Converting Stencils to Accumulations for Communication-Avoiding Optimization in Geometric Multigrid

Protonu Basu[1], Samuel Williams[2], Brian Van Straalen[2], Leonid Oliker[2], Mary Hall[1]
[1]School of Computing, University of Utah, Salt Lake City, UT 84112
[2]Lawrence Berkeley National Laboratory, Berkeley, CA 94720

ABSTRACT

This paper describes a compiler transformation on stencil operators that automatically converts a standard stencil representation into an accumulation. We use this as an enabling transformation to optimize the stencil operators in the context of Geometric Multigrid (GMG), a widely used method to solve partial differential equations. GMG has four stencil operators, the smoother, residual, restriction, and interpolation some of which require inter-process and inter-thread communication. This new optimization allows us, at each level of a GMG V-Cycle, to fuse all operators when recursing down the V-Cycle, and all smooth operations when returning up the V-Cycle. In turn, this fusion allows us to create a parallel wavefront across the fused operators that reduces communication. Thus, these combined optimizations reduce vertical (through the memory hierarchy) data movement and horizontal (inter-thread and inter-process) messages and synchronization.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers, Optimization, Code Generation

Keywords

Geometric Multigrid, Compiler Optimization, Autotuning

1. INTRODUCTION

Stencil operations, or nearest-neighbor structured grid computations, are at the heart of many key numerical applications, making their efficient execution increasingly critical across a broad range of domains, including fluid dynamics, combustion, electromagnetics and MRI imaging. Optimizing stencil computations is a well studied area, employing techniques like cache oblivious algorithms, time skewing, wavefront optimizations and overlapped tiling [5, 10, 11, 17, 19, 24, 27, 32–34]. In many of these efforts, the problems were simplified as compared to real-world applications, by using 2-dimensional or constant-coefficient stencils without control flow. Moreover, most of them consider the stencil computations in isolation rather than examining the solver or numerical method as a whole.

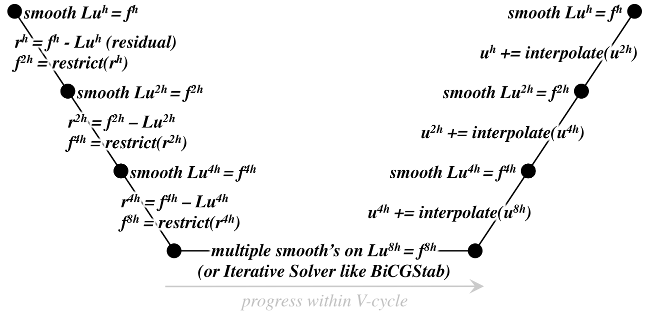
smooth $Lu^h = f^h$
$r^h = f^h - Lu^h$ (residual)
$f^{2h} = restrict(r^h)$
smooth $Lu^{2h} = f^{2h}$
$r^{2h} = f^{2h} - Lu^{2h}$
$f^{4h} = restrict(r^{2h})$
smooth $Lu^{4h} = f^{4h}$
$r^{4h} = f^{4h} - Lu^{4h}$
$f^{8h} = restrict(r^{4h})$
multiple smooth's on $Lu^{8h} = f^{8h}$
(or Iterative Solver like BiCGStab)
progress within V-cycle
u^{4h} += interpolate(u^{8h})
smooth $Lu^{4h} = f^{4h}$
u^{2h} += interpolate(u^{4h})
smooth $Lu^{2h} = f^{2h}$
u^h += interpolate(u^{2h})
smooth $Lu^h = f^h$

Figure 1: The Multigrid V-cycle for solving $Lu^h = f^h$. Note, superscripts denote grid spacing.

This paper examines stencil optimizations in the context of Geometric Multigrid (GMG), a family of algorithms used to accelerate the convergence of iterative solvers for linear systems. To enable a portable solution, we use a compiler framework to generate optimized code for the *miniGMG* benchmark [18, 29, 30], which proxies the multigrid solvers in Adaptive Mesh Refinement (AMR) applications.

Geometric Multigrid solvers perform a few basic operations, which can be seen in Figure 1. The smoother, residual and restriction operations are all stencil computations. The smoother is composed of stencils and pointwise updates to a structured grid. Our work in this paper focuses on *variable-coefficient* stencils for the smoother and residual operators and a *constant-coefficient* stencil for the restriction operator. In Geometric Multigrid, the smooth operation usually dominates execution time and variable-coefficient smoothers in GMG are heavily memory bound as not only the stencil points but also their coefficients must be read from memory. As a result, managing data movement through the memory hierarchy and reducing inter-process communication are key to achieving high performance.

Communication-avoiding optimizations targeting GMG on modern architectures have been used in both manual tuning and been built into compiler frameworks [1, 30]. Such optimizations reduce vertical communication through the memory hierarchy and horizontal communication across processes or threads, sometimes at the expense of redundant computation. Previously described communication-avoiding optimizations built into the loop transformation framework

WOSC '14 October 20, 2014, Portland, OR, USA.

