CS542200 Parallel Programming
Homework 4: FlashAttention

Kai-Yuan Jeng 鄭凱元
113062529
kaiyuanjeng@gapp.nthu.edu.tw

1 Implementation

This section details the implementation of the FlashAttention forward pass on a single GPU using CUDA.

1.1 FlashAttention Forward Pass

The core innovation of FlashAttention lies in *tiling* (or matrix blocking) the Query (Q), Key (K), and Value (V) matrices to effectively leverage the GPU memory hierarchy. As described in the original FlashAttention paper, the algorithm minimizes accesses to the slow High Bandwidth Memory (HBM, corresponding to CUDA Global Memory) by performing the majority of computations within the fast on-chip SRAM (corresponding to CUDA Shared Memory).

Matrix Blocking The input matrices $Q, K, V \in \mathbb{R}^{N \times d}$ are defined by the sequence length N and the embedding dimension d. We partition the sequence length N based on the block sizes defined in the following macros:

```
#define BR 128
#define BC 16
```

To fit the required tiles into the limited Shared Memory (SRAM) of a CUDA block, we must satisfy the constraint:

$$(BR + 2 \times BC) \times d \times 4 \text{ bytes} \leq \text{SRAM Capacity}$$

where BR is the block size for the Query rows, BC is the block size for Key/Value columns, and 4 represents the size of a single-precision floating-point number. Given the typical embedding size $d = 64$, and a standard SRAM limit (e.g., 48KB or 96KB depending on the architecture), we selected $BR = 128$ and $BC = 16$ to balance occupancy and register usage. The exact dynamic shared memory allocation is calculated as:

```
size_t sram_size = (BR * (d + PADDING) + BC * (d + PADDING) + BC * (d + PADDING)) *
↪  sizeof(float);
```

Note that we introduced a `PADDING` term to the stride to avoid shared memory bank conflicts during the dot-product computation.

Score Computation and Online Softmax Inside the kernel, each thread computes the attention scores for a specific row of the Query block. The partial attention scores $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T$ and the local row maximum $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij})$ are computed as follows:

```
float S_ij[BC];
float m_ij_tilde = -FLT_MAX;

for (int k = 0; k < BC; k++) {
    float score = 0.0f;
```

```
    // Compute dot product Q_i * K_j^T
    for (int t = 0; t < d; t++)
        score += sm_Q[tx * (d + PADDING) + t] * sm_K[k * (d + PADDING) + t];

    score *= (1.0f / sqrtf((float)d)); // Scaling
    S_ij[k] = score;
    m_ij_tilde = fmaxf(m_ij_tilde, score);
}
```

Updating Scaling Factors and Accumulators To ensure numerical stability without materializing the full $N \times N$ matrix, we employ the *Online Softmax* technique. We maintain three running statistics in registers for each thread:

- m_i: The maximum score seen so far for the current row.
- l_i: The running sum of exponentials (normalization factor).
- O_i: The accumulated output vector.

When processing a new block j, we first update the global maximum m_i^{new} and calculate the correction factor $\alpha = e^{m_i - m_i^{new}}$ to rescale the existing accumulator. This corresponds to the following logic:

```
// Update global max statistic
float m_i_new = fmaxf(m_i, m_ij_tilde);

// Calculate scaling factor for numerical stability
float exp_m_diff_old = expf(m_i - m_i_new);

// Calculate P_ij terms for the current block
float P_ij[BC];
float l_ij_sum_exp = 0.0f;

for (int k = 0; k < BC; k++) {
    P_ij[k] = expf(S_ij[k] - m_i_new);
    l_ij_sum_exp += P_ij[k];
}

// Update the running normalization factor l_i
float l_i_new = l_i * exp_m_diff_old + l_ij_sum_exp;
```

Finally, the output accumulator O_i is updated by rescaling the previous results and adding the contribution from the current block's Values V_j:

```
// Update O_i: rescale previous result and add new P_ij * V_j
for (int t = 0; t < d; t++) {
    float P_ij_V_j = 0.0f;
    for (int k = 0; k < BC; k++)
        P_ij_V_j += P_ij[k] * sm_V[k * (d + PADDING) + t];

    O_i[t] = O_i[t] * exp_m_diff_old + P_ij_V_j;
}

// Update loop variables for the next iteration
m_i = m_i_new;
l_i = l_i_new;
```

After iterating through all blocks of K and V, the final output is obtained by normalizing O_i with the final sum of exponentials l_i: $O_i \leftarrow O_i / l_i$.

