NTHU CS542200 Parallel Programming Homework 4: FlashAttention

112062591 李威辰

implementation

此作業的目標是要將FlashAttention中間的矩陣運算利用CUDA做平行化。

1. **輸入資料**

```
void input(char *input_filename) {
    FILE *file = fopen(input_filename, "rb");

    fread(&B, sizeof(int), 1, file);
    fread(&N, sizeof(int), 1, file);
    fread(&d, sizeof(int), 1, file);

    Q = (float *)malloc(B * N * d * sizeof(float));
    K = (float *)malloc(B * N * d * sizeof(float));
    V = (float *)malloc(B * N * d * sizeof(float));
    O = (float *)malloc(B * N * d * sizeof(float));

    for (int i = 0; i < B; i++) {
        fread(Q + (i * N * d), sizeof(float), N * d, file);
        fread(K + (i * N * d), sizeof(float), N * d, file);
        fread(V + (i * N * d), sizeof(float), N * d, file);
    }
    memset(O, 0x00, B * N * d * sizeof(float));

    fclose(file);
}
```

B : batch size

Q, K, V : Query, Key, and Value matrices(Nxd) in HBM (High Bandwidth Memory).

O : Output matrices(Nxd).

2. **flashattention運算**

   將運算分成B個batch做，每個iteration做一次flashattention運算

```
for (int i = 0; i < B; i++) {
        flash_attention(
            Q + (i * N * d),
            K + (i * N * d),
            V + (i * N * d),
            O + (i * N * d),
            N,
            d
        );
    }
```

以下為主要運算函數:

```
void flash_attention(float *q, float *k, float *v, float *o, int N, int d)
    float *d_q, *d_k, *d_v, *d_o;
    float *d_sij, *d_pij, *d_mij, *d_lij, *d_m, *d_l;

    int size_q = N * d * sizeof(float);
    int size_o = N * d * sizeof(float);
    int size_temp = N * N * sizeof(float);

    cudaMalloc(&d_q, size_q);
    cudaMalloc(&d_k, size_q);
    cudaMalloc(&d_v, size_q);
    cudaMalloc(&d_o, size_o);
    cudaMalloc(&d_sij, size_temp);
    cudaMalloc(&d_pij, size_temp);
    cudaMalloc(&d_mij, N * sizeof(float));
    cudaMalloc(&d_lij, N * sizeof(float));
    cudaMalloc(&d_m, N * sizeof(float));
    cudaMalloc(&d_l, N * sizeof(float));

    cudaMemcpy(d_q, q, size_q, cudaMemcpyHostToDevice);
    cudaMemcpy(d_k, k, size_q, cudaMemcpyHostToDevice);
    cudaMemcpy(d_v, v, size_q, cudaMemcpyHostToDevice);
    cudaMemcpy(d_o, o, size_o, cudaMemcpyHostToDevice);

    cudaMemset(d_m, 0.0F, N * sizeof(float));
    cudaMemset(d_l, 0.0F, N * sizeof(float));


    dim3 blockSize(32, 32);
    dim3 gridSize((N + 31) / 32, (N + 31) / 32);

    QKDotAndScalar<<<gridSize, blockSize>>>(d_q, d_k, d_sij, d, 1.0 / sqrt(
    cudaDeviceSynchronize();
    RowMax<<<(N + 1023) / 1024, 1024>>>(d_sij, d_mij, N, N);
    cudaDeviceSynchronize();
    MinusMaxAndExp<<<gridSize, blockSize>>>(d_sij, d_pij, d_mij, N, N);
    cudaDeviceSynchronize();
    RowSum<<<(N + 1023) / 1024, 1024>>>(d_pij, d_lij, N, N);
    cudaDeviceSynchronize();
    UpdateMiLiOi<<<gridSize, blockSize>>>(d_m, d_l, d_o, d_mij, d_lij, d_pi
    cudaDeviceSynchronize();

    cudaMemcpy(o, d_o, size_o, cudaMemcpyDeviceToHost);

    cudaFree(d_q);
    cudaFree(d_k);
    cudaFree(d_v);
    cudaFree(d_o);
    cudaFree(d_sij);
    cudaFree(d_pij);
    cudaFree(d_mij);
    cudaFree(d_lij);
    cudaFree(d_m);
    cudaFree(d_l);
}
```

先將中間會用到的陣列做初始化，並用cudaMalloc和cudaMemcpy將資料從host複製到GPU上並用d_標記讓後續可以做平行運算。

```
float *d_q, *d_k, *d_v, *d_o;
float *d_sij, *d_pij, *d_mij, *d_lij, *d_m, *d_l;

int size_q = N * d * sizeof(float);
int size_o = N * d * sizeof(float);
int size_temp = N * N * sizeof(float);

cudaMalloc(&d_q, size_q);
cudaMalloc(&d_k, size_q);
cudaMalloc(&d_v, size_q);
cudaMalloc(&d_o, size_o);
cudaMalloc(&d_sij, size_temp);
cudaMalloc(&d_pij, size_temp);
cudaMalloc(&d_mij, N * sizeof(float));
cudaMalloc(&d_lij, N * sizeof(float));
cudaMalloc(&d_m, N * sizeof(float));
cudaMalloc(&d_l, N * sizeof(float));

cudaMemcpy(d_q, q, size_q, cudaMemcpyHostToDevice);
cudaMemcpy(d_k, k, size_q, cudaMemcpyHostToDevice);
cudaMemcpy(d_v, v, size_q, cudaMemcpyHostToDevice);
cudaMemcpy(d_o, o, size_o, cudaMemcpyHostToDevice);

cudaMemset(d_m, 0.0F, N * sizeof(float));
cudaMemset(d_l, 0.0F, N * sizeof(float));
```