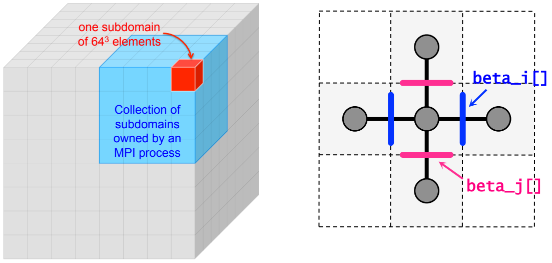
one subdomain of 64³ elements
Collection of subdomains owned by an MPI process
beta_i[]
beta_j[]

Figure 2: (left)Domain decomposition in miniGMG from global problem to individual MPI processes to individual boxes (subdomains) of elements. (right) 2D visualization of the local 3D stencil operation.

CHiLL, a polyhedral transformation and code generation framework, optimize just the smooth operator and are used in conjunction with autotuning to find the optimal ghost-zone and wavefront depth [1].

This paper extends the capabilities of CHiLL to include a novel transformation that automatically converts a stencil computation into an accumulation. The transformation allows the compiler to fuse the residual and restriction stencil operations together, matching the previous manual optimization [30]. We then take communication-avoiding a step further by combining our new optimization with previous work by fusing all smoother operations with the residual and restriction operators. This fusion allows the compiler to create an even deeper wavefront (with a corresponding deeper ghost-zone) with nested thread-level parallelism exploited to manage the larger working sets. Finally, we demonstrate that this additional collection of communication-avoiding optimization improves performance for the in-place stencil updates in Gauss-Seidel Red-black (GSRB) smoothers, but does not pay off for Jacobi-style smoothers. Therefore, we conclude that any optimization framework used for the stencils in GMG must be able to evaluate different optimizations within their execution context (architecture and stencil configuration) to determine which optimizations will achieve the best performance.

2. MULTIGRID AND miniGMG

Elliptic PDE solvers are ubiquitous in scientific computing. Multigrid is an iterative method for solving these elliptic PDEs in linear time. Multigrid solvers calculate a correction to the solution at the current grid resolution using a coarser grid approximation and may be expressed recursively. Geometric multigrid (GMG) uses a structured mesh and a corresponding stencil operator. As restriction and interpolation are geometric, coarse grids have half the number of cells in each dimension (1/8 overall).

Figure 1 visualizes the structure of a multigrid V-cycle for solving $Lu^h = f^h$ in which L is the operator, u is the solution, f is the right-hand side, and superscripts represent grid spacings. At each grid spacing, a *smoother* reduces the error in the solution. The literature presents many smoothers of varying convergence and computational characteristics. The right-hand side of a coarse grid is constructed from the geometric *restriction* of the *residual* $(f^h - Lu^h)$ of the current fine grid. For simplicity, many multigrid solvers terminate when geometric restriction can no longer be performed locally. At this point, the calculation uses a "bottom solver" or coarse grid solver such as BiCGStab [22] or multiple pointwise smooths. With a coarsest grid solution computed, the multigrid algorithm interpolates this correction to progressively finer grids, and subsequently applies the smoother to reduce the error.

Operation	Sweeps	#Flops	#Bytes	Radius
2×7-point VC GSRB	4	50	256	1
4×7-point VC Jacobi	4	104	288	1
8-pt Restriction	1	4	10	1

Table 1: Overview of Operator Characteristics. Note, sweeps represent the number of stencil traversals through the grid per up or down the V-cycle while #Flops and #Bytes represent per cell totals.

This paper uses the miniGMG proxy multigrid benchmark [18, 29] originally evaluated in Williams et al. [30] and our previous work [1]. As shown in Figure 2(left), the benchmark creates a block structured 3D domain partitioned into subdomains (boxes) which are distributed among processes. Boxes must exchange ghost zones via MPI (or buffer copy if local) before application of an operator.

This work explores compiler optimization for the 7-point variable-coefficient (face-averaged coefficients) Helmholtz with either two Gauss-Seidel Red-Black (GSRB) relaxations or four weighted Jacobi relaxations per smooth. Figure 2(right) visualizes this operator, while Table 1 presents a breakdown of the operator characteristics. We use the piecewise constant restriction and prolongation typical of finite volume methods. Although these restriction and prolongation (interpolation) operators do not require ghost zone communication, they are a type of stencil and thus amenable to our optimizations.

To highlight the generality and effectiveness of compiler-based optimizations, we decompose the smooth operations into three separate loop nests: the first applies the Laplacian to the current solution and writes to the temporary array, the second reads the temporary array and forms the Helmholtz operator, and the third performs either a weighted Jacobi or a Gauss-Seidel Red-Black (GSRB) smooth of the current solution. This is slightly different than the version of the code online [18] where a few combinations of stencil and smoother were manually fused. Figures 3, 4 and 5 illustrate the three operators used as input to the compiler. We now describe the compiler-driven optimization transformations used in our study.

