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

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

Single Author Info:
UserName:- Anitta Jose(000920671)

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

Project Description:

The goal of the project is to analyze the benefits of hardware acceleration using CUDA in an MPI environment. We used 3 different NAS Parallel Benchmarks for the analysis. The benchmarks chosen are MG, FT and CG.

Tools Used:

Gprof: Gprof is a profiling program which collects and arranges statistics on your programs.Basically, it looks into each of your functions and inserts code at the head and tail of each one to collect timing information. We found out the bottleneck function using this and then tried to perform optimizations on that.

pgf95: The PGI 9.0 release includes the PGI Accelerator™ FORTRAN and C99 compilers supporting x64+NVIDIA Linux systems. 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.

In addition to the accelerator capability this can be used as a fortran Cuda compiler.

Metrics for evaluation: We are using the **Mops/s** and **total execution time** for the comparison of the optimization applied to that of the original code. Once we move the hotspot loop to cuda kernel Gprof doestn't shows much time with the bottleneck function. This may be because it does not record the cuda kernel time.

Optimization Approach and Results:

MG:

Introduction:

The gProf analysis on MG showed the bottleneck in the resid function. On analysis of the function revealed these nested loops:

```
 do i3=2,n3-1
    do i2=2,n2-1
       do i1=1,n1
          u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
     >           + u(i1,i2,i3-1) + u(i1,i2,i3+1)
          u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
     >           + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
       enddo
       do i1=2,n1-1
          r(i1,i2,i3) = v(i1,i2,i3)
     >                - a(0) * u(i1,i2,i3)
c-------------------------------------------------------------------
cssume a(1) = 0      (Enable 2 lines below if a(1) not= 0)
c-------------------------------------------------------------------
c    >                - a(1) * ( u(i1-1,i2,i3) + u(i1+1,i2,i3)
c    >                       + u1(i1) )
c-------------------------------------------------------------------
     >                - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
     >                - a(3) * ( u2(i1-1) + u2(i1+1) )
       enddo
    enddo
 enddo
```

So we focused on converting this part of code to CUDA. We used PGI directives for accelerating this section of code.

Approach, Issues and Results:

1)The arrays u1 and u2 are shared by the 2 inner loops, so when we applied the directives we got the error saying that the section cannot be parallelized as the array is shared. So we merged these 2 inner loops to the following:

```
do i3=2, n3-1
  do i2=2, n2-1
    do i1=2,n1-1
      u3 = u(i1-1,i2-1,i3) + u(i1-1,i2+1,i3) + u(i1-1,i2,i3-1) + u(i1-1,i2,i3+1)
      u4 = u(i1+1,i2-1,i3) + u(i1+1,i2+1,i3) + u(i1+1,i2,i3-1) + u(i1+1,i2,i3+1)
      u5 = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1) + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
      u6 = u(i1-1,i2-1,i3-1) + u(i1-1,i2+1,i3-1) + u(i1-1,i2-1,i3+1) + u(i1-1,i2+1,i3+1)
      u7 = u(i1+1,i2-1,i3-1) + u(i1+1,i2+1,i3-1) + u(i1+1,i2-1,i3+1) + u(i1+1,i2+1,i3+1)
      r(i1,i2,i3) = v(i1,i2,i3) - a(0) * u(i1,i2,i3) - a(2) * (u5 + u3 + u4) - a(3) * (u6 + u7)
```

```
   enddo
 enddo
enddo
```

Instead of calculating the array values u1 and u2 for all i and then calculating r, we stored u1(i-1), u1(i+1), u2(i), u2(i-1), u2(i+1)(which are used in the calculation of the r array) into local variables u3, u4, u5, u6, u7 and calculated r for each i. Even thought this meant extra calculations this is acceptable in this case as the GPGPU is good in delivering the associated performance involved.

The final code which was used to test the performance was given below with the directives:

```
        u8=a(0)
        u9=a(2)
        u10=a(3)
!$acc region copyin(u,v,u8,u9,u10) copyout(r)
!$acc do private(u3,u4,u5,u6,u7)
        do i3=2, n3-1
        do i2=2, n2-1
            do i1=2,n1-1

              u3 = u(i1-1,i2-1,i3) + u(i1-1,i2+1,i3)
     >             + u(i1-1,i2,i3-1) + u(i1-1,i2,i3+1)
               u4 = u(i1+1,i2-1,i3) + u(i1+1,i2+1,i3)
     >             + u(i1+1,i2,i3-1) + u(i1+1,i2,i3+1)
               u5 = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
     >             + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
               u6 = u(i1-1,i2-1,i3-1) + u(i1-1,i2+1,i3-1)
     >             + u(i1-1,i2-1,i3+1) + u(i1-1,i2+1,i3+1)
               u7 = u(i1+1,i2-1,i3-1) + u(i1+1,i2+1,i3-1)
     >             + u(i1+1,i2-1,i3+1) + u(i1+1,i2+1,i3+1)
              r(i1,i2,i3) = v(i1,i2,i3)
     >                    - u8 * u(i1,i2,i3)
     >                    - u9 * ( u5 + u3 + u4 )
     >                    - u10 * ( u6 + u7 )
          enddo
         enddo
      enddo
!$acc end region
```

We had also replaced the array a by the variables u8, u9, u10 as the array ‘a’ was giving "Unaligned memory access" error.

