Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs

Tuowen Zhao, Mary Hall
School of Computing
University of Utah
{ztuowen,mhall}@cs.utah.edu

Protonu Basu
Facebook
protonu@fb.com

Samuel Williams, Hans Johansen
Computational Research Division
Lawrence Berkeley National Lab
{swwilliams,hjohansen}@lbl.gov

ABSTRACT

Stencil computations in real-world scientific applications may contain multiple interrelated stencils, have multiple input grids, and use higher order discretizations with high arithmetic intensity and complex expression structures. In combination, these properties place immense demands on the memory hierarchy that limit performance. Blocking techniques like tiling are used to exploit reuse in caches. Additional fine-grain data blocking can also reduce TLB, hardware prefetch, and cache pressure.

In this paper, we present a code generation approach designed to further improve tiled stencil performance by exploiting reuse within the block, increasing instruction-level parallelism, and exposing opportunities for the backend compiler to eliminate redundant computation. It also enables efficient vector code generation for CPUs and GPUs. For a wide range of complex stencil computations, we are able to achieve substantial speedups over tiled baselines for the Intel KNL, Intel Skylake-X, and NVIDIA P100 architectures.

CCS CONCEPTS
• Software and its engineering → Source code generation;
• Computing methodologies → Parallel programming languages.

KEYWORDS
Compiler Optimization, Stencil, vectorization

ACM Reference Format:

1 INTRODUCTION

Stencil computations are ubiquitous in scientific applications that solve partial differential equations using the finite difference or finite volume methods, where the derivative at each point in space is calculated as a weighted sum of neighboring point values (a “stencil”). A stencil’s order of accuracy is the exponent on the relationship between grid spacing (array size) and error — both small grid spacings (large arrays) and high order can result in low error. A stencil’s order greatly impacts the optimizations needed to achieve high performance. Low-order discretizations result in smaller stencils that have limited data reuse, are typically bound by memory bandwidth, and thus underutilize the compute capability afforded by manycore, wide vector, and GPU architectures. Much of the prior work in this field has been based on lower order stencils and has thus focused on techniques to reduce main memory data movement [6, 13, 14, 20, 21, 23, 28, 33, 36, 38, 41, 45].

As processor architectures become more compute-intensive [37], computational scientists are increasingly turning to high-order schemes that perform more computation per point (more compute-intensive) but can attain equal error with larger grid spacings (smaller arrays). Although higher-order stencils inherently result in higher arithmetic intensity, they also place immense pressure on register file, cache, TLB, and hardware prefetchers. Worse still, to further utilize available compute capability, stencil computations are often a composition of multiple high-order stencils, such as the $8^{th}$-order hiperterm kernel, described in [12], and depicted in Figure 1. It computes five stencils that operate on eight input fields.

Prior work on optimizing high-order stencils leverages the associativity of the weighted sums in a stencil computation; such operations can therefore be safely reordered to achieve the same result within round-off tolerances. Consequently, execution order can be optimized to exploit data reuse, and thus reduce memory load/store operations and reduce register pressure [3, 10, 24, 25, 29]. Prior associative reordering methods for stencils are limited in several ways. Most focus on reuse of individual data elements, with an eye towards optimizing scalar registers [24, 25]. Where reuse of vectors is considered to support vector code generation, it is limited to isotropic, constant-coefficient stencils [3], or arises from a post-pass vectorization, preceded by DLT (data-layout transformation) optimization [29]. In some cases, cross-iteration reuse is identified as a byproduct of loop unrolling [10, 25]. Only one of these approaches targets GPUs, and it exploits reuse just within an expression [24].

This paper addresses these limitations, describing a vector code generator for general stencil computations targeting both CPUs and GPUs. It identifies data reuse without unrolling within a fine-grain block of a stencil computation. For further optimization gains, this approach to reuse analysis and vectorization can also work in tandem with a fine-grained blocked data layout that decomposes the original grid domain into small, fixed-size multi-dimensional...
subdomains [2, 17, 40], such as bricks [44], which have been shown to achieve performance portability across CPU and GPU. Bricks are stored contiguously in memory to enable a number of optimizations. First, accesses within a brick are part of a single address stream, mitigating the negative impact of blocking on hardware prefetchers and TLB. Second, when combined with vector folding [39], an individual dimension can be smaller than the vector width; this flexibility can reduce cache and register pressure for complex stencils like hypTern. Other stencil optimizations such as temporal blocking and wavefront parallelism are beyond the scope of this paper, but are complementary and can be combined with our method.