3. STENCIL OPERATORS AS POINTWISE ACCUMULATION

The novel optimization described in this paper reorders constant coefficient stencil operations by exploiting their associativity. The stencil operator is converted to an accumulation of right-hand side terms (weighted points) into the output location for the stencil. As presented, this optimization works for box-shaped stencils where each input element is used in the computation of only a single output element. This case arises in Figure 5, where the restriction operator in miniGMG uses the eight values from a fine grid to compute a single value of a coarse grid. As mentioned before, the restriction operation is integral to geometric multigrid, and the piecewise constant restriction used here is common to finite volume methods. The compiler can therefore safely

```
/* Laplacian(phi) = b div beta grad phi */
for (k=0;j<N;k++)
  for (j=0;j<N;j++)
    for (i=0;i<N;i++)
    /* statement S0 */
    temp[k][j][i] = b*h2inv*(
       beta_i[k][j][i+1]*( phi[k][j][i+1]-phi[k][j][i]  )
      -beta_i[k][j][i]  *( phi[k][j][i]  -phi[k][j][i-1])
      +beta_j[k][j+1][i]*( phi[k][j+1][i]-phi[k][j][i]  )
      -beta_j[k][j][i]  *( phi[k][j][i]  -phi[k][j-1][i])
      +beta_k[k+1][j][i]*( phi[k+1][j][i]-phi[k][j][i]  )
      -beta_k[k][j][i]  *( phi[k][j][i]  -phi[k-1][j][i])
    );

/* Helmholtz(phi) = (a alpha I - laplacian)*phi */
for (k=0;j<N;k++)
  for (j=0;j<N;j++)
    for (i=0;i<N;i++)
    /* statement S1 */
    temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i]
                        -temp[k][j][i];

/* GSRB relaxation: phi = phi - lambda(helmholtz-rhs) */
/* Jacobi smoother is similar but without if-condition */
/* and reads, writes separate grids */
for (k=0;j<N;k++)
  for (j=0;j<N;j++)
    for(i=0;i<N;i++){
    if((i+j+k+color)%2==0)
    /* color is 0 for Red pass, 1 for black */
    /* statement S2 */
    phi[k][j][i] = phi[k][j][i] - lambda[k][j][i]*(
                     temp[k][j][i]-rhs[k][j][i]
                   );
  }
```

Figure 3: The Smooth operator.

reorder these calculations by converting the array accesses in the stencil into two loop nests. The first loop nest initializes the output grid, and the second loop nest iterates through points in the input grid and accumulates their values in the output grid. The resulting new loops are then fused together to reduce stencil grid sweeps, and thus improve DRAM traffic, as grids are likely to exceed last-level cache capacity. In the resultant fused loop nest, the leading plane in the output grid (the smaller coarse grid) is first initialized to zero. Subsequently, points in the trailing input plane are read, restricted, and stored to the output (coarse) grid.

While this optimization could be used for register and other optimizations of stencils, as similar techniques described in [7,25], we instead use it here as an enabling transformation for the communication-avoiding optimizations described in the subsequent subsections.

3.1 Using Reordering Transformation to Fuse Operators

The optimization shown as the output of Figure 5 enables the fusion of the restriction operator with the preceding residual operator in Figure 4, as it eliminates the fusion-preventing dependences on the variable `res`. This fusion is a vertical communication-avoiding optimization, as it exposes reuse of `res` in cache. However, since execution time is dominated by the smooth operator, we would like to additionally fuse smooth, residual and restriction together.

In our prior work on compiler-based optimization of GMG, the compiler fused the Laplacian, Helmholtz and GSRB of Figure 3, employing array data-flow analysis to contract the iteration space and eliminate a fusion-preventing dependence [1]. In addition, as described in Section 2, the smooth is applied multiple time steps in a row. To further fuse the multiple smooths together, our prior work introduced *ghost zones* that store data associated with redundant computation across time steps, a well-known horizontal communication-avoiding optimization. This allowed multiple applications of smooth per communication exchange phase, with one layer of ghost zone per smooth application.

```
/* Input: Residual Operation */
for(k=0;k<K;k++)
  for(j=0;j<J;j++)
    for(i=0;i<I;i++)
      /* statement S3 : Compute residual */
      res[k][j][i] = rhs[k][j][i]
        - a * alpha[k][j][i] * phi[k][j][i]
        - b*h2inv*(
           beta_i[k][j][i+1]*( phi[k][j][i+1]-phi[k][j][i]  )
          -beta_i[k][j][i]  *( phi[k][j][i]  -phi[k][j][i-1])
          +beta_j[k][j+1][i]*( phi[k][j+1][i]-phi[k][j][i]  )
          -beta_j[k][j][i]  *( phi[k][j][i]  -phi[k][j-1][i])
          +beta_k[k+1][j][i]*( phi[k+1][j][i]-phi[k][j][i]  )
          -beta_k[k][j][i]  *( phi[k][j][i]  -phi[k-1][j][i])
        );
```

Figure 4: The Residual operator.

```
/* Input: Restriction Operation */
for(k=0;k<K;k+=2)
  for(j=0;j<J;j+=2)
    for(i=0;i<I;i+=2)
      coarser_res[k/2][j/2][i/2] = 0.125 *(
        res[k  ][j  ][i] + res[k  ][j  ][i+1] +
        res[k  ][j+1][i] + res[k  ][j+1][i+1] +
        res[k+1][j  ][i] + res[k+1][j  ][i+1] +
        res[k+1][j+1][i] + res[k+1][j+1][i+1]
       );

/* Output: Restriction as a Scatter Operation */
for(k=0;k<K;k+=2)
  for(j=0;j<J;j+=2)
    for(i=0;i<I;i+=2)
      /* statement S4 : Initialize coarse_res*/
      coarser_res[k/2][j/2][i/2] = 0;

for(k=0;k<K;k++)
  for(j=0;j<J;j++)
    for(i=0;i<I;i++)
      /* statement S5 : Restrict fine_res to coarse_res */
      coarser_res[k/2][j/2][i/2] += 0.125* res[k][j][i];
```

Figure 5: Example code for the Restriction operation and the transformed output.

