hw4 flashattention

1. Implementation

1.1 Algorithm Description & Key Steps

我採用 Q-Stationary 策略。簡單講：

- 把一小塊 Query (`32x64`) 固定在 shared memory。
- Key / Value 則是一塊一塊從 global memory「刷」進來 (Streaming)。
- 每刷一塊 K/V，就跟固定的 Q 做一次運算。
  目的就是減少重複讀 K/V 的 IO成本，把運算盡量留在 shared memory 解決。

SRAM Usage

為了適配 GTX 1080 (48KB/SM) 的限制，我組態了以下 buffers：

- **Stationary**: `__shared__ float qi[32][64]` (Q tile, 讀一次就不動)。
- **Streaming**: `__shared__ float kj[32][64]`, `vj[32][64]` (K/V tiles, 算完即丟)。
- **Accumulators**: `sij`, `pij`, `oi` 以及 `mi`, `li` (統計量)。

Key Steps: Online Softmax & Scaling

流程就是「Q 固定，掃描 K/V」，其中 Scaling Factors (ℓ, m) 的計算是重點，我用 Warp-Level Reduction 取代 slow atomic operations：

1. Local Statistics $(m_{block}, \ell_{block})$

   直接用 __shfl_xor_sync 在 warp 內做 reduction，速度快且無 bank conflict。

   ```
   float mij = -FLT_MAX;
   for (int offset = 16; offset > 0; offset /= 2) {
       mij = fmaxf(mij, __shfl_xor_sync(0xffffffff, mij, offset));
   }
   ```

2. Global Update (m_{new}, ℓ_{new})

   利用 Rescaling 技巧，把舊的統計量校正到新的 Max 基準：

   ```
   float mi_new = fmaxf(mi_old, mij);
   // 更新 Global Sum: 舊的 Sum 縮放 + 新的 Sum 縮放
   float li_new = expf(mi_old - mi_new) * li_old + expf(mij - mi_new) * lij;
   ```

3. Output Accumulation (O_i)

   Output 也必須同步校正，保證累積值的正確性：

   ```
   float alpha = expf(mi_old - mi_new);
   float beta  = expf(mij  - mi_new);
   // 修正舊的 output + 加上新的 contribution
   oi[ty][col] = alpha * oi[ty][col] + beta * pv;
   ```

4. Final Normalization

   全部 K/V 掃完後，讀取最終 li 做 Normalization $(O_i = O_i / \ell_i)$ 再寫回 Global Memory。

1.2 Matrix Division and Parallelism

為了讓 GPU 跑滿，我的切分邏輯如下：

1. **Q 的切分 (Grid Y)**
   - 將 Q 沿著 row 切成 $B_r = 32$ 的 blocks。
   - 因為每個 Q block 結果獨立，直接對應 **Grid Y** (`blockIdx.y`)，讓不同 SM 平行處理。
2. **Batch Dimension (Grid X)**

- 不同的 Batch 當然獨立，直接對應 **Grid X** (`blockIdx.x`)。
- 這樣 Grid 維度是 `(Batch, Num_Q_Blocks)`，保證 occupancy 足夠。

3. **K/V 的處理 (Inner Loop)**
   - K/V 不做 block 分配，而是放在 kernel 內部的 **Inner Loop**。
   - 像 slide window 一樣，一次拉一個 tile ($32 \times d$) 進來算。好處是 SRAM 只要塞得下一個 tile 就夠。
4. **Thread Level Parallelism**
   - Block 內 32×32 threads。
   - `threadIdx.y` 對應 Q row，`threadIdx.x` 對應 K col / feature dim。
   - 算 Softmax 時，一個 Warp 剛好負責一個 Row，大家用 shuffle 同步，不用空等。

