Compiler Generation and Autotuning of Communication-Avoiding Operators for Geometric Multigrid

Protonu Basu
Anand Venkat
Mary Hall
University of Utah
Salt Lake City, Utah 84112

Samuel Williams
Brian Van Straalen
Leonid Oliker
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

***Abstract*—This paper describes a compiler approach to introducing communication-avoiding optimizations in geometric multigrid (GMG), one of the most popular methods for solving partial differential equations. Communication-avoiding optimizations reduce vertical communication through the memory hierarchy and horizontal communication across processes or threads, usually at the expense of introducing redundant computation. We focus on applying these optimizations to the smooth operator, which successively reduces the error and accounts for the largest fraction of the GMG execution time. Our compiler technology applies both novel and known transformations to derive an implementation comparable to manually-tuned code. To make the approach portable, an underlying autotuning system explores the tradeoff between reduced communication and increased computation, as well as tradeoffs in threading schemes, to automatically identify the best implementation for a particular architecture and at each computation phase. Results show that we are able to quadruple the performance of the smooth operation on the finest grids while attaining performance within 94% of manually-tuned code. Overall we improve the overall multigrid solve time by 2.5× without sacrificing programer productivity.**

I. Introduction

Geometric multigrid (GMG) is an important family of algorithms used by computational scientists to accelerate the convergence of iterative solvers for linear systems. In GMG, floating-point computation is dwarfed by the overhead of data movement, making managing the memory hierarchy and cross-processor communication critical to achieving high performance. Prior work on optimizing the stencil computations that are contained within GMG have led to techniques like cache oblivious algorithms, time skewing, wavefront optimizations and overlapped tiling [8], [12], [13], [19], [20], [23], [28], [32], [35], [36]. For many of these efforts, the problems were simplified as compared to real-world applications, using 2-dimensional or constant-coefficient stencils without control flow and starting from a sequential specification rather than parallel specification.

As modern architectures continue to grow in core count and exhibit a hierarchy of complex inter-thread and inter-process interactions, new *communication-avoiding* techniques have been introduced for GMG that encapsulate several of the optimizations mentioned above. Communication-avoiding optimizations reduce vertical communication through the memory hierarchy and horizontal communication across processes or threads, usually at the expense of introducing redundant computation. Programmers generally need to introduce these optimizations manually, while attempting to discern the optimal combination of parameters, thus resulting in a significant growth in code complexity and non-portability across different architectures. This paper describes how to perform these optimizations automatically by a compiler, generating high-performing code from a relatively straightforward expression of a set of operators in a scalable MPI implementation. Our starting point is the miniGMG benchmark that is intended to represent use of GMG in real-world scalable applications. This benchmark was previously manually tuned across several architectures in [30] and compared to the Roofline Performance Model [33]. Our goal is to automate the communication-avoiding optimizations applied manually in [30], and lay the groundwork for a domain-specific optimization framework supporting GMG in scalable applications.

We focus on optimizations required for the smooth operator, the most compute-intensive portion of GMG. Vertical communication-avoiding optimizations necessitate the support for data-flow analysis, which must be incorporated into the transformations to enable (1) fusing several operators so that intermediate data remains in cache from definition to use; (2) avoiding writes back to memory of temporary data; as well as (3) creating a wavefront so that multiple planes can share data in cache with a minimal cache working set. A horizontal communication-avoiding optimization adds ghost zones to reduce the frequency of inter-processor communication at the expense of redundant computation. While many of these optimizations rely on composing standard loop transformations, most would not be implemented in a standard compiler as they are either domain-specific (e.g., introduction of ghost zones), or specifically effective for GMG classes of computations.

The compiler system of this research effort employs *autotuning* to make these optimizations portable across different architectures. Autotuning systems employ empirical techniques to evaluate the suitability of a search space of possible implementations of a computation. Communication-avoiding smooth operators aggregate communication and perform some redundant work in the ghost zones in order to dramatically improve locality in the last-level cache. The wavefront op-

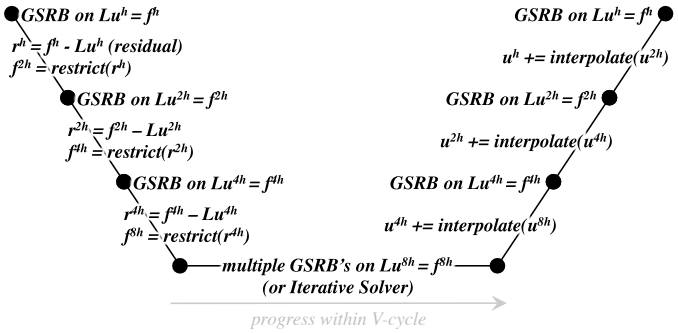

Fig. 1. The Multigrid V-cycle for solving $Lu^h = f^h$. Superscripts represent grid spacing. For large problems, a high-performance, iterative solver is employed at the bottom (coarsest grids).

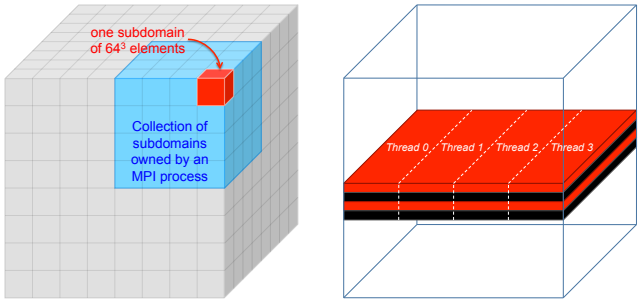

Fig. 2. Visualization of the domain/process/subdomain hierarchy in miniGMG (left) and the threaded GSRB wavefront strategy within the application of smooth() to each subdomain (right).

timization exploits reuse deeper in the memory hierarchy (L1/L2), but risks exceeding capacity limitations if applied too aggressively. The granularity of profitable thread-level parallelism depends on both architecture and the level in the GMG V-cycle. Through autotuning, these tradeoffs can be explored to select the context-specific optimal solutions, across a variety of architectures.

