HW4

> 112062706 林泳成

Implementation

```
#pragma unroll 320
for (int batchIdx = 0; batchIdx < B; batchIdx++) {
    flash_attention(
        d_Q + (batchIdx * N * d),
        d_K + (batchIdx * N * d),
        d_V + (batchIdx * N * d),
        d_O + (batchIdx * N * d), d_l, d_m
    );
}
```

由於batch可能會很大，SRAM的大小有限，所以flash attention會分batch操作。

```
void flash_attention(float *q, float *k, float *v, float *o, float* d_l, float* d_m){
    cudaMemset(d_l, 0x00, N * sizeof(float));
    cudaMemset(d_m, FLT_MIN, N * sizeof(float));

    dim3 threadsPerBlock(32, 32);
    dim3 blocksPerGrid(tr, 1);
    // Outer Loop
    for (int outerIdx = 0; outerIdx < tc; outerIdx++) {
        flash_attention_kernel<<<blocksPerGrid, threadsPerBlock>>>(k + outerIdx * bc * d, v + outerIdx * bc * d, q, o, d_l, d_m, d, 1.0 / sqrt(d));
    }
}
```

Inner loop是針對每個query q切一塊tile丟進SRAM做計算

Outer loop是把Key k和value v個別切一塊tile丟進SRAM做計算

每個batch也有自己的l和m要維護

進到kernel前會把k和v切成切成$bc * d$大小的tile，也會把q切成$N * d$大小的tile

```
__shared__ float shared_sij[TileSize * (TileSize+1)];
__shared__ float shared_pij[TileSize * (TileSize+1)];
__shared__ float shared_mij[TileSize];
__shared__ float shared_lij[TileSize];
__shared__ float qi[TileSize * 65];
__shared__ float kj[TileSize * 65];
__shared__ float vj[TileSize * 65];
__shared__ float mi[TileSize];
__shared__ float li[TileSize];
__shared__ float oi[TileSize * 65];

if(bcIdx == 0){
    for(int t = 0;t < d;++t){
        qi[brIdx * (d+1) + t] = q[globalBrIdx * d + t];
    }
}
if(brIdx == 0){
    for(int t = 0;t < d;++t){
        kj[bcIdx * (d+1) + t] = d_kj[bcIdx * d + t];
    }
}
if(bcIdx == 0){
    mi[brIdx] = d_m[globalBrIdx];
    li[brIdx] = d_l[globalBrIdx];
    for(int t = 0;t < d;++t){
        oi[brIdx * (d+1) + t] = o[globalBrIdx * d + t];
    }
}
if(brIdx == 0){
    for(int t = 0;t < d;++t){
        vj[bcIdx * (d+1) + t] = d_vj[bcIdx * d + t];
    }
}
__syncthreads();
```

kernel內部會同時把所有q都去對k和v做運算(q切成的tile會在kernel裡被放進share memory中，因此一個block就會擁有一個q的tile)

```cpp
// Query Loop
    // QKDotAndScalar
    shared_sij[brIdx * (bc+1) + bcIdx] = 0.0F;
    for (int t = 0; t < d; t++) {
        shared_sij[brIdx * (bc+1) + bcIdx] += qi[brIdx * (d+1) + t] * kj[bcIdx * (d+1) + t];
    }
    shared_sij[brIdx * (bc+1) + bcIdx] *= scalar;
    __syncthreads();
    // RowMax
    if(bcIdx == 0){
        shared_mij[brIdx] = shared_sij[brIdx * (bc+1)];
        for (int j = 0; j < bc; j++) {
            shared_mij[brIdx] = _max(shared_mij[brIdx], shared_sij[brIdx * (bc+1) + j]);
        }
    }
    __syncthreads();
    // MinusMaxAndExp
    shared_pij[brIdx * (bc+1) + bcIdx] = __expf(shared_sij[brIdx * (bc+1) + bcIdx] - shared_mij[brIdx]);
    __syncthreads();
    // RowSum
    if(bcIdx == 0){
        shared_lij[brIdx] = 0.0F;
        for (int j = 0; j < bc; j++) {
            shared_lij[brIdx] += shared_pij[brIdx * (bc+1) + j];
        }
    }
    __syncthreads();
```

```cpp
// UpdateMiLiOi
if(bcIdx == 0){
    float mi_new = _max(mi[brIdx], shared_mij[brIdx]);
    float li_new = __expf(mi[brIdx] - mi_new) * li[brIdx] + __expf(shared_mij[brIdx] - mi_new) * shared_lij[brIdx];
    for (int j = 0; j < d; j++) {
        float pv = 0.0F;
        for (int t = 0; t < bc; t++) {
            pv += shared_pij[brIdx * (bc+1) + t] * vj[t * (d+1) + j];
        }
        o[globalBrIdx * d + j] = (li[brIdx] * __expf(mi[brIdx] - mi_new) * oi[brIdx * (d+1) + j] + __expf(shared_mij[brIdx] - mi_new) * pv) / li_new;
    }
    d_m[globalBrIdx] = mi_new;
    d_l[globalBrIdx] = li_new;
}
__syncthreads();
```