The performance of this modified code was a little slower than the main code when it was run on 1 processor with class B input. We figured out that the operation involves copying the large 3d arrays u, v and copying out a large array r. The large memory copy operations

took up all the time. This was also evident when we tried to run class C input in a single processor, when we got a CUDA memalloc error.

So we tried the running the code for a larger number of processors, so that the input array might be divided among multiple processors and matrix to be copied from each processor to its GPGPU will be reduced and the performance might be improved. We ran MG with class C for 8 processors, where we could notice a little performance improvement. The results are given below:

Results for Normal code:

MG Benchmark Completed.
Class = C
Size = 512x 512x 512
Iterations = 20
Time in seconds = 84.35
Total processes = 8
Compiled procs = 8
Mop/s total = 1845.76
Mop/s/process = 230.72
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 06 Dec 2009

Results for CUDA code:

MG Benchmark Completed.
Class = C
Size = 512x 512x 512
Iterations = 20
Time in seconds = 81.41
Total processes = 8
Compiled procs = 8
Mop/s total = 1912.45
Mop/s/process = 239.06
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 06 Dec 2009

2) Since there are heavy memory operations involved, we tried improving the performance by the use of blocking. This actually improved the performance for a single processor too. The changed code for blocking is posted below:

```
do i3block=2, n3-1, BLOCK2
 do i2block=2, n2-1, BLOCK3
  do i3=i3block,min(i3block+BLOCK3-1,n3-1)
   do i2=i2block,min(i2block+BLOCK2-1,n2-1)
    do i1=2,n1-1
     u3 = u(i1-1,i2-1,i3) + u(i1-1,i2+1,i3) + u(i1-1,i2,i3-1) + u(i1-1,i2,i3+1)
     u4 = u(i1+1,i2-1,i3) + u(i1+1,i2+1,i3) + u(i1+1,i2,i3-1) + u(i1+1,i2,i3+1)
     u5 = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1) + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
     u6 = u(i1-1,i2-1,i3-1) + u(i1-1,i2+1,i3-1) + u(i1-1,i2-1,i3+1) + u(i1-1,i2+1,i3+1)
     u7 = u(i1+1,i2-1,i3-1) + u(i1+1,i2+1,i3-1) + u(i1+1,i2-1,i3+1) + u(i1+1,i2+1,i3+1)
     r(i1,i2,i3) = v(i1,i2,i3) - u8 * u(i1,i2,i3) - u9 * ( u5 + u3 + u4 ) - u10 * ( u6 + u7 )
    enddo
   enddo
  enddo
 enddo
enddo
```

[Referred the paper "Optimizing the NPB MG benchmark for multi-core AMD Opteron microprocessors" by Stephen Whalen, Cray, Inc.]

There was an improvement in performance for even a single processor with class B input. The results are given below:

Original Code:

```
MG Benchmark Completed.
Class           =                        B
Size            =          256x 256x 256
Iterations      =                       20
Time in seconds =                    29.90
Total processes =                        1
Compiled procs  =                        1
Mop/s total     =                   650.92
Mop/s/process   =                   650.92
Operation type  =          floating point
Verification    =             SUCCESSFUL
Version         =                      3.3
Compile date    =              06 Dec 2009
```

Modified code with Blocking:

```
MG Benchmark Completed.
Class           =                        B
Size            =          256x 256x 256
Iterations      =                       20
Time in seconds =                    27.11
Total processes =                        1
```

Compiled procs = 1
Mop/s total = 717.89
Mop/s/process = 717.89
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 06 Dec 2009

3) So the next step was to combine blocking with CUDA. But this did not work as expected. The accelerator directives were simply ignored when applied on the blocked version of the loop as the kernel could not be split.

We also tried making the first 2 loops, which take care of blocking factor run in the host by using the directive "!$acc do host" and then had the normal "!$acc do" directive for the inner loops, but the PGI compiler stalled when the input was given.

Open Issues:

Instead of copying the entire array, we could do so in blocks and try to further improve the performance. Since large memory copy is involved before computations, much of performance could not be achieved from the GPGPU.

