# Flash Attention Implemented on the Vortex GPGPU Platform

Richelle Shim, Eliot Yoon

## Introduction

Transformers dominate the current applied AI space, as they are used in natural language processing, computer vision, and audio processing. However, the attention mechanism has quadratic memory and compute cost, making them inefficient for long sequences and bottlenecking their ability to provide context. As a solution, FlashAttention is a revolutionary I/O-aware attention algorithm that aims to maximize GPU efficiency by tiling computations to fit in on-chip memory, overlapping memory transfers and computation, and fusing softmax and value accumulation. Its implementations are optimized for NVIDIA GPUs, with FlashAttention leveraging CUDA and FlashAttention-3 specifically optimized for H100s. However, because Nvidia's GPUs are proprietary and expensive, they are not accessible for academic and open research. On the other hand, Vortex is an open-source, full-stack RISC-V, general-purpose GPU platform that provides a flexible environment and makes exploring GPU microarchitecture, compiler, and runtime co-design accessible for developers and academics. This project will implement and run FlashAttention on Vortex to demonstrate how optimized compute kernels like FlashAttention can be integrated into custom GPU platforms. This project bridges algorithmic innovation in deep learning with open-source GPU research.

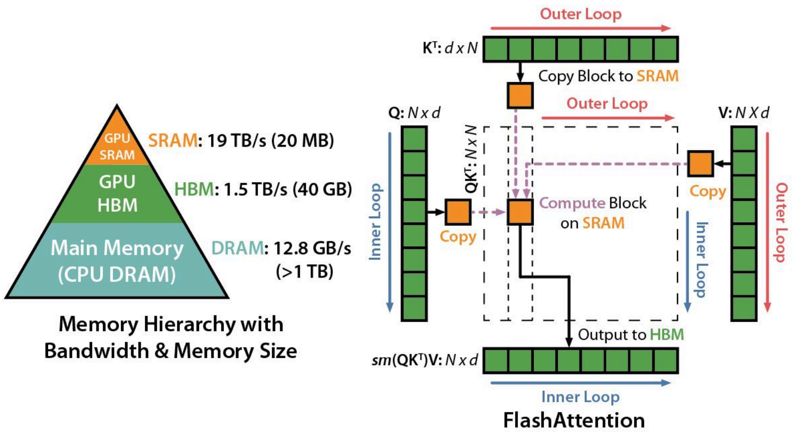
## Implementation

The standard Attention formula is

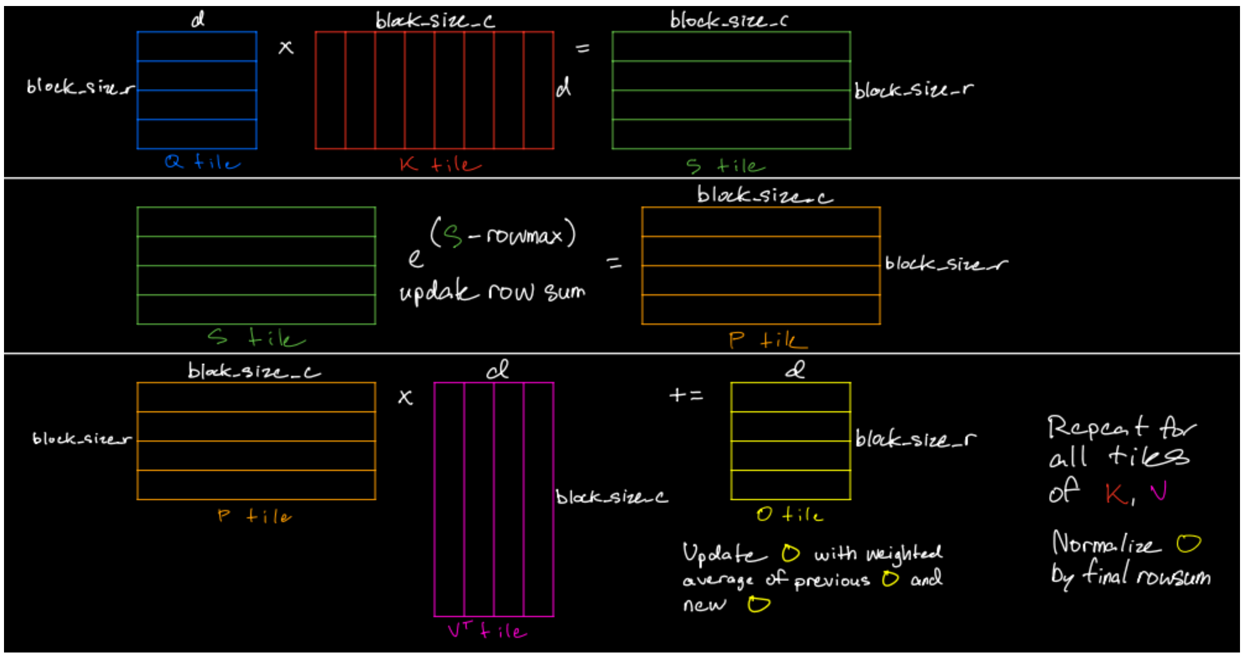
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where query (Q), key (K), and value (V) matrices are all N x $d_k$ projections of the input tokens. First, the attention score matrix S = $QK^T$ is computed, which is an N x N matrix where each row contains the similarity scores between a given query vector and all key vectors. Next, a row-wise softmax is applied to S to obtain the N x N probability matrix P. Finally, the N x $d_k$ output O = PV is computed. Attention requires materializing the full N x N score and probability matrices in global memory, causing computation to be memory-bandwidth bound and inefficient for long sequences. FlashAttention aims to improve on these inefficiencies.

