Assessing benefits of hardware acceleration using CUDA in an MPI environment.

URL:- http://www4.ncsu.edu/~axavier/ps-project.html

Single Author Info:
UserName:- Sreekanth Mavila(000916577)

Group Info:-
UserName:- Allen Pradeep Xavier(000935568)
UserName:- Anitta Jose(000920671)
UserName :- Sreekanth Mavila (000916577)

MILESTONES ACHIEVED:

1. Identifying the exact location of bottleneck in the function identified using gprof.

MG:- In the function **resid** which computes residual (using the formulae r = v - Au) there is a 3 level nested loop. In this there are 2 inner loops also. The time per call for this function is 0.07s and this is invoked 170 times. This causes a total execution time of 12.43s which accounts for 42.52% of the total execution time. This nested loop is the one which needs to be accelerated.

The **psinv** function also does some stencil computation inside a nested loop. This function takes 0.03 s to complete and is invoked 168 times. This accounts for a execution time of 5.74. The nested loop in this can also be accelerated.

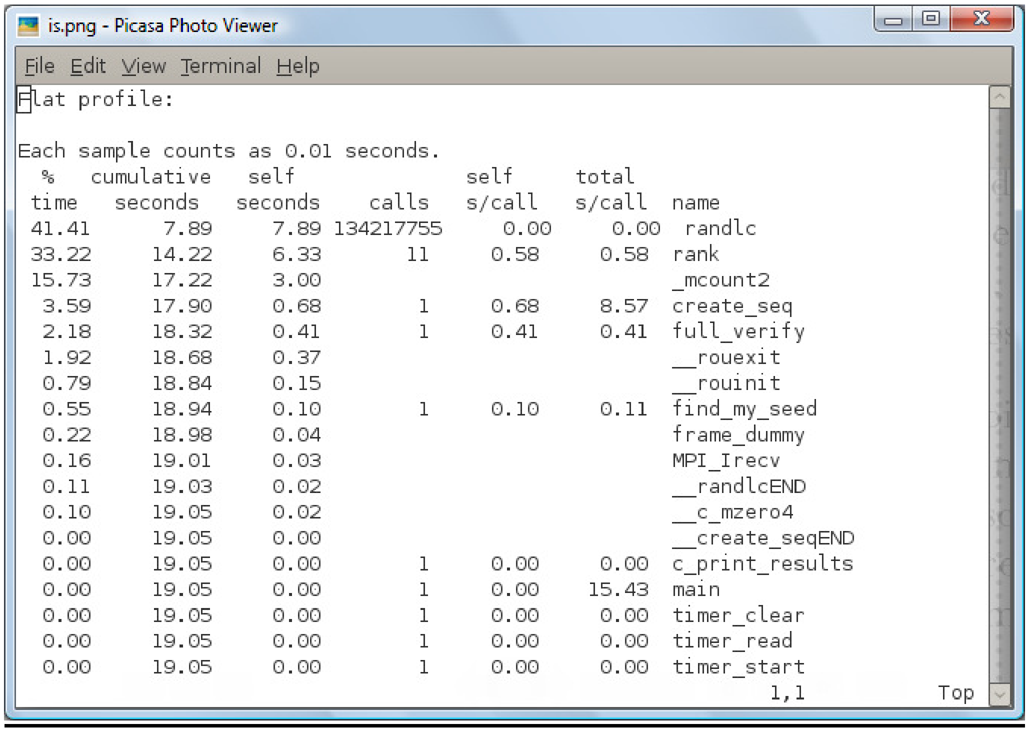
FT:- Bottleneck in computation was identified using the gprof tool. Analysis for class B with 4 processors showed that the following relevant results.

%time	cumulative seconds	self seconds	calls	selfs/call	total s/call	name
54.52	32.65	32.65	923648	0.00	0	fftz2_
11.39	39.47	6.82	44	0.15	0.61	cffts1_
10.64	45.84	6.37	22	0.29	0.9	cffts2_
6.61	49.8	3.96	22	0.18	0.18	transpose2_local_
5.99	53.39	3.59	22	0.16	0.16	transpose2_finish_
5.35	56.59	3.2	20	0.16	0.16	evolve_
1.54	57.51	0.92	112640	0.00	0	cfftz_

1.14	58.2	0.68				__fmth_i_dexp
0.7	58.62	0.42				net_recv
0.47	58.9	0.28	2	0.14	0.14	compute_indexmap_
0.37	59.12	0.22	128	0.00	0	vranlc_
0.31	59.31	0.19	2	0.09	0.27	compute_initial_conditions_
0.2	59.43	0.12	166	0.00	0	randlc_

As we can see from the above table, sub routine ***fftz2*** is taking around 54.52%. But the number of calls made to the function is 923648 and is computationally less intensive. Analysis further shows that subroutines ***cffts1*** and ***cffts2*** takes around 6.5 seconds and being considerably intensive computation.