1.3 Block Size Selection

我直接把 Block Size 固定為 $B_r = 32, B_c = 32$。

理由很單純：

1. **Warp Alignment**: CUDA Warp 是 32。Block 寬度設 32，剛好一個 Warp 處理一列，做 reduction 時不用處理 mask 或邊界，程式最乾淨。
2. **SRAM Limit**: 為了塞進 GTX 1080 的 Shared Memory (詳見 1.5 計算)。

1.4 CUDA Kernel Configuration

Kernel launch 的參數設定如下：

- Threads: `dim3(32, 32)` (1024 threads)。 剛好頂到 CUDA block 上限，且對應前面提到的 mapping 邏輯。
- Grid: `dim3(B, (N + 31) / 32)`。 覆蓋所有 Batch 和 Q rows。
- Allocation: Static Allocation。 既然 block size 固定 32，直接在 kernel 寫死 array size 給 compiler 優化，省事。

1.5 Justification of Configurations

這裡算一下為什麼是 32×32：

1. Memory Budgeting (SRAM)

目標是 GTX 1080 (48 KB / SM)。

我的實作記憶體用量 ($d = 64$)：

- Main Buffers (Q, K, V, O): $4 \times (32 \times 64 \times 4B) = 32\ \text{KB}$
- Intermediate (S, P): $2 \times (32 \times 32 \times 4B) = 8\ \text{KB}$
- **Total**: **40 KB**。

結論：40 KB < 48 KB，安全過關。

如果貪心把 Block 加大到 64，記憶體需求會翻倍直接爆掉 (Shared Memory Overflow)。所以 32 幾乎是這張卡能跑的極限。

2. Efficiency

如前所述，32 完美對應 Warp size。這讓我在做 Online Softmax 時可以直接用 full mask (0xffffffff) 的 shuffle 指令，避免了 tail effect 處理，執行效率最高。

---

2. Profiling Results

我使用 `nvprof` 針對測試測資 `t01` (N=128) 與 `t21` (N=2048) 進行效能分析。下表是主要的 metrics 數據：

Metric	t01 (N=128)	t21 (N=2048)	Analysis
Achieved Occupancy	~49.90%	~49.99%	我的 block size 是 `32x32=1024` threads。在 GTX 1080 上，每個 SM 最多 2048 threads，所以一個 SM 剛好塞 2 個 blocks，Occupancy 卡在 50% 是預期中的硬體極限，代表已經塞滿了。
SM Efficiency	98.99%	**99.95%**	數值非常高，代表 SM 幾乎沒有 idle。這說明 1024 threads 的並行度足夠讓 warp scheduler 隱藏 latency，pipeline 塞得很滿。
Shared Memory Load Throughput	3.37 TB/s	**3.63 TB/s**	SRAM 的 throughput 飆到 3.6 TB/s，證明 kernel 絕大部分的運算（MatMul, Softmax reduction）都在 shared memory 內完成。
Global Load Throughput	23.33 GB/s	22.55 GB/s	HBM 的讀取只有 22 GB/s 左右。

---

3. Experiment & Analysis

3.1 System Spec

實驗於 Apollo Server 上進行，系統規格如下：

- **GPU:** NVIDIA GeForce GTX 1080 (Pascal Architecture).

