Compiler-Directed Transformation
for Higher-Order Stencils

Protonu Basu,
Mary Hall
School of Computing, University of Utah
protonu@cs.utah.edu

Samuel Williams, Brian Van Straalen,
Leonid Oliker, Phillip Colella
Lawrence Berkeley National Laboratory
SWWilliams@lbl.gov

Abstract—As the cost of data movement increasingly dominates performance, developers of finite-volume and finite-difference solutions for partial differential equations (PDEs) are exploring novel higher-order stencils that increase numerical accuracy and computational intensity. This paper describes a new compiler reordering transformation applied to stencil operators that performs partial sums in buffers, and reuses the partial sums in computing multiple results. This optimization has multiple effects on improving stencil performance that are particularly important to higher-order stencils: exploits data reuse, reduces floating-point operations, and exposes efficient SIMD parallelism to backend compilers. We study the benefit of this optimization in the context of Geometric Multigrid (GMG), a widely used method to solve PDEs, using four different Jacobi smoothers built from 7-, 13-, 27- and 125-point stencils. We quantify performance, speedup, and numerical accuracy, and use the Roofline model to qualify our results. Ultimately, we obtain over $4\times$ speedup on the smoothers themselves and up to a $3\times$ speedup on the multigrid solver. Finally, we demonstrate that high-order multigrid solvers have the potential of reducing total data movement and energy by several orders of magnitude.

Keywords—Compiler Optimization; Stencil; High-Order; Multigrid; Mehrstellen;

I. INTRODUCTION

Finite-Volume/Finite-Difference solutions for partial differential equations (PDE) have, at their base, computations of *stencils*. A stencil is a linear transformation of the form $L(a)_{\mathbf{i}} = \sum_{\mathbf{j}} \alpha_{\mathbf{j}} a_{\mathbf{i}+\mathbf{j}}$. Here $\mathbf{i}, \mathbf{j}$ are integer tuples, a is a multidimensional array of floating-point data that approximates on a rectangular grid the solution to some differential equation, and $L(a)$ approximates the application of a differential operator to a function evaluated on a rectangular grid. The error in the approximation is proportional to some integer power p of the grid spacing (p is referred to as the *order of accuracy* of the discretization). Most prior work on optimizing stencil computations focus on lower-order methods (typically $p = 2$) where there is limited reuse of data and computations are notoriously memory bound. Much of this prior work reduces the amount of data movement by fusing multiple stencil sweeps through techniques like cache oblivious, time skewing, wavefront or overlapped tiling [1–12].

For a given partial differential operator, the number of points required in a stencil typically increases as a polynomial in the order of accuracy. This is in contrast to the exponential dependence on p in the reduction of the error. Furthermore, larger stencils, while requiring larger numbers of floating-point operations, can be organized to require a comparable degree of main memory data movement as their lower-order coun-

terparts. Thus, with processor architectures becoming more compute-intensive [13], high-order schemes are increasingly important as they can achieve greater accuracy with less data movement. This property, combined with the need for computational scientists to minimize the memory capacity required to obtain a given level of error in their simulations, have been motivating the effort to increase the order of accuracy of stencil-based algorithms for PDE.

In this paper, we introduce and evaluate a novel compiler transformation which implements *array common subexpression elimination* [14] by recognizing and exploiting the high degree of symmetry in stencil coefficients and generating code that exposes efficient SIMD parallelism to backend compilers. By recognizing that stencil computations are associative and can be reordered, array common subexpression elimination identifies common floating point operations across loop iterations and reuses these calculations. In this paper, we decompose the three-dimensional stencils to a number of two-dimensional planes, compute *partial sums* of the elements, and then buffer these partial sums across iterations to derive the output values. This optimization exploits data reuse, thus improving memory hierarchy performance, while also eliminating redundant floating-point computation. Reducing floating-point computation is particularly valuable for large higher-order stencils, as they can be compute bound, and their computation can also stress the register capacity [15].

To investigate the efficacy of this approach, we apply it to various discretizations of Poisson's equation, with orders of accuracy p ranging from 2 to 10. Numerical solutions to Poisson's equation are ubiquitous in simulations of a variety of physical problems including fluid dynamics, astrophysics, electromagnetics, and plasma physics. In addition, the stencils used here are good proxies for higher-order stencil operators from a broad range of PDE problems arising in computational science. We evaluate our approach in the context both of applying the operators in a stand-alone fashion, and within a Geometric Multigrid (GMG) method for solving Poisson's equation using these discretizations. We combine partial sums with the communication-avoiding and thread scheduling optimizations of prior work [16, 17]. Our compiler's ability to compose transformations allows it to take a higher-order smoother, remove its computation bottleneck and make it bandwidth-limited, enabaling it to then further apply bandwidth-reducing optimizations. Overall, this paper makes the following contributions:

- We describe the partial sum optimization within the CHiLL compiler [18], which goes beyond related manual [19–

21] and compiler optimizations [14, 15] by simultaneously addressing DRAM and cache bandwidth while reducing floating-point computation and facilitating SIMDization.

- Overall we demonstrate up to a $3\times$ speedup for the entire multigrid solver. Our study is in contrast to much of the prior work in this area, which has been dominated by looking at the stencil operator in isolation.
- We demonstrate our 10th order finite-volume method delivers roughly a $1000\times$ increase in accuracy for every $8\times$ increase in memory vs. $32{,}768\times$ increase in memory for the 2nd order (requisite memory $\approx$ error$^{-3/order}$) with only a factor of two difference in stencil performance.

II. Stencil Definitions and Accuracy

We will consider a collection of test problems for stencil calculations all of which are discretizations of Poisson's equation $\Delta(\phi) = f$, where $\Delta\phi \equiv \sum_{i=0}^{d} \frac{\partial^2 \phi}{\partial x_i^2}$, Here f is some given function, and we want to solve for ϕ. For this study, we will assume that ϕ and f are periodic functions on $[0,1]^d$, i.e. $\phi(\mathbf{x}) = \phi(\mathbf{x} + \mathbf{q})$ for all integer tuples $\mathbf{q}$, and similarly for f. We compute approximate solutions on a rectangular lattice $\phi_\mathbf{i}^h \approx \phi(\mathbf{i}h)$ by solving the stencil equations (superscripts denote grid spacing) $L^h \phi^h = f^h$, where $f^h = f(\mathbf{i}h)$, $h = 1/N$ for some integer N. In that case, the periodicity of the exact solution translates into periodicity on the lattice: $\phi_{\mathbf{i}+\mathbf{q}1}^h = \phi_\mathbf{i}^h$, and similarly for f^h. Thus, we will evaluate our operator and solve our equations on the grid $[0, N-1]^d$, and use periodicity to evaluate stencil dependences that are not on that grid.

