GPU IMPLEMENTATION OF HPGMG-FV

Nikolay Sakharnykh, Simon Layton, Kate Clark
HYBRID MULTIGRID
Heterogeneous implementation using CUDA + OpenMP

Fine grids are offloaded to GPU (TOC), coarse grids are processed on CPU (LOC)
HYBRID MULTIGRID

CPU-only: Haswell Xeon E5-2698 v3, 16-core, single socket
CPU + Kepler GPU: Ivy Bridge Xeon E5-2690 v2, 10-core, single socket + NVIDIA Tesla K40 with ECC on
CPU + Maxwell GPU: Sandy Bridge i7-3930K, 6-core + NVIDIA GeForce Titan X
HYBRID MULTIGRID

Threshold benchmark

GPU: NVIDIA Quadro M6000, CPU: Intel Ivy Bridge Xeon E5-2690 v2
HPGMG-FV entities naturally map to GPU hierarchy
UNIFIED MEMORY

No changes to data structures
No explicit data movements
Single pointer for CPU and GPU data

Minimal modifications to the original code:

(1) `malloc` replaced with `cudaMallocManaged` for levels accessed by GPU
(2) Invoke CUDA kernel if level size is greater than threshold
COMPUTE BOTTLENECK

Cost of operations

 MOST TIME SPENT ON STENCILS

 VOLUME

 SURFACE
STENCILS ON GPU
Parallelization strategies

3D thread blocks
Provides more than enough parallelism

2D thread blocks
Parallelize over XY plane and march along Z
STENCILS ON GPU
One kernel for many smoothers

```c
void smooth_kernel(level_type level,...) {
    // prologue
    const double * __restrict__ rhs = ...;
    ...
    for(k=0; k<dimz; k++) {
        // apply operator
        const double Ax = apply_op_ijk();

        // smoother
        #ifdef CHEBY
            xo[ijk] = x[ijk] + ... + c2*lambda*(rhs[ijk]-Ax);
        #elif JACOBI
            xo[ijk] = x[ijk] + (2.0/3.0)*lambda*(rhs[ijk]-Ax);
        #elif GSRB
            xo[ijk] = x[ijk] + RB[ij]*lambda*(rhs[ijk]-Ax);
        #endif
    }
}
```

THREAD BLOCK MARCHES FROM 0 TO DIMZ
CHEBYSHEV POLYNOMIALS
JACOBI
GAUSS SEIDEL RED-BLACK

APPLY STENCIL OPERATION POISSON, HELMHOLTZ (FV2,FV4)
STENCILS ON GPU
Optimization: register caching

38 REGS IN KERNEL WITHOUT STENCIL

// load k and k-1 planes into registers
double xc0 = x[ijk - kStride];
double xc1 = x[ijk]; ...

for(k=0; k<dimz; k++) {
    // load k+1 plane into registers
    xc2 = x[ijk + kStride]; ...

    // apply operator
    const double Ax = apply_op_ijk();

    // smoother
    xo[ijk] = xc1 + ...;

    // update k and k-1 planes in registers
    xc0 = xc1; xc1 = xc2; ...
}

TOTAL REG USAGE: 56 FOR FV2 AND 128 FOR FV4

7-POINT STENCIL, 18 REGS

const double Ax =
- b*h2inv *
STENCIL_TWELFTH *(
+ bicy * (xrl - xcl) + bici * (xl1 - xcl) + bjcy * (xlu - xcl) + bjci * (zd1 - xcl) + bkc2 * (xc2 - xcl) + bkc1 * (xc0 - xcl));

4TH ORDER STENCIL, 90 REGS

const double Ax =
- b*h2inv *
STENCIL_TWELFTH *(
+ bicy * (xrl - xcl) + bici * (xl1 - xcl) + bjcy * (xlu - xcl) + bjci * (zd1 - xcl) + bkc2 * (xc2 - xcl) + bkc1 * (xc0 - xcl) + 0.25*STENCIL_TWELFTH *(
  + (bid - bju) * (xld - xd1 - xlu + xu1) + (bic2 - bic0) * (xl2 - xc2 - x10 + xc0) + (bjr - bj1) * (xru - xrl - xlu + xl1) + (bjc2 - bjc0) * (xu2 - xc2 - xu0 + xc0) + (bxr1 - bkl2) * (x0 - xrl - x10 + xl1) + (bkdl - bku1) * (xd0 - xdl - xu0 + xu1) + (bird - biru) * (xrd - xd1 - xru + xu1) + (bir2 - bir0) * (xr2 - xc2 - хр + xc0) + (bjrd - bj1d) * (xrd - xrl - xld + xl1) + (bjsd - bj1s) * (xrd - xrl - xld + xl1) + (bxr2 - bk12) * (x2 - xrl - xld + xl1) + (bkdr2 - bk12) * (x2 - xrl - xld + xl1));

up to 1.5x speed-up!
STENCILS ON GPU

Optimization: read-only loads

// macros
#ifdef USE_TEX
#define X(i) ( __ldg(&x[i]) )
#else
#define X(i) ( x[i] )
#endif

