(/mp/5983?show=code) Tiled Matrix Multiplication Attempt

Attempt Summary

Submit Attempt for Grading

Remember to answer the questions before clicking.

Dataset Id:	5
Created:	less than a minute ago (2022-03-05 09:06:35 +0000 UTC)
Status:	Correct solution for this dataset.

Timer Output

Kind	Location	Time (ms)	Message
Generic	main.cu::91	13.077868	Importing data and creating memory on host
GPU	main.cu::108	0.840356	Allocating GPU memory.
GPU	main.cu::120	0.119089	Copying input memory to the GPU.
Compute	main.cu::133	0.254237	Performing CUDA computation
Copy	main.cu::141	0.190876	Copying output memory to the CPU
GPU	main.cu::148	0.102842	Freeing GPU Memory

Logger Output

Level	Location	Message
Trace	main::104	The dimensions of A are 200 x 100
Trace	main::105	The dimensions of B are 100 x 256
Trace	main::106	The dimensions of C are 200 x 256

Program Code

```
#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)

#define TILE_WIDTH  32
#define BLOCK_SIZE 8
int ceil(int a, int b){
  return (a + b -1)/b;
}


// 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 MP
  __shared__ float TileP[TILE_WIDTH][TILE_WIDTH];
  __shared__ float TileQ[TILE_WIDTH][TILE_WIDTH];

  int block_x = blockIdx.x;
  int block_y = blockIdx.y;
  int thread_x = threadIdx.x;
```

```
  int thread_y = threadIdx.y;

  int row = (block_y * blockDim.y) + thread_y;
  int col = (block_x * blockDim.x) + thread_x;
  int next = (numAColumns + BLOCK_SIZE - 1) / BLOCK_SIZE;
  float hold = 0;

  for (int a = 0; a < next; a++){
    //load the first matrix tile
    if((a * BLOCK_SIZE + thread_x) > numAColumns){
      TileP[thread_y][thread_x] = 0.0;
    }
    else{
      TileP[thread_y][thread_x] = A[row * numAColumns + a * BLOCK_SIZE +
    }
    //load the second matrix tile
    if((a * BLOCK_SIZE + thread_y) >= numBRows){
      TileQ[thread_y][thread_x] = 0.0;
    }
    else{
      TileQ[thread_y][thread_x] = B[(a * BLOCK_SIZE + thread_y) * numBCol
    }

    __syncthreads();
    //perform multiplication calculation
    for (int b = 0; b < BLOCK_SIZE; b++){
      hold += TileP[thread_y][b] * TileQ[b][thread_x];
    }
    __syncthreads();
  }

  if (row < numCRows && col < numCColumns){
    C[row * numCColumns + col] = hold;
  }

  //__syncthreads();


}

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
```

```
78   float *deviceA;
79   float *deviceB;
80   float *deviceC;
81   int numARows;    // number of rows in the matrix A
82   int numAColumns; // number of columns in the matrix A
83   int numBRows;    // number of rows in the matrix B
84   int numBColumns; // number of columns in the matrix B
85   int numCRows;    // number of rows in the matrix C (you have to set thi
86   int numCColumns; // number of columns in the matrix C (you have to set
87                    // this)
88
89   args = wbArg_read(argc, argv);
90
91   wbTime_start(Generic, "Importing data and creating memory on host");
92   hostA = (float *)wbImport(wbArg_getInputFile(args, 0), &numARows,
93                             &numAColumns);
94   hostB = (float *)wbImport(wbArg_getInputFile(args, 1), &numBRows,
95                             &numBColumns);
96   //@@ Set numCRows and numCColumns
97   numCRows = numARows;
98   numCColumns = numBColumns;
99   //@@ Allocate the hostC matrix
100  wbTime_stop(Generic, "Importing data and creating memory on host");
101
102  hostC = (float *) malloc((numCRows * numCColumns) * sizeof(float));
103
104  wbLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);
105  wbLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);
106  wbLog(TRACE, "The dimensions of C are ", numCRows, " x ", numCColumns);
107
108  wbTime_start(GPU, "Allocating GPU memory.");
109  //@@ Allocate GPU memory here
110
111  int size_of_A = numARows * numAColumns * sizeof(float);
112  int size_of_B = numBRows * numBColumns * sizeof(float);
113  int size_of_C = numCRows * numCColumns * sizeof(float);
114  cudaMalloc((void **) &deviceA, size_of_A);
115  cudaMalloc((void **) &deviceB, size_of_B);
116  cudaMalloc((void **) &deviceC, size_of_C);
117
118  wbTime_stop(GPU, "Allocating GPU memory.");
119
120  wbTime_start(GPU, "Copying input memory to the GPU.");
121  //@@ Copy memory to the GPU here
122
```

```
  cudaMemcpy(deviceA, hostA, size_of_A, cudaMemcpyHostToDevice);
  cudaMemcpy(deviceB, hostB, size_of_B, cudaMemcpyHostToDevice);

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

  //@@ Initialize the grid and block dimensions here

  dim3 dimensionBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
  dim3 dimensionGrid(ceil(numCColumns, BLOCK_SIZE), ceil(numCRows, BLOCK_

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

  matrixMultiplyShared<<<dimensionGrid, dimensionBlock>>>(deviceA, device

  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, size_of_C, 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;
}

```

Designed and architected by Abdul Dakkak (https://www.dakkak.dev/).