In this work, we examine the four representative stencils operators shown in Figure 1. Points with the same color have the same value for the coefficient. Stencils with the high degree of symmetry shown here are a consequence of the use of centered-difference approximations on rectangular grids, which is ubiquitous in discretizations of Poisson's equation and other constant-coefficient, elliptic operators on such grids. The standard approximation for explicit integrations use either the well known 7-point [22] or 13-point stencils [23], with second- and fourth-order accuracy, respectively. The sixth-order (27-point) and tenth-order (125-point stencils) in our study are what are known as Mehrstellen stencils. They achieve their stated accuracy only if the right-hand side is pre-processed. Rather than solving $L^h \phi^h = f^h$, one solves $L^h \phi^h = M f^h$, where M is a stencil operator whose coefficients sum to 1. This is a one-time operation applied to the right-hand side, prior to applying whatever solution algorithm that is used (e.g. geometric multigrid (GMG) as is used here), and hence does not have significant impact on the performance of the solver. The 27-point stencil and the associated Mehrstellen correction stencil are classical, and can be found in [22]. The 125-point stencil and its associated Mehrstellen correction stencil are new, and will be published elsewhere [24]. For the purposes of this paper, it is only necessary to know the symmetries of the operator, which are summarized in Figure 1. The accuracy claims for this method will be verified in Section V.

III. Stencil Reordering: Partial Sums

For almost all stencils, there is data and operation reuse between neighboring points. This reuse is more significant for higher-order stencils, which examine many more neighboring
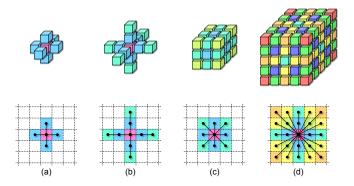

Fig. 1. (top) Visualizations of the discretized 3D Laplacian operators (stencils) used in this paper. (bottom) 2D cross sections through the centers of 3D stencils. Color is used to denote the coefficient associated with that point. The 27- and 125-point stencils have complex symmetries that we exploit.

input points to compute each output point. In this paper, we develop a compiler transformation that exploits this reuse to reduce loads, while removing floating-point operations that are redundant across multiple output calculations. Thus, we improve both computation and memory costs of the higher-order stencils. The transformation recognizes that stencil computation can be reordered, and therefore computes and reuses partial results in *partial sums*.

In stencil computations, *out-of-place* updates are loop nest computations where the right-hand sides are read-only arrays per stencil sweep (e.g., Jacobi). *In-place* stencil computations read and write the same array each sweep (e.g., Gauss-Seidel). Stencils may be the sum of pairwise products of two or more arrays (*variable-coefficient*) or a weighted sum of a single array (*constant-coefficient*). The partial sum transformation described in this paper targets constant-coefficient, out-of-place stencils. This section illustrates the partial sum transformation, while details of the compiler implementation are presented in Section IV, and performance impact as well as interaction with other optimizations is described in Section V.

For an illustration of computing stencils via partial sums, consider the 9-point 2D stencil of Figure 2(top). (This is the 2D analog of the 27-point 3D stencil of Figure 1(c).) When calculating out[j][i], inputs in[j][i+1], in[j-1][i+1] and in[j+1][i+1] are reused in the next two iterations of the i loop. If we conceptualize the stencil as a box as shown in the diagram of Figure 2(bottom), these points are the *right edge* for iteration $\langle j, i \rangle$, the *center* for iteration $\langle j, i+1 \rangle$ and the *left edge* for iteration $\langle j, i+2 \rangle$. Therefore, we employ an optimization that avoids loading all nine inputs, but instead loads only the right edge while reusing data from the previous two iterations of the i loop.

At iteration $\langle j, i \rangle$ the right edge, points in[j][i+1], in[j-1][i+1] and in[j+1][i+1] are loaded. We capture the contribution these points make to the outputs at $\langle j, i \rangle$, $\langle j, i+1 \rangle$ and $\langle j, i+2 \rangle$ by calculating the weighted sum of the loaded edge with coefficients corresponding to the right, left and center edge, as illustrated in Figure 3. The compiler constructs an array of coefficients to be used in the partial sum transformation. If we visualize the array of coefficients as a box as in Figure 3, with its entries corresponding to coefficients of the stencil, then the weighted sum of input array points with the
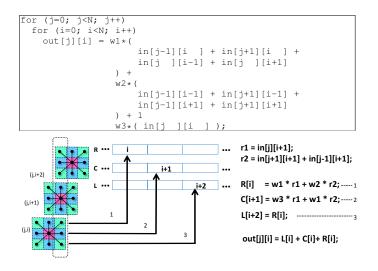
```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    out[j][i] = w1*(
                    in[j-1][i  ] + in[j+1][i  ] +
                    in[j  ][i-1] + in[j  ][i+1]
                ) +
                w2*(
                    in[j-1][i-1] + in[j+1][i-1] +
                    in[j-1][i+1] + in[j+1][i+1]
                ) + 1
                w3*( in[j  ][i  ] );
```


Fig. 2.   The input code for a 2D 9-point stencil (top). Partial sums are buffered and then summed to produce the output (bottom).

right edge of the array of coefficients computes B0, the center computes B1 and the left edge computes B2. B0 is used to compute the output at $\langle j,i\rangle$, while B1 and B2 are buffered for outputs at the next two iterations of the i loop.

As mentioned earlier, symmetries in stencil coefficients are ubiquitous, and we exploit symmetry to reduce floating-point operations in computing B0, B1 and B2. The colors in the array of coefficients in Figure 3 represent symmetry, with each color representing a unique coefficient. The code implementing partial sums in Figure 2 can now be explained using Figure 3. Symmetry about the $j-$axis means B0 and B2 (entries for R[i] and L[i+2]) are equal and do not have to be recomputed. Symmetry about the $i-$axis means `in[j-1][i+1]` and `in[j+1][i+1]` will always be multiplied by the same coefficient, and thus we store their sum in $r2$; `in[j][i+1]` is loaded to $r1$. B0 is the weighted sum of $r1$ and $r2$ with coefficients $w1$ (blue) and $w2$ (green), respectively, and B1 with coefficients $w3$ (red) and $w2$ (green).

In a similar manner, for 3D stencils we compute partial sums of 2D planes instead of 1D lines. Figure 4 shows a 2D cross-section of a 3D stencil, and the symmetries present. Figure 5 shows simplified generated code for the 3D 27-point stencil. Three buffers are allocated, for the left, center and right planes; and three scalars are created for three unique coefficients in each 2D plane. In the general case, the number of buffers required is twice the *radius* of the stencil plus one (radius is 1 for 7- and 27-point stencils, 2 for 13- and 125-point stencils).

