Cuda Acceleration for BoomerAMG implementation

Keerthana Boloor
Poonam Shidlyali
Karthikeyan Sivaraj

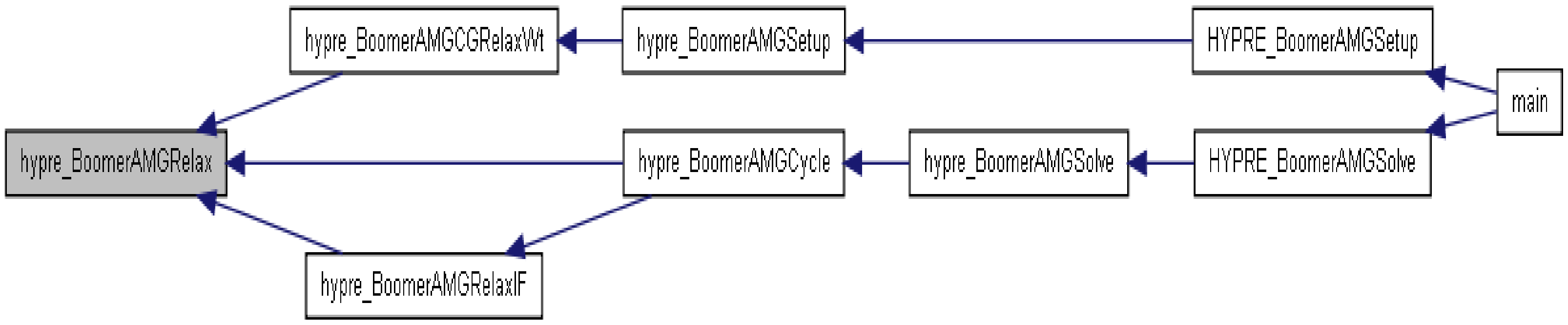
Progress till date:

All:
We have all completed a read through of the code and understand what needs to be done for porting AMG to Cuda.

We have decided that we will first port basic low level (common to all 3 problems) calls to Cuda. We have arrived at this decision because of the extensive use of dynamic memory allocation in all the calls. So we have decided to move the portions of code in AMG where we know the amount of memory to be allocated. Below lists a set of functions each team member has ported partially (iterations only) to CUDA. We have compiled this without openmp and ported all the openmp calls to CUDA along with time consuming iterations in the below given functions.

1.
```
int hypre_BoomerAMGRelax(hypre_ParCSRMatrix *A,
                         hypre_ParVector    *f,
                         int                *cf_marker,
                         int                 relax_type,
                         int                 relax_points,
                         double              relax_weight,
                         double              omega,
                         hypre_ParVector    *u,
                         hypre_ParVector    *Vtemp )
```

Ported by **Poonam**
The importance of the function in the solver is as shown by the caller graph:



We port the pieces of code which are most time consuming in this function (the iterations) as shown below[Please look at line 1638 of file par_relax.c]. We have provided this code as an example.

Memory allocated for:
```
tmp_data
A_diag_data
f_data
A_diag_i
A_offd_i
```

```
A_offd_j
u_data
Vext_data
```

Code in CUDA

```
__global__ void relax_on_CUDA(double* tmp_data, double* u_data, double *Vext_data,
double *A_diag_data, int* A_diag_i, int* A_offd_i, int* A_offd_j, double*
f_data,int n, int num_threads )
{

        int i,ii,j,jj,ns,ne,res,rest,size;
        unsigned int j = blockIdx.x * blockDim.x + threadIdx.x;

            size = n/num_threads;
            rest = n - size*num_threads;
            if (j < rest)
            {
               ns = j*size+j;
               ne = (j+1)*size+j+1;
            }
            else
            {
               ns = j*size+rest;
               ne = (j+1)*size+rest;
            }
            for (i = ns; i < ne; i++) /* interior points first */
            {

               /*-----------------------------------------------------------
                * If diagonal is nonzero, relax point i; otherwise, skip it.
                *-----------------------------------------------------------*/

               if ( A_diag_data[A_diag_i[i]] != zero)
               {
                  res = f_data[i];
                  for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++)
                  {
                     ii = A_diag_j[jj];
                     if (ii >= ns && ii < ne)
                     {
                         res -= A_diag_data[jj] * u_data[ii];
                     }
                     else
                         res -= A_diag_data[jj] * tmp_data[ii];
                  }
                  for (jj = A_offd_i[i]; jj < A_offd_i[i+1]; jj++)
                  {
                     ii = A_offd_j[jj];
                     res -= A_offd_data[jj] * Vext_data[ii];
                  }
                  u_data[i] = res / A_diag_data[A_diag_i[i]];
               }
            }
            for (i = ne-1; i > ns-1; i--)      /* interior points first */
            {

               /*-----------------------------------------------------------
                * If diagonal is nonzero, relax point i; otherwise, skip it.
                *-----------------------------------------------------------*/
```

