CSC548 Parallel System Project Report2

Abhishek Dhanotia, Fang Liu and Fei Meng
{adhanot, fliu3, fmeng}@ncsu.edu

Task 1: Parallelizing CG Benchmark on CUDA

Abhishek Dhanotia (adhanot@ncsu.edu)

1. Analyzing the CG code to determine part of the program can be moved to CUDA kernel. Used gprof for profiling computation hotspots.
    a. Conj_grad function consumes most of the computation time in the program.
    b. This function implements a matrix-vector multiplication on a sparse matrix as shown below

```
1110        do cgit = 1, cgitmax
1117            do j=1,lastrow-firstrow+1
1118                sum = 0.d0
1119                do k=rowstr(j),rowstr(j+1)-1
1120                    sum = sum + a(k)*p(colidx(k))
1121                enddo
1122                w(j) = sum
1123            enddo
```

The computations are done only on the non-zero elements of the matrix. *rowstr(j)* stores the 1st non-zero entry if the matrix and for row j. Array 'a' holds only the non zero entries of the matrix on which computation needs to be performed. This part of the code consumes almost 83% of computation time for class B inputs and 92% of computation time for Class B inputs.

When moving this code to the CUDA kernel, each thread can get iterations from the outermost loop. Depending on the number of threads available, computations can be distributed accordingly form the outermost loop iterating over cgit, then over rows and finally over elements within a row.

2. Compiling the CG Fortran code for CUDA
    a. Tried using the F2C-ACC converter for generating a C code. However we observed that there were multiple issues associated with it.
        i. F2CC doesn't allow certain commenting styles and syntax. So hand-modified the code and removed all such problems.
        ii. Subsequently, when trying to convert the code, F2CC reports that some constructs are not supported at all. So the option of using F2C is ruled out.

    b. Using PGI compiler for compiling the FORTRAN code.
        i. Using the make structure from the NAS benchmark

ii. Faced some linker issues when compiling MPI constructs. It seems to be resolved now.

3. Generating results for CG runtimes using MPI only parallelization.
    a. Table below shows the runtimes for CG using MPI on 1, 2 and 4 processors.

Overall Time/sec	1 Proc	2 Proc	4 Proc
Class A	6.603	3.233	1.794
Class B	1001.7	149.11	77.54

4. Implementing the hotspot as a CUDA kernel. Learning how to write the kernel code in FORTRAN using the manuals provided with PGI compiler.

Open Issues and Future Work:

1. Complete implementation of the CUDA kernel. Compile the host and kernel code simultaneously using the same make file structure. (to be finished by Nov 14)
2. Completing and running the CUDA kernel implementation and running the code on single GPU. (to be finished by Nov 17)
3. Running the complete program on multiple GPUs, generating the results and analysing performance. (To be finished by Nov 20)

Task 2: Parallelizing DT Benchmark on CUDA

Fang Liu (fliu3@ncsu.edu)

DT benchmark has constant number of processes for a specified class and it takes graph type as a parameter. For example, in Class A, BH and WH use 21 processes and SH uses 80 processes. Using gprof profiling, we identify the hotspot function RandomFeatures contribute to 33.3% of execution for BH and 100% of execution for WH in class A.

Milestones

1. Compile and run DT MPI benchmark on os40~55 machines under the environment of CUDA 2.3. (Nov 6, 2009)
2. Implement the identified hotspot into CUDA kernel. Debug the code and make sure the correctness of the parallelized code. (Nov 9, 2009)
3. Collect and compare CUDA and MPI results. (Nov 11, 2009)

Implementation of CUDA kernel

The major loop in RandomFeatures is the iteration of generating array elements that would be sent to nodes connected to it on the data flow graph. Each iteration generates 4 independent elements stores them in 4 consecutive locations in the array. Therefore, the iteration stride is 4. Loop carried dependency is presented every two consecutive iterations. To parallelize the loop, we fix the thread block size to 4, in which 4 threads independently work on 4 streams. We implement and investigate various numbers of thread blocks.

Preliminary Results and Analysis

Table 1 shows the overall execution time (unit: second) for MPI version in which all code is executed on the hosts and CUDA version in which kernel code is extracted and executed on CUDA devices.

Table 1 Overall Execution Time for DT with Class A

Overall Time/sec	MPI	CUDA 1block	CUDA 4block	CUDA 16block	CUDA 32block	CUDA 64block	CUDA 128block
Class A BH	2.43	3.42	3.51	3.21	3.42	3.36	3.55
Class A WH	2.86	3	3.01	2.99	2.98	3	3
Class A SH	7.62	6.91	6.67	6.53	6.52	6.87	6.57

From the above table, it looks like CUDA acceleration does not have performance benefit over the MPI version. To understand the reasons, we measure the execution time (unit: micro-second) of the parallel code (Table 2). For MPI version, it is the MPI wall clock time of the loop identified as parallelizable. For CUDA version, it is the measured CUDA time, including memory allocation & initialization, kernel call and data moving between the host and device.

Table 2 CUDA Time for DT with Class A

CUDA Time/usec	MPI	CUDA 1block	CUDA 4block	CUDA 16block	CUDA 32block	CUDA 64block	CUDA 128block
Class A BH	15625	112.97	106.22	102.33	104.06	104.46	107.53
Class A WH	11718.75	112.98	106.45	102.30	103.30	104.55	107.36
Class A SH	19531.25	110.12	103.32	99.08	110.28	133.21	120.83

From Table 2, it looks like the CUDA parallelized code does show much better performance over MPI version, contradicting with the observation from Table 1.

Open Issues and Future Work

Figure out the reasons why current results deviate from the expectation and improve the CUDA implementation. (to be finished by Nov 18)

Task 3: Parallelizing IS Benchmark on CUDA

Fei Meng(fmeng@ncsu.edu)

Milestones

1. Figure out the computation hotspot function "randlc" which is called tens of thousands of times, using gprof for profiling.
2. Understand the mechanism/algorithm of "randlc" and identify the hotspot where is called.
3. Implement the "randlc" part as kernel, which is executed on CUDA on os clusters.
4. MPI-CUDA hybrid integration.

Open Issues and Future Work:

1. Basically "randlc" needs to be called sequentially according to its logic outlined in the benchmark official document. Currently I calculate the initial value for each kernel part using another programme before I run the kernel. Then I feed the kernel using the results. So one possible improvement of the implementation is to incorporate the pre-process into the host or kernel part. (to be finished by Nov. 15)
2. Another further work is about the second time-consuming function "rank", according to gprof result. It is possible that rank costs time because of plenty calls to MPI routines though. So another possible improvement of the implementation is using CUDA to accelerate rank function. (to be finished by Nov. 22)