CSC548 Parallel System Project Final Report

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

Task 1: Parallelizing NAS CG Benchmark for CUDA

Abhishek Dhanotia (adhanot@ncsu.edu)

1. The kernel part of the CG benchmark which was implemented in CUDA
   (1) Conj_grad function consumes most of the computation time in the program.
   (2) This function implements a matrix-vector multiplication on a sparse matrix as shown below:

```
    do cgit = 1, cgitmax
       do j=1,lastrow-firstrow+1
          sum = 0.d0
          do k=rowstr(j),rowstr(j+1)-1
             sum = sum + a(k)*p(colidx(k))
          enddo
          w(j) = sum
 Some MPI Sends and Receives
   enddo
```

2. Work distribution for each thread
   (1) Each iteration in the j loop involves 2*(rowstr(j+1) – rowstr(j)) computations
   (2) 2 Approaches for parallelization were followed
   (3) Approach 1 – Invoking a kernel for each iteration of the j loop.
      a. For class A inputs, j goes from 1 to 14000 which means invoking 14000*5 CUDA kernels
      b. Each thread in the invoked kernel would do a part of the k iterations
   (4) Approach 2 – Invoking 1 kernel for each iteration of cgit
      a. For class A inputs, cgit iterates over 1 to 15 meaning there are a total of 15 kernels invoked by the program
      b. Each thread in this case would involve going over the complete k iterations of a particular j iteration

3. Issues in compiling Fortran code for CUDA
   (1) PGI compiler was used to code and compile the CUDA version of the CG benchmark
   (2) However, there were a lot of issues involved in the compilation and running the code with PGI. The major ones are listed below
      a. The compiler does not seem to support usage of shared memory in all combinations. It gives some compilation errors when trying to copy data

from shared to global memory directly and vice versa. Hence for the given code, global memory was only used for all the computations

b. When invoking large number of threads, the code gives a “copyin Memcpy Failed” error even when there is no memory is being used inside the kernel. This was a major bottleneck as the kernel could not be run for more than 16 threads with 16 thread blocks in most cases.

c. The result of this report can thus not be generalized for all possible combilations of Thread and thread block dimensions

(3) The data copying from device to host and vice versa takes up unexpectedly large amount of time thereby not allowing for speedup in all simulations if overall time is considered.

(4) The kernel invoking time also seems to be a bottleneck as it takes up a lot of wall clock time invoking a kernel from the conj_grad function. The reason for this and the memory copy problem could not be determined.

(5) The cudaEvent APIs didn’t work on PGI (giving some compilation errors) so had to use the get_time function of fortran for calculating timings

Results and Analysis

1. Time for Data transfer from Host to Device and vice versa

(1) The Data copy from ***host to device*** takes **8.89 seconds** on average per iteration

(2) The Data copy from **device to host** takes **2.62 seconds** on average per iteration

2. Kernel execution time using host and then using 2 different kernels

Case 1: Running 100 iterations of the j loop ignoring the time for data transfer for Class A input of the CG benchmark

(1) Time taken on host = 22.6 ms

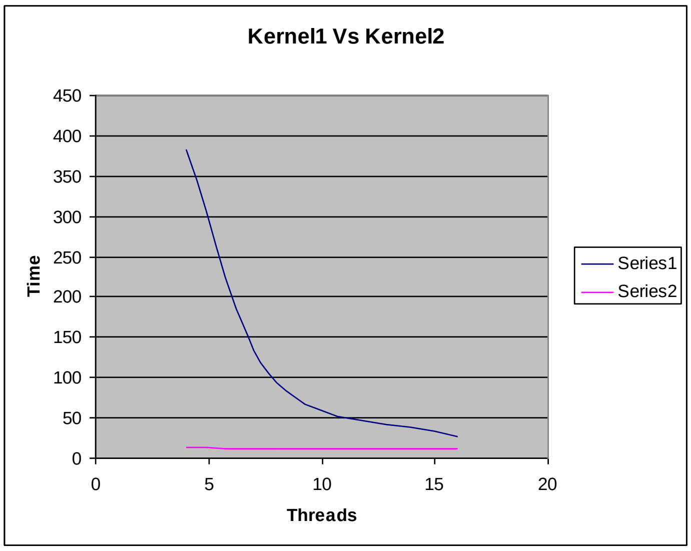


Figure1: Execution time of Kernel1 Vs Kernel2. (X Axis denotes NxN dimension for kernels)

