HADACORE: TENSOR CORE ACCELERATED HADAMARD TRANSFORM KERNEL

Krish Agarwal*
University of Texas at Austin
krishagarwal@utexas.edu

Rishi Astra*
University of Texas at Austin
raa3897@utexas.edu

Adnan Hoque
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
adnan.hoque1@ibm.com

Mudhakar Srivatsa
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
msrivats@us.ibm.com

Raghu Ganti
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
rganti@us.ibm.com

Less Wright
Meta AI
less@meta.com

Sijia Chen
Meta AI
sijiac@meta.com

ABSTRACT

We present HadaCore, a modified Fast Walsh-Hadamard Transform (FWHT) algorithm optimized for the Tensor Cores present in modern GPU hardware. HadaCore follows the recursive structure of the original FWHT algorithm, achieving the same asymptotic runtime complexity but leveraging a hardware-aware work decomposition that benefits from Tensor Core acceleration. This reduces bottlenecks from compute and data exchange. On Nvidia A100 and H100 GPUs, HadaCore achieves speedups of 1.1–1.4x and 1.0–1.3x, with a peak gain of 3.5x and 3.6x respectively, when compared to the existing state-of-the-art implementation of the original algorithm. We also show that our FP16 implementation is numerically accurate, enabling comparable accuracy on MMLU benchmarks when used in an end-to-end Llama3 inference run with quantized (FP8) attention.[1]

***K*eywords** Hadamard Transform · GPU · Tensor Cores · Optimization · Quantization · Deep Learning · LLM · Foundation Models

1 Introduction

QuaRot [1] and SpinQuant [6] both propose methods to increase the numerical accuracy of INT4 and INT8 quantization in LLMs. These methods rotate model weights and activations since a rotation is statistically likely to reduce the magnitude of outliers, as it "distributes" extreme values among other (less extreme) dimensions, and rotation is also an easily invertible operation using the inverse of a rotation matrix. These methods can also improve FP8 inference accuracy, such as in FlashAttention-3 [7].

Placing these rotation matrices into the model runtime introduces extra overhead. Naively, these rotations would be full matrix multiplications. For Llama-2 7B, if we used INT8 quantization, we might expect a theoretical 2x speedup compared to FP16 inference, but using FP16 matmuls for these rotations could increase the linear layer computation to 110% of the original FP16 model (supposing INT8 is half the cost).

*Equal contribution.

[1]Our code is publicly available at https://github.com/pytorch-labs/applied-ai/tree/main/kernels/cuda/inference/hadamard_transform

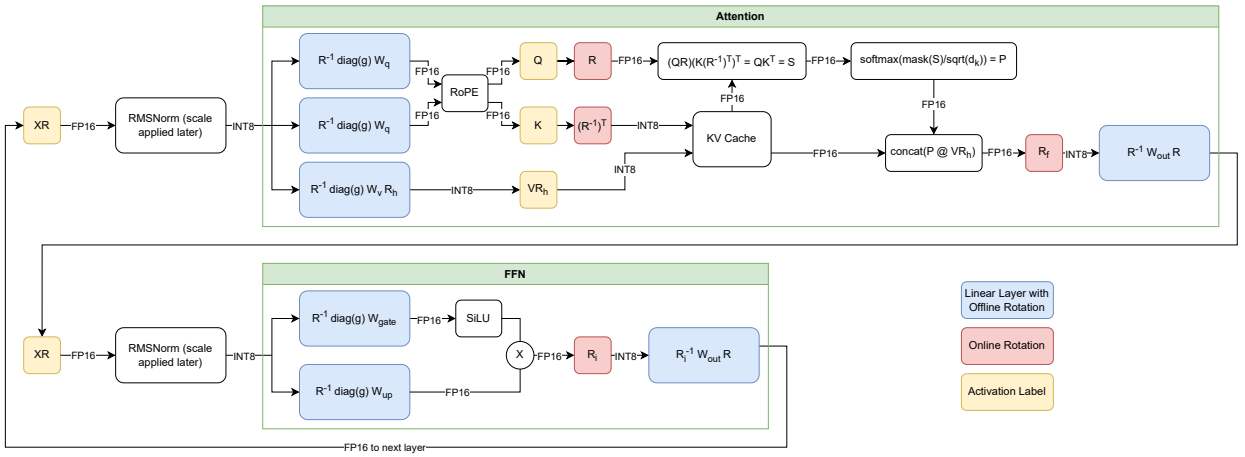


Figure 1: Transformer block showing online (red) and offline rotations (blue) in QuaRot.

QuaRot and SpinQuant therefore opt to use Walsh-Hadamard matrices, a special type of rotation matrix that can be applied much faster than a matrix multiplication (matmul) using the Fast Walsh-Hadamard Transform algorithm. Our work focuses on improving the performance of this transform.

2 Background

2.1 Hadamard Matrices

Hadamard matrices are square, orthogonal matrices with values exclusively either 1 or -1 ($\pm\frac{1}{\sqrt{d}}$ for a d-size Hadamard matrix when normalized). Walsh-Hadamard matrices are power-of-2 size Hadamard matrices which can be constructed recursively. Because of this recursive construction, multiplying an $m \times n$ activation matrix x with an n-sized Walsh-Hadamard matrix H can be performed in $O(mn \log(n))$ time as opposed to the $O(mn^2)$ time if general matrix multiplication was used. Furthermore, this algorithm, called the Fast Walsh-Hadamard Transform (FWHT), does not store H in memory, reducing matmul memory bottlenecks.

2.2 Fast Walsh-Hadamard Transform Algorithm

The Fast Walsh-Hadamard Transform algorithm [4] uses a recursive formula to apply an n-size Walsh-Hadamard transform to a vector/matrix x given the x already transformed by an $(n/2)$-sized Hadamard transform. This is very related to Sylvester's construction of Hadamard matrices [8], except that the Hadamard transform is applied without materializing the corresponding matrix.

```
def fwht(a) -> None:
    # In-place Fast Walsh-Hadamard Transform of array a.
    # Written by Wikipedia Contributors.
    h = 1
    while h < len(a):
        # perform FWHT
        for i in range(0, len(a), h * 2):
            for j in range(i, i + h):
                x = a[j]
                y = a[j + h]
                a[j] = x + y
                a[j + h] = x - y
        # normalize and increment
        a /= math.sqrt(2)
        h *= 2
```

In the above code, the outer loop represents the recursive iteration (the current Hadamard transform size is $2h$). The inner loop completes the current Hadamard transform size, with the middle loop repeating this to span all elements (e.g. if our vector is of size 8 or matrix of size $n \times 8$, the size-2 Hadamard would be repeated 4 times to span all elements, of size 4 repeated twice, of size 8 repeated once).

This algorithm has roughly $m \log_2(n) \cdot n$ floating point operations when applied to an $m \times n$ matrix, making it $O(mn \log(n))$.