IS:-This benchmark does sorting of large key set by ranking them according to their value. Initially the entire set is distributed among all the processors present. Bucket sorting logic is used for ranking. The keys in one processor are initially put into different buffers (buckets) based on the key value. Rank of a key depends on its position. Then the NUMKEYS(Maximum number of keys in a processor)keys from lower buckets is moved to 1st processor ,next NUMKEYS in the 2nd processor and likewise using MPI communication calls and then the next iteration is performed . There are 10 buckets and 10 iterations being performed which makes sure the key set is properly sorted. There are different levels of verification (partial verification and full verification) being done to ensure a sorted key set. This is not a computation intensive function.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 41.41      7.89     7.89 134217755     0.00     0.00  randlc
 33.22     14.22     6.33       11     0.58     0.58  rank
 15.73     17.22     3.00                             _mcount2
  3.59     17.90     0.68        1     0.68     8.57  create_seq
  2.18     18.32     0.41        1     0.41     0.41  full_verify
  1.92     18.68     0.37                             __rouexit
  0.79     18.84     0.15                             __rouinit
  0.55     18.94     0.10        1     0.10     0.11  find_my_seed
  0.22     18.98     0.04                             frame_dummy
  0.16     19.01     0.03                             MPI_Irecv
  0.11     19.03     0.02                             __randlcEND
  0.10     19.05     0.02                             __c_mzero4
  0.00     19.05     0.00                             __create_seqEND
  0.00     19.05     0.00        1     0.00     0.00  c_print_results
  0.00     19.05     0.00        1     0.00    15.43  main
  0.00     19.05     0.00        1     0.00     0.00  timer_clear
  0.00     19.05     0.00        1     0.00     0.00  timer_read
  0.00     19.05     0.00        1     0.00     0.00  timer_start
```

The only computation involved is the generation of keys, which might be parallelized.

2. **Writing CUDA Kernel. Decide on Fortran/C/Using FORTRAN to CUDA compiler / Using PGI FORTRAN CUDA compiler.**

Below are the options we tried.

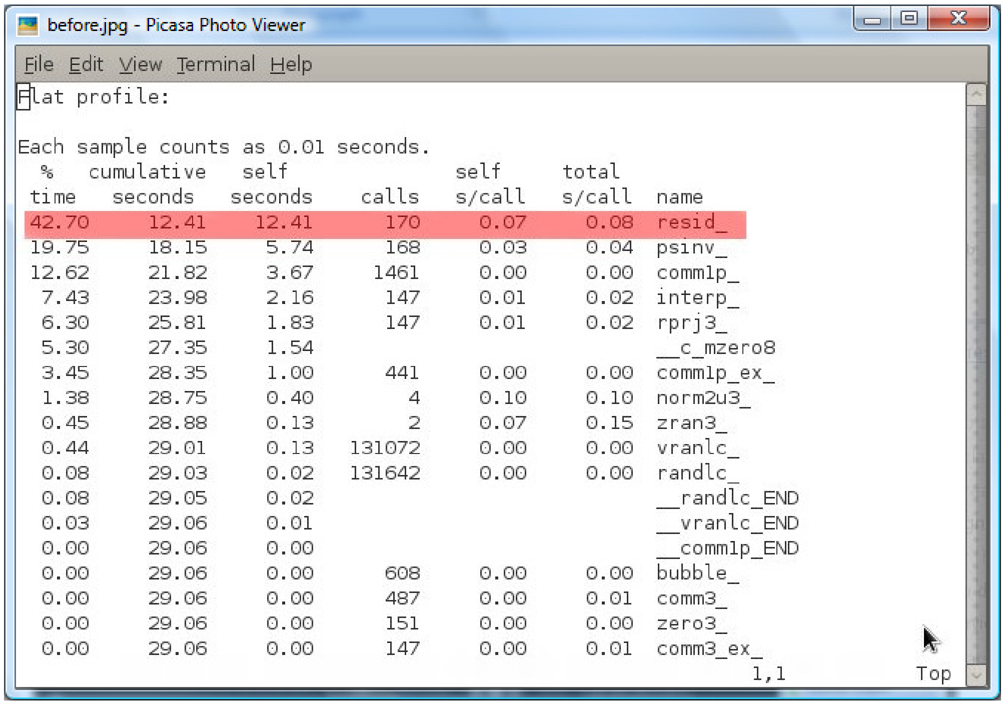
a. F2C-ACC: - The specification says that this will convert a FORTRAN code to CUDA/C. When we tried for conversion to CUDA we got 2 kinds of errors which are
    a. “Language Construct not supported” for keywords like “common”.
    b. “Type not supported” for keywords like “double precision” , “external”

