hw4 FlashAttention

ID: 112062520 / Name: 戴維恩

Implementation

這次的 FlashAttention 實作，我使用的 thread dimension 如下所示：

```
// B is the data batch_size
// BS is the FlashAttention block size
int BS = 32;
dim3 grid_dim(B);
dim3 block_dim(32, BS);
```

如上所示，一個 grid 會處理一個 batch 的資料，若一個 batch 的 `Q` / `K` / `V` 的維度皆為 `N * d` ，且給定的 `block_size` 為 `32` ，那麼總共就會有 `TS = ceil(N / 32)` 這麼多個 block 需要處理。在我的實作中，不管 row 還是 column 的 `block_size (BS)` 皆為 `32` 。

在 `forward_kernel` 中，若 `threadIdx.y = 0` ，就會做每個 block index 為 `0` 的那個 row，共做 `TS` 次；若 `threadIdx.y = 1` 則做 index 為 `1` 的那個 row ，以此類推，每個 `threadIdx.y` 都會做 `TS` 次，每次做一個 row。 `forward_kernel` 大致上的結構如下：

```
// in forward kernel function
int ty = threadIdx.y;
int TS = ceil((float)N / BS);

// for each block in K
for (int k = 0; k < TS; k++) {

    // for each block in Q
    for (int q = 0; q < TS; q++) {
        // if ty = 0, then calculate row 0 in this block
        // if ty = 1, then calculate row 1 in this block
        // ....

        // for each column in K
        for (int c = 0; c < BS; c++) {
            // pseudo code
            do qk = q*k.
            store result in array S.
        }
        find the max value 'mij' in the resulting row ty.

        // exp(qk)
        for (int c = 0; c < BS; c++) {
            do qk = exp(qk - mij).
        }
```

```
            calculate the sum 'lij' of this resulting row ty.

            // qkv
            for (int c = 0; c < BS; c++) {
                do qkv = qk * v.
            }

            // calculate the global max value 'm_new' of this row.
            m_new = max(m_old, mij).
            // calculate the global sum value 'l_new' of this row.
            l_new = (exp(m_old - m_new) * l_old) + (exp(mij - m_new) * lij).

            use m_old, m_new, l_old, l_new, mij to calculate the output.

            __syncthreads();
        }
        __syncthreads();
    }
```

接下來會進一步解釋詳細的實作，首先是 shared memory 的使用，總共會有四個陣列需要儲存：`qmem` / `kmem` / `vmem` / `smem_s`。前三者的大小皆為 `BS * d`，分別用來儲存 `Q` / `K` / `V` 的原始資料；`smem_s` 則是用來儲存 `qk` 的結果，大小為 `BS * BS`。

每一輪在讀取 `K` 原始資料時，與處理計算的方式一樣，`threadIdx.y = 0` 會去讀 column `0`，`threadIdx.y = 1` 讀 column `1`，以此類推直到 BS 個 column 都被讀取完畢：

```
for (int k = 0; k < TS; k++) {

    // block offset
    int block_off = batch_off + (block_size * k);
    kmem[ty * d + tx] = K[block_off + ty * d + tx];
    vmem[ty * d + tx] = V[block_off + ty * d + tx];
    if (d == 64) {
        kmem[ty * d + tx + 32] = K[block_off + (ty * d + tx + 32)];
        vmem[ty * d + tx + 32] = V[block_off + (ty * d + tx + 32)];
    }
    __syncthreads();
    // do the rest of the calculation
    // ...
}
```

而一個 column 中每個 element 會交給不同 `threadIdx.x` 去讀取，這裡的 `blockDim.x` 固定為 `32`，所以一次可以讀取 `32` 個 element。不過因為測資 `d` 的最大值是 `64`，因此要考慮到這個情況下每個 thread 需要讀取兩個 element。

讀取 `Q` 的作法與上述類似，也是每個 `threadIdx.y` 會讀取一個 row，row 裡的每個 element 交由不同的 `threadIdx.x` 去讀取：

```
for (int k = 0; k < TS; k++) {
    // read K, V ...

    for (int q = 0; q < TS; q++) {
```

```
        int block_off_q = batch_off + (block_size * q);
        qmem[ty * d + tx] = Q[block_off_q + ty * d + tx];
        if (d == 64) {
            qmem[ty * d + tx + 32] = Q[block_off_q + ty * d + tx + 32];
        }
        __syncthreads();
        // do the rest of the calculation
        // ...
    }
}
```