2.3 Parallelization and Running on the GPU

In the base case of applying a size-2 Hadamard transform, work needs to be done on every pair of adjacent elements. Then in the next iteration, to accomplish a 4×4 Hadamard transform, work needs to be done on every pair of elements separated by 2 (i.e. 0 and 2, 1 and 3, 4 and 6, 5 and 7, etc.). This repeats for higher Hadamard sizes by nature of the algorithm. Importantly, each iteration depends on the output from the previous iteration.

The elements accessed in the inner code do not overlap between iterations of the inner and middle loop. This makes it simple to parallelize the 2 inner loops, which together span all m elements of the x vector/matrix, on a GPU. If x is a matrix, we can also parallelize across n rows. However, we cannot parallelize the outermost loop due to each iteration depending on the previous.

Additionally, there are synchronization challenges. Suppose that, per iteration, each thread does work on a single pair of elements. That thread will not be working on the same pair in the next iteration, which means it must have access to the results from another thread's work in this iteration to use in the next iteration.

Modern GPUs group threads into warps of 32 threads and further into threadblocks (CTAs) of up to 1024 threads. If each thread processes 2 elements (with the simple parallelization previously explained), we can process up to 64 elements (iteration 5) by exchanging data in the warp and up to 2048 (iteration 11) elements by exchanging within a threadblock through shared memory. If our Hadamard transform size is greater, each thread must process more than 2 elements, or expensive cross-threadblock synchronization and global memory are required.

2.4 `fast-hadamard-transform` Library

The Dao AI lab provides an optimized CUDA `fast-hadamard-transform` library [3] that implements the Fast Walsh-Hadamard Transform algorithm.

The library performs a right-Hadamard transform, parallelizing across rows of the input matrix through the launch grid and within each row by using up to 256 threads per row. The library processes 8 elements per thread (using fewer elements per thread was probably not worth the extra data movement and synchronization). We can interpret a row processed by a single threadblock to be shaped as `(n_chunk, warps_per_threadblock, threads_per_warp, 8)`.

- Each thread handles `n_chunk` chunks of 8 elements each (each 8 elements are contiguous, all threads' first chunks are contiguous together, then all second chunks, etc.)
- First process each chunk "owned" by individual threads (no syncing)
- Exchange elements within each warp and process them to carry out the next few iterations (no explicit synchronization required)
- Synchronize across the threadblock and transpose data so each warp has the data it needs for the next iterations
- Process elements within each warp again
- Synchronize and transpose data back
- Have each thread transpose its data so it has 8 chunks of `n_chunk` elements (as opposed to the other way around) and process each chunk within each thread (no syncing)

These optimizations provide 15 iterations of the algorithm with only 2 threadblock syncs, allowing up to a 2^{15}-size Hadamard transform.

3 Method

Modern Nvidia GPUs have specialized hardware called Tensor Cores [2], which can compute matrix multiplications on matrix sizes such as 16×16 about 8x faster than CUDA cores (general purpose GPU hardware). Using these Tensor

Cores would therefore boost our performance. However, the original fast Hadamard transform algorithm is explicitly an alternative to using matrix multiplication, and thus it cannot be directly run on Tensor Cores. The `mma` instruction can perform 16×16 by 16×8 matrix multiplication, so we can use 2 such operations to perform a 16×16 by 16×16 matrix multiplication. We can therefore perform a 16-size Hadamard transform on a 16×16 matrix chunk using these 2 Tensor Core `mma` operations. Furthermore, we can use inline assembly (PTX) to use the Tensor Cores with data already in registers.

3.1 Hadamard Sizes up to 256

Naively, if we were to integrate the above into the original Fast Walsh-Hadamard Transform algorithm, we could treat a size-16 Hadamard transform as a base case as opposed to size-2 Hadamard being the base case, for which we would use a Tensor Core. Just by using this to replace the first 4 outer iterations of the regular algorithm, this new algorithm takes $Cmn \log_2 \left(\frac{n}{16}\right)$ time, where C is the latency of the 2 Tensor Core `mma` instructions.

Although a $\frac{mn}{16} \times 16$ by 16×16 matmul using Tensor Cores instead of four $\frac{mn}{2} \times 2$ by 2×2 matmuls uses 2x more floating-point operations for the same result, Tensor Cores have ~8x the FLOPS and reduce the number of iterations thus reducing threadblock/global synchronizations.

To extend this, we can rearrange the elements such that applying the base case again achieves the next 4 iterations. First, we can apply the 16-size Hadamard to a 1×256 row vector chunk by reshaping it to 16×16. This effectively applies the 16-size Hadamard to groups of 16 elements in the row. Then to rearrange elements, we can transpose the 16×16 result and apply the 16×16 base case again (now acting on elements 16 apart due to the transpose). Finally, if we transpose the result back and view it again as a 1×256 row vector, we achieve a size-256 Hadamard transform.

Figure 2: 1×256 vectors (rows) visualized for rotating by a size-256 Hadamard. The batches (number of rows) can be split among warps of the GPU.

3.2 Supporting Sizes Larger than 256

So far, we would only achieve up to a 256-size Hadamard with the above strategy because every contiguous chunk of h elements must be synced to achieve an h-sized Hadamard. Suppose that for a $1 \times n$ row vector we are trying to do an n-size Hadamard transform, where $n = 2^k > 256$. We can first view this as a $\frac{n}{256} \times 256$ matrix and apply a 256-size Hadamard transform to $\frac{n}{256}$ separate chunks with the method described above. To go beyond 256, we can transpose the current result to a $256 \times \frac{n}{256}$ matrix and apply an $\frac{n}{256}$-sized Hadamard transform, again using Tensor Core `mma`s, and transpose the result back. However, we would lose parallelism if we were to have a single warp process the full $1 \times n$ vector, especially for larger sizes like 2^{15}.

The approach we use is to process a full $1 \times n$ vector per threadblock, where each threadblock has `warps_per_block` warps and each warp processes `num_chunks` chunks of 256 elements. We choose `warps_per_block` and `num_chunks` such that $256 \cdot$ `warps_per_block` $\cdot$ `num_chunks` $= n$. Our strategy is the following:

1. Have each warp process its chunks of 256 elements as described above (each 256-element chunk is a row in the $\frac{n}{256} \times 256$ view).
2. Store the result back in shared memory.
3. Sync across the threadblock.
4. Read from shared memory transposed so that each warp has $\frac{256 \cdot \texttt{num\_chunks}}{n/256}$ columns of $\frac{n}{256}$ elements from the $\frac{n}{256} \times 256$ view (this still amounts to each warp processing `num_chunks` chunks of 256 elements each).
5. Process the new data in chunks of 256 (where each chunk comprises of one or more columns) by using the same method as up to size 256 (just like the Dao AI Lab kernel, HadaCore supports up to size $2^{15} = 256 \cdot 128$, so this works out).

