

1 (50점) Matrix Multiplication Single GPU

- 자신의 병렬화 방식에 대한 설명.

1) GPU 사용

```
void matmul(const float *A, const float *B, float *C, int M, int N, int K) {  
  
    // Upload A and B matrix to every GPU  
    CUDA_CALL(cudaMemcpy(a_d, A, M * K * sizeof(float), cudaMemcpyHostToDevice));  
    CUDA_CALL(cudaMemcpy(b_d, B, K * N * sizeof(float), cudaMemcpyHostToDevice));  
  
    // // // Launch kernel on every GPU  
    dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE, 1);  
    dim3 gridDim((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (M + BLOCK_SIZE - 1) / BLOCK_SIZE, 1);  
  
    matmul_kernel<<<gridDim, blockDim>>>(a_d, b_d, c_d, M, N, K);  
  
    // dim3 threads(BLOCK_SIZE, BLOCK_SIZE);  
    // dim3 blocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (M + BLOCK_SIZE - 1) / BLOCK_SIZE);  
  
    // matmul_kernel_vec4<<<blocks, threads>>>(a_d, b_d, c_d, M, N, K);  
  
    cudaError_t err = cudaGetLastError();  
    if (err != cudaSuccess) {  
        printf("Kernel Launch Error: %s\n", cudaGetErrorString(err));  
    }  
  
    CUDA_CALL(cudaDeviceSynchronize());  
  
    // Download C matrix from GPUs  
    CUDA_CALL(cudaMemcpy(C, c_d, M * N * sizeof(float), cudaMemcpyDeviceToHost));  
  
    // DO NOT REMOVE; NEEDED FOR TIME MEASURE  
    CUDA_CALL(cudaDeviceSynchronize());  
}  
  
void matmul_initialize(int M, int N, int K) {
```

CUDA GPU를 통해서 빠른 처리를 시도.

- 성능 최적화를 위한 적용한 방법 및 고려 사항들에 대한 논의.

1) Tiling

```
__shared__ float tileA[BLOCK_SIZE][BLOCK_SIZE];  
__shared__ float tileB[BLOCK_SIZE][BLOCK_SIZE];
```

fileA를 block_size * block_size의 float pointer로도 해봤지만 2dim array로 하는 게 더 빠르다.

2) Padding

```

// Loop over tiles of A and B required for C[row, col]
for (int t = 0; t < (K + BLOCK_SIZE - 1) / BLOCK_SIZE; t++) {
    // Load elements into shared memory (if within bounds)
    if (row < M && t * BLOCK_SIZE + threadIdx.x < K) {
        tileA[threadIdx.y][threadIdx.x] = A[row * K + t * BLOCK_SIZE + threadIdx.x];
    } else {
        tileA[threadIdx.y][threadIdx.x] = 0.0f;
    }

    if (col < N && t * BLOCK_SIZE + threadIdx.y < K) {
        tileB[threadIdx.y][threadIdx.x] = B[(t * BLOCK_SIZE + threadIdx.y) * N + col];
    } else {
        tileB[threadIdx.y][threadIdx.x] = 0.0f;
    }
}

```

4096사이즈라 패딩이 필요없지만 안정성을 위해서 첨부. if문을 제거하였을 때 성능이 두드러지게 빨라지지도 않아서 추가했다.

3) syncthreads and sum -> assign

```

// Synchronize to ensure all threads have loaded their tiles
__syncthreads();

// Compute partial dot product for the current tile
for (int k = 0; k < BLOCK_SIZE; k++) {
    sum += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];
}

// Synchronize before loading the next tile
__syncthreads();

// Write the result to the output matrix (if within bounds)
if (row < M && col < N) {
    C[row * N + col] = sum;
}

```

기존 행렬곱처럼 A의 행 * B의 열을 곱해 모두 더하는 방식으로 행렬곱을 수행했다.