FT:

Introduction

We started off the project by analyzing the gprof output from the NAS PB FT code. We were able to find out the major hot spots in code. The interim report showed the result and the main functions which may be optimized where *fftz2, cfftz and transpose2_local.* We used PGI compiler for optimizing the FT code on CUDA target.

Approach, Issues and Resolution

- fftz was the major function which caused bottleneck. There were 100s of thousands of call made to fftz2 and hence transforming the CUDA will cause too many overheads due to large number of kernel invocations. We tried expanding the function by inline expansion of fftz. But the results were not fruitful.

- cffts2 and cffts3 had too many library calls which prevented the transformation of the code to CUDA.

- Another function which showed bottleneck was cfftz. This also had the same problem as fftz2, too many invocations.

- We tried applying PGI accelerator directives in all the above mentioned functions. But, PGI Accelerator directive doesn't have any support for all the variables used in the benchmark code, for example "complex" data types.

Due to the reasons mentioned above, we tried to optimized the function *transpose2_local.* Due to the above mentioned limitations of PGI accelerator directives, we did the transformation by writing Fortran CUDA code. PGI provide us the compiler to build CUDA Fortran code. The only dependency issue is that PGI compiler for CUDA did not support few FORTRAN keywords. Hence we had to change those keywords and data types to make the code work. The file should have ".cuf" extension also.

Below is the code loop which we tried to optimize

```
     if (n1 .lt. transblock .or. n2 .lt. transblock) then
        if (n1 .ge. n2) then
           do j = 1, n2
              do i = 1, n1
                 xout(j, i) = xin(i, j)
              end do
           end do
        else
           do i = 1, n1
              do j = 1, n2
                 xout(j, i) = xin(i, j)
              end do
           end do
        endif
     else
        do j = 0, n2-1, transblock
           do i = 0, n1-1, transblock

!-----------------------------------------------------------------------
! Note: compiler should be able to take j+jj out of inner loop
!-----------------------------------------------------------------------
              do jj = 1, transblock
                 do ii = 1, transblock
                    z(jj,ii) = xin(i+ii, j+jj)
                 end do
              end do

              do ii = 1, transblock
                 do jj = 1, transblock
                    xout(j+jj, i+ii) = z(jj,ii)
                 end do
              end do

           end do
        end do
     endif
```

Below is the kernel module corresponding to the above loop

```
module ker_mod
        use cudafor
            contains
            attributes(global) subroutine kernel1( XIN, XOUT, N1, N2)

!-----------------------------------------------------------------
!-----------------------------------------------------------------

                    integer, value :: N1,N2
                    real :: XIN(N1,N2), XOUT(N2,N1)
                    integer :: tx,i,ty,j

                            tx = threadidx%x
                            ty = threadidx%y
                            i = (blockidx%x-1) * 16 + tx
                            j = (blockidx%y-1) * 16 + ty

                            if (i .lt. N1 .and. j .lt. N2) XOUT(j,i) = XIN(i,j)

            end subroutine kernel1
    end module ker_mod
```

Below is the code snippet which invokes the above kernel

```
      !Create the grid and block dimensions
          dimGrid = dim3( ii, jj, 1 )
          dimBlock = dim3( 16, 16, 1 )
          !New Code Ends here

      if (timers_enabled) call timer_start(T_transxzloc)

!---------------------------------------------------------------------
!c If possible, block the transpose for cache memory systems.
!c How much does this help? Example: R8000 Power Challenge (90 MHz)
!c Blocked version decreases time spend in this routine
!c from 14 seconds to 5.2 seconds on 8 nodes class A.
!---------------------------------------------------------------------

          !dimGrid = dim3( ii, jj, 1 )
          !dimBlock = dim3( 16, 16, 1 )
      call kernel1<<<dimGrid,dimBlock>>> (xcin, xcout, n1, n2)
```

Results for Normal Code
FT Benchmark Completed.
Class = A
Size = 256x 256x 128
Iterations = 6
Time in seconds = 13.37
Total processes = 1
Compiled procs = 1
Mop/s total = 533.72
Mop/s/process = 533.72
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 05 Dec 2009

Results for CUDA
FT Benchmark Completed.
Class = A
Size = 256x 256x 128
Iterations = 6
Time in seconds = 13.43

Total processes = 1
Compiled procs = 1
Mop/s total = 531.39
Mop/s/process = 531.39
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 05 Dec 2009

The results show slight decrease in the performance though both show very similar results. Our assumption is that if we increase the problem size we can get an improvement as happened in MG and CG. But we could not run CLASS=B for NPROC=1 . It looks like we need more than 1 processors for CLASS=B for FT. But inorder to run in more than one processor we need to change the kernel subroutine so as to take into account task id also.