In this paper, we fuse these multiple smooth applications together with residual and restriction. Since typically there is communication of variable `phi` between smooth and residual, the fusion will require an additional layer of ghost zone to effectively avoid this communication. With all these operators fused together, we only need one communication phase to exchange ghost-zone data for the application of the smooth, residual and restriction, thus further eliminating expensive horizontal communication.

3.2 Communication Avoiding Optimizations: Wavefront Computation

The fusion and larger ghost-zones not only improves performance, but also presents an opportunity to further reduce vertical communication to the DRAM by creating a wavefront computation. Figures 6 and 7 illustrate the wavefront optimization — which reduces multiple sweeps of the grid into a single sweep — for smooth, and for smooth fused with

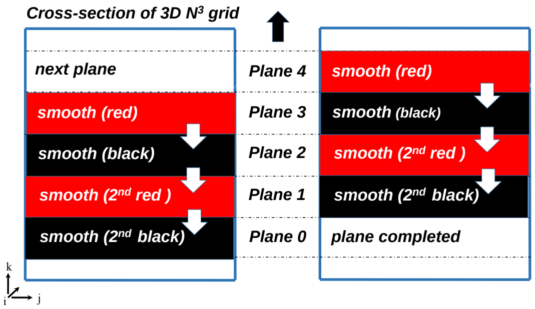

Figure 6: Figure illustrates a cross-sectional view of a four deep wavefront for GSRB which processes a red, black, red and then black planes in that order.

residual and restriction, respectively.

Figure 6 shows the progress of a four deep wavefront for the GSRB smoother (using a four deep ghost zone). The left and right grids in the figure are the before and after snapshots of the wavefront as it sweeps through the 3D grid. Planes (3,2,1,0) are held in memory and processed in that order as shown by the white arrows in the figure; the trailing plane, plane 0, has been fully processed after the second black smooth is applied to it. The wavefront then moves on to process planes (4,3,2,1). Thus after a single grid sweep, all planes have been processed. CHiLL creates the wavefront via the following transformations: overlapped ghost zones, loop skewing and permutation [1].

This paper extends the wavefront optimization, as shown in Figure 7, which illustrates a deeper wavefront where the two GSRB computations, the residual and restriction are all fused into one sweep. Since the residual operation requires ghost zone data, the subdomains are required to exchange ghost zones between computing the GSRB smoother and the residual. By using a five deep ghost zone instead of four, the optimization avoids horizontal communication between smooth and residual. Our approach can then exploit the larger ghost zone to create the deeper wavefront, computing smooth, residual and restriction in a single sweep.

Deeper wavefronts increases the working set even further, thus necessitating collaborative threading to effectively reduce working set pressure by having more than one thread work on a subdomain (box). Figure 8 shows possible threading strategies on Hopper that contains 6 cores/chip, where x boxes are being processed in parallel, while y threads collaborate inside a box ($x * y$ is the number of cores per chip).

Figure 9 shows CHiLL-generated code corresponding to Figure 7 for a 64^3 box. There are three threads working collaboratively inside the box, and they synchronize using spin locks. The code illustrates that the *k-loop* was skewed against the time *t-loop*, and then they are permuted, making k the outer-loop and giving a single grid sweep (in the k-dimension). The smooths are followed by initializing a plane of the output coarser grid, and then the computation of the residual and the restriction.

In a similar manner, wavefront computations can be generated for Jacobi style stencil, but Jacobi stencils present additional challenges to the memory system. Jacobi reads and writes to different arrays leading to an even larger working set. Collaborative threading for Jacobi stencils is less effective, since due to dependences, threads must synchronize after computing each plane, unlike GSRB, where all the planes in the wavefront can be computed before the threads need to synchronize. Thus deeper wavefront may not always help Jacobi smoothers.

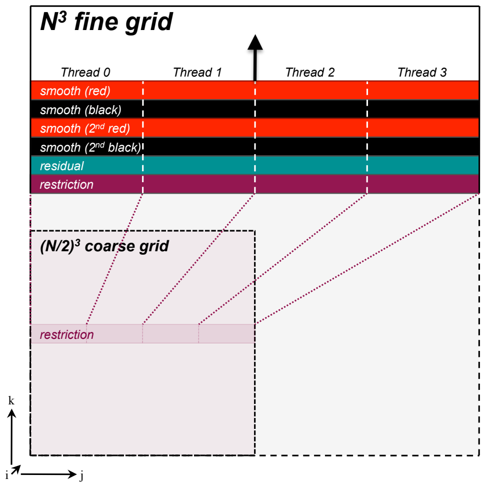

Figure 7: Visualization of the fused and threaded GSRB+Residual+Restriction wavefront strategy exploited when pre-smoothing in the V-Cycle. This wavefront uses a deeper wavefront than the figure above. Hence it is able to compute the GSRB smoother, residual and restriction in one sweep of the fine grid. Note, Jacobi or Chebyshev smoothers can be similarly implemented. Residual and restriction are not present when post-smoothing.

