HIGH-ORDER DISCRETIZATIONS FOR GEOMETRIC MULTI-GRID METHODS

Nikolay Sakharnykh (NVIDIA), Dusan Stosic (UFPE), 4/4/2016
MULTI-GRID METHODS

Introduction

Multi-grid solves elliptic PDEs \((Ax=b)\) using a hierarchical approach

solution to hard problem is expressed as solution to an easier problem

accelerates iterative method and provides \(O(N)\) complexity
MULTI-GRID METHODS

Applications

EXTERNAL FLOW OVER BODY

RESERVOIR SIMULATION

LATTICE QCD (BROOKHAVEN NATIONAL LABORATORY)
MULTI-GRID METHODS
Geometric vs algebraic

Ax=b

GMG
Uses a structured grid
Operator (A) is a stencil
Constructs coarse grid directly
More efficient approach
Requires geometric information

AMG
Uses an arbitrary graph
Operator (A) is a sparse matrix
Constructs coarse grid using RAP product
More general approach
Performance optimization is challenging
NVIDIA AMGX
Black-box library for AMG

Fast, scalable linear solvers, emphasis on iterative methods

Flexible toolkit for GPU accelerated $Ax = b$ solver

Simple API makes it easy to solve your problems faster

Used in industry-leading CFD packages!
NVIDIA AMGX

Example usage

```c
// One header
#include "amgx_c.h"

// Read config file
AMGX_create_config(&cfg, cfgfile);
// Create resources based on config
AMGX_resources_create_simple(&res, cfg);
// Create solver object, A, x, b, set precision
AMGX_solver_create(&solver, res, mode, cfg);
AMGX_matrix_create(&A, res, mode);
AMGX_vector_create(&x, res, mode);
AMGX_vector_create(&b, res, mode);
// Read coefficients from a file
AMGX_read_system(&A, &x, &b, matrixfile);
// Setup and Solve Loop
AMGX_solver_setup(solver, A);
AMGX_solver_solve(solver, b, x);
// Download Result
AMGX_download_vector(&x)
```

Configuration file

```plaintext
solver(main)=FGMRES
main:max_iters=100
main:convergence=RELATIVE_MAX
main:tolerance=0.1
main:preconditioner(AMG)=AMG
    amg:algorithm=AGGREGATION
    amg:selector=SIZE_8
    amg:cycle=V
    amg:max_iters=1
    amg:max_levels=10
    amg:smoother(smooother)=BLOCK_JACOBI
    amg:relaxation_factor= 0.75
    amg:presweeps=1
    amg:postsweeps=2
    amg:coarsest_sweeps=4
determinism_flag=1
```
HPGMG
High-Performance Geometric Multi-Grid

Lawrence Berkeley National Laboratory

FVM and FEM variants, we focus on FVM
Proxy AMR and Low Mach Combustion codes
Used in Top500 benchmarking

http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg/
HPGMG implements F-cycle which has better convergence rate than V-cycle.

Poisson or Helmholtz operator using 2nd or 4th order discretization.
Order describes the relationship between grid spacing and error.

Why do we need higher-order discretizations?
1. Improving accuracy without significant increase in memory usage
2. Higher flops per byte ratio enables better utilization of modern architectures

Many solvers are moving towards high-order schemes for production codes.
HPGMG
v0.2: second-order

2nd order: 7-point stencil

6 x 2-point stencils
HPGMG
v0.3: fourth-order

4th order: 25-point stencil

18 x 4-point stencils, Flop:Byte ~1.0
HPGMG

4th order vs 2nd order

Performs 4x the FP operations

MPI: sends 3x the messages, doubles the size (2-deep halos)

DRAM memory footprint is the same (assuming no overfetch)

Attains lower relative residual: $\sim 10^{-9}$ for a single F-cycle
HYBRID IMPLEMENTATION
HYBRID IMPLEMENTATION
Take advantage of both architectures

Fine levels are executed on throughput-optimized processors (GPU)
Coarse levels are executed on latency-optimized processors (CPU)
HYBRID IMPLEMENTATION

What is the optimal threshold?

HPGMG v0.3 hybrid performance

execute on GPU if >10K grid points

All levels on GPU

All levels on CPU
MEMORY MANAGEMENT

HPGMG-FV entities naturally map to GPU hierarchy
Vector data within a level is disjoint
Requires one copy per box

Vector data within a level is contiguous
Requires one copy per vector
MEMORY MANAGEMENT
Using Unified Memory

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

Use `cudaMallocManaged` for allocations
UNIFIED MEMORY
Simplified GPU programming

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

```c
void smooth(level_type *level,...){
    ...
    if(level->use_cuda) {
        // run on GPU
        cuda_cheby_smooth(level,...);
    }
    else {
        // run on CPU
        #pragma omp parallel for
        for(block = 0; block < num_blocks; block++)
            ...
    }
}
```
UNIFIED MEMORY

What about performance?