While a reference implementation of the 9-point stencil necessitates 11 floating-point operations (8 adds, 3 multiplies) to compute each output point, our approach requires only 9 floating-point operations (3 adds and 4 multiplies for the partial sums plus 2 adds to sum the symmetric partial results). The reduction in floating-point operations becomes more significant with the size and dimension of the stencil operator; for example, the 125-point stencil has 124 adds, but once optimized it has only 38, for a more than $3\times$ reduction.

An additional advantage of this approach is that it results in code amenable to SIMD code generation, AVX and SSE for

our target architectures. Intuitively, SIMDization is enabled because the calculations in the loop, including the partial sum calculations, have unit stride across the innermost i dimension. While we are unable to isolate the SIMDization benefits, we performed an experiment to illustrate the differences between the 125-point stencil code before and after the partial sum optimization. Using Intel Architecture Code Analyzer (IACA), we profiled both versions of the code on an Intel Westmere i5-540M platform.[1] In the partial-sum-optimized code, all floating-point adds use SIMD instructions, whereas only about two-thirds of the adds in the baseline code use SIMD instructions. However, L1 and shuffle bandwidth can limit the ultimate benefit from increased SIMDization.

Comparison with Array Common Subexpression Elimination. A compiler formulation of a related reordering transformation called array common subexpression elimination was described in [14]. Array common subexpression elimination is implemented using an abstraction called a *tablet*, which records the structure of the stencil inputs and their coefficients. To capture reuse, redundancy conditions, heuristics and benefit functions are used to generate *subtablets* of the tablet. The subtablets are used to compute partial sums which are reused via scalar temporary variables. This method of exploiting reuse through the subtablets is more complex but more general than the partial sums method. The subtablet construction allows for reuse of points other than a plane and enables handling of multi-statement stencils. However, exploiting reuse in scalar registers introduces a scalar dependence across loop iterations and inhibits instruction-level and SIMD parallelization by native backend compilers.

In contrast, the partial sums approach always picks the *leading plane* of points, and it buffers the computed partial results in vectors. Picking the leading plane and explicitly looking for symmetry to reduce redundant computation is simpler than using heuristics, redundancy conditions and cost functions to discover reuse and symmetry from subtablets. Further, partial sums avoids introducing dependences and generates code easily vectorized by the native backend compiler. The resulting code no longer has cross-iteration reuse in the innermost loop and thus avoids the *data stream alignment* problem which results in inefficient SIMD code, as described in [25].

Interplay with communication-avoiding optimizations. The partial sum optimization reduces floating-point operations in compute-bound kernels to make them more memory bound. Once memory bound, communication-avoiding optimizations can be used to further improve performance, such as overlapped tiling (via larger ghost zones), loop fusion and wavefronts [8, 16, 17, 26]. Overlapped tiling reduces inter-processor communication. Loop fusion and wavefront computation reduce communication to DRAM by fusing multiple grid sweeps into one. Wavefronts are generated by loop skewing followed by loop permutation. Skewing breaks data dependences allowing permutation, and, the loop skew factor must increase with stencil radius. Wavefront exploits reuse but increases the working set, which may result in spills from faster caches. CHiLL can generate nested parallel OpenMP code to reduce the working set per thread. As will be empirically demonstrated in Section V, if used together, partial sums and communication-

---

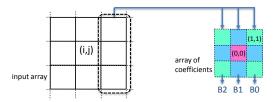[1]IACA does not run on our target machines.

Fig. 3. Illustration of deriving coefficients for partial sum. The right edge from the input array is loaded and multiplied by weights stored in the array of coefficients. The sum of products of the loaded points and the right, center and left edges of the array of coefficients are B0, B1 and B2, respectively.
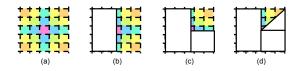

Fig. 4. Increasing symmetries in coefficients allows us to increasingly reduce floating-point computation. Symmetry about the j−axis (in b), permits discarding half the coefficients, and symmetry about the i−axis (c) and the diagonal (d) lets the compiler consider even fewer coefficients.

```
int radius = 1; // distance of farthest stencil point
                // from origin per dimension
// allocate 2*radius+1 buffered partial sums,
// N is grid (box) dimension
// create (radius+1)*(radius+2)/2 temporaries
double B0[N], B1[N], B2[N];
double r1, r2, r3;

for (k=0; k<N; k++){
  for (j=0; j<N; j++){
    // preamble code sets up the pipeline
    ....
    // steady state computation
    for(i=0; i<(N-radius); i++){
      r1 = phi[k][j][i+1];
      r2 = phi[k+1][j  ][i+1] + phi[k-1][j  ][i+1] +
           phi[k  ][j-1][i+1] + phi[k  ][j+1][i+1];
      r3 = phi[k+1][j+1][i+1] + phi[k+1][j-1][i+1]+
           phi[k-1][j+1][i+1] + phi[k-1][j+1][i+1];
      B2[i]   = w1*r1 + w2*r2 + w3*r3;
      B1[i+1] = w2*r1 + w3*r2 + w4*r3;
      B0[i+2] = B2[i];
    }
    for(i=0; i<(N-radius); i++)
      phi_new[k][j][i] = B0[i] + B1[i] + B2[i];
  ...
  // cleanup code to avoid extra computation
  ...
  }
}
```

Fig. 5. Output code for 3D 27-point stencil, optimized using partial sums.

avoiding optimizations can achieve substantial performance gains for high-order stencils.

IV. COMPILER IMPLEMENTATION

The partial sum transformation described in the previous section has been implemented in the CHiLL polyhedral transformation and code generation framework [18] and extends previous communication-avoiding optimizations in CHiLL targeting GMG [16, 17]. Building new transformations into a polyhedral framework easily allows for composition of transformations as long as data dependences are not violated. We compose partial sums with loop transformations to target both computation and communication bottlenecks.

The input to CHiLL is a source code written in C or Fortran

and a *transformation recipe* describing the set of transformations to be composed to optimize the provided source [27]. This recipe can be written by an expert programmer, or derived automatically by a compiler decision algorithm. A new script command for the partial sum transformation has been implemented in CHiLL, which identifies the stencil statement to which this transformation should be applied. In this paper, which explores the impact of different optimization strategies, the transformation recipes were manually written, extending the recipes for communication-avoiding optimization described in [16] to also invoke the partial sum transformation. This section describes the abstractions used by the compiler in the automation of the partial sum transformation.