kernel內部就是把算有該做的操做都列出來，因為有些中繼資料(e.g. sij, mij, pij, lij)會有dependency，所以需要在做完這些操作後同步化block裡的threads
l和m會在最後一個步驟做更新。l和m會在每個qkv算完後更新，且每個batch會各自維護l和m

br和bc選擇和sequential版本一樣的32, 32，因為block的上限也是$32 * 32$個threads，而且這兩個維度最好都要是32的倍數

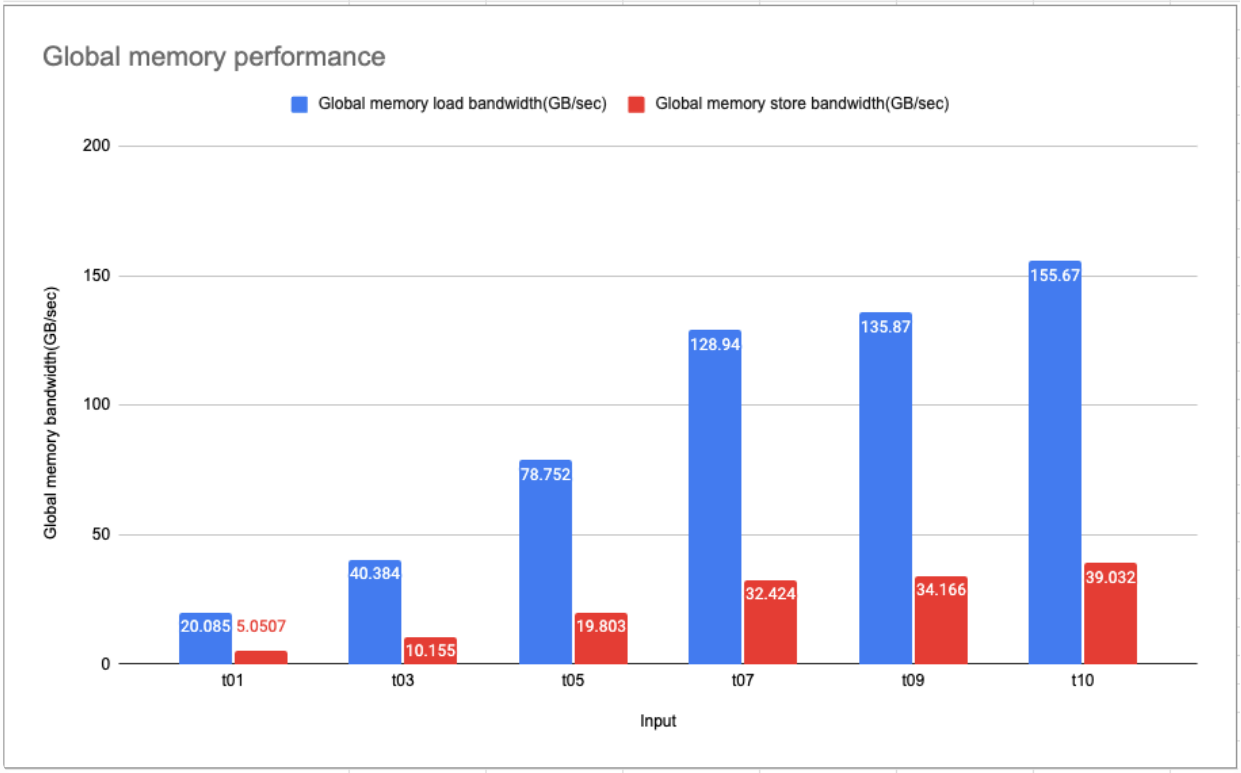
至於share memory和grid dimension的配置是根據flash attention的inner loop會需要計算整個$N * d$，而block size又是$32 * 32$也就是$tile_size * tile_size$，所以grid size要有一個維度是$\frac{N}{tile_size}$
Share memory是根據flash attention的tile大小和input d的最大值來設定的
而這樣的設定會在d = 64時可以完整用到42240 bytes的share memory(硬體可支援上限： 49152 bytes)

