기초 GPU 프로그래밍 HW2 보고서

20211584 장준영

1. 사용된 GPU

NVIDIA GeForce RTX 3070 Ti 8GB

2. 실행 결과

```
#define VAL_IDX 1 /* 0, 1, 2 */
#define TS 32
#define WPT 8
#define RTS (TS / WPT)

int _Arow[3] = {1024, 1 << 14, 1021},
    _Acol[3] = {2048, 1 << 12, 2039},
    _Brow[3] = {_Acol[0], _Acol[1], _Acol[2]},
    _Bcol[3] = {3200, 1 << 13, 3203},
    _Crow[3] = {_Arow[0], _Arow[1], _Arow[2]},
    _Ccol[3] = {_Bcol[0], _Bcol[1], _Bcol[2]};
```

각기 다른 세 가지의 크기를 VAL_IDX를 바꿔가면서 테스트할 수 있다. 첫 번째는 샘플 코드와 동일하게 적당히 큰 크기의 행렬이고, 두 번째는 메모리 초과가 나지 않는 선에서 host에서 충분한 시간을 들여야만 계산이 가능한 가장 큰 크기의 행렬이고, 세 번째는 각각의 행 열 크기가 네 자리 수의 소수로만 이루어진 행렬이다.

[1] VAL_IDX = 0

```
[1] Host time(double) = 1926.354370(ms) ------------------------------
[Absolute relative errors between h_A_flt and h_A_hf] average = 0.000169, maximum = 0.043153
[Absolute relative errors between h_B_flt and h_B_hf] average = 0.000169, maximum = 0.311828

[2] GPU time(CUDA Cores/float) = 10.755500(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000000, maximum = 0.000003

[3] GPU time(CUDA Cores/float/shared memory) = 7.726100(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000000, maximum = 0.000003

[4] GPU time(CUDA Cores/float/shared memory/More-Work-per-Thread) = 2.615800(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000000, maximum = 0.000003

[5] GPU time(Tensor Cores/half) = 1.972400(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half)] average = 0.000022, maximum = 0.000061

[6] GPU time(Tensor Cores/half/shared memory) = 1.095600(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half/shared memory)] average = 0.001421, maximum = 0.023716

[7] GPU time(cuBlas/float) = 1.746700(ms) ------------------------------
[Absolute relative errors(cuBlas/float)] average = 0.000000, maximum = 0.000003


C:₩Users₩JYJang₩Desktop₩univ₩7₩GPU₩hw2₩13.Mat_mult₩Mat_mult₩matmulTensor₩matmulTensor₩x64₩Release₩matmulT.exe(프로세스
 29184개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

[2] VAL_IDX = 1

```
[1] Host time(double) = 163011.125000(ms) ------------------------------
[Absolute relative errors between h_A_flt and h_A_hf] average = 0.000169, maximum = 1.000000
[Absolute relative errors between h_B_flt and h_B_hf] average = 0.000169, maximum = 1.000000

[2] GPU time(CUDA Cores/float) = 792.626587(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000001, maximum = 0.000005

[3] GPU time(CUDA Cores/float/shared memory) = 631.427429(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000001, maximum = 0.000005

[4] GPU time(CUDA Cores/float/shared memory/More-Work-per-Thread) = 248.797302(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000001, maximum = 0.000005

[5] GPU time(Tensor Cores/half) = 408.043091(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half)] average = 0.000045, maximum = 0.000078

[6] GPU time(Tensor Cores/half/shared memory) = 127.874802(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half/shared memory)] average = 0.002011, maximum = 0.024263

[7] GPU time(cuBlas/float) = 69.069000(ms) ------------------------------
[Absolute relative errors(cuBlas/float)] average = 0.000001, maximum = 0.000005


C:₩Users₩JYJang₩Desktop₩univ₩7₩GPU₩hw2₩13.Mat_mult₩Mat_mult₩matmulTensor₩matmulTensor₩x64₩Release₩matmulT.exe(프로세스
 1716개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