This paper makes the following contributions: (1) it presents a vector code generation algorithm for general stencil computations that exploits data reuse within a block without unrolling, and targets both CPUs and GPUs; (2) it compares the effectiveness of the code generation approach for iteration space tiling vs. bricks on CPUs, isolating the benefits of each; (3) it offers the first description of node-level vector code generation for bricks; (4) it presents performance results on 24 stencils, including real-world proxy stencils such as hypTern, demonstrating performance gains on Intel Knights Landing (Xeon Phi) processors (up to 3.4×), Intel Xeon Skylake-X (1.3×), and NVIDIA P100 (1.6×).

2 BACKGROUND AND MOTIVATION

In this section, we motivate our approach, using the hypTern kernel, with code and the compiler’s expression tree shown in Figure 1. Stencils like hypTern exhibit high temporal reuse across stencil iterations, e.g., 
\[ \text{cons}[i][m][k][j][i+1] \text{ and cons}[i][m][k][j][i-1] \]

two iterations later. It is common to use tiling to exploit this reuse in caches or unrolling/unroll-and-jam to enable optimizations for reuse in registers. Additional array common subexpressions within and across expressions, such as the results of the shaded operators at the bottom of Figure 1, can also be reused in registers. Due to the complexity of hypTern, exploiting such register reuse can lead to severe register pressure; exposing cross-iteration reuse using unrolling may increase register pressure, and even cause instruction cache misses. In addition, hypTern has high arithmetic intensity, with 358 floating-point operations per iteration. Achieving high performance also demands efficient use of wide SIMD units in CPUs and SIMT threads in GPUs.

Another consideration is that hypTern places immense pressure on the TLB and hardware prefetcher due to the number of independent data streams. One k-j plane of hypTern requires 133 simultaneously active read or write data streams (corresponding to different registers, cache lines and potentially, TLB entries). Tiling will exacerbate this problem. To reduce the number of data streams for such stencils, prior work has developed variations of blocked data layouts, where the original grid domain is decomposed into small, fixed-sized multi-dimensional subdomains [2, 17, 40]. In this paper, we expand on the concept of bricks, where these subdomains are stored contiguously in memory [44]. Using an 8x8x8 brick size and stencil radius \( \leq 8 \), we access the elements within a brick using a single stream as opposed to 64 streams for a tiled code. The computation inside one brick would be similar to an 8x8x8 tiled stencil.

Taken together, this paper describes a vector code generator that can balance the aforementioned optimization requirements of high-order stencils such as hypTern. Our approach exploits reuse within a multi-dimensional data block, arising from either tiling, which reorders the computation, or bricks, which also reorganizes the data layout. As stencils are known to pose challenges to vectorization due to issues of alignment [15], the approach must expose aligned vector operations. Additionally, our approach further reduces arithmetic intensity by exposing opportunities for array common subexpression elimination [10]. The remainder of this section provides the foundation for the code generation approach.

2.1 Stencils as Gather or Scatter Operations

The kernel of a stencil computation typically contains a weighted sum of neighboring points. Such sums are most commonly expressed as gather operations, as in the 5-point 2D stencil code of Figure 2(a), where the value of this sum is calculated for each iteration of a loop nest by gathering its neighboring inputs (some of which are widely spaced in memory), individually weighting them, and summing them. Figure 3(a) visualizes this gather computational pattern for the 5-point stencil code.

However, one can observe that these weighted sums are associative and can be reordered without changing the meaning of the computation. This concept is associative reordering. Therefore, an alternative implementation of the 5-point stencil is a scatter operation, where one input is weighted and scattered to all the neighboring points that use it as a term in the sum. Figure 3(b) shows the resultant scatter pattern for the 5-point stencil. Scatters have several advantages including minimizing the number of loads, and improving instruction level parallelism, but may increase the number of stores. For high-order stencils that access a large number of inputs to compute each output point, an approach that favors reducing loads is preferable to one that reduces stores. We also find that the output data often resides in registers, particularly on GPUs, or in L1 cache, so store cost is typically low. Scatter also matches the strengths and weaknesses of bricks. Loads are more costly, because accesses that cross brick boundaries introduce indexing overhead due to the adjacency list, and unaligned loads are not applicable.

In the following, we will select some portions of the computation with high reuse to be computed using scatter.

2.2 Overview of Approach

The current code generator uses a domain-specific frontend, implemented in Python, that accepts stencil descriptions as input, shown in Figure 2(c). The output of the code generator is integrated into C code as in Figure 2(d). From this specification, we can generate either tiled or brick code that incorporates scatter, as used in the experiments of Section 5. The data layout for the tiled code is a 3D array. Individual bricks are stored in contiguous memory, and a collection of bricks that represents a domain are organized as a graph using an adjacency list. To compute a stencil, one iterates over all indices of bricks and, for each brick, computes the stencil at all \((k, j, i)\) in the dimensions of a brick.