- matmul single.cu의 각 부분에 대한 설명. matmul initialize, matmul, matmul finalize 함수 각각 에서 사용하는 CUDA API 및 각 API에 대한 간략한 설명. (API 당 한문장이면 충분).

```

void matmul_initialize(int M, int N, int K) {
    size_t a_size = M * (K / 4) * sizeof(float4); // M x (K / 4) float4 elements
    size_t b_size = (K / 4) * (N / 4) * sizeof(float4); // (K / 4) x (N / 4) float4 elements
    size_t c_size = M * (N / 4) * sizeof(float4); // M x (N / 4) float4 elements

    int num_devices;
    // Only root process do something
    CUDA_CALL(cudaGetDeviceCount(&num_devices));

    if (num_devices <= 0) {
        printf("No CUDA device found. Aborting\n");
        exit(1);
    }

    // Allocate device memory

    CUDA_CALL(cudaMalloc(&a_d, M * K * sizeof(float)));
    CUDA_CALL(cudaMalloc(&b_d, K * N * sizeof(float)));
    CUDA_CALL(cudaMalloc(&c_d, M * N * sizeof(float)));
}

```

CudaGetDeviceCount: 사용가능한 GPU 개수를 반환

CudaMalloc: a_d라는 변수에 $M * K * \text{sizeof(float)}$ 크기의 GPU단 메모리를 할당.

```

void matmul(const float *A, const float *B, float *C, int M, int N, int K) {
    // Upload A and B matrix to every GPU
    CUDA_CALL(cudaMemcpy(a_d, A, M * K * sizeof(float), cudaMemcpyHostToDevice));
    CUDA_CALL(cudaMemcpy(b_d, B, K * N * sizeof(float), cudaMemcpyHostToDevice));

    // // // Launch kernel on every GPU
    dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE, 1);
    dim3 gridDim((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (M + BLOCK_SIZE - 1) / BLOCK_SIZE, 1);

    matmul_kernel<<<gridDim, blockDim>>>(a_d, b_d, c_d, M, N, K);

    // dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    // dim3 blocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (M + BLOCK_SIZE - 1) / BLOCK_SIZE);

    // matmul_kernel_vec4<<<blocks, threads>>>(a_d, b_d, c_d, M, N, K);

    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Kernel Launch Error: %s\n", cudaGetErrorString(err));
    }

    CUDA_CALL(cudaDeviceSynchronize());

    // Download C matrix from GPUs
    CUDA_CALL(cudaMemcpy(C, c_d, M * N * sizeof(float), cudaMemcpyDeviceToHost));

    // DO NOT REMOVE; NEEDED FOR TIME MEASURE
    CUDA_CALL(cudaDeviceSynchronize());
}

```

CudaMemcpy: Host(CPU) <-> Device(GPU) 간 메모리 전송. cudaMemcpyHostToDevice는 host -> device, cudaMemcpyDeviceToHost는 host <- device

cudaDeviceSynchronize: Device를 동기화. 프로세스가 종료될때까지 기다림.

```
void matmul_finalize() {  
  
    // Free GPU memory  
    CUDA_CALL(cudaFree(a_d));  
    CUDA_CALL(cudaFree(b_d));  
    CUDA_CALL(cudaFree(c_d));  
}
```

cudaFree: GPU에 잡힌 메모리 해제

- OpenCL에 비해 CUDA의 장점 1문장, CUDA에 비해 OpenCL의 장점 1문장 (총 2문장).

CUDA: NVIDIA GPU에 최적화. 명령어가 간단해서 stream 짜기가 편함

OpenCL; 범용성 있어서 여러 heterodevice programming에 이용됨

- 자신이 적용한 최적화 방식을 정리하고, 각각에 대한 성능 실험 결과. (Matrix multiplication은 프로젝트 에도 핵심적인 부분이므로 해당 실험을 적극적으로 해보길 권장함.)

다음의 명령어로 확인해 보았다. ./run.sh 4096 4096 4096 -n 5 -v. BLOCK_SIZE = 32일 때 가장 성능이 좋았다. 64 이상은 메모리에 올려지지 않았다. (GFLOPS는 소숫점단위는 버림)

BLOCK_SIZE = 4, GFLOPS: 327

BLOCK_SIZE = 8, GFLOPS: 875

BLOCK_SIZE = 16, GFLOPS: 1238

BLOCK_SIZE = 32, GFLOPS: 1410

BLOCK_SIZE = 64, Kernel Launch Error: invalid configuration argument

2 (50점) Matrix Multiplication Multi GPU

- 자신의 병렬화 방식에 대한 설명.