[3] VAL_IDX = 2

```
[1] Host time(double) = 1822.021973(ms) ------------------------------
[Absolute relative errors between h_A_flt and h_A_hf] average = 0.000169, maximum = 0.043153
[Absolute relative errors between h_B_flt and h_B_hf] average = 0.000169, maximum = 0.311828

[2] GPU time(CUDA Cores/float) = 10.809800(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000000, maximum = 0.000003

[3] GPU time(CUDA Cores/float/shared memory) = 8.035100(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.011303, maximum = 0.097202

[4] GPU time(CUDA Cores/float/shared memory/More-Work-per-Thread) = 3.303900(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000000, maximum = 0.000003

[5] GPU time(Tensor Cores/half) = 3.152100(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half)] average = 0.016497, maximum = 0.998408

[6] GPU time(Tensor Cores/half/shared memory) = 1.084600(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half/shared memory)] average = 0.016215, maximum = 1.000000

[7] GPU time(cuBlas/float) = 1.784300(ms) ------------------------------
[Absolute relative errors(cuBlas/float)] average = 0.000000, maximum = 0.000003


C:₩Users₩JYJang₩Desktop₩univ₩7₩GPU₩hw2₩13.Mat_mult₩Mat_mult₩matmulTensor₩matmulTensor₩x64₩Release₩matmulT.exe(프로세스
 10056개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 구현 방법

(※ 강의 자료에 구현이 되어있지 않거나 그 방법과 다르게 구현한 경우만 설명함.)

[1] MM_HOST

```
void MM_HOST(double *C, float *A, float *B, int Ay, int Ax, int Bx)
{
    for (int i = 0; i < Ay; i++)
    {
        for (int k = 0; k < Ax; k++)
        {
            double Aik = (double)A[i * Ax + k];
            for (int j = 0; j < Bx; j++)
            {
                C[i * Bx + j] += Aik * (double)B[k * Bx + j];
            }
        }
    }
}
```

기존의 host에서의 행렬곱 연산은 C의 한 요소를 계산하기 위해 A의 행과 B의 열 안의

전체 요소를 참조하는 방식이었다. 이 방식은 A와 B에 대한 메모리 접근이 많아 비효율적이기 때문에, A의 한 요소에 대해 이를 참조하는 C의 요소를 계산하는 방식으로 수정하였다. 이를 통해 메모리 접근을 줄이고, 실제 동작 속도를 30% 정도 감소시킬 수 있었다.

[2] MM_DEVICE_GM

```
__global__
void MM_DEVICE_GM(float *__restrict C, const float *__restrict A, const float *__restrict B,
                  int Ay, int Ax, int Bx)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

    if (ty >= Ay || tx >= Bx)
        return;

    float ctmp = 0.0f;
#pragma unroll 16
    for (int k = 0; k < Ax; k++)
    {
        ctmp += A[ty * Ax + k] * B[k * Bx + tx];
    }
    C[ty * Bx + tx] = ctmp;
}
```

제공된 코드에서 (1)C의 요소 값에 대해 global memory에 access 하지 않고 레지스터에 accumulation을 할 수 있도록 ctmp 변수를 선언하고 배열을 __restrict 타입으로 정의하였고, (2)전처리 옵션으로 루프 언롤링을 추가하였다.

[3] MM_DEVICE_SM_MWPT

More Work per Thread 기법이 적용된 이 함수는 강의 자료의 코드를 참고하지 않고, 확실한 이해를 위해 https://cnugteren.github.io/tutorial/pages/page5.html의 openCL 코드를 CUDA로 옮겨오는 것을 시도하였다. 구현에 있어 openCL과 CUDA의 가장 큰 차이점은 행렬 표현 방식으로, openCL 코드에선 column-major 방식을 사용하고 CUDA에선 row-major 방식을 사용한다. 이 차이점을 유념하여, 표현식 및 전역 변수 참조를 수정해가며 코드 이전을 진행하였다.