然後依照本課程提供的sequential程式碼改成平行化，分成五個階段做運算

```
dim3 blockSize(32, 32);
dim3 gridSize((N + 31) / 32, (N + 31) / 32);

QKDotAndScalar<<<gridSize, blockSize>>>(d_q, d_k, d_sij, d, 1.0 / sqrt(d),
cudaDeviceSynchronize();
RowMax<<<(N + 1023) / 1024, 1024>>>(d_sij, d_mij, N, N);
cudaDeviceSynchronize();
MinusMaxAndExp<<<gridSize, blockSize>>>(d_sij, d_pij, d_mij, N, N);
cudaDeviceSynchronize();
RowSum<<<(N + 1023) / 1024, 1024>>>(d_pij, d_lij, N, N);
cudaDeviceSynchronize();
UpdateMiLiOi<<<gridSize, blockSize>>>(d_m, d_l, d_o, d_mij, d_lij, d_pij, d
cudaDeviceSynchronize();
```

QKDotAndScalar: 此函數將Q、K做矩陣相乘在乘上一個係數存到d_sij，一個Block裝有32x32threads(block factor=32, 一個block的資料量為q = k =32 xd)，所以Blocks per grid的數量為(N + 31) / 32 x (N + 31) / 32 取ceil，總threads數量為NxN，然後每個threads做平行計算該行即該列的矩陣相乘。

```
__global__ void QKDotAndScalar(float *q, float *k, float *out, int d, float
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float value = 0.0F;
        for (int t = 0; t < d; t++) {
            value += q[row * d + t] * k[col * d + t];
        }
        out[row * N + col] = value * scalar;
    }
}
```

RowMax: 將d_sij的每一列最大值存到d_mij中，有N列，一個Block裝有1024 threads(block factor=1024, 一個block的資料量為out = in =1024x N)，所以Blocks per grid的數量為(N + 1023) / 1024取ceil，總threads為N，每個threads平行取N列的最大值。

```
__global__ void RowMax(float *in, float *out, int N, int br) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N) {
        float max_val = -FLT_MAX;
        for (int j = 0; j < br; j++) {
            max_val = _max(max_val, in[row * br + j]);
        }
        out[row] = max_val;
    }
}
```

MinusMaxAndExp : 將d_sij每一列的值與該列最大值相減(d_mij[row])再做exponential 運算，一個Block裝有32x32threads(block factor=32, 一個block的資料量為out = in = 32 x 32)，所以Blocks per grid的數量為(N + 31) / 32 x (N + 31) / 32 取ceil，總threads數量為NxN，然後每個threads做平行計算。

```
__global__ void MinusMaxAndExp(float *in, float *out, float *max_vals, int
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < N && col < br) {
        out[row * br + col] = exp(in[row * br + col] - max_vals[row]);
    }
}
```

RowSum : 將d_pij的每列數值將起來存到d_lij中，有N列，一個Block裝有1024 threads(block factor=1024, 一個block的資料量為in=1024x N)，所以Blocks per grid的數量為(N + 1023) / 1024取ceil，總threads為N，每個threads平行做N列的元素相加。

```
__global__ void RowSum(float *in, float *out, int N, int br) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N) {
        float sum = 0.0F;
        for (int j = 0; j < br; j++) {
            sum += in[row * br + j];
        }
        out[row] = sum;
    }
}
```

UpdateMiLiOi : 可以看sequential code 會發現這裡主要在做兩個矩陣的相乘運算d_pij和d_v，所以一樣跟上述QKDotAndScalar函數的平行方法類似，但因為這個是NxN 乘上Nxd出來結果為Nxd，所以這裡總共分為Nxd個threads做運算，一個Block裝有32x32threads，所以Blocks per grid的數量為(N + 31) / 32 x (N + 31) / 32 取ceil，總threads數量為NxN，只取其中Nxd個threads做運算，最後再只取col = blockIdx.x * blockDim.x + threadIdx.x = 0的threads(一行N)做mi和li的更新值。
(block factor=32, 一個block的資料量為mi = li = 32xN, pij = 32xN, vj =32 xd, oi = 32 x32)

```
__global__ void UpdateMiLiOi(float *mi, float *li, float *oi, float *mij, f
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < br && col < d) {
        float mi_new = _max(mi[row], mij[row]);
        float li_new = exp(mi[row] - mi_new) * li[row] + exp(mij[row] - mi_

        float pv = 0.0F;
        for (int t = 0; t < br; t++) {
            pv += pij[row * br + t] * vj[t * d + col];
        }
        oi[row * d + col] = (li[row] * exp(mi[row] - mi_new) * oi[row * d +

        if (col == 0) {  // Only update `mi` and `li` once per row
            mi[row] = mi_new;
            li[row] = li_new;
        }
    }
}
```