接著進到計算 `qk` 的部分，每次由一個 `Q` 的 row 跟一個 `K` 的 column 去計算；正常情況下，每個 thread 會做一次乘法，若 `d = 64` 的情況，則每個 thread 做兩個 element 的乘法：

```
for (int k = 0; k < TS; k++) {
    // read K, V
    for (int q = 0; q < TS; q++) {
        // read Q
        for (int c = 0; c < BS; c++) {
            float qk = 0;
            qk += qmem[ty * d + tx] * kmem[c * d + tx];
            if (d == 64) {
                qk += qmem[ty * d + tx + 32] * kmem[c * d + tx + 32];
            }

            qk = warpReduce(qk);

            if (tx == 0) {
                qk *= softmax_scale;
                smem_s[ty * BS + c] = qk;
                mij = max(mij, qk);
            }
        }
        mij = __shfl_sync(0xffffffff, mij, 0);
        // do the rest of the calculation
        // ...
    }
}
```

而這裡每個 element 的計算交給不同 thread 來做，我們最後需要的是一整排的結果加總，因此需要在 thread 之間做加總。這裡用 `warpReduce` 的方法加總所有 y 座標相同的 thread 的 `qk` 值到 `threadIdx.x = 0` 這根 thread，並且可以順便去計算最大值 `mij`。當這一排最終的 `mij` 被計算出來之後，可以使用 `__shfl_sync` 分享這個資訊給其他 y 座標相同的 thread。

接著用 `mij` 值去計算 softmax 的 exponential，這邊只有 x 座標為 `0` 的 thread 才會去計算；並且把計算完的值加總，便可以得到 `lij`。由於只有 x 座標為 `0` 的 thread 有計算這個資訊，因此要用 `__shfl_sync` 去共享資訊：

```
for (int k = 0; k < TS; k++) {
    // read K, V
```

```
    for (int q = 0; q < TS; q++) {
        // read Q
        // calculate qk, mij
        // ...
        if (tx == 0) {
            #pragma unroll 16
            for (int c = 0; c < BS; c++) {
                smem_s[ty * BS + c] = __expf(smem_s[ty * BS + c] - mij);
                lij += smem_s[ty * BS + c];
            }
        }

        lij = __shfl_sync(0xffffffff, lij, 0);
        // do the rest of the calculation
        // ...
    }
}
```

最後就是計算 `qkv` ，以及用 `mij` / `lij` 處理最終輸出答案的部分。首先先計算 `qkv` 的原始值，這裡每個 thread 最多做兩排計算，取決於 `d` 值的大小：

```
for (int k = 0; k < TS; k++) {
    // read K, V
    for (int q = 0; q < TS; q++) {
        // read Q
        // calculate exp(qk), mij, lij
        // ...
        float qkv = 0;
        float qkv_2 = 0;
        for (int c = 0; c < BS; c++) {
            qkv += smem_s[ty * BS + c] * vmem[c * d + tx];
            if (d == 64) {
                qkv_2 += smem_s[ty * BS + c] * vmem[c * d + tx + 32];
            }
        }
        // do the rest
        // ...
    }
}
```

接著計算全域的最大值及加總，每個 row 的全域最大值及加總會存在一個 global memory 的陣列 `m` / `l` 裡：

```
for (int k = 0; k < TS; k++) {
    // read K, V
    for (int q = 0; q < TS; q++) {
        // read Q
        // calculate qkv
        // ...
        float m_old = m[lm_off + q * BS + ty];
        float l_old = l[lm_off + q * BS + ty];
```

```
            float m_new = max(m_old, mij);
            float l_new = (__expf(m_old - m_new) * l_old) + (__expf(mij - m_new) * l

            if (tx == 0) {
                m[lm_off + q * BS + ty] = m_new;
                l[lm_off + q * BS + ty] = l_new;
            }
            // ...
        }
    }
```

最後用到目前為止所有算出來的資訊，計算最終的 output 值，計算方式參照 FlashAttention 公式：

```
O[block_off_q + ty * d + tx] =
    (1 / l_new) *
    (
        (l_old * __expf(m_old - m_new) * O[block_off_q + ty * d + tx]) +
        (__expf(mij - m_new) * qkv)
    );
```