To detect reuse within and across stencils, we formulate the problem on an expression directed acyclic graph (DAG) of the stencil kernel as in Figure 1. We identify the operands in the DAG that
are reused within or across iterations of the same expression. A profitability analysis determines whether a scatter should be used to optimize the redundant loads, and derives an iteration schedule. Operators containing operands for which a scatter is profitable are marked for reordering. The unmarked portions of the computation will use gather operations. We then group the marked subexpressions into stages to be computed together to capitalize on reuse across subexpression DAGs. Common subexpressions, which by definition use the same input, are likely to be grouped together. Indirectly, this may result in common subexpression elimination (CSE) [3, 10], as the backend compiler can more easily detect common subexpressions if they are adjacent instructions.

With the dimensions of the block, we use the stages and scatter schedule to produce vectorized code. Given the results of analysis, the code generator derives new loop bounds for the resulting tile, constrained by the boundaries of the buffers, with loop peeling as needed to implement the full block. Thus, instead of unrolling first and identifying reuse patterns in the unrolled code, we identify reuse based on indexing expressions of operands, and create the loops indicated by the profitability analysis. The code generator performs a few additional optimizations during vectorization. Our approach, with vectorization, is portable across both CPUs and GPUs. The code generation technique is detailed in the next two sections. Section 3 discusses the analysis that identifies reuse and decides how to split the computation into stages. Section 4 describes the actual vector code generation.

3 REUSE-BASED EXPRESSION SPLITTING

This section describes how to split a stencil kernel into compute stages based on its reuse pattern. We first build an expression directed acyclic graph (DAG) from the code. For simplicity, we illustrate the algorithm using the running example of Figure 2 whose expression DAG is shown in Figure 4, but also refer to the DAG for hypterm in Figure 1 when discussing operator grouping. The code generation framework obtains this graph by identifying the assignments to grids in the original abstract syntax tree (AST), and the operators, constants and grid references on the right hand side. The output grid is the target of the DAG, and the operand grids are

...
In this phase we start from the expression DAG using two major steps (1) identify reuse profitability and select associative operators to be reordered by postorder traversal of the expression DAG; (2) identify opportunities to group operators across expressions into stages to further improve data reuse.

Figure 4: Expression DAG for the 5-point stencil of Figure 2.

In this phase we start from the expression DAG using two major steps (1) identify reuse profitability and select associative operators to be reordered by postorder traversal of the expression DAG; (2) identify opportunities to group operators across expressions into stages to further improve data reuse.

3.1 Reuse Profitability in Operators

The first step of the code generation algorithm is to mark associative operators in the expression DAG for reordering based on a profitability analysis. We define the profitability of scatter using the reduction of the number of inputs that are simultaneously live for each iteration step. We calculate profitability using a postorder traversal of the expression DAG, and we then mark the operators to be reordered that exceed a profitability threshold.

The postorder traversal collects $R(E)$, grids and offsets from the DAG for expression $E$, as in Equation 1. The number of elements in this set is then the number of unique reads when calculating this subexpression as how they appear in the expression DAG.

$$R(E) = \{(g, \delta)|\text{grid } g \text{ appears in } E \text{ with offset } \delta\} \quad (1)$$

When $E$ is an associative operator, it consists of multiple terms (subexpressions), $E_i$. When we shift the terms of the associative operator between iterations we are effectively adding a per-term constant $\delta_i$ to the offset of the corresponding term. This shift, $\delta_i$, is sometimes referred to as the retiming vector in loop shifting [29]. If we add the shifts to all terms of the operator then we can collect the set of grids and offsets with Equation 2. The number of elements in this set is then the number of unique reads when calculating the associative operator with each term shifted by $\Delta = \{\delta_i\}$.

$$R(E^\Delta) = \bigcup_i \{(g, \delta + \delta_i)|\langle g, \delta \rangle \in R(E_i)\} \quad (2)$$

Using the cardinality of the two sets from Equation 1 and Equation 2, we can evaluate the profitability $P$ of a reordered expression $E^\Delta$ as in Equation 3. Reordering is marked as profitable if reads are reduced by a large fraction; that is, when $P$ exceeds a global threshold $t_1 \geq 1$.

$$P(E, E^\Delta) = \frac{|R(E)|}{|R(E^\Delta)|} \geq t_1 \quad (3)$$
Assuming this profitability is above the predefined threshold (dashed arrow), Algorithm 1 is used to try to minimize distinct references by adding shifts to each term (numbers on edge). If the operator should be reordered (per Equation 3), the DAG is marked with shift $\delta_i$.