The main contribution of this work is the first exploration of compiler-directed communication-avoiding optimization for GMG. As compared to prior research on domain-specific compilers for the stencil computations that are included within GMG [15], [25], [37], our work more closely addresses the needs of real-world applications because it optimizes in the context of an existing scalable parallel benchmark (miniGMG), and it examines the complex and more representative operator Gauss-Seidel Red Black rather than the simpler Jacobi. Additionally, we enable autotuning to derive the ghost zone depth and threading strategies, thus allowing the automation of differing optimization schemes across individual levels of the V-cycle. This infrastructure is therefore adaptable to next-generation platforms with increasing memory-hierarchy and threading complexity. Overall, we demonstrate portability with a 4x improvement for the most time consuming smooth of the V-cycle, while attaining up to 94% of previously published, highly hand-tuned performance [30].

II. Geometric Multigrid

Multigrid solvers calculate a correction to the solution at the current grid resolution using a solution on a coarser grid. This process may be expressed recursively. Geometric multigrid (GMG) begins with a structured mesh, where each progressively coarser grid contains half the grid points in each dimension. Given the fact that the operators are the same irrespective of grid spacing, this exponential reduction in grid sizes can bound multigrid's computational complexity to O(N) where N is the number of variables. When performance is highly correlated to computational complexity, the time spent on the finer grids will dominate the run time.

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, multiple *smooth operators* reduce the error in the solution. The *smooth* can be a simple relaxation such as Jacobi, or something more complex like a Gauss-Seidel, Red-Black (GSRB).

The right-hand side of the next coarser grid is defined as the *restriction* of the *residual* ($f^h - Lu^h$). Eventually, the grid (or collection of grids) cannot be coarsened any further using geometric multigrid. At that point, most algorithms switch to a bottom solver that can be as simple as multiple relaxations or as complicated as algebraic multigrid, a Krylov iterative solver, or a direct sparse solver. Once the coarsest grid is solved, the multigrid algorithm applies the solution (a correction) to progressively finer grids. This requires an interpolation of u^{2h} onto u^h. At each level, a smooth operator is applied to the new correction.

For problems on relatively few nodes, the performance of *smooth* on the finer grids dominates the run time. In this paper, we therefore focus on the smooth bottleneck, optimizing both the simpler Jacobi that is common to compiler papers [15], [18], [23], and the more complex Gauss-Seidel, Red-Black (GSRB), which predominates real-word applications and requires data-flow analysis and other support for control flow. Overall we demonstrate that our compiler infrastructure can successively optimize both of these relaxation techniques and deliver high performance across our evaluated platforms.

A. mini*GMG Benchmark*

Our work builds on the compact multigrid solver benchmark of Williams et al. [30], called miniGMG. As shown in Figure 2(left), the miniGMG benchmark creates a global 3D domain, and partitions it into subdomains of sizes similar to those found in real-world AMR multigrid applications. The list of subdomains is then partitioned amongst MPI processes. All subdomains must exchange ghost zones after each computation phase, via an MPI call. However, when on the same node, the code is optimized to perform a buffer copy.

To provide direct comparisons to [30], we use the same double-precision, finite volume discretization of the variable-coefficient operator $a\vec{\alpha}I - b\nabla\vec{\beta}\nabla$, with the same periodic boundary conditions, and replicate the 256^3 problem (per compute node) on all platforms. Developers often wish to maintain flexibility and thus create smooth operators by composing multiple simpler operators, as captured in the excerpt of the baseline miniGMG benchmark shown in Figure 3. In this code, the Helmholtz operator requires calculation of

```
/* 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) */

 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]);
    }
```

(a) Smooth operator with Gauss-Seidel Red-Black.

```
/* Go down the v-cycle.... */

for(level=0; level<NumLevel; level++){
    d=ghostZoneDepth[level];
    for ( smooth=0; smooth<NumSmooths; smooth+=d ){
        /* communication phase...the boxes exhange boundaries with neighbors */
        exchange_boundary_phi();
        exchange_boundary_rhs();
        /* Apply smooth on each box in parallel */
        # pragma omp parallel for private (box) num_threads(y)
        for (box=0; box<NumBoxInSubdomain; box++){
            color=smooth;
            gsrb_smooth_function(Domain→SubDomain[box],phi,rhs,color);
        }
    }
    compute_residual();
    /* Restrict to form the coarse and smaller grid */
    /* We go down the v-cycle, ie. from a 64 ∧ 3 grid to a 32 ∧ 3 grid */
    compute_restriction();
}/* down....*/

/* bottom solve.... */
d=ghostZoneDepth[bottom_level];
for (smooth=0; smooth<NumBottomSmooths; smooth+=d){
    exchange_boundary_phi();
    exchange_boundary_rhs();
    /* Apply smooth on each box in parallel */
    # pragma omp parallel for private (box) num_threads(y)
    for (box=0; box<NumBoxInSubdomain; box++){
        color=smooth;
        gsrb_smooth_function(Domain→SubDomain[box],phi,rhs,color);
    }
}/* bottom solve */

/* back up the v-cycle..... */
```

(b) Pseudo-code for a single iteration of the V-cycle.

Fig. 3. Baseline smooth code using Gauss-Seidel Red-Black and outer V-cycle code that includes domain decomposition.

the Laplacian. Thus, the smooth operator in the input code calculates the Laplacian, Helmholtz and either a Gauss-Seidel or Jacobi relaxation in sequence. The Laplacian operator is a seven point, variable-coefficient stencil derived from a finite-volume calculation, while the Helmholtz and relaxation codes nominally scale and add vectors (grids) together. In the code, these become nested loops over the box list, and the spatial dimensions update either every or every other element.

III. Optimizations for Smooth

Starting with the baseline miniGMG code of Figure 3(a), we now describe how our compiler transforms the code to realize both vertical and horizontal communication-avoiding optimizations. The compiler first fuses the multiple smooth operators together. In the case of GSRB relaxation, the control flow guarding the update to `phi` may prevent fusion in some compilers. Incorporating data-flow analysis allows us to fuse the loops safely by contracting the iteration space of the first two statements (see next section). Fusion is itself a vertical communication-avoiding optimization, since the results computed by one operator will remain in cache when used as input by the next operator; an additional communication-avoiding optimization is to replace the array `temp` with a scalar and not write it back to memory on completion. The compiler generates the code in Figure 4(a), with placeholders for the statements corresponding to Figure 3(a).