```
__global__
void MM_DEVICE_SM_MWPT(float *__restrict C, const float *__restrict A, const float *__restrict B,
                       int Ay, int Ax, int Bx)
{
    __shared__ float Atile[TS][TS];
    __shared__ float Btile[TS][TS];
    float accum[WPT] = {0.0f};

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x * blockDim.x;
    int by = blockIdx.y * blockDim.y * WPT;

    int Row = by + ty; // Row index for C
    int Col = bx + tx; // Column index for C
```

우선 openCL 코드에도 존재하듯 연산에 사용할 기본적인 변수를 선언한다. shared memory 영역의 타일과 값을 한번에 축적할 (레지스터의) accum, 쓰레드 인덱스 및 그리드 내 블락 오프셋, 현재 쓰레드가 위치한 C에서의 row와 column 번호 등을 계산한다.

```
for (int t = 0; t < (Ax + TS - 1) / TS; t++)
{
    // Load one tile of A and B into shared memory
    for (int w = 0; w < WPT; w++)
    {
        int Arow = Row + w * RTS;
        int Acol = t * TS + tx;
        int Brow = t * TS + ty + w * RTS;
        int Bcol = Col;
        if (Arow < Ay && Acol < Ax)
            Atile[ty + w * RTS][tx] = A[Arow * Ax + Acol];
        else
            Atile[ty + w * RTS][tx] = 0.0f;
        if (Brow < Ax && Bcol < Bx)
            Btile[ty + w * RTS][tx] = B[Brow * Bx + Bcol];
        else
            Btile[ty + w * RTS][tx] = 0.0f;
    }
    __syncthreads();

    // Perform the computation for a single tile
    for (int k = 0; k < TS; k++)
    {
        for (int w = 0; w < WPT; w++)
        {
            accum[w] += Atile[ty + w * RTS][k] * Btile[k][tx];
        }
    }
    __syncthreads();
}
```

이후 계산한 기본 변수 값을 바탕으로 Atile과 Btile을 로딩한다. 타일 로딩이 끝났다면 race condition을 없애기 위해 __syncthreads를 호출하고, 이후 accum에 값을 계산한다. 이 모든 과정에서 WPT만큼 반복하며 한 쓰레드가 여러 번의 일을 하는 것을 볼 수 있고, 같은 shared memory 영역의 값을 참조하는 계산을 한 번에 처리함으로써 memory access 횟수도 줄이는 것을 확인할 수 있다.

```
    // Store the final results in C
    for (int w = 0; w < WPT; w++)
    {
        int Cindex = (Row + w * RTS) * Bx + Col;
        if ((Row + w * RTS) < Ay && Col < Bx)
            C[Cindex] = accum[w];
    }
}
```

이후 레지스터에 누적한 값을 적절한 C의 인덱스에 할당한다. 타일링이 완전히 나누어 떨어지지 않는 구간에 대해서도 진행됐기 때문에, 경계 처리 과정을 추가하여 안정성을 높였다.

[4] MM_DEVICE_CUBLAS

```cpp
void MM_DEVICE_CUBLAS(float *C, float *A, float *B,
                      int Ay, int Ax, int Bx)
{
    cublasHandle_t handle;
    cublasCreate(&handle);