(2) Time taken using approach 1 on different number of threads

Nblocks* BlockDim	4x4	8x8	16x16
Time(ms)	383	93	27

(3) Time taken using approach 2 on different number of threads

Nblocks* BlockDim	4x4	8x8	16x16
Time(ms)	14	12.54	11.37

As can be seen from the results, approach 2 gives better performance over approach 1. The main drawback for approach 1 is the spawning of a large number of kernels which is a huge overhead. On the other hand, approach 2 spawns only a few number of kernels thereby reducing the overhead and making the simulation faster

Case 2: Running 10000 iterations of the j loop ignoring the time for data transfer for Class A input of the CG benchmark.

(1) Total execution time for the main program kernel on the host for each iteration is **396 ms (average)**
(2) Kernel run times for different number of threads for approach 2

Nblocks* BlockDim	4x4	8x8	16x16	32x32	128x128
Time(ms)	12.23	12.12	11.8	11.46	11.14

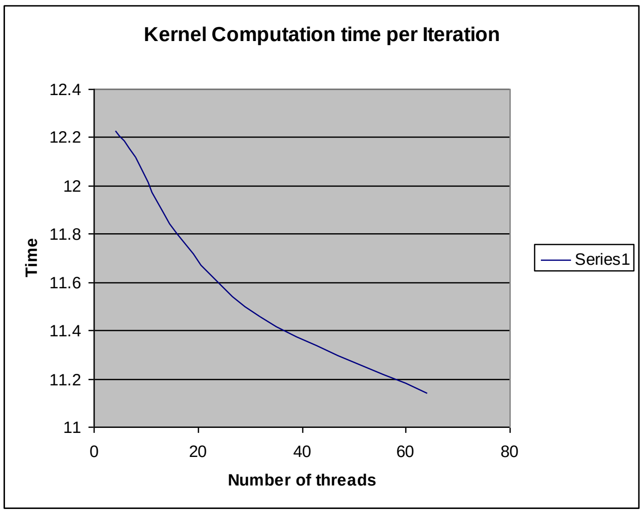


Figure2: Kernel2 Time per computation for different number of threads. X Axis denotes NxN dimension for the kernel

The kernel execution time for the core computation of CG becomes much faster when CUDA is used. However, if the time taken for data transfers between the Host and the Device are considered, it turns out that the overhead far surpasses the advantages of using the kernel for computation. It seems that the PGI compiler is not optimizing data transfers between host and device and hence the overall speedup numbers don't give a clear picture of the CUDA acceleration capabilities

Open Problems/ Future Work

(1) The PGI Fortran compiler is still in its beta version and there are several issues which still need to be resolved. Hence running full scale simulations was difficult as there were some runtime issues which could not be resolved.

(2) The compiler gave some memory allocation errors when shared memory was used. Hence all the results were generated based on calculations in the global memory. If shared memory is used, that would further speedup the kernel computation times

(3) Device to Host data transfers (and vice versa) took up an unexpectedly large amount thereby resulting in an overall slowdown when running the code on CUDA. This needs to be looked upon. Maybe adding some compiler optimizations/accelerators would help.

(4) The CG benchmark could not be run completely due to some runtime errors when running the code for a large number of iterations. So the generated results are for a scaled down version of the benchmark.

Task 2: Parallelizing DT Benchmark on CUDA

Fang Liu (fliu3@ncsu.edu)

DT benchmark takes one argument: BH, WH or SH which specifies the communication graph Black Holes, White Holes or Shuffle respectively. The number of nodes in a communication graph is a constant value once the Class type is specified, and the number of executable processors should not be less that number. Specifically, for Class A, BH, WH and SH require 21, 21 and 80 nodes respectively.

1. Computation Hotspots

Running MPI version of DT benchmark on hery2, we can identify the computation hotspot function *RandomFeatures* as shown in Table 1 below:

Table 1 Profiling Computation Hotspots for Different Communication Graph

Graph	% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
BH	33.33	0.01	0.01	1	10.00	10.00	RandomFeatures
WH	100.00	0.01	0.01	1	10.00	10.00	RandomFeatures
SH	0.54	1.74	0.01	1	10.00	10.00	RandomFeatures

2. Algorithm Analysis

The major computation in *RandomFeatures* is a nest loop which generates data elements and stores them in a large array of a node:

```
for (i = 0; i < len; i += fdim){
    for (j = 0; j < fdim; j++) {
        h_seed[j] = (h_seed[j] * ng[j]) % n[j];
        feat->val[i+j] = h_seed[j];
    }
}
```

The inner loop is actually a vector operation with dimension of fdim since the arrays h_seed[ ], ng[ ] and n[ ] have fdim independent elements. The outer loop is loop-carried dependent in which current iteration is dependent on the previous iteration. Based on this observation, we can offload the computation to CUDA threads at the granularity of fdim in which CUDA threads operate on at least fdim independent data streams. In the source code, fdim is fixed to value of 4. Therefore, the thread block size is multiples of 4.