In the pseudo-code that invokes smooth in Figure 3(b), for each subdomain, the smooth operation is called on all the boxes in parallel. After the smooth operator, the boxes go through a communication phase, where they exchange boundary data with their neighbors. An important communication-avoiding optimization is to create *ghost zones*, which replicate some of the input data across processes and threads. Through the use of ghost zones, the computation can perform several sweeps of the grid per communication step, trading off increased computation for lower communication costs. For the seven-point second-order stencil we consider for our study, an N-deep ghost zone allows N sweeps of the grid between communication. For higher order stencils, the ghost zone depth required increases with order.

We have added a domain-specific compiler transformation to automatically generate a loop over the depth of ghost zones, and modify the iteration space of the operator accordingly to perform the redundant computation. Each sweep of the grid consumes a layer of the ghost zone, and this gives rise to a hyper-trapezoidal iteration space of computation, where the volume shrinks in all dimensions on every smooth to perform computation only on valid data. The GSRB smooth has a red and a black pass, where the points updated depend on their coordinate and the value of `color`. As shown in Figure 3(b), for a grid with a one-deep ghost zone, the value of `color` is updated every time smooth is called. Adding ghost zones requires that the code track the value of `color` and modify the if-condition accordingly. The compiler merges the if-condition into the loop bounds of the innermost loop to generate the

```
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){
    S0(k,j,i); /* Laplacian */
    S1(k,j,i); /* Helhmoltz */
    S2(k,j,i); /* GSRB      */
   }
 }
```

(a) Fused operators.

```
/* d = ghost zone depth */
for (t=0;j<d;t++)
 for (k=t-(d-1);j<N+(d-1)-t;k++)
  for (j=t-(d-1);j<N+(d-1)-t;j++)
   for (i=t-(d-1);i<N+(d-1)-t;i++){
    if((i+j+k+color+t)%2==0){
     S0(t,k,j,i); /* Laplacian */
     S1(t,k,j,i); /* Helhmoltz */
     S2(t,k,j,i); /* GSRB      */
    }
 }
```

(b) After adding ghost zones.

```
/* d = ghost zone depth */
 if (1 <= d && 3<=2*d+N && 3<=2*d+N)
  for (k =-d+1; k<=N+d-2; k++)
   for (t = 0; t<=min(min(min(d-1,(2*d +N-3)/2),
                (d+k-1)/2),(2*d+N-3)/2); t+=1)
  #pragma openmp parallel for private (j,i) num_threads(x)
    for (j = -d+t+1; j<=d+N-t-2; j++)
     for (i =-d+t+1+(-k-color-j-(-d + t + 1)) % 2;
          i<=d-t+N-2; i+=2) {
        S0(t,k-t,j,i); /* Laplacian */
        S1(t,k-t,j,i); /* Helhmoltz */
        S2(t,k-t,j,i); /* GSRB      */
     }
```

(c) Wavefront and threading.

```
for(k = -3; k <= 66; k++)
  for(t = 0; t <= min(3,intFloor(t+3,2)); t++) {
    #pragma openmp parallel for private (j,i) num_threads(x)
    for(j = t-3; j <= -t+66; j++)
      for(i= t-3+intMod(-k-color-j-(t-3),2); i<=-t+66; i+=2) {
        S0(t,k-t,j,i); /* Laplacian */
        S1(t,k-t,j,i); /* Helhmoltz */
        S2(t,k-t,j,i); /* GSRB      */
      }
    }
```

(d) Final code specialized for box size 64^3 and 4-deep ghost zone.

Fig. 4. Steps of optimization process.

transformed code in Figure 4(b).

Adding ghost zones increases DRAM traffic from multiple sweeps over the grid; this vertical communication can be reduced by a streaming strategy called wavefront [34]. A number of planes (up to the number of ghost zones) can be held in memory at once, as shown in Figure 2, and the number of sweeps of the grid reduced by the depth (the number of planes). Keeping multiple planes in memory increases the working set, which may exceed on-chip memory. We generate multi-threaded code via OpenMP to share planes across threads and reduce the working set per thread. Larger grids have a bigger working set than the smaller grids down the V-cycle, which suggests that the system should assign more threads per box for the larger grids, and fewer threads for the smaller grids — ultimately the thread distribution is optimized via autotuning. The threaded wavefront code is in Figure 4(c), which assigns x OpenMP threads per box (intra-box parallelism). Outer-loop parallelism in the harness code of Figure 3(b) assigns y threads to each box (inter-box parallelism).

The autotuning phase tunes the ghost zone depth at each level (*ghostZoneDepth[level]* in Figure 3(b)) and the number of threads x and y controlling intra- and inter-box parallelism, respectively. During the tuning process, constant values for these are bound during code generation, resulting in very efficient context-specific specialized code such as in Figure 4(d).

IV. Compiler Implementation

We now describe the compiler implementation that generates the desired code from the previous section. This compiler is domain specific, in that it employs optimizations that are designed for multigrid applications. The input to the compiler is the code excerpt shown in Figure 3. We performed the implementation using CHiLL, which is a polyhedral transformation and code generation framework [6]. The remainder of this section describes the abstractions used in CHiLL, how these abstractions are used to represent the transformations described in the previous section, the algorithm that performs these transformations, and how autotuning is employed to optimize the code for different architectures.

A. Abstractions in CHiLL

CHiLL is polyhedral transformation and code generation framework designed specifically for autotuning. A polyhedral model represents each statement's execution in the loop nest as a lattice point in the space constrained by loop bounds, known as the *iteration space*. Then a loop transformation can be simply viewed as mapping from one iteration space to another, and various transformations can be composed. CHiLL manipulates iteration spaces derived from the original program, using a *data dependence graph* as an abstraction to reason about the safety of the transformations under consideration [1].