Profiling Results

由於profiling會增加額外的執行時間，所以如果跑的資料量太大會導致timeout，因此這裡實驗使用複雜度較低的測資。

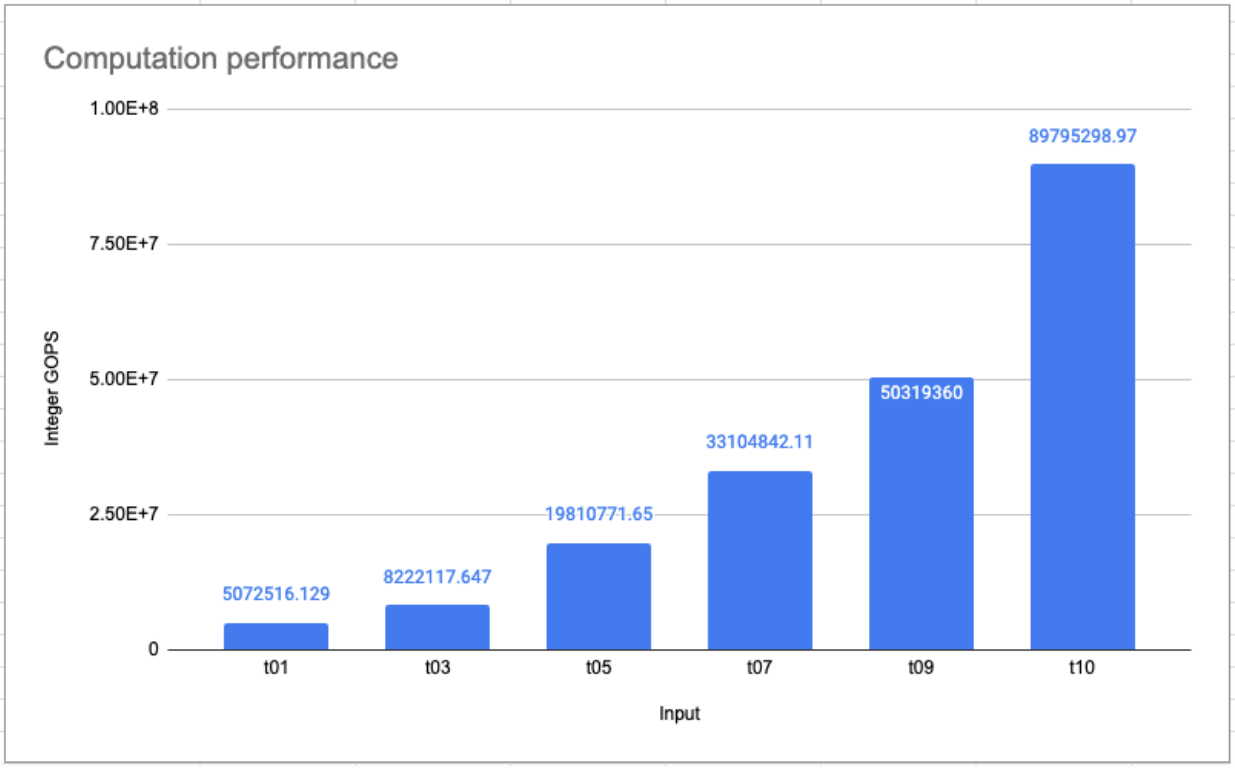
由實驗可以看出N和d的維度會大幅影響整體計算時間，因為flash attention把大矩陣切成很多小矩陣，會產生很多矩陣的運算，所以當每個小矩陣越大計算時間就會增加很多。