1) One CPU thread – one GPU device

```

void matmul(const float *A, const float *B, float *C, int M, int N, int K) {
    pthread_t threads[MAX_NUM_GPU];
    ThreadData thread_data[MAX_NUM_GPU];

    // Create and launch threads
    for (int i = 0; i < num_devices; i++) {
        thread_data[i].device_id = i;
        thread_data[i].A = A;
        thread_data[i].B = B;
        thread_data[i].C = C;
        thread_data[i].M = M;
        thread_data[i].N = N;
        thread_data[i].K = K;
        thread_data[i].Mbegin = Mbegin[i];
        thread_data[i].Mend = Mend[i];

        pthread_create(&threads[i], NULL, threaded_matmul, &thread_data[i]);
        // pthread_create(&threads[i], NULL, temp, &thread_data[i]);
    }

    // Join threads
    for (int i = 0; i < num_devices; i++) {
        pthread_join(threads[i], NULL);
    }
}

```

하나의 matmul 연산 당 하나의 CPU 스레드를 할당하여 병렬로 이루어지게 했다.

2) In-thread initialization

```

void *threaded_matmul(void *arg) {
    ThreadData *data = (ThreadData *)arg;

    // Set GPU for this thread
    CUDA_CALL(cudaSetDevice(data->device_id));

    // Allocate memory on the GPU
    float *a_d, *b_d, *c_d;
    CUDA_CALL(cudaMalloc(&a_d, (data->Mend - data->Mbegin) * data->K * sizeof(float)));
    CUDA_CALL(cudaMalloc(&b_d, data->K * data->N * sizeof(float)));
    CUDA_CALL(cudaMalloc(&c_d, (data->Mend - data->Mbegin) * data->N * sizeof(float)));

    // Copy matrices to the GPU
    CUDA_CALL(cudaMemcpy(a_d, data->A + data->Mbegin * data->K,
                        (data->Mend - data->Mbegin) * data->K * sizeof(float),
                        cudaMemcpyHostToDevice));
    CUDA_CALL(cudaMemcpy(b_d, data->B, data->K * data->N * sizeof(float),
                        cudaMemcpyHostToDevice));

    // Configure grid and block dimensions
    dim3 blockDim(TILE_SIZE, TILE_SIZE, 1);
    dim3 gridDim((data->N + TILE_SIZE - 1) / TILE_SIZE,
                (data->Mend - data->Mbegin + TILE_SIZE - 1) / TILE_SIZE,
                1);

    // Launch the kernel
    matmul_kernel<<<gridDim, blockDim>>>(a_d, b_d, c_d, data->M, data->N, data->K);

    // Synchronize GPU
    CUDA_CALL(cudaDeviceSynchronize());

    // Copy the result matrix back to host memory
    CUDA_CALL(cudaMemcpy(data->C + data->Mbegin * data->N, c_d,
                        (data->Mend - data->Mbegin) * data->N * sizeof(float),
                        cudaMemcpyDeviceToHost));

    // Free GPU memory
    CUDA_CALL(cudaFree(a_d));
    CUDA_CALL(cudaFree(b_d));
    CUDA_CALL(cudaFree(c_d));
}

```

CPU 스레드 안에서 malloc, memcpy를 실행해서 병렬로 이루어지게 했다. Free도 스레드 안에서 이루어지도록 작성함.

3) No finalize

```

void matmul_finalize() {
}

```

Thread join하기 전에 free했으므로 finalize할 것이 없다.

4) 동일한 kernel operation 사용

```

__global__ void matmul_kernel(const float *A, const float *B, float *C) {
    // Shared memory for storing tiles of A and B
    __shared__ float tileA[TILE_SIZE][TILE_SIZE];
    __shared__ float tileB[TILE_SIZE][TILE_SIZE];

    // Calculate thread row and column within the output matrix
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    // Accumulator for the dot product
    float sum = 0.0f;

    // Loop over tiles of A and B required for C[row, col]
    for (int t = 0; t < (K + TILE_SIZE - 1) / TILE_SIZE; t++) {
        // Load elements into shared memory (if within bounds)
        if (row < M && t * TILE_SIZE + threadIdx.x < K) {
            tileA[threadIdx.y][threadIdx.x] = A[row * K + t * TILE_SIZE + threadIdx.x];
        } else {
            tileA[threadIdx.y][threadIdx.x] = 0.0f;
        }

        if (col < N && t * TILE_SIZE + threadIdx.y < K) {
            tileB[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * N + col];
        } else {
            tileB[threadIdx.y][threadIdx.x] = 0.0f;
        }

        // Synchronize to ensure all threads have loaded their tiles
        __syncthreads();

        // Compute partial dot product for the current tile
        for (int k = 0; k < TILE_SIZE; k++) {
            sum += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];
        }

        // Synchronize before loading the next tile
        __syncthreads();
    }

    // Write the result to the output matrix (if within bounds)
    if (row < M && col < N) {
        C[row * N + col] = sum;
    }
}

```

Matmul_single과 동일.