    const float alpha = 1.0f;
    const float beta = 0.0f;
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                Bx, Ay, Ax,
                &alpha, // 1
                B, Bx,
                A, Ax,
                &beta, // 0
                C, Bx);
}
```

cuBLAS 라이브러리를 사용했기 때문에 구현 자체는 아주 단순하다. 여기서 유념해야 할 점은, cuBLAS 라이브러리의 행렬 표현은 column-major 방식이라는 것이다. 따라서, C=A*B의 연산이더라도 곱셈 순서를 바꾸어 row-major 연산의 결과가 나올 수 있도록 해야 한다. 또, alpha와 beta는 C:=alpha*A*B+beta*C의 계수로, 1과 0으로 설정하면 기본적인 행렬곱 연산을 할 수 있다.

4. 결과 분석

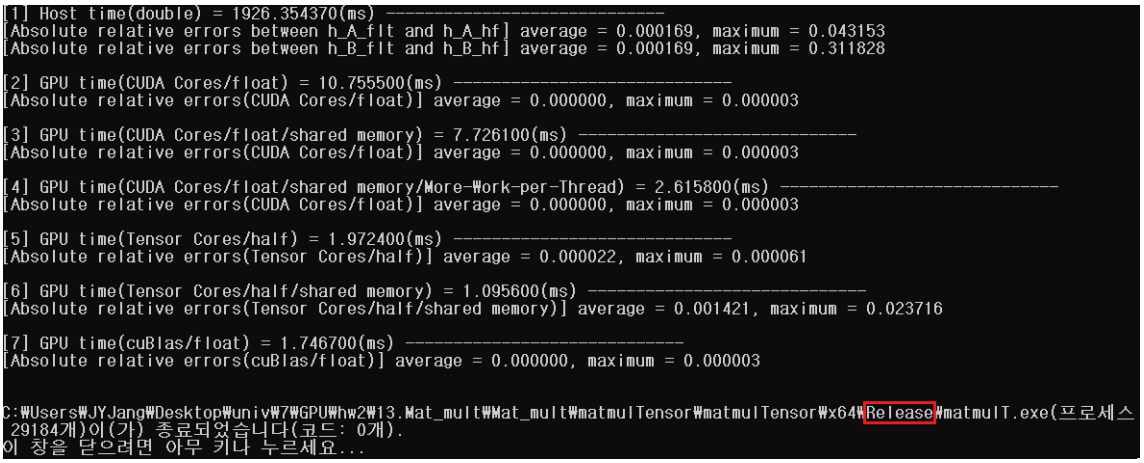
[1] 속도

```
[1] Host time(double) = 163011.125000(ms) ------------------------------
[Absolute relative errors between h_A_flt and h_A_hf] average = 0.000169, maximum = 1.000000
[Absolute relative errors between h_B_flt and h_B_hf] average = 0.000169, maximum = 1.000000

[2] GPU time(CUDA Cores/float) = 792.626587(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000001, maximum = 0.000005

[3] GPU time(CUDA Cores/float/shared memory) = 631.427429(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000001, maximum = 0.000005

[4] GPU time(CUDA Cores/float/shared memory/More-Work-per-Thread) = 248.797302(ms) ------------------------------
[Absolute relative errors(CUDA Cores/float)] average = 0.000001, maximum = 0.000005

[5] GPU time(Tensor Cores/half) = 408.043091(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half)] average = 0.000045, maximum = 0.000078

[6] GPU time(Tensor Cores/half/shared memory) = 127.874802(ms) ------------------------------
[Absolute relative errors(Tensor Cores/half/shared memory)] average = 0.002011, maximum = 0.024263

[7] GPU time(cuBlas/float) = 69.069000(ms) ------------------------------
[Absolute relative errors(cuBlas/float)] average = 0.000001, maximum = 0.000005


C:₩Users₩JYJang₩Desktop₩univ₩7₩GPU₩hw2₩13.Mat_mult₩Mat_mult₩matmulTensor₩matmulTensor₩x64₩Release₩matmulT.exe(프로세스
 1716개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

