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:
Tuowen Zhao, Mary Hall, Protonu Basu, and Samuel Williams, Hans Johansen. 2019. Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356210

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 `hypterm` 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 `hypterm`. 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 `hypterm`, 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 `hypterm` kernel, with code and the compiler's expression tree shown in Figure 1. Stencils like `hypterm` exhibit high temporal reuse across stencil iterations, e.g., `cons[imx][k][j][i+1]` and `cons[imx][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 `hypterm`, 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, `hypterm` 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 `hypterm` places immense pressure on the TLB and hardware prefetcher due to the number of independent data streams. One k-j plane of `hypterm` 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 8×8×8 brick size and stencil radius ≤ 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 8×8×8 tiled stencil.

Taken together, this paper describes a vector code generator that can balance the aforementioned optimization requirements of high-order stencils such as `hypterm`. 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

```
flux[irho][k][j][i] =
        -dxinv0 * (
           ALP * cons[imx][k][j][i + 1] + -ALP * cons[imx][k][j][i - 1] +
           BET * cons[imx][k][j][i + 2] + -BET * cons[imx][k][j][i - 2] +
           GAM * cons[imx][k][j][i + 3] + -GAM * cons[imx][k][j][i - 3] +
           DEL * cons[imx][k][j][i + 4] + -DEL * cons[imx][k][j][i - 4]
        );
flux[irho][k][j][i] +=
        -dxinv1 * (
           ALP * cons[imy][k][j + 1][i] + -ALP * cons[imy][k][j - 1][i] +
           BET * cons[imy][k][j + 3][i] + -BET * cons[imy][k][j - 2][i] +
           GAM * cons[imy][k][j + 3][i] + -GAM * cons[imy][k][j - 3][i] +
           DEL * cons[imy][k][j + 4][i] + -DEL * cons[imy][k][j - 4][i]
        );
flux[irho][k][j][i] += -dxinv2 * ...
```

```
flux[imy][k][j][i] =
        -dxinv0 * (
           ALP * cons[imy][k][j][i + 1] * qup1 + -ALP * cons[imy][k][j][i - 1] * qum1 +
           BET * cons[imy][k][j][i + 2] * qup2 + -BET * cons[imy][k][j][i - 2] * qum2 +
           GAM * cons[imy][k][j][i + 3] * qup3 + -GAM * cons[imy][k][j][i - 3] * qum3 +
           DEL * cons[imy][k][j][i + 4] * qup4 + -DEL * cons[imy][k][j][i - 4] * qum4
        );
flux[imy][k][j][i] +=
        -dxinv1 * (
           ALP * cons[imy][k][j + 1][i] * qvp1 + -ALP * cons[imy][k][j - 1][i] * qvm1 +
           ALP * q[qpres][k][j + 1][i] + -ALP * q[qpres][k][j - 1][i] +
           BET * cons[imy][k][j + 2][i] * qvp2 + -BET * cons[imy][k][j - 2][i] * qvm2 +
           BET * q[qpres][k][j + 2][i] + -BET * q[qpres][k][j - 2][i] +
           ... );
```

