Module 4 Lab
CUDA Tiled Matrix Multiplication

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

Implement a tiled dense matrix multiplication routine using shared memory.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed "Matrix Multiplication" Lab
- You have completed the required module lectures

INSTRUCTIONS

Edit the code in the code tab to perform the following:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory
- implement the matrix-matrix multiplication routine using shared memory and tiling

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the Bitbucket repository. A description on how to use the CMake tool in along with how to build the labs for local development found in the README document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./TiledMatrixMultiplication\_Template -e <expected.raw> \
  -i <input0.raw>,<input1.raw> -o <output.raw> -t matrix
```

where `<expected.raw>` is the expected output, `<input0.raw>,<input1.raw>` is the input dataset, and `<output.raw>` is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

QUESTIONS

(1) How many floating operations are being performed in your matrix multiply kernel? explain.

ANSWER: **One dot-product per output matrix element.** $2 * numACols * numCRows * numCCols$

(2) How many global memory reads are being performed by your kernel? explain.

ANSWER: **Each thread does** $2 * ceil(numACols/TILE_WIDTH)$ **and there are** $numCCols * numCRows$ **active threads.**

(3) How many global memory writes are being performed by your kernel? explain.

ANSWER: **Only the output matrix is written.** $numCRows * numCCols$

(4) Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: **Adjusting kernel launch parameters for optimal occupancy,**

(5) Compare the implementation difficulty of this kernel compared to the previous MP. What difficulties did you have with this implementation?

ANSWER: **Address index translation from the global space to the shared memory space is a new place to make errors.**

(6) Suppose you have matrices with dimensions bigger than the max thread dimensions. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in this case.

ANSWER: **Each thread could do 4 or 9 or 16 adjacent elements of the matrix (thread coarsening).**

(7) Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.

ANSWER: **Just like the input is tiled for shared memory, the input could be tiled on the host side for kernel execution..**

CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
#include <wb.h>

#define wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
      return -1;                                                          \
    }                                                                     \
  } while (0)

// Compute C = A * B
__global__ void matrixMultiplyShared(float *A, float *B, float *C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns) {
  //@@ Insert code to implement matrix multiplication here
  //@@ You have to use shared memory for this lab
}

int main(int argc, char **argv) {
  wbArg_t args;
  float *hostA; // The A matrix
  float *hostB; // The B matrix
  float *hostC; // The output C matrix
  float *deviceA;
  float *deviceB;
  float *deviceC;
  int numARows;    // number of rows in the matrix A
  int numAColumns; // number of columns in the matrix A
  int numBRows;    // number of rows in the matrix B
  int numBColumns; // number of columns in the matrix B
  int numCRows;    // number of rows in the matrix C (you have to set this)
  int numCColumns; // number of columns in the matrix C (you have to set
                   // this)

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostA = (float *)wbImport(wbArg_getInputFile(args, 0), &numARows,
                            &numAColumns);
  hostB = (float *)wbImport(wbArg_getInputFile(args, 1), &numBRows,
                            &numBColumns);
  //@@ Set numCRows and numCColumns
```

```
  numCRows    = 0;
  numCColumns = 0;
  //@@ Allocate the hostC matrix
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);
  wbLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here

  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here

  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here

  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Launch the GPU Kernel here

  cudaDeviceSynchronize();
  wbTime_stop(Compute, "Performing CUDA computation");

  wbTime_start(Copy, "Copying output memory to the CPU");
  //@@ Copy the GPU memory back to the CPU here

  wbTime_stop(Copy, "Copying output memory to the CPU");

  wbTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here

  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostC, numCRows, numCColumns);

  free(hostA);
  free(hostB);
  free(hostC);

  return 0;
}
```

CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```
#include <wb.h>

#define wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      return -1;                                                          \
    }                                                                     \
  } while (0)