Input	t01	t03	t05	t07	t09	t10
B	320	160	80	40	20	10
N	128	256	512	1024	2048	2048
d	32	32	32	32	32	64
Global memory load bandwidth(GB/sec)	20.085	40.384	78.752	128.94	135.87	155.67
Global memory store bandwidth(GB/sec)	5.0507	10.155	19.803	32.424	34.166	39.032



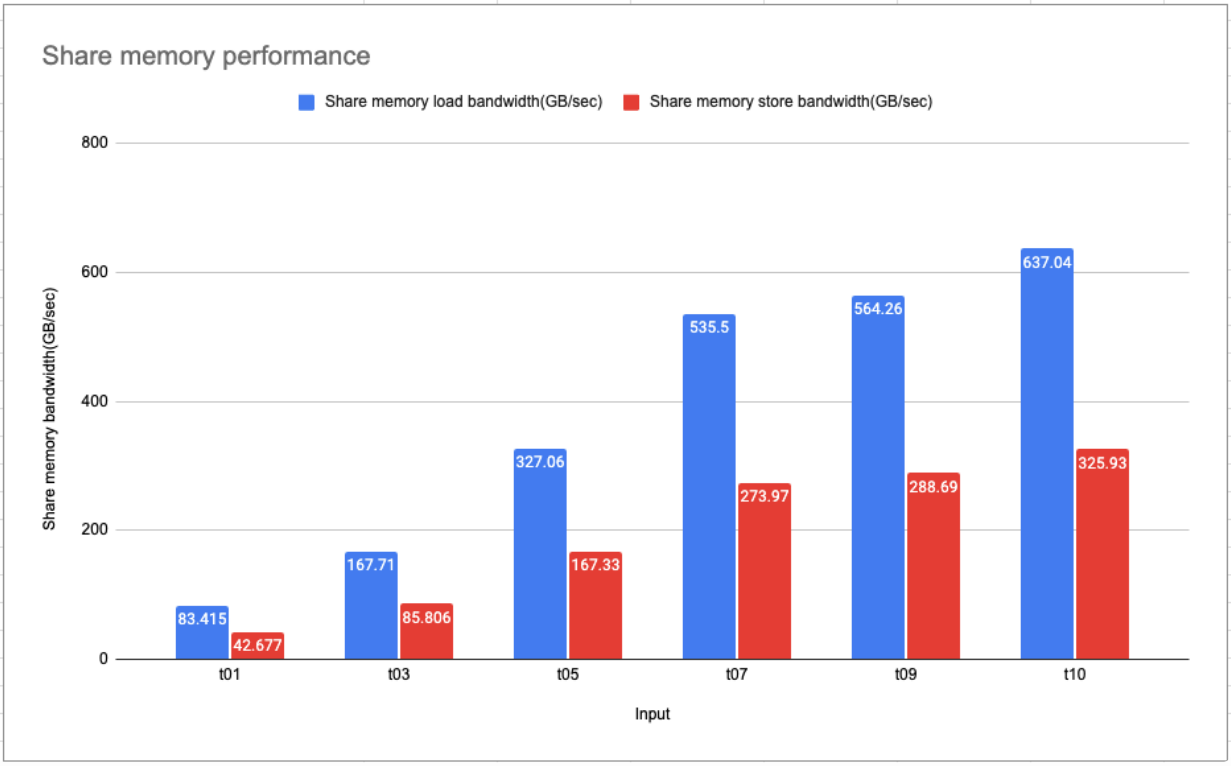
▲當資料的複雜度逐漸變大時，GPU global memory的使用量會逐漸上升，造成頻寬使用效率提升

Input	t01	t03	t05	t07	t09	t10
B	320	13600	800	300	500	4
N	128	128	512	2048	2048	32768
d	32	32	64	32	64	32
inst_integer	628992	1257984	2515968	5031936	10063872	17420288
Execution time(sec)	0.124	0.153	0.127	0.152	0.2	0.194
Integer GOPS	5072516.129	8222117.647	19810771.65	33104842.11	50319360	89795298.97