In a polyhedral model, optimized code is generated from the rewritten iteration spaces by scanning the polyhedra representing the iteration spaces of an optimized loop nest from the first dimension to the last. The quality of the generated code directly impacts performance. Therefore, at the heart of CHiLL is a code generator called CodeGen+ that has advanced the state of the art in polyhedral scanning in the presence of conditionals [5], as arise in the GSRB code. CodeGen+ seamlessly combines iteration spaces and guards through a specialized polyhedral AST, as detailed elsewhere [5].

The input to CHiLL is a source code written in C or Fortran, and optionally, a *transformation recipe* describing the set of transformations to be composed to optimize the provided source [14]. This recipe can be written by an expert

programmer, or derived automatically by a compiler decision algorithm [16].

B. Communication-Avoiding Transformations

As we describe each of the communication-avoiding transformations for GMG, we show how the previously-described abstractions are manipulated by CHiLL.

Operator fusion: The input code in Figure 3 consists of three statements S0, S1, and S2, that correspond to the three smooth operators Laplacian, Helmholtz and GSRB, respectively. Once parsed by CHiLL, the iteration spaces corresponding to these operators are as follows:

S0 := { [k,j,i] : 0 ≤ k<N && 0 ≤ j < N && 0 ≤ i <N};
S1 := { [k,j,i] : 0 ≤ k < N && 0 ≤ j < N && 0 ≤ i<N};
S2 := { [k,j,i] : 0 ≤ k < N && 0 ≤ j < N && 0 ≤ i<N &&
k + j + i + 2α + color = 0};

Note that S2 has an additional term in its iteration space related to checking the color for the current element. Operator fusion falls out implicitly from the *iteration space alignment algorithm*, which attempts to carve out a unified iteration space for the imperfect loop nest of the original code [6]. With smooth operators such as Jacobi, iteration space alignment is performed by default in CHiLL. In GSRB, the difficult challenge is to rule out any fusion-preventing dependences when the iteration spaces are not a perfect match.

By default, CHiLL reports a fusion-preventing dependence between S2 and S0 related to the reads and writes of `phi`. However, we make the observation that the iteration spaces for the Laplacian and Helmholtz operators (statements S0 and S1) may compute values of `temp` that are never used by the GSRB of S2. *Array data-flow analysis* can be used to analyze the iteration spaces and access expressions and derive a conservative approximation of the elements of `temp` defined in S0 and S1 and used in S2 [11]. Our compiler determines that the array region read by S2 is a proper subset of the regions defined by S0 and S1. Since `temp` is a local variable redefined on every sweep and it is not live after the smooth operator is completed, it is safe to contract the iteration spaces of S0 and S1 to match that of S2. After the compiler contracts the iteration space, the fusion-preventing dependences are eliminated and CHiLL is able to safely fuse the loops. The iteration space contraction used here is an example of a domain-specific optimization that was proven safe by the compiler, but is more likely to be profitable for GMG operators where Red-Black conditional execution is common.

In the fused code, the compiler recognizes that array `temp` is a local variable, and does not need to be rewritten back to memory. Because there are no dependences on `temp` crossing iteration boundaries, *scalar replacement* is then employed to make this a scalar that is overwritten on each iteration of the loop.

Introducing ghost zones: Once fused, the iteration spaces from the previous section end up with a combined iteration space that matches that of statement S2. We observe that introducing ghost zones as in the previous section is really just introducing a new loop t and changing the bounds for each of the loops in the fused loop nest to compute ghost regions and generate a hyper-trapezoidal iteration space.

Due to the presence of the if-condition in the GSRB smooth, the iteration space is a hyper-trapezoid with holes. The iteration space IS has two distinct components, arising from the loop nest and also the relation (k+j+i+2α+color=0) which represents the if-condition; the iteration space is the conjunction of these terms. We added a new domain-specific transformation `add_ghost_zones`, which maps the old iteration space with the new loop t using the following mapping:

IS : iteration space of the input loop nest
IS′ : iteration space in the modified loop nest

map := { [k,j,i] → [t,k′,j′,i′] :
0 ≤ t<d && k-d+1+t ≤ k′ < k+d-t &&
j-d+1+t ≤ j′ < j+d-t && i-d+1+t ≤ i′ < i+d-t}

IS′ := map (IS);

The variable `color` gets updated with every sweep of the grid, so its value will also be affected by the additional loop. For this purpose, we apply another mapping to `color`:

map′ := { [color] → [color+t] }

This will cause the value of color to be updated everywhere it appears, including within the statements. Although in our current implementation this relation is provided to the implementation, it could be derived automatically through analysis or domain knowledge. This gives a new relation (k+j+i+2α+color+t=0) for the if-condition. The conjunct of the new loop-nest iteration space and the new term gives us the final modified iteration space.

Wavefront and Multithreading: The compiler next generates a wavefront computation [34] using a loop skew and permute, skewing the outermost loop which sweeps the grid, loop `k` in Figure 4(b), against the smooth iteration loop added in the last optimization. Skewing is used to break a dependence that would otherwise prevent permute, using an integer factor in each dimension that depends on the dependence distance in the stencil operation: [1,1] for GSRB and [2,1] for Jacobi. The `k` and `t` loops are then permuted, and the `j` loop is marked for OpenMP parallelization. These are standard loop transformations available in CHiLL.

Autotuning Opportunities: We employ autotuning for two sets of parameters:

• *Ghost zone and wavefront depth:* The ghost zone depth governs both amount of redundant computation performed, and frequency of MPI communication. In the current code generation strategy, the ghost zone depth and number of planes in the wavefront are identical. As memory bandwidth is a key limitation only for larger box sizes, the optimal value for ghost zone depth varies for different box sizes in the V-cycle.

• *Multithreading:* Multithreading is nested at two levels. As shown in Figure 3, the miniGMG already uses a set of OpenMP threads for each box. The final generated code also introduces multiple threads per box, one for each of the planes in the wavefront. Since the box size varies across V-cycles in

a GMG computation, the optimal number of threads per box also varies during the computation.