FlashAttention computes the same result as standard attention, but reformulates the algorithm. Instead of computing the score matrix S at once, FlashAttention computes S tile by tile. The Q and O matrices are partitioned into tiles of size $b_r$ x d and K and V into tiles of size $b_c$ x d. Each Q tile iterates over every K and V tile to compute its corresponding O tile. In each iteration, FlashAttention multiplies the Q tile by a transposed K tile to compute a corresponding $b_r$ x $b_c$ partial S tile of attention scores. Then a streaming softmax algorithm is used to compute softmax even without access to the full row of attention scores. For each row of the S tile, which contains a subset of $b_c$ attention scores for the corresponding Q tile row, the algorithm computes the row maximum and updates the running maximum. It then applies a partial softmax to obtain the P tile. The probabilities are not normalized in this step, but the row sum (the softmax denominator) is updated. Finally, FlashAttention multiplies the P tile with the V tile and updates the O tile as a weighted average of the new result and the previous O tile. The new weight is computed as the exponential of the difference between the row max for the current S block and the updated running max. The old weight is the exponential of the difference between the old running max and updated running max. After all iterations are completed, each row of the O tile is normalized by its row sum. The main difference between Attention and FlashAttention is the tiling mechanism, and a key point of the FlashAttention algorithm is that *only* the output is written to global memory and not the scores and probabilities. Another difference is that all of the operations can be fused into a single kernel, as opposed to standard Attention which generally requires multiple kernels.



For this project, we implemented the FlashAttention forward pass on the Vortex GPU. The host program allows users to set configuration details, including which implementation of FlashAttention to use, and randomly generates input data. It loads the input data into HBM, sets the proper grid and block size to allow for tiling, and calls the kernel. The kernel program loads blocks from the input matrices into SRAM (local memory), runs the forward pass, and writes the output to HBM. We have 2 implementations of FlashAttention. The first implementation is a scalar SIMT implementation, and the second implementation is a tensor-core accelerated implementation using the Vortex Tensor Compute Unit (TCU). Both implementations follow the same high level FlashAttention algorithm.



The SIMT implementation computes FlashAttention using standard scalar and vector ALU instructions, following the algorithm above. Each thread block corresponds to a tile of Q, and each thread corresponds to a single row of the tile. The thread block iterates over every K and V tile, and the tiles are loaded into local SRAM to reduce memory latency. The scores, probabilities, outputs, and other auxiliary variables such as row maximums and sums are stored in registers. After iterating over all key-value tiles, each thread normalizes its accumulated output vector by the final softmax normalization factor and writes to the output matrix in global memory.

