Optimizing Geometric Multigrid Method Computation using a DSL Approach

Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat, Uday Bondhugula

Department of Computer Science and Automation
Indian Institute of Science
Bengaluru 560012, India

14th Nov 2017
Iterative Numerical Methods

Visualization of heat transfer in a pump casing; created by solving the heat equation.

- Numerical approximation for partial differential equations
- Approximation is improved every iteration
- Applications in fields of engineering and physical sciences
Poisson’s equation

\[ \nabla^2 u = f. \]
Poisson’s equation

\[ \nabla^2 u = f. \]

Finite difference discretization; approximate second derivative

What about \( A^{-1} \)?
Multigrid Methods

- **Poisson’s equation**

\[ \nabla^2 u = f.\]

- Finite difference discretization; approximate second derivative

- Solve \( f = Au \), where \( A \) is a sparse banded matrix (\( u \) is a linearization of the unknown on the multi-dimensional grid)

- What about \( A^{-1} \)?
Multigrid Methods

- Involving discretizations at multiple scales
- A fast-forward technique in the iterative solving process
- Used as direct solvers as well as preconditioners
Multigrid Methods

- Unit of execution: multigrid cycle
- Iterate until convergence / for fixed number

\[ n_{\text{iters}} < N \| V_{\text{error}} < \rho \]
Multigrid Cycles: V-Cycle

- Smoother
- Defect/Residual
- Correction
- Restrict/Reciprocate
- Interpolate/Prolongation
Multigrid Methods

- Heterogeneous composition of basic computations
- Typically expressed as a recursive algorithm
- Optimizations limited to one multigrid cycle

### Algorithm 1: V-cycle$^h$

```plaintext
Input : $v^h, f^h$
1 Relax $v^h$ for $n_1$ iterations  // pre-smoothing
2 if coarsest level then
3     Relax $v^h$ for $n_2$ iterations  // coarse smoothing
4     $r^h \leftarrow f^h - A^h v^h$  // residual
5     $r^{2h} \leftarrow I^h_2 r^h$  // restriction
6     $e^{2h} \leftarrow 0$
7     $e^{2h} \leftarrow V$-cycle$^{2h}(e^{2h}, r^{2h})$
8     $e^h \leftarrow I^h_2 e^{2h}$  // interpolation
9     $v^h \leftarrow v^h + e^h$  // correction
10 Relax $v^h$ for $n_3$ iterations  // post smoothing
11 return $v^h$
```
Multigrid Methods: Basic Steps

Smoothing

\[ f(x, y) = \sum_{\sigma_x = -1}^{+1} \sum_{\sigma_y = -1}^{+1} g(x + \sigma_x, y + \sigma_y) \cdot w(\sigma_x, \sigma_y) \]
Multigrid Methods: Basic Steps

Restriction/Decimation

\[ f(x, y) = \sum_{\sigma_x = -1}^{+1} \sum_{\sigma_y = -1}^{+1} g(2x + \sigma_x, 2y + \sigma_y) \cdot w(\sigma_x, \sigma_y) \]
Multigrid Methods: Basic Steps

Interpolation/Prolongation

\[ f(x, y) = \sum_{\sigma_x = 0}^{+1} \sum_{\sigma_y = 0}^{+1} g((x + \sigma_x)/2, (y + \sigma_y)/2) \cdot w(\sigma_x, \sigma_y, x, y) \]
Multigrid Methods: Basic Steps

Point-wise

\[ f(x, y) = \alpha \cdot g(x, y) \]
Multigrid Cycles: W-Cycle

- Smoother
- Defect/Residual
- Correction

- Restrict/Reciprocate
- Interpolate/Prolongation
Naive vs Optimized Implementation

- Naive parallelization (OpenMP, vector pragmas (icpc)): 5.4x
- Optimized code generated by a DSL compiler (PolyMage): tiling + fusion + scratchpads, etc. (2.27× over naive)
- Enhancements in PolyMage: + storage opt + pooled memory + multidim parallelism, etc. (5.92× over naive)

**Multigrid-2D (Jacobi) (24 cores)**

<table>
<thead>
<tr>
<th></th>
<th>par</th>
<th>opt</th>
<th>opt+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>2.39</td>
<td>1.05</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Naive vs Optimized Implementation

Multigrid-2D (Jacobi) (24 cores)

- **Problem:** Speeding up complex pipelines like multigrid cycles manually is hard and tedious
- **Goal:** Figure out complex and right set of optimizations automatically

- Naive parallelization (OpenMP, vector pragmas (icpc): 5.4x
- Optimized code generated by a DSL compiler (PolyMage): tiling + fusion + scratchpads, etc. (2.27× over naive)
- Enhancements in PolyMage: + storage opt + pooled memory + multidim parallelism, etc. (5.92× over naive)
Related Work