Integrating these transformations into the compiler allows the flexibility of selectively applying them on only the suitable architectural configurations, as demonstrated in Section 5.

4. COMPILER IMPLEMENTATION

The transformations described in the previous section have been implemented in the CHiLL polyhedral transformation and code generation framework [3] and extend previous communication-avoiding optimizations in CHiLL targeting GMG [1]. This section describes the abstractions used by the compiler and the implementation of the transformation that converts a stencil to a pointwise accumulation.

We make the following assumptions about the input code to our framework. Stencil computations may be either out-of-place or in-place. Out-of-place updates are loop nest computations where the right-hand sides are read-only matrices per time step of the stencil computation (e.g., Jacobi). In-place stencil computations read and write the same matrix each time step (e.g., Gauss-Seidel). Stencils may be the sum of pairwise products of two or more matrices (variable-coefficient) or a weighted sum of a single matrix (constant coefficient).

The accumulation transformation described in Section 3 targets constant coefficient out-of-place stencils, such as the 8-point stencil of the restriction operator. As is standard with polyhedral compiler frameworks, we require that all subscript expressions are *affine*, or linear combinations of the loop indices and loop-invariant variables.

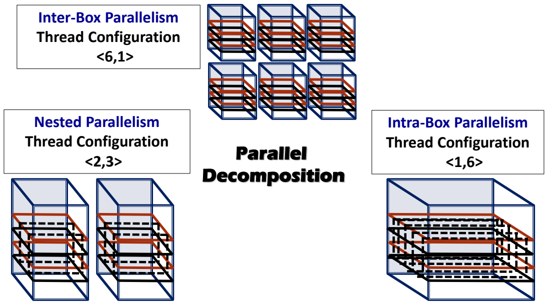

Figure 8: Space of parallel decompositions on Hopper which has 6-cores per chip. All the boxes in a subdomain may work in parallel, or all the threads may work on one box collaboratively, or nested parallelism may be used.

Background on Polyhedral Compiler Frameworks. In miniGMG and other representative applications, stencils are implemented as multi-dimensional loop nest computations (assumed to be 3-dimensional in this paper). In CHiLL, we represent this loop nest by an iteration space *IS*, which mathematically represents the polyhedra corresponding to points in the 3D iteration space as follows:

$$IS = \{[l_1, l_2, l_3] : 0 \leq l_1, l_2, l_3 < N\} \qquad (1)$$

By convention, l_3 is the innermost loop. It is standard to normalize iteration spaces to start at 0 as in this example. Bounds constraints can be far more complex, but for simplicity, we show an upper bound that is a constant or variable. The compiler applies a series of affine transformations that map from the original iteration space to transformed iteration spaces. A separate dependence graph is used to determine the safety of the transformations to be applied. When all transformations are completed, polyhedra scanning is used to scan the iteration spaces and generate transformed loop nests [2]. The array index expressions are mapped to the new iteration space, but are typically otherwise copied from the original input. In the following, the transformation we describe modifies both iteration spaces and statements.

4.1 Converting to Accumulation

Examining the statement associated with a stencil computation, the compiler computes a bounding box of the points in the stencil statement, such that each dimension derives its lower and upper bound from the minimum and maximum values in that dimension. Concretely, at every iteration $\vec{I} = \langle i_1, i_2, i_3 \rangle \in IS$, we compute a single output of the stencil. For a 2D stencil, for example, this requires loading $\{[i_1][i_2]$, $[i_1][i_2 \pm 1]$,$[i_1 \pm 1][i_2]$,$[i_1 \pm 1][i_2 \pm 1]\}$ from input array `in`, and writing the weighted sum of these points to output array **out**$[i_1][i_2]$ on every iteration. Using the above notation, we can rewrite an out-of-place constant coefficient *p*-point stencil as a weighted sum of *p* points.

$$\text{out}[i_1][i_2][i_3] = \sum_{m=1}^{p} w_m * \text{in}[i_1{+}o_1^m][i_2{+}o_2^m][i_3{+}o_3^m]$$

Using this notation, each stencil point m is characterized by its constant coefficient w_m, and the vector offset from the iteration $\vec{I}$, $\vec{O}_m = \langle o_1^m, o_2^m, o_3^m \rangle$.

The bounding box is computed from the lower and upper bounds for each dimension.

```
#pragma omp parallel  private(...)\
                    shared(locks) num_threads(3)
{
tid = omp_get_thread_num();
num_threads = omp_get_num_threads();
left = min(tid - 1,0);
right = max(tid + 1,num_threads - 1);

for(k = -4; k <= 67; k++){
  for(t = 0; t <= min(3,intFloor(k+4,2)); t++){
    for(j = 24*tid-4; j <= 24*tid + 19; j++){
      for(i= -4+intMod(-k-color-j-(t-4),2); i<=67; i+=2) {
        S0(t,k-t,j,i); /* Laplacian */
        S1(t,k-t,j,i); /* Helhmoltz */
        S2(t,k-t,j,i); /* GSRB      */
      }}} /* End t,j,i */

  if(4<=k && intMod(k,2) == 0){
    for(j = max(24 * tid - 4,0);
         j <= min(62,24 * tid + 18); j += 2){
      for(i = 0; i <= 62; i += 2 ) {
      S4(t,k-t,j,i); /* Initialize coarse_res  */
      }}} /* End if */

  if(4<=k){
    for(j = max(24 * tid - 4,0);
         j <= min(62,24 * tid + 18); j ++){
      for(i = 0; i <= 63; i ++ ) {
      S3(t,k-t,j,i); /* Compute residual  */
      S5(t,k-t,j,i); /* Restrict fine_res to coarse_res */
      }}} /* End if */

  /* After computing the 5-deep wavefront */
  /* Threads sync with their right and left neighbors */

  locks[tid] = k;
  if(left!= tid){ while (locks[left] < k) pause();}
  if(right!= tid){ while (locks[right] < k) pause();}

    }/* End k */

}/* End OMP region */
```