3. Simulation Results

Class A is the one has the largest working set size among the classes that have verified results. If the CUDA kernel leads to correct result, the standard output will show “Deviation = 0.000000” and “Verification = SUCCESSFUL”. Hence this is the criterion for implementing CUDA kernel function.

We collect results for kernel computation time as well as total execution time. We compare the results between CUDA in which the kernel computation is extracted and running on the device while the remaining code is running on the host and MPI in which the entire execution including kernel computation and remaining code is running the host only.

(1) Kernel Time on CUDA vs. MPI

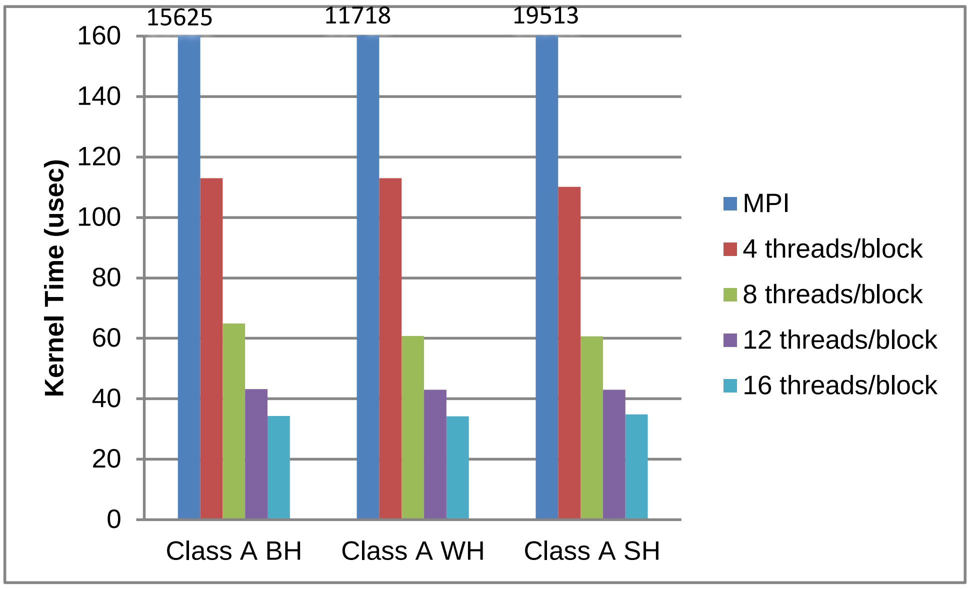
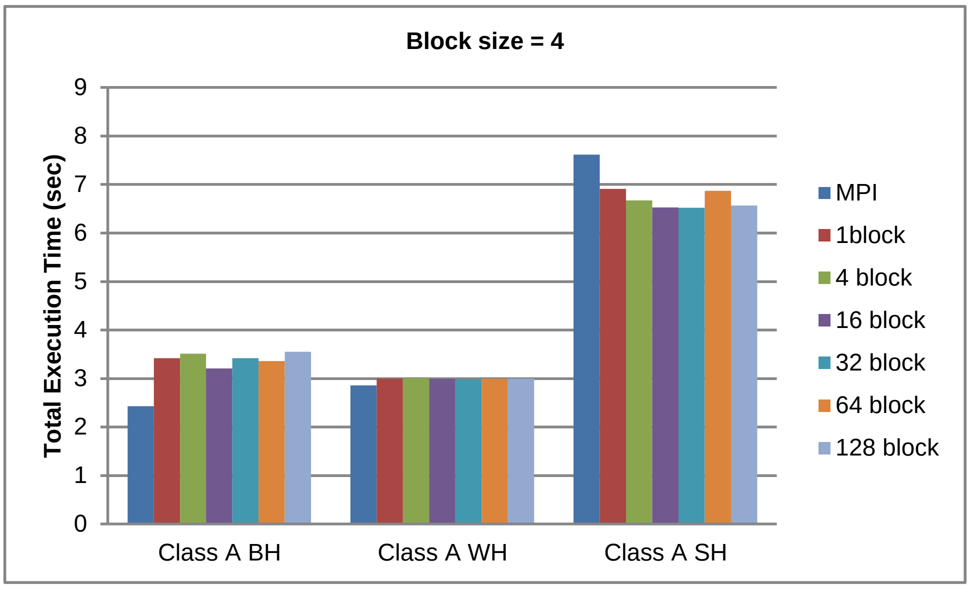