Modification of the Original Algorithm Our implementation adapts Algorithm 1 **?** by swapping the loop order to exploit GPU parallelism. In the original algorithm, the outer loop iterates over K/V blocks (Line 5), requiring the intermediate results (O_i, ℓ_i, m_i) for each Q_i to be repeatedly loaded and stored to HBM (Lines 8 & 13), which incurs high memory traffic. In our CUDA design, we map the Q blocks to CUDA Thread Blocks (parallelizing the original inner loop over i). Each

Thread Block keeps the intermediate accumulators (O_i, ℓ_i, m_i) exclusively in Registers, iterating through all K/V blocks in the inner loop. This modification removes the need for intermediate HBM accesses entirely, performing only one final write-back per query row, which significantly reduces global memory bandwidth usage.

1.2 Parallelization Strategy and Kernel Configuration

Our implementation maps the FlashAttention algorithm to the GPU architecture by parallelizing across two dimensions: the batch size B and the sequence length N.

Grid and Block Dimensions To efficiently distribute the workload, we configure the CUDA execution grid as follows:

- **Grid Y-Dimension (Batch Parallelism):** We map the batch dimension directly to `blockIdx.y`. Since the attention mechanism is independent across different batches, this allows the GPU to process multiple sequences in parallel. In our implementation using CUDA Streams, this corresponds to `blocks_per_grid.y = B_chunk`.
- **Grid X-Dimension (Sequence Parallelism):** We partition the Query matrix Q into blocks of size B_r. The number of blocks in the x-dimension is calculated as $\lceil N/B_r \rceil$. Each CUDA block (identified by `blockIdx.x`) is responsible for computing the attention output for B_r rows of the Query matrix.
- **Thread Configuration:** We set the block dimension to `dim3(BR, 1, 1)`. Since $B_r = 128$, each CUDA block contains 128 threads. This one-to-one mapping between threads and Query rows within a block simplifies the logic: thread tx computes the attention row for the query vector $Q_{row\_offset+tx}$.

The kernel launch configuration in the code is:

```
dim3 blocks_per_grid((N + BR - 1) / BR, B_chunk);
dim3 threads_per_block(BR);
size_t sram_size = (BR * (d + PADDING) + BC * (d + PADDING) * 2) * sizeof(float);
flash_attention_kernel<<<blocks_per_grid, threads_per_block, sram_size, stream>>>(...);
```

Justification for Block Sizes and SRAM Usage The choice of block sizes $B_r = 128$ and $B_c = 16$ is critical for performance and is derived from the hardware constraints of the GPU (Shared Memory capacity and Register file):

1. **SRAM Constraints:** The algorithm requires storing one block of Q ($B_r \times d$) and blocks of K and V ($B_c \times d$ each) in Shared Memory. With $d = 64$, the memory footprint is approximately:

   $$\text{Mem} \approx (128 \times 64 + 16 \times 64 + 16 \times 64) \times 4 \text{ bytes} \approx 40 \text{ KB}$$

   This fits comfortably within the 48KB (or higher) shared memory limit per Streaming Multiprocessor (SM) on modern NVIDIA GPUs, leaving room for the `PADDING` overhead to resolve bank conflicts.
2. **Occupancy and Registers:** Using $B_r = 128$ creates 128 threads per block. Since 128 is a multiple of the warp size (32), this ensures no threads are idle within active warps. Additionally, a smaller $B_c = 16$ minimizes the register pressure for the inner loop calculations, allowing the compiler to unroll loops effectively without spilling to local memory.
3. **Tiling Ratio ($B_r \gg B_c$):** We intentionally chose B_r to be larger than B_c. Since we load the Q block once and reuse it against many K/V blocks loaded from HBM, a larger B_r maximizes data reuse of the Query matrix in SRAM. Conversely, a smaller B_c allows the inner loop to fit in registers and shared memory, maintaining high compute throughput. This optimal configuration is also evidenced from our experiment (See Section **??**).

2 Profiling Results

We profile our submitted CUDA implementation (`hw4.cu`) using the NVIDIA `nvprof` tool on a GeForce GTX 1080 GPU.

