HW6

2024-27759 인문대학 언어학과 장효형

- 자신의 병렬화 방식에 대한 설명.

1) MPI nodes

```
void matmul_initialize(int M, int N, int K) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_gpu = M / (NUM_GPUS * 4);  // Total 4 nodes, each with 4 GPUs

    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaStreamCreate(&streams[i]);

        int local_rows = rows_per_gpu;
        size_t size_A = local_rows * K * sizeof(float);
        size_t size_B = K * N * sizeof(float);
        size_t size_C = local_rows * N * sizeof(float);

        cudaMalloc((void **)&d_A[i], size_A);
        cudaMalloc((void **)&d_B[i], size_B);
        cudaMalloc((void **)&d_C[i], size_C);

        block = dim3(BLOCK_SIZE, BLOCK_SIZE);
        grid = dim3((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (local_rows + BLOCK_SIZE - 1) / BLOCK_SIZE);
    }
}
```

본 캡쳐에는 나와있지 않지만, 4개의 노드 모두를 사용하였다.

2) Matmul

```
for (int i = 0; i < NUM_GPUS; i++) {
    cudaSetDevice(i);
    size_t size_B = K * N * sizeof(float);
    cudaMemcpyAsync(d_B[i], host_B, size_B, cudaMemcpyHostToDevice, streams[i]);
}


// Divide A and perform computation on each GPU
#pragma omp parallel for schedule(static) num_threads(num_gpus)
for (int i = 0; i < NUM_GPUS; i++) {
    cudaSetDevice(i);
    size_t size_A = rows_per_gpu * K * sizeof(float);
    size_t size_C = rows_per_gpu * N * sizeof(float);

    // Copy a chunk of A to each GPU asynchronously
    cudaMemcpyAsync(d_A[i], local_A + i * rows_per_gpu * K, size_A, cudaMemcpyHostToDevice, streams[i]);

    // Launch the kernel asynchronously
    matmul_kernel<<<grid, block, 0, streams[i]>>>(d_A[i], d_B[i], d_C[i], rows_per_gpu, N, K);

    // Copy result back to host asynchronously
    cudaMemcpyAsync(host_C + i * rows_per_gpu * N, d_C[i], size_C, cudaMemcpyDeviceToHost, streams[i]);
}

// Wait for all GPUs to finish
for (int i = 0; i < NUM_GPUS; i++) {
    cudaSetDevice(i);
    cudaStreamSynchronize(streams[i]);
}
```

Using 4 gpus, to compute operation parallelly

- 성능 최적화를 위한 적용한 방법 및 고려 사항들에 대한 논의.

3) Matmul_initialize, matmul_finalize

```
void matmul_initialize(int M, int N, int K) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_gpu = M / (NUM_GPUS * 4);  // Total 4 nodes, each with 4 GPUs

    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaStreamCreate(&streams[i]);

        int local_rows = rows_per_gpu;
        size_t size_A = local_rows * K * sizeof(float);
        size_t size_B = K * N * sizeof(float);
        size_t size_C = local_rows * N * sizeof(float);

        cudaMalloc((void **)&d_A[i], size_A);
        cudaMalloc((void **)&d_B[i], size_B);
        cudaMalloc((void **)&d_C[i], size_C);

        block = dim3(BLOCK_SIZE, BLOCK_SIZE);
        grid = dim3((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (local_rows + BLOCK_SIZE - 1) / BLOCK_SIZE);
    }
}
```

```
// Finalization function: free memory and destroy streams
void matmul_finalize() {
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaFree(d_A[i]);
        cudaFree(d_B[i]);
        cudaFree(d_C[i]);
        cudaStreamDestroy(streams[i]);
    }
    cudaDeviceReset();
}
```

putting redundent initializing(memory allocation) and finalizing(freeing memory) code outside of matmul, to reduce computation time.

