Optimizing Geometric Multigrid Method Computation using a DSL Approach

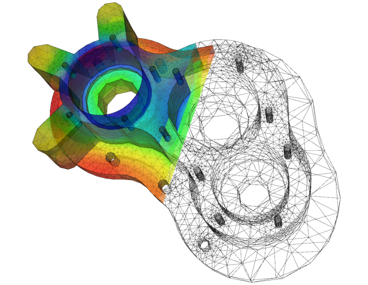
Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat,
Uday Bondhugula

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

14th Nov 2017

Outline

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

Multigrid Methods

- **Poisson's equation**

$$\nabla^2 u = f.$$

Multigrid Methods

- **Poisson's equation**

$$\nabla^2 u = f.$$

- Finite difference discretization; approximate second derivative

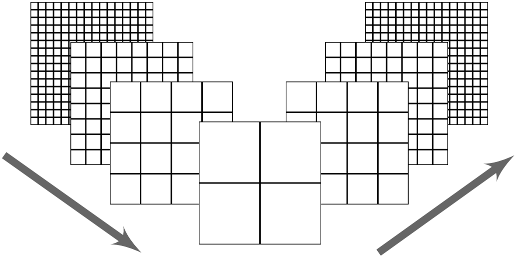
Multigrid Methods

- **Poisson's equation**

$$\nabla^2 u = f.$$

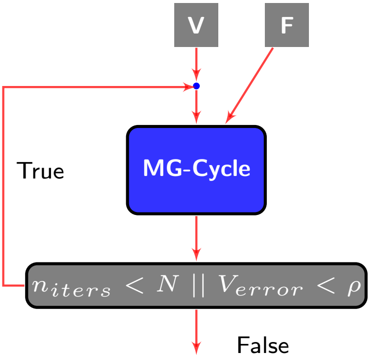
- Finite difference discretization; approximate second derivative
- Solve $f = Au$, where A is a sparse banded matrix (u is a linearization of the unknown on the multi-dimensional grid)
- What about A^{-1}?

Multigrid Methods



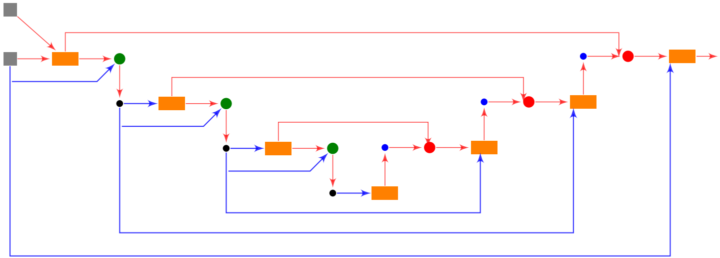
- Involving discretizations at multiple scales
- A fast-forward technique in the iterative solving process
- Used as direct solvers as well as preconditioners

Multigrid Methods



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

Multigrid Cycles: V-Cycle



Smoother

Defect/Residual

Correction

Restrict/Reciprocate

Interpolate/Prolongation

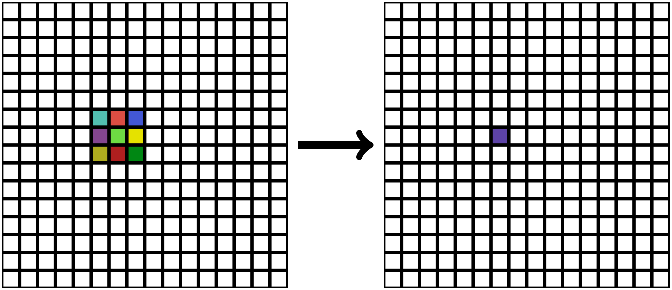
Multigrid Methods

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

Algorithm 1: V-cycle^h

```
Input : v^h, f^h
Relax v^h for n_1 iterations
// pre-smoothing
if coarsest level then
    Relax v^h for n_2 iterations    // coarse smoothing
r^h ← f^h − A^h v^h                 // residual
r^{2h} ← I_h^{2h} r^h               // restriction
e^{2h} ← 0
e^{2h} ← V-cycle^{2h}(e^{2h}, r^{2h})
e^h ← I_{2h}^h e^{2h}               // interpolation
v^h ← v^h + e^h                     // correction
Relax v^h for n_3 iterations        // post smoothing
return v^h
```