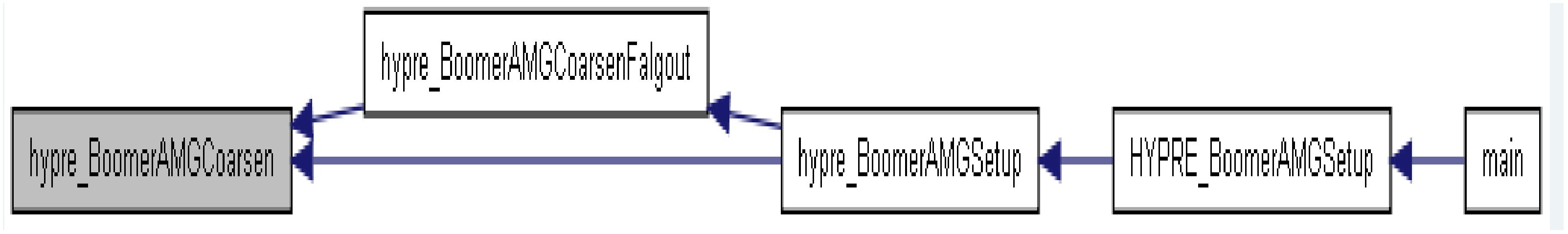
```
                    if ( A_diag_data[A_diag_i[i]] != zero)
                    {
                       res = f_data[i];
                       for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++)
                       {
                          ii = A_diag_j[jj];
                         if (ii >= ns && ii < ne)
                         {
                             res -= A_diag_data[jj] * u_data[ii];
                         }
                         else
                             res -= A_diag_data[jj] * tmp_data[ii];
                       }
                       for (jj = A_offd_i[i]; jj < A_offd_i[i+1]; jj++)
                       {
                          ii = A_offd_j[jj];
                          res -= A_offd_data[jj] * Vext_data[ii];
                       }
                       u_data[i] = res / A_diag_data[A_diag_i[i]];
                    }
                 }

         }
```

Issues:
Compilation errors

Other code change location for Cuda calls that we have changed are:

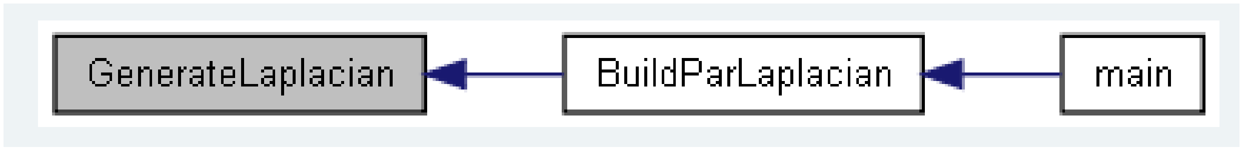
2. int hypre_BoomerAMGCoarsen(hypre_ParCSRMatrix   *S,
    hypre_ParCSRMatrix   *A,
    int   CF_init,
    int   debug_flag,
    int   **CF_marker_ptr)

Ported by **Keerthana**
This function consumes maximum time.
Caller Graph:



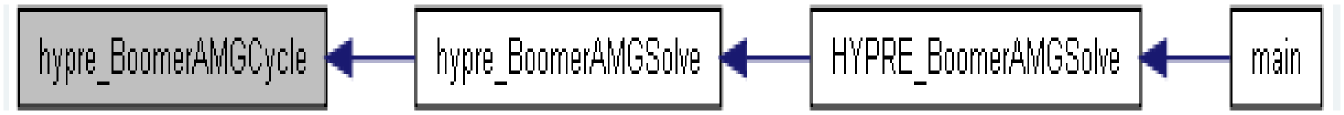
```
3.  HYPRE_ParCSRMatrix  GenerateLaplacian( MPI_Comm comm,
              HYPRE_BigInt      nx,
              HYPRE_BigInt      ny,
              HYPRE_BigInt      nz,
              int      P,
              int      Q,
              int      R,
              int      p,
              int      q,
              int      r,
              double  *value,
              HYPRE_ParVector *rhs_ptr,
              HYPRE_ParVector *x_ptr)
```

Ported by **Keerthana**



```
4. int hypre_BoomerAMGCycle( void  *amg_vdata, hypre_ParVector  **F_array, hypre_ParVector
 **U_array   )
```

Ported by **Karthik**.



In all these functions like shown in the example we have ported the iterations to Cuda by making sure that we have allocated enough memory in the shared memory space of each thread in the co-processor.

Issues: We are currently having compilation problems with all functions.

Scheduled work

1. Complete and evaluate the cuda migration –All – November 14th (We are having compilation errors in some functions, we will need to solve them to proceed)

2. Port higher function calls to Cuda via Cuda MPI in all 3 problems – All – November 21st

3. Evaluate the entire Laplacian calculation for various input sizes in CUDA – Keerthana – November 24th

4. Evaluate the entire Laplacian-27 pt calculation for various input sizes in CUDA – Poonam – November 24th

5. Evaluate the entire jumps calculation for various input sizes in CUDA – Karthik – November 24th