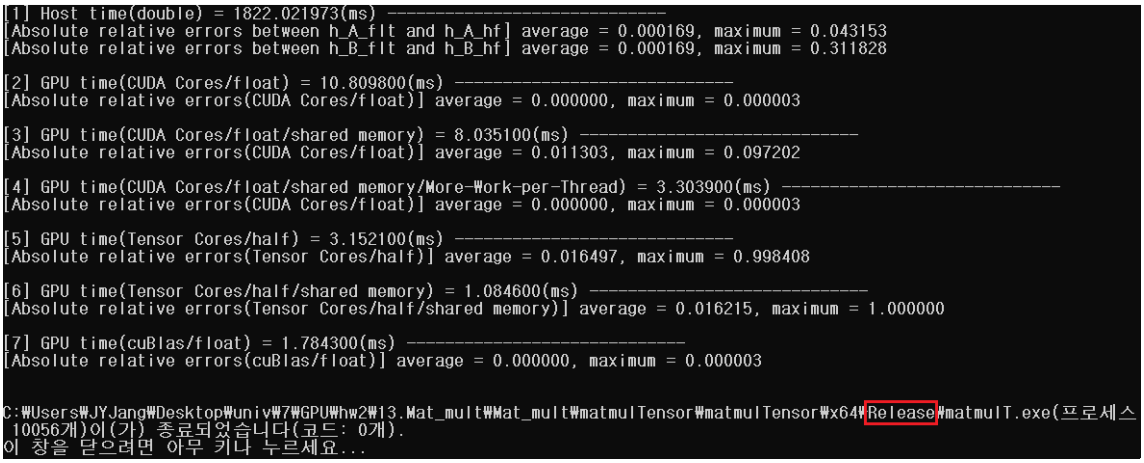
속도 차이를 가장 잘 확인할 수 있는 VAL_IDX=1인 경우의 결과이다. 호스트에서의 계산은 O(n^3)의 시간 복잡도에서 미리 예상할 수 있듯이 굉장히 오랜 시간이 걸린다. 디바이스에서 병렬적으로 계산한 시간은 호스트에서 계산한 시간의 최대 0.5%로, 엄청난 시간 절감을 한 것을 볼 수 있다. 라이브러리를 사용하지 않고 직접 구현한 [2], [3], [4]번 방법은 순차적으로 더 깊은 최적화를 진행했고, 그에 따라 시간이 점점 줄어드는 것을

볼 수 있다. 또, 행렬 크기가 커지면 커질수록 그에 따른 쓰레드 생성과 memory access 가 exponential하게 증가하므로, 이를 줄이고 재활용할수록 속도 측면에서 큰 효율을 얻을 수 있었다. Tensorcore를 사용한 연산에서 A와 B는 half 타입의 행렬로, 낮은 precision 으로 인해 빠른 속도를 얻을 수 있었다. 그 외에도 행렬곱 연산에 최적화된 하드웨어를 이용하기 때문에 상당히 빠른 속도로 계산을 마치는 모습을 볼 수 있다. 마지막으로 cuBLAS는 독보적으로 빠른 속도를 보였는데, 극도로 정밀하게 구현된 내부 연산 덕분인 것 같다.

[2] 오차



오차 역시 일반적인 상황에 대한 값을 가장 잘 보이고 있는 VAL_IDX=0의 경우를 먼저 살펴본다. [1]에서 완전히 동일한 값을 바탕으로 자료형만 다른 두 행렬을 비교했을 때, precision loss로 인해 약간의 오차가 발생하는 것을 볼 수 있다. 이를 바탕으로, float 타입으로 연산을 진행한 [2], [3], [4], [7]번은 오차가 거의 발생하지 않고, half 타입으로 연산을 진행한 [5], [6]번은 약간의 오차가 발생함을 알 수 있다.



Tensorcore를 사용할 때 타일 크기로 나누어떨어지지 않는 수가 입력되었을 때를 확인하기 위한 경우이다(VAL_IDX=2). [5], [6]번의 오차를 보면 최대 오차 값이 거의 1에 육박하는 것을 볼 수 있는데, 이는 타일 사이즈에 맞춰 Ax/16과 Bx/16을 호출하고 이에 맞게 데이터를 로딩하는데에서 발생하는 오류이다. CUDA WMMA API를 사용하는 경우 타일의 크기는 고정되어 있기 때문에, 함수 내부 예외 처리보단 기존 행렬에 패딩을 추가해 타일 크기의 배수가 되도록 하는 것이 적절하다.