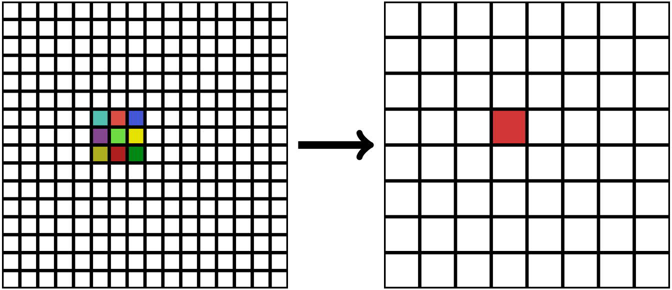
Multigrid Methods: Basic Steps



Smoothing

$$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(x+\sigma_x, y+\sigma_y) \cdot w(\sigma_x, \sigma_y)$$

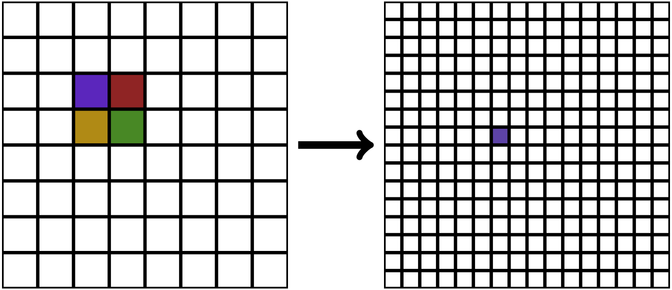
Multigrid Methods: Basic Steps



Restriction/Decimation

$$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(2x+\sigma_x, 2y+\sigma_y) \cdot w(\sigma_x, \sigma_y)$$

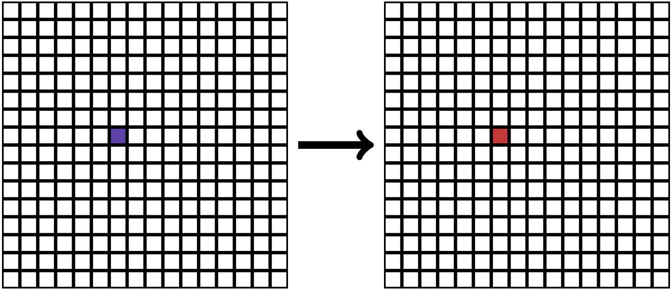
Multigrid Methods: Basic Steps



Interpolation/Prolongation

$$f(x,y) = \sum_{\sigma_x=0}^{+1} \sum_{\sigma_y=0}^{+1} g((x+\sigma_x)/2,(y+\sigma_y)/2) \cdot w(\sigma_x,\sigma_y,x,y)$$

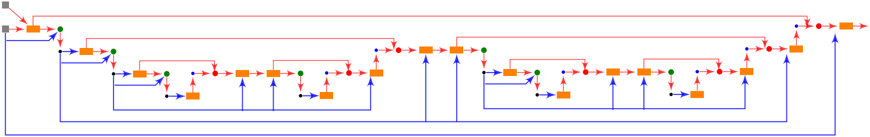
Multigrid Methods: Basic Steps



Point-wise

$$f(x, y) = \alpha \cdot g(x, y)$$

Multigrid Cycles: W-Cycle



Smoother

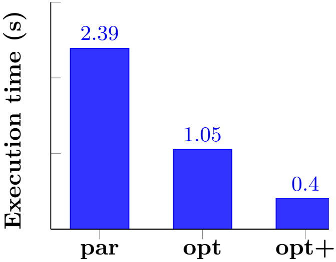
Defect/Residual

Correction

Restrict/Reciprocate

Interpolate/Prolongation

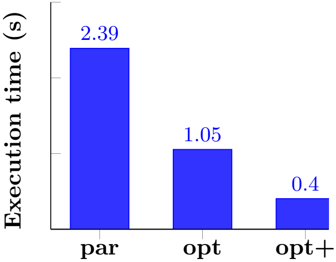
Naive vs Optimized Implementation



Multigrid-2D (Jacobi)
(24 cores)