b. f2c:- This is used for converting a FORTRAN code to C. We were not able to successfully use this as because there was nothing specified about how to specify the included files. When we executed f2c it gave undefined error for all the MPI calls.

c. PGI Accelerator: - The PGI 9.0 release includes the PGI Accelerator™ FORTRAN and C99 compilers supporting x64+NVIDIA Linux systems; PGF95 and PGCC accelerator compilers are supported on all Intel and AMD x64 processor-based systems with CUDA-enabled NVIDIA GPUs. This has option for specifying directives similar to OpenMP which can be used for acceleration. The PGI Accelerator compilers automatically analyze whole program structure and data, split portions of the application between the x64 CPU and GPU as specified by user directives, define and generate an optimized mapping of loops to automatically use the parallel cores, hardware threading capabilities and SIMD vector capabilities of modern GPUs. We were able to compile the codes using PGF95 and PGCC. So we have decided to go ahead with pgi compiler.

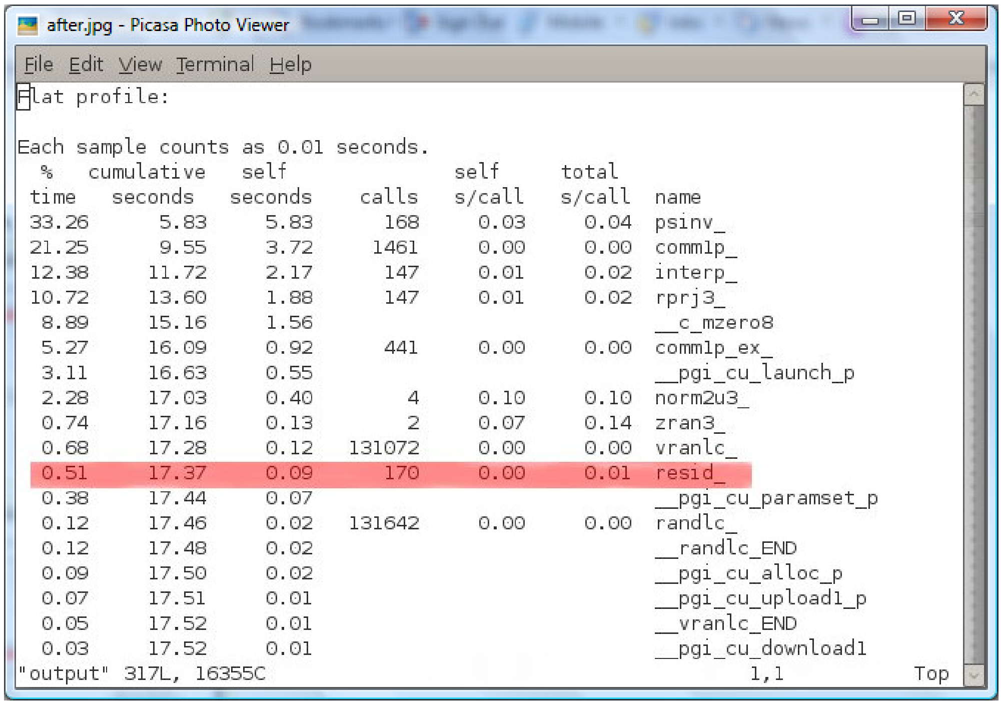
CURRENT STATUS:

MG: - We added pgi accelerator compiler directive in the **resid** function for the nested loop. It showed improvement in this particular function. Below given diagrams shows the improvement before and after adding compiler directive..


before.jpg - Picasa Photo Viewer
File Edit View Terminal Help

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 42.70     12.41    12.41      170     0.07     0.08  resid_
 19.75     18.15     5.74      168     0.03     0.04  psinv_
 12.62     21.82     3.67     1461     0.00     0.00  comm1p_
  7.43     23.98     2.16      147     0.01     0.02  interp_
  6.30     25.81     1.83      147     0.01     0.02  rprj3_
  5.30     27.35     1.54                             __c_mzero8
  3.45     28.35     1.00      441     0.00     0.00  comm1p_ex_
  1.38     28.75     0.40        4     0.10     0.10  norm2u3_
  0.45     28.88     0.13        2     0.07     0.15  zran3_
  0.44     29.01     0.13   131072     0.00     0.00  vranlc_
  0.08     29.03     0.02   131642     0.00     0.00  randlc_
  0.08     29.05     0.02                             __randlc_END
  0.03     29.06     0.01                             __vranlc_END
  0.00     29.06     0.00                             __comm1p_END
  0.00     29.06     0.00      608     0.00     0.00  bubble_
  0.00     29.06     0.00      487     0.00     0.01  comm3_
  0.00     29.06     0.00      151     0.00     0.00  zero3_
  0.00     29.06     0.00      147     0.00     0.01  comm3_ex_
                                                  1,1           Top
```