Figure 9: Simplified generated code for threaded wavefront with GSRB and residual and restriction fused. The code is specialized for a 64^3 box, with a 5-deep ghost zone and 3-threads working inside a box.

$$lb_1 = min_m o_1^m, \quad lb_2 = min_m o_2^m, \quad lb_3 = min_m o_3^m$$
$$ub_1 = max_m o_1^m, \quad ub_2 = max_m o_2^m, \quad ub_3 = max_m o_3^m$$

$$BoundingBox = \{[b_1,b_2,b_3]\colon lb_1 \leq b_1 \leq ub_1,\ lb_2 \leq b_2 \leq ub_2,\ lb_3 \leq b_3 \leq ub_3\ \}$$

If the set of points of the stencil and its bounding box represent the identical volume, as in the case of the 8-point restriction operation, we call this a *full* stencil. In this paper, we will restrict this accumulation optimization to computations on full stencils. If we are willing to tolerate complex control flow to determine what points to execute, and therefore potentially inefficient code, more general stencils can be supported by the compiler using operations such as convex hull and set difference, which are supported in CHiLL/Omega+.

The key concept is that the compiler rewrites the statement representing the stencil as an accumulation and modifies the loop nest accordingly. From the perspective of a polyhedral compiler, it adds additional loops to the iteration space *IS* for the computation, and replaces the stencil statement with a different statement that reads and accumulates into the output variable. For correctness, the compiler must also create a new statement and iteration space to initialize the output variable. These steps are shown in Figure 10.

Initialize Output.

```
Assume stencil statement S0 has iteration space IS
Create initialization statement S1 with iteration space IS.
```

out$[l_1][l_2][l_3]$ = 0;

```
Insert S1 lexicographically before S0.
```

Expand the iteration space for the accumulation.

```
Create a mapping M for IS to incorporate BoundingBox.
```

$$\begin{aligned} M := \{[l_1, l_2, l_3] \to [l'_1, b_1, l'_2, b_2, l'_3, b_3] :& \\ l'_1 = l_1, l'_2 = l_2, l'_3 = l_3,& \\ lb_1, lb_2, lb_3 \le b_1, b_2, b_3 \le ub_1, ub_2, ub_3\}& \end{aligned}$$

```
Apply M to IS to create new iteration space IS'
IS' = M(IS)
/* In special case IS' can be simplified to        */
/* coalesce the loops and is described in the text */
```

Replace S0 with the new statement S2.

```
Create new statement S2 with iteration space IS'.
```

out$[l_1][l_2][l_3]$ += in$[l_1{+}b_1][l_2{+}b_2][l_3{+}b_3]$;

Figure 10: Code generation steps for accumulation transformation.

The example below illustrates this approach for the restriction operator, The iteration space *IS* is:

$$IS = \{[k, j, i] : \exists\alpha : (0 \le k, j, i < N \;\&\&\; i, j, k = 2 * \alpha)\}$$

The original stencil statement is modified and two new statements are created, *S1*(initialization) and *S2*(accumulation) are as follows (c_1 is the common constant coefficient of the stencil, if the stencil has unique coefficients, we will use coeff$[b_1][b_2][b_3]$ instead):

S1: coarser_res[k/2][j/2][i/2] = 0;
S2: coarser_res[k/2][j/2][i/2] += c_1 * res$[k{+}b_1][j{+}b_2][i{+}b_3]$;

The iteration space for the initialization statement *S1* is the original iteration space for the restriction loop, which we call *IS*. The iteration for *S2* is modified. In Figure 10, the relation M maps the original iteration space *IS* to the new iteration space IS.

IS' = *M* (*IS*)

For the restriction computation, where there is a stride on the original loops and the bounding box is contained within the stride, it is beneficial to apply an additional coalescing transformation to IS to eliminate the loops associated with the bounding box and convert the k, j, and i loops to unit stride accesses.

$$\begin{aligned} M' := \{[k, b_1, j, b_2, i, b_3] \to [k', j', i'] :& \\ k - b_1 \le k' \le k + b_1, j - b_2 \le j' \le j + b_2,& \\ i - b_3 \le i' \le i + b_3\}& \end{aligned}$$

IS_{final} = *M'* (*IS'*)

When the code generator scans the loop nests associated with IS and IS_{final}, the compiler then derives the code shown as the output of Figure 5.