Table 1: Performance Comparison: Measured vs. Theoretical Peak

Metric	Measured (Avg)	Theoretical Peak	Utilization
Global Memory Throughput[1]	17.91 GB/s	320.32 GB/s	5.6%
Shared Memory Throughput[2]	2949.498 GB/s	4439.04 GB/s	66.4%
Achieved Occupancy	12.42%	100.00%	12.42%

[1] Sum of global load (17.372 GB/s) and store (539.05 MB/s).
[2] Sum of shared load (2931.6 GB/s) and store (17.898 GB/s).

Table 2: Detailed Profiling Metrics for `flash_attention_kernel`

Metric	Min	Max	Avg
Achieved Occupancy	0.124183	0.124200	0.124193
SM Efficiency	98.05%	99.67%	98.87%
Global Mem Load	17.168 GB/s	17.652 GB/s	17.372 GB/s
Global Mem Store	532.72 MB/s	547.75 MB/s	539.05 MB/s
Shared Mem Load	2897.2 GB/s	2978.9 GB/s	2931.6 GB/s
Shared Mem Store	17.688 GB/s	18.187 GB/s	17.898 GB/s

Theoretical Peak Calculation The theoretical peak values in Table 1 are derived from the hardware specifications of the NVIDIA GeForce GTX 1080 obtained via the `deviceQuery` tool.

- **Global Memory Bandwidth:** With a memory clock rate of 5005 MHz and a 256-bit memory bus width, the theoretical bandwidth is calculated as

$$\text{BW}_{\text{global}} = \text{Mem Clock} \times 2 \times \frac{\text{Bus Width}}{8} = 5005\ \text{MHz} \times 2 \times 32\ \text{B} \approx 320.32\ \text{GB/s},$$

  where the factor of 2 accounts for the Double Data Rate (DDR) effective transfer relative to the reported memory clock.
- **Shared Memory Bandwidth:** The GTX 1080 (Pascal architecture) contains 20 Streaming Multiprocessors (SMs). Each SM features 32 shared memory banks with a 4-byte width (32-bit), capable of processing one transaction per clock cycle. Using the maximum boost clock of 1734 MHz, the theoretical shared memory bandwidth is

$$\text{BW}_{\text{shared}} = \text{SMs} \times \text{Banks} \times \text{Width} \times \text{Clock} = 20 \times 32 \times 4\ \text{B} \times 1.734\ \text{GHz} \approx 4439.04\ \text{GB/s}.$$

The resulting kernel-level metrics are summarized in Table 2. We select the testcase `t22` with the following command to obtain the statistics.

```
srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics \
achieved_occupancy,sm_efficiency,gld_throughput,\
gst_throughput,shared_load_throughput,shared_store_throughput \
./hw4 ./testcases/t22 temp.out
```

Comparing our measured results with these theoretical peaks, our implementation achieves approximately $(17.372 + 0.53905)/320.32 \approx 5.6\%$ of the available global memory bandwidth and $(2931.6 + 17.898)/4439.04 \approx 66.4\%$ of the shared memory throughput.

Analysis of Results

- **High Shared Memory Utilization:** The exceptionally high shared memory utilization (66.4%) confirms the efficacy of our tiling strategy. The kernel aggressively reuses data within the SRAM (specifically the Query block), reducing the need to fetch data from HBM.
- **Low Global Memory Utilization as a Feature:** The low global memory utilization (5.6%) is not a bottleneck but rather a design goal of FlashAttention. By keeping intermediate accumulators (O, l, m) in registers and performing $\mathcal{O}(N^2 d)$ operations with only $\mathcal{O}(N^2/B_c)$ memory accesses, we have successfully made the kernel compute-bound rather than memory-bound.

- **Occupancy Explanation:** The achieved occupancy of $\approx 12.4\%$ is strictly limited by the Shared Memory usage. Each thread block consumes approximately 40 KB of Shared Memory. Although `deviceQuery` reports a per-block limit of 48 KB (49,152 bytes), the Pascal architecture (Compute Capability 6.1) features a physical capacity of 96 KB of Shared Memory per SM. This configuration allows exactly 2 active blocks per SM (40 KB $\times\ 2 = 80$ KB $<$ 96 KB), as a third block would exceed the SM's capacity. With 128 threads per block, this results in 256 active threads per SM. Given the hardware limit of 2048 threads per SM, the theoretical occupancy is $256/2048 = 12.5\%$, which aligns perfectly with our measured 12.42%. Despite this low occupancy, the high SM Efficiency (98.87%) indicates that the active warps are fully saturating the compute units, and latency hiding is sufficient.