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 33.26      5.83     5.83      168     0.03     0.04  psinv_
 21.25      9.55     3.72     1461     0.00     0.00  comm1p_
 12.38     11.72     2.17      147     0.01     0.02  interp_
 10.72     13.60     1.88      147     0.01     0.02  rprj3_
  8.89     15.16     1.56                             __c_mzero8
  5.27     16.09     0.92      441     0.00     0.00  comm1p_ex_
  3.11     16.63     0.55                             __pgi_cu_launch_p
  2.28     17.03     0.40        4     0.10     0.10  norm2u3_
  0.74     17.16     0.13        2     0.07     0.14  zran3_
  0.68     17.28     0.12   131072     0.00     0.00  vranlc_
  0.51     17.37     0.09      170     0.00     0.01  resid_
  0.38     17.44     0.07                             __pgi_cu_paramset_p
  0.12     17.46     0.02   131642     0.00     0.00  randlc_
  0.12     17.48     0.02                             __randlc_END
  0.09     17.50     0.02                             __pgi_cu_alloc_p
  0.07     17.51     0.01                             __pgi_cu_upload1_p
  0.05     17.52     0.01                             __vranlc_END
  0.03     17.52     0.01                             __pgi_cu_download1
"output" 317L, 16355C                             1,1           Top
```

Fig:- After adding compiler directive.

But the problem we faced was that once we added the compiler directive the total execution time increased by 3 times. Now we are analyzing this problem.

<u>**FT**</u>**: -** Based on the understanding from the gprof we understood the computation is intensive for the subroutines called during the FFT computation: ***cffts1****,* ***cffts2****,* ***cffts3*****,** each called for different layouts given. Code walk though identified that all these functions calls the subroutine ***cfftz*** which in turn calls subroutine ***fftz****.* This accounts for the large number of calls for the functions ***cfftz*** and ***fftz****.* One option for accelerating the code will be writing these nested subroutines inline to the main functions and then applying the CUDA directives.

Similar to MG we tried adding directives for FT so that we can confirm our analysis was proper. But we figured out that the compiler directive is not taking effect. Today we repeated the same as above with MG. We found out that the directives are not taking effect.

IS:- Trying to figure out how to optimize the rank function which is not very computation intensive. We are also looking at optimizing the random number generator function.

<u>TASK DISTRIBUTION:</u>

Task Owner	Task
Group Task	Analysis of F2C-ACC, f2c, installation of pgi compiler environment settings, added compiler directive for acceleration, compilation using the pgi accelerator directives.
Allen Pradeep Xavier	Performance evaluation of MG, Identifying the bottleneck function, performance evaluation after adding compiler directive.
Anitta Jose	Performance evaluation of IS, Identifying the bottleneck function, Understanding the IS implementation.
Sreekanth Mavila	Performance evaluation of FT, Identifying the bottleneck function, Converting all the function calls in the hotspot function to inline functions.

<u>Milestones for the coming weeks</u>

1. Investigating the cause for pgi compiler directive not taking effect in the recent execution.
2. Investigating the cause slow down in the overall execution time when the directive is added
3. Identifying the correct location for optimization of IS as the hotspot function shown as per gprof is not computation intensive
4. If directive is not supported write kernel functions for CUDA.