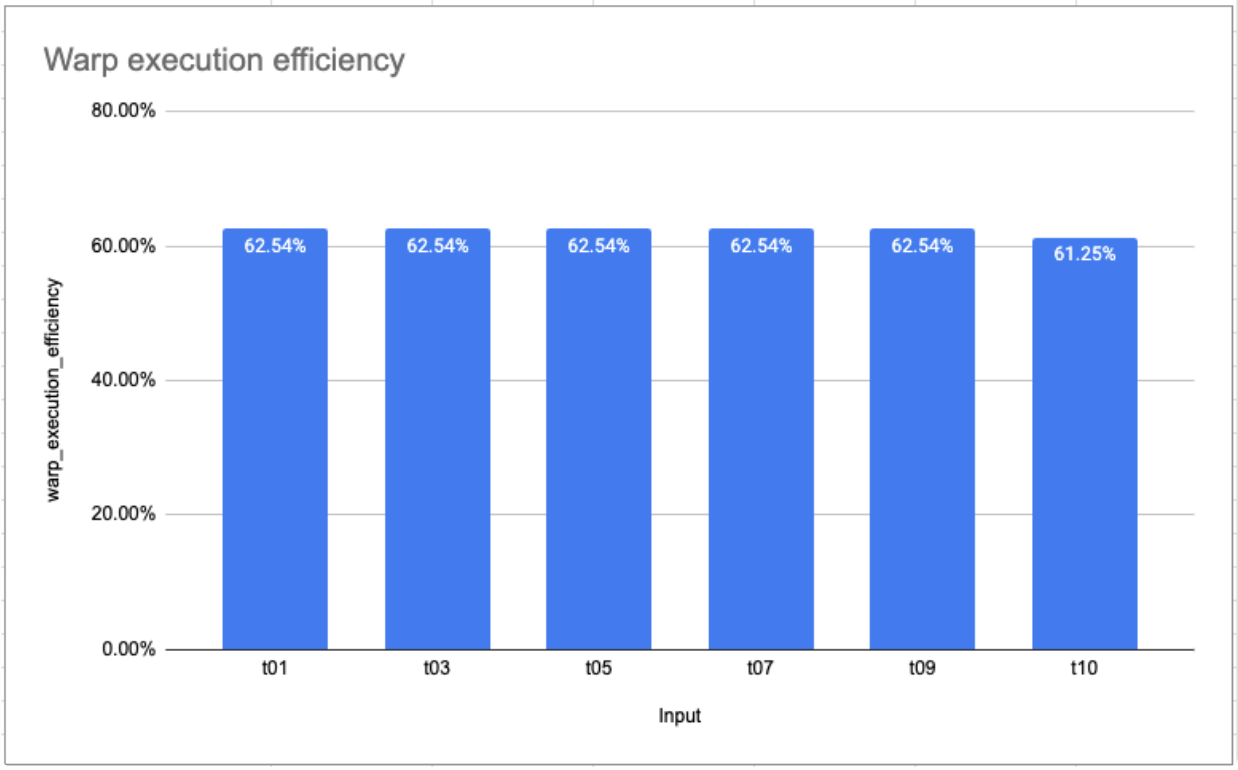
▲當資料的複雜度逐漸變大時，GPU 每秒的計算量也會逐漸提升，也就是能提升硬體的使用效率

Input	t01	t03	t05	t07	t09	t10
B	320	13600	800	300	500	4
N	128	128	512	2048	2048	32768
d	32	32	64	32	64	32
Share memory load bandwidth(GB/sec)	83.415	167.71	327.06	535.5	564.26	637.04
Share memory store bandwidth(GB/sec)	42.677	85.806	167.33	273.97	288.69	325.93



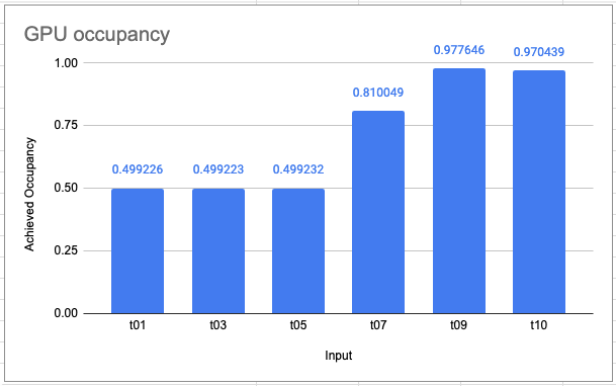
▲當資料的複雜度逐漸變大時，能塞進share memory的資料變多，GPU share memory的使用量會逐漸上升