3 Experiment and Analysis

This section presents a systematic evaluation of our CUDA implementation, incorporating various optimization strategies and detailed profiling analysis.

3.1 System Specification

The experiments were conducted on the **Apollo** cluster's GPU compute node. The operating environment is a virtualized instance running on a 64-bit Linux system. System specifications were retrieved using `lscpu` and `./deviceQuery`.

- **CPU:** The node is powered by an **Intel Xeon Silver 4210** processor with a base frequency of 2.20 GHz. The experimental instance is configured with **4 vCPUs**. The architecture features a 32 KiB L1d cache, 32 KiB L1i cache, and a 1 MiB L2 cache per core, along with a shared L3 cache (reported as 16 MiB in the virtualized environment).
- **GPU:** The system is equipped with an **NVIDIA GeForce GTX 1080** GPU based on the **Pascal** architecture (Compute Capability 6.1). It features **2560 CUDA cores**, 20 Streaming Multiprocessors (SMs), and **8 GB of GDDR5X** global memory. The 256-bit memory interface operates at a clock rate of 5005 MHz, yielding a theoretical bandwidth of 320.3 GB/s.
- **Software Environment:** The system runs NVIDIA Driver version 560.35.03 and **CUDA 12.6**. All code was compiled using `nvcc` with optimization flags appropriate for the hardware architecture.

3.2 Optimization Analysis

We evaluated the performance impact of each optimization step using testcase `t22`. Figure 1 summarizes the execution time reduction achieved by each technique applied cumulatively to our baseline implementation.

Baseline and Kernel Fusion The baseline version is a naive implementation of FlashAttention, with all main functionalities integrated into a single kernel (see Table 3 for profiling metrics). Initially, we observed that kernel launch overhead was significant when launching kernels within a host loop for each query row.

2D Alignment and Occupancy To address the launch overhead, we optimized the grid configuration by mapping the batch size to the grid's y-dimension. This allows a single kernel launch to process the entire batch:

```
dim3 blocks_per_grid((N + BR - 1) / BR, B);
dim3 threads_per_block(BR);
// Dynamic shared memory allocation
size_t sram_size = (BR * d + BC * d + BC * d) * sizeof(float);
flash_attention_kernel<<<blocks_per_grid, threads_per_block, sram_size>>>(...);
```

This 2D alignment strategy yielded a significant speedup of approximately **29.5%** (from 61.1s to 43.1s), demonstrating the importance of minimizing host-side overhead. Further occupancy

improvements were achieved by adding `__launch_bounds__(BR, 2)` and using `__restrict__` pointers, contributing an additional ≈8.7% reduction in execution time.

Block Size Tuning We conducted a comprehensive parameter sweep to identify the optimal block sizes (B_r, B_c). Figure 2 illustrates the total execution time across all testcases for various configurations. The results indicate that the configuration $(B_r, B_c) = (128, 16)$ offers the best performance (Total time: 41806 ms). This confirms our analysis in Section 1, where a larger B_r maximizes Query reuse in SRAM while a smaller B_c keeps register pressure manageable.

Coalesced Memory Access We optimized global memory transactions by ensuring coalesced access patterns. In the naive implementation, threads loaded data with a stride of d, leading to uncoalesced accesses:

```
// Naive: Strided access (Uncoalesced)
for (int t = tx; t < BC; t += bd) {
    for (int k = 0; k < d; k++) {
        sm_K[t * d + k] = d_K[(j + t) * d + k];
    }
}
```

We restructured the loading logic to ensure that consecutive threads access consecutive memory addresses:

```
// Optimized: Continuous access (Coalesced)
for (int i = tx; i < BC * d; i += bd) {
    const int r = i / d;
    const int c = i % d;
    sm_K[r * (d + PADDING) + c] = d_K[(j * d) + i];
}
```

This change aligns memory requests with cache lines, significantly improving global memory throughput.

Shared Memory Padding (Bank Conflict Avoidance) The most dramatic performance improvement came from resolving shared memory bank conflicts. By adding a padding term to the shared memory layout (‘d + PADDING‘), we altered the stride to avoid mapping column accesses to the same memory banks. As shown in Figure 1, this optimization reduced execution time by **67.5%** (from 38.9s to 12.7s), highlighting that bank conflicts were a major bottleneck in the dot-product computation.