4) Paddig

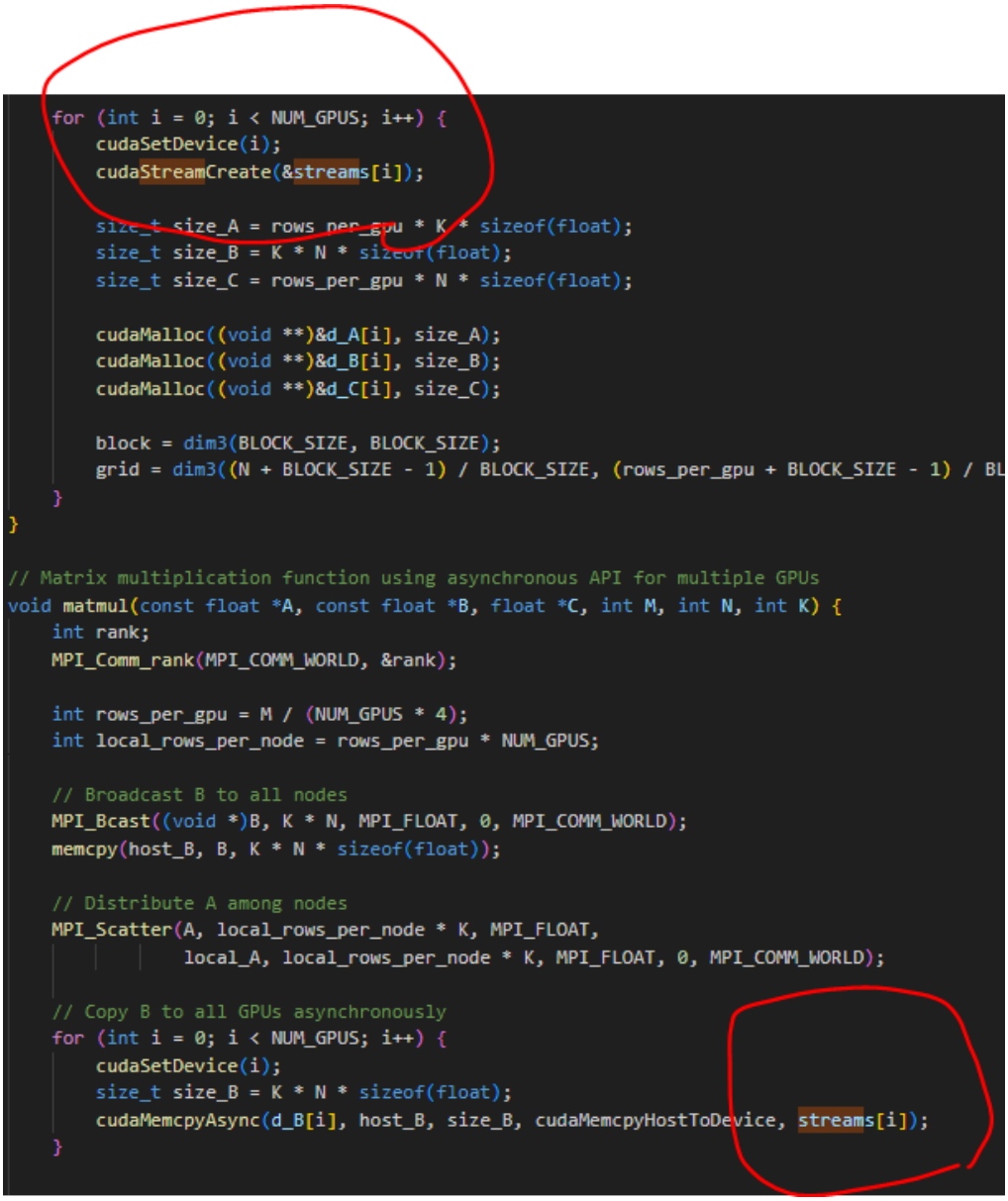
```cuda
__global__ void matmul_kernel(const float *A, const float *B, float *C, int M, int N, int K) {
    __shared__ float tileA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float tileB[BLOCK_SIZE][BLOCK_SIZE];

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

    float sum = 0.0f;

    for (int t = 0; t < (K + BLOCK_SIZE - 1) / BLOCK_SIZE; t++) {
        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;
        }

        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; k++) {
            sum += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < M && col < N) {
        C[row * N + col] = sum;
    }
}
```

Put padding in mamul_kernle, to be able for a arbitrary matrix to run in the code. We don't need the padding code in current settings, but this is appended for theh safety of executions