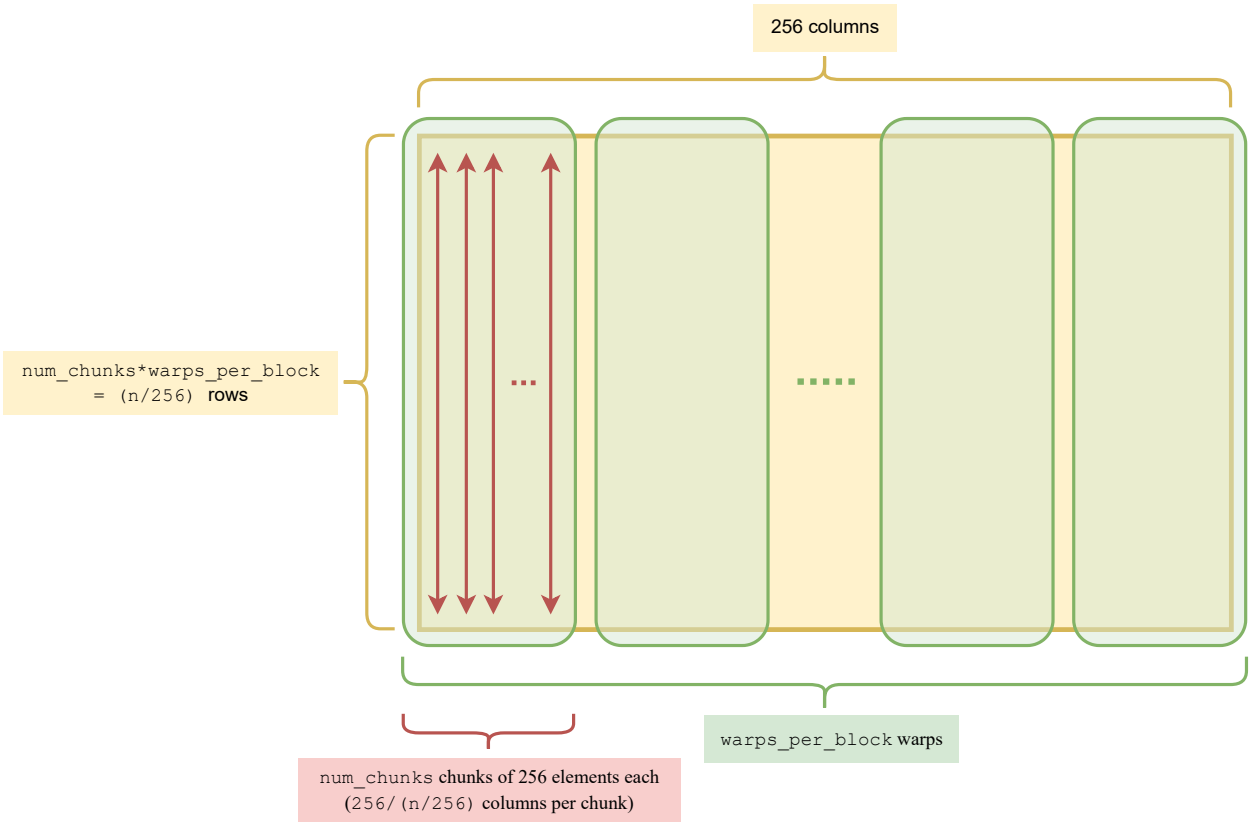


Figure 3: The 1×256 vectors now transposed, allowing us to operate on the factor $\frac{n}{256}$.

Loading transposed data means that the loads are uncoalesced. We solve this in the following ways:

- For sizes 512, 1024, and 2048, due to the Tensor Core register layout, we are able to simply load each chunk coalesced and then use warp-level shuffle operations to get each thread its own data.
- For sizes 4096 and above, we can coalesce our loads by loading all chunks upfront into registers and then doing warp-level shuffle operations to get each thread its own data.

3.3 Non-Power of 16 Sizes

The proposed algorithm relies on the desired Hadamard size being a power of 16 rather than a general power of 2, but this issue is easily resolved. If the desired Hadamard size is $d \neq 16^n$, we can factorize $d = 2^m \cdot 16^n$, where $0 < m < 4$. The 16^n-size Hadamard can be achieved with n iterations of applying a 16-size Hadamard with intermediate rearranging as specified above, and the 2^m size Hadamard can be achieved in one extra iteration by using an appropriate 16×16 matrix (e.g. for $m = 3$, have an 8×8 Hadamard repeated on the diagonal). In this way, the algorithm is the same as if the Hadamard size was the next power of 16, except that the last 16×16 matrix is a diagonal tiling of a smaller Hadamard.

3.4 Algorithm Overview

The original Fast-Walsh Hadamard Transform algorithm achieves a 2^n-size Hadamard transform given an input already transformed by a 2^{n-1}-size Hadamard. This is analogous to producing a 2^n size Hadamard using the Kronecker product of a 2^{n-1}-size Hadamard with a size-2 Hadamard. Similarly, by using a 16-size Hadamard per iteration, our algorithm is analogous to using a Kronecker product of a 2^{n-4} size Hadamard with a size-16 Hadamard to achieve a 2^n size Hadamard transform, without materializing the Hadamard matrix in memory.

Our algorithm requires $\left(\frac{mn}{16}\right)(16 \cdot 16)\lceil \log_{16}(n) \rceil = 16mn\lceil \log_{16}(n) \rceil \geq 16mn \log_{16}(n) = 4mn \log_2(n)$ floating-point operations. Comparatively, the regular algorithm requires $\left(\frac{mn}{2}\right)(2 \cdot 2) \log_2(n) = 2mn \log_2(n)$ floating-point-operations. With at least 2x the floating-point operations, we might expect our algorithm to be slower, but we still expect it to be faster because:

- By using Tensor Cores, we have 8x the FLOPS available to us.
- By using Tensor Core `mma` operations, we require less explicit data shuffling, meaning less warp stalls due to synchronization.
- This also means we perform far fewer non-Hadamard Transform operations: the `fast-hadamard-transform` library uses complicated indexing to achieve its warp-level data shuffling, meaning a much higher ALU load; in comparison, our warp-level shuffling is a simple hardware transpose.
- Our algorithm is more flexible to varying threadblock sizes, which means we are able to optimize our configurations to achieve higher occupancy compared to the `fast-hadamard-transform` library (this is especially apparent for a 128-size Hadamard transform as shown in our results section below).

4 Results

4.1 Performance Tests

We compare HadaCore's performance against the Dao AI lab's `fast-hadamard-transform` library on both an A100 and an H100. We measure our relative speeds across different Hadamard sizes, and we also vary the element count (analogous to number of input matrix rows). We chose to distinguish element count instead of row count since runtimes for different Hadamard sizes are most similar for the same amount of data rather than the batch size. The graphs and tables of our results are available in Appendix A.