We make the following assumptions about the input code to our framework. 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. The partial sum optimization requires that the subscript expressions are *separable*, such that each dimension references just a single loop index. Partial Sum applies to out-of-place stencils. Out-of-place updates are loop nest computations where the right-hand sides are read-only matrices per stencil sweep (e.g. Jacobi). Currently we are limited to constant coefficient out-of-place stencils.

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

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

By convention, l_3 is the innermost loop of a 3D loop nest. It is standard to normalize iteration spaces to start at 0. Bounds constraints can be far more complex, but for simplicity of explanation, 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 [28]. The array index expressions are mapped to the new iteration space. In the following we will describe the partial sum transformation, which modifies both iteration spaces and statements.

Abstractions for Partial Sums. Examining the statement associated with a stencil computation, the compiler builds four abstractions to perform the partial sum optimization: (1) *StencilPoints* refers to the set of points that comprise the stencil, offset from a specific iteration in the 3D iteration space; (2) *BB* is the axis-aligned bounding box of *StencilPoints*, such that each dimension derives its lower and upper bound from the minimum and maximum values in that dimension; (3) *Coeff* is a 3D array the same size as *BB* to hold the coefficients for the points in the stencil; and, (4) *Buffer* is a set of arrays that are used to hold the partial sums in the generated code. This subsection describes how these abstractions are derived automatically by the compiler and used by the code generator

to produce code analogous to the example in Figure 5. For simplicity, we assume a unit stride for the loop iteration spaces, but extensions for non-unit-stride loops are straightforward.

Deriving StencilPoints and BoundingBox (BB). At every iteration $\vec{I} = \langle i_1, i_2, i_3 \rangle \in IS$, we compute a single output of the stencil. We can rewrite an out-of-place, constant-coefficient p-point stencil as a weighted sum of p points, with w_m representing the coefficient for point m. A vector offset from iteration $\vec{I}$ for each point m is then $\vec{O}_m = \langle o_1^m, o_2^m, o_3^m \rangle$. The compiler computes BB from lower and upper bounds for each dimension of these offsets (i.e., $lb_1 = min_m o_1^m$, $ub_1 = max_m o_1^m$, ...). This notation gives rise to the following definitions:

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

$$StencilPoints = \bigcup_{m=1}^{p} \vec{O}_m$$

$$BB = \{[b_1, b_2, b_3] : \quad lb_1 \le b_1 \le ub_1, lb_2 \le b_2 \le ub_2, \\ lb_3 \le b_3 \le ub_3\}$$

Deriving Coefficients (Coeff). If *StencilPoints* and *BB* represent the identical volume, as in the 27-point stencil, we call this a *full* stencil. The star-shaped 7-point stencil operators of Figure 1(a) are not full. A set difference mathematically determines the holes in the stencil.

The compiler creates an array *Coeff*, the same size as *BB*, to store the coefficients for the partial sum. For simplicity of explanation, we will assume *Coeff* is centered at $\langle 0, 0, 0 \rangle$ and allows negative indices. Points $\vec{B} = \langle b_1, b_2, b_3 \rangle \in BB$ which belong to *StencilHoles* are set to zero in the array of coefficients, and others are assigned appropriate constant values. We can then rewrite the stencil computation at each output point $\vec{I}$, using the following equations:

$$StencilHoles = BB - StencilPoints$$

$$Coeff[b_1][b_2][b_3] = \begin{cases} 0 & : \quad (\vec{B}) \in StencilHoles \\ w_m & : \quad (\vec{B} = \vec{O}_m) \in StencilPoints \end{cases}$$

$$out[i_1][i_2][i_3] = \sum_{\vec{B} \in BB} Coeff[b_1][b_2][b_3] * in[i_1 + b_1][i_2 + b_2][i_3 + b_3]$$

Deriving Partial Sums and Buffers. We form partial sums for subsets of *BB*, planes for a 3D stencil or similarly, lines for a 2D stencil as in Figure 2. *Plane$_k$* is the p_1, p_2 plane inside the *BB* at $p_3 = k$. The *BB* can be partitioned into $(ub_3 - lb_3 + 1)$ such planes (corresponding to the stencil radius plus 1); there is a corresponding plane in *Coeff* such that points in *BB* are array indices for elements in *Coeff*. We can compute an output point $\vec{I}$ as a sum of partial sums PS_k at each plane $k \in BB$, which is staged in a buffer. There are $(ub_3 - lb_3 + 1)$ buffers, each as wide as the trip count of the inner loop. Within the innermost loop, the values in the buffers are summed to compute the output. These rewrites of the stencil are captured as follows:

$$Plane_k = \{[p_1, p_2, p_3] : lb_1, lb_2 \le p_1, p_2 \le ub_1, ub_2; p_3 = k\}$$

$$PS_k(\vec{I}) = \sum_{\vec{P} \in Plane_k} Coeff[p1][p2][p3] * \\ in[i_1 + p1][i_2 + p2][i_3 + p3]$$

$$Buffer_k[i_3] = PS_k(\vec{I})$$

$$out[i_1][i_2][i_3] = \sum_{k=lb_3}^{ub_3} Buffer_k[i_3]$$

Exploiting Reuse in Partial Sums. The optimizations derived from partial sums recognize that loads associated with a plane have reuse in the third dimension. At iteration $\vec{I}$, the rightmost plane *Plane$_r$*$(\vec{I})$ where $r = ub_3$ is of particular importance. This is the leading plane in the bounding box of the stencil (in the direction of increasing third dimension), and it corresponds to the right edge of the 2D stencil described in section III. We load only this plane, and reuse it for other output points. Let $\vec{I}^k = \langle i_1, i_2, i_3 + k \rangle \in IS, 0 \le k \le (ub_3 - lb_3)$. *Plane$_r$*$(\vec{I})$ is the same set of points as *Plane$_{r-1}$*$(\vec{I}^1)$, and *Plane$_{r-k}$*$(\vec{I}^k)$ in general. *Plane$_r$*$(\vec{I})$ can thus be used to compute $(ub_3 - lb_3 + 1)$ partial sums. The partial sums are computed by using points from *Plane$_r$*$(\vec{I})$ and sweeping through the planes of *Coeff*; the results are buffered in *Buffer$_{r-k}$*$[i_3 + k]$, for the range of values of k. *Buffer* and output at $(\vec{I})$ are computed from *PS* as in the previous step.