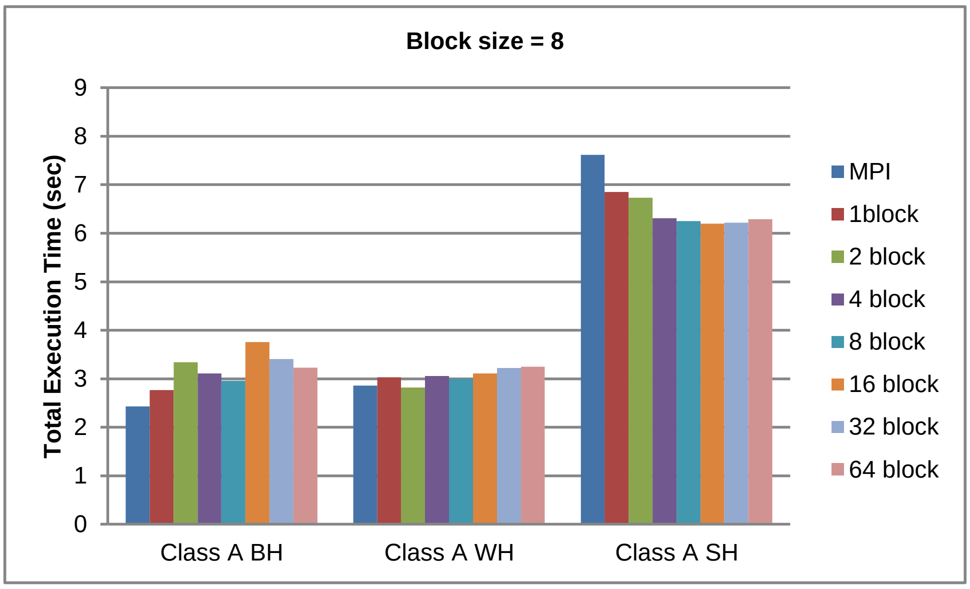
Figure1 Kernel time on CUDA with various thread block sizes vs. MPI

Figure1 shows the kernel computation time when it runs on the host (MPI bars) compared to when it runs on the device with various thread block sizes for different graph types of Class A. From the figure we can see that kernel computation on the host is two orders of magnitude larger than the kernel computation including *cudaMalloc* and *cudaMemcpy* on the device (~11000 vs. 110), indicating the significant performance benefits provided by CUDA device. Moreover, as the thread block size increases (the number of blocks is fixed to 1), the CUDA time decreases almost linearly due to the increased parallelism of CUDA threads provided.