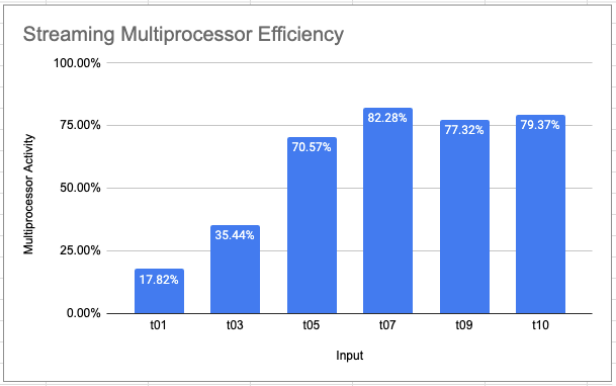
Input	t01	t03	t05	t07	t09	t10
B	320	13600	800	300	500	4
N	128	128	512	2048	2048	32768
d	32	32	64	32	64	32
warp_execution_efficiency	62.54%	62.54%	62.54%	62.54%	62.54%	61.25%



▲GPU warp內的平均使用效率不會隨著資料的複雜度變大而增加。但是由於kernel function內部會有一些工作是只有一部份的threads會做，所以免不了產生branch，所以warp efficiency沒有很高(記憶體優化已經盡量做到比較好了)

Input	t01	t03	t05	t07	t09	t10
B	320	13600	800	300	500	4
N	128	128	512	2048	2048	32768
d	32	32	64	32	64	32
Achieved Occupancy	0.499226	0.499223	0.499232	0.810049	0.977646	0.970439
Multiprocessor Activity	17.82%	35.44%	70.57%	82.28%	77.32%	79.37%





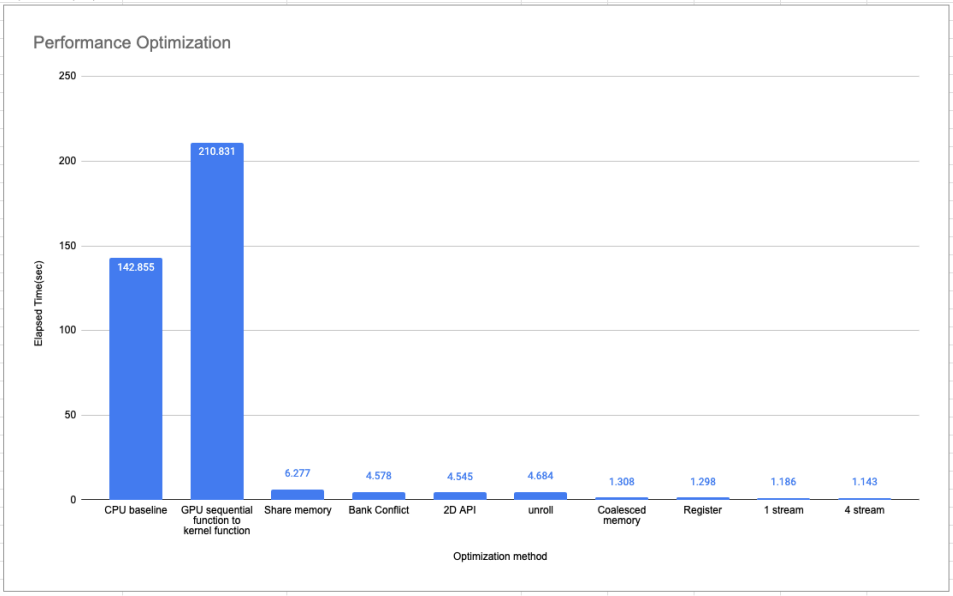
▲GPU的佔用率和Streaming Multiprocessor的使用效率有隨著計算量上升而有顯著提升，代表GPU的資源有逐漸被佔滿

Experiment & Analysis

Optimization

優化的實驗是使用t28的測資(B=4, N=16384, d=64)，因為這個複雜度的sequential運行時間很常，但是GPU優化可以跑得很快。

Optimization method	CPU baseline	GPU sequential function to kernel function	Share memory	Bank Conflict	2D API	unroll	Coalesced memory	Register	1 stream	4 stream
Elapsed Time(sec)	142.855	210.831	6.277	4.578	4.545	4.684	1.308	1.298	1.186	1.143