Putting it together: As the goal of this work is to develop domain-specific optimization techniques for GMG, the compiler algorithm can be specialized for MG implementations that involve composing a set of operators together. The code for the smooth operators that included either GSRB or Jacobi was generated by instantiating a template transformation recipe that is then applied to the input code to generate the optimized code. The recipe that is generated for the smooth that includes GSRB is the following:

```
original()
add_ghosts([S0],L1,d)      #ghost depth is d
skew ([S0], L2, [1,1])     #skew L2 by 1, L1 by 1
permute ([L2,L1,L3,L4])    #new loop order
map_to_openmp_region (L2) #parallelize across boxes
```

The commands in this recipe refer to applying a transformation to a statement at a particular loop level. Once fused, the same transformations are applied to the set of statements S0, S1 and S2 when applied to S0. The only differences for the recipe for the smooth including Jacobi is that there are two statements to which the transformations are applied corresponding to odd/even iterations, and the dependence distance for skewing is different.

During code generation, the value for ghost zone depth and box size are bound to constants. Through an external python script, autotuning varies d from 1 to 2*n where n ranges from 1 to 2, and the thread values x and y ($x*y = 6$ for Hopper and 8 for Edison) for the fused and wavefront variants.This search space of 24 points per level can be explored in a few hours, but with additional optimizations, sophisticated search algorithms are needed to increase the efficiency of the search (e.g., [26]). The resulting application will select the best implementation among precompiled choices.

Generalizing to other GMG Applications: This general approach would apply to other GMG implementations that use a similar composition of operators. When other operators are optimized by the compiler, the approach must be generalized for their associated stencils, deriving the appropriate ghost zone depth based on the dependence distances arising from the stencil shape.

V. Experimental Results

In this section we present an overview of our experimental platforms and a detailed analysis of our performance results.

A. Evaluated Platforms

We evaluate the benefits of our compiler technology on two commodity processor architectures similar to those used by Williams et al. in [30]. Their details are summarized in Table I and explained below. Our code was compiled with icc version 13.0.1 with flags -O3 -fno-alias -fno-fnalias (Hopper:-msse3, Edison:-xSSSE3).

Edison: is a Cray XC30 MPP at NERSC. Each node contains two 8-core Xeon Sandy Bridge chips. Each chip has four DDR3-1600 memory controllers.Each superscalar out-of-order core implements the 4-way AVX SIMD instruction set and includes both a 32KB L1 and a 256B L2 cache. The cores on a chip are connected by a ring to a 20MB L3 cache. The relatively large last-level cache makes capturing locality easier for 64^3 boxes.

Core Architecture	**Intel SNBe**	**AMD Opteron**
Clock (GHz)	2.60	2.1
Double-precision GFlop/s	20.80	8.4
Data Cache (KB)	32+256	64+512
Memory Parallelism	HW-prefetch	HW-prefetch
Chip Architecture	**Intel Xeon E5-2670**	**AMD Opteron 6172**
Cores	8	6
Last-level Cache	20 MB	5 MB
Double-precision GFlop/s	166.4	50.4
STREAM Bandwidth	38 GB/s	12 GB/s
Memory Capacity	32 GB	8 GB
System	**Cray XC30 (Edison)**	**Cray XE6 (Hopper)**
CPUs/Node	2	4

TABLE I. **Overview of Evaluated Platforms**.

Hopper: is a Cray XE6 MPP at NERSC. Each node is in effect four 6-core Opteron chips each with two DDR3-1333 memory controllers.There are thus four (non-uniform memory access) NUMA nodes per compute nodes. Each superscalar out-of-order core implements the 2-way SSE3 SIMD instruction set and includes both a 64KB L1 and a 512KB L2 cache, while each socket includes a 6MB L3 cache with 1MB reserved for the probe filter. The relatively small last-level cache makes capturing locality difficult on fine grid operations.

B. GSRB Demonstration and Analysis

To provide a base case, we first ran the 256^3 problem using a GSRB relaxation (from Figure 3) with one process per NUMA node on Edison and Hopper. Figure 5 presents tuned performance for both the smooth operation (top) and the overall multigrid solve (bottom). Performance (speedup) has been normalized to the baseline implementation on either Hopper or Edison. Moreover, we separate the time spent in each level of the V-cycle and analyze them individually.

By fusing the operators in smooth, the compiler yields dramatic speedups in smooth time on the finer grids. However, the benefit degrades as one descends through the V-cycle. This effect is caused by the fact that in the baseline implementation, the working set of each triply-nested loop within smooth exceeds cache capacity for the fine grids. As a result, to construct the Helmholtz, the Laplacian must be refetched from DRAM. Eventually this working set becomes small enough that it fits in the last-level cache at which point the disparity between on chip-bandwidth and compute capability limits the performance benefit.

Even more impressive, our compiler may now insert additional ghosts zones and generate a wavefront-based update for the smooth operation. Once again, on the finer (large) grids, the performance benefit arises from trading DRAM accesses for cache accesses. We generally see a roughly 66% increase in smooth performance on the finer grids except for Edison where the benefit is roughly 40% for the finest level. The benefit is