- 성능 최적화를 위한 적용한 방법 및 고려 사항들에 대한 논의.

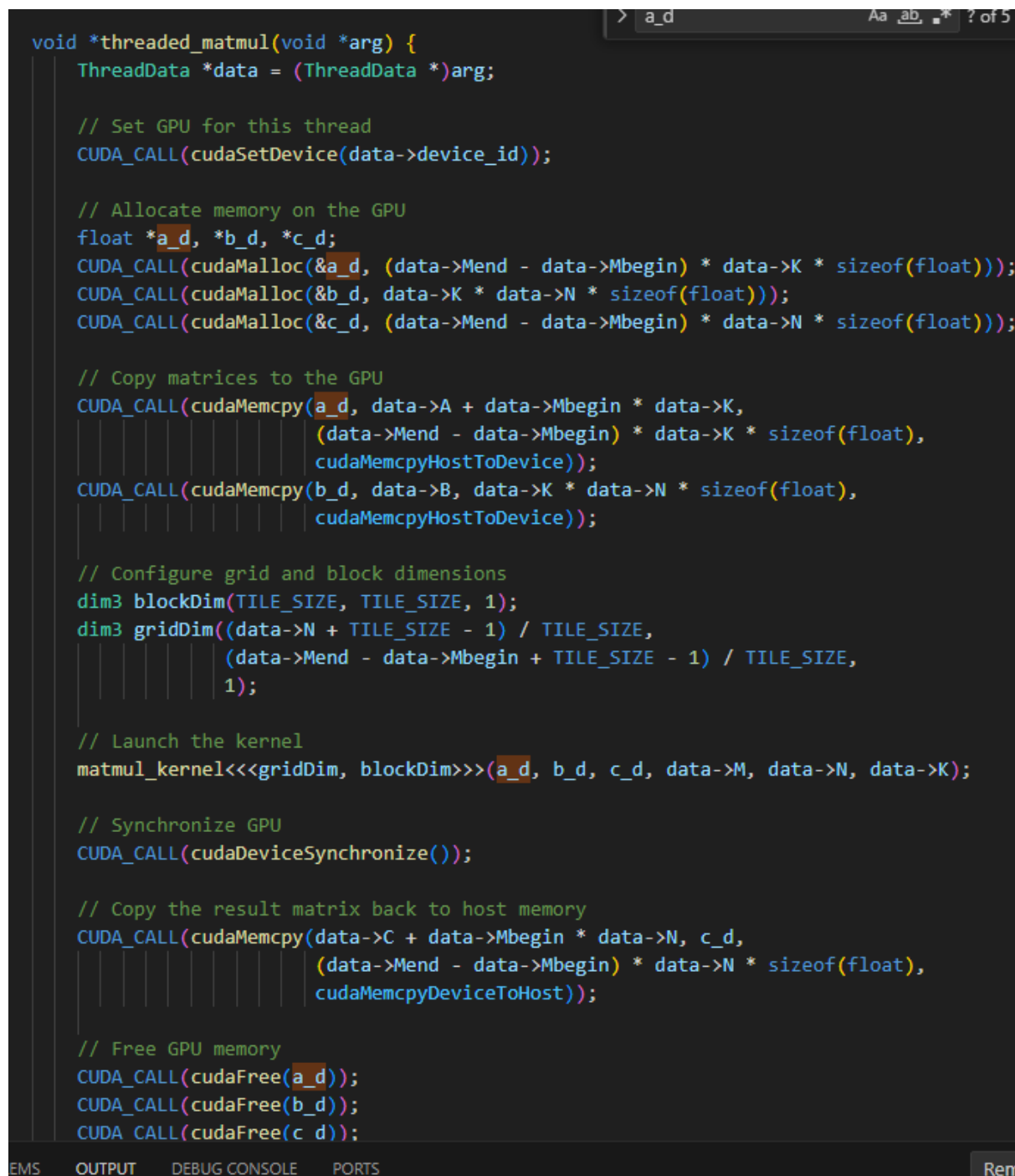
Matmul_single과 동일하게 2d tiling, syncthread, summation strategy 등을 적용했다.