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

Naive vs Optimized Implementation



Multigrid-2D (Jacobi) (24 cores)

- 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)

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

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

Outline

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

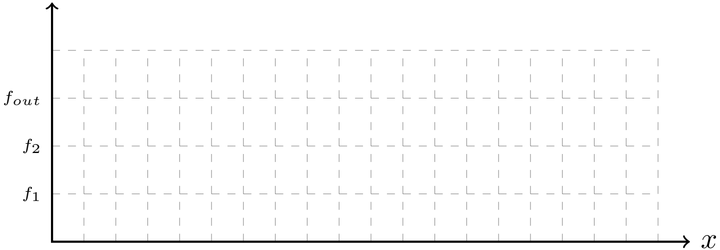
  Group functions, tile each group, generate storage mappings

V-cycle Algorithm in PolyMage

```
N = Parameter(Int, 'n')
V = Grid(Double, "V", [N+2, N+2])
F = 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, l)
        r_2h[l] = restrict(r_h[l], l)
        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
```

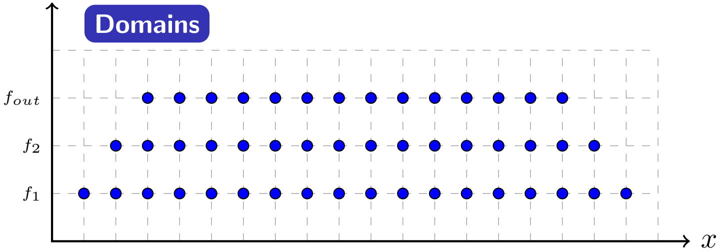
```
def smoother(v, f, l, n):
  ...
  W = TStencil(([y, x], [extent[l], extent[l]]),
               Double, T)
  W.defn = [ v(y, x) - weight *
    (Stencil(v, [y, x],
      [[ 0, -1,  0],
       [-1,  4, -1],
       [ 0, -1,  0]], 1.0/h[l]**2) - f(y, x)) ]
  ...
  return W

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], [1, 1]])\
                 * 0.25
    P = Interp(([y, x], [extent[l], extent[l]]),
               Double)
    P.defn = [ expr ]
    return P
```

Polyhedral Representation



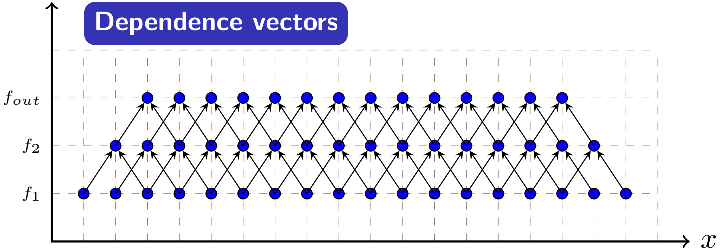

```
x = Variable()
f_in = Grid(Double, [18])
f_1 = Function(([x], [Interval(0, 17)]), Double)
f_1.defn = [ f_in(x) + 1 ]
f_2 = Function(([x], [Interval(1, 16)]), Double)
f_2.defn = [ f_1(x-1) + f_1(x+1) ]
f_out = Function(([x], [Interval(2, 15)]), Double)
f_out.defn = [ f_2(x-1) . f_2(x+1) ]
```

Polyhedral Representation



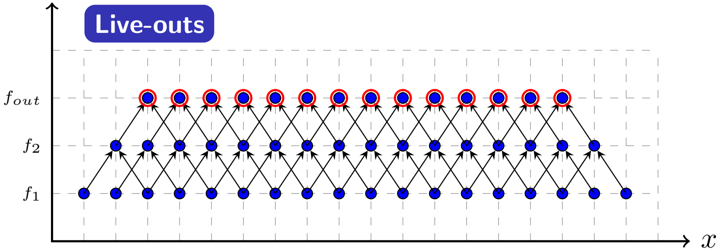

```
x = Variable()
f_in = Grid(Double, [18])
f_1 = Function(([x], [Interval(0, 17)]), Double)
f_1.defn = [ f_in(x) + 1 ]
f_2 = Function(([x], [Interval(1, 16)]), Double)
f_2.defn = [ f_1(x-1) + f_1(x+1) ]
f_out = Function(([x], [Interval(2, 15)]), Double)
f_out.defn = [ f_2(x-1) . f_2(x+1) ]
```