We can now derive the partial sums from just the load of *Plane$_r$*$(\vec{I})$. The following is computed $\forall_k, 0 \le k \le (ub_3 - lb_3)$:

$$PS_{r-k}(\vec{I}^k) = \sum_{\vec{P} \in Plane_r} Coeff[p_1][p_2][ub_3 - k] * \\ in[i_1 + p1][i_2 + p2][i_3 + p3]$$

Exploiting Symmetry to Reduce Floating-Point Operations. Figure 4 shows that several coefficients in each plane of Coeff have the same value. This means multiple points in *Plane$_r$* use the same coefficient to compute partial sums. For each unique coefficient, we sum the points in *Plane$_r$* that use it. This sum is stored, and multiplied by the appropiate weight to calculate the partial sums, and does not have to be recomputed. In our current implementation we check for symmetry in *BB* about the axes and diagonals in a plane and across planes before implementing our floating-point reducing optimization. This technique can be easily extended to work with fewer degrees of symmetry. When all the symmetries are present, *BB* is a cube such that upper and lower bounds are the same, $lb = -ub$, and the coefficients in each $\langle p1, p2 \rangle$ plane are symmetric about the $p2$ and $p3$ axes and diagonals.

In any 2D plane of *Coeff*, the coordinates of unique coefficients are defined as $UC = \{\langle p1, p2 \rangle : 0 \le p1 \le p2 \le ub\}$, corresponding to the colored octant in Figure 4(d). From reflections about the axes and diagonals, for any point $\langle p1, p2 \rangle \in UC$, the set of reflected points $Ref(p1, p2) = \{\langle \pm p1, \pm p2, ub \rangle, \langle \pm p2, \pm p1, ub \rangle\}$ in *Plane$_r$* are weighted with the same coefficient *Coeff*[p1][p2][ub-k]. The sum of the points in the input `in`, corresponding to $\langle p1, p2 \rangle$ is $R_{p_1 p_2}$. We rewrite partial sums using the factored terms to exploit symmetry. For stencils with zero valued

Allocate buffer objects:
Assume stencil statement $S0$ has iteration space *IS* defined in Eqn.(1).
Create $ub_3 - lb_3 + 1$ ($2ub+1$, when symmetrical) buffers, $Buffer_{lb_3}$, ..., $Buffer_{ub_3}$. Each buffer is an array of length N (from *IS*).
Create $(1+ub)(2+ub)/2$ scalars to hold sums $\mathbf{R_{p_1 p_2}}$ for each unique coefficient in a plane.

Insert statements to compute buffers.
Create a new compound statement $S1$ that has $(1+ub)(2+ub)/2$ statements to compute the sums $\mathbf{R_{p_1 p_2}}$.
To $S1$ append $ub_3 - lb_3 + 1$ statements to compute the buffers.
Each statement is of the form in Eqn.(2).
Since all values in *Coeff* are constant, these are copied directly into the statement (not an array reference).
Insert $S1$ lexicographically before $S0$.

Update *S0* to use buffers.
Create a new statement $S2$ that computes the output at $\vec{I}$ from the sum of all buffers.
Replace $S0$ with $S2$.

Update *IS*.
Decrease the number of iterations of *IS* to avoid going off the end of the buffers.
Create new iteration space, *IS'* = $\{[l_1, l_2, l_3] : 0 \leq l_1, l_2 < N \&\& 0 \leq l_3 < N - (ub_3 - lb_3) \}$
Peel off remaining iterations and use *S0*.

Fig. 6.   Code generation steps for partial sum transformation.

coefficients in *Coeff*, like the 7-point stencil, we take care not to generate redundant factored terms. The following is computed $\forall_k, 0 \leq k \leq (ub - lb)$:

$$R_{p_1 p_2} = \sum_{\langle x,y \rangle \in Ref(p1,p2)} in[x][y][ub]$$

$$PS_{r-k}(\vec{I^k}) = \sum_{\langle p1,p2 \rangle \in UC} Coeff[p_1][p_2][ub-k] * R_{p_1 p_2}$$

$$Buffer_{r-k}[i_3 + k] = PS_{r-k} \tag{2}$$

Code Generation. Once the compiler has performed the rewriting steps described above, it must generate the transformed code. The steps of code generation are described in detail in Figure 6. The compiler must create the buffer objects, and compute their values using the rewriting shown in Eqn.(2) directly above. It must also modify the original stencil statement to refer to the buffers rather than the input code. As we near the upper bound while iterating through the inner-loop, there will be no need to calculate all the partial sums, and we will go off the end of the allocated buffers. The figure shows the restricted iteration space, and peeling to address this. We also generate a code variant where *IS'* equals the original *IS*, which does extra computation, allocates buffers longer than the loop bound, but has no cleanup code. Both the code variants perform similarly.

	Cray XC30 (Edison)	Cray XE6 (Hopper)
CPU	Intel Xeon E5-2695v2	AMD Opteron 6172
Core Frequency	2.4 GHz	2.1 GHz
D$ per core	32+256 KB	64+512 KB
Cores per CPU	12	6
L3 Cache per CPU	30 MB	5 MB
CPUs per Node	2	4
DP GFlop/s	460.8	201.6
STREAM	88 GB/s	48 GB/s
Compiler	icc 14.0.2	icc 13.1.3

TABLE I.   **Overview of Evaluated Platforms**

V.   EXPERIMENTAL RESULTS

We now present performance results and analysis using our optimizing compiler technology. Additionally, we apply the Roofline performance model [13] to help quantify attainable performance. The Roofline Model uses bound and bottleneck analysis to represent architecture performance as a function of requisite data movement and computation. As such, it provides a nominal upper bound to attainable performance.

A. Evaluated Platforms

In this paper, we evaluate the benefits of our compiler technology using the systems detailed in Table I. On each platform, we used the installed Intel compiler 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 [29]. Each core implements the 4-way AVX SIMD instruction set and includes both a 32KB L1 and a 256B L2 cache. With a high flop:byte ratio (and exacerbated by TurboBoost), we expect this machine to be memory-limited for most operations without communication-avoiding optimizations. Note, Edison's memory was upgraded to DDR3-1866 subsequent to this paper.

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 [30]. Each core uses the 2-way SSE3 SIMD instruction set and includes both a 64KB L1 and a 512KB L2 cache. Hopper's lower machine balance may result in the 125-point operator being compute-limited.

B. Evaluation Benchmark — miniGMG

Multigrid is a fast linear solver that uses an iterative and recursive approach to solve elliptic PDEs. Each iteration of a multigrid solve requires performing a V-Cycle. As shown in Figure 7, a V-Cycle involves performing stencil operations (smooths) on progressively coarser (smaller) grids, solving a coarse grid problem, and then using that coarse-grid solution to correct the fine-grid solution. Unfortunately, as one may only apply the stencil four times during each smooth, there is limited data reuse.