- matmul_multi.cu의 각 부분에 대한 설명.

matmul_initialize, matmul, matmul_finalize 함수 각각에 서 사용하는 CUDA API 및 각 API에 대한 간략한 설명. (API 당 한문장이면 충분).

Matmul_initialize는 건들지 않음(matmul_single과 동일)

Matmul_finalize는 빈 값(thread_join하기 이전에 모든 작업 수행)



```
void *threaded_matmul(void *arg) {
    ThreadData *data = (ThreadData *)arg;

    // Set GPU for this thread
    CUDA_CALL(cudaSetDevice(data->device_id));

    // Allocate memory on the GPU
    float *a_d, *b_d, *c_d;
    CUDA_CALL(cudaMalloc(&a_d, (data->Mend - data->Mbegin) * data->K * sizeof(float)));
    CUDA_CALL(cudaMalloc(&b_d, data->K * data->N * sizeof(float)));
    CUDA_CALL(cudaMalloc(&c_d, (data->Mend - data->Mbegin) * data->N * sizeof(float)));

    // Copy matrices to the GPU
    CUDA_CALL(cudaMemcpy(a_d, data->A + data->Mbegin * data->K,
        (data->Mend - data->Mbegin) * data->K * sizeof(float),
        cudaMemcpyHostToDevice));
    CUDA_CALL(cudaMemcpy(b_d, data->B, data->K * data->N * sizeof(float),
        cudaMemcpyHostToDevice));

    // Configure grid and block dimensions
    dim3 blockDim(TILE_SIZE, TILE_SIZE, 1);
    dim3 gridDim((data->N + TILE_SIZE - 1) / TILE_SIZE,
        (data->Mend - data->Mbegin + TILE_SIZE - 1) / TILE_SIZE,
        1);

    // Launch the kernel
    matmul_kernel<<<gridDim, blockDim>>>(a_d, b_d, c_d, data->M, data->N, data->K);

    // Synchronize GPU
    CUDA_CALL(cudaDeviceSynchronize());

    // Copy the result matrix back to host memory
    CUDA_CALL(cudaMemcpy(data->C + data->Mbegin * data->N, c_d,
        (data->Mend - data->Mbegin) * data->N * sizeof(float),
        cudaMemcpyDeviceToHost));

    // Free GPU memory
    CUDA_CALL(cudaFree(a_d));
    CUDA_CALL(cudaFree(b_d));
    CUDA_CALL(cudaFree(c_d));
}
```

cudaSetDevice: n번째 device를 사용. 이후 코드는 모두 해당 GPU에서 이루어짐.

Cudamalloc, cudamemcpy, cudafree는 앞서 설명했으므로 생략함.

- 자신이 적용한 최적화 방식을 정리하고, 각각에 대한 성능 실험 결과. (Matrix multiplication은 프로젝트 에도 핵심적인 부분이므로 해당 실험을 적극적으로 해보길 권장함.)

다음의 명령어로 확인해 보았다. `./run.sh 4096 4096 4096 -n 5 -v. BLOCK_SIZE = 8` 일 때 가장 성능이 좋았으나, 실험을 반복할 때마다 GFLOPS 값이 편차가 크게 나왔다(최대 2600). 1500GFLOPS 정도 나오면 소요시간이 0.09초 정도 소요되는데, 아마 스레드를 시작하고 종료할 때 발생하는 스레드 간의 시행속도 차이가 영향을 크게 미치는 것으로 보인다. 64 이상은 메모리에 올려지지 않았다. (GFLOPS는 소숫점단위는 버림)

BLOCK_SIZE = 4, GFLOPS: 880

BLOCK_SIZE = 8, GFLOPS: 1549

BLOCK_SIZE = 16, GFLOPS: 1505

BLOCK_SIZE = 16 GFLOPS: 1546 Kernel Launch Error: invalid configuration argument