Polyhedral Representation



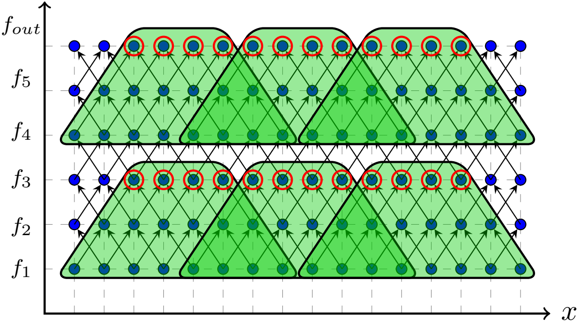
Function	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(1,1),(1,-1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(1,1),(1,-1)$
$f_1(x) = f_{in}(x)$	

Polyhedral Representation



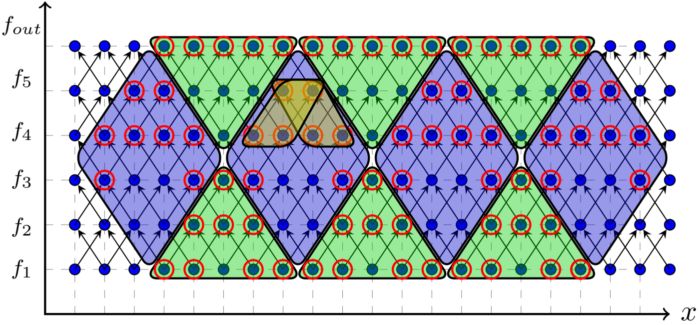
Function	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(1,1),(1,-1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(1,1),(1,-1)$
$f_1(x) = f_{in}(x)$	

Tiling Smoothing Steps: Overlapped Tiling



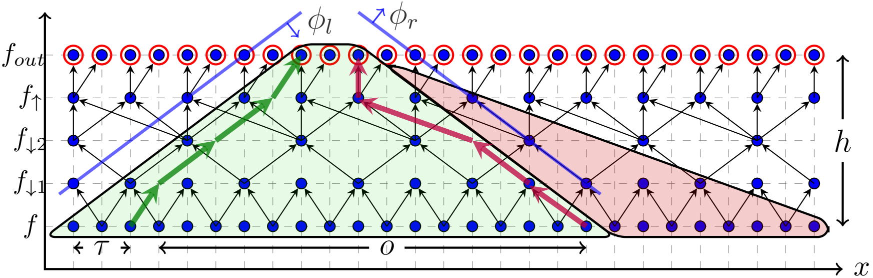
- Good for parallelism, locality, local buffers
- Redundant computation at boundaries

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

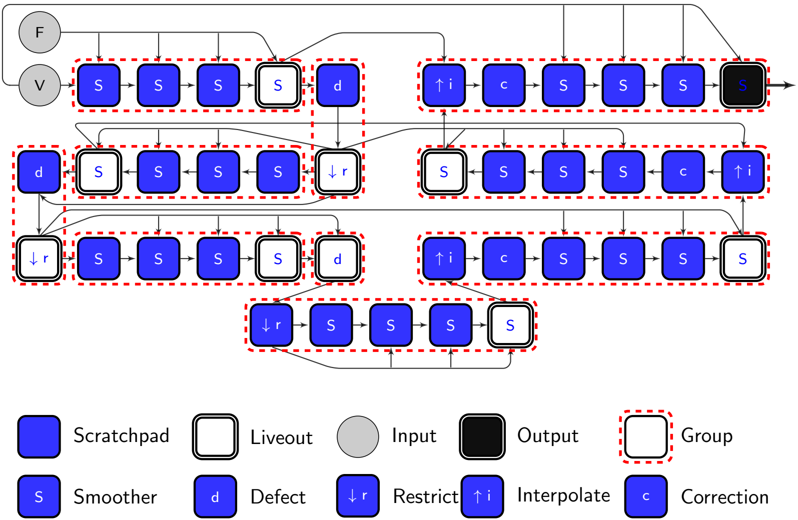
Grouping criteria

- 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

Grouping



F
V
S
d
↓ r
↑ i
c
Scratchpad
Liveout
Input
Output
Group
Smoother
Defect
Restrict
Interpolate
Correction