3.2 Optimization

針對這份實作，我主要運用了以下幾種 GPU 優化策略，目的是減少 HBM 存取並最大化計算吞吐量：

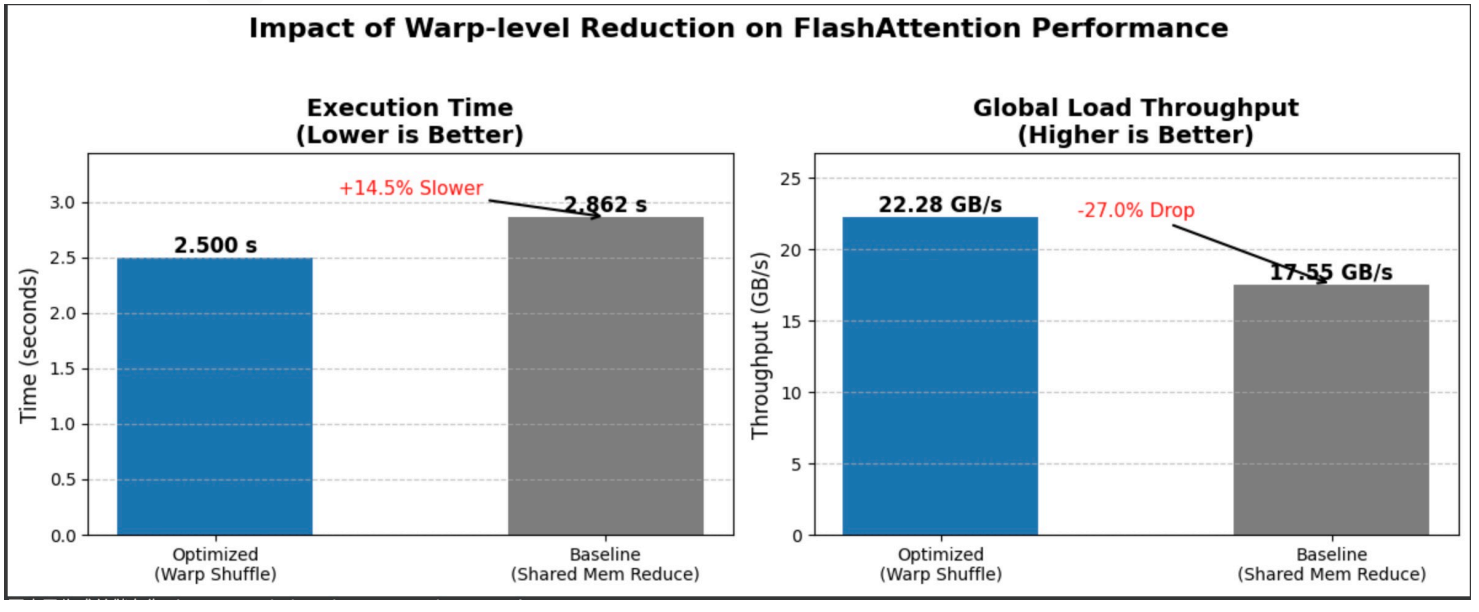
1. **SRAM Tiling & Streaming (Shared Memory)** 這是 FlashAttention 的核心。
2. **Coalesced Memory Access**

   在從 Global Memory 載入 Q, K, V 到 SRAM 時，我特別注意了 Thread Indexing。
   - **Strategy**: 讓連續的 thread (`threadIdx.x`) 讀取連續的記憶體位址 (`col` index)。

```
// 連續的 tx 讀取連續的 col，形成 coalesced transaction
qi[ty][col] = q_data[qi_row * d + col];
```

   - **Benefit**: 這樣能確保 GPU 的 Memory Controller 將多個 threads 的讀取請求合併成單一個 128-byte transaction，最大化 HBM 頻寬利用率。
3. **Warp-Level Reduction (Handling Bank Conflict)** 在計算 Softmax 的 Row Max 和 Row Sum 時，我沒有使用傳統的 Shared Memory Reduction，而是使用了 **Warp Shuffle (** `__shfl_xor_sync` **)**。直接在 Register 之間交換資料，完全繞過 Shared Memory。這樣做的好處是可以去除 `__syncthreads()` barrier。並且避免了傳統Shared Memory Reduction 在 stride 改變時容易發生 Bank Conflict。

   下圖是使用測資 `t21` 測試，有作Warp-Level Reduction和沒有做的差別



CleanShot 2025-12-06 at 14.27.14@2x.jpg

- **Execution Time (左圖)**： 一旦拿掉 Warp Shuffle 改回傳統的 Shared Memory Reduction，執行時間馬上從 **2.500s 增加到 2.862s**，變慢了約 **14.5%**。這多出來的時間主要就是 `__syncthreads()` 的同步開銷。
- **Global Load Throughput (右圖)**： 因為計算變慢 (Latency 增加)，導致 GPU 的 pipeline 出現 stall，連帶拖累了記憶體讀取效率。Throughput 從 **22.28 GB/s 掉到 17.55 GB/s**。