CUDA Streams (Pipeline Parallelism) Finally, we exploited the independence of batches by using CUDA Streams. We divided the batch size B into 8 chunks, assigning each to a separate stream:

```
const int B_chunk = B / NUM_STREAM;
// ... loop over streams ...
cudaMemcpyAsync(..., streams[i]);
flash_attention(..., streams[i]);
cudaMemcpyAsync(..., streams[i]);
```

This allows the GPU to overlap memory transfers (H2D/D2H) with kernel execution. This technique provided a further **38.7%** improvement, bringing the execution time down to 7.75s.

4 AMD GPU Porting and Analysis

4.1 Implementation

To port our implementation from CUDA to the AMD ROCm platform (HIP), we initially performed a direct syntax translation by replacing all `cuda` prefixes with `hip`. This simple porting served as our baseline implementation for the AMD MI100 GPU. We used this baseline to establish initial performance metrics before applying architecture-specific optimizations.

Table 3: Detailed Profiling Metrics for Baseline Version at `t22`

Metric	Min	Max	Avg
Achieved Occupancy	0.044690	0.044788	0.044723
SM Efficiency	99.31%	99.79%	99.62%
Global Mem Load	44.983 GB/s	46.575 GB/s	46.302 GB/s
Global Mem Store	357.08 MB/s	369.71 MB/s	367.55 MB/s
Shared Mem Load	3035.2 GB/s	3142.5 GB/s	3124.2 GB/s
Shared Mem Store	179.93 GB/s	186.30 GB/s	185.21 GB/s

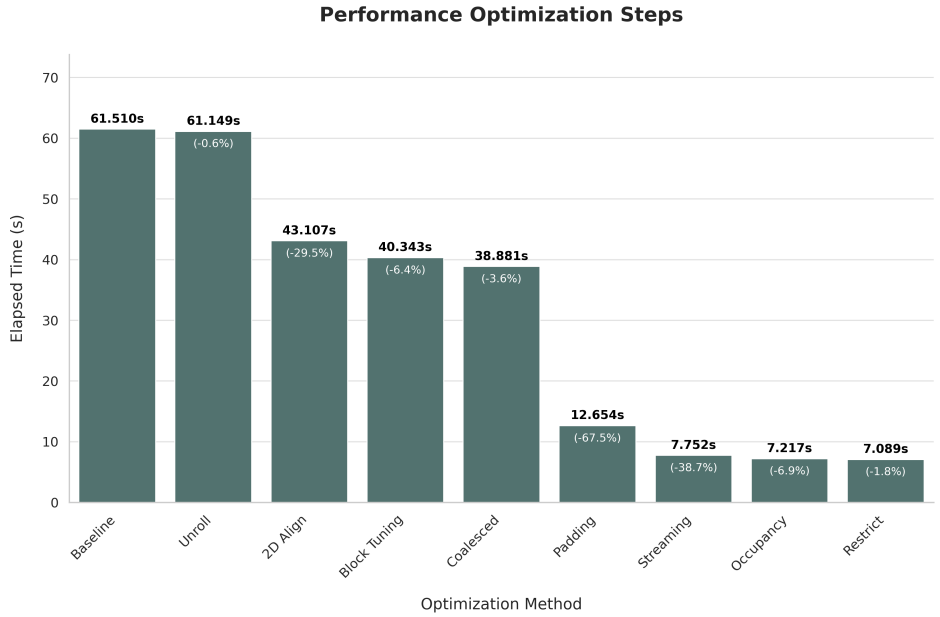

Figure 1: Step-by-step performance improvement. Significant speedups are observed from 2D Alignment (-29.5%), Padding (-67.5%), and Streaming (-38.7%).