Storage Optimizations

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

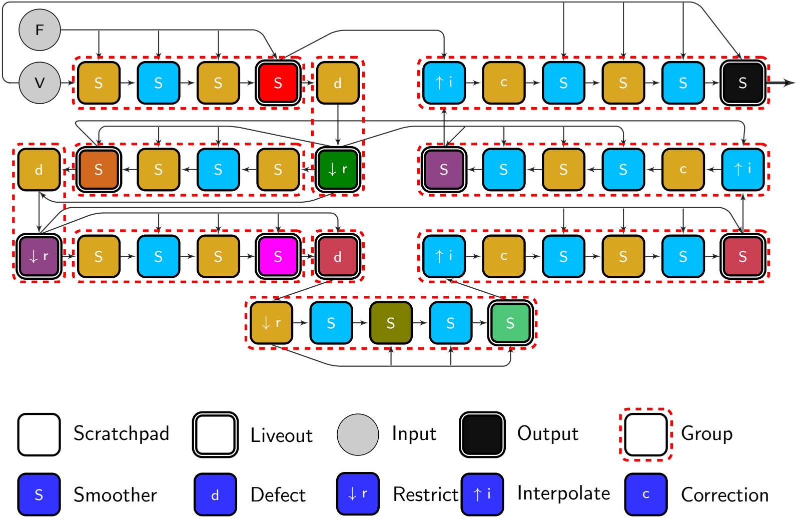
Intra-Group Storage Reuse

- **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

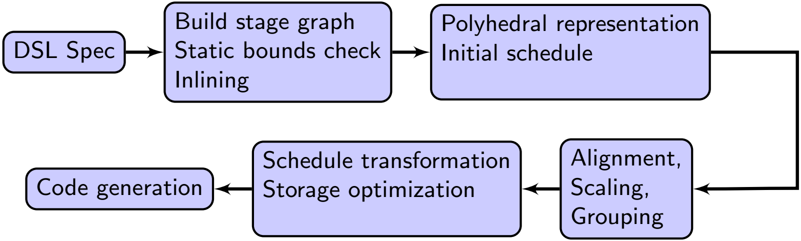
Inter-Group Storage Reuse

- **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

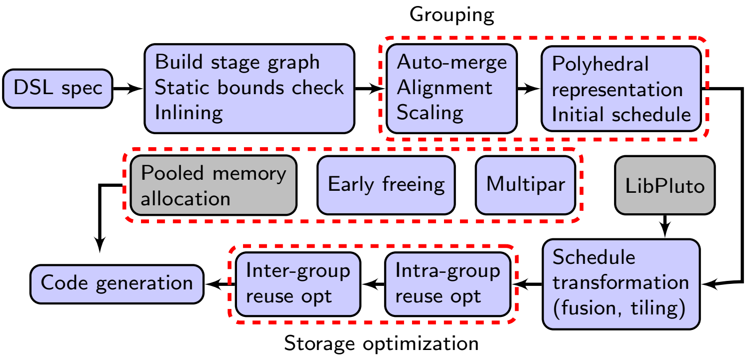
Storage Allocation for Grouping



PolyMage Compiler Flow



PolyMage Compiler Flow



Availability

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

  Multigrid benchmarks
  sandbox/apps/python/multigrid/

Outline

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

Benchmark	Grid size, cycle #iters	
	Class B	Class C
2D	8192^2 (10)	16384^2 (10)
3D	256^3 (25)	512^3 (10)
NAS-MG	256^3 (20)	512^3 (20)

