

Module 4 Lab

Basic Matrix Multiplication

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

Implement a basic dense matrix multiplication routine. Optimizations such as tiling and usage of shared memory are not required for this lab.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed “Vector Addition” Lab
- You have completed the required teaching kit modules

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

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:

```
./BasicMatrixMultiplication_Stream_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: **numCRows * numCCols dot-products = 2 * numCRows * numCCols * numACols.**

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

ANSWER: **numCRows * numCCols dot-products = 2 * numCRows * numCCols * numACols.**

- (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 possible optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: **Tiling the input matrices in shared memory to reduce the number of global memory reads..**

- (5) Name three applications of matrix multiplication.

ANSWER: **Matrix Multiplication is present in almost every compute-intensive application. Neural Networks, Graphics, PDEs.**

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 in the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
1
2 #include <wb.h>
3
4 #define wbCheck(stmt)
```

\

```

5     do {
6         cudaError_t err = stmt;
7         if (err != cudaSuccess) {
8             wbLog(ERROR, "Failed to run stmt ", #stmt);
9             wbLog(ERROR, "Got CUDA error ... ", cudaGetErrorString(err));
10            return -1;
11        }
12    } while (0)
13
14    // Compute C = A * B
15    __global__ void matrixMultiply(float *A, float *B, float *C, int numRows,
16                                  int numAColumns, int numBRows,
17                                  int numBColumns, int numCRows,
18                                  int numCColumns) {
19        ///@@ Insert code to implement matrix multiplication here
20    }
21
22    int main(int argc, char **argv) {
23        wbArg_t args;
24        float *hostA; // The A matrix
25        float *hostB; // The B matrix
26        float *hostC; // The output C matrix
27        float *deviceA;
28        float *deviceB;
29        float *deviceC;
30        int numRows; // number of rows in the matrix A
31        int numAColumns; // number of columns in the matrix A
32        int numBRows; // number of rows in the matrix B
33        int numBColumns; // number of columns in the matrix B
34        int numCRows; // number of rows in the matrix C (you have to set this)
35        int numCColumns; // number of columns in the matrix C (you have to set
36                          // this)
37
38        args = wbArg_read(argc, argv);
39
40        wbTime_start(Generic, "Importing data and creating memory on host");
41        hostA = (float *)wbImport(wbArg_getInputFile(args, 0), &numARows,
42                                  &numAColumns);
43        hostB = (float *)wbImport(wbArg_getInputFile(args, 1), &numBRows,
44                                  &numBColumns);
45        ///@@ Set numCRows and numCColumns
46        numCRows = 0;
47        numCColumns = 0;
48        ///@@ Allocate the hostC matrix
49        wbTime_stop(Generic, "Importing data and creating memory on host");
50
51        wbLog	TRACE, "The dimensions of A are ", numRows, " x ", numAColumns);
52        wbLog	TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);
53
54        wbTime_start(GPU, "Allocating GPU memory.");
55        ///@@ Allocate GPU memory here
56
57        wbTime_stop(GPU, "Allocating GPU memory.");

```

```

58
59     wbTime_start(GPU, "Copying input memory to the GPU.");
60     /// Copy memory to the GPU here
61
62     wbTime_stop(GPU, "Copying input memory to the GPU.");
63
64     /// Initialize the grid and block dimensions here
65
66     wbTime_start(Compute, "Performing CUDA computation");
67     /// Launch the GPU Kernel here
68
69     cudaDeviceSynchronize();
70     wbTime_stop(Compute, "Performing CUDA computation");
71
72     wbTime_start(Copy, "Copying output memory to the CPU");
73     /// Copy the GPU memory back to the CPU here
74
75     wbTime_stop(Copy, "Copying output memory to the CPU");
76
77     wbTime_start(GPU, "Freeing GPU Memory");
78     /// Free the GPU memory here
79
80     wbTime_stop(GPU, "Freeing GPU Memory");
81
82     wbSolution(args, hostC, numCRows, numCColumns);
83
84     free(hostA);
85     free(hostB);
86     free(hostC);
87
88     return 0;
89 }

```

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.

```

1  #include <wb.h>
2
3  // Compute C = A * B
4  // Sgemv stands for single precision general matrix-vector multiply
5  __global__ void sgemm(float *A, float *B, float *C, int numARows,
6                      int numAColumns, int numBRows, int numBColumns) {
7      /// Insert code to implement matrix multiplication here
8      int row = blockIdx.y * blockDim.y + threadIdx.y;
9      int col = blockIdx.x * blockDim.x + threadIdx.x;
10     if (row < numARows && col < numBColumns) {
11         float sum = 0;
12         for (int ii = 0; ii < numAColumns; ii++) {
13             sum += A[row * numAColumns + ii] * B[ii * numBColumns + col];