**Problem:** excessive faults and migrations at CPU-GPU crossover points
UNIFIED MEMORY
Eliminating page migrations and faults

Level N (large) is shared between CPU and GPU

Level N+1 (small) is shared between CPU and GPU

Solution: allocate the first CPU level with cudaMallocHost (zero-copy memory)
UNIFIED MEMORY

CPU levels allocated with cudaMallocManaged

CPU levels allocated with cudaMallocHost

+20\% speed-up!

no page faults
KERNEL OPTIMIZATIONS
MULTI-GRID BOTTLENECK

Cost of operations

LEVEL

KERNEL TIME / TOTAL TIME

0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5 6
level

MOST TIME SPENT ON STENCILS

Smoother
Interpolation
Copy_blocks
Residual
Restriction
Apply_bc

0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5 6
level

VOLUME

SURFACE

Smoother
Interpolation
Copy_blocks
Residual
Restriction
Apply_bc
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
Baseline 2D kernel

```c
__global__ void smooth_kernel(level_type level,...) {
    blockCopy_type block = level.my_blocks[blockIdx.x];
    // prologue
    const double * __restrict__ rhs = ...;
    ...
    for(int 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
    }
}
```

APPLY STENCIL OPERATION
POISSON, HELMHOLTZ (FV2,FV4)

THREAD BLOCK MARCHES FROM 0 TO DIMZ

CHEBYSHEV POLYNOMIALS

JACOBI

GAUSS SEIDEL RED-BLACK
STENCILS ON GPU
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*(
    + bic1 * (zl1 - xc1)
    + bjcl * (xtu - xcl)
    + bjcl * (zlu - xcl)
    + bjc2 * (xc2 - xcl)
    + bkcl * (xc0 - xcl)
);

4TH ORDER STENCIL, 90 REGS

const double Ax =
- b*h2inv*STENCIL_TWELFTH*(
    + bir1 * (xr1 - xc1)
    + bkl1 * (xlr - xcl)
    + bju1 * (xru - xcl)
    + bjc1 * (xu1 - xcl)
    + bkcl * (xc0 - xcl)
    + bkc1 * (xc2 - xcl)
    + bjc2 * (xu0 - xcl)
    + bkc2 * (xc1 - xcl)
    + 0.25*STENCIL_TWELFTH*(
        + (bid - bju) * (xld - xdl - xlu + xu1)
        + (bic2 - bic0) * (xlu - xc2 - xlu + xc0)
        + (bbr - bjl) * (xru - xrl - xru + xll)
        + (bjc2 - bjc0) * (xu2 - xc2 - xu0 + xc0)
        + (bkl2 - bkli) * (xru - xrl - xlu + xll)
        + (bkd1 - bku1) * (xd0 - xdl - xlu + xu1)
        + (bbr2 - bkl2) * (xru - xrl - xru + xll)
        + (bjrd - bjl4) * (xrx - xrl - xld + xll)
        + (bkd2 - bkli) * (xd0 - xdl - xlu + xu1)
        + (bkr2 - bkli) * (xr0 - xrl - xlu + xll)
        + (bkrd - bkli) * (xd0 - xdl - xlu + xu1)
    ));

up to 1.5x speed-up!
STENCILS ON GPU
Using caches on SM

By default global loads are not cached in L1 on Kepler

Use -Xptxas="-dlcm=ca" to enable L1

128B load granularity
Better for coalesced access

Use __ldg intrinsic for read-only accesses

32B load granularity
Better for scattered access

up to 1.4x speed-up!

up to 1.3x speed-up!
STENCILS ON GPU

Shared memory approach

Copy to registers by staging in shared memory
Reduce redundant accesses along XY plane
Double buffering to avoid synchronization

Store only ‘\(x\)’ and ‘\(\beta k\)’ in shared memory

```c
int tx = threadIdx.x + R;
int ty = threadIdx.y + R;
sx[ty][tx], sx[ty][tx+1]
```
STENCILS ON GPU
Optimizations summary on Kepler

4th-order GSRB smoother performance

- baseline
- tex
- l1
- reg
- reg+smem
- reg+tex
- reg+l1

Kernel speed-up

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>1x</td>
</tr>
<tr>
<td>tex</td>
<td>1x</td>
</tr>
<tr>
<td>l1</td>
<td>1.5x</td>
</tr>
<tr>
<td>reg</td>
<td>1.5x</td>
</tr>
<tr>
<td>reg+smem</td>
<td>2x</td>
</tr>
<tr>
<td>reg+tex</td>
<td>2x</td>
</tr>
<tr>
<td>reg+l1</td>
<td>2x</td>
</tr>
</tbody>
</table>
# STENCILS ON GPU