Experimental Setup

Table: Architecture details

	2-socket Intel Xeon E5-2690 v3
Clock	2.60 GHz
Cores / socket	12
Total cores	24
Hyperthreading	disabled
L1 cache / core	64 KB
L2 cache / core	512 KB
L3 cache / socket	30,720 KB
Memory	96 GB DDR4 ECC 2133 MHz
Compiler	Intel C/C++ and Fortran compiler (icc/icpc and ifort) 16.0.0
Compiler flags	-O3 -xhost -openmp -ipo
Linux kernel	3.10.0 (64-bit) (Cent OS 7.1)

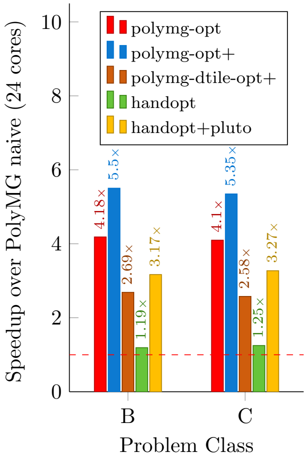
Lines of Code

Benchmark	Stages (= # of DAG nodes)	Lines of code: PolyMage DSL (Python)	Lines of code: HandOpt (C)	Lines of code: HandOpt + Pluto (C)	Lines of generated code: *polymage-opt* (C/C++)	Lines of generated code: *polymage-opt+* (C/C++)
V-2D-4-4-4	40	160	140	150	2324	2496
V-2D-10-0-0	42				2155	2059
W-2D-4-4-4	100	165	145	155	6156	6768
W-2D-10-0-0	98				4306	4711
V-3D-4-4-4	40	220	185	200	4889	4457
V-3D-10-0-0	42				4593	4179
W-3D-4-4-4	100	225	190	205	12184	11535
W-3D-10-0-0	98				9237	7897
NAS-MG	34	180	500	-	2010	2013