Profiling Results

```
pp24s057@apollo-login:~/hw4$ srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics achieved_occupancy,sm_efficiency,gld_throughput,gst_thr
oughput,shared_load_throughput,shared_store_throughput ./hw4 testcases/t25 t25.out
==1817351== NVPROF is profiling process 1817351, command: ./hw4 testcases/t25 t25.out
==1817351== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==1817351== Profiling application: ./hw4 testcases/t25 t25.out
==1817351== Profiling result:
==1817351== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: forward_kernel(float const *, float const *, float const *, int, int, int, int, float, float*, float*, float*)
          1                        achieved_occupancy                             Achieved Occupancy    0.500000    0.500000    0.500000
          1                             sm_efficiency                       Multiprocessor Activity      75.02%      75.02%      75.02%
          1                            gld_throughput                        Global Load Throughput  15.006GB/s  15.006GB/s  15.006GB/s
          1                            gst_throughput                       Global Store Throughput  8.9756GB/s  8.9756GB/s  8.9756GB/s
          1                     shared_load_throughput                 Shared Memory Load Throughput  957.40GB/s  957.40GB/s  957.40GB/s
          1                    shared_store_throughput                Shared Memory Store Throughput  388.99GB/s  388.99GB/s  388.99GB/s
```

Experiment & Analysis

System Spec

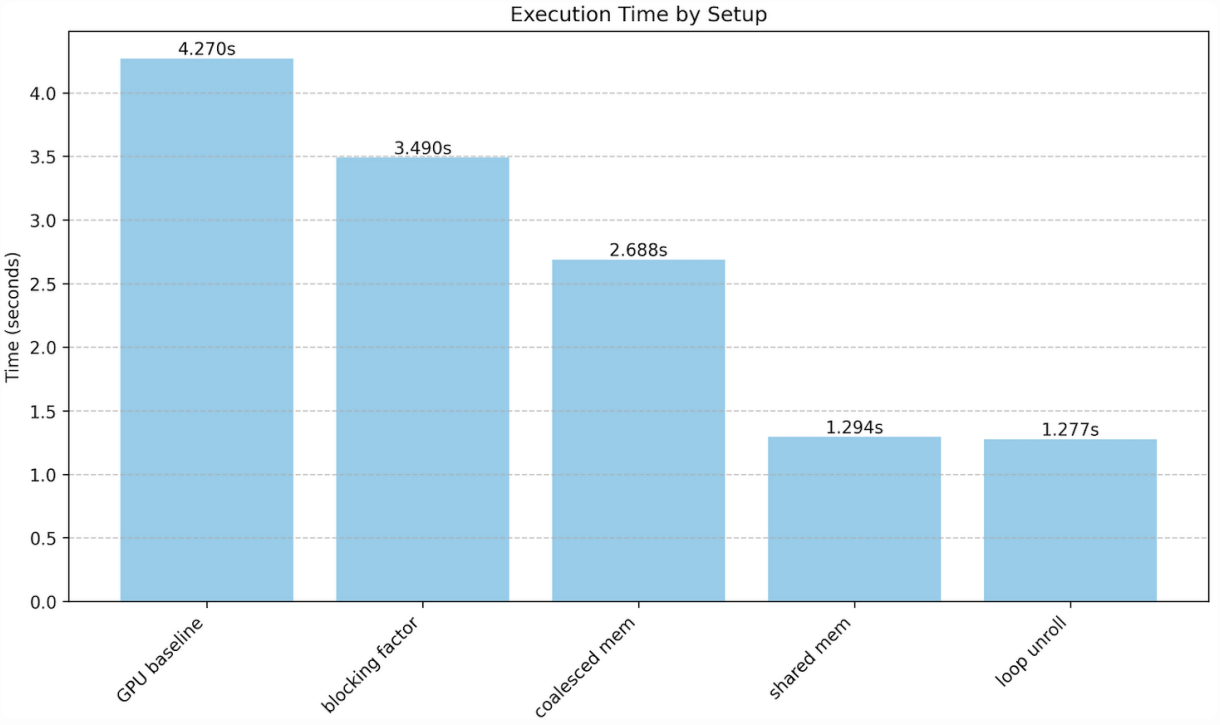
實驗使用課程提供的 `apollo GPU` 伺服器，全部使用助教提供的測資 `t25` 做數據的紀錄，並使用 `nvprof` 或是 `CUDA event` 這兩種方法紀錄程式執行時間。

