Dokumen Wolfman

Table of Contents

CUDA

```
    int SIZE = N * N;
    matrix_C.resize(SIZE);
    // Allocate memory on the device
    D_array<double> d_matrix_A(SIZE);
    D_array<double> d_matrix_B(SIZE);
    D_array<double> d_matrix_C(SIZE);
    d_matrix_A.set(&matrix_A[0], SIZE);
    d_matrix_B.set(&matrix_B[0], SIZE);
    matrixMultiplication(d_matrix_A.getData(), d_matrix_B.getData(), d_matrix_C.getData(), N);
    cudaDeviceSynchronize();
    d_matrix_C.get(&matrix_C[0], SIZE);
```

```
__global__ void matrixMultiplicationKernel(double* A, double* B, double* C, int N) {
    int ROW = blockIdx.y * blockDim.y + threadIdx.y;
    int COL = blockIdx.x * blockDim.x + threadIdx.x;
    double tmpSum = 0;
    if (ROW < N && COL < N) {
        // each thread computes one element of the block sub-matrix
        for (int i = 0; i < N; i++) {
            tmpSum += A[ROW * N + i] * B[i * N + COL];
        }
        C[ROW * N + COL] = tmpSum;
    }
}

void matrixMultiplication(double *A, double *B, double *C, int N) {
    // declare the number of blocks per grid and the number of threads per block
    int threadsPerBlock = 32; // 32x32 = 1024 threads per block
    dim3 blockDim(threadsPerBlock, threadsPerBlock);
    dim3 gridDim((N + threadsPerBlock - 1) / threadsPerBlock, (N + threadsPerBlock - 1) / threadsPerBlock);
    matrixMultiplicationKernel<<<gridDim, blockDim>>>(A, B, C, N);
}
```

Dua Bagian paling krusial untuk mengubah pemrograman serial menjadi parallel. Pada gambar pertama kita mengalokasikan memory pada GPU yang nantinya kita akan mentransfer data dari CPU ke GPU. Mengapa begitu? Karena pemrograman GPU CUDA tidak bisa memproses secara langsung data yang datang dari CPU sehingga perlu pengalokasian terlebih dahulu. Saat data yang ada sudah ada, kita bisa melakukan pemrosesan dengan memanggil fungsi “kernel” yaitu __global__ void matrixMultliplicationKernel(). Fungsi ini dipanggil dan pemrosesan yang dilakukan tergantung pada saat pemanggillan. Pada gambar terdapat matrixMultiplicationKernel<<<gridDim, blockDim>>>(A, B, C, N). Pada bagian <<<>>> adalah hal yang lumrah pada pemrograman CUDA yang biasa disebut dengan “triple chevron” yang menspesifikkan jumlah block yang dipakai dan juga jumlah thread per block yang dipakai.

Python

```python
def multiply_chunk(matrix1, matrix2, row_start, row_end, n):
    result_chunk = np.zeros((row_end - row_start, n), dtype=np.float64)
    for i in range(row_start, row_end):
        for j in range(n):
            result_chunk[i - row_start, j] = np.dot(matrix1[i, :], matrix2[:, j])
    return result_chunk

def parallel_matrix_multiplication(matrix1, matrix2, n, num_workers=4):
    result_matrix = np.zeros((n, n), dtype=np.float64)
    chunk_size = (n + num_workers - 1) // num_workers  # Ceiling division

    with ThreadPoolExecutor(max_workers=num_workers) as executor:
        futures = []
        for i in range(0, n, chunk_size):
            row_end = min(i + chunk_size, n)
            futures.append(executor.submit(multiply_chunk, matrix1, matrix2, i, row_end, n))

        idx = 0
        for future in futures:
            result_chunk = future.result()
            row_start = idx * chunk_size
            result_matrix[row_start:row_start + result_chunk.shape[0], :] = result_chunk
            idx += 1

    return result_matrix
```

Pada Python saya menggunakan library Numpy untuk melakukan perkalian matrix. Ada dua fungsi yang krusial pada Python ini.

- Multiply_Chunk(): Pada bagian ini akan dilakukan perkalian matrix untuk subset dari baris dari matrix1 dan matrix2.
- Parallel_matrix_multiplication(): Pada bagian ini terjadi pemrosesan secara paralel. Pada bagian ini proses yang terjadi adalah membagi matrix menjadi beberapa subset baris dan menggunakan beberapa thread

bagian yang mengubah program serial menjadi program parallel adalah bagian ini

```python
with ThreadPoolExecutor(max_workers=num_workers) as executor:
    futures = []
    for i in range(0, n, chunk_size):
        row_end = min(i + chunk_size, n)
        futures.append(executor.submit(multiply_chunk, matrix1, matrix2, i, row_end, n))

    idx = 0
    for future in futures:
        result_chunk = future.result()
        row_start = idx * chunk_size
        result_matrix[row_start:row_start + result_chunk.shape[0], :] = result_chunk
        idx += 1
```

ThreadPoolExecutor: membuat thread pool dengan jumlah workers yang sudah ditentukan.

Executor.submit: meng-assign fungsi multiply_chunk ke executor untuk dikerjakan secara parallel oleh para workers.

Future.result(): mengambil hasil pekerjaan yang sudah dikerjakan oleh workers.

OpenMP

Untuk bagian ini saya kurang tau, karena menggunakan kode OpenMP hanya menambah library <omp.h> saja, tetapi kode-kode yang lainnya tidak berubah.

#include <omp.h>

#pragma omp parallel for collapse(2)

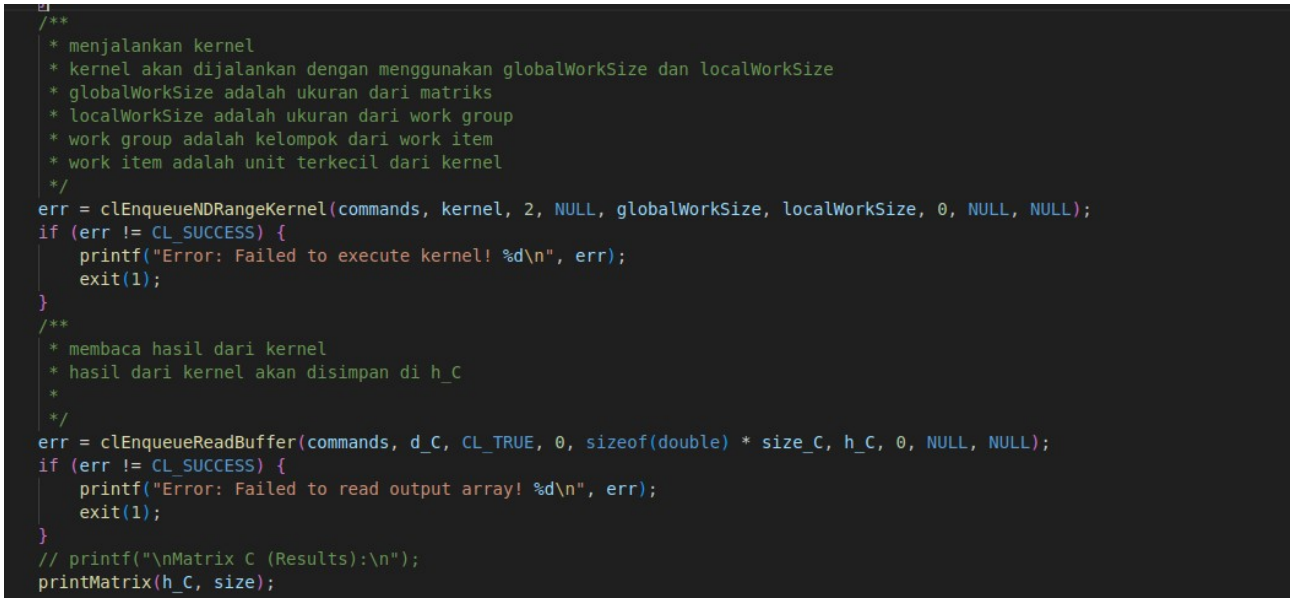
Yak, hanya menambah dua baris di atas saja.

OpenCL

OpenCL ini masih saudaraan sama CUDA karena OpenCL (Open Computing Language) adalah low-level API untuk heterogeneous computing yang berjalan pada GPU yang berbasis CUDA (nvidia).

```
 *
 */
cl_device_id device_id;
cl_context context;
cl_command_queue commands;
cl_program program;
cl_kernel kernel;
cl_mem d_A, d_B, d_C;
srand(2014);
printf("Initializing OpenCL device...\n");
/**
 * device dan platform
 * platform adalah perangkat lunak yang digunakan untuk menjalankan program OpenCL
 */
cl_uint dev_cnt = 0; //melakukan inisialisasi device
clGetPlatformIDs(0, 0, &dev_cnt);
cl_platform_id platform_ids[100]; //melakukan inisialisasi platform
clGetPlatformIDs(dev_cnt, platform_ids, NULL);
int gpu = 1;
/**
 * membuat device
```

Pada bagian program di atas sampai ke bawah merupakan bagian dimulainya program secara paraleli dimulai. Pemrosesannya terjadi pada bagian dibawah ini.



```
/**
 * menjalankan kernel
 * kernel akan dijalankan dengan menggunakan globalWorkSize dan localWorkSize
 * globalWorkSize adalah ukuran dari matriks
 * localWorkSize adalah ukuran dari work group
 * work group adalah kelompok dari work item
 * work item adalah unit terkecil dari kernel
 */
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
if (err != CL_SUCCESS) {
    printf("Error: Failed to execute kernel! %d\n", err);
    exit(1);
}
/**
 * membaca hasil dari kernel
 * hasil dari kernel akan disimpan di h_C
 *
 */
err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, sizeof(double) * size_C, h_C, 0, NULL, NULL);
if (err != CL_SUCCESS) {
    printf("Error: Failed to read output array! %d\n", err);
    exit(1);
}
// printf("\nMatrix C (Results):\n");
printMatrix(h_C, size);
```

Hasil akhirnya akan di-print oleh fungsi printMatrix(h_C, size)

Java

Bagian program paralel ada pada static class MatrixMultiplicationTask extends RecursiveTask<double[][]>. Terlebih pada bagian di bawah akan dibuat pool thread yang khusus digunakan untuk menjalankan task secara paralel. Pool.invoke digunakan untuk menjalankan MatrixMultiplicationTask().

```
ForkJoinPool pool = new ForkJoinPool();
double[][] result = pool.invoke(new MatrixMultiplicationTask(matrix1, matrix2, startRow:0, size));
```

Hasilnya akan digabung pada bagian akhir.

```
double[][] result1 = task1.join();
double[][] result2 = task2.join();

/**
 * penggabung hasil dari dua task yang sudah dijalankan
 * task1 dan task2
 *
 * task1 menghitung hasil perkalian matriks dari baris 0 sampai mid
 * task2 menghitung hasil perkalian matriks dari baris mid sampai endRow
 *
 * hasil dari task1 dan task2 digabungkan menjadi satu matriks
 *
 * hasil dari task1 disimpan di result1
 * hasil dari task2 disimpan di result2
 */
for(int i = 0; i < result.length; i++){
    for(int j = 0; j < result[0].length; j++){
        if(i < mid) {
            result[i][j] = result1[i][j];
        }else{
            result[i][j] = result2[i - mid][j];
        }
    }
}
```