**Assignment3 Report (CSE436, Summer 2016)**

**Kazumi Malhan, 06/20/2016**

1. Backgrounds and Motivation

In recent years, the graphics processing unit (GPU) is getting more attention for parallel programming, as it has high throughput and optimized for parallel execution with its large amount of cores. However, GPU programming involves new overhead such as data movement between CPU and GPU memory. In assignment 3, matrix multiplication function is implemented in five different types: serial, OpenMP, CUDA with global memory, CUDA with shared memory, and CUDA library. The performance of these functions and time consumption distribution are analyzed to understand the effectiveness and cost of GPU programming.

2. Function Implementation Description

**matmul\_cuda\_v1\_vanilla function (Host)**

The function first determines the size need to be allocated on GPU memory by multiplying size of matrix with size of REAL data type. Next, for each matrix A, B, and C, pointer is declared. cudaMalloc function is used to allocate necessary GPU memory and pointed with previously declared pointer. cudaMemcpy moves the matrix A and B from CPU memory to GPU memory. Since matrix multiplication deals with 2-D data, dim3 data type to specify block size and grid size. Per instruction, block size is fixed with 16 \* 16 = 256 threads. The size of grid is calculated by dividing size of matrix with size of block. The kernel function matmul\_shared\_kernel is launched with specified block and grid size and required parameters are passed. When computation completes, cudaMemcpy moves C matrix back from GPU to CPU memory. Finally, cudaFree function is used to free allocated GPU memory.

**matmul\_global\_kernel function (Device)**

First, local variables are declared. Next, absolute row and column location of each thread is calculated using following equation 1 and 2. Next, each threads calculate the result of one element of C matrix by using for loop. This result is calculated with using temp variable rather than accessing to C matrix, as accessing local variable is faster than accessing global variables. Finally, if the thread location is within C matrix size, the calculated result is written into C matrix in global memory.

Row = blockIdx.y \* blockDim.y + threadIdx.y, Equation (1)

Column = blockIdx.x \* blockDim.x + threadIdx.x Equation (2)

**matmul\_cuda\_v1\_shmem function (Host)**

Please refer to matmul\_cuda\_v1\_vanilla for detail. This function is almost same except it calls kernel function that uses shared memory instead of global memory.

**matmul\_shared\_kernel function (Device)**

The function declares local variables. The x and y location of thread is stored in local variable as it is frequently used by the function. Next, absolute row and column location of each thread is calculated using equation 1 and 2 listed above. The shared memory is declared with “\_\_shared\_\_” keyword, which stores the sub-matrix during calculation.

**matmul\_cuda\_v1\_cublas function (Host)**

3. Hardware and Compiler Information

4. Performance Report

All of performance data collection is done on **yoko.secs.oakland.edu** Linux server via VPN connection.

5. Conclusion

References