Semi-automatic ones

- S. Williams, D. Kalamkar et al. [SC 2012]
  Communication aggregation: for dist-mem
  Wavefront schedule at node
  Fusion of Residual and Restrict
  Manual optimization and implementations

- Ghysels and Vanroose [SIAM J. Scientific Computing 2015]
  Pluto tiling for smoothing steps
  Manual optimization and implementation
  Code modifications for array access

- P. Basu, M. Hall et al. [HiPC’13, IPDPS’15]
  Wavefront schedule, fusion
  Semi-automatic (script-driven)
  No tiling transformations
Using PolyMage DSL to Optimize MG Cycles

- **PolyMage** [ASPLOS 2015]: DSL/Compiler for image processing pipelines
  - **DSL embedded in Python**
  - **Automatic Optimizing Code Generator**
    - Uses domain-specific cost models to apply complex combinations of **tiling** and **fusion** to optimize for **parallelism** and **locality**
PolyMage for Multigrid

- Express grid computations as functions
- Abstracts away computations from schedule and storage mapping
- Added new constructs for time-iterated stencils (smoothing)
- Use the **polyhedral framework** to transform, optimize, and parallelize
  
  Group functions, tile each group, generate storage mappings
V-cycle Algorithm in PolyMage

\[ N = \text{Parameter(Int, 'n')} \]
\[ V = \text{Grid(Double, "V", [N+2, N+2])} \]
\[ F = \text{Grid(Double, "F", [N+2, N+2])} \]

... def rec_v_cycle(v, f, l):
    # coarsest level
    if l == 0:
        smooth_p1[l] = smoother(v, f, l, n3)
        return smooth_p1[l][n3]

    # finer levels
    else:
        smooth_p1[l] = smoother(v, f, l, n1)
        r_h[l] = defect(smooth_p1[l][n1], f, 1)
        r_2h[l] = restrict(r_h[l], 1)
        e_2h[l] = rec_v_cycle(None, r_2h[l], l-1)
        e_h[l] = interpolate(e_2h[l], l)
        v_c[l] = correct(smooth_p1[l][n1], e_h[l], l)
        smooth_p2[l] = smoother(v_c[l], f, l, n2)
        return smooth_p2[l][n2]

def restrict(v, l):
    R = Restrict(([y, x], [extent[l], extent[l]]),
                  Double)
    R.defn = [ Stencil(v, (y, x),
                        [[1, 2, 1],
                         [2, 4, 2],
                         [1, 2, 1]], 1.0/16, factor=1//2) ]
    return R

def smoother(v, f, l, n):
    ... W = TStencil(([y, x], [extent[l], extent[l]]),
                      Double, T)
    W.defn = [ v(y, x) - weight *
               (Stencil(v, [y, x],
                            [[ 0, -1, 0],
                             [-1, 4, -1],
                             [ 0, -1, 0]], 1.0/h[l]**2) - f(y, x) ]

    return W

def interpolate(v, l):
    ... expr = [{}, {}]
    expr[0][0] = Stencil(v, (y, x), [1])
    expr[0][1] = Stencil(v, (y, x), [1, 1])
    * 0.5
    expr[1][0] = Stencil(v, (y, x), [1, 1])
    * 0.5
    expr[1][1] = Stencil(v, (y, x), [1, 1])
    * 0.25
    P = Interp(([y, x], [extent[l], extent[l]]),
                  Double)
    P.defn = [ expr ]
    return P
x = Variable()
fin = Grid(Double, [18])
f1 = Function([[x], [Interval(0, 17)]), Double)
f1.defn = [ fin(x) + 1 ]
f2 = Function([[x], [Interval(1, 16)]), Double)
f2.defn = [ f1(x-1) + f1(x+1) ]
fout = Function([[x], [Interval(2, 15)]), Double)
fout.defn = [ f2(x-1) . f2(x+1) ]
Polyhedral Representation

\[ x = \text{Variable}() \]
\[ f_{in} = \text{Grid}(\text{Double}, [18]) \]
\[ f_1 = \text{Function}(([x], [\text{Interval}(0, 17)]), \text{Double}) \]
\[ f_1.\text{defn} = [f_{in}(x) + 1] \]
\[ f_2 = \text{Function}(([x], [\text{Interval}(1, 16)]), \text{Double}) \]
\[ f_2.\text{defn} = [f_1(x-1) + f_1(x+1)] \]
\[ f_{out} = \text{Function}(([x], [\text{Interval}(2, 15)]), \text{Double}) \]
\[ f_{out}.\text{defn} = [f_2(x-1) \cdot f_2(x+1)] \]
### Polyhedral Representation

**Dependence vectors**

![Dependence vectors diagram](image)