Note that $|R(E)|$ is a property of the stencil computation. In order to maximize $P$, we only need to find a set of shifts, $\Delta$, that minimize $|R(E^\Delta)|$.

This process is illustrated in Figure 5 for the 5-point stencil. During post-order traversal, when an associative operator is encountered, Algorithm 1 is used to inspect its terms; this is denoted in the figure by a dashed arrow. We note that in the original computation there are five distinct references, thus $|R(E)| = 5$. The algorithm assigns each term with one shift value that is marked on the edges; with shift this produces the same reference: In: 0, -1

Thus, from Equation 2, the number of distinct references after the shift is 1. This gives us a reuse profitability of 5 from Equation 3. Assuming this profitability is above the predefined threshold $t_1$, this addition is marked in the original graph with the shift values produced from Algorithm 1, $\delta$, annotated to edges leading to each term.

The number of possible shift amounts for one term is related to the radius of the stencil; the total search space is exponential in the number of terms. This search space is potentially too large to search exhaustively. However, the set of offsets that minimizes $P(E^\Delta)$ has the properties of Equation 4: the minimum only arises when for each term, it either has no common grid input with any other terms, or it has at least one read that can be reused after shifting.

$$\forall E_i, \text{ either }$$

$$\exists (g, \bar{\delta}) \in R(E_i), g \notin R(\Xi - \{E_i\}) \text{, or }$$

$$\exists (g, \bar{\delta}) \in R(E_i), (g, \bar{\delta} + \delta_i) \in R((\Xi - \{E_i\})^{\Lambda}(\bar{\delta}_i))$$

Proof. Equation 4 can be proven using contradiction by assuming the negation: $|R(E^\Delta)|$ is the minimum but has an $E_i$ that shares a common grid input and does not have reuse with shift $\delta_i$. In this case, we can change $\delta_i$ so that at least one read is reused with some other term. We have the new $|R(E^\Delta')|$, which will be smaller by at least one. It contradicts the assumption that $\Delta^*$ gives the minimum. Thus, Equation 4 must be true.

Based on (4), we developed a fast greedy algorithm in Algorithm 1. The complexity of this algorithm is $O(n^4 \log n)$ where $n = \sum |R(E_i)|$. This $n$ is bounded by the number of reads in the original stencil code, $N$. In fact, this is only a loose upper bound for the complexity, and for associative operators that only have one offset for all reads in one term, such as the hypertext stencil in Figure 1 and many of other stencils, its complexity is only $O(n)$. After the scatter is decided, we can then compute the profitability and mark associative operators for grouping.

**Algorithm 1** Greedy algorithm for deciding shift amounts for child subexpressions of $E$

1. Initialize $\Delta = \{\delta_i\} = \emptyset$
2. Repeat for all child subexpressions $E_i$
   3. For all $(g, \delta) \in R(E_i)$
      4. For all $(g', \bar{\delta}) \in R(E_{j \neq i})$
         5. $g = g'$
            6. $\delta' = \bar{\delta} + \delta - \bar{\delta}$
               7. Compute new shift $\delta_i' = \delta_i$
               8. Update $\Delta$
      9. If obtains a lower $|R(E^\Delta)|$
         10. $\delta_i = \delta_i'$
    11. End for
   12. End for
3. End for

   Until $\Delta$ isn’t updated

   Return $\Delta$

### 3.2 Cross-operator Reuse

Step 2 in our framework will enable optimization of loads across parts of the expression DAG by attempting to group reordered operators utilizing the same data as input to be computed together. Each marked computation can be computed once all the values from its subexpression are available, which includes all subexpressions that are marked. This creates a dependence graph that includes all marked subexpressions where edges are contracted from the original expression DAG.

Beyond optimizing for reuse across subexpressions, operator grouping also impacts the number of buffers that are simultaneously live. We use a modified topological sort based on a priority tuple measure similar to the one used in [25] to break ties during the sort when multiple alternatives are present. Applying this process results in a linearized sequence that tries to balance buffer usage and the number of operators that can be computed concurrently. This step has a complexity of at most $O(m^2)$ where $m$ is the number of marked subexpressions.

We can then scan the sequence and merge nearby operators into one stage based on a measure of hardware pressure, which is derived...
from (1) the number of distinct inputs; and, (2) the number of buffers accessed. Both of them can represent the grouped operators’ pressure on the registers. The first value can be computed using Algorithm 1. To determine viability, we compute a weighted sum by the algorithm as profitable for reordering, a gather, using the number of operators in the expression DAG. For any groups not identified, we compute all compute stages are created, we obtain a sequence of groups reached faster, preventing aggressive grouping.