// operator
#define apply_op_ijk() ( H0 - b*h2inv* ( + BI(ijk+1)*( X(ijk+1) ) - X(ijk) ) 
+ BI(ijk) *( X(ijk-1) ) - X(ijk) ) 
+ BJ(ijk+jStride)*( X(ijk+jStride) - X(ijk) ) 
+ BJ(ijk) *( X(ijk-jStride) - X(ijk) ) 
+ BK(ijk+kStride)*( X(ijk+kStride) - X(ijk) ) 
+ BK(ijk) *( X(ijk-kStride) - X(ijk) ) )

POISSON AND HELMHOLTZ WITH 7-POINT STENCIL

MACROS FOR GRIDPOINTS AND COEFFICIENTS

LESS INTRUSIVE THAN SMEM, BUT BETTER PERF

up to 1.3x speed-up!
MULTI-GPU
CUDA-aware MPI

DATA MIGRATIONS (50%)

SINGLE NODE WITH P2P BETWEEN TWO K40s
MULTI-GPU SCALING

Single node with P2P

GPU: NVIDIA Tesla K40, ECC on
Results acquired by Sam Williams using the latest code: https://bitbucket.org/nsakharnykh/hpgmg-cuda
HPGMG RANKING LIST

**ORNL Titan (GPU) - 9.156E+11 DOF/s, 32M DOF per GPU**

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Site</th>
<th>DOF/s</th>
<th>Fraction of System</th>
<th>Parallelization</th>
<th>DOF per Process</th>
<th>Top500 Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mira</td>
<td>Argonne</td>
<td>7.21E+11</td>
<td>100%</td>
<td>49152 64</td>
<td>16M</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>K</td>
<td>RIKEN</td>
<td>7.12E+11</td>
<td>73%</td>
<td>64000 8</td>
<td>2M</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Edison</td>
<td>NERSC</td>
<td>3.85E+11</td>
<td>100%</td>
<td>131072 1</td>
<td>4M</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Titan (CPU-only)</td>
<td>Oak Ridge</td>
<td>2.53E+11</td>
<td>88%</td>
<td>32768 8</td>
<td>16M</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Stampede (CPU-only)</td>
<td>TACC</td>
<td>1.49E+11</td>
<td>64%</td>
<td>8192 8</td>
<td>2M</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>Hopper</td>
<td>NERSC</td>
<td>1.21E+11</td>
<td>86%</td>
<td>21952 6</td>
<td>2M</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>Piz Daint (CPU-only)</td>
<td>CSCS</td>
<td>1.02E+11</td>
<td>78%</td>
<td>4096 8</td>
<td>18M</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>SuperMUC</td>
<td>LRZ</td>
<td>7.13E+10</td>
<td>15%</td>
<td>2744 8</td>
<td>16M</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>BiFrost</td>
<td>NSC</td>
<td>4.67E+10</td>
<td>98%</td>
<td>1260 16</td>
<td>176M</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Stampede (MIC-only)</td>
<td>TACC</td>
<td>2.16E+10</td>
<td>8%</td>
<td>512 180</td>
<td>16M</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>Peregrine (IVB-only)</td>
<td>NREL</td>
<td>1.08E+10</td>
<td>18%</td>
<td>512 12</td>
<td>2M</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>Carver</td>
<td>NERSC</td>
<td>1.35E+09</td>
<td>5%</td>
<td>125 4</td>
<td>2M</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>Babbage (MIC-only)</td>
<td>NERSC</td>
<td>8.24E+08</td>
<td>30%</td>
<td>27 180</td>
<td>16M</td>
<td>-</td>
</tr>
</tbody>
</table>

[https://hpgmg.org/2015/02/06/updated-hpgmg-fv-ranking/](https://hpgmg.org/2015/02/06/updated-hpgmg-fv-ranking/)
WORK IN PROGRESS

2nd order code is publically available on bitbucket:

https://bitbucket.org/nsakharnykh/hpgmg-cuda

Initial 4th order implementation is ready and will be uploaded soon!

Preliminary results show about 2-3x slower performance compared to 2nd order

Future work:

Exploring finer-grained parallelization for intermediate/coarse levels
CONCLUSIONS

Both LOC and TOC architectures are utilized for best performance of HPGMG

Unified Memory greatly simplifies code porting to GPU architecture

Kernel optimizations are not intrusive and should apply to future architectures

The code scales well up to 16K GPUs on large supercomputing clusters