**Function**

<table>
<thead>
<tr>
<th>Function</th>
<th>Dependence Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{out}(x) = f_2(x - 1) \cdot f_2(x + 1)$</td>
<td>$(1, 1), (1, -1)$</td>
</tr>
<tr>
<td>$f_2(x) = f_1(x - 1) + f_1(x + 1)$</td>
<td>$(1, 1), (1, -1)$</td>
</tr>
<tr>
<td>$f_1(x) = f_{in}(x)$</td>
<td></td>
</tr>
</tbody>
</table>
Polyhedral Representation

\[ f_{\text{out}}(x) = f_2(x - 1) \cdot f_2(x + 1) \quad (1, 1), (1, -1) \]
\[ f_2(x) = f_1(x - 1) + f_1(x + 1) \quad (1, 1), (1, -1) \]
\[ f_1(x) = f_{\text{in}}(x) \]
Tiling Smoothing Steps: Overlapped Tiling

- Good for parallelism, locality, local buffers
- Redundant computation at boundaries
Tiling Smoothing Steps: Diamond Tiling

- Good for parallelism, locality, but not local buffers
- No redundant computation
Overlapped Tiling for Multiscale
Grouping Stages for Overlapped Tiling

**Grouping criteria**
- Exponential number of valid groupings
- Redundant computation vs reuse (locality)
  *Overlap, tile sizes, parameter estimates*

**Grouping heuristic**
- Greedy iterative algorithm
- Only fuse stages that can be overlap tiled up to a maximum group size limit
Storage Optimizations

- **Storage for Functions**
  - Reuse of buffers across groups and within groups
  - Precise liveness information is available
Intra-Group Storage Reuse

- **Not just about byte count**
  - Accommodate larger tile sizes
  - Maximum performance gains for a configuration

- **Approach**
  - Simple greedy algorithm to colour the nodes
  - Requires schedule information
Inter-Group Storage Reuse

- **Larger gains in memory used**
  - Significant reduction in memory consumption
  - Less page activity
  - Higher chances of fitting in LLC

- **Approach**
  - Classification based on parametric bounds
  - Alloc/free at group granularity

- Pooled allocation and early freeing: across multiple calls and within pipeline call
Storage Allocation for Grouping
PolyMage Compiler Flow

- DSL Spec
- Build stage graph
- Static bounds check
- Inlining
- Polyhedral representation
- Initial schedule
- Schedule transformation
- Storage optimization
- Alignment, Scaling, Grouping
- Code generation
PolyMage Compiler Flow

- DSL spec
- Build stage graph
- Static bounds check
- Inlining
- Auto-merge
- Alignment
- Scaling
- Polyhedral representation
- Initial schedule
- Pooled memory allocation
- Early freeing
- Multipar
- LibPluto
- Code generation
- Inter-group reuse opt
- Intra-group reuse opt
- Schedule transformation (fusion, tiling)

Grouping

Storage optimization
Availability

- Available as part of PolyMage repository
  https://bitbucket.org/udayb/polymage.git

Multigrid benchmarks
sandbox/apps/python/multigrid/
Benchmarks

- Multigrid applications solving Poisson’s equation
  \[ \nabla^2 u = f. \]

- Configurations
  - V-cycle and W-cycle
  - 10-0-0 and 4-4-4
  - Four levels of discretization

- NAS-MG from NAS-PB 3.2
### Benchmarks

**Table:** Problem size configurations: the same problem sizes were used for V-cycle and W-cycle and for 4-4-4 and 10-0-0

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Grid size, cycle #iters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class B</td>
</tr>
<tr>
<td>2D</td>
<td>8192² (10)</td>
</tr>
<tr>
<td>3D</td>
<td>256³ (25)</td>
</tr>
<tr>
<td>NAS-MG</td>
<td>256³ (20)</td>
</tr>
</tbody>
</table>
## Experimental Setup