(2) Total Execution Time on CUDA vs. MPI



(a) block size = 4



(b) block size = 8

Figure2 Total execution time of CUDA version vs. MPI version

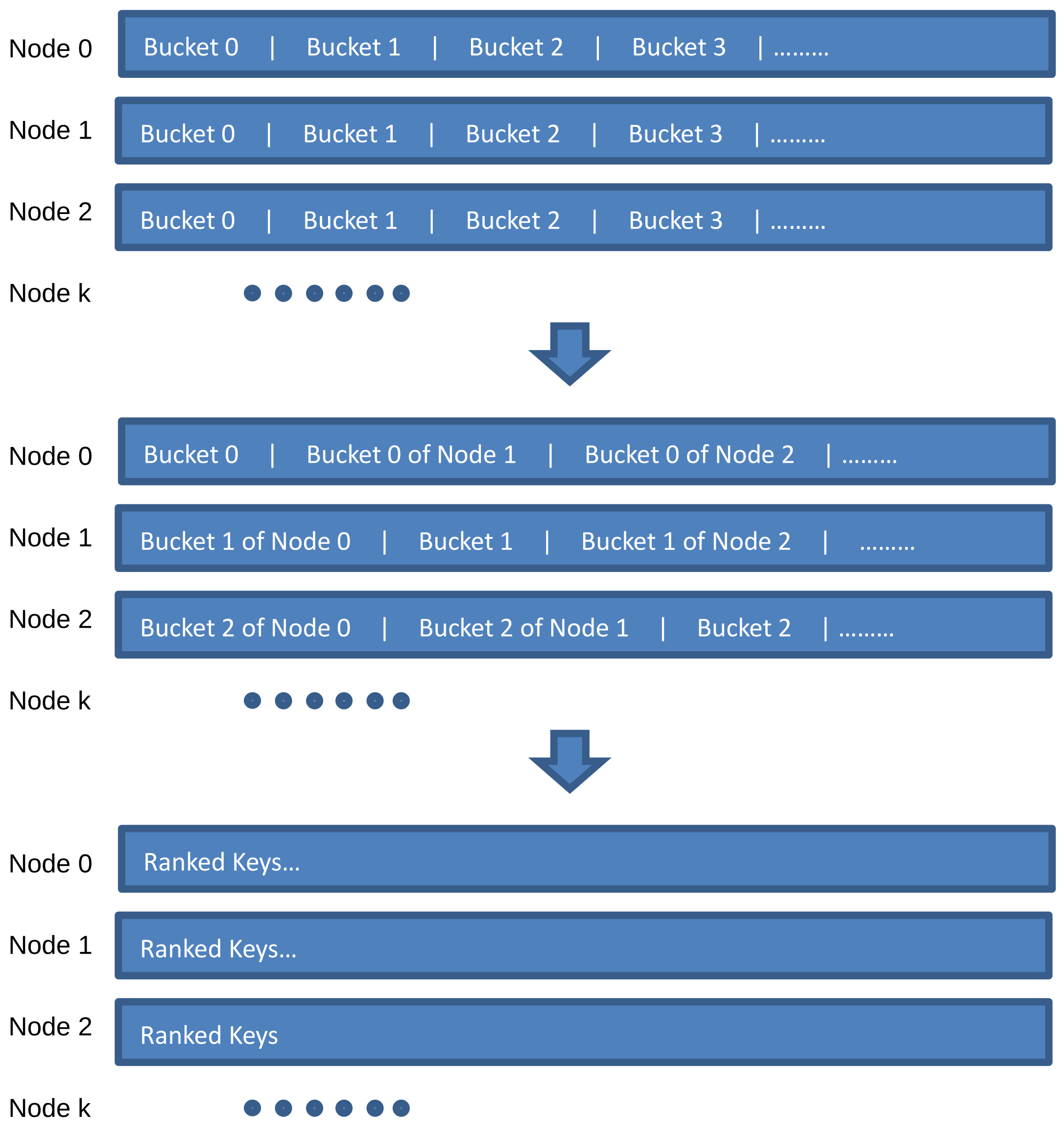
Figure 2 shows the entire execution time of DT benchmark when it runs only on the hosts (MPI bars) compared to when its kernel computation is running on the CUDA device with various numbers of blocks for (a) 4 threads per block and (b) 8 threads per block. From both figures we can see that kernel code running on the device does not contributes much to the entire execution although itself does show performance benefits (Figure 1) for communication graph BH and WH. For graph type of SH, CUDA version improve the overall performance. One possible reason for this might be DT benchmark is more sensitive to communication than computation while parallelizing the kernel on CUDA cannot reduce communication overheads in BH and WH. However, in SH, more nodes are involved in computation and more overlapping of communication in the graph, so parallelizing kernel computation on CUDA can improve total execution time.

Task 3: Parallelizing IS Benchmark on CUDA

Fei Meng (fmeng@ncsu.edu)

IS benchmark inside NAS Benchmark package is to parallel sort over small integers. The algorithm deployed inside the program is “bucket sorting”. Bucket sorting works by partitioning an array into a number of buckets, and each bucket is then sorted individually. In this parallel benchmark, every parallel node first sort its own numbers into individual buckets, then all the nodes exchange their own buckets with all the other nodes to get its own partition. In the end, everyone sort the final partition to get all numbers ranked correctly.

Here is the basic idea of the algorithm.