Here are some notes about our results:

- We have a noticeable increase in performance for an element count of 8M on the A100 (16M on the H100) from an optimization we performed with using in-place rotations. This is explained in further detail in Appendix B.
- Using a Hadamard size of 512 and a small element count has noticeably worse speedup compared to our other results. 512 is the smallest size that requires more than one 256-size fragment to be synchronized. This means a Hadamard size of 512 incurs roughly the same overhead that all Hadamard sizes above 256 incur from threadblock syncing and shared memory data shuffling. While the `fast-hadamard-transform` CUDA kernel also performs the same threadblock sync above a size of 256, our shared memory data shuffling is likely more expensive due to adhering to Tensor Core register layouts. Additionally, our implementation incurs the full cost of a 16×16 matmul (a 2×2 Hadamard is tiled along the diagonal) while the Dao AI Lab's implementation only incurs an extra outer iteration of the general algorithm.
- Similarly, Hadamard size 8K has a relatively lower speedup since it requires the full 4 iterations of 16×16 matmuls (16^3 = 4K < 8K), the same amount as 32K, while 4K would require 3. Again, the `fast-hadamard-transform` library only incurs the cost of an extra outer iteration to go from 4K to 8K.
- With our coalescing scheme for loading transposed data from shared memory, we don't necessarily achieve full coalescing for Hadamard sizes 8K and above. For sizes 8K, 16K, and 32K, we require each warp to process 8, 16, and 32 chunks respectively for full coalescing. However, this may sacrifice parallelism, so we empirically select configurations for each of these Hadamard sizes. This results in varying runtimes between sizes that use the same amount of computation, such as 8K/16K/32K.
- The H100 results are overall worse than the A100 results. This is likely due to architectural differences, a different compute/bandwidth ratio, and different loading instructions. We focused on pre-Hopper GPUs during development.

4.2 End-To-End Tests

Beyond basic unit tests that check the output of HadaCore against the output of an explicit Hadamard matrix multiplication, we analyze MMLU [5] accuracy for Llama-3.1 8B to compare implementations that use FP8 attention with and without rotations. These tests validate the usefulness of Hadamard rotations for reducing quantization error as well as demonstrate that HadaCore does not sacrifice numerical accuracy by being faster.

	Average 5-Shot MMLU Accuracy (↑)	
FP16 Baseline	65.38	
FP8 attention (no rotation)	64.40	
	Dao AI Lab Kernel	**HadaCore**
FP8 attention (with rotation)	65.45	65.09

The results above show that HadaCore maintains a comparable quantization error reduction as the Dao AI Lab kernel for Hadamard rotations in FP8 attention.

5 Conclusion

We showcased our speedups achieved by moving the Fast Walsh-Hadamard transform algorithm into a CUDA kernel that leverages Tensor Core acceleration. HadaCore achieves speedups of around 1.1–1.4x and up to 3.5x the speed of the Dao AI CUDA kernel. Further, by running MMLU benchmarks on Llama-3.1 8B with FP8 attention, we show that rotating with HadaCore achieves similar quantization error reduction as the Dao AI CUDA kernel while providing computational acceleration. In the future, we plan to implement a version of HadaCore for OpenAI Triton [9], do more optimizations for the newer H100 Hopper architecture, and experiment with more advanced techniques such as kernel fusion to support fused Hadamard transform and quantization.

References

[1] Saleh Ashkboos et al. *QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs*. 2024. arXiv: 2404.00456 [cs.LG]. URL: https://arxiv.org/abs/2404.00456.

[2] Nvidia Corporation. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-matrix-functions. 2024.

[3] Tri Dao. *Fast Hadamard Transform in CUDA, with a PyTorch interface*. https://github.com/Dao-AILab/fast-hadamard-transform. 2024.

[4] Fino and Algazi. "Unified Matrix Treatment of the Fast Walsh-Hadamard Transform". In: *IEEE Transactions on Computers* C-25.11 (1976), pp. 1142–1146. DOI: 10.1109/TC.1976.1674569.

[5] Dan Hendrycks et al. *Measuring Massive Multitask Language Understanding*. 2021. arXiv: 2009.03300 [cs.CY]. URL: https://arxiv.org/abs/2009.03300.

[6] Zechun Liu et al. *SpinQuant: LLM quantization with learned rotations*. 2024. arXiv: 2405.16406 [cs.LG]. URL: https://arxiv.org/abs/2405.16406.

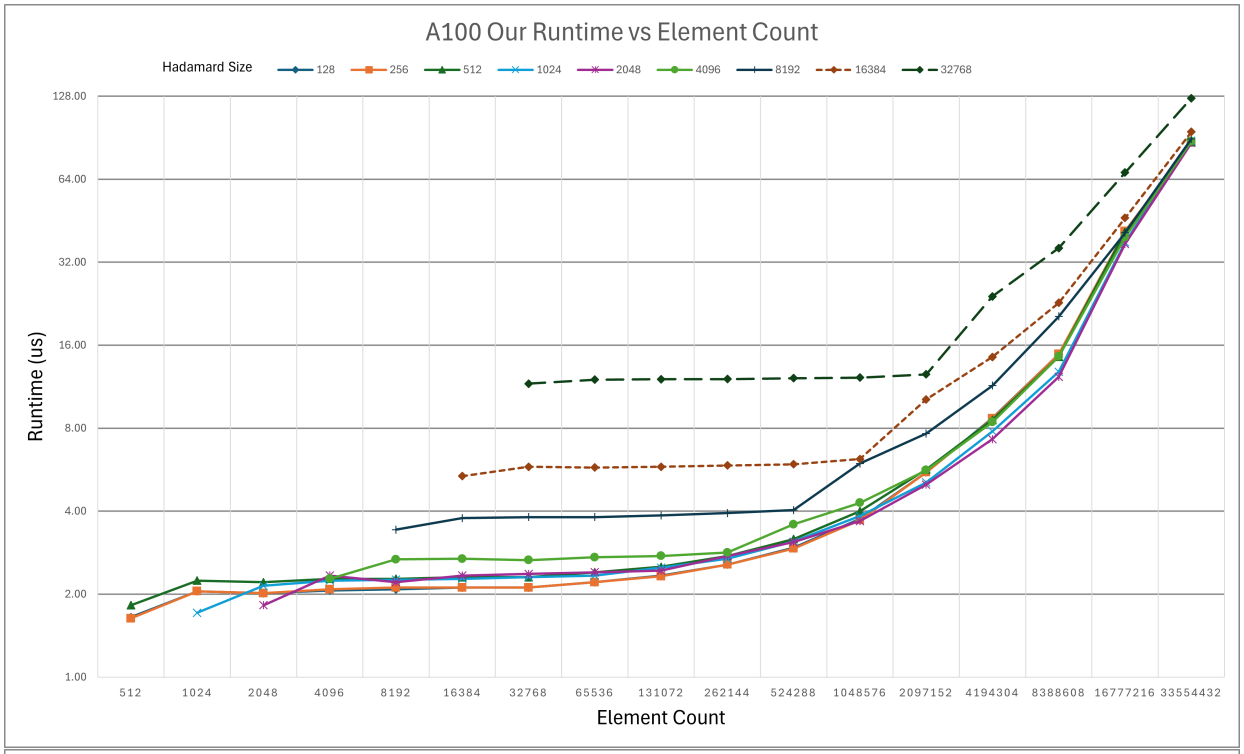
[7] Jay Shah et al. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. 2024. arXiv: 2407.08608 [cs.LG]. URL: https://arxiv.org/abs/2407.08608.