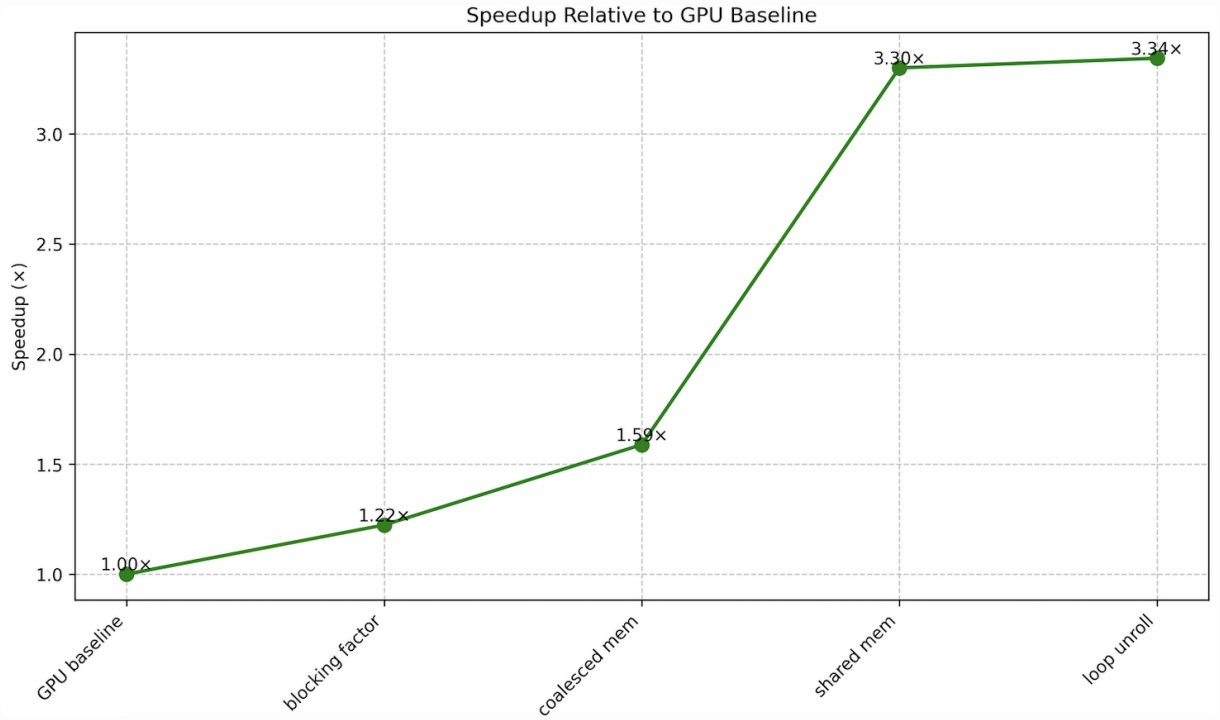
Optimization

優化的部分，我總共實作五個不同的版本，如下所示：

- GPU baseline：完全沒有實作任何優化的 GPU 版本
- blocking factor：測試 `BS` 使用最大值 `32` 的實驗結果
- coalesced memory：實驗 row major 讀取與 column major 讀取的差異
- shared memory：實驗使用 shared memory 存取 `Q` / `K` / `V` 資料的差異