4.1 Vector Scatter

While we focus on scattered computation with vectors, this is derived from reordering using scalar values. We first use the shifts computed for each term. For iteration \( p \) within the tile, we can determine the shifted loop index, the source of the scatter, by subtracting \( \delta_i \) as in Equation 6. Each of the sources will scatter the corresponding subexpression that is shifted by adding \( \delta_i \) to all the original array indices. The scatter can then walk through all such \( \bar{p} \) and compute the value for each term to the destination with \( \bar{p} + \delta_i \).

\[
\bar{p}' = p - \delta_i
\]  

When generating vector code, we add restrictions to the destination so that only destinations that are aligned are written. To accommodate wide vectors and small data blocking, we assume the vector is a multidimensional vector-sized rectangle, that exhibits a length \( v(d) \) on each tile dimension \( d \). Then we have an aligned

4 VECTOR CODE GENERATION

Once all compute stages are created, we obtain a sequence of groups of operators in the expression DAG. For any groups not identified by the algorithm as profitable for reordering, a gather, using the original computation, will be generated. Other stages contain associative operators that are profitable for reordering; these operators will have a shift, \( \delta_i \), associated with each term, \( \{E_i\} \), of the operator. We can then generate code for each stage in the sequence using either gather or scatter. In this section, we describe how vector code is generated from \( \{E_i\} \) and \( \Delta \) with loops. We also present additional optimizations that we used. A walk through of the code generation is shown in Figure 6 for the 5-point stencil.

The techniques in this section are based on scanning the iteration space and the expression DAG side-by-side, which results in a complexity of \( O(|S||V|) \), where \( |S| \) is the size of the tile and \( |V| \) is the number of nodes in the expression DAG.

Figure 6: Vector code generation for 5-point stencil example in Figure 2,5. The generated code employs dimensional splitting and reuses aligned vector load.
We employ two additional optimizations when generating scatter

### 4.2 Optimizations

These optimizations are illustrated in the upper half of Figure 6. Upper left of the figure shows how vector loads are generated when we are traversing the contiguous direction while the two aligned vectors will be the same for several iterations. These aligned loads can then be cached and reused for consecutive iterations. Upper right of the figure shows how we can create two temporary vectors vl and vr to cache the aligned vectors. With this optimization, we only load these values three times, \(2 + 0 + 1\), instead of five, \(2 + 1 + 2\).

We also employ dimensional splitting of the stencil to further reduce the final code size. We observe that Equation 8 is only related to the shift selected for each term, and it is often more constrained than Equation 9. If we directly generate code for the 5-point stencil example in Figure 6, the calculation for terms on the I-dimension is unnecessarily peeled because of the two terms on the J-dimension. We can then group the terms based on their values to increase the range of Equation 8. We achieve this by picking one dimension at a time and group the terms based on their shifts on the other dimensions. We then select the most frequent groups to be computed together. Larger loops can be created for the other dimensions where they have a common shift value. We repeat this process for each of the dimensions to fully split the stencils. This process is applied to the 5-point stencil in Figure 6. For stencils such as CNS, Figure 1, we can achieve a perfect split as peeling only happens for one dimension at a time. Without dimensional splitting, no loops are possible for the stencil with \(8 \times 8 \times 8\) tile.

### 4.3 Vectorizing on GPU

As observed by prior work [46], GPUs offer the same vector merging capability as alignr intrinsic on the CPU using either shared memory or shuffle instructions, \texttt{__shfl_up} and \texttt{__shfl_down}. This allows us to transfer our vectorization method onto NVIDIA GPUs and use each warp as one vector that has a length of 32. Also, through use of vector folding and combinations of multiple shuffles we enable support for smaller data blocks such as \(8 \times 8 \times 8\) or \(4 \times 4 \times 4\).

## 5 EXPERIMENTAL RESULTS

This section presents performance results for the generated code for both CPUs and GPUs, applied to tiled or brick code.

### 5.1 Target Architectures

**Intel Knights Landing.** The Intel Xeon Phi 7250 Knights Landing (KNL) has 68 physical cores organized into a 2D on-chip mesh of 34 tiles each with two CPU cores\(^1\) and a shared 1MB L2 cache. Each core has a private 32KB L1 data cache, implements 4-way multithreading, and has two AVX-512 vector processing units (VPUs). AVX-512 instructions operate on 8 double-precision or 16 single-precision floating-point data elements in a SIMD fashion. Its theoretical peak performance is 2611.2 GFLOP/s double-precision fused multiply-and-add. Each chip has both standard DDR4 DRAM memory and high-bandwidth MCDRAM memory that we configured as a last level cache using the quadcache mode, which yields a peak STREAM performance of 332 GB/s.