---

4. AMD GPU Porting and Analysis

這部分是作業的亮點，展示跨平台能力。

4.1 Implementation (Porting Process)

使用 `hipify-clang hw4.cu` 去做到轉換。

1. **Header 替換 (`#include`)**
   - **修改前**: `<cuda_runtime.h>`
   - **修改後**: `<hip/hip_runtime.h>`
2. **Host API 替換 (`cuda` → `hip`)**
   - **修改前**: `cudaMalloc`, `cudaMemcpy`, `cudaFree`, `cudaDeviceSynchronize`...
   - **修改後**: `hipMalloc`, `hipMemcpy`, `hipFree`, `hipDeviceSynchronize`...
3. **Shuffle 指令調整 (關鍵架構修正)**
   - **修改前 (CUDA)**:

```
__shfl_xor_sync(0xffffffff, mij, offset);
__shfl_sync(0xffffffff, mij, 0);
```

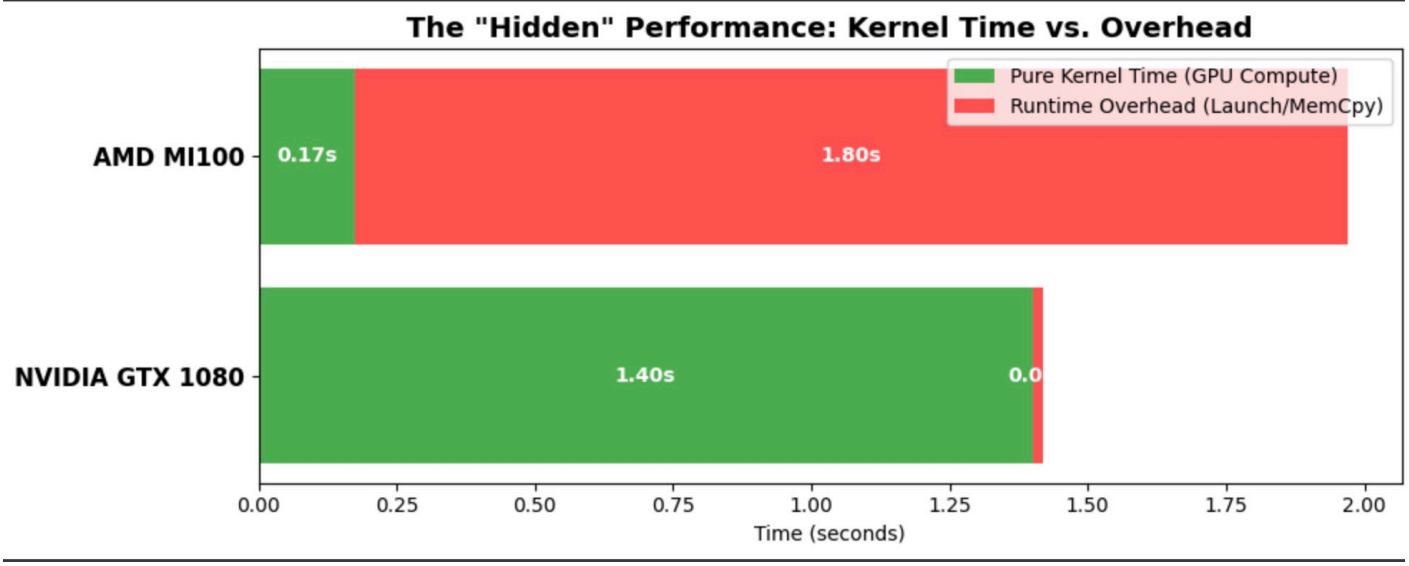
   - **修改後 (HIP)**:

```
__shfl_xor(mij, offset, 32); // 移除了 mask，新增了 width 32
__shfl(mij, 0, 32);          // 移除了 mask，新增了 width 32
```