Node 0
Bucket 0 | Bucket 1 | Bucket 2 | Bucket 3 | ....
Node 1
Bucket 0 | Bucket 1 | Bucket 2 | Bucket 3 | ....
Node 2
Bucket 0 | Bucket 1 | Bucket 2 | Bucket 3 | ....
Node k
Node 0
Bucket 0 | Bucket 0 of Node 1 | Bucket 0 of Node 2 | ....
Node 1
Bucket 1 of Node 0 | Bucket 1 | Bucket 1 of Node 2 | ....
Node 2
Bucket 2 of Node 0 | Bucket 2 of Node 1 | Bucket 2 | ....
Node k
Node 0
Ranked Keys...
Node 1
Ranked Keys...
Node 2
Ranked Keys
Node k

1. Hotspots

Through gprof and the analysis of the program, the hotspots are across several domains. First, randomly generating the original numbers of every single node takes almost half of the final consuming time on that node. I was trying to do parallelism on this time-consuming part, but Dr. Mueller denied the work in this direction. The second hotspot of the program is on the MPI functions like “Allreduce” and “Alltoallv”. Since CUDA parallelism could do nothing on this part, I neglected parallelism here too. The final part of the benchmark which might be improved lies on three subroutines inside the rank function. Details of the analysis of these three routines is below in section two.

2. Algorithm Analysis

Three different routines parallelized in my program are as follows:

1) Bucket size initialize.

```
for( i=0; i<NUM_KEYS; i++ )
        bucket_size[key_array[i] >> shift]++;
```

Each bucket size is obtained after a full scan of the key array. Scanning of one key increases its corresponding bucket size by one. I use CUDA to parallelize this routine. Specifically, every thread is assigned to one key in the array and all the threads increase its own bucket size. An “atomicAdd” function is used inside the kernel to keep consistency of the shared size across the threads.

2) Sorting local numbers according to the bucket size, i.e assigned numbers to the right Positions to be prepared for exchange with other nodes.

```
for( i=0; i<NUM_KEYS; i++ )
{
        key = key_array[i];
        key_buff1[bucket_ptrs[key >> shift]++] = key;
}
```

Similar with routine one, each thread takes care of one key in the array and put it to the right position of the buffer array. “atomicAdd” is also used here.

3) Final calculation of the ranked keys. Each node count the numbers of equivalent keys and record this size.

```
for( i=0; i<j; i++ )
        key_buff_ptr[key_buff2[i]]++;
```

Similar with routine one, each node do its own assignment to increase the size.

3. Simulation Results

I test my program w/ or w/o CUDA optimization using different configurations of the benchmark. CLASS A, B and C are all tested with NPROCS varies as 2, 4 and 8. With class B and C, the total running time seems to be quite different. With CUDA support, it runs much faster than before. Here are the gprof outcomes of all the combinations.

Config	% time of rank (w/ | w/o CUDA)		absolute time of rank (w/ | w/o CUDA)		overall time of IS (w | w/o CUDA)		Rank Speedup	Overall Speedup
A/2	0.71	34.72	0.01	0.75	1.41	2.16	75	1.53
A/4	0.81	24.67	0.01	0.37	1.23	1.5	37	1.22
A/8	1.06	19.79	0.01	0.19	0.94	0.96	19	1.02
B/2	1.48	37.05	0.08	3.19	5.4	8.61	39.88	1.59
B/4	1.3	23.87	0.06	1.58	4.63	6.62	26.33	1.42
B/8	0.78	28.53	0.02	0.89	2.58	3.12	44.5	1.39
C/2	0	39.08	0	15.32	16.95	39.2	N/A	2.31
C/4	0.54	34.71	0.01	6.82	8.18	19.65	682	2.4
C/8	1.57	19.94	0.19	3.68	12.08	18.46	19.37	1.53

From the results shown in the table above, GPU significantly improve the performance of rank function, with speedup even high as 682 for C/4. The overall performance remains similar because most of the time consumed in the program relies on the randlc(generating random numbers) and MPI routines. The first chart below is the comparison between rank w/ and w/o CUDA support.