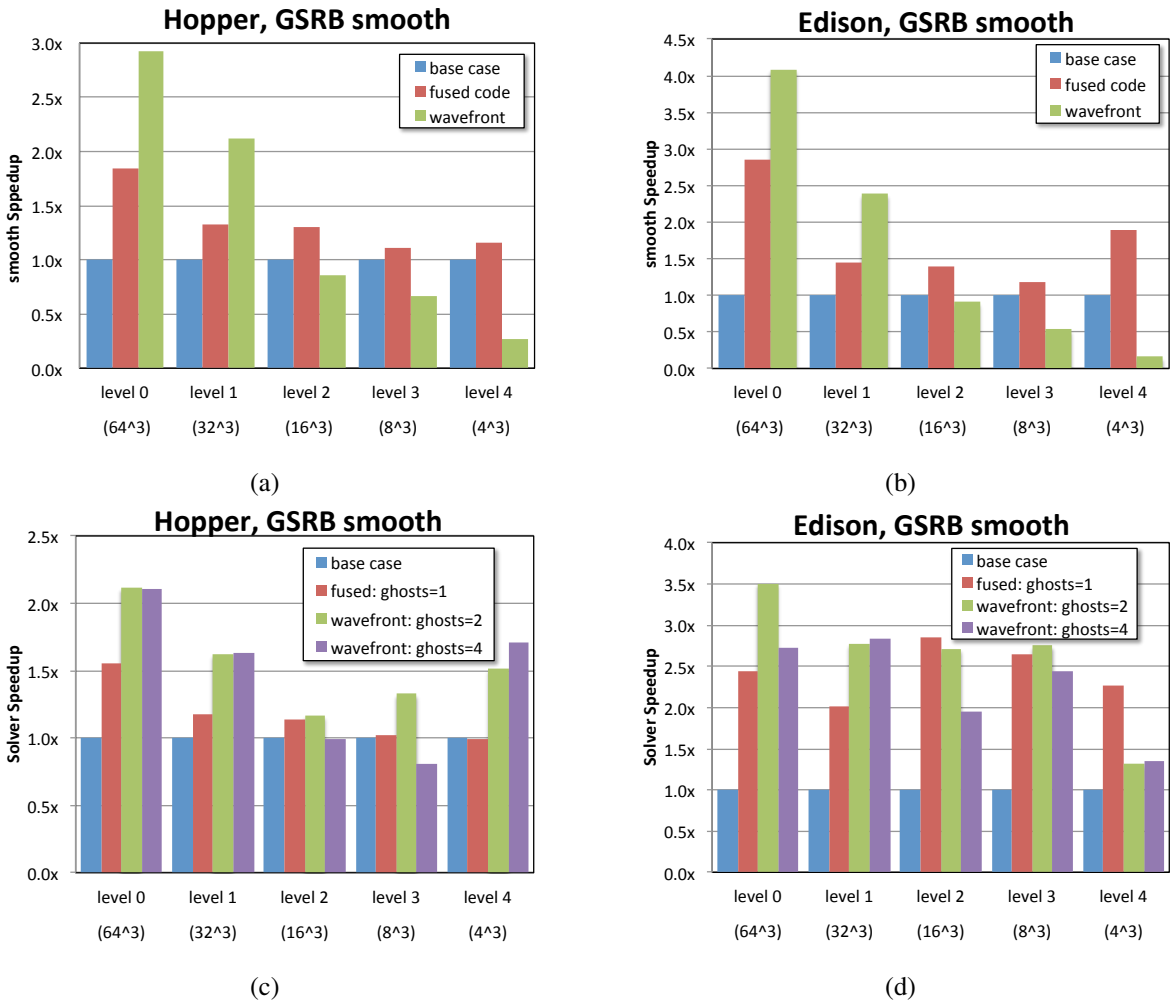

Fig. 5. Speedups relative to the baseline code with fusion and tuned communication-avoiding and threading as a function of level in the V-cycle for the 256^3 problem for smooth time (a,b) and overall multigrid solve time (c,d).

likely smaller on Edison as its massive 20 MB cache is nearly capable of capturing the locality required on the finest grids.

The local computation (smooth) for the wavefront performs poorly on the coarsest grids as the size of those grids has ballooned from approximately 12KB per box to nearly 100KB. The increase in data movement impedes performance. Figures 5(c) and (d) show the overall time required for the multigrid solve. Although smooth may be slower on the coarse grids, the reduction in inter-box communication can actually accelerate the multigrid operations. We see this benefit applies to all levels of the V-cycle. However, where as on the coarsest grids, the benefit of reduced inter-box communication compensates for reduced smooth performance, the additional cost of communication actually impedes smooth performance on the fine grids. This presents an interesting tuning space for our compiler infrastructure, which was addressed via autotuning.

To understand the salient characteristics of performance, let us examine tuning along one axis at a time. First, Figure 6 presents smooth performance as a function of fusion, wavefront, and ghost-zone depth. We restrict our presentation to the two finest grids which nominally dominate the run time. On Hopper with its relatively small caches, increasing ghost zone depth (increasing the grid size) does not improve performance when using only the fused version. However, the wavefront technique does reduce the cache working set and allows for a significant speedup. Conversely, on Edison and its large caches, increasing ghost zone depth allows for more on-chip locality and thus improved performance. Our compiler framework does not unroll the inner-loop, nor does it generate explicitly SIMDized code as yet. Thus the benefit of increasing the ghost zone depth to 4, which increases computation, is currently limited by the code generation capabilities of the back-end compiler.

Similarly, Figure 7 shows how performance varies as a function of threading. The compiler generates a nested OpenMP region to deal with inter- and intra-box parallelism. Thus on our evaluated platforms using today's technology, it generally seems to be best to maximize inter-box parallelism within the wavefront.

C. Jacobi Demonstration

To this point we have focused on GSRB as a relaxation scheme as it is typical of real applications. However, for completeness, we also demonstrate that CHiLL can optimize Jacobi relaxations in an analogous way. Figure 8 presents progressively optimized smooth performance using the Jacobi relaxation normalized to the base case for each level in the