5) Tiling

   The matrix are tiled into BLOCK_SIZE, which operates as a hyperparemeter

6) Streams

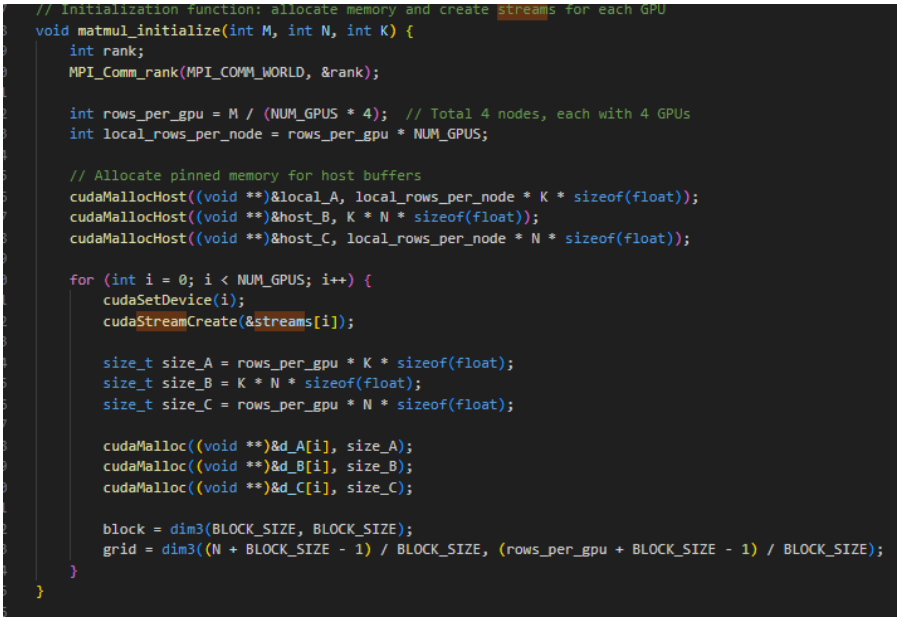


```
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaStreamCreate(&streams[i]);

        size_t size_A = rows_per_gpu * K * sizeof(float);
        size_t size_B = K * N * sizeof(float);
        size_t size_C = rows_per_gpu * N * sizeof(float);

        cudaMalloc((void **)&d_A[i], size_A);
        cudaMalloc((void **)&d_B[i], size_B);
        cudaMalloc((void **)&d_C[i], size_C);

        block = dim3(BLOCK_SIZE, BLOCK_SIZE);
        grid = dim3((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (rows_per_gpu + BLOCK_SIZE - 1) / BL
    }
}

// Matrix multiplication function using asynchronous API for multiple GPUs
void matmul(const float *A, const float *B, float *C, int M, int N, int K) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_gpu = M / (NUM_GPUS * 4);
    int local_rows_per_node = rows_per_gpu * NUM_GPUS;

    // Broadcast B to all nodes
    MPI_Bcast((void *)B, K * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    memcpy(host_B, B, K * N * sizeof(float));

    // Distribute A among nodes
    MPI_Scatter(A, local_rows_per_node * K, MPI_FLOAT,
                local_A, local_rows_per_node * K, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // Copy B to all GPUs asynchronously
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        size_t size_B = K * N * sizeof(float);
        cudaMemcpyAsync(d_B[i], host_B, size_B, cudaMemcpyHostToDevice, streams[i]);
    }
```

In each GPU devices of the node, I create cudaStreams to perform memory copy asynchronously

• matmul.c의 각 부분에 대한 설명. matmul initialize, matmul, matmul finalize 함수 각각에서 사용 하는 CUDA API 및 각 API에 대한 간략한 설명. (API 당 한문장이면 충분).