[8] James Sylvester. "LX. Thoughts on inverse orthogonal matrices, simultaneous signsuccessions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers". In: *Philosophical Magazine Series 1* 34 (1867), pp. 461–475. URL: https://api.semanticscholar.org/CorpusID:118420043.

[9] Philippe Tillet. *CUDA C++ Programming Guide*. https://openai.com/index/triton/. 2021.

A Graphs and Tables

A.1 Graphs



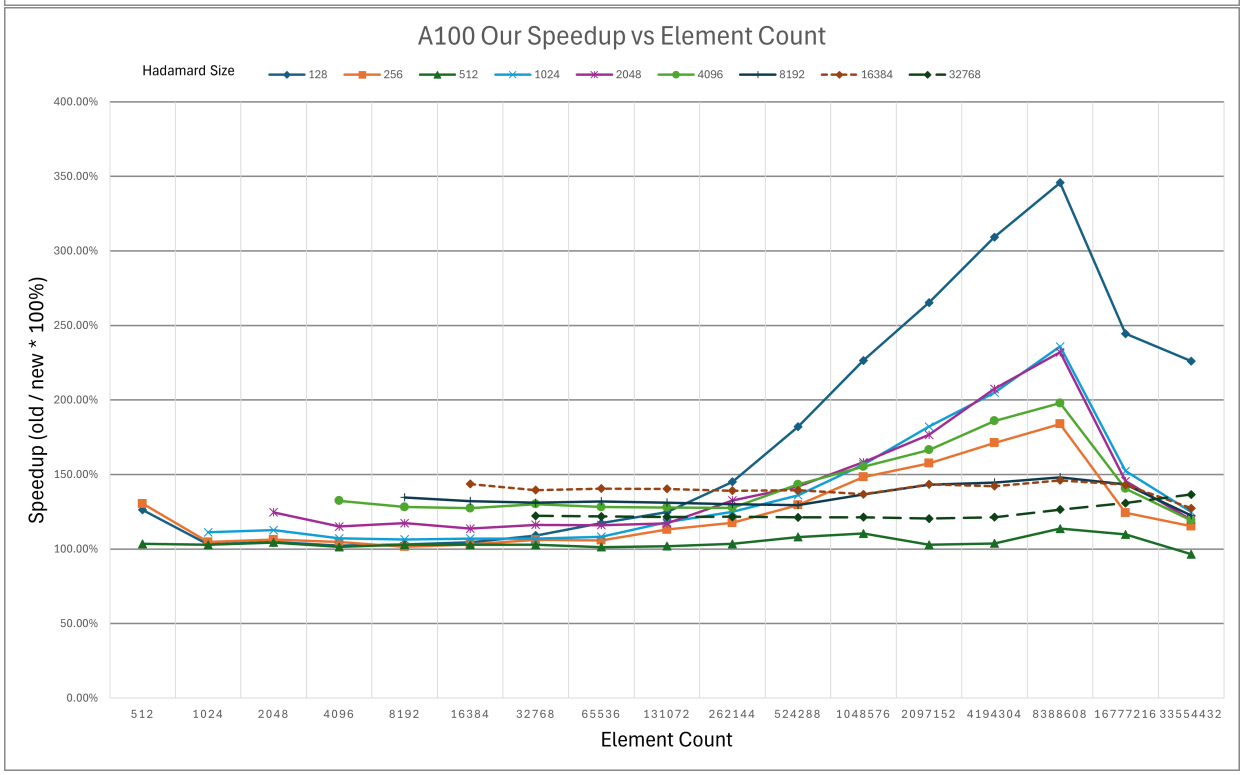
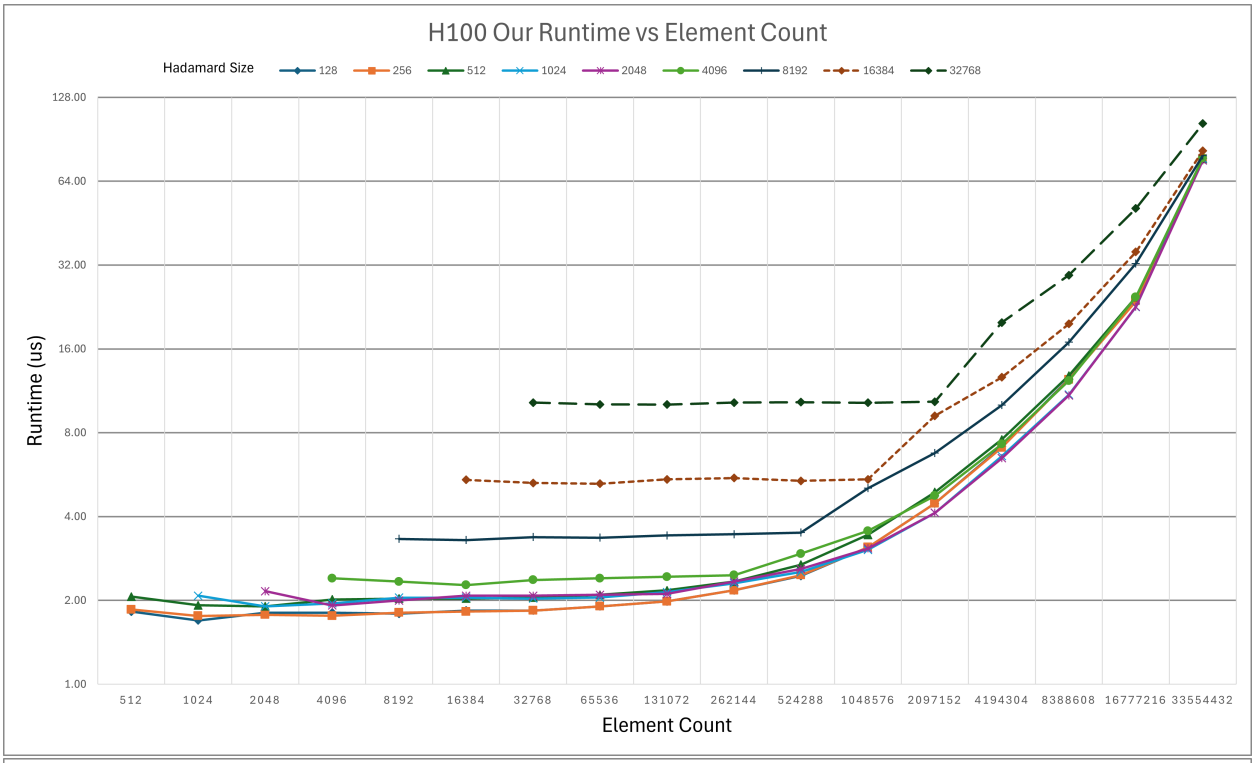


Figure 4: Graphs of runtime and speedup of HadaCore against the Dao AI Lab kernel, measured on an A100-PCIe. Each series represents a different rotation size, and the x-axis is the element count of the input tensor.