**Table:** Architecture details

<table>
<thead>
<tr>
<th></th>
<th>2-socket Intel Xeon E5-2690 v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>2.60 GHz</td>
</tr>
<tr>
<td>Cores / socket</td>
<td>12</td>
</tr>
<tr>
<td>Total cores</td>
<td>24</td>
</tr>
<tr>
<td>Hyperthreading</td>
<td>disabled</td>
</tr>
<tr>
<td>L1 cache / core</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 cache / core</td>
<td>512 KB</td>
</tr>
<tr>
<td>L3 cache / socket</td>
<td>30,720 KB</td>
</tr>
<tr>
<td>Memory</td>
<td>96 GB DDR4 ECC 2133 MHz</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel C/C++ and Fortran compiler (icc/icpc and ifort) 16.0.0</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3 -xhost -openmp -ipo</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>3.10.0 (64-bit) (Cent OS 7.1)</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Stages</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>(= # of DAG nodes)</td>
</tr>
<tr>
<td>V-2D-4-4-4</td>
<td>40</td>
</tr>
<tr>
<td>V-2D-10-0-0</td>
<td>42</td>
</tr>
<tr>
<td>W-2D-4-4-4</td>
<td>100</td>
</tr>
<tr>
<td>W-2D-10-0-0</td>
<td>98</td>
</tr>
<tr>
<td>V-3D-4-4-4</td>
<td>40</td>
</tr>
<tr>
<td>V-3D-10-0-0</td>
<td>42</td>
</tr>
<tr>
<td>W-3D-4-4-4</td>
<td>100</td>
</tr>
<tr>
<td>W-3D-10-0-0</td>
<td>98</td>
</tr>
<tr>
<td>NAS-MG</td>
<td>34</td>
</tr>
</tbody>
</table>
Comparison with Hand Opt and Pluto

- Ghysels and Vanroose [SIAM J. Sci. Comp. 2015]
  - Manual storage reuse optimization
  - Multidimensional parallelism
  - Pluto
    - Diamond tiling for concurrent startup
    - Tuned over 25 tile size configurations
2D Benchmarks: V-cycle

(a) 2D-V-10-0-0

(b) 2D-V-4-4-4
2D Benchmarks: W-cycle

(c) 2D-W-10-0-0

(d) 2D-W-4-4-4
3D Benchmarks: V-cycle

(e) 3D-V-10-0-0

(f) 3D-V-4-4-4
3D Benchmarks: W-cycle

(g) 3D-W-10-0-0

(h) 3D-W-4-4-4
Problem Class
- Speedup over PolyMG naive (24 cores)

- polymg-opt
- polymg-opt+
- reference

- B
- C

- Speedup: 1.12x, 1.24x, 1.37x, 1.25x, 1.53x, 1.02x
Storage Optimization Gains

V-cycle for 10-0-0
Summary of Improvements

- **1.31×** Mean speedup of Opt+ over Opt
  - 1.30× : 2D
  - 1.33× : 3D
- **3.21×** Mean speedup of Opt+ over Naive
  - 4.74× : 2D
  - 2.18× : 3D
- **1.23×** Mean speedup of Opt+ over HandOpt+Pluto
- **NAS-MG 17%** better over Opt+
Conclusions

- Significant performance improvement (automatic) through DSL approach
- Overlapped tiling - very effective for 2D GMG
- Diamond tiling better than Overlapped tiling for most of the 3D benchmarks
- Storage allocation and optimization - important for performance

Future Directions
- Distributed memory
Thank You

Acknowledgments
Tiling Smoothing Steps

3D stencil for class C
void pipeline_Vcycle(int N, double * F,
    double * V, double *& W)
{
    /* Live out allocation */
    /* users : ['T9_pre_L3'] */
    double * _arr_10_2;
    _arr_10_2 = (double *) (pool_allocate(sizeof(double) *
        (2+N)*(2+N)));

#pragma omp parallel for schedule (static) collapse (2)
for (int T_i = -1; T_i <= N/32; T_i+=1) {
    for (int T_j = -1; T_j <= N/512; T_j+=1) {
        /* Scratchpads */
        /* users : ['T8_pre_L3', 'T6_pre_L3', 'T4_pre_L3',
            'T2_pre_L3', 'T0_pre_L3'] */
        double _buf_2_0[(50 * 530)];
        /* users : ['T7_pre_L3', 'T5_pre_L3', 'T3_pre_L3',
            'T1_pre_L3'] */
        double _buf_2_1[(50 * 530)];

        int ub_i = min(N, 32*T_i + 49);
        int lb_i = max(1, 32*T_i);
        for (int i = lb_i; i <= ub_i; i++1) {
            int ub_j = min(N, 512*T_j + 529);
            int lb_j = max(1, 512*T_j);

#pragma ivdep
            for (int j = lb_j; (j <= ub_j); j+=1) {
                _buf_2_0[(-32*T_i+i)*530 + -512*T_j+j] = ...;
            }
        }

        int ub_i = min(N, 32*T_i + 48);
        int lb_i = max(1, 32*T_i);
        for (int i = lb_i; i <= ub_i; i++1) {
            int ub_j = min(N, 512*T_j + 528);
            int lb_j = max(1, 512*T_j);

#pragma ivdep
            for (int j = lb_j; (j <= ub_j); j+=1) {
                _buf_2_1[(-32*T_i+i)*530 + -512*T_j+j] = ...;
            }
        }
    }
}

...
NAS-MG V-Cycle

- Smoother
- Defect/Residual
- Correction
- Restrict/Reciprocate
- Interpolate/Prolongation