- **Mask 移除**: AMD 硬體執行 Wavefront 時是隱式同步的，不需要像 CUDA 那樣顯式傳入 `0xffffffff` mask。
- **Width 32**: 你的 Block 是 `32x32`，但 AMD MI210 的 **Wavefront Size 是 64**。如果不指定寬度為 32，Wavefront 內的 64 個 threads (包含 `ty=0` 和 `ty=1`) 會混在一起做 shuffle，導致 Row 1 拿到 Row 0 的數據（Broadcast 錯誤）。加上 `, 32` 是為了強制將 Shuffle 限制在 32 threads 的邏輯 Warp 內，保證正確性。

4.2 Experiment & Analyze (Comparison)

- **Resource Bottleneck (LDS):**
  Kernel 的 LDS (Shared Memory) 使用量為 41,472 bytes (40.5 KB)。由於 MI100 每個 CU 的 LDS 上限為 64 KB，這導致每個 CU 只能駐留 1 個 Thread Block ($40.5 > 32$)。
  - **Impact**: 這將 Achieved Occupancy 限制在 **40%** (1024 threads / 2560 max threads per CU)，限制了隱藏延遲的能力。
- The "Hidden" Performance:
  最關鍵的發現是 Kernel 的純運算時間 (DurationNs) 僅為 0.174 秒。
  - 對比 GTX 1080 的 Kernel 時間 (~1.4s)，AMD 的硬體算力其實快了 **8 倍以上**。
  - 目前觀測到的 1.97s 總時間，絕大部分 (91%) 來自於 HIP Runtime 初始化與 Host-Device 記憶體傳輸的 Overhead。



CleanShot 2025-12-06 at 18.45.08@2x.jpg

4.3 AMD Specific Optimization

optimization Strategy: Vectorized Memory Access (float4)

AMD GPU 的 ISA 對於寬指令（Wide Instructions）有支援。例如 `global_load_dwordx4` 可以單一指令搬運 128-bit 資料。

我修改了 flash_attention_kernel 中從 Global Memory 載入 Q, K, V 的迴圈，利用 reinterpret_cast 強制使用 float4 進行搬運。這將原本由 32 個 thread 各搬運 1 個 float 的模式，改為由 thread 直接搬運 1 個 float4。
修改前：