#define TILE_WIDTH 16

// Compute C = A * B
__global__ void matrixMultiply(float *A, float *B, float *C, int numARows,
                               int numAColumns, int numBRows,
                               int numBColumns, int numCRows,
                               int numCColumns) {
  //@@ Insert code to implement matrix multiplication here
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      Row = by * TILE_WIDTH + ty, Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

  for (int m = 0; m < (numAColumns - 1) / TILE_WIDTH + 1; ++m) {
    if (Row < numARows && m * TILE_WIDTH + tx < numAColumns)
      ds_M[ty][tx] = A[Row * numAColumns + m * TILE_WIDTH + tx];
    else
      ds_M[ty][tx] = 0;
    if (Col < numBColumns && m * TILE_WIDTH + ty < numBRows)
      ds_N[ty][tx] = B[(m * TILE_WIDTH + ty) * numBColumns + Col];
    else
      ds_N[ty][tx] = 0;

    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  if (Row < numCRows && Col < numCColumns)
    C[Row * numCColumns + Col] = Pvalue;
}

int main(int argc, char **argv) {
  wbArg_t args;
  float *hostA; // The A matrix
  float *hostB; // The B matrix
  float *hostC; // The output C matrix
  float *deviceA;
  float *deviceB;
  float *deviceC;
  int numARows;    // number of rows in the matrix A
```

```
  int numAColumns; // number of columns in the matrix A
  int numBRows;    // number of rows in the matrix B
  int numBColumns; // number of columns in the matrix B
  int numCRows;    // number of rows in the matrix C (you have to set this)
  int numCColumns; // number of columns in the matrix C (you have to set
                   // this)

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostA = (float *)wbImport(wbArg_getInputFile(args, 0), &numARows,
                            &numAColumns);
  hostB = (float *)wbImport(wbArg_getInputFile(args, 1), &numBRows,
                            &numBColumns);
  //@@ Set numCRows and numCColumns
  numCRows    = numARows;
  numCColumns = numBColumns;
  //@@ Allocate the hostC matrix
  hostC = (float *)malloc(sizeof(float) * numCRows * numCColumns);
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);
  wbLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here
  cudaMalloc(&deviceA, sizeof(float) * numARows * numAColumns);
  cudaMalloc(&deviceB, sizeof(float) * numBRows * numBColumns);
  cudaMalloc(&deviceC, sizeof(float) * numCRows * numCColumns);

  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here
  cudaMemcpy(deviceA, hostA, sizeof(float) * numARows * numAColumns,
             cudaMemcpyHostToDevice);
  cudaMemcpy(deviceB, hostB, sizeof(float) * numBRows * numBColumns,
             cudaMemcpyHostToDevice);

  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here
  dim3 dimGrid((numCColumns - 1) / TILE_WIDTH + 1,
               (numCRows - 1) / TILE_WIDTH + 1, 1);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Launch the GPU Kernel here
  matrixMultiply<<<dimGrid, dimBlock>>>(
      deviceA, deviceB, deviceC, numARows, numAColumns, numBRows,
      numBColumns, numCRows, numCColumns);

  cudaDeviceSynchronize();
```

```
    wbTime_stop(Compute, "Performing CUDA computation");

    wbTime_start(Copy, "Copying output memory to the CPU");
    //@@ Copy the GPU memory back to the CPU here
    cudaMemcpy(hostC, deviceC, sizeof(float) * numCRows * numCColumns,
               cudaMemcpyDeviceToHost);

    wbTime_stop(Copy, "Copying output memory to the CPU");

    wbTime_start(GPU, "Freeing GPU Memory");
    //@@ Free the GPU memory here
    cudaFree(deviceA);
    cudaFree(deviceB);
    cudaFree(deviceC);

    wbTime_stop(GPU, "Freeing GPU Memory");

    wbSolution(args, hostC, numCRows, numCColumns);

    free(hostA);
    free(hostB);
    free(hostC);

    return 0;
  }
```