**Intel Xeon Gold (Skylake-X).** The Intel Xeon Gold 6130 CPU has 16 physical cores. Each core has a private 32KB L1 data cache and 1MB L2 cache, implements 4-way multithreading, and has two AVX-512 vector processing units (VPUs). Each core has a nominal frequency of 2.1GHz. The whole CPU has a theoretical peak performance is 1075.2 GFLOP/s. Concurrently, 6 DDR4 memory controllers provide a STREAM bandwidth of 85 GB/s.

**NVIDIA P100.** The P100 GPU has 56 streaming multiprocessors. Each streaming multiprocessor has 64 single-precision and 32 double-precision CUDA cores and has a warp size of 32. Each streaming multiprocessor has a dedicated texture/L1 cache. The P100 has a theoretical peak single-precision performance of 9.3 TFLOP/s, a peak double-precision performance of 4.7 TFLOP/s, and a GPU STREAM bandwidth of 586 GB/s.
Figure 7: Performance on KNL and Skylake-X. For real-world stencil kernels, smaller blocking sizes may be beneficial because more input grids can put higher pressure on the cache. For higher-order and real-world stencils, the brick approach often provides the best speedup: On KNL, hypterm – 3.4× and f3d125pt – 1.9×. On Skylake-X, hypterm – 1.3× and f3d125pt – 2.1×.

Table 1: Stencils used in experiments. Grid represents the number of grids the stencil operates on, while FLOPS represents the number of FLOPS performed per point.

<table>
<thead>
<tr>
<th>Name</th>
<th>Grid</th>
<th>FLOPS</th>
<th>Name</th>
<th>Grid</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencils from HPGMG</td>
<td></td>
<td></td>
<td>Stencils from CNS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chebyshev</td>
<td>6</td>
<td>39</td>
<td>poisson</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>helm-v2</td>
<td>7</td>
<td>22</td>
<td>helm-v4</td>
<td>7</td>
<td>115</td>
</tr>
<tr>
<td>Stencils from CNS</td>
<td></td>
<td></td>
<td>Stencils from CNS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hypterm</td>
<td>13</td>
<td>358</td>
<td>differm</td>
<td>11</td>
<td>415</td>
</tr>
<tr>
<td>s2d5pt</td>
<td>2</td>
<td>9</td>
<td>s2d9pt</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>s2d13pt</td>
<td>2</td>
<td>25</td>
<td>s2d17pt</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>s2d21pt</td>
<td>2</td>
<td>41</td>
<td>s2d25pt</td>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>s2d29pt</td>
<td>2</td>
<td>49</td>
<td>s2d33pt</td>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>2D Star-Shaped Stencils</td>
<td></td>
<td></td>
<td>2D Full Stencils</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2d9pt</td>
<td>2</td>
<td>17</td>
<td>f2d25pt</td>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>f2d49pt</td>
<td>2</td>
<td>97</td>
<td>f2d81pt</td>
<td>2</td>
<td>161</td>
</tr>
<tr>
<td>3D Star-Shaped Stencils</td>
<td></td>
<td></td>
<td>3D Full Stencils</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3d7pt</td>
<td>2</td>
<td>13</td>
<td>s3d13pt</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>s3d19pt</td>
<td>2</td>
<td>37</td>
<td>s3d25pt</td>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>3D Full Stencils</td>
<td></td>
<td></td>
<td>f3d27pt</td>
<td>2</td>
<td>53</td>
</tr>
</tbody>
</table>

5.2 Benchmarks and Proxy Codes

We use three categories of stencil kernels of varying order shown in Table 1. The first category includes smoothers from the HPGMG benchmark suite [1]. The second category includes high-order stencil computations from the Compressible Navier-Stokes computation [12]. The third category includes synthetic stencils to capture the memory-compute ratio of different kinds of stencil shapes and radii. Many of the real stencils are a composition of these kernel patterns. Stencils are named according to their class (“f” or “s”) and the number of points where each point is weighted individually. Class “s” refers to stencils for the Laplacian second derivatives, that only operate on elements along each of the axes (star-shaped) in each dimension. For Class “s”, the stencil radius is just half the order, there are no off-axis points in the stencil. Class “f” refers to stencils for the “compact” Laplacian that touch all points in a cube (dense), with Manhattan distance equal to the radius. The radius is again just half the order; in some applications these stencils can produce more accurate solutions.

The baseline code is written in C. Our code generator generates code from the stencil written as python expressions in a python script, as in Figure 2(c). This specification can be used as a standalone DSL or as an intermediate output from the parser. The thresholds described in Section 3 are exposed as parameters to the code generator. We used $t_1 = 1.5$ (reuse estimate), $t_2 = 20$ (simultaneously active buffers), and $k = 2$ (weight assigned to buffers) for our experiments. For all the stencils our code generator runs

1We use 32 tiles for a total of 64 cores in all our experiments to isolate system overhead.
peeled region from data indirection, the code size for bricks is 54KB. This is much higher than the L1 instruction cache size of 32KB.

