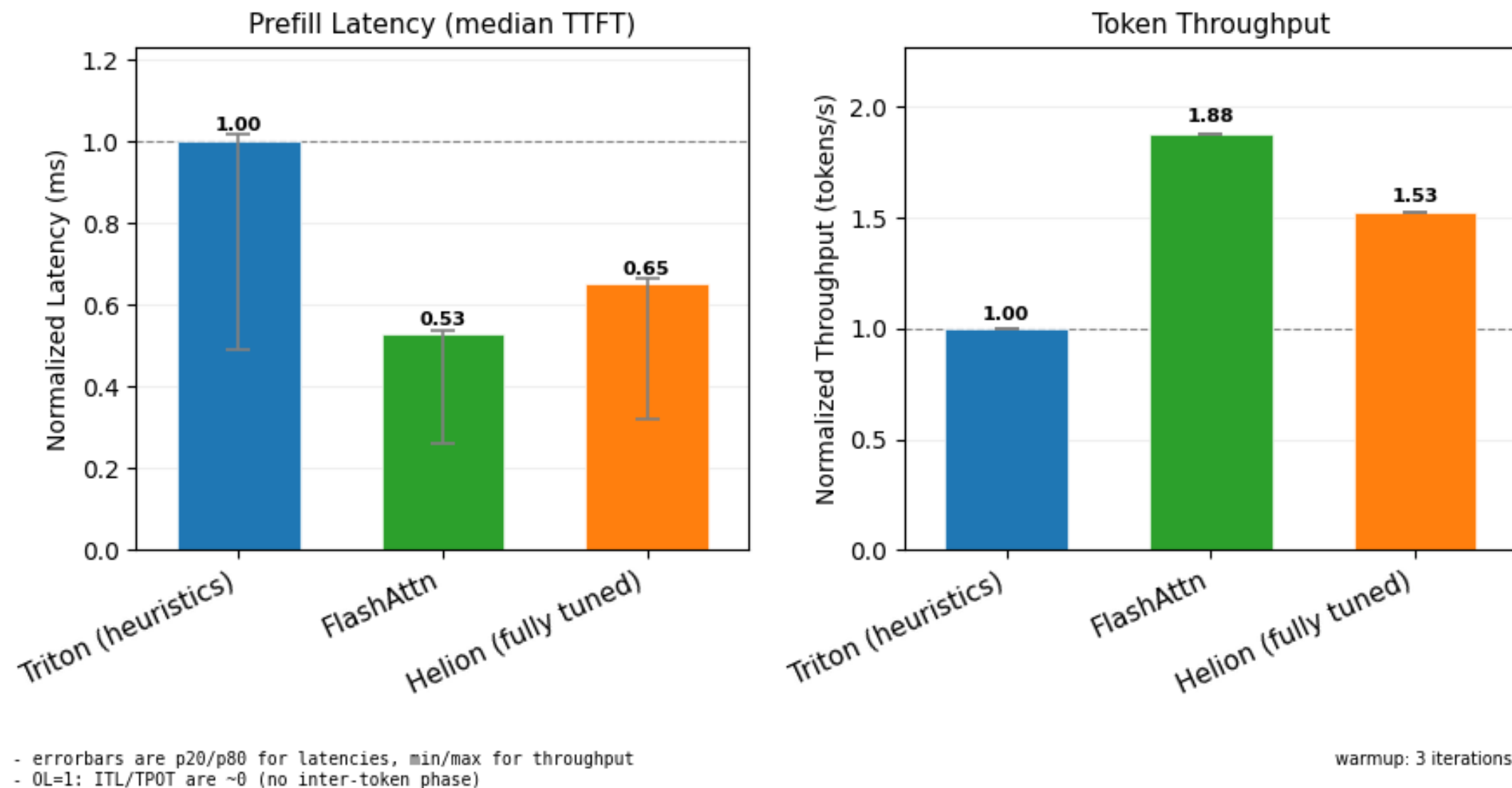
Paged Attention in Helion vs. flashinfer on B200



- Helion speedup vs. flashinfer (backend=auto): min=1.296, max=1.823
- Helion speedup vs. flashinfer (backend=trtllm-gen): min=0.382, max=0.996
- Micro-benchmark mirroring vLLM setup, with varlen batches, using evaluation with cuda graphs → no software overhead (with a distribution of sequences that have on average 45% of the maximum sequence length)

E2E: Some numbers can be impressive

vllm bench serve — random comprehensive — Qwen3-32B on H100 (IL=8192, OL=1, C=8)
(normalized to Triton (heuristics))
(Qwen3-32B on H100, IL=8192, OL=1, concurrency=8, no prefix caching)



- In micro-benchmarks: Helion exhibits a speedup vs. flash_attn3 with min=0.582, max=1.157
- End-to-end: similar result, but ~24hour tuning and the speedup of 50% of Triton vs Helion does not translate to other workloads
- However, using the same tuned kernels (with a heuristic via a decision tree) for shareGPT: Helion achieves only ~85% of Triton (throughput)
- → “out-of-the-box” performance for any workload remains a challenge

Conclusions?

- Performance portability critical for ubiquitous AI
- Helion as ~1.5 year old language *has demonstrated to achieve performance portability* across platforms & hardware types
 - It is also easy to use for programmers with experience in Triton or Pytorch
- Despite it's high abstraction level, Helion can express most kernel optimizations
 - And even outperform kernels written in a “lower level” language like Triton or CUDA
- Some open questions remain:
 - How to manage autotuner search space sizes
 - How to make AOT performant & robust
 - How to “fine-tune” kernels at e.g. vllm startup time
- → looks like it will stay interesting!



...looking forward to all your questions!

Dr. Burkhard Ringlein

✉ ngl@zurich.ibm.com

in [burkhard-ringlein](https://www.linkedin.com/in/burkhard-ringlein)