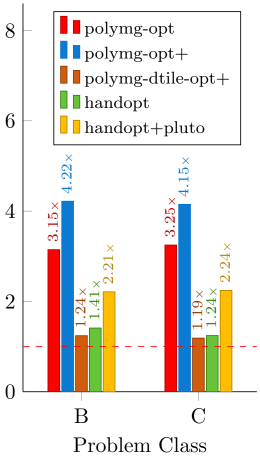
Comparison with Hand Opt and Pluto

- **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

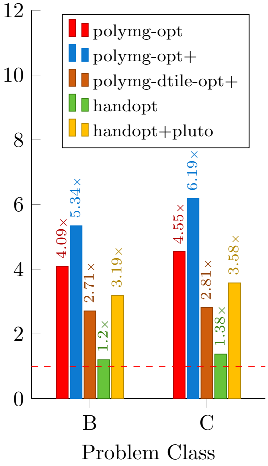


(a) 2D-V-10-0-0

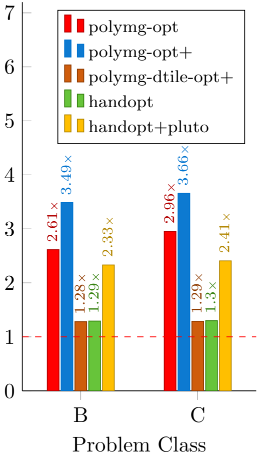


(b) 2D-V-4-4-4

2D Benchmarks: W-cycle

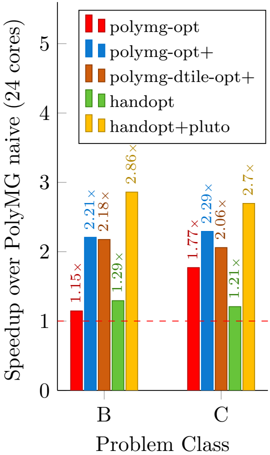


(c) 2D-W-10-0-0

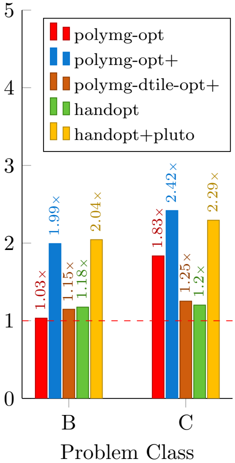


(d) 2D-W-4-4-4

3D Benchmarks: V-cycle

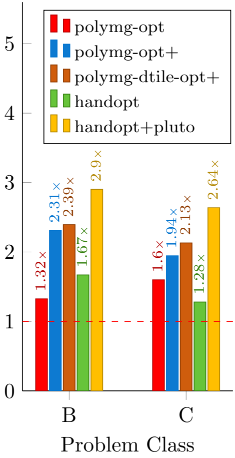


(e) 3D-V-10-0-0

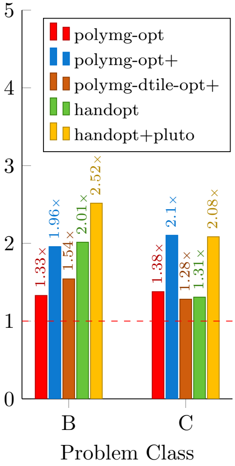


(f) 3D-V-4-4-4

3D Benchmarks: W-cycle

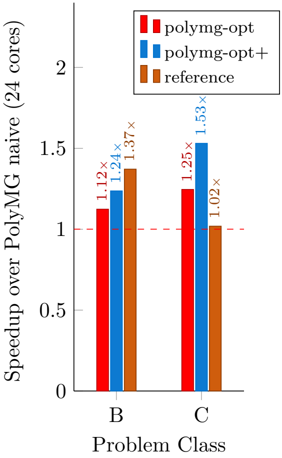


(g) 3D-W-10-0-0



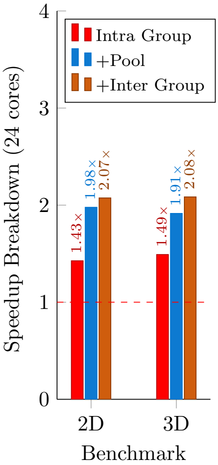
(h) 3D-W-4-4-4

NAS-MG



polymg-opt
polymg-opt+
reference
Speedup over PolyMG naive (24 cores)
0
0.5
1
1.5
2
1.12×
1.24×
1.37×
1.25×
1.53×
1.02×
B
C
Problem Class

Storage Optimization Gains



V-cycle for 10-0-0

Summary of Improvements

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

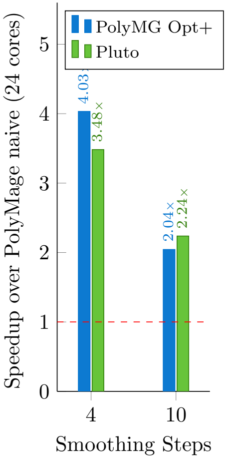
Conclusions

- **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

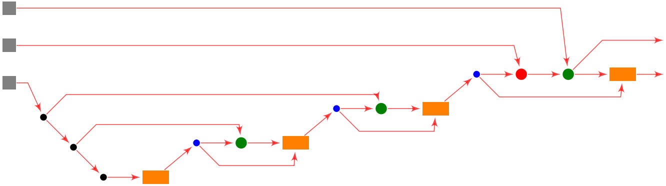


3D stencil for class C

Sample Output

```
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 Ti = -1; Ti <= N/32; Ti+=1) {
    for (int Tj = -1; Tj <= N/512; Tj+=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 ubi = min(N, 32*Ti + 49);
      int lbi = max(1, 32*Ti);
      for (int i = lbi; i <= ubi; i+=1) {
        int ubj = min(N, 512*Tj + 529);
        int lbj = max(1, 512*Tj);
#pragma ivdep
        for (int j = lbj; (j <= ubj); j+=1) {
          _buf_2_0[(-32*Ti+i)*530 + -512*Tj+j] = ...;
        }}

      int ubi = min(N, 32*Ti + 48);
      int lbi = max(1, 32*Ti);
      for (int i = lbi; i <= ubi; i+=1) {
        int ubj = min(N, 512*Tj + 528);
        int lbj = max(1, 512*Tj);
#pragma ivdep
        for (int j = lbj; (j <= ubj); j+=1) {
          _buf_2_1[(-32*Ti+i)*530 + -512*Tj+j] = ...;
        }}
      ...
  }}
  ...
  pool_deallocate(_arr_10_2);
  ...
}
```

NAS-MG V-Cycle



Smoother

Defect/Residual

Correction

Restrict/Reciprocate

Interpolate/Prolongation