CSCI 563 : Parallel Computing
Assignment 3 : Sparse Matrix Vector Multiplication

Vinh Le
Colorado School of Mines Computer Science 2018

April 5, 2018

1 Sparse Matrix Vector Multiplication

In this assignment, we will be writing code in Nvidia's CUDA programming language to utilize the Graphical Processing Unit's (GPU) powerful and massive parallel systems in solving sparse matrix vector multiplication (SpVM). SpVM is important in many scientific applications. A matrix is sparse in that many of its elements are zero. We are given two data sources from the MatrixMarket, that is currently in the sorted coordinate representation. We will be utilizing the compressed row storage (CRS) representation in the code therefore we will need to convert the code from the sorted coordinate representation to the compressed row storage representation. In the given c code the program reads in the data, converts the data in to the CRS format then utilizes the CPU to compute the SpVM.

For the CUDA code it will be very similar to the c code's structure. First, we will create and allocate memory for all of the variables and arrays. Second, we will copy the data to the device and run the CUDA program. Finally, we will copy the finished array back to the host and we will compare it to the array produced by the CPU implementation.

For the CUDA implementation, after copying the data to the device, we will use 32 threads (1 warp) to do the calculations. A shared partial sum array is initialized and set to 0. Each thread will access its own element, therefore, there will be no data race condition. After the 32 partial sums are complete there is a syncthreads call to make sure that all threads are at the same place. Finally, the 32 threads are used in a halving manner where in the first iteration the sum is added to the first half of the thread id elements, then in the second iteration only 16 threads are working and is added to the first half of the thread id elements. Through this program we utilize memory coalescing and load balancing techniques.

2 Memory Coalescing

When all threads of a warp execute a load instruction and only one DRAM request will be made, the access is fully coalesced. In this implementation, we utilize 32 threads which is the size of the warp and then we use the "A[(expression with terms independent of threadIdx.x) + threadIdx.x]" form which will only have one load instruction for each thread execution. therefore this implementation is fully coalesced

3 Load Balancing

In this implementation of SpVM using the GPU the load is distributed fairly. During the initial partial summation, in each iteration each thread is used at most once. Then a syncthreads is called to create a barrier to stop fast threads. Then in the secondary summation 32 thread are initially used then in the second iteration 16 threads are working then 8 threads are used and so on. This method is the best distribution of work for the implementation.

4 Correctness

Finally, when both implementations are complete (CPU and GPU) the difference is taken and compared to a TOLERANCE value (which is set to 0.001). This checks the correctness of the GPU implementation against the "correct" CPU implementation. In many runs the Number of Failures is equal to 0.