Open Issues

Make necessary changes in the kernel subroutine so as check the results for bigger problem size and more number of processors.

CG

Introduction

CG approximates the largest eigenvalue of a sparse, symmetric, positive definite matrix, using inverse iteration[2]. This algorithm comes up with an eigenvalue estimate for a number of iterations("outer iterations", different values depending on the CLASS) using 25 iterations of the conjugate gradient method[2].

Approach, Issues and Results

Gprof gave the bottleneck function as conj_grad. From the implementation details we see that it is the matrix-vector multiplication that is the major bottleneck in this particular function[1][2]. Below is the function which we need to parallelize. We used accelerator directive to optimize.

```
c-----------------------------------------------------------------------
c  q = A.p
c  The partition submatrix-vector multiply: use workspace w
c-----------------------------------------------------------------------
        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
        enddo
```

- **Issue with copying p:** We added the compiler directive around the loop as below

```
c-----------------------------------------------------------------------
c  q = A.p
c  The partition submatrix-vector multiply: use workspace w
c-----------------------------------------------------------------------
!$acc region
        do j=1,lastrow-firstrow+1
           sum = 0.d0
           do k=rowstr(j),rowstr(j+1)-1
              sum = sum + a(k)*p(colidx(k))
           enddo
           w(j) = sum
        enddo
!$acc end region
```

It gave compilation error as below.

```
conj_grad:
   1123, Accelerator region ignored
   1124, Accelerator restriction: size of the GPU copy of an array depends on va
lues computed in this loop
   1126, Accelerator restriction: size of the GPU copy of 'p' is unknown
         Accelerator restriction: one or more arrays have unknown size
```

Defining the array size for p solved the issue.

- **Issue with privatizing "sum": variable:** Then there was another compilation error as below. It got resolved by privatizing sum variable.

```
PGF90-F-0000-Internal compiler error. no matching ref for symbol     697 (cg.f:
1408)
PGF90/x86-64 Linux 10.0-0: compilation aborted
```

Once these issues were resolved we ran this for CLASS=B and CLASS C. Below are the results. The accelerator directives showed a clear improvement in the execution time. For CLASS=C the improvement was almost 2X with NPROCS=1.

CLASS	**NPROCS**	**Problem Size**	**Mops/s(per processor)without directives**	**Mops/s (per processor)with directives**	**Total Time in seconds without directives**	**Total Time in seconds(with directives)**
B	1	75000	110.74	150.22	494.01	364.20
C	1	150000	80.44	151.40	1782.02	946.80
C	2	150000	82.52	106.72	868.57	671.63

We see that as the problem size increases we get much improvement.

Open Issues: As the number of processors being used increase there is a decrease in the performance achieved. As seen with the CLASS=C, with NPROCS=1 there was a 2X improvement with hardware acceleration. But with NPROCS=2 we don't get 2X improvement.

IS:

First we tried optimization in IS. But the operations in IS are all memory intensive. There are very less computation involved. The only function which is comparatively computation intensive is the **randlc,** the random number generator. But there are so many function calls

for this. If we change this calculation to Cuda kernel, cost of kernel invocation will nullify the effect of any improvement we achieve.

Inorder to try out whether any optimization is achieved we tried applying accelerator directives to many different loop. We could not apply because of 2 reasons

- There was too much dependency in the loop iterations which made parallelization using directives impossible.
- There were some computations involving INT_TYPE2 which maps to type long, which is currently not supported by PGI accelerator.

Conclusion:

The above analysis shows that we are able to achieve improvement in terms of execution time and Mops/s with hardware acceleration. We get better performance improvement as we increase the problem size.

TASK DISTRIBUTION:

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 and blocking Web page updation, Helping in solving errors in FT,CG benchmarks
Anitta Jose	Performance evaluation of IS, Identifying the bottleneck function, Understanding the IS implementation, Identifying bottleneck in CG benchmark, Performance evaluation for CG after adding compiler directives
Sreekanth Mavila	Performance evaluation of FT, Identifying the bottleneck function, Converting all the function calls in the hotspot function to inline functions. Writing cuda fortran kernel, Performance evaluation with cuda fortran kernel, Helping in solving errors in MG,CG benchmarks

References

1. http://www.nersc.gov/nusers/systems/franklin/programming/cg_opt.pdf(Optimizing the NPB CG benchmark for multi-core AMD Opteron microprocessors by Stephen Whalen, Cray, Inc. August 29, 2007)

2. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Report RNR-94-007, NASA Advanced Supercomputing Division, March 1994.

3. http://www.nersc.gov/nusers/systems/franklin/programming/mg_opt.pdf