We used Intel VTune Amplifier to obtain profiling results to further dissect the performance difference between different versions. We selected two of the complex stencils to show in Figure 8. When comparing brick to baseline on KNL, cache misses are reduced by up to 19×, and TLB misses are reduced by up to 49×. Similarly, on Skylake-X, brick also reduced L1 misses by up to 8× and TLB-related metrics up to 14×. In addition, even though the code generator increases the number of stores, these appear to be serviced from L1 due to the decrease in cache misses as compared to baseline. The effect of better locality is more pronounced on KNL with more threads and much smaller cache per thread. For f3d125pt, our code generation can also reduce the total number of loads.

For the real-world stencils we noticed that tile offers similar or worse performance compared to baseline. This is due to the inherently higher L1 cache pressure for these stencils. Table 1 shows the number of grids referenced for each of these stencils. While we prefer each of the grids to be located in the fastest cache available for reuse, the buffers from vector scatter increase L1 pressure and detract from the better locality it provides. As seen in Figure 8, the L1 miss count is even across all versions of the code for hypertext. As noted previously, KNL thread cores have less cache capacity: baseline and tile tend to achieve the best performance on 4×4×8 tiles or sometimes 4×4×4 tiles. The smaller brick version, which relies on vector folding, improves KNL performance due to reduced stores and improved TLB behavior.

### 5.4 GPU Performance: NVIDIA P100

Figure 9 presents NVIDIA P100 performance. We compare the performance of three code variants:

- **baseline version** (a gather) that is tiled using the thread-block decomposition of CUDA, where each CUDA block is comprised of 4×4×32 (K, J, I) threads for 3D or 8×32 for 2D. Each thread computes one stencil output. We also could spawn 32 threads for each block and iterate on the (K, J) dimension to further imitate the number of threads used for the brick version, but it is always slower.
- **tile version** that applies our code generation on a tiled thread-block, where it compute (K, J, I) elements using 1 threads. The tile size is the same as the baseline. Only aligned loads and reuse of those are not applied.
- **brick version** where fine-grained data blocking is used [44] with the code generator. The subdomain that each CUDA block will compute is the same as baseline, but only 32 threads (one warp) are in each block to compute all stencils in the subdomain, like a vector with width of 32. We also tried smaller sizes such as 4×4×8 and 4×4×4 for real-world stencils; these are reported as smaller brick. Note that the I-dimension of these sizes are smaller than the vector width on the GPU which is the warp size, 32.

In Figure 9, the baseline shows very little performance variation. This is because FLOP/s are limited by data dependences. Our approach reduces this bottleneck by exploiting input reuse. The NVCC compiler can generate code where the buffers introduced
Figure 8: Profiling metrics on KNL and Skylake-X. Each bar is the raw counter value, normalized to the baseline total number of stores. The generated code provides a dramatic reduction in cache and TLB misses and page walks. On KNL, L1 L2 and TLB misses are reduced as much as 19×, 7×, and 49× respectively. On Skylake-X, we also reduced L1 misses by up to 8× and TLB-related metrics up to 14×. These indicates much better data locality and cache reuse.

Figure 9: Performance on NVIDIA P100. Brick code generation produces the best performance for many of the stencils. For synthetic stencils, using either 32 threads or full block results has little effect, but smaller block sizes reduce cache pressure for real stencils. Bricks speedup many stencils, for example poisson (1.6×) and f3d125pt (7.0×).

by vector scatter reside in registers. With sufficient registers, one element is read and scattered to multiple destinations in registers so that the write cost is low compared to read. The effect of holding buffers in registers is reflected in the drop of performance between (s2d21pt,s2d25pt) and (s3d13pt,s3d19pt). Also note that due to alignment, fewer registers are required than for gather. For example, for f3d125pt only 25 destination register are used when scattering one input, whereas in gather 125 input are used.
The thresholds used in the experiment are fixed using lenient values. These thresholds are especially relaxed for simple stencils. Our reuse estimate of $t_1 = 1.5$ applies to all associative operators that have reuse potential in our tests. Aside from the value 1 that denotes no potential reuse, the lowest reuse potential is $12/7 \approx 1.86$ in helm-v2. This is due to the fact that helm-v2 contains terms with multiple grid references that are not perfectly reused after applying the shifts.