經過上述的函數就運算完畢，最後再將CUDA device的結果d_o用Memcpy傳回host即為答案(最後記得釋放記憶體)。

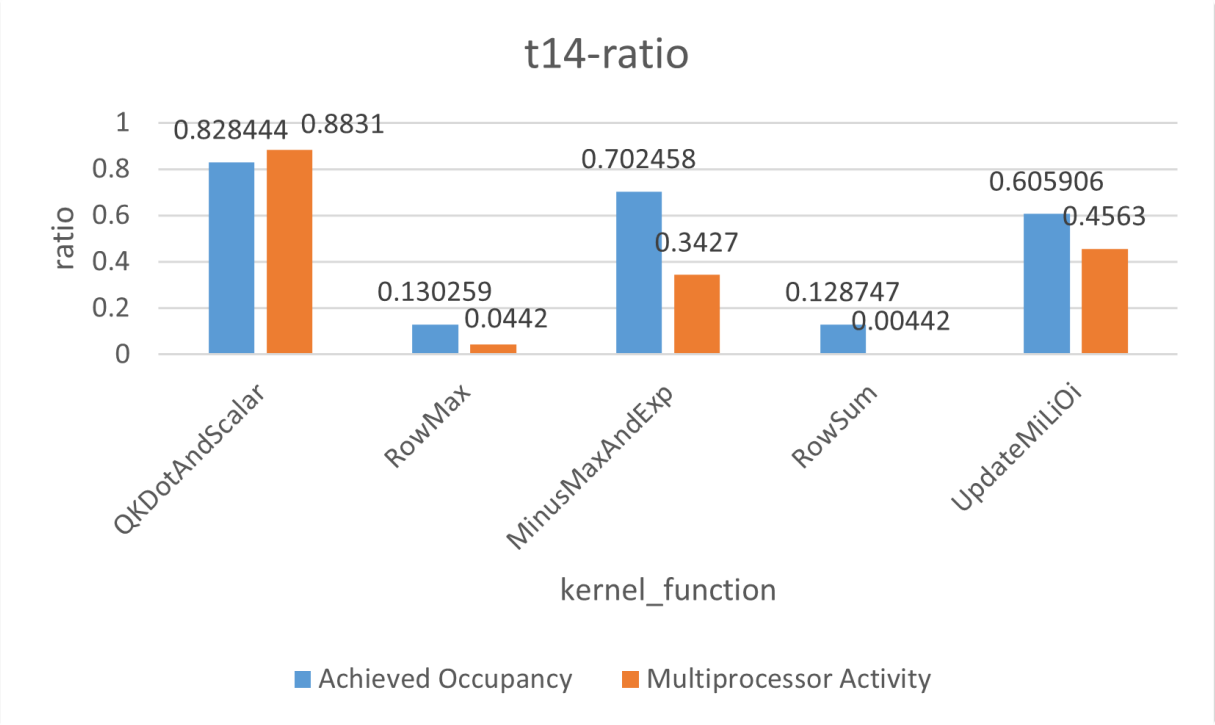
```
cudaMemcpy(o, d_o, size_o, cudaMemcpyDeviceToHost);

cudaFree(d_q);
cudaFree(d_k);
cudaFree(d_v);
cudaFree(d_o);
cudaFree(d_sij);
cudaFree(d_pij);
cudaFree(d_mij);
cudaFree(d_lij);
cudaFree(d_m);
cudaFree(d_l);
```

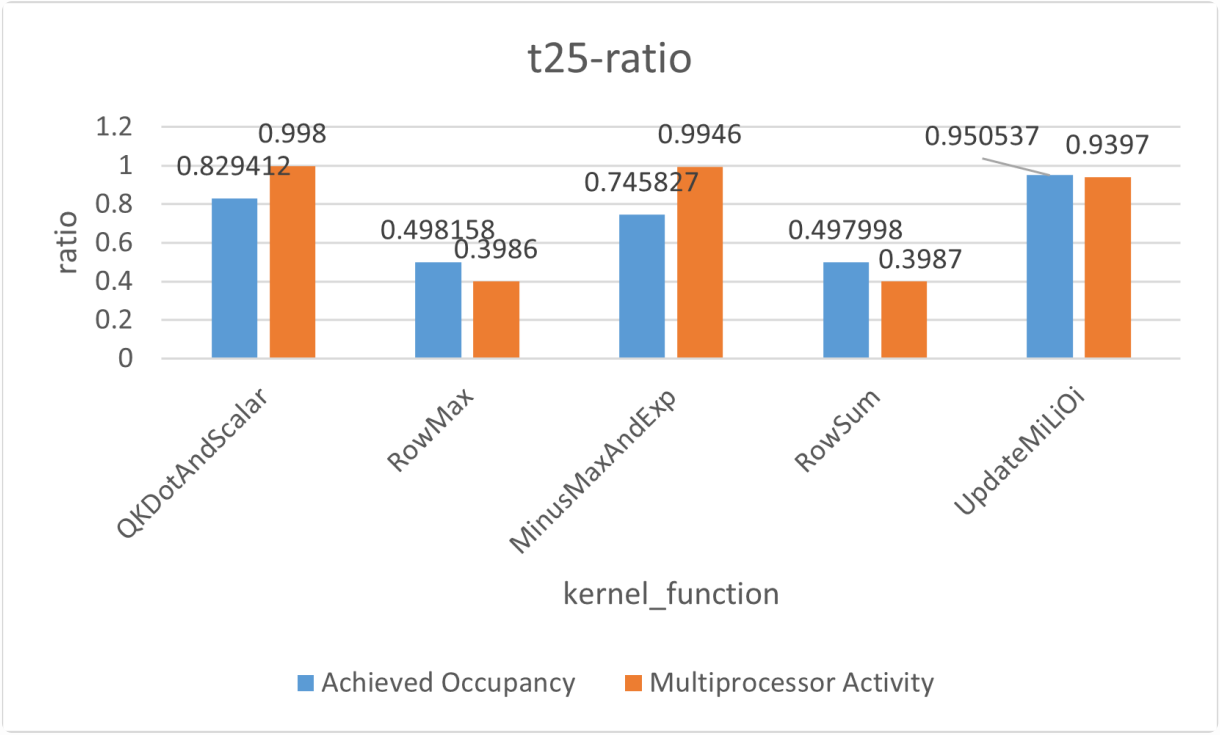
Profiling Results

以下測量都是在本課程提供之Apollo-GPU平台

Occupancy and sm efficiency

以下使用nvprof和t14和t25兩筆測資去測量的Occupancy and sm efficiency:

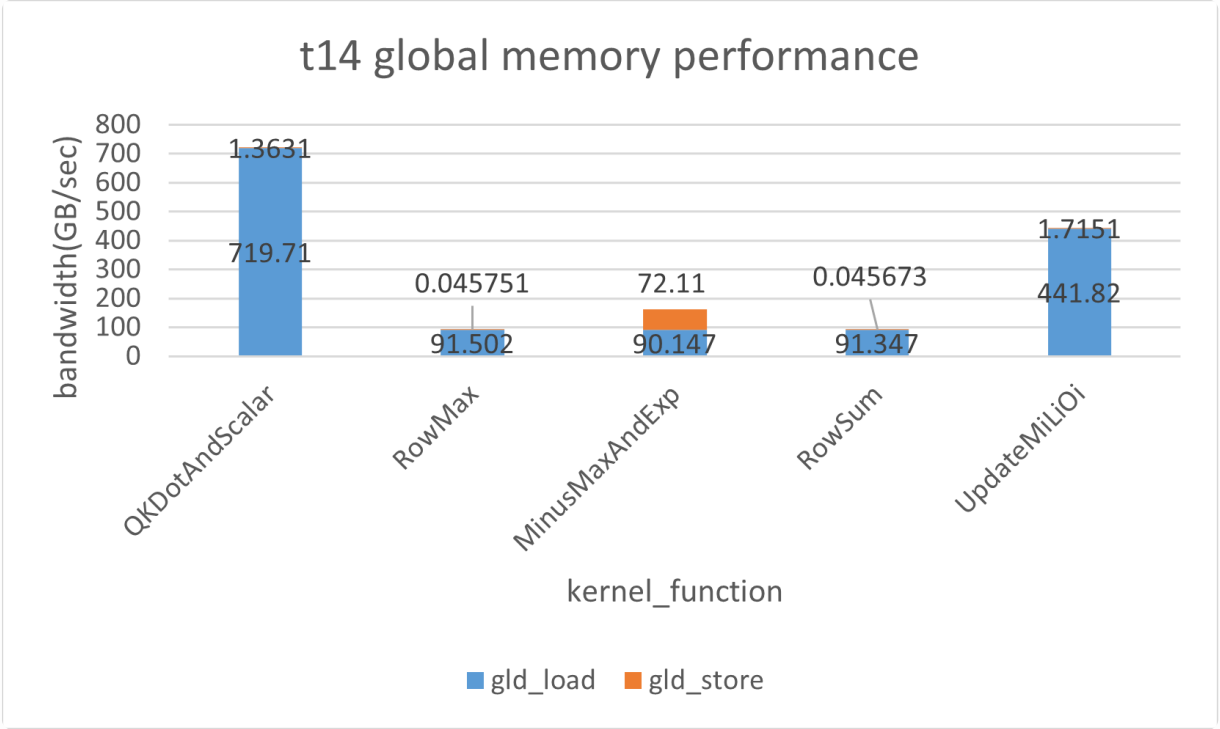


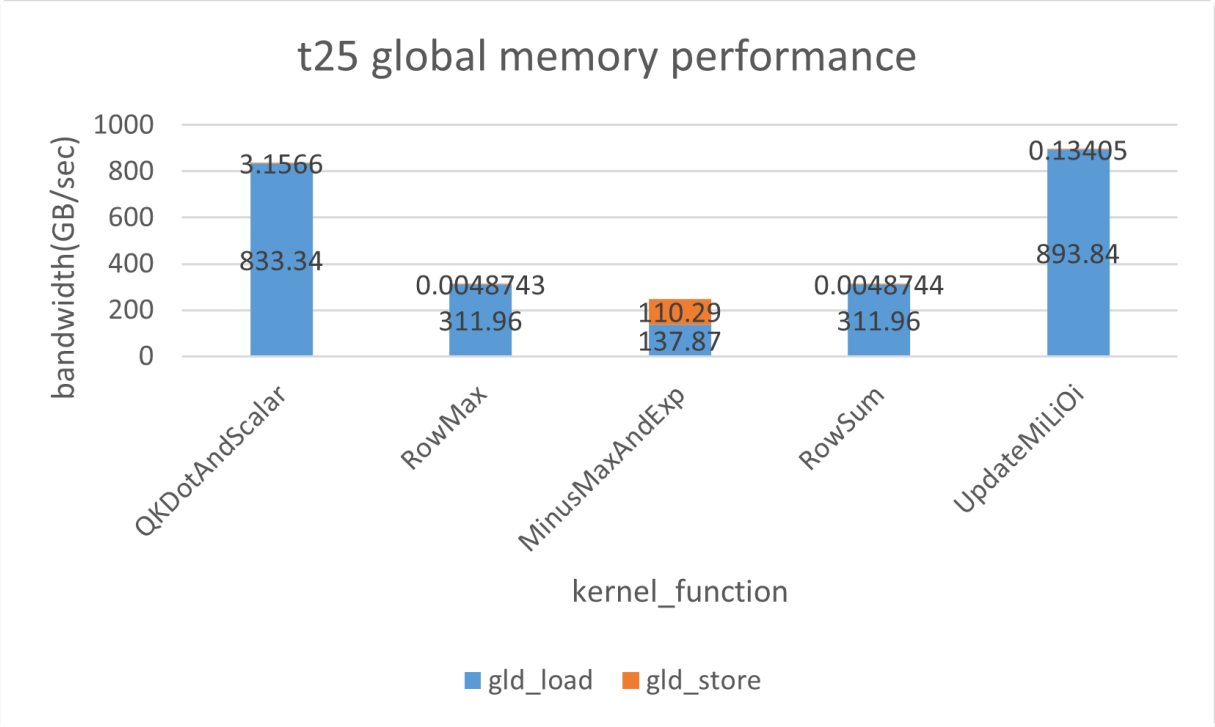


Achieved Occupancy(GPU上執行核心的使用率)和Multiprocessor Activity(GPU活動時間與總時間的比例)是用來看GPU的利用資源比例，所以期望越接近1越好，可以發現RowMax和RowSum是比率比較低的，原因是因為此兩函數使用一個block一行1024個threads去切分計算，所以當N很小時(t14中的N=256和t25中的N=8192)就會使資源使用比例下降，但每個工作也都有被完整的平行化計算，所以也不算有因為比例低而降低速度的問題。

Global data throughput

以下使用nvprof和t14和t25兩筆測資去測量的Global data throughput(load and store):