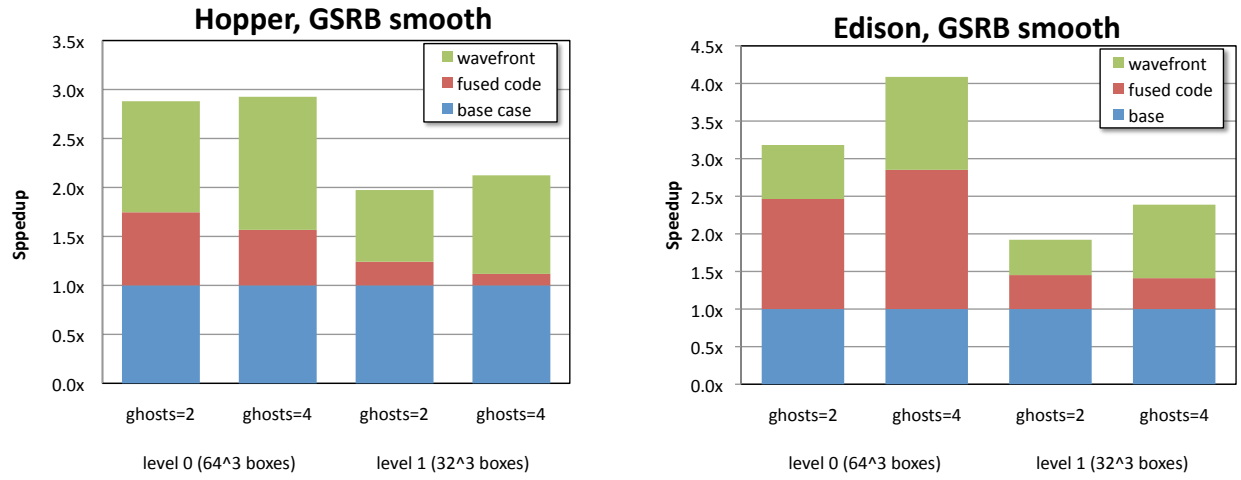

Fig. 6. Speedup for smooth as a function of optimization and ghost-zone depth (with tuned threading) for levels 0 and 1 of the V-cycle for the 256^3 solve.

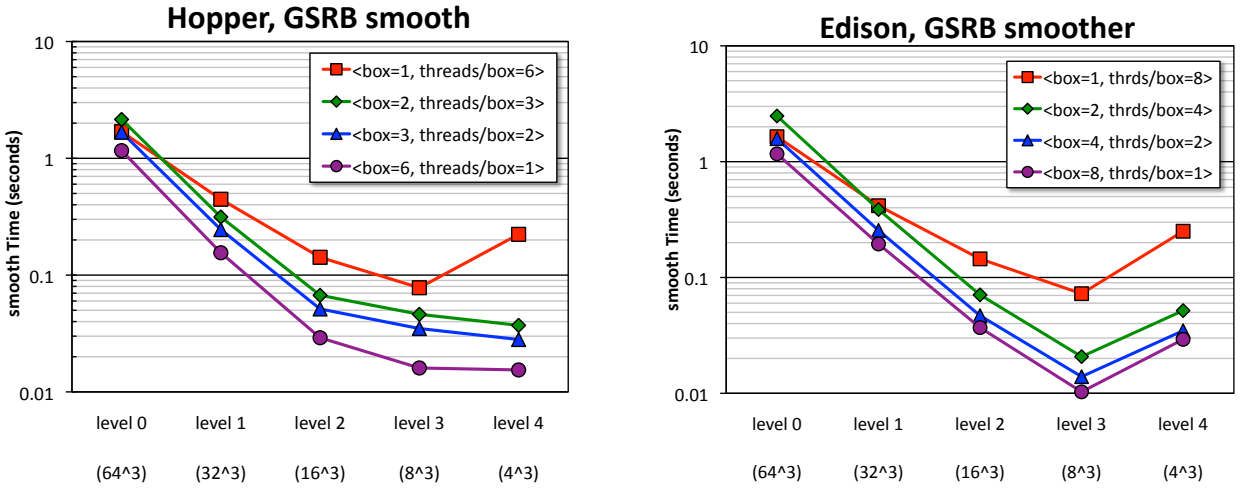

Fig. 7. Total time spent in smooth as a function of nested parallelism for the V-cycle with the optimized wavefront variant of GSRB smooth with ghost zone=4 for the 256^3 problem.

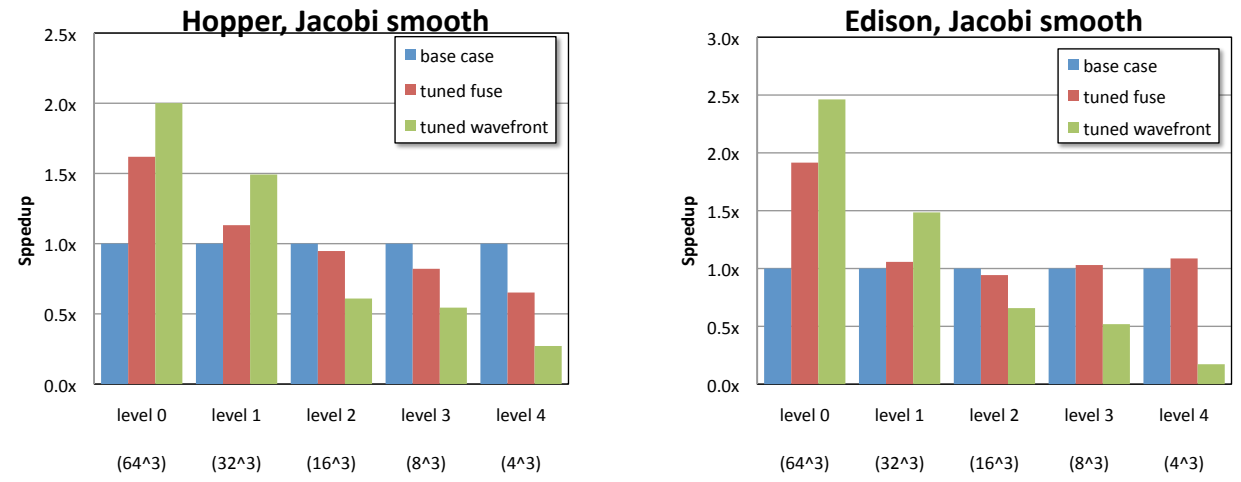

Fig. 8. Smooth speedup relative to the baseline code with tuned fusion and tuned communication-avoiding as a function of level in the V-cycle for the 256^3 problem with Jacobi.