- loop unroll：在部分 for-loop 使用 unroll 技巧





由實驗結果可以看見，實作了所有優化的版本，相較於一開始的 GPU baseline，可以加速達到接近 3.5 倍的效果。

Others

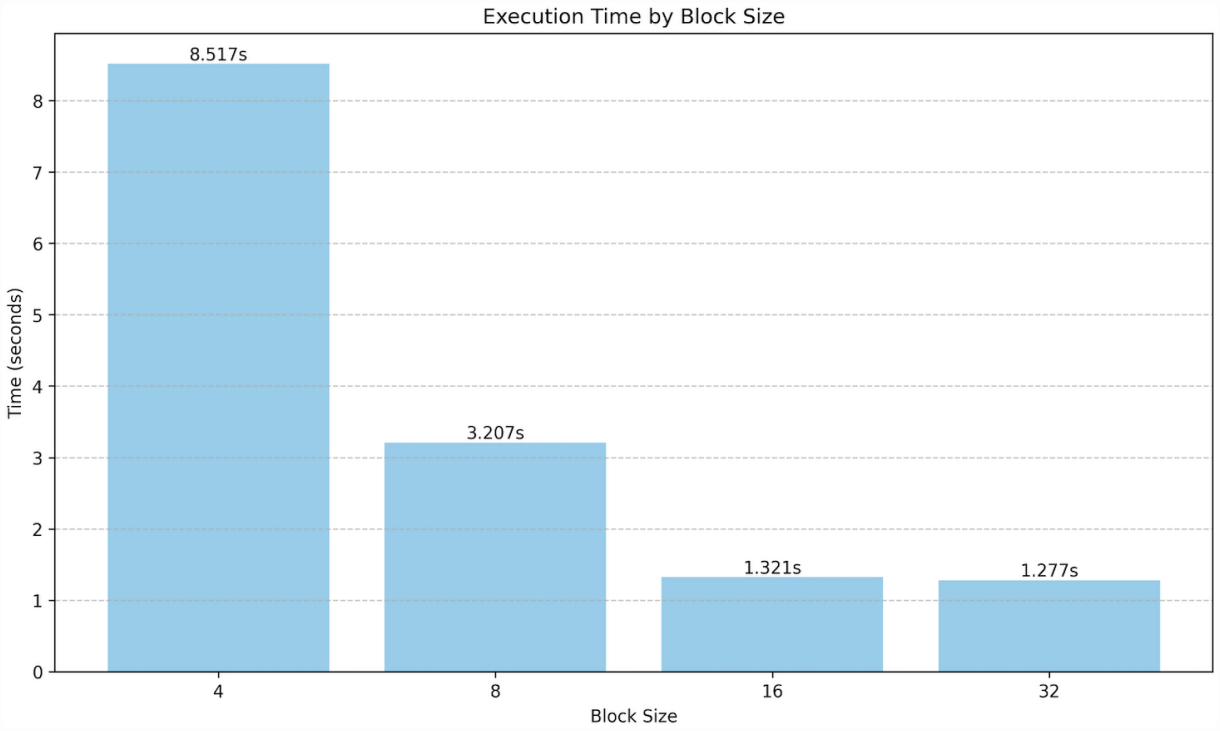
我總共實作了三個其他實驗，分別為：

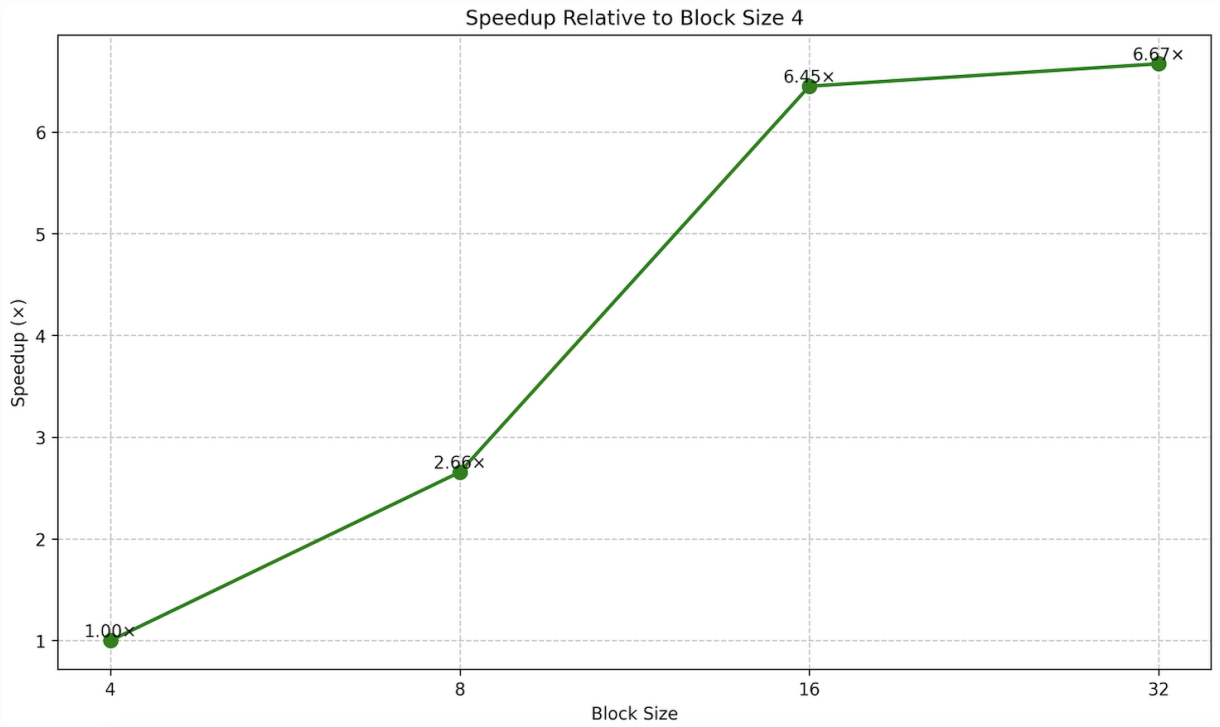
- `BS` 大小對速度的影響 (Blocking Factor)
- 不同 `BS` 對記憶體頻寬的影響 (Memory Bandwidth)