Our estimate of $t_2 = 20, k = 2$ is a proxy for how much register pressure the algorithm incurs. These criteria are also only effective for complex stencils and are rarely met for simple stencils. All synthetic stencils have a value for Equation 5 of 3. For complex stencils like helm-v4 and hypter-m, it is theoretically possible to reach this limit. However, this does not happen as our code generator strategy requires reuse between operators in one stage but limits the number of operators that can be put in one stage. This requirement may be proven to be too greedy and further tuning of these parameters is yet to be explored.

6 RELATED WORK

Optimization efforts for stencil computations can be broadly classified as memory access optimization techniques, and optimization methods to improve computation, although in practice, there is often significant interaction between them.

Most of the optimization effort has focused on stencils applied on large grids that are usually bound by capacity or compulsory cache misses, leading to a variety of studies on spatial and temporal tiling [6, 7, 11, 13, 19–23, 26–28, 33–36, 38, 41, 45]. In addition, domain-specific compilers have recently been developed for parallel code generation from a stylized stencil specification [4, 30, 42, 43] or from a code excerpt [16].

The aforementioned tiling techniques have focused on loop or iteration space tiling. In addition to loop tiling, researchers have also tiled or blocked data (space). Data along with loop tiling efforts have been addressed by [2, 17, 31, 40]. TiDa [31] uses coarse-grained data blocking, where the entire grid is tiled into sub-grids, each with its own ghost zone. Fine-grained data blocking is explored in Bricks [44] and Briquettes [17]. YASK [40] and RTM on the Cell processor [2]. All the fine-grained blocking techniques targeted large, compute-intensive stencils, and the small data blocks (bricks) do not have per-block ghost zones.
The fine-grained data blocks used in our research are similar to briquettes in [17], but there is significant difference in our approaches. Briquettes were designed to perform 3D stencils split into 1D stencils, thus requiring multiple sweeps to compute the output. Furthermore, a data transpose was required between each 1D stencil sweep to ensure good SIMDization. Their code generation required data staging tailored for 1D stencils. In contrast to Briquettes, we optimize 3D stencils without manual dimensional splitting and perform complex stencil reordering in addition to fine-grained data blocking to improve computation by reducing reads and improving SIMDization.

YASK is a C++ template-based approach to generating code for large stencils with fine-grained data blocks. YASK autotuned their data block size, and used smaller data blocks than our method (e.g. $2 \times 2 \times 4$ instead of $8^3$). They can generate code for clusters of vectors using unrolling and common expression elimination to improve reuse, which is less feasible for complex stencils. They did not directly target stencil reordering as presented in our paper.

Stencil reordering, one of the main characteristics of our approach, has been explored in different ways. Manual optimization of stencil computations has led to techniques such as semi-stencils to reduce loads [8] and using common subexpression elimination after unrolling to reduce floating-point computations, improve register reuse, reduce register pressure, or improve instruction level parallelism [5, 7, 25]. Some works target specific properties of the associative operation in stencils. Deitz et al. [10] describes an automated approach to common subexpression elimination for sum-of-product computation and is not applicable to uniquely weighted or variable coefficient stencils where no common subexpressions exist. Basu et al. [3] uses partial sums to reorder constant-coefficient isotropic stencils. Stock et al. [29] uses statement splitting to enable loop shifting to expose reuse of the same input and autotuning to determine the shift amount. They also do not target code generation for the GPU. In comparison, the research presented in this paper illustrates a new powerful stencil reordering method which works on general stencils without manual optimizations. Our method targets tiled stencil computation to improve cache and register reuse, and is designed to work with fine-grained data blocking on modern architectures with wide SIMD units.

High-order PDEs can also be implemented as dense matrix operations as in [32]; this paper and other previous compiler/code-generation research treats higher-order stencils as computations on structured grids. Using dense linear algebra primitives, although technically feasible for finite difference and finite volume methods, would be inefficient for our stencils as most of the entries in the resultant matrices would be zero.

7 CONCLUSION

High-order stencil computations are simultaneously best-suited to the trends in computer architecture (limited bandwidth coupled with high arithmetic intensity) and most-often underperforming. To that end, in this paper, we introduce a novel compiler optimization to exploit reuse and vectorization in block stencil computations. When coupled with a fine-grained blocked data layout (bricks) this produces code that reduces vector loads and alignment operations, exposes opportunities to eliminate redundant computation, and reduces the data footprint of stencils in the memory hierarchy. We show our approach improves the performance of real stencils compared to a tiled baseline by up to $3.4 \times$ on a Intel Knights Landing (Xeon Phi) processor, up to $1.3 \times$ on Intel Xeon Skylake-X, and up to $1.6 \times$ on NVIDIA P100.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration. This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Sciences Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231.

REFERENCES

Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs


