Project 6 - Group 5

Team Members

Group Member	R#
Michael Beebe	R11772231
Diego Salas Noain	R11794236
Bandar Alkhalil	R11836831
Yongjian Zhao	R11915830
Denish Otieno	R11743138
Shiva Kumar Neekishetty	R11842757

Required Software

- MPI implementation (we are using Open MPI)
- C Compiler (such as gcc or clang)
- Make
- Bash

Instructions

Get an interactive job on the Matador partition (HPCC) to get access to an NVIDIA V100 GPU and the NVIDIA CUDA Compiler.

```
salloc -c 1 -t 60 -p matador
```

Compile

To change the MPI wrapper to something other than `mpicc` (such as `mpich`), edit line 1 of the Makefile. If you are using OpenMPI on the HPCC's Nocona partition, no changes to the Makefile are needed.

```
make
```

Run

```
./run.sh
```

If you get an error saying "permission denied", run

```
chmod +x run.sh
```

then rerun `./run.sh`

Clean Build

```
make clean
```

Code Breakdown

All source code can be found in the `src` directory.

Kernel Function (`__global__ void update_matrix()<<<...>>>`)

This function implements the core of the Floyd-Warshall algorithm in parallel.

- **Initialization**:
  - `int k = blockIdx.x;`: Represents the current phase of the algorithm.
  - `int i = threadIdx.y + blockDim.y * blockIdx.y;`: Calculates the row index.
  - `int j = threadIdx.x + blockDim.x * blockIdx.z;`: Calculates the column index.
- **Boundary Check**:
  - Ensures that the computation is within the matrix limits.
- **Shared Memory Allocation**:
  - `__shared__ int kRow[MATRIX_SIZE];`
  - `__shared__ int kCol[MATRIX_SIZE];`
  - Stores the k-th row and column of the matrix for quick access.
- **Data Loading into Shared Memory**:
  - Loads the k-th row and column to reduce global memory access.
- **Synchronization**:
  - `__syncthreads();`: Ensures all threads have loaded data into shared memory.
- **Matrix Update**:
  - Updates the matrix if a shorter path is found through vertex `k`.

```
__global__ void update_matrix(int *D, int n) {
    int k = blockIdx.x; // Current phase based on block index
    int i = threadIdx.y + blockDim.y * blockIdx.y; // Row index
    int j = threadIdx.x + blockDim.x * blockIdx.z; // Column index

    if (i < n && j < n) {
        __shared__ int kRow[MATRIX_SIZE];
        __shared__ int kCol[MATRIX_SIZE];

        // Load the k-th row and column into shared memory
        if (threadIdx.x == 0 && i < n) kCol[i] = D[i * n + k];
        if (threadIdx.y == 0 && j < n) kRow[j] = D[k * n + j];

        __syncthreads(); // Ensure loading is complete

        // Update the matrix
        if (i != j) {
            atomicMin(&D[i * n + j], kCol[i] + kRow[j]);
        }
    }
}
```

Function (`void run_update_matrix()`)

Handles the execution and management of the CUDA kernel.

- **Device Memory Allocation**:
  - Allocates memory for the matrix on the GPU.
- **Data Transfer**:
  - Copies matrix data from the host to the device.
- **Kernel Configuration and Launch**:
  - Configures grid and block dimensions.
  - Launches the kernel for each phase `k`.
- **Synchronization and Data Retrieval**:
  - Waits for GPU operations to complete.
  - Copies the updated matrix back to the host.
- **Memory Deallocation**:
  - Frees GPU memory.

```
void run_update_matrix(int *D, int n) {
    int *dev_D;

    // Allocate memory on the device
    cudaMalloc((void**)&dev_D, n * n * sizeof(int));
    cudaMemcpy(dev_D, D, n * n * sizeof(int), cudaMemcpyHostToDevice);

    // Define grid and block sizes
    dim3 blocks(n, 1, 1); // One block per phase
    dim3 threadsPerBlock(THREADS_PER_BLOCK / n, n);

    // Launch the kernel
    for (int k = 0; k < n; ++k) {
        update_matrix<<<blocks, threadsPerBlock>>>(dev_D, n);
    }

    // Synchronize and copy back results
    cudaDeviceSynchronize();
    cudaMemcpy(D, dev_D, n * n * sizeof(int), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(dev_D);
}
```

Matrix Initialization Function (`void initialize_matrix()`)

Initializes the matrix with random weights.

- **Random Weight Assignment**:
    - Non-diagonal elements get random weights.
    - Diagonal elements are set to zero.

```
void initialize_matrix(int *D, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                D[i * n + j] = 0;
            } else {
                D[i * n + j] = rand() % 10 + 1; // Random weights between 1 and 10
            }
        }
    }
}
```

Matrix Printing Function (`void print_matrix()`)

Displays the matrix.

- **Formatted Output**:
    - Prints the matrix elements row by row.

```
void print_matrix(int *D, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", D[i * n + j]);
        }
        printf("\n");
    }
}
```

Main Function (`int main()`)

Manages the overall execution flow.

- **Memory Allocation and Initialization**:
    - Allocates memory for the matrix.
    - Initializes the matrix with random data.
- **Processing and Display**:
    - Executes the Floyd-Warshall algorithm on the GPU.
    - Displays the matrix before and after processing.
- **Resource Cleanup**:
    - Frees allocated memory.

```
int main() {
  int n = MATRIX_SIZE; // Use the maximum matrix size
  int *D = (int*)malloc(n * n * sizeof(int));

  // Initialize D with random data
  initialize_matrix(D, n);

  printf("\nOriginal Matrix:\n");
  print_matrix(D, n);

  run_update_matrix(D, n);

  printf("\nMatrix after Floyd-Warshall:\n");
  print_matrix(D, n);
  printf("\n");

  free(D);
  return 0;
}
```

Output

```
gpu-21-6:/coursework/cs5379/cs5379-parallel-processing_Michael/project6$ ./run.sh

Original Matrix:
0 4 7 8
6 0 4 6
7 3 0 10
2 3 8 0

Matrix after Floyd-Warshall:
0 4 7 8
6 0 4 6
7 3 0 9
2 3 7 0
```