Although not strictly a polyhedral transformation, the new statements and loop iteration spaces are essentially converting subscript expressions to loop iteration spaces, and rely heavily on the polyhedral framework abstractions. A similar approach is used in CHiLL to introduce statements and modify the iteration space when performing loop unrolling [14].

	Intel	**AMD**
Core Architecture	**Ivy Bridge**	**Opteron**
Clock (GHz)	2.40	2.1
DP GFlop/s	19.2	8.4
Data Cache (KB)	32+256	64+512
	Intel	**AMD**
Chip Architecture	**Xeon E5-2695v2**	**Opteron 6172**
Cores	12	6
Last-level Cache	30 MB	5 MB
DP GFlop/s	230.4	50.4
STREAM Bandwidth	45 GB/s	12 GB/s
Memory Capacity	32 GB	8 GB
	Cray XC30	**Cray XE6**
System	**(Edison)**	**(Hopper)**
CPUs/Node	2	4
Compiler	icc 14.0.0	icc 13.1.3

Table 2: Overview of Evaluated Platforms .

5. EXPERIMENTAL RESULTS

In this paper, we evaluate the benefits of our compiler technology using two different system architectures detailed in Table 2. In both cases, we use the default Intel compiler available on the NERSC machines with `-O3 -fno-alias -fno-fnalias` and either `-xAVX` or `-msse3`.

Edison is a Cray XC30 MPP at NERSC. Each node contains two 12-core Xeon Ivy Bridge chips each with four DDR3-1600 memory controllers and a 30MB L3 cache [9]. Each core implements the 4-way AVX SIMD instruction set and includes both a 32KB L1 and a 256B L2 cache.

Hopper is a Cray XE6 MPP at NERSC. Each node contains four 6-core Opteron chips each with two DDR3-1333 memory controllers and a 6MB L3 cache [13]. Each core implements the 2-way SSE3 SIMD instruction set and includes both a 64KB L1 and a 512KB L2 cache.

In all experiments, our finest grid in miniGMG is 256^3 cells per compute node decomposed into $64{\times}64^3$ boxes distributed among MPI processes with 1 process per NUMA node (2 processes on Edison, 4 processes on Hopper). The V-Cycle is terminated when each box reaches 4^3 cells. We run a fixed 10 V-Cycles with a point relaxation bottom solver.

Figure 11 quantifies the overall speedup on the MG solver attained via the CHiLL compiler as a function of optimizations employed for the 7-point variable-coefficient operator using either the Jacobi or the GSRB smoother. Observe that applying the best communication-avoiding wavefront smoother (requiring 4 ghost zones) can improve performance by up to 2.4×. At that point, the residual and restriction operations can be fused to create the RHS for the next coarser grid. Although this significantly accelerates these operations, the overall benefit is relatively small given the total data movement eliminated is a small fraction of the overall MG solver, which used 4 pre-smooths and 4 post-smooths. Nevertheless, we effectively fuse all operations of pre-smoothing into a single wavefront (“+Wavefront(everything)”) that performs all smooths, residual, and restriction with a single communication phase requiring the exchange of a five deep ghost zone.

The performance benefit is highly dependent on architecture and smoother. A communication-avoiding Jacobi smoother requires a significantly larger per-cache working

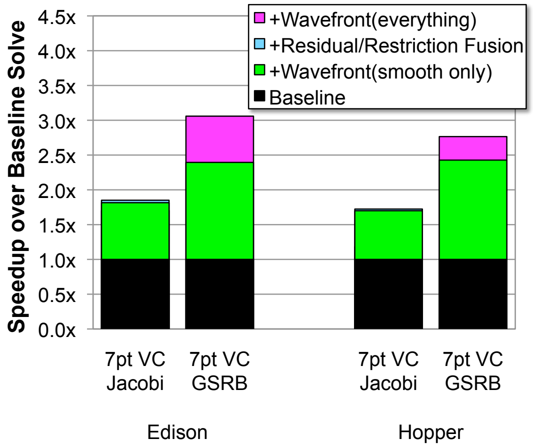

Figure 11: Speedup for the MG Solver attained from incrementally-enabled optimizations obtained in the CHiLL compiler categorized by smoother and architecture. Note, "VC" is variable-coefficient.

set than a communication-avoiding GSRB smoother using multiple threads per box. As GSRB is more likely to maintain a working set in the L2, it should be no surprise that we see a more significant benefit from a communication-avoiding GSRB smoother — up to 2.5× on Edison. The working set continues to balloon as one fuses the Residual and Restriction calculations. At this point, it is unlikely that the working set will be maintained in the L2, thus resulting in reduced bandwidth to the L3 and a differentiation of Hopper and Edison — with the latter attaining a significant net speedup over 3× for the variable-coefficient GSRB.

6. RELATED WORK

In the past, operations on large structured grids could easily be bound by capacity misses in cache, leading to a variety of studies on blocking and tiling optimizations [6,8,15,21,23, 28,31]. In recent years, numerous efforts have focused on increasing temporal locality by fusing multiple stencil sweeps through techniques like cache oblivious, time skewing, wavefront or overlapped tiling [5,10,11,16,17,19,24,27,32–34,36]. In addition, domain-specific compilers have recently been developed for parallel code generation from a stylized stencil specification [4,26,35] or from a code excerpt [12]. Manual optimization of stencil computations has developed techniques such as *semi-stencils* to reduce loads [7].