This paper uses the miniGMG compact multigrid benchmark which is over 2000 lines of C and includes a dozen performance-critical routines [31]. Previous work studied some of the performance and productivity aspects of miniGMG [16, 26]. The benchmark creates a block structured 3D grid partitioned into subdomains (boxes) which are distributed among
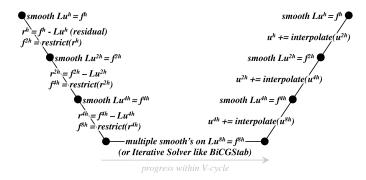
Fig. 7. The Multigrid V-cycle for solving $Lu^h = f^h$. Note, superscripts denote grid spacing.
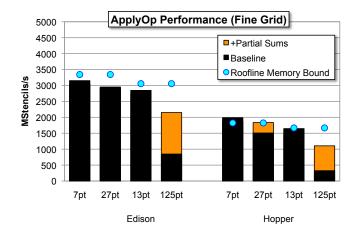

Fig. 8. ApplyOp ($y=Ax$) stencil performance attained with the CHiLL compiler by optimization, operator, and platform. Note, the Roofline memory bound is for the non-communication-avoiding implementation. Partial sums can move compute-limited operations towards a memory-limited state.

processes. In all experiments, our finest grid in miniGMG is 256^3 cells. This 256^3 grid is decomposed into disjoint 64^3 boxes (requiring ghost zone exchanges) distributed among multiple processes on one compute node. To optimize for NUMA, we run with one MPI process per NUMA node (2 per Edison node and 4 per Hopper node). miniGMG implements a multigrid V-Cycle that is terminated when each box reaches 4^3 cells. At each level, we apply four weighted Jacobi smooths using the relevant stencil shown in Figure 1. Our experiments run a fixed 10 V-Cycles with a point relaxation bottom solver that performs 48 smooths.

To highlight the generality, productivity, and effectiveness of our compiler optimizations, we decompose the smooth operations into three separate loop nests: the first applies one of the stencils to an array and writes to the temporary array, the second reads the temporary array and forms either the Poisson or Helmholtz operator, while the third performs a weighted Jacobi update of the current solution. This structure is slightly different than the manually-optimized version of the code online [31] where a few combinations of stencil and smoother were manually fused.

We construct a manufactured solution:

$$u_{true} = sin^{13}(2\pi x)sin^{13}(2\pi y)sin^{13}(2\pi z)+$$
$$sin^{13}(6\pi x)sin^{13}(6\pi y)sin^{13}(6\pi z) \text{ on } [0,1]^3$$

and symbolically apply the Laplacian to it to find a nominal right-hand side f. We then apply the appropriate Mehrstellen correction and solve $L^h u^h = M f^h$. We compare u^h and u_{true} under the l_2 norm to calculate error.

C. Stencil Performance

To highlight the memory bottleneck on modern multicore processors as well as differentiate ApplyOp ($y = Ax$) characteristics from smoother characteristics, Figure 8 presents ApplyOp (stencil in isolation) performance on the finest (256^3) grid using either the baseline implementation, or CHiLL with the Partial Sums optimization. We provide a memory bound based (blue circle) on the Roofline performance model [13] derived from the size of the arrays (including ghost zones) and the bandwidths of the target machines listed in Table I. As the compiler does not generate the `movntpd` cache-bypass instruction (verified with compiler flags), we assume 24 bytes
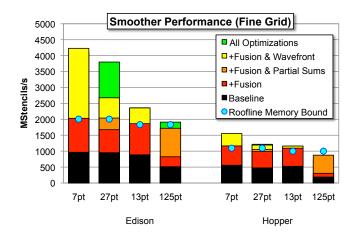
per stencil. As the 13- and 125-point operators require a 2-deep ghost zone, their Roofline bound is slightly lower.

As shown in Figure 8 baseline performance is close to the Roofline bound for most cases. The use of the partial sums optimization to eliminate unnecessary floating-point adds and regularize the computation for efficient SIMDization significantly improved performance and ensured performance came close to the Roofline bound. Note, however, that the compute-intensive 125-point stencil fell well below its Roofline bound.

D. Smoother Performance on the Fine Grid

The core operation in a multigrid solver is a smoother. It is applied multiple times in sequence on each level of the multigrid V-Cycle. As such, we may apply a wavefront or time-skewing technique to overcome the memory bandwidth limit for ApplyOp. In this paper we use a fixed 256^3 problem for all experiments. Although this ensures we may compare smoother performance when using different discretizations of the Laplacian (different stencils) it also implies that the result using a 125-point operator will be more accurate. Please note that unlike a simple stencil operation ($y = Ax$), a smoother ($x_{new} = x + wD^{-1}(b - Ax)$) requires significantly more data movement including reading arrays for the right-hand side b and the inverse of the diagonal D^{-1}. As a result, our smoother is nominally memory-limited for all operators.

Figure 9 presents smoother performance on the finest grid (256^3) as a function of operator, platform, and optimization. Once again we have included a Roofline bound (blue circle) to indicate the nominal performance bound of a smoother prior to any kind of communication-avoiding algorithmic change. Observe that the baseline implementation of the smoother is well below the Roofline bound in all cases. This is a result of the smoother being applied in two stages — application of the linear operator, and using that result to correct x. Enabling the operator "Fusion" optimization in CHiLL allows the compiler to automatically fuse these operations and eliminate the access to the intermediate temporary array. This significantly reduces data movement and allows the more memory-intensive 7- and 13-point smoother to reach their Roofline bounds.
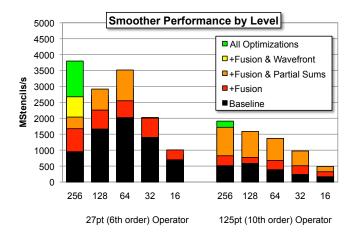
Fig. 9. Jacobi smoother performance attained with the CHiLL compiler by optimization, operator, and platform. NOTE, the Roofline memory bound is for the non-communication-avoiding (no wavefront) implementation and is lower than ApplyOp due to additional data movement like the RHS. The wavefront transformation allows CHiLL to exceed this limit.

Fig. 10. Jacobi smoother performance on Edison attained with the CHiLL compiler as a function of level in the V-Cycle (256^3 fine grids down to 16^3 coarse grids) for the 27- and 125-point operators. Observe that the reference implementation of the memory-limited 27-point operator receives a cache boost on the coarser levels while the compute-limited 125-point does not.

Unfortunately, the more compute-intensive 27- and 125-point stencils demand efficient SIMDization to reach their Roofline bounds. Enabling the "Partial Sums" optimization in CHiLL allows the compiler to automatically restructure these stencils to eliminate superfluous additions and regiment the computation for SIMDization by the backend compiler. The benefit is clear — a more than doubling of 125-point smoother performance and performance near the Roofline bound for all operators.