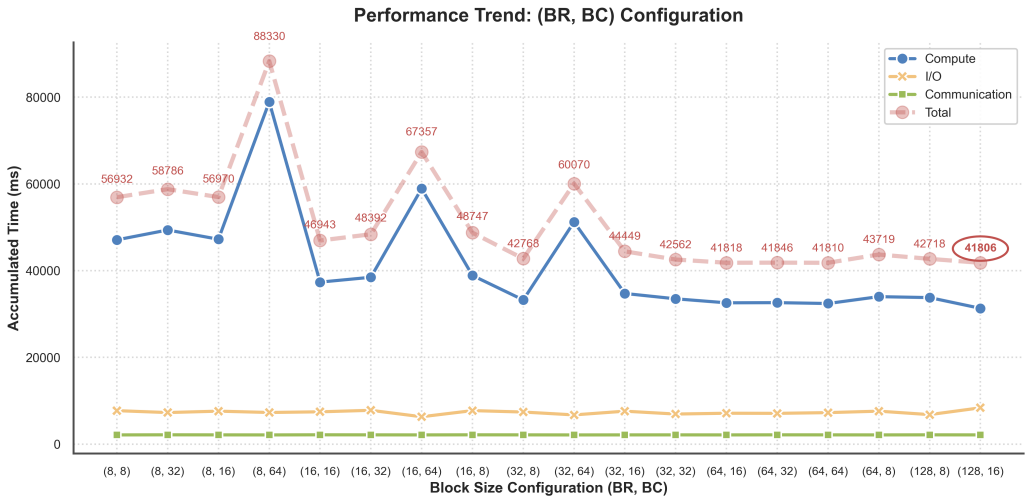

Figure 2: Performance trend across different (B_r, B_c) configurations. The configuration $(128, 16)$ yields the lowest total accumulated time across all testcases.

4.2 Experiment & Analysis

Profiling Tools and Metrics We utilized `rocprof-compute` to collect detailed hardware metrics using testcase `t22`. To characterize memory behavior and occupancy, we extracted the following metrics from the analysis report:

- **Occupancy**: Derived from *Wavefront Occupancy* (Metric 2.1.15).
- **Shared Memory Bandwidth**: Derived from *Theoretical LDS Bandwidth* (Metric 2.1.16).
- **Global Memory Load/Store**: Derived from *L2-Fabric Read BW* (Metric 2.1.22) and *L2-Fabric Write BW* (Metric 2.1.23).

The profiling commands used are as follows:

```
source /home/pp25/share/venv/bin/activate
srun -p amd -N1 -n1 --gres=gpu:1 rocprof-compute profile -n temp -- ./hw4-amd
↪  ./testcases-amd/t22 temp.out
srun -p amd -N1 -n1 --gres=gpu:1 rocprof-compute analyze -p workloads/temp/MI100/
```

Analysis of Direct Porting (Baseline HIP) The performance of the directly ported code is summarized in Table 4. The results reveal significant underutilization of the MI100's resources. The Wavefront Occupancy is merely **4.63%**, and the LDS (Shared Memory) bandwidth utilization is only **6.02%**. This indicates a major performance bottleneck.

Performance Discrepancy: GTX 1080 vs. MI100 The poor performance on the MI100 compared to the GTX 1080 can be attributed to fundamental architectural differences:

- **Warp vs. Wavefront Size:** Our NVIDIA code was optimized for a warp size of 32 and a block size of 128 threads. In contrast, the AMD MI100 (CDNA architecture) uses a wavefront size of 64. A block size of 128 creates two wavefronts, which may not be the most efficient configuration for resource allocation and scheduling on this architecture.
- **Memory Subsystem:** The MI100 has a vastly different memory hierarchy, including a much higher theoretical LDS bandwidth (≈23 TB/s vs. ≈4.4 TB/s on GTX 1080). Our unoptimized code fails to leverage this potential, as shown by the low 6.02% utilization. The padding strategy designed for NVIDIA's 32-bank shared memory may not be optimal for AMD's LDS bank structure.

LDS Constraints and Workgroup Sizing When porting to the AMD MI100 (CDNA architecture), we must re-evaluate the block size parameters B_r and B_c against the hardware's Local Data Share (LDS) limitations. The MI100 provides **64 KB of LDS per Compute Unit (CU)**, which is also the maximum allocation limit for a single workgroup. The memory requirement for our tiling strategy remains:

$$\text{Mem} = (B_r + 2 \times B_c) \times (d + \text{PADDING}) \times 4 \text{ bytes}$$

With our baseline configuration of $B_r = 128, B_c = 16$, $d = 64$, and PADDING $= 1$:

$$\text{Mem} \approx (128 + 32) \times 65 \times 4 \text{ bytes} \approx 41,600 \text{ bytes} \approx 40.6 \text{ KB}$$

This allocation (≈ 40.6 KB) safely fits within the MI100's 64 KB limit. However, attempting to increase the tile sizes, for instance to $(B_r, B_c) = (128, 64)$, would require:

$$\text{Mem} \approx (128 + 128) \times 65 \times 4 \text{ bytes} \approx 66,560 \text{ bytes} \approx 65 \text{ KB}$$

This exceeds the 64 KB per-workgroup limit, making such a configuration invalid. Therefore, our optimization space for (B_r, B_c) is strictly bounded by this 64 KB ceiling. Furthermore, unlike NVIDIA GPUs where multiple blocks can share the SM's shared memory if space permits, the MI100's LDS allocation strategy and wavefront scheduling (64 threads/wavefront) suggest that simply fitting into memory is insufficient; we must also align B_r with the native wavefront size (64) to maximize utilization.