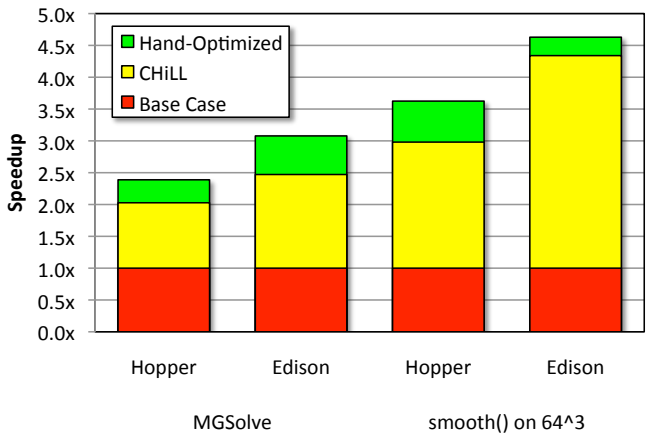

Fig. 9. Overall speedup for both the multigrid solver and the smooth operation on the finest grid compared to baseline and hand-optimized code.

V-cycle. Our Jacobi implementation alternates writes between the correction and a temporary array. CHiLL can restructure this computation to fuse the three operations (Laplacian, Helmholtz, Jacobi) and eliminate superfluous data movement. Nevertheless, without a cache bypass operation, this will nominally incur additional data movement over GSRB.

D. Overall Speedup

Figure 9 summarizes the overall speedup CHiLL attains over the baseline optimized by the icc compiler, and compares with the hand-tuned code of Williams et al. [30]. As the focus of this paper was the time-dominating smooth operation, we see the biggest gains there: speedups of 2.9× and 4.1× over the baseline, and within 82% and 94% of hand-tuned performance on Hopper and Edison, respectively. As smooth is only one component in a multigrid solver, the benefits are amortized in the full GMG application. Nevertheless, our framework attains nearly a 2.5× increase in overall solver performance on Edison.

VI. 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 [9], [10], [17], [21], [22], [29], [31]. However, a number of factors have made such approaches progressively obsolete on modern platforms. On-chip caches have grown by orders of magnitude and are increasingly able to capture sufficient locality for the fixed box sizes associated with typical MG methods. The rapid increase in on-chip parallelism has also quickly out-stripped available DRAM bandwidth resulting in bandwidth-bound performance.

Thus, 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 [8], [12], [13], [19], [20], [23], [28], [32], [35], [36], [38]. Many of these efforts examined 2D or constant-coefficient problems — features rarely seen in real-world applications.

Chan et al. explored how, using an autotuned approach, one could restructure the MG V-cycle to improve time-to-solution in the context of a 2D, constant-coefficient Laplacian [4]. This approach is orthogonal to our implemented optimizations and their technique could be incorporated in future work.

Studies have explored the performance of algebraic multigrid on GPUs [2], [3], while Sturmer et al. examined geometric multigrid [24]. Closely related work from Treibig implements a 2D GSRB on SIMD architectures by separating and reordering the red and black elements [27], additionally a 3D multigrid on an IA-64 (Itanium) is implemented via temporal blocking.

The most closely related work are domain-specific compilers for parallel code generation from a stylized stencil specification [7], [25], [37] or from a code excerpt [15]. Pochoir uses a cache oblivious strategy, which limits the control over the code generation [25]. The other compilers introduce parallelism and ghost regions through tiling and expanding both the data set and the tile size, rather than starting with already parallel code [15], [37]. These tiling approaches do not produce the hyper-trapezoidal loop nests of this paper, but rather compute and then ignore some incorrect results. None of these approaches appear capable of supporting the optimization of a collection of operators, particularly if GSRB is included.

Our work expands on all of these efforts by providing a unique set of optimization strategies for multi- and manycore architectures.

VII. Conclusion

In this paper, we have described autotuning compiler technology to automate communication-avoiding optimizations for the smooth operator in a geometric multigrid computation. Our results show that the optimizations lead to speedups as high as 4×, and that different optimization strategies are needed at different levels of the V-cycle. Gains also vary for the two smooth operators GSRB and Jacobi, and for the two different architectures, thus pointing to the value of autotuning in finding the best set of optimizations for a given execution context. As compared with related domain-specific compiler research, our work is distinguished by focusing on the needs of scalable GMG applications, starting with a parallel code and considering the composition of smooth operators including the complex Gauss-Seidel Red-Black operator. As compared with manually-tuned codes, the automatically-generated code captures the communication-avoiding optimizations, while attaining nearly equivalent performance. Future work on miniGMG will focus on further performance enhancements by generating architecture-specific code for the increasingly wide SIMD units such as AVX, and the addition of prefetching. Our long term goal is to develop an optimization framework that generalizes to a variety of GMG operators used in important scientific simulations. Of particular interest are higher order stencils and different boundary conditions.

Acknowledgments

Authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH-11231. Utah researchers were partially supported by DOE award DE-SC0008682 and National Science Foundation award CCF-1018881. 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 No. DE-AC02-05CH11231.

REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kauffman, 2002.

[2] N. Bell, S. Dalton, and L. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, June 2011.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[4] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. Autotuning multigrid with petabricks. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, Nov. 2009.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[12] 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.

[13] P. Ghysels, P. Kosiewicz, 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.

[14] M. W. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *Proc. of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, Oct 2009.

[15] 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.

[16] 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.

[17] M. Kowarschik and C. Wei. 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] M. Sturmer, H. Kostler, and U. Rude. How to optimize geometric multigrid methods on GPUS. In *Proc. of the 15th Copper Mountain Conference on Multigrid Methods*, Copper Mountain, CO, March, 2011.

[25] 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.

[26] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *International Parallel and Distributed Processing Symposium*, Apr. 2009.

[27] J. Treibig. *Efficiency improvements of iterative numerical algorithms on modern architectures*. PhD thesis, University Erlangen, 2008.

[28] 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.

[29] 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.

[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 autotuning. 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] S. Williams, A. Watterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, April 2009.

[34] M. Wolfe. Loop skewing: the wavefront method revisited. *Int. J. Parallel Program.*, 15(4):279–293, Oct. 1986.

[35] 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.

[36] 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.

[37] 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.

[38] 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.