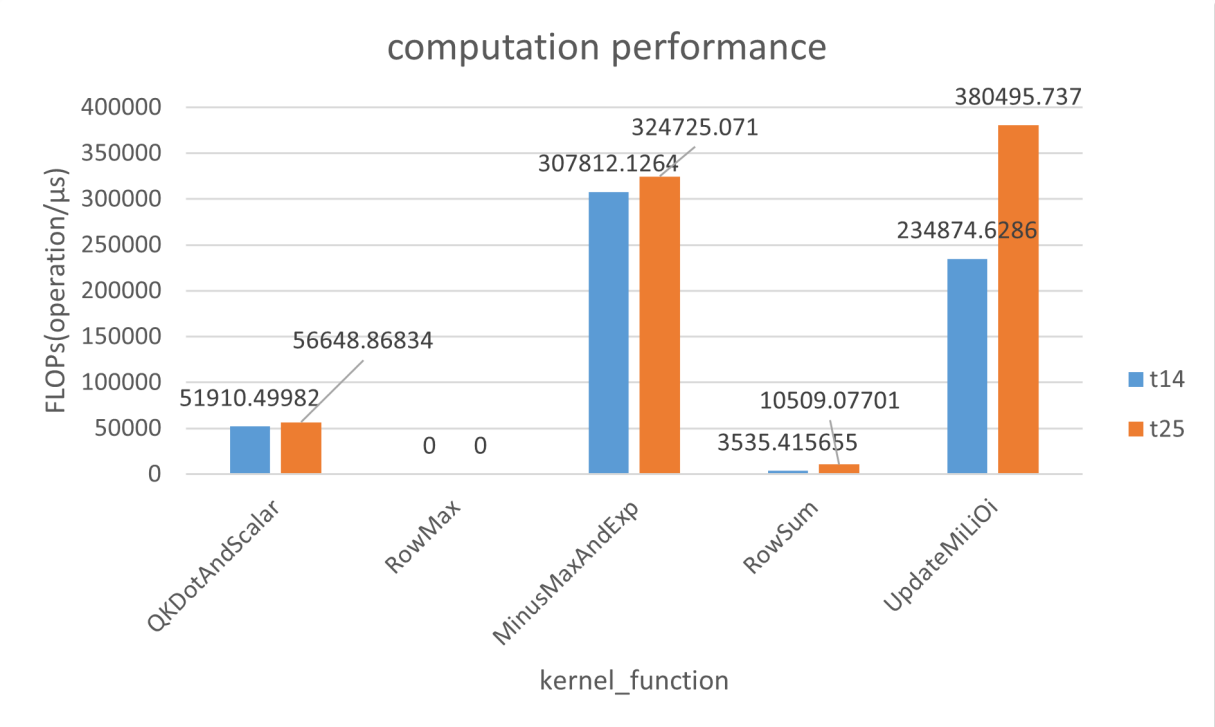




可以發現 kernel function QKDotAndScalar和UpdateMiLiOi的load是最高的，因為裡面的資料運算此函數最多，相比其他三個只有load單個N×N矩陣，而MinusMaxAndExp的store最高因為他是輸出更新N×N矩陣的數值相比其他只有輸出一行N×1矩陣，而這裡有一個比較奇怪的事情是QKDotAndScalar雖然也是輸出N×N矩陣而他的throughput比較低，猜測原因是因為他在做矩陣的每一行列的平行運算計算量較大(而且每個threads中間還有一個迴圈)造成運算時間大Bandwidth就比較低。

Computation performance

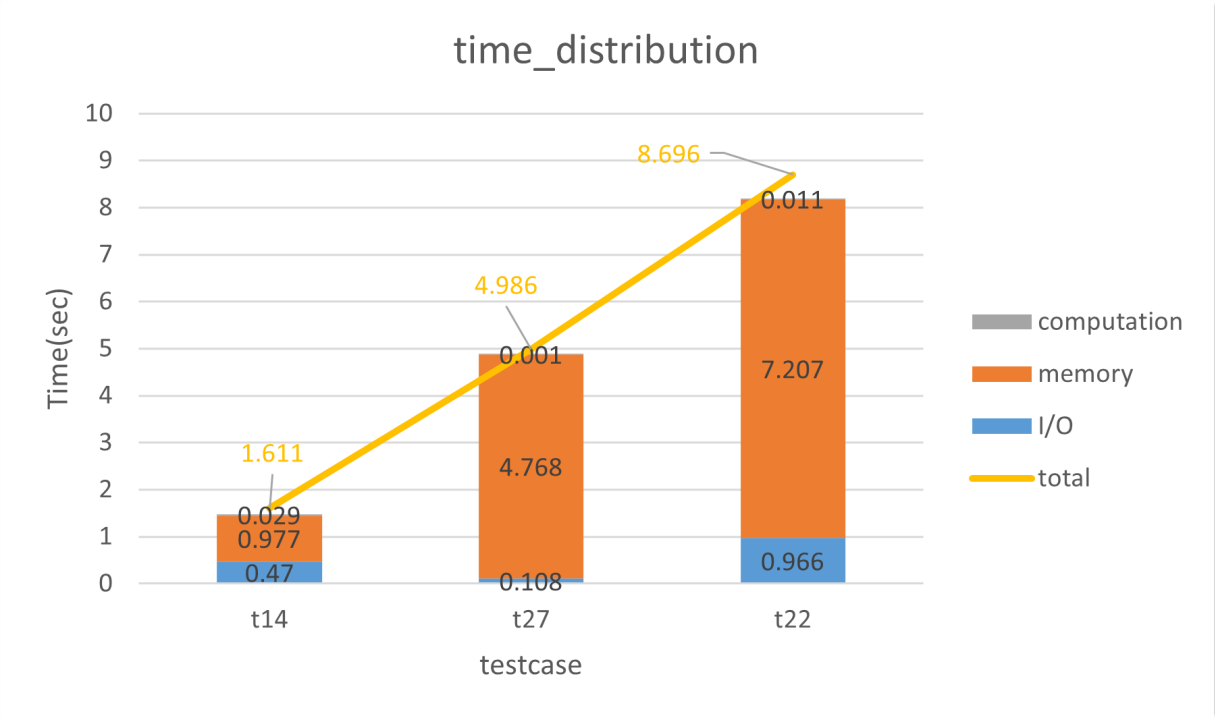
以下使用nvprof和t14和t25兩筆測資去測量的FLOPs(operation/μs):



這裡可以發現在t25的整體表現好FLOPs較高尤其是UpdateMiLiOi，猜測有可能是因為t25: (B, N, d): (30, 8192, 32)、t14: (B, N, d): (2000, 256, 64)，由於batch size t25<t14，造成t14多次跑迴圈可能有其他的時間不是在做運算，而使浮點數運算的時間占比下降，所以t14FLOPs降低。