Prior work developed manual optimizations for multigrid that incorporated communication-avoiding optimizations such as fusion of operators, wavefront parallelism, and hierarchical threading [30]. In addition, lower-level optimizations such as explicit use of SIMD and prefetch intrinsics improved the performance of the inner loops. Subsequently, the communication-avoiding optimizations were implemented in an autotuning compiler and demonstrated for just the smooth operator [1]. The compiler-generated code yielded comparable performance as compared to manual optimization, even though some of the optimizations from the manual version were not included. In this paper, we expand the compiler-directed optimizations to include a stencil reordering transformation that rewrites stencils as accumulations. This rewriting enables fusion of the residual and restriction operators, and the possibility of fusing of all of the operators in a V-cycle. When optimizing a complex proxy application such as miniGMG, we have to optimize several stencil operators and how they work together. To the best of our knowledge, the DSL Halide [20] which focuses on image processing pipelines is the only other framework to do so.

Recent work reorders stencil computations from the direct specification of the stencil as an update from a set of input points [25]. Stencils are converted to a reduction and then loop shifting exposes register reuse of the same input, contributing to different output. Their work focuses on exploiting reuse in registers for higher-order stencils. However, it does not automatically derive the accumulation from general stencil code or initialize the output.

7. CONCLUSION

This paper expands on prior work to develop automatic compiler and autotuning technology that is able to achieve manually-tuned levels of performance (or better in some cases) for Geometric Multigrid. Our work describes a specific optimization that converts a stencil to an accumulation. Our paper applies this optimization to enable fusion of smooth, residual and restriction operators. This in turn permits the compiler to introduce a deeper ghost zone and reduce the number of sweeps in the grid, thus providing a performance gain of over 3× as compared to the baseline implementation on Edison, and a 20% gain as compared to optimizations on only the smooth operator for variable-coefficient GSRB smoothers on Edison.

Our long-term goal is to develop domain-specific optimizations for stencil computations, with a particular focus on the context of GMG. The miniGMG proxy application permits exploration of compiler techniques in the context of different stencil computations, data sets and target architectures. Optimizing every possible combination of operator, smoother and target architecture is not feasible. Therefore, the availability of an optimization framework that can selectively apply such transformations and automatically tune for the right implementation provides a powerful starting point for productively deriving high-performance GMG implementations for current and future architectures.

Acknowledgments

Authors from Lawrence Berkeley National Laboratory were supported by the U.S. Department of Energy's Advanced Scientific Computing Research Program under contract DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

8. REFERENCES

[1] P. Basu, S. Williams, A. Venkat, B. Van Straalen, M. Hall, and L. Oliker. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *High Performance Computing Conference (HIPC)*, 2013.

[2] C. Chen. Polyhedra scanning revisited. In *Proc. of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[3] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[4] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, 2011.

[5] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[6] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC 2008)*, 2008.

[7] R. De La Cruz, M. Araya-Polo, and J. M. Cela. Introducing the semi-stencil algorithm. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, PPAM'09, pages 496–506, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] C. C. Douglas, J. Hu, M. Kowarschik, U. RÃijde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal*, 10:21–40, 2000.

[9] Edison:. `http://www.nersc.gov/users/computational-systems/edison`.

[10] M. Frigo and V. Strumpen. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS05)*, Boston, MA, 2005.

[11] P. Ghysels, P. KÅĆosiewicz, and W. Vanroose. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra with Applications*, 19(2):253–267, 2012.

[12] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proc. of the 26th ACM international conference on Supercomputing*, ICS '12. ACM, 2012.

[13] Hopper:. `http://www.nersc.gov/users/computational-systems/hopper`.

[14] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25, Jan. 2013.

[15] M. Kowarschik and C. WeiÃ§. Dimepack - a cache-optimized multigrid library. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), volume I*, pages 425–430. CSREA, CSREA Press, 2001.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proc. of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, New York, NY, USA, 2007. ACM.

[17] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

[18] miniGMG compact benchmark. `http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/xtune/miniGMG`.

[19] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10. IEEE Computer Society, 2010.

[20] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[21] G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proc. of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.

[22] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, Philadelpha, PA, 2003.

[23] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[24] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[25] K. Stock, M. Kong, T. Grosser, F. Rastello, L.-N. Pouchet, and J. R. P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edimburgh, UK, June 2014.

[26] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proc. of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[27] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *International Computer Software and Applications Conference*, pages 579–586, 2009.

[28] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, Miami, Florida, 2008.

[29] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. mplementation and optimization of miniGMG - a compact geometric multigrid benchmark. Technical Report LBNL 6676E, Lawrence Berkeley National Laboratory, December 2012.

[30] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.

[31] S. Williams, L. Oliker, J. Carter, and J. Shalf. Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning. In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[32] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proc. of the 3rd Conference on Computing Frontiers*, 2006.

[33] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Interational Conference on Parallel and Distributed Computing Systems*, 2000.

[34] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager. Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method. *Progress in Computational Fluid Dynamics*, 8, 2008.

[35] Y. Zhang and F. Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proc. of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, Mar. 2012.

[36] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proc. of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, Mar. 2012.