Table 4: Profiling Metrics for Baseline HIP Version on MI100 (t22)

Metric	Avg	Peak	Pct of Peak
Wavefront Occupancy	222.11	4800.0	4.63%
Theoretical LDS Bandwidth	1389.43 GB/s	23070.72 GB/s	6.02%
L2-Fabric Read BW	0.99 GB/s	1228.8 GB/s	0.08%
L2-Fabric Write BW	13.9 GB/s	1228.8 GB/s	1.13%

Table 5: Profiling Metrics for Block Optimized HIP Version ($B_r = 64, B_c = 8$) on MI100 (t22). Parentheses indicate absolute improvement over baseline.

Metric	Avg	Peak	Pct of Peak
Wavefront Occupancy	331.48	4800.0	6.91% (+2.28%)
Theoretical LDS Bandwidth	1870.12 GB/s	23070.72 GB/s	8.11% (+2.09%)
L2-Fabric Read BW	10.69 GB/s	1228.8 GB/s	0.87% (+0.79%)
L2-Fabric Write BW	65.73 GB/s	1228.8 GB/s	5.35% (+4.22%)

AMD-Specific Optimization: Workgroup Size Tuning To address the low occupancy observed in the baseline port and better adapt to the AMD CDNA architecture, we hypothesized that the NVIDIA-optimized block size ($B_r = 128$) was suboptimal for the MI100's execution model. The MI100 executes threads in wavefronts of 64, unlike NVIDIA's warps of 32. To verify this and identify the optimal configuration, we conducted a parameter sweep experiment on the AMD platform. The results, presented in Figure 3, reveal a distinct performance trend compared to the NVIDIA results. On the MI100, the configuration $(B_r, B_c) = (64, 8)$ achieved the lowest total execution time (7916 ms). Consequently, we modified the block size macros to $B_r = 64$ and $B_c = 8$.

This adjustment is strictly AMD-specific because it aligns the workgroup size (64 threads) exactly with the hardware's fundamental execution unit (a single wavefront). This contrasts with our NVIDIA implementation, where a larger block size (128 threads = 4 warps) was preferred to maximize shared memory data reuse. On AMD, the single-wavefront workgroup strategy yields two key benefits:

1. **Higher Occupancy:** By reducing the per-workgroup resource footprint (LDS and VGPRs), the scheduler can dispatch more concurrent workgroups to each Compute Unit (CU).
2. **Reduced Synchronization Overhead:** With a 1:1 mapping between workgroup and wavefront, intra-block synchronization barriers (`__syncthreads()`) become implicit or significantly cheaper, improving instruction pipeline efficiency.

The impact of this optimization is quantified in Table 5. Compared to the baseline, the Wavefront Occupancy increased by **2.28%** (absolute), and the effective LDS bandwidth utilization improved by **2.09%**. While these absolute gains appear modest, they translate to a significant reduction in total kernel execution time, confirming that minimizing scheduling overhead is crucial on this architecture.

Comparison of Final Results: NVIDIA vs. AMD Despite the optimization, architectural distinctions remain. The NVIDIA GTX 1080 (Pascal) implementation benefited significantly from aggressive tiling (128×16) to exploit shared memory reuse. In contrast, the AMD MI100 (CDNA) favored a "leaner" configuration (64×8) that prioritizes thread-level parallelism and scheduling efficiency over data caching per block. Furthermore, while the GTX 1080 is a mature consumer architecture with drivers highly tuned for general CUDA workloads, the MI100 is a specialized data-center accelerator. Fully unlocking its ≈23 TB/s LDS bandwidth potential would likely require deeper, low-level optimizations—such as manual data swizzling to match its specific bank conflict patterns or using inline assembly for vector memory operations—which goes beyond the scope of a direct port.