The TCU implementation is specialized for an 8 x 8 tile shape and enforces $d_k = 8$, $b_r = 8$, $b_c = 8$. Each thread block processes an 8 x 8 tile of the Q matrix. The Q tile is loaded locally into the on-chip SRAM and padded to match the TCU's internal tile dimensions. For each iteration over the sequence, an 8 x 8 tile of the K and V matrices is then loaded into SRAM and transposed to use in matrix multiplication. To compute the attention score, one matrix multiply-accumulate operation is executed, which computes Q x $K^T$ for the entire tile at once, so 64 dot products are computed at once to replace the per-thread scalar dot products. This results in an 8 x 8 score tile which is then stored in shared memory. Next, the streaming softmax is implemented, and the softmax computation is parallelized across threads. Each thread computes partial max and partial sums for their assigned columns of the S tile. The partial results are stored in SRAM and used to obtain the row maximum and normalization factor. The kernel uses streaming softmax by updating the running maximum and normalization factor for each row incrementally across the key-value tiles. Then, after computing the softmax probabilities for a tile, the kernel applies them to the corresponding V tile using a second tensor-core MMA operation, which computes P x V for the tile and produces a partial output block, which is accumulated into a register-resident output buffer. After all key-value tiles have been processed, the kernel performs a final normalization step using the accumulated softmax normalization factors and writes the output tile to global memory. The main difference between the SIMT implementation and the TCU implementation is the computation of Q x $K^T$ and P x V using the TCU's mma_sync to compute all 64 dot products.

## Results

Both the SIMT implementation and TCU implementation matched the accuracy of the baseline Attention implementation. On a configuration of 4 cores, 2 warps/core, 8 threads/warp, N of 64, $d_k$ of 8, and memory clock ratio of 4, the three implementations gave the following results:

| | **Instructions** | **Cycles** | **IPC** | **Memory Latency (Cycles)** |
|---|---|---|---|---|
| **Attention** | **1513613** | **2984804** | **0.507** | **2907** |
| **FlashAttention** | **634145** | **3693943** | **0.172** | **988** |
| **FlashAttention (TCU)** | **2120872** | **4794023** | **0.442** | **767** |

The results differ from those in the presentation slides due to minor modifications. The Attention program was fixed, as originally it did not report full metrics. The memory clock ratio was also increased from 1 to 4 in order to better simulate a GPU's difference in latency between global and local memory accesses.

While FlashAttention is designed to improve performance by reducing global memory traffic, the results of our implementation do not fully reflect these improvements. Although both FlashAttention implementations significantly reduce memory latency, they still result in higher execution cycles than Attention. Attention also achieves the highest IPC.

FlashAttention introduces extra control logic, synchronization, and softmax computations compared to standard Attention. Thus, FlashAttention's advantages become significant only for very large sequence lengths and head dimensions. Our experiments were done with a sequence length of 64 and head dimension of 8, which means the computation dominates execution time. The reduced memory traffic does not offset the additional computational overhead and the dcache also mitigates the latency of the global memory accesses, especially for small input matrices.

Architectural constraints of Vortex may also explain the FlashAttention performance. Each thread on Vortex is limited to 32 floating-point registers, and FlashAttention relies heavily on registers to store intermediate results. This constrains our program's maximum possible head dimension and block size, since each thread needs to store d output values and $b_c$ intermediate values. Comparatively, the original FlashAttention paper used NVIDIA GPUs, and some CUDA architectures support up to 255 registers per thread. A potential optimization we may explore in the future is to further divide up the work for a single row amongst multiple threads, allowing for larger values of d even with the register constraints.

Although the TCU implementation showed improvements on the IPC by replacing scalar-fused multiply-add operations with MMA operations, both implementations ultimately did not perform better than the standard attention implementation.

## Artifacts

Source Repository: https://github.com/eyoon1131/vortex.git

To reproduce the above results, one can run the provided *attention* and *flash* programs in the *tests/regression* folder. After setting up Vortex and configuring the *build* directory, one can run the standard *attention* program, which runs 3 kernels and outputs 3 sets of metrics, with the following:

```
CONFIGS="-DNUM_CORES=4 -DNUM_WARPS=2 -DNUM_THREADS=8 -DMEM_CLOCK_RATIO=4" ./ci/blackbox.sh
--driver=simx --app=attention --args="-n 64 -d 8" --perf=2
```

The SIMT *flash* program can be run with:

```
CONFIGS="-DNUM_CORES=4 -DNUM_WARPS=2 -DNUM_THREADS=8 -DMEM_CLOCK_RATIO=4" ./ci/blackbox.sh
--driver=simx --app=flash --args="-n 64 -d 8" --perf=2
```

The TCU *flash* program can be run by enabling the TCU and adding an argument t = 1:

```
CONFIGS="-DNUM_CORES=4 -DNUM_WARPS=2 -DNUM_THREADS=8 -DMEM_CLOCK_RATIO=4 -DEXT_TCU_ENABLE"
./ci/blackbox.sh --driver=simx --app=flash --args="-n 64 -d 8 -t 1" --perf=2
```