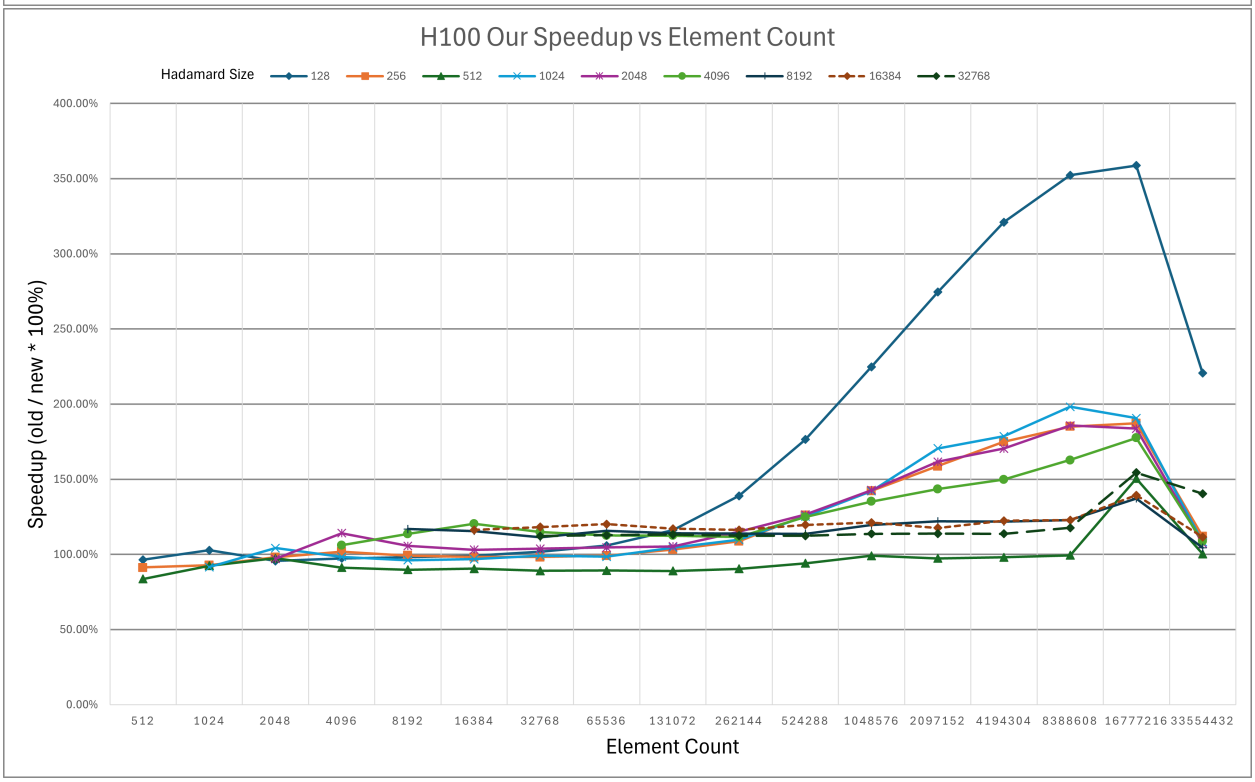


Figure 5: Graphs of runtime and speedup of HadaCore against the Dao AI Lab kernel, measured on an H100-PCIe. Each series represents a different rotation size, and the x-axis is the element count of the input tensor.

A.2 Tables

Hadamard Size \ Element Count	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432
128	1.65	2.05	2.02	2.06	2.08	2.11	2.11	2.21	2.34	2.56	2.94	3.74	5.54	8.70	14.85	40.88	87.18
256	1.63	2.05	2.02	2.08	2.11	2.11	2.11	2.21	2.32	2.56	2.93	3.71	5.54	8.67	14.85	41.46	86.93
512	1.82	2.24	2.21	2.27	2.27	2.30	2.30	2.40	2.51	2.75	3.17	4.00	5.66	8.58	14.54	41.15	87.12
1024		1.71	2.14	2.24	2.26	2.27	2.30	2.34	2.50	2.69	3.10	3.84	5.09	7.79	12.80	37.81	88.38
2048			1.82	2.34	2.21	2.34	2.37	2.40	2.43	2.75	3.09	3.68	4.99	7.30	12.29	37.28	86.70
4096				2.27	2.67	2.69	2.66	2.72	2.75	2.83	3.58	4.29	5.63	8.42	14.56	39.46	87.87
8192					3.42	3.78	3.81	3.81	3.86	3.94	4.03	5.95	7.65	11.41	20.34	40.91	89.66
16384						5.36	5.79	5.76	5.79	5.86	5.92	6.18	10.18	14.50	22.78	46.31	95.06
32768							11.60	12.00	12.03	12.05	12.14	12.21	12.54	24.03	36.00	67.66	125.95

(a) Runtime

Hadamard Size \ Element Count	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432
128	126.21%	103.13%	104.76%	102.33%	103.08%	104.55%	109.09%	117.39%	124.66%	145.00%	182.07%	226.50%	265.32%	309.20%	345.69%	244.50%	226.02%
256	130.39%	104.69%	106.35%	104.62%	101.52%	103.03%	106.06%	105.80%	113.10%	117.50%	129.54%	148.28%	157.52%	171.21%	183.84%	124.35%	115.30%
512	103.51%	102.86%	104.35%	101.41%	102.82%	102.78%	102.78%	101.33%	101.91%	103.49%	108.08%	110.40%	102.82%	103.73%	113.64%	109.80%	96.49%
1024		111.21%	112.69%	107.14%	106.38%	107.04%	106.94%	108.22%	117.95%	125.00%	136.08%	156.67%	182.08%	204.93%	235.88%	152.27%	124.87%
2048			124.56%	115.07%	117.39%	113.70%	116.22%	116.00%	117.11%	132.56%	141.97%	158.26%	176.60%	207.46%	232.04%	145.53%	119.58%
4096				132.39%	128.14%	127.38%	130.12%	128.24%	127.91%	127.68%	143.30%	155.22%	166.48%	185.93%	197.92%	140.51%	119.26%
8192					134.58%	132.20%	131.09%	131.93%	131.12%	130.08%	129.37%	136.56%	143.10%	144.60%	148.07%	143.25%	122.55%
16384						143.58%	139.51%	140.56%	140.33%	139.07%	139.46%	136.79%	143.40%	142.16%	146.07%	143.12%	127.32%
32768							122.20%	121.87%	121.54%	121.65%	121.21%	121.36%	120.41%	121.31%	126.44%	130.83%	136.61%

(b) Speedup

Figure 6: Tables of runtime (in µs) and speedup of HadaCore against the Dao AI Lab kernel, measured on an A100-PCIe.