- 不同測資執行時間分佈 (Time Distribution)

Blocking Factor

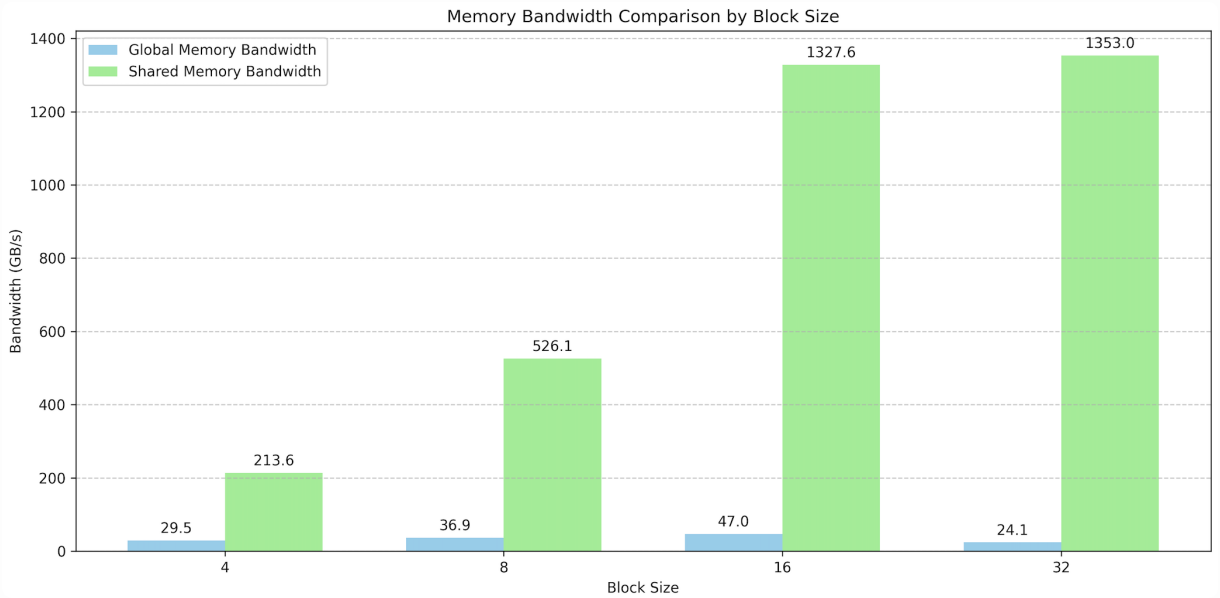
首先對於 `BS` 大小不同的執行結果顯示，`BS` 越大，程式的計算速度越快；其中當 `BS` 從 `4` 調整到 `8` 時的差距最大，而 `16` 到 `32` 的差異則不多。由調整 `BS` 的方式可以讓程式速度提升接近 7 倍。
這個結果顯示 `BS` 調大雖然可以增加 GPU 的計算量（thread 數量變多），藉此提高程式效能，但過多的 thread 可能會造成讀取資料上的 bank conflict，抵銷掉部分優化效果。





