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 (float) 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. It is notable that row is y direction, and column is x direction. Next, each thread calculates 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 much 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. Size of shared memory is same as block size because shared memory is per block. Inside the for loop, sub-matrix generation and matrix calculation is performed. In order to generate submatrix, the if statement is used to ensure that only thread inside matrix size is performing this operation. Since original matrix is stored in row-major fashion, modified version of $(i, j) = i * \mathrm{N} + j$ equation is used to copy data into shared memory. It is important to note that BLOCK_SIZE*i is added to the equation to consider offset. After the sub-matrix generation, __syncthreads() function is called to ensure that all threads have generated sub-matrix. Next, another for loop is used to calculate result of one element of C matrix, and stored into temp variable. Again, __syncthreads() is used afterward to ensure the completion of result calculation. Finally, if the thread is within matrix size, result is copied to C matrix.

matmul_cuda_v1_cublas function (Host)

Similar to other host functions, size of GPU memory allocation is calculated, allocated on GPU using cudaMalloc, and pointed with declared pointers for matrix A, B, and C. Next, cublas handle is created to use cublas library. The data copy from CPU memory to GPU memory is performed using cublasSetMatrix function. The matrix multiplication operation is performed with cublasSgemm function. There are multiple version of cublas matrix multiplication function exists, but this version is selected as our data is REAL (float). It is notable that "CUBLAS_OP_T" option is passed to the function because this function expects column-major matrix while input matrix is row-major matrix. Also, alpha and beta is declared before the function as we need to pass the address of these variables to the function. After the calculation, calculated data is copied from GPU memory to CPU memory using cublasGetMatrix function. Finally, allocated memory on GPU is de-allocated, and cublas handle is destroyed. There is code outside of this function that transpose the resulting C matrix from column-major to row-major. The code is placed outside of the function to accurately measure the execution time of cublas function, and not to include extra memory allocation time.

3. Hardware and Compiler Information

All of performance data collection is done on **yoko.secs.oakland.edu** Linux server via VPN connection. This machine contains 56 threads (2 threads per core) and has x86-64 architecture. The CPU model is Intel Xeon E-2683 v3 @ 2.00GHz. The machine was configured to run at 1.2 GHz when code was executed. The machine has following cache configuration. L1i:32K, L1d: 32K, L2: 256K, L3: 36M, with 94 GB of RAM. The GPU used on this machine is NVIDIA Tesla K20Xm. It contains 2688 CUDA cores (14 Multiprocessors, 192 GUDA cores/MP). The GPU has 5.7 GB of global memory, each block contains 49KB of shared memory and about 56636 registers.

To obtain cpu information, "lscpu", "cat /proc/meminfo", and "cat /proc/cpuinfo" command can be used from terminal. To obtain GPU information some of the information is obtained with "nvidia-smi". The most of the information is obtained by running deviceQuery code provided in NVIDIA CUDA samples.

GCC compiler version 4.8.5 and NVCC compiler release 7.5 are used to generate the code for this assignment. To generate the code, "nvcc –Xcompiler -fopenmp matmul.cu -lpthread -lcublas -o matmul" command is used. –Xcompiler allows us to directly specify options for host compiler. GCC information is obtained with "gcc --version" while NVCC information is got with "NVCC --version".

4. Performance Report

The **figure 1** below shows the performance difference among OpenMP, and three different CUDA functions. Since yoko.secs.oakland.edu has 56 CPUs, code is executed with number of tasks specified as 56. Before timing was measured, same function is called beforehand to warm up the GPU. The figure clearly shows that GPU functions are significantly faster than CPU function. For example, cuBLAS function is 105 times faster than OpenMP at matrix size of 2048. The reason of this difference is that CPU has 56 thread working on the program while GPU has thousands of threads working on same amount of problem. Among three GPU functions, cuBLAS is the fastest, followed by shared memory version. The execution time difference between global and shared memory at 2048 is 2.21. As the size of matrix increases, the time delay due to accessing global memory increases. The execution time of cuBLAS function was significantly shorter than shared memory version. For example, cuBLAS is 3.5 times faster at 2048 (7.8 times faster than global memory function). The