Hadamard Size \ Element Count	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432
128	1.82	1.70	1.81	1.81	1.79	1.84	1.84	1.90	1.98	2.18	2.45	3.10	4.46	7.14	12.48	24.08	77.26
256	1.86	1.76	1.78	1.76	1.81	1.82	1.84	1.90	1.98	2.18	2.46	3.10	4.46	7.07	12.42	23.74	77.25
512	2.06	1.92	1.90	2.02	2.03	2.03	2.05	2.10	2.18	2.34	2.69	3.44	4.90	7.57	12.83	24.58	76.96
1024		2.08	1.90	1.95	2.05	2.05	2.03	2.05	2.14	2.31	2.53	3.04	4.13	6.59	10.96	22.64	77.22
2048			2.16	1.92	2.00	2.08	2.08	2.10	2.11	2.34	2.59	3.07	4.13	6.48	10.90	22.70	76.03
4096				2.40	2.34	2.27	2.37	2.40	2.43	2.46	2.94	3.55	4.75	7.28	12.30	24.56	77.63
8192					3.33	3.30	3.38	3.36	3.42	3.46	3.50	5.06	6.77	10.05	16.91	32.42	79.15
16384						5.42	5.28	5.25	5.44	5.50	5.38	5.44	9.20	12.66	19.68	35.66	82.30
32768							10.27	10.11	10.10	10.27	10.30	10.26	10.34	19.89	29.44	51.07	103.12

(a) Runtime

Hadamard Size \ Element Count	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432
128	96.49%	102.83%	95.60%	97.37%	98.19%	99.13%	101.74%	105.88%	116.13%	138.95%	176.47%	224.74%	274.55%	321.08%	352.31%	358.73%	220.71%
256	91.35%	92.73%	98.20%	101.79%	99.17%	98.25%	98.23%	99.13%	103.23%	108.82%	125.97%	142.28%	158.78%	174.89%	185.05%	187.20%	111.99%
512	83.72%	92.47%	97.48%	91.27%	89.78%	90.55%	89.06%	89.31%	88.97%	90.43%	94.08%	99.08%	97.38%	98.10%	99.38%	150.52%	100.33%
1024		91.54%	104.23%	98.36%	96.14%	96.87%	99.21%	98.44%	104.50%	109.67%	125.32%	142.09%	170.54%	178.64%	198.25%	190.60%	108.25%
2048			97.06%	114.17%	105.60%	103.10%	103.82%	104.58%	105.30%	115.07%	126.52%	142.71%	161.64%	170.38%	185.76%	183.72%	107.30%
4096				106.02%	113.70%	120.42%	114.86%	112.64%	112.50%	111.69%	125.00%	135.15%	143.43%	149.88%	162.82%	177.46%	108.57%
8192					116.83%	115.55%	111.37%	115.71%	114.02%	113.89%	113.71%	119.62%	121.98%	121.81%	122.80%	137.26%	104.47%
16384						116.22%	118.18%	120.12%	117.06%	116.29%	119.63%	121.19%	117.57%	122.38%	122.85%	139.39%	111.55%
32768							112.15%	112.82%	113.15%	112.47%	112.42%	113.73%	113.78%	113.75%	117.72%	154.39%	140.43%

(b) Speedup

Figure 7: Tables of runtime (in µs) and speedup of HadaCore against the Dao AI Lab kernel, measured on an H100-PCIe.

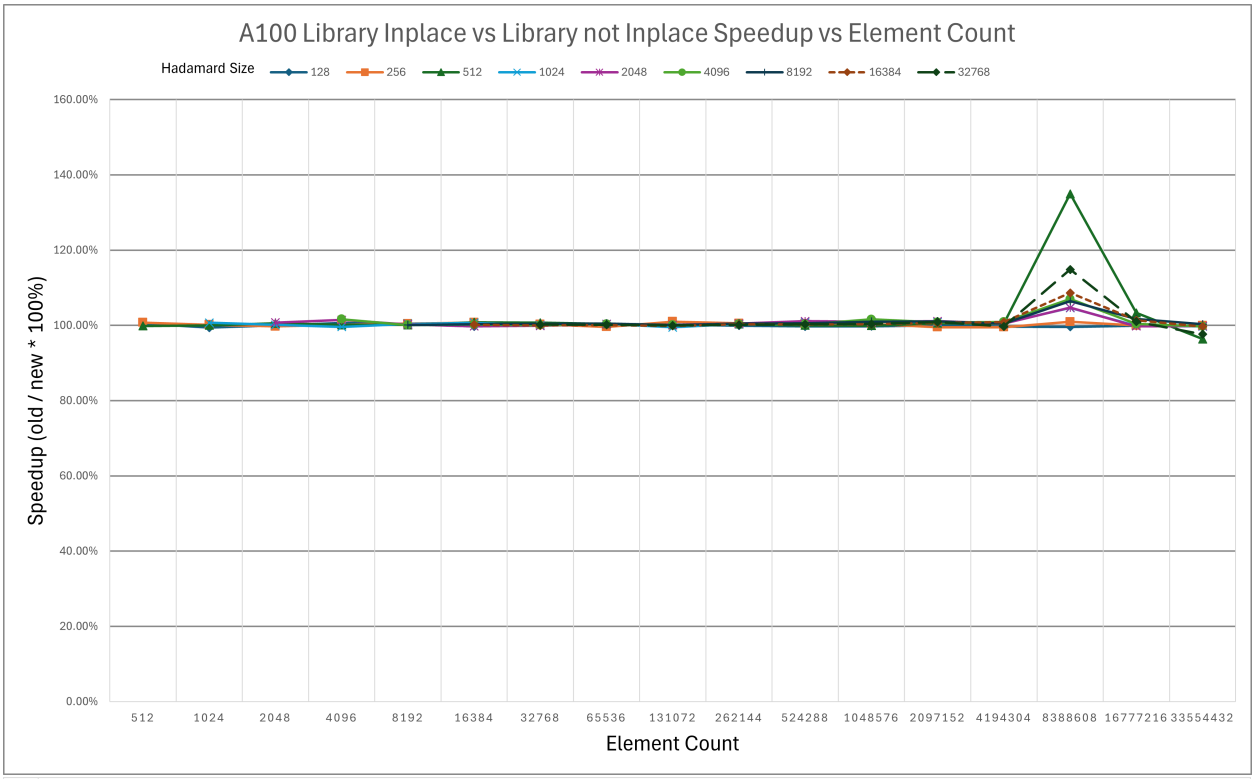
B In-Place Rotation

One simple optimization is modifying the input tensor in-place. If our tensor is 16M elements of FP16 (e.g. 4096×4096), it will fit in ~32MB of cache. However, if we have a separate destination tensor, we need double that, ~64MB. The H100, A100, and L40S have 50MB, 40MB, and 48MB of L2 cache respectively, so for these enterprise GPUs, the source and destination tensors will evict each other's lines from cache. While this analysis should theoretically only apply for sizes larger than half the L2 cache size but less than the full L2 cache size (like 32MB), in practice it might be different depending on the eviction policy and other memory usage.