5 Experience & Conclusion

This assignment provided a profound understanding of modern deep learning acceleration techniques, specifically focusing on the FlashAttention mechanism. Unlike the previous Floyd-Warshall as-

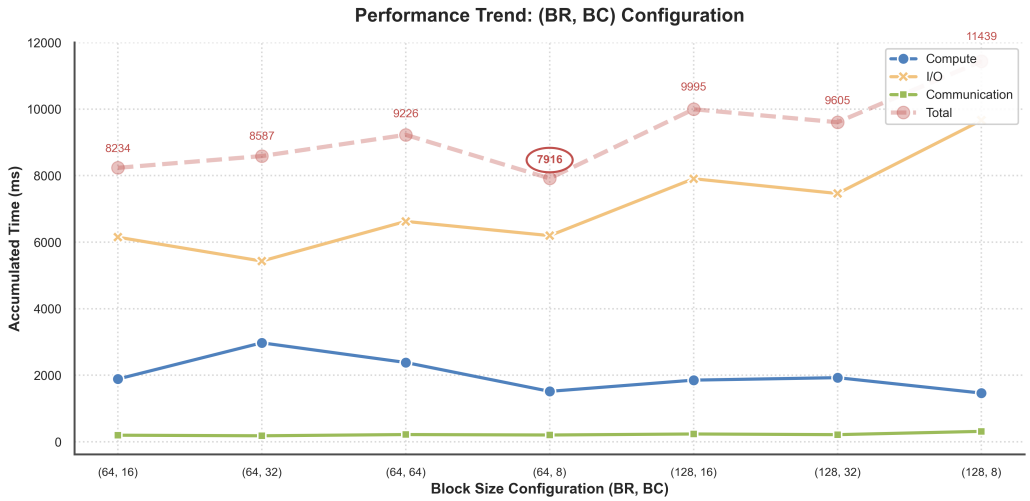

Figure 3: Performance trend on AMD MI100. Unlike NVIDIA, the smaller configuration $(B_r, B_c) = (64, 8)$ yields the best performance (7916 ms), confirming the benefit of aligning block size with the 64-thread wavefront.

signment, which was primarily bounded by global memory bandwidth due to its $\mathcal{O}(N^3)$ memory accesses, FlashAttention effectively transforms the problem into a compute-bound regime. Through the implementation of tiling and kernel fusion, we learned how to trade increased arithmetic intensity for reduced HBM access, which is the governing principle of high-performance AI kernels today.

Key Learnings The most significant takeaway is the "Compute-over-Memory" philosophy. By implementing FlashAttention, we observed firsthand how performing $\mathcal{O}(N^2)$ computations while only accessing $\mathcal{O}(N^2/B_c)$ memory allows the GPU to saturate its Tensor/CUDA cores rather than stalling on memory controllers. Another critical insight came from the architectural comparison between NVIDIA and AMD. While the syntax between CUDA and HIP is nearly identical, the hardware behaviors—specifically the wavefront size (32 vs. 64) and shared memory (LDS) constraints—dictate different optimization strategies. Our experiments showed that a block size optimal for Pascal (128 threads) resulted in poor occupancy on the MI100, reinforcing the lesson that "functional portability" (code runs) is vastly different from "performance portability" (code runs fast).

Difficulties Encountered The primary challenge in this assignment was not the code complexity itself, but the conceptual leap required to understand the *Online Softmax* algorithm and the Loop Reordering strategy. Correctly implementing the streaming calculation of the normalization factors (m_i, l_i) while ensuring numerical stability required a deep understanding of the mathematical derivation. Furthermore, mapping this logic to the GPU—specifically deciding to invert the original loop order (parallelizing over Q, looping over K, V)—was a non-trivial design choice that fundamentally determined the kernel's efficiency. Debugging the shared memory bank conflicts was also demanding. Identifying that a stride of $d = 64$ was causing serialized access and solving it with a simple `PADDING` term was a satisfying but subtle optimization that could easily be overlooked without careful profiling.

Feedback This assignment was an excellent exercise in modern accelerator programming. To further enhance the learning experience, we strongly suggest that the teaching staff provide a set of optimized reference implementations (for both CUDA and HIP) after the submission deadline. Access to "gold-standard" code would allow students to benchmark their own memory access patterns, understand the gap between their solutions and the theoretical peak, and discover advanced optimization techniques (e.g., warp-shuffle primitives or assembly-level tuning) that they might have missed.