**Bottleneck analysis on Tesla K40**

<table>
<thead>
<tr>
<th></th>
<th>BASELINE</th>
<th>OPTIMIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>METRIC</strong></td>
<td><strong>FV2</strong></td>
<td><strong>FV4</strong></td>
</tr>
<tr>
<td>OCCUPANCY</td>
<td>0.55</td>
<td>0.25</td>
</tr>
<tr>
<td>FOOTPRINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>1.10 GB</td>
<td>1.09 GB</td>
</tr>
<tr>
<td>L2</td>
<td>2.22 GB</td>
<td>8.22 GB</td>
</tr>
<tr>
<td>BANDWIDTH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>0.76</td>
<td>0.20</td>
</tr>
<tr>
<td>L2</td>
<td>0.69</td>
<td>0.67</td>
</tr>
<tr>
<td>TIME</td>
<td>5.63 ms</td>
<td>20.65 ms</td>
</tr>
</tbody>
</table>

FV2 LIMITED BY DRAM, FV4 LIMITED BY L2

DECREASE IN L2 FOOTPRINT

INCREASE IN DRAM BW
MAXWELL?
Poor double precision performance 😞

2nd order scheme works really well, limited by DRAM bandwidth

Improves with register/caching optimizations

4th order scheme starts to bottleneck in DP unit on Maxwell

Performance stays flat across all optimizations

We have more flops to execute, but not enough DP throughput
Experiment with clang went smoothly overall

Compilation time is 20% shorter for clang 3.9 compared to nvcc 7.5

Minor issues with CUB due to PTX intrinsics

GSRB smoother kernel runtime:

<table>
<thead>
<tr>
<th></th>
<th>Baseline (ms)</th>
<th>Optimized (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvcc 7.5</td>
<td>25.5</td>
<td>17.0</td>
</tr>
<tr>
<td>clang 3.9</td>
<td>27.2</td>
<td>19.4</td>
</tr>
<tr>
<td>clang vs nvcc</td>
<td>+7%</td>
<td>+14%</td>
</tr>
</tbody>
</table>

http://llvm.org/docs/CompileCudaWithLLVM.html
MULTI-GPU
MPI communications

Communication scheme:

1 - send data (boundary->MPI buffer)
2 - local exchange (internal->internal)
3 - receive data (MPI buffer->boundary)

Intra-level: boundary ghost region exchange

Inter-level: restriction and interpolation
MULTI-GPU

MPI vs SHMEM

Boundary exchange code

<table>
<thead>
<tr>
<th>MPI</th>
<th>SHMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyKernel(BOUNDARY-TO-BUFFER) cudaDeviceSync MPI_Irecv + MPI_Isend CopyKernel(INTERNAL-TO-INTERNAL) MPI_Waitall CopyKernel(BUFFER-TO-BOUNDARY)</td>
<td>CopyKernel(ALL-TO-ALL) shmem_barrier_all</td>
</tr>
</tbody>
</table>

S6378 - Simplifying Multi-GPU Communication with NVSHMEM

Wednesday @ 16:30, Room 211B
MULTI-GPU
Unified Memory and MPI

Remember that we replaced malloc with cudaMallocManaged

Unified Memory and regular MPI implementation

Requires unmanaged staging buffer

Does not play well with RDMA protocols

cudaMallocManaged + regular MPI
Pinned allocations and CUDA-aware MPI

Unified Memory and CUDA-aware MPI

Supported in OpenMPI >= 1.8.5 and MVAPICH2-GDR >= 2.2b

Avoids CUDA IPC which is not supported for Unified Memory
 Use zero-copy to allocate MPI buffers, write them directly from GPU kernels

Can outperform RDMA for large sizes at scale
MULTI-GPU RESULTS

Weak scaling within a single node

CPU: Intel E5-2698 v3@2.3GHz 3.6GHz Turbo (Haswell-EP) HT off
GPU: NVIDIA Tesla K80
MULTI-GPU RESULTS
Weak scaling across thousands nodes

ORNL Titan performance of HPGMG v0.3

*results are obtained by Sam Williams (Lawrence Berkeley National Laboratory)
TAKEAWAYS

Both CPU and GPU architectures are utilized for best performance of HPGMG

Unified Memory greatly simplifies code porting to GPU architecture

Optimizations are not intrusive and should apply to future architectures

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

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

Parallel Forall blog post: http://nvda.ly/YFAcl

S6216 - The Future of Unified Memory
   Tuesday @ 16:00, Room 212A

CUDA Libraries Hangout. AmgX, cuBLAS, cuSolver, cuSparse
   Thursday @ 10:00, Pod B
THANK YOU

JOIN THE CONVERSATION
#GTC16  

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join