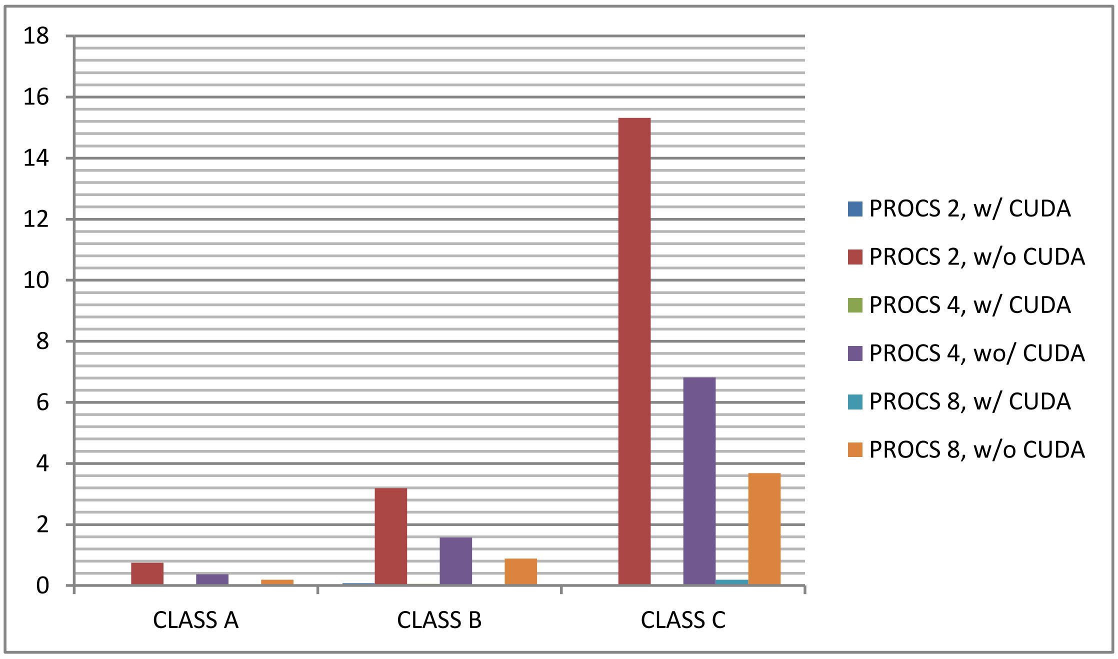


Figure 1 runtime of rank() w/ and w/o CUDA

The second chart is about the overall speedup of IS.

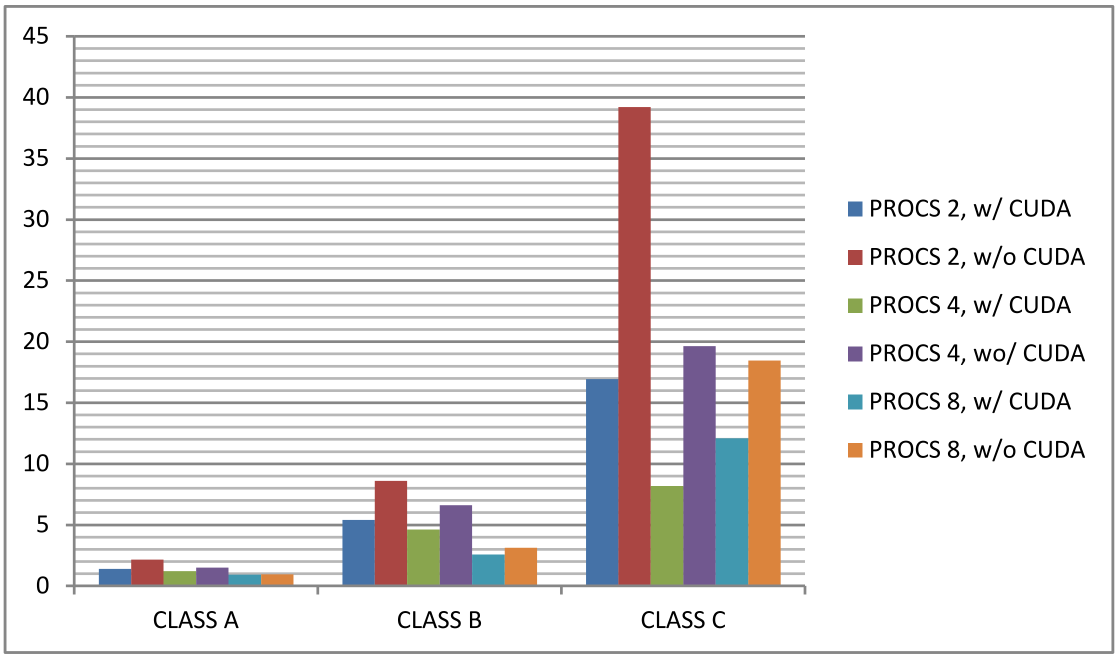


Figure 2 runtime of IS w/ and w/o CUDA