1) Matmul_initialize



```
// Initialization function: allocate memory and create streams for each GPU
void matmul_initialize(int M, int N, int K) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_gpu = M / (NUM_GPUS * 4);  // Total 4 nodes, each with 4 GPUs
    int local_rows_per_node = rows_per_gpu * NUM_GPUS;

    // Allocate pinned memory for host buffers
    cudaMallocHost((void **)&local_A, local_rows_per_node * K * sizeof(float));
    cudaMallocHost((void **)&host_B, K * N * sizeof(float));
    cudaMallocHost((void **)&host_C, local_rows_per_node * N * sizeof(float));

    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaStreamCreate(&streams[i]);

        size_t size_A = rows_per_gpu * K * sizeof(float);
        size_t size_B = K * N * sizeof(float);
        size_t size_C = rows_per_gpu * N * sizeof(float);

        cudaMalloc((void **)&d_A[i], size_A);
        cudaMalloc((void **)&d_B[i], size_B);
        cudaMalloc((void **)&d_C[i], size_C);

        block = dim3(BLOCK_SIZE, BLOCK_SIZE);
        grid = dim3((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (rows_per_gpu + BLOCK_SIZE - 1) / BLOCK_SIZE);
    }
}
```

MPI_Comm_rank: get rank and save the rank to a variable('rank')

cudaMallocHost: memory alloc to pinned memroy, of designated size

cudaSetdevice: set which devices to run. Every following code is executed in the devices

cudaMalloc: memory allocation to GPU

2) Matmul

```
void matmul(const float *A, const float *B, float *C, int M, int N, int K) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_gpu = M / (NUM_GPUS * 4);
    int local_rows_per_node = rows_per_gpu * NUM_GPUS;

    // Broadcast B to all nodes
    MPI_Bcast((void *)B, K * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    memcpy(host_B, B, K * N * sizeof(float));

    // Distribute A among nodes
    MPI_Scatter(A, local_rows_per_node * K, MPI_FLOAT,
                local_A, local_rows_per_node * K, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // Copy B to all GPUs asynchronously
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        size_t size_B = K * N * sizeof(float);
        cudaMemcpyAsync(d_B[i], host_B, size_B, cudaMemcpyHostToDevice, streams[i]);
    }

    // Divide A and perform computation on each GPU
    #pragma omp parallel for schedule(static) num_threads(NUM_GPUS)
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        size_t size_A = rows_per_gpu * K * sizeof(float);
        size_t size_C = rows_per_gpu * N * sizeof(float);

        // Copy a chunk of A to each GPU asynchronously
        cudaMemcpyAsync(d_A[i], local_A + i * rows_per_gpu * K, size_A, cudaMemcpyHostToDevice, streams[i]);

        // Launch the kernel asynchronously
        matmul_kernel<<<grid, block, 0, streams[i]>>>(d_A[i], d_B[i], d_C[i], rows_per_gpu, N, K);

        // Copy result back to host asynchronously
        cudaMemcpyAsync(host_C + i * rows_per_gpu * N, d_C[i], size_C, cudaMemcpyDeviceToHost, streams[i]);
    }

    // Wait for all GPUs to finish
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaStreamSynchronize(streams[i]);
    }

    // Gather results from all nodes
    MPI_Gather(host_C, local_rows_per_node * N, MPI_FLOAT,
               rank == 0 ? C : NULL, local_rows_per_node * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
}
```

MPI_Bcast: broadcast a whole B matrix to each node

Memcpy: cpy memory

MPI_Scatter: scatter(split by num_nodes) a matrix and spread to each node

cudaMemcpy(Async): copy memory from host to device(cudaMemcpyHostToDevice) or deevice to host(cudaMemcpyDeviceToHost)

cudaStraemSynchronize: synchronize streams in terms of its operations

MPI_Gather: gather the data in each node to certain rank(in this, to rank 0)

3) Matmul_finalize

```
void matmul_finalize() {
    for (int i = 0; i < NUM_GPUS; i++) {
        cudaSetDevice(i);
        cudaFree(d_A[i]);
        cudaFree(d_B[i]);
        cudaFree(d_C[i]);
        cudaStreamDestroy(streams[i]);
    }

    // Free pinned host memory
    cudaFreeHost(local_A);
    cudaFreeHost(host_B);
    cudaFreeHost(host_C);

    cudaDeviceReset();
}
```

cudaFree: free allocated memory in device

cudaStreamDestroy: remove streams

cudaFreeHost: free allocated memory in host

cudaDeviceReset: remove device setting

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

BLOCK_SIZE=32, 16, 8, 4, 2로 나누어 실험해 보았다. 다음 커맨드로 실험했다.

./run.sh 65536 4096 4096 -n 10 -v

결과는 다음과 같다.

BLOCK_SIZE	GFLOPS
32	**6031**
16	5741
8	5237
4	3283