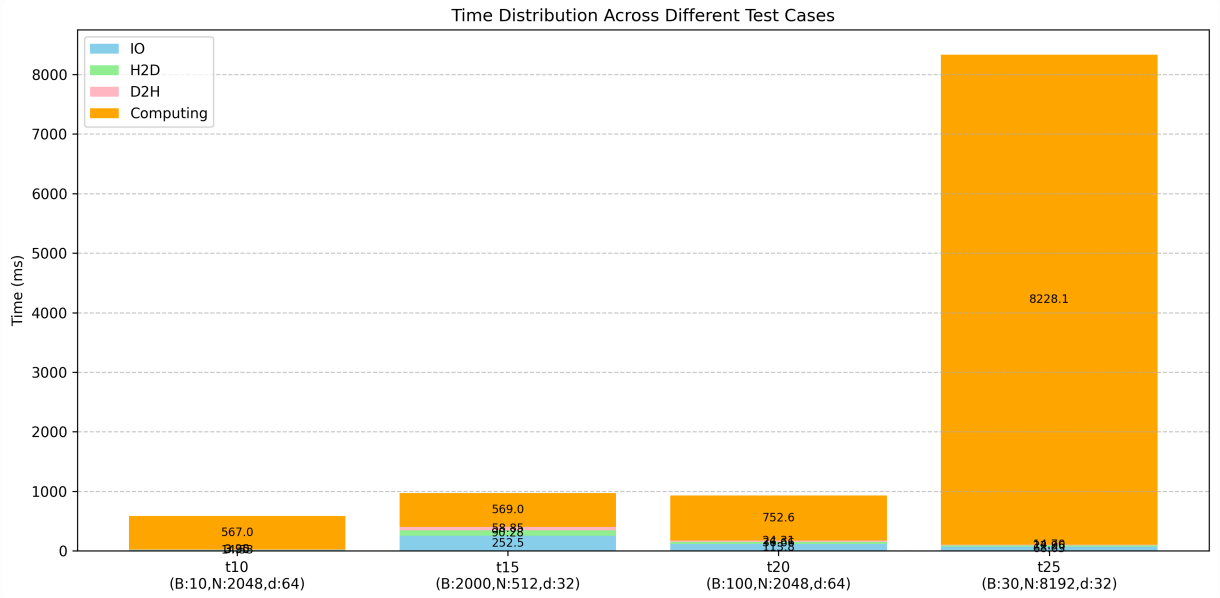
Memory Bandwidth

接著是記憶體頻寬的實驗，我針對 `BS` 為 `4` / `8` / `16` / `32` 去進行實驗，測試每種設定的 `global memory` 以及 `shared memory` 的使用頻寬。
由實驗結果可以明顯看到，`BS` 越大，總記憶體頻寬變越大；記憶體頻寬跟讀寫總量有正相關，當 thread 數量越多，會執行的讀取以及寫入總量變越大，因此這個實驗結果算是意料之中。其中可以看到 `BS` 對 shared memory 的頻寬影響較大，這是因為 `forward_kernel` 裡主要使用的記憶體種類為 shared memory，因此對他影響甚巨。



Time Distribution

最後是時間組成的實驗，這裡使用了許多不同 input size 的測資去做實驗。從實驗結果可以得知一個有趣的現象：輸入資料的 `batch_size (B)` 主要是影響 `IO` 以及 memory copy (`H2D` / `D2H`) 的時間，對計算時間影響不大；而每個 batch 的資料大小 (`N`) 則對計算時間有劇烈的影響。這裡可以得知我的程式實作方法，對於 `N` 比較大的情況，在計算時間上會比較吃虧，這也是一個可以進一步優化的方向。



Experience & conclusion

在實作 FlashAttention 的過程中，我獲得了許多難得的經驗。這個作業讓我深入理解了記憶體存取模式對效能的重要性。透過實驗發現，善用 shared memory 能大幅提升程式效能，最終達到了 3.5 倍的加速；而在 `BS` 大小的實驗中，我觀察到較大的 `BS` 通常能帶來更好的效能，然而過大的 `BS` 大小會導致記憶體存取衝突，這顯示了在效能優化的同時，需要找到最佳平衡點的 trade-off。

整個實作過程其實遇過蠻多挑戰，包含複雜的 indexing 計算、thread 的使用效率與記憶體存取效率的平衡，還有 debug 的困難性。這些挑戰都需要仔細驗證邏輯正確性和反覆測試才能解決，對我來說是還蠻特殊的經驗。

這個作業讓我體會到 GPU 優化是一個需要同時考慮硬體特性、記憶體模式和演算法設計的多面向挑戰。透過不同優化技術的組合應用，不僅達到了顯著的效能提升，也加深了我對 GPU 平行程式的理解。