Share memory & Bank conflict

```
__shared__ float shared_sij[TileSize * (TileSize+1)];
__shared__ float shared_pij[TileSize * (TileSize+1)];
__shared__ float shared_mij[TileSize];
__shared__ float shared_lij[TileSize];
__shared__ float qi[TileSize * 65];
__shared__ float kj[TileSize * 65];
__shared__ float vj[TileSize * 65];
__shared__ float mi[TileSize];
__shared__ float li[TileSize];
__shared__ float oi[TileSize * 65];
```

把中繼的tile都放到share memory可以大幅減少access time，share memory也是flash attention最主要能表現好的原因。

為了解決share memory的bank conflict，在矩陣的最尾端加上一個padding就能簡單解決同一個thread一直access同一個bank的資料。

2D API

```
cudaMallocPitch(&d_O, &pitchO, N * d * sizeof(float), B);
cudaMallocPitch(&d_Q, &pitchQ, N * d * sizeof(float), B);
cudaMallocPitch(&d_K, &pitchK, N * d * sizeof(float), B);
cudaMallocPitch(&d_V, &pitchV, N * d * sizeof(float), B);

cudaMemcpy2D(d_O, pitchO, O, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice);
cudaMemcpy2D(d_Q, pitchQ, Q, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice);
cudaMemcpy2D(d_K, pitchK, K, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice);
cudaMemcpy2D(d_V, pitchV, V, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice);


// float *h_l = (float *)malloc(N * sizeof(float));
// float *h_m = (float *)malloc(N * sizeof(float));

cudaMalloc(&d_l, N * sizeof(float));
cudaMalloc(&d_m, N * sizeof(float));

for (int batchIdx = 0; batchIdx < B; batchIdx++) {
    flash_attention(
        d_Q + (batchIdx * N * d),
        d_K + (batchIdx * N * d),
        d_V + (batchIdx * N * d),
        d_O + (batchIdx * N * d), d_l, d_m, batchIdx
    );
}
cudaMemcpy2D(O, N * d * sizeof(float), d_O, pitchO, N * d * sizeof(float), B, cudaMemcpyDeviceToHost);
```

2D API會先把記憶體補滿，這樣整個二維陣列就會是連續記憶體，複製起來也會比較快。

unroll

```
#pragma unroll 320
for (int batchIdx = 0; batchIdx < B; batchIdx++) {
    flash_attention(
        d_Q + (batchIdx * N * d),
        d_K + (batchIdx * N * d),
        d_V + (batchIdx * N * d),
        d_O + (batchIdx * N * d), d_l, d_m, batchIdx
    );
}
```

有for迴圈就可以unroll一夏，只是因為不知道輸入的batch數量，所以就塞了一個中間值。基本上沒差多少。

Coalesced memory

```
int bcIdx = threadIdx.y;
int brIdx = threadIdx.x;
int globalBrIdx = blockIdx.x * blockDim.x + threadIdx.x;
int bc = blockDim.y;
```

因為brIdx比較常在陣列第一個維度的索引值，所以就讓相同y，不同x的thread去access會比較快。

Register

```
float sij = 0.0F;
for (int t = 0; t < d; t++) {
    sij += qi[brIdx * (d+1) + t] * kj[bcIdx * (d+1) + t];
}
shared_sij[brIdx * (bc+1) + bcIdx] = sij * scalar;
```

```
if(bcIdx == 0){
    float mij = shared_sij[brIdx * (bc+1)];
    for (int j = 0; j < bc; j++) {
        mij = _max(mij, shared_sij[brIdx * (bc+1) + j]);
    }
    shared_mij[brIdx] = mij;
}
```

```
if(bcIdx == 0){
    float lij = 0.0F;
    for (int j = 0; j < bc; j++) {
        lij += shared_pij[brIdx * (bc+1) + j];
    }
    shared_lij[brIdx] = lij;
}
```

因為register的access time會比share memory的array快一點點，所以迴圈中會重複使用到的share memory access就改成register。