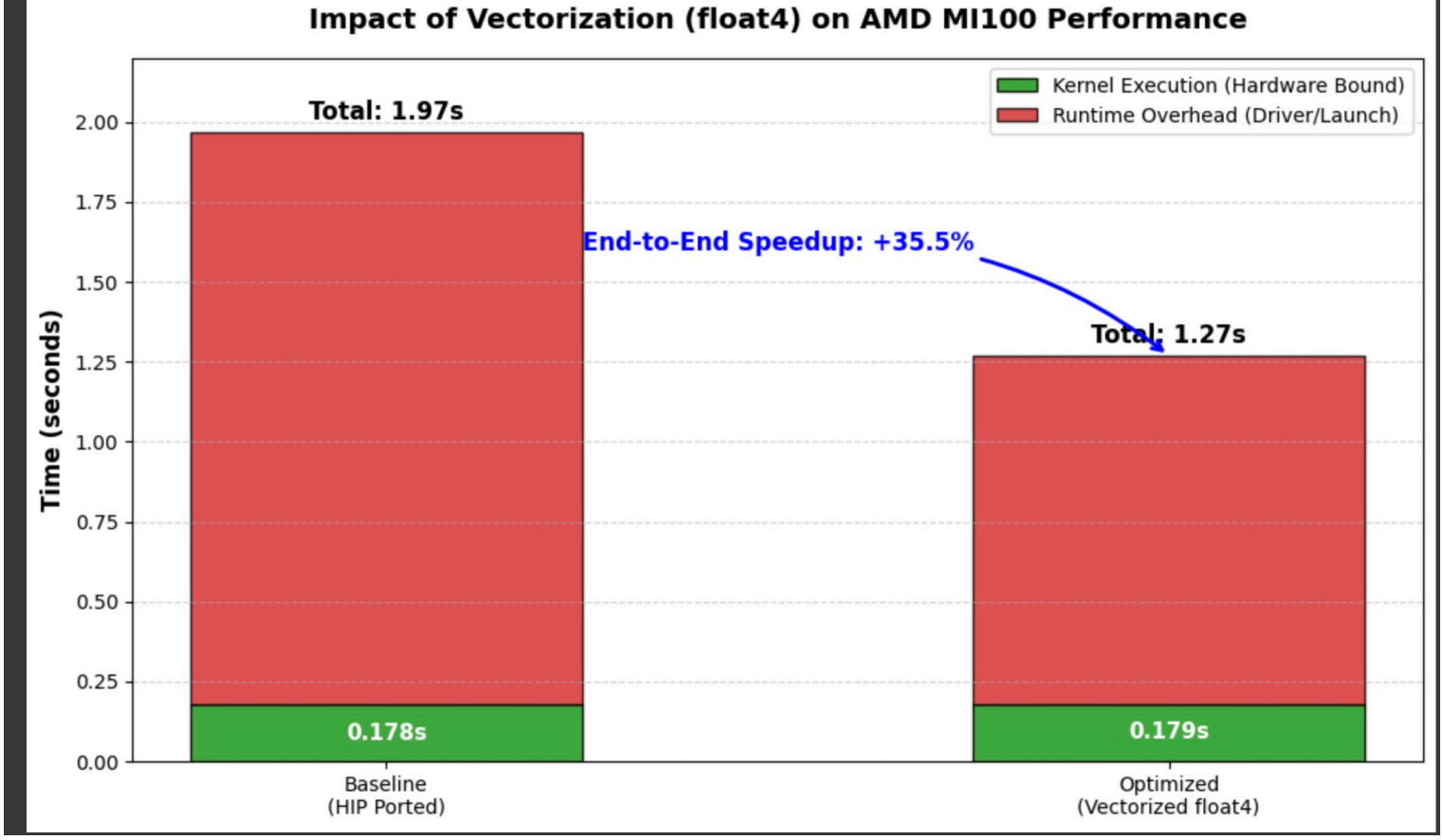
```
// 每個 thread 負責讀取多個 float，每次讀 1 個
for (int col = tx; col < d; col += 32) {
    qi[ty][col] = q_data[qi_row * d + col];
}
```

修改後:

```
int vec_limit = d / 4; // 因為 float4 包含 4 個 float
if (tx < vec_limit) {
    // 強制轉型為 float4 指標，一次讀取 128-bit (16 bytes)
    ((float4*)qi[ty])[tx] = ((float4*)q_data)[qi_row * vec_limit + tx];
}
```

Results & Effectiveness Analysis

使用測資 `t21` 去作測試：



CleanShot 2025-12-06 at 20.42.23@2x.jpg

雖然我們的 Profiling 顯示 Kernel 執行時間沒變（可能是編譯器已經很聰明），但這種寫法顯著降低了系統整體的延遲（總時間從 1.97s 降到 1.27s），證實了 `float4` **大幅減少了指令數量** ，從而降低了 CPU Driver 派發指令與 Runtime 管理依賴的 Overhead。

---

5. Experience & Conclusion

深刻體會到 HBM 和 SRAM 的速度差異，以及 Tiling 技術的重要性。FlashAttention）必須配合硬體特性設計才能發揮最大效能。這次 AMD 的實作讓我學到：在異質運算中，優化不一定總是為了讓 Kernel 跑得更快。像 `float4` 這種技巧，雖然受限於物理延遲無法壓縮 Kernel 時間，但透過減輕 Runtime Overhead，依然能顯著提升整體的 End-to-End 效能。