Time distribution

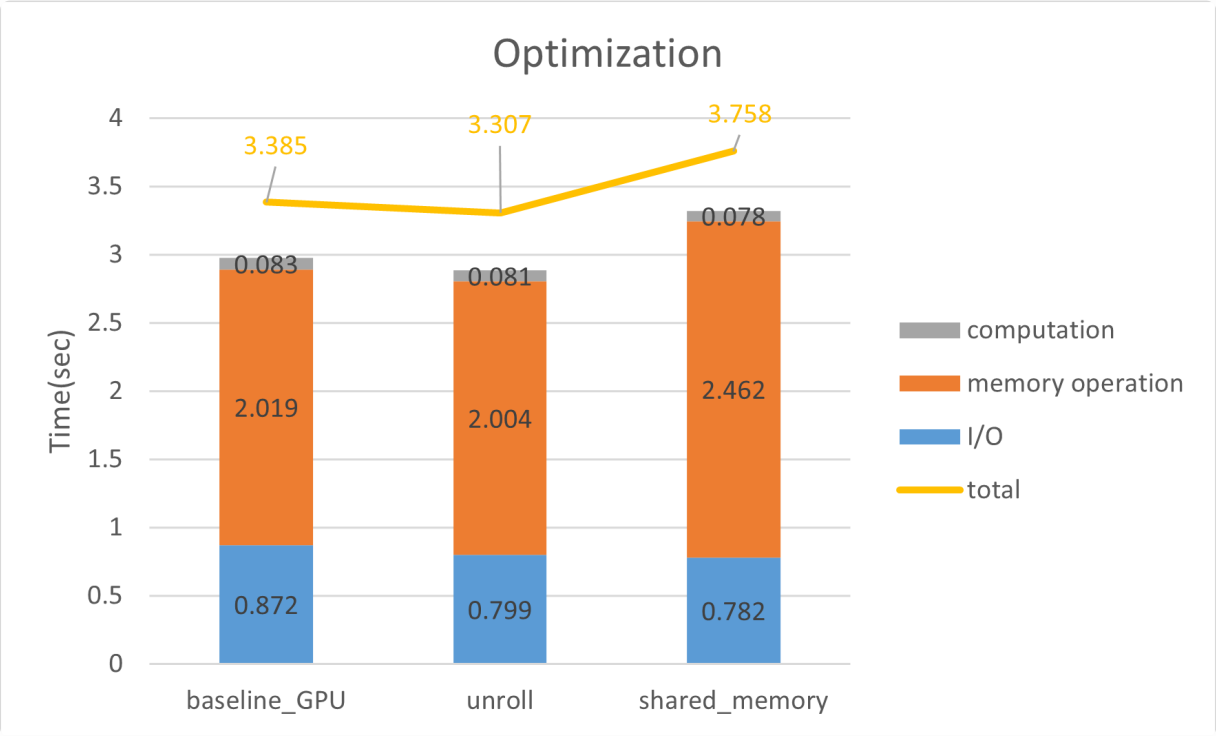
這裡要看此程式碼各個部分執行所需花的時間，以下使用gettimeofday和t14、t27和t22測資去測量時間(sec):



可以發現大部分的時間都花在memory的相關操作上(malloc、memcpy等)，然後再來是IO operation，computation占比最少，但這個結果符合預期，因為CUDA的記憶體複製和宣告(cudaMalloc、cudaMemcpy)等是最花時間的並且IO有一些網路伺服器等外在因素，所以此程式才會在computation以外的時間占比很大。

Optimization

以下使用用gettimeofday和t13測資去測量時間(sec):



這裡用了unroll和shared memory去優化看看程式碼，看會不會有performance的提升。這裡可以發現unroll有些微的速度提升，但因為提升太小(unroll是加在computation裡但卻幾乎沒有什麼提升)很有可能只是一些像是網路等因素造成unroll剛好比較快而已。

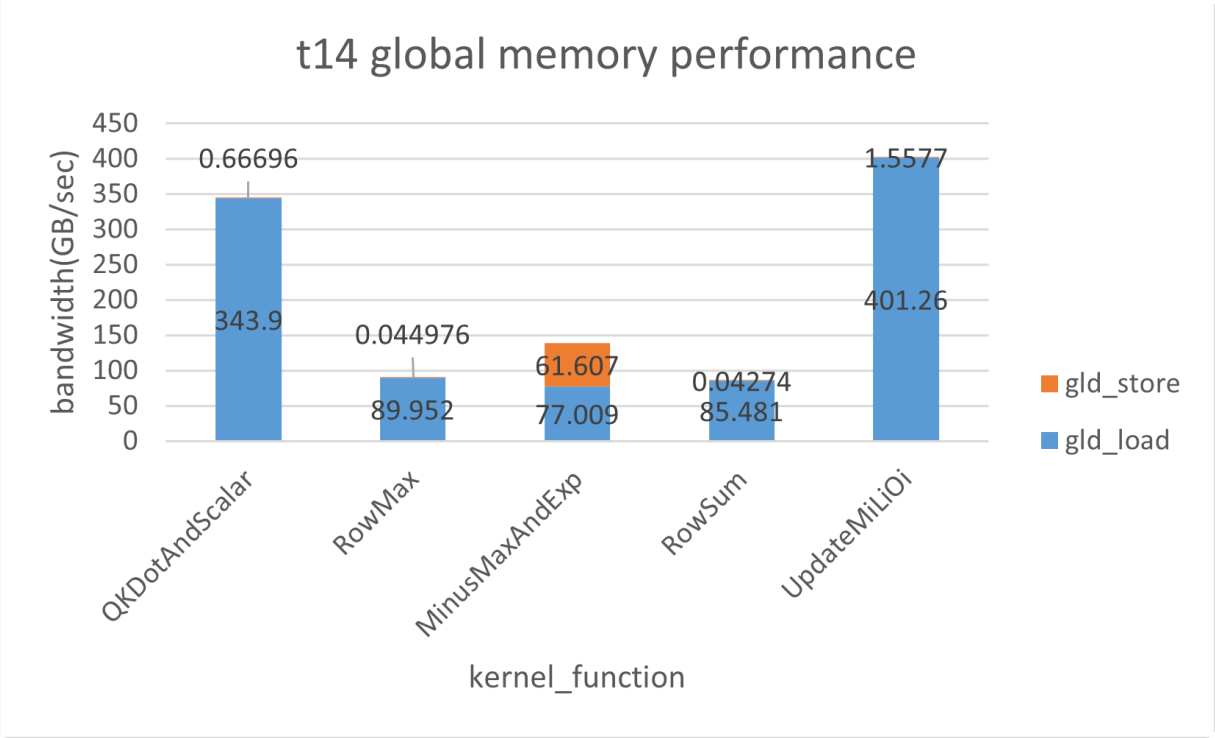
shared memory則是由於input大小的關係，所以無法有太多的優化，shared memory有容量限制，本GPU有共48KB的shared memory，因為此作業的矩陣都是浮點數，所以一共可以存12KB個，如果存超過此範圍就不行。舉個簡單的實例，RowSum一個block需要1024x N資料量，若想裝進shared memory N就必須小於12，所以很難實現shared memory，所以這裡大致在QKDotAndScalar(data amount:2x32xd)、MinusMaxAndExp(data amount:32 x 32)和UpdateMiLiOi(4 x 32 + N x 32)裡一部分的資料改成shared memory。