1 Stream & 4 Stream

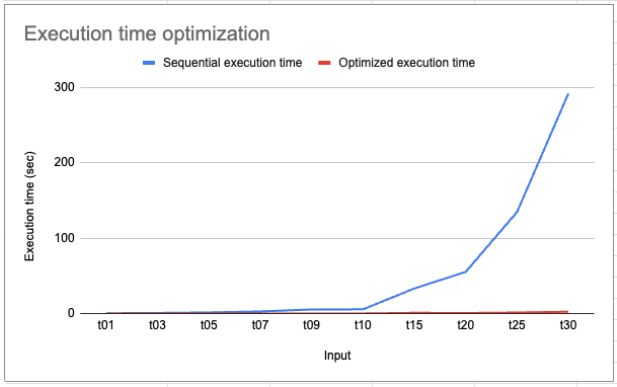
```
cudaMemcpy2DAsync(d_O, pitchO, O, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream);
cudaMemcpy2DAsync(d_Q, pitchQ, Q, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream);
cudaMemcpy2DAsync(d_K, pitchK, K, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream);
cudaMemcpy2DAsync(d_V, pitchV, V, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream);
```

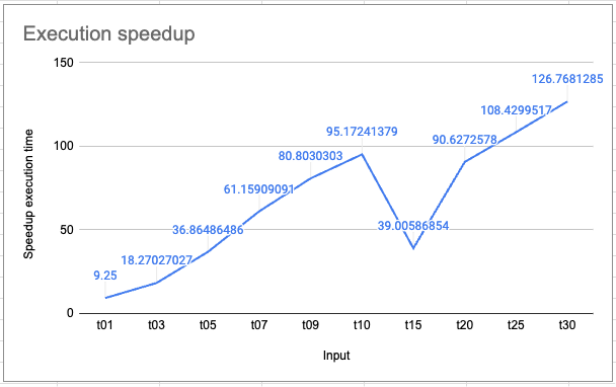
```
cudaMemcpy2DAsync(d_O, pitchO, O, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream[0]);
cudaMemcpy2DAsync(d_Q, pitchQ, Q, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream[1]);
cudaMemcpy2DAsync(d_K, pitchK, K, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream[2]);
cudaMemcpy2DAsync(d_V, pitchV, V, N * d * sizeof(float), N * d * sizeof(float), B, cudaMemcpyHostToDevice, stream[3]);
```

由於在程式一開始會需要把host的Q, K, V, O搬到device，所以如果都在stream 0搬會沒辦法產生overlap。因此接下來要決定的就是要針對每一個copy function都創建一個stream還是只用一個額外stream就好，畢竟只複製一次(等於是create stream的overhead和實際搬移資料所花費時間之間的選擇)。實驗結果是大部分測資都會需要搬一小段時間，而這個overlap的時間有超過create stream所花費時間。所以選擇用4個stream針對每個array都創建一個stream來搬資料。

Other graph (Execution time speedup)

Input	t01	t03	t05	t07	t09	t10	t15	t20	t25	t30
B	320	160	80	40	20	10	2000	100	30	2
N	128	256	512	1024	2048	2048	512	2048	8192	32768
d	32	32	32	32	32	64	32	64	32	64
Sequential execution time	0.333	0.676	1.364	2.691	5.333	5.52	33.233	55.192	134.67	291.947
Optimized execution time	0.036	0.037	0.037	0.044	0.066	0.058	0.852	0.609	1.242	2.303
Speedup execution time	9.25	18.27027027	36.86486486	61.15909091	80.8030303	95.17241379	39.00586854	90.6272578	108.4299517	126.7681285





使用flash attention把array切成多個tile並使用GPU來平行access搬到share memory裡的資料可以加速原本的sequential code很多倍。

當資料的複雜度上升(尤其是N和d上升)時，CPU sequential的執行時間上升幅度很大。但當使用GPU加速時，執行時間上升幅度不會這麼大，而且也不一定會隨著N和d上升執行時間就跟著上升。以t15測資為例，sequential會因為整體資料的複雜度上升而上升，但是在GPU的情境下t15執行時間卻比t20慢。主要原因應該也是因為N和d的上升對sequential很致命，但對GPU來說不會，因此B上升太多才導致t15算得比較久。

Conclusion

從把sequential flash attention改成GPU code的過程中，我學到了tile algorithm的實際操作方式，也瞭解了在頻繁access 較小array的情況下share memory會比global memory快上很多。因此當問題的array較大時，把大array切成小tile能加速access速度很多。

從這次作業中也發現coalesced memory比起其他加速手法真的能快上很多倍。另一個加速的重點是在可容許些微錯誤的情境下，使用合理的flag來優化浮點數計算速度，也可以加速計算很多。