Naively, one may conclude that reaching the Roofline-bound represents the upper end of performance. However, this simply implies that a new set of algorithmic optimizations are required to further improve performance. As smooths are applied in sequence within the multigrid V-Cycle, it is possible, to view their execution as a quadruply nested loop. Manually reordering these loops is beneficial, but unproductive [26]. Conversely, our additions to CHiLL allow the compiler to automatically add ghost zones and restructure the loops into a communication-avoiding "wavefront" without loss of accuracy (the result is bit identical) or productivity. As seen in Figure 9, our approach attains roughly a $2\times$ performance boost for the 7- and 27-point smoother on Edison. (Note, "All Optimizations" include tuned nested OpenMP.) Whereas Edison is heavily memory-limited, Hopper is not. As such, the benefit of a communication-avoiding algorithm is limited on Hopper. Communication-avoiding 13- and 125-point smoothers suffer on two axes. First, generating wavefronts for these operators require skewing loops by the larger stencil radius. This larger skew factor increases the working set, increases cache pressure and makes it difficult to fit the working set in the fastest caches. Second, the 125-point operator is likely compute-bound on Hopper and nearly compute-bound on Edison. Thus, the potential benefit from communication-avoiding is small.

E. Smoother Performance Throughout the V-Cycle

Unlike simple explicit methods that only need to attain high performance for a stencil on a large grid, multigrid requires high performance on grids of exponentially varying size. In Figure 10, we explore the performance of the 27- and 125-point smoothers on Edison as they operate on coarser (smaller) grids. Examining the baseline implementation for the 27-point operator shows the expected rise in performance when moving to the coarser grids, which nominally fit in ever lower levels of cache. Note, the first smooth at each level will inevitably read from DRAM. As such, high cache bandwidths only amortize this slow initial smooth. On small grids, efficient 12-way OpenMP multithreading becomes impossible and performance drops. The 125-point smoother sees a similar behavior but to a lesser degree as it is ultimately compute-limited.

As optimizations are enabled in CHiLL, we see the compiler can nearly sustain constant performance for the 256^3, 128^3, and 64^3 levels for the 27-point operator by automatically tuning for the optimal optimizations. Similarly, Table II shows the compiler continually shifts the set and parameterization of the optimizations employed for the 125-point smoother at each level of the V-Cycle. A manually-optimized implementation would likely only target the fine grid and would thus deliver lower performance on the coarse grids, while significantly increasing programmer overhead.

The partial sums optimization requires a two pass approach in which the first creates a few auxiliary results. The cost of this initial pass is amortized on large arrays but becomes an impediment on small arrays. Thus the benefit of partial sums decreases on the smaller grids.

F. miniGMG Solver Performance and Error

Figure 11 presents the performance of the miniGMG multigrid solver using either the existing Intel compiler (baseline) or our optimizing CHiLL compiler as a function of discretization and platform. Performance is expressed in millions of degrees of freedom solved per second (DOF/s). For this scalar problem, fine-grid cell is one degree of freedom. Thus, solving a 256^3 grid in 1 second would equate to 16.78 million DOF/s. Generally, the CHiLL compiler can provide an overall speedup of about $2\times$ using all available optimizations. The attained speedup was a bit less on the 13-point operator as it did not benefit from partial sums and achieving high performance on a communication-avoiding wavefront is particularly challenging.
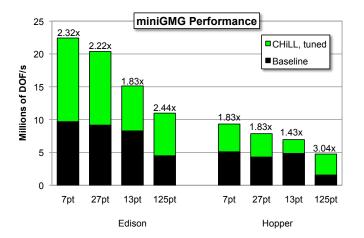
Fig. 11. miniGMG performance (millions of degrees of freedom solved per second) using either the Intel compiler (baseline) or the CHiLL compiler. Note, the labels indicate the overall solver speedup attained via CHiLL. The performance of the 10th order 125-point solver is within a factor of 2 of the 2nd order 7-point solver but provides nearly a million times lower error.

Fig. 12. Error attained in miniGMG as a function of operator and grid size. Note, a 1GB vector represents a single 512^3 grid. Multigrid will require several of these grids. Observe that the 10th order method delivers a three-digit increase in accuracy for every $8\times$ increase in memory.

When comparing raw performance (MStencil/s), results show that the 7- and 27-point operators were comparable with the 125-point operator, attaining about half the throughput. However, this only conveys half the story. As highlighted in Figure 12, miniGMG with the Mehrstellen correction attains roughly $2,370\times$ and $377,000\times$ better accuracy for our test problem using the 27- or 125-point operators (respectively) than miniGMG attained using the 7-point operator. As we move to ever finer domains, the benefit increases. Thus, if the goal were to solve to a particular numerical error, using the Mehrstellen correction with these discretizations reduces total data movement by *several orders of magnitude*.

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 [19, 32–37]. In recent years, numerous efforts have focused on increasing temporal locality by fusing multiple stencil sweeps through

125-point smoother on Edison

Level	256^3	128^3	4^3	32^3	16^3
Box Size	64^3	32^3	16^3	8^3	4^3
Operator Fusion	✓	✓	✓	✓	✓
Partial Sums	✓	✓	✓	✓	✓
Wavefront Depth	2	-	-	-	-
Nested OpenMP	<4,3>	<12,1>	<12,1>	<12,1>	<12,1>

125-point smoother on Hopper

Level	256^3	128^3	4^3	32^3	16^3
Box Size	64^3	32^3	16^3	8^3	4^3
Operator Fusion	✓	✓	✓	✓	✓
Partial Sums	✓	✓	✓	✓	✓
Wavefront Depth	1	-	-	-	-
Nested OpenMP	<6,1>	<6,1>	<6,1>	<6,1>	<6,1>

TABLE II. CHiLL WAS ABLE TO SELECT OPTIMIZATIONS UNIQUELY FOR EACH MULTIGRID LEVEL AND PLATFORM. <#,#>DENOTES THE NUMBER OF INTER- AND INTRA-BOX THREADS WITH NESTED OPENMP.
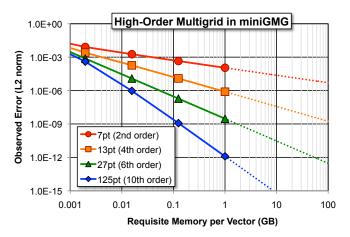
techniques like cache oblivious, time skewing, wavefront or overlapped tiling [1–12, 38]. In addition, domain-specific compilers have recently been developed for parallel code generation from a stylized stencil specification [39–41] or from a code excerpt [42].

Prior work developed manual optimizations for multigrid that incorporated communication-avoiding optimizations such as fusion of operators, wavefront parallelism, and hierarchical threading [26]. 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 [16, 17]. In this paper, we expand the compiler-directed optimizations to include partial sums. 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 [43] which focuses on image processing pipelines is the only other framework to do so.