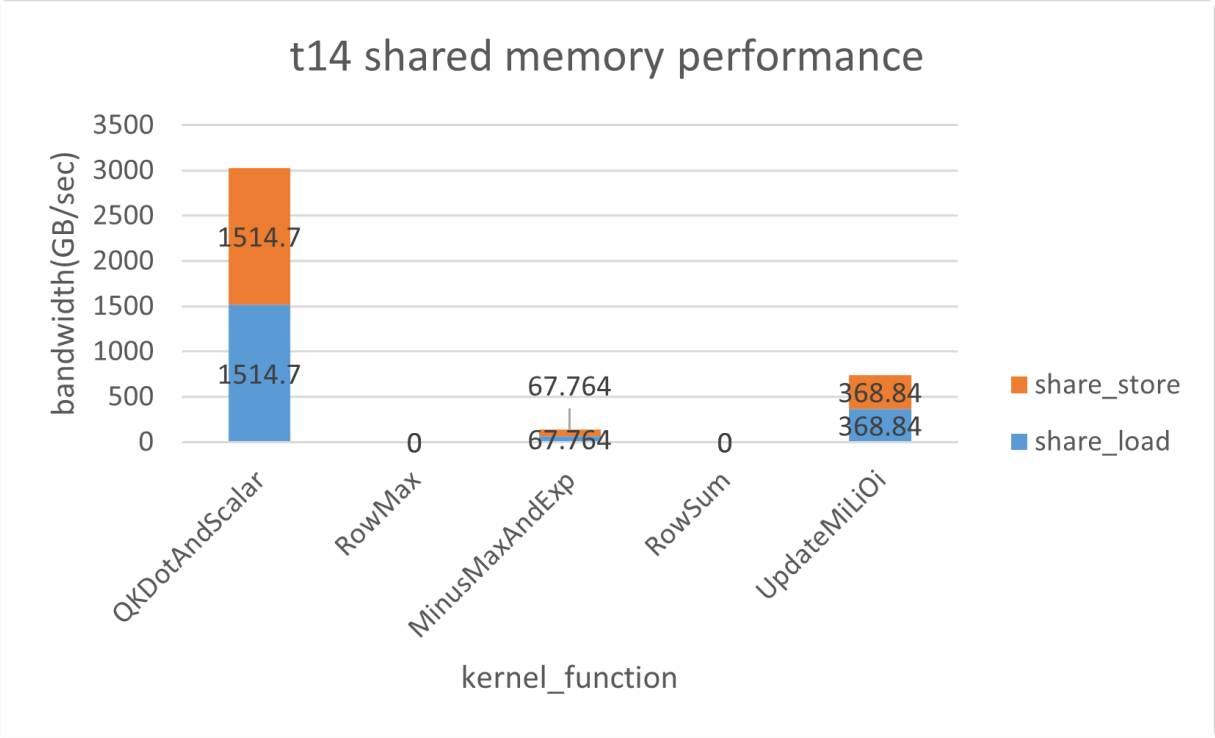
這裡很明顯shared memory反而是降低了performance，有可能是因為在執行計算時，要先把資料裝進shared memory(有些甚至要用迴圈裝入例如QKDotAndScalar)和要等待threads_synchronization的時間確保計算資料無誤，而這些過程在資料量很大時會嚴重增加運行時間，所以有可能導致整體效能不好。

```
int sharedMemSize_QKD = 2 * 32 * d * sizeof(float);
__global__ void QKDotAndScalar(float *q, float *k, float *out, int d, float
    int i = threadIdx.y;
    int j = threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= N || col >= N) return;

    extern __shared__ float shared_mem[];
    float* shared_q = shared_mem;
    float* shared_k = shared_q + 32 * d;

    #pragma unroll 32
    for (int t = 0; t < d; t++) {
        shared_q[i * d + t] = q[row * d + t];
        shared_k[j * d + t] = k[col * d + t];
    }
    __syncthreads();
    float value = 0.0F;

    for (int t = 0; t < d; t++) {
        value += shared_q[i * d + t] * shared_k[j * d + t];
    }
    out[row * N + col] = value * scalar;
}

int sharedMemSize_Min = 1024 * sizeof(float);
__global__ void MinusMaxAndExp(float *in, float *out, float *max_vals, int
    int i = threadIdx.y;
    int j = threadIdx.x;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row >= N || col >= N) return;
    extern __shared__ float shared_mem[];
    float *shared_in = shared_mem;
    shared_in[i*32+j] = in[row * N + col];
    __syncthreads();
    out[row * N + col] = exp(shared_in[i*32+j] - max_vals[row]);
}

    int sharedMemSize_Upd = (4 * 32 + N *32) * sizeof(float);
__global__ void UpdateMiLiOi(float *mi, float *li, float *oi, float *mij, f
    int i = threadIdx.y;
    int j = threadIdx.x;

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row >= N || col >= d) return;
    extern __shared__ float shared_mem[];
    float *shared_mi = shared_mem;
    float *shared_mij = shared_mi + 32;
    shared_mi[i] = mi[row];
    shared_mij[i] = mij[row];
    float *shared_li = shared_mij + 32;
    float *shared_lij = shared_li + 32;
    shared_li[i] = li[row];
    shared lij[i] = lij[row];
```