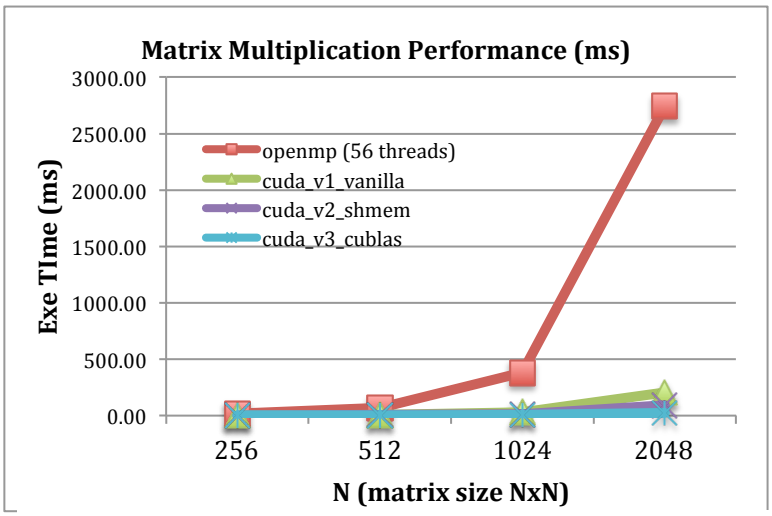

Figure 1: Performance difference among four function with various matrix size

cuBLAS function utilizes the faster memory as much as possible, and customizes the implementation specifically to devices to get most efficient performance. The notable finding was that cuBLAS function was slower than other two GPU functions when matrix size was 256.

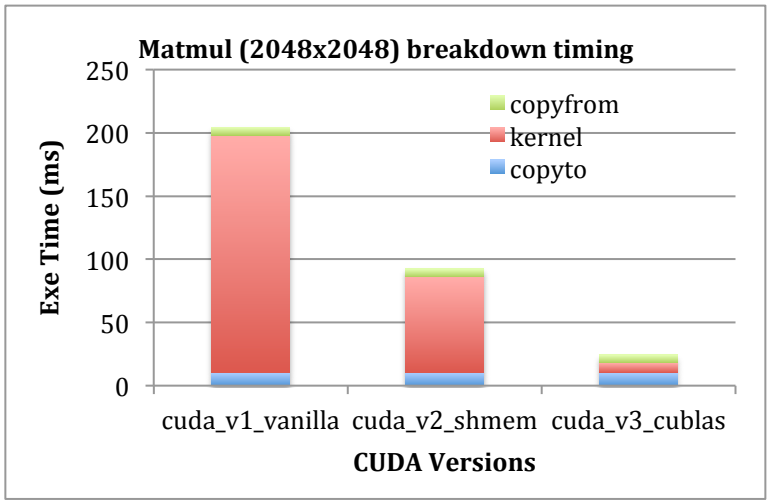

Figure 2: Timing breakdown of three CUDA functions

The **figure 2** on the left side shows the timing breakdown of three GPU functions. The data is collected by commenting out other functions, and measure timing using nvprof. Among three functions the time takes to move the data using cudaMalloc or cublasGetMatrix / cublasSetMatrix) were about same. Therefore, the execution time difference between the functions are due to kernel execution. By using shared memory, the kernel function becomes 2.5 times faster. The cuBLAS function was 9.7 times faster than shared memory function, and 24.2 times faster than global memory version. The graph shows the significance of memory access time and effectiveness of using library function.

5. Conclusion

The use of graphics processing unit (GPU) has significant advantage over multi-threaded CPU program for matrix multiplication operation (Single instruction on multiple data). This is because GPU uses thousands of threads while CPU only has around 60 threads. This advantage overcomes the overhead of moving data in between CPU and GPU memory. Also, GPU focuses on data parallelism while CPU focuses on task parallelism. Within GPU functions, use of shared memory is faster than using global memory. Since the matrix multiplication has large number of memory access, this overhead impacts the performance a lot. Using cuBLAS library is the best option for matrix multiplication as it shows almost 8 times faster than using global memory. This is because the function is highly optimized by CUDA export to perform as efficiently as possible. The timing breakdown shows that time takes to copy data between CPU and GPU memory are same among all three functions, and kernel execution time is really makes the difference in function execution time. The assignment demonstrated that effective use of library is better than creating own function for better performance.

References

cuBLAS Documentation (http://docs.nvidia.com/cuda/cublas/)