This optimization can be applied to the `fast-hadamard-transform` library with a small change in `/csrc/fast_hadamard_transform.cpp`:

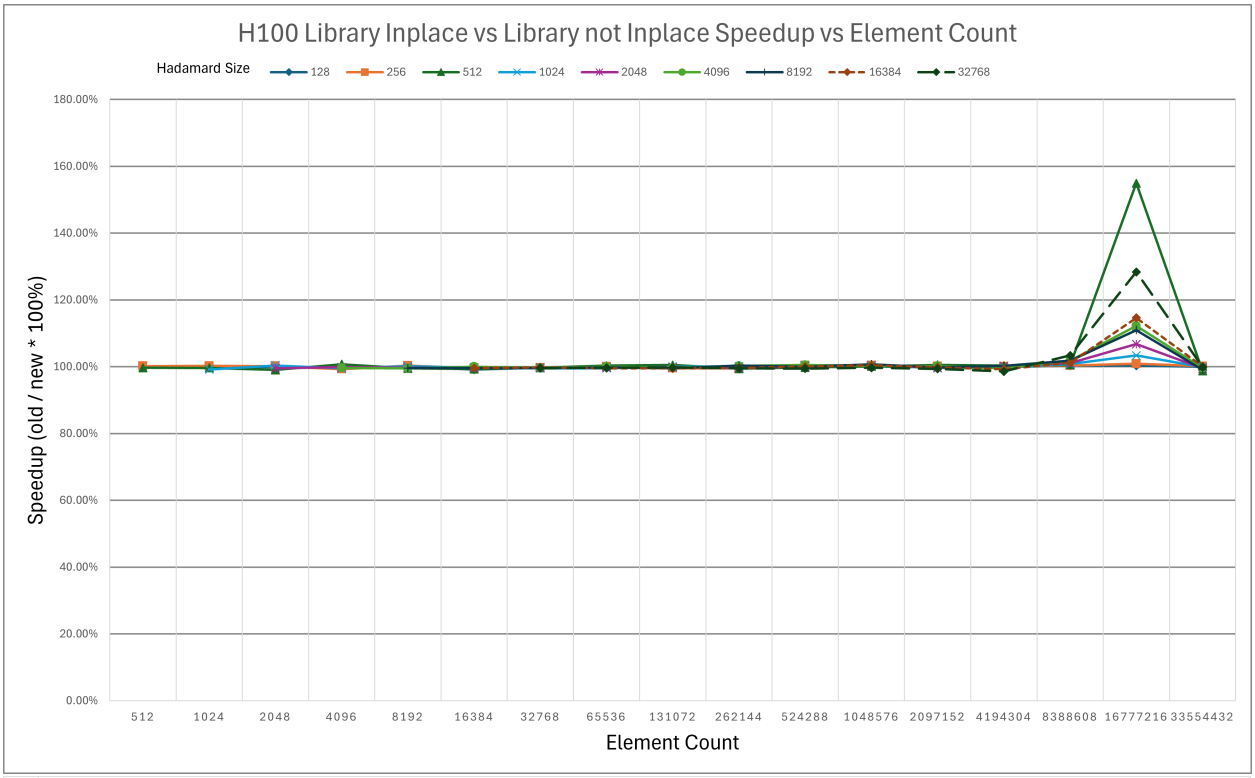
```
- at::Tensor out = torch::empty_like(x);
+ at::Tensor out = x; // torch::empty_like(x);
```

We verified that the result does not change empirically. The `fast-hadamard-transform` library now outperforms memcpy (which they used as a lower bound in their benchmarks) in some cases.



Hadamard Size \ Element Count	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432
128	100.46%	99.39%	100.00%	100.61%	100.30%	99.71%	100.42%	100.25%	100.22%	99.91%	99.70%	99.77%	100.00%	99.69%	99.60%	99.88%	99.78%
256	100.75%	100.15%	99.71%	100.15%	100.44%	100.74%	100.28%	99.60%	100.97%	100.53%	100.42%	100.41%	99.52%	99.53%	100.96%	99.98%	99.93%
512	99.83%	100.00%	100.55%	100.14%	100.13%	100.82%	100.68%	100.40%	100.13%	100.33%	100.00%	99.92%	100.56%	100.43%	134.95%	103.30%	96.38%
1024		100.68%	100.13%	99.60%	100.40%	100.65%	100.00%	100.38%	99.46%	100.37%	100.76%	100.96%	100.87%	100.62%	104.62%	99.86%	99.72%
2048			100.74%	101.44%	100.23%	99.76%	99.89%	100.35%	100.11%	100.44%	101.09%	100.93%	101.09%	100.44%	104.71%	99.77%	99.79%
4096				101.62%	100.10%	100.19%	100.00%	100.37%	100.09%	100.35%	100.25%	101.59%	100.81%	100.90%	107.01%	100.37%	99.72%
8192					100.07%	100.32%	100.38%	100.51%	100.06%	100.50%	100.55%	100.98%	101.07%	100.50%	106.44%	101.68%	100.29%
16384						100.17%	99.96%	100.08%	99.84%	100.28%	100.23%	100.30%	100.63%	100.85%	108.63%	101.33%	99.70%
32768							100.20%	100.17%	99.98%	100.13%	100.30%	100.39%	100.89%	99.72%	114.82%	100.98%	97.64%

Figure 8: Graph and table of speedup when the Dao AI Lab `fast-hadamard-transform` is modified to perform the operation in-place, measured on an A100-PCie.



Hadamard Size \ Element Count	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432
128	100.00%	100.01%	99.83%	99.80%	100.00%	99.12%	99.66%	99.69%	100.00%	100.11%	100.22%	100.04%	99.82%	99.90%	100.30%	100.31%	99.97%
256	100.20%	100.21%	100.18%	99.28%	100.33%	99.47%	99.65%	99.99%	99.53%	99.47%	100.41%	100.36%	100.13%	100.02%	100.32%	100.87%	100.11%
512	99.65%	99.65%	99.05%	100.71%	99.47%	99.31%	99.65%	100.33%	100.49%	99.41%	99.75%	100.09%	100.53%	100.30%	100.55%	154.96%	98.72%
1024		99.16%	100.35%	99.50%	100.16%	99.68%	99.54%	99.54%	100.00%	100.12%	100.40%	100.45%	99.59%	99.65%	100.77%	103.34%	99.91%
2048			99.53%	99.85%	99.68%	99.71%	99.71%	99.72%	99.84%	99.64%	100.09%	100.43%	99.67%	100.20%	101.05%	106.78%	99.49%
4096				99.63%	99.51%	99.99%	99.65%	99.99%	99.65%	100.25%	100.51%	100.19%	100.18%	99.48%	101.79%	112.23%	99.56%
8192					99.58%	99.58%	99.58%	99.76%	99.51%	100.32%	100.00%	100.74%	99.81%	100.23%	101.81%	110.92%	99.06%
16384						99.60%	99.80%	99.55%	99.51%	99.61%	100.05%	100.38%	99.80%	99.26%	101.14%	114.62%	100.07%
32768							99.53%	99.69%	99.78%	99.47%	99.42%	99.67%	99.35%	98.59%	103.35%	128.35%	99.85%

Figure 9: Graph and table of speedup when the Dao AI Lab `fast-hadamard-transform` is modified to perform the operation in-place, measured on an H100-PCie.