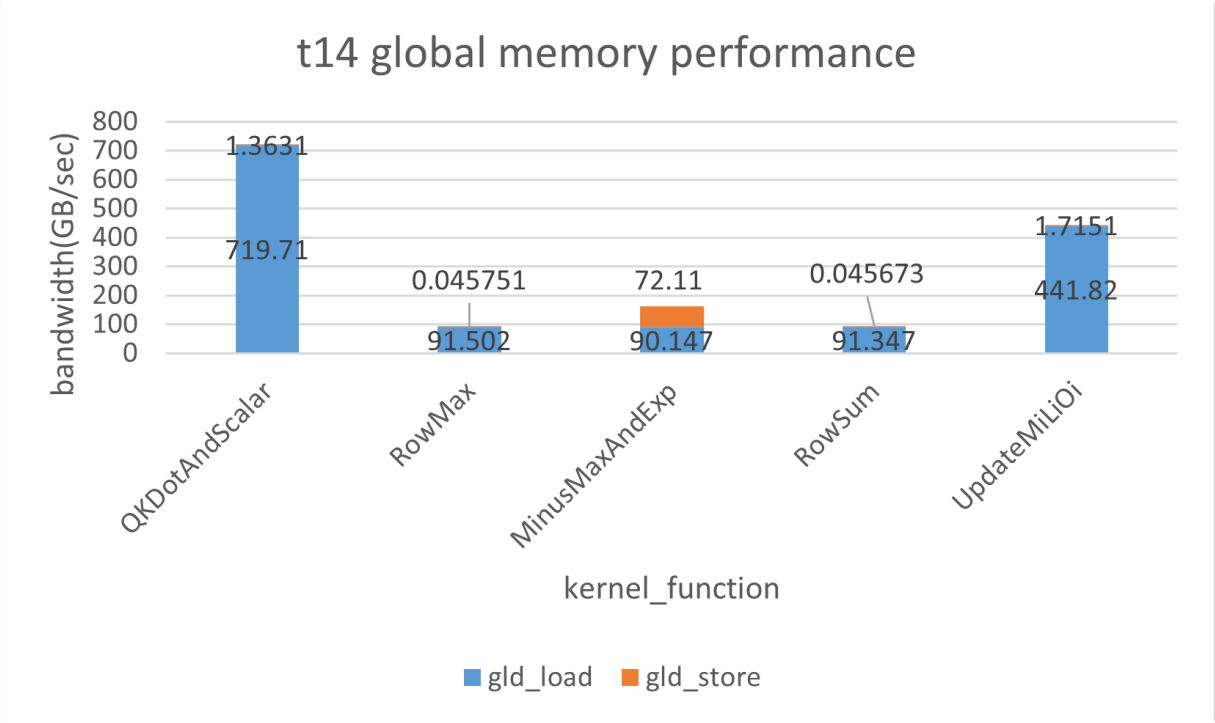
```
60      float *shared_pij = shared_lij + 32;
61      #pragma unroll 128
62      for (int t = 0; t < N; t++){
63          shared_pij[i * N + t] = pij[row * N + t];
64      }
65      __syncthreads();
66      float mi_new = _max(shared_mi[i], shared_mij[i]);
67      float li_new = exp(shared_mi[i] - mi_new) * shared_li[i] + exp(shared_m
68
69      float pv = 0.0F;
70      #pragma unroll 128
71      for (int t = 0; t < N; t++) {
72          pv += shared_pij[i * N + t] * vj[t * d + col];
73      }
74      oi[row * d + col] = (shared_li[i] * exp(shared_mi[i] - mi_new) * oi[row
75
76      if (col == 0) {  // Only update `mi` and `li` once per row
77          mi[row] = mi_new;
78          li[row] = li_new;
79      }
80  }
```

以下使用nvprof和t13測資去測量的Global/Shared memory data throughput(load and store):





下圖是上面沒用shared memory的程式拿來做比較:



可以發現用shared memory的global+shared memory的bandwidth還是有上升，但就是上述講的原因導致整體的performance無上升。

CUDA v.s. AMD

這裡原本要做cuda和amd的比較，因為在作業三的時候發現amd的IO以及記憶體操作都有比較快有明顯在大資料量時的速度提升，所以想說把本程式compile成amd，但好像答案是錯的所以整個運算

時間的計算有問題，但還是把實驗資料補上(用gettimeofday計算時間以及t14、t25測資):

t14	CUDA	AMD
I/O	0.47	0.461
memory	0.977	0.003
computati	0.029	0
total	1.611	0.463
t25	CUDA	AMD
I/O	0.166	0.261
memory	3.329	0.001
computati	0.001	0
total	3.802	0.261

Conclutions

本次作業發現在computation times裡有一些跟運算無關的事情會降低FLOPs，並且memory的相關運作會是時間佔的部分比很大所以寫程事實要注意記憶體存取等優化，然後是shared memory會在block裡資料大時很難去優化，所以可能可以去分多一點block去做運算或其他的分割方式讓shared memory可以執行，增加throughput和整體程式的效能。
本次作業讓我學到原來在資料量很大的時候去做不同分割滿足每個函數是一件不太容易的事情，尤其在優化時看到memory操作占最多時間的時候會有點不知道怎麼優化，因為memcpy和malloc的時間很難控制，幾乎無法優化可能還要重新書寫程式碼的架構和運算規則去優化，所以我還有很多要學習的地方~~~