Manual optimization of stencil computations has developed techniques such as *semi-stencils* to reduce loads [21] and using array common subexpression elimination after unrolling to reduce floating-point computations [19, 44]. In [19, 44], the authors unroll the loops of the stencil computation to expose array common subexpressions and reorder computation to reduce floating point operations using a stencil-specific code generator. Our approach is automated, and does not rely on unrolling. As discussed in Section III, Deitz et al. describe an automated approach to common subexpression elimination in the ZPL compiler [14]. Polyhedral techniques for recon-figurable computing construct custom storage structures to exploit reuse [45], but are limited to reuse between consecutive iterations and do not consider higher-order stencils.

Recent work reorders stencil computations from the direct specification of the stencil as an update from a set of input points [15]. Stencils are converted to a reduction and then loop shifting exposes register reuse of the same input, contributing to different output. Their approach does not reduce floating point computations.

The data layout transformation (DLT) was developed to eliminate the data stream alignment problem in generating SIMD code for stencil computations [25]. DLT transposes stencil inputs so that multiple computations can be performed in SIMD registers without the costly shifting of data across iterations of the innermost loop. Partial sums access aligned planes and buffer them in separate arrays; thus, we have addressed stream alignment without requiring the data transpose.

In summary, our compiler's ability to compose transformations allows it to take a higher-order smoother, remove its computation bottleneck with partial sums and make it bandwidth-limited, and then further apply DRAM bandwidth-reducing optimizations. No prior work has combined reducing floating-point operations with communication-avoiding optimization for higher-order stencils.

VII. Conclusions and Future Work

High-order discretizations of the Laplacian often result in compute-intensive stencils that perform more than an order of magnitude more floating-point operations per point than the traditional 2nd order discretization. The paradigm shift from compute-limited architectures to bandwidth-limited architectures has revitalized interest in these methods. In this paper, we explored several novel augmentations to the CHiLL compiler designed to improve the computational performance of these stencils. Using the miniGMG multigrid benchmark, we showed that the compiler could nearly quadruple the performance of the 125-point Jacobi smoother on Edison and reach the Roofline performance bound. Moreover, we showed the compiler could tune optimizations independently by platform and multigrid level.

To highlight the true potential of high-order methods, we examined the reduction in error as we increase resolution. We show that our 10th order method using the compact 125-point stencil in conjunction with a Mehrstellen correction attains 10th order accuracy and provides a $1000\times$ increase in accuracy for every $8\times$ increase in memory. This has the tremendous potential of reducing total data movement and total energy (across a supercomputer) by many orders of magnitude compared to the 2nd order multigrid solver.

Future work will explore integration of these techniques into the HPGMG benchmark which implements a true distributed V-Cycle. This will demonstrate the benefit of our optimizations at large scale and will facilitate implementations on Xeon Phi and GPU-based platforms. In addition, we plan to leverage task based parallelism and explore tiling and threading strategies to target the newer platforms.

References

[1] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the Cell processor for scientific computing," in *Computing Frontiers*, 2006.

[2] 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 *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[3] 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*, 2009.

[4] M. Frigo and V. Strumpen, "Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations," in *International Conference on Supercomputing*, 2005.

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

[6] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Parallel and Distributed Computing Systems*, 2000.

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

[8] 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*, 2009.

[9] 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*, 2008.

[10] P. Ghysels, P. Kosiewicz, and W. Vanroose, "Improving the arithmetic intensity of multigrid with the help of polynomial smoothers," *Numerical Linear Algebra with Applications*, 2012.

[11] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *International Symposium on Code Generation and Optimization (CGO)*, 2012.

[12] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2007.

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

[14] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Eliminating redundancies in sum-of-product array computations," in *International Conference on Supercomputing*, 2001.

[15] 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, 2014.

[16] 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.

[17] P. Basu, S. Williams, B. V. Straalen, L. Oliker, and M. Hall, "Converting stencils to accumulations for communication-avoiding optimization in geometric multigrid," in *Workshop on Stencil Computations (WOSC)*, 2014.

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

[19] 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 *Supercomputing (SC)*, 2008.

[20] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Auto-tuning the 27-point stencil for multicore," in *International Workshop on Automatic Performance Tuning (iWAPT)*, 2009.

[21] R. De La Cruz, M. Araya-Polo, and J. M. Cela, "Introducing the semi-stencil algorithm," in *International Conference on Parallel Processing and Applied Mathematics: Part I (PPAM)*, 2010.

[22] L. Collatz, *The numerical treatment of differential equations*. Springer, 1960.

[23] Q. Zhang, H. Johansen, and P. Colella, "A fourth-order accurate finite-volume method with structured adaptive mesh refinement for solving the advection-diffusion equation," *SIAM J. Sci. Comput.*, vol. 34, no. 2, pp. 179–201, 2012. [Online]. Available: http://dx.doi.org/10.1137/110820105

[24] P. Colella, C. Kavoulklis, P. McCorquodale, and B. V. Straalen, "A high-performanceimplementation of the method of local corrections for constant-coefficient elliptic pde," in preparation.

[25] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC/ETAPS)*, 2011.

[26] 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 *Supercomputing (SC)*, 2012.

[27] 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*, 2009.

[28] C. Chen, "Polyhedra scanning revisited," in *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2012.

[29] "Edison:," www.nersc.gov/users/computational-systems/edison.

[30] "Hopper:," www.nersc.gov/users/computational-systems/hopper.

[31] S. Williams, A. Almgren, M. Adams, and B. V. Straalen, "miniGMG," crd.lbl.gov/groups-depts/ftg/projects/current-projects/xtune/miniGMG, 2013.

[32] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

[33] G. Rivera and C. Tseng, "Tiling optimizations for 3D scientific computations," in *Supercomputing (SC)*, 2000.

[34] 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)*, 2008.

[35] S. Williams, L. Oliker, J. Carter, and J. Shalf, "Extracting ultra-scale lattice Boltzmann performance via hierarchical and distributed auto-tuning," in *Supercomputing (SC)*, 2011.

[36] M. Kowarschik and C. Wei, "Dimepack - a cache-optimized multigrid library," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), volume I*, 2001.

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

[38] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 79–84. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513905

[39] 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.

[40] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *ACM symposium on Parallelism in algorithms and architectures*, 2011.

[41] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *International Symposium on Code Generation and Optimization (CGO)*, 2012.

[42] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *International Conference on Supercomputing (ICS)*, 2012.

[43] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[44] K. Datta, "Auto-tuning stencil codes for cache-based multicore platforms," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2009.

[45] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013.