```

```

14     }
15     C[row * numBColumns + col] = sum;
16 }
17 }
18
19 #define wbCheck(stmt)                                     \
20     do {                                                 \
21         cudaError_t err = stmt;                         \
22         if (err != cudaSuccess) {                       \
23             wbLog(ERROR, "Failed to run stmt ", #stmt); \
24             return -1;                                   \
25         }                                                \
26     } while (0)
27
28 int main(int argc, char **argv) {
29     wbArg_t args;
30     float *hostA; // The A matrix
31     float *hostB; // The B matrix
32     float *hostC; // The output C matrix
33     float *deviceA;
34     float *deviceB;
35     float *deviceC;
36     int numARows;    // number of rows in the matrix A
37     int numAColumns; // number of columns in the matrix A
38     int numBRows;    // number of rows in the matrix B
39     int numBColumns; // number of columns in the matrix B
40     int numCRows;
41     int numCColumns;
42
43     args = wbArg_read(argc, argv);
44
45     wbTime_start(Generic, "Importing data and creating memory on host");
46     hostA = (float *)wbImport(wbArg_getInputFile(args, 0), &numARows,
47                               &numAColumns);
48     hostB = (float *)wbImport(wbArg_getInputFile(args, 1), &numBRows,
49                               &numBColumns);
50     ///@@ Allocate the hostC matrix
51     hostC = (float *)malloc(numARows * numBColumns * sizeof(float));
52     wbTime_stop(Generic, "Importing data and creating memory on host");
53
54     numCRows    = numARows;
55     numCColumns = numBColumns;
56
57     wbLog	TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);
58     wbLog	TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);
59     wbLog	TRACE, "The dimensions of C are ", numCRows, " x ", numCColumns);
60
61     wbTime_start(GPU, "Allocating GPU memory.");
62     ///@@ Allocate GPU memory here
63     wbCheck(cudaMalloc((void **)&deviceA,
64                        numARows * numAColumns * sizeof(float)));
65     wbCheck(cudaMalloc((void **)&deviceB,
66                        numBRows * numBColumns * sizeof(float)));

```

```

67     wbCheck(cudaMalloc((void **)&deviceC,
68                       numRows * numBColumns * sizeof(float)));
69     wbTime_stop(GPU, "Allocating GPU memory.");
70
71     wbTime_start(GPU, "Copying input memory to the GPU.");
72     //@@ Copy memory to the GPU here
73     wbCheck(cudaMemcpy(deviceA, hostA,
74                       numRows * numAColumns * sizeof(float),
75                       cudaMemcpyHostToDevice));
76     wbCheck(cudaMemcpy(deviceB, hostB,
77                       numRows * numBColumns * sizeof(float),
78                       cudaMemcpyHostToDevice));
79     wbTime_stop(GPU, "Copying input memory to the GPU.");
80
81     //@@ Initialize the grid and block dimensions here
82     dim3 blockDim(16, 16);
83     dim3 gridDim(ceil(((float)numAColumns) / blockDim.x),
84                ceil(((float)numBRows) / blockDim.y));
85
86     wbLog	TRACE, "The block dimensions are ", blockDim.x, " x ", blockDim.y);
87     wbLog	TRACE, "The grid dimensions are ", gridDim.x, " x ", gridDim.y);
88
89     wbTime_start(Compute, "Performing CUDA computation");
90     //@@ Launch the GPU Kernel here
91     wbCheck(cudaMemset(deviceC, 0, numRows * numBColumns * sizeof(float)));
92     sgemm<<<gridDim, blockDim>>>(deviceA, deviceB, deviceC, numRows,
93                                numAColumns, numBRows, numBColumns);
94     cudaDeviceSynchronize();
95     wbTime_stop(Compute, "Performing CUDA computation");
96
97     wbTime_start(Copy, "Copying output memory to the CPU");
98     //@@ Copy the GPU memory back to the CPU here
99
100    wbCheck(cudaMemcpy(hostC, deviceC,
101                      numRows * numBColumns * sizeof(float),
102                      cudaMemcpyDeviceToHost));
103    wbTime_stop(Copy, "Copying output memory to the CPU");
104
105    wbTime_start(GPU, "Freeing GPU Memory");
106    //@@ Free the GPU memory here
107    cudaFree(deviceA);
108    cudaFree(deviceB);
109    cudaFree(deviceC);
110    wbTime_stop(GPU, "Freeing GPU Memory");
111
112    wbSolution(args, hostC, numRows, numBColumns);
113
114    free(hostA);
115    free(hostB);
116    free(hostC);
117
118    return 0;
119 }

```

© ⓘ ⓘ This work is licensed by UIUC and NVIDIA (2016) under a [Creative Commons Attribution-NonCommercial 4.0 License](#).