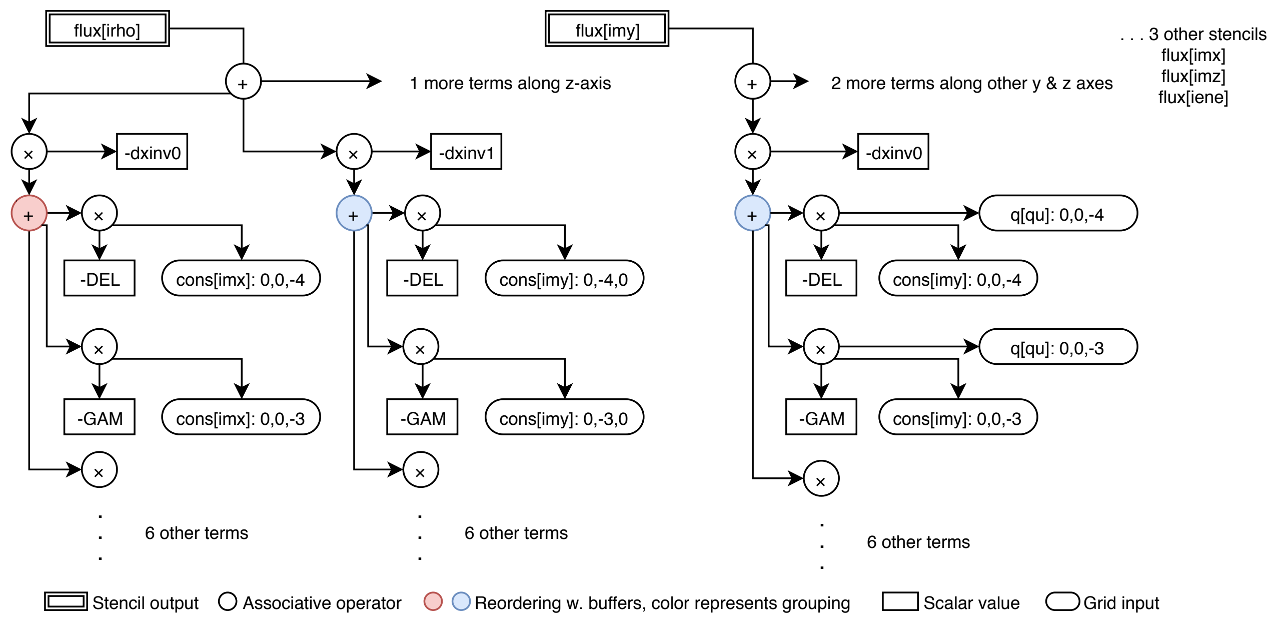


Figure 1: CNS's hypterm stencil excerpt (top) and derived expression trees (bottom).

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 (ASE) [3, 10], as the backend compiler can more easily detect such 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

```
for (j = tj; j < tj + 4; ++j)
for (i = ti; i < ti + 4; ++i) {
  c = In[j][i] * coeff[0]
    + In[j][i+1] * coeff[1]
    + In[j][i-1] * coeff[2]
    + In[j+1][i] * coeff[3]
    + In[j-1][i] * coeff[4];
  Out[j][i] = c * vel[j][i];
}
```

(a) Stencil in a tiled region.

```
float buf[4][4];
/*
* buf computed using vector scatter
*   as in Figure 5.
*/
// Vectorization directive
for (j = tj; j < tj + 4; ++j)
for (i = ti; i < ti + 4; ++i) {
  Out[j][i] = buf[j-tj][i-ti] * vel[j][i];
}
```

(b) Split stencil with buffer to expose reuse.

```
  # Declarations
  i = Index(0) ...
  In = Grid("In", 2) ...
  coeff = [ConstRef('coeff[0]'), ...]

  c = In(i,j) * coeff[0] + In(i+1,j) * coeff
   [1] + In(i-1,j) * coeff[2]) + In(i,j+1) *
   coeff[3] + In(i,j-1) * coeff[4]

  Out(i,j).assign(c)
```

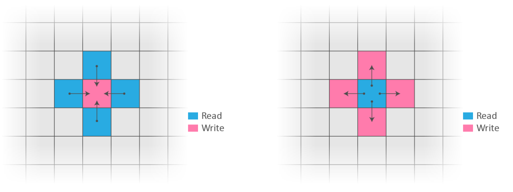
(c) Input to the code generator: `kernel.py`

```
  // tile control loops
  for (...) // Tile starting at tj, ti
      tile("kernel.py","FLEX",(4,4), // Tile
   dimension
          ("tj","ti"));

  // iterating over all brick
  for (...) // brick index b
      brick("kernel.py","AVX512",(4,4),
          (2,4) /* Folding */,b);
```

(d) Adding kernel to C code.

Figure 2: A 5-point stencil example.



(a) Gather **(b) Scatter**

Figure 3: Gather vs. scatter operations for 5 points in 2D.

annotated with their name and offset from the iterator. In Figure 2, with iterator `[j,i]`, reference `In[j][i-1]` is represented by node `In:0,-1`. Neighboring associative operators of the same type are combined to a single operator with three or more terms, resulting in the 5-way addition in Figure 4. Observe that such DAGs can represent a broad variety of stencil codes: variable coefficients, multiple stencils, and several inputs multiplied by the same coefficient.

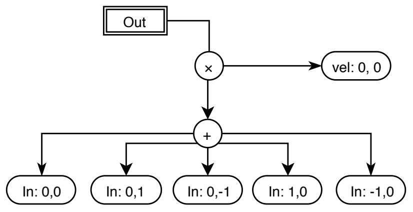


Figure 4: Expression DAG for the 5-point stencil of Figure 2.

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.

$$R(E) = \{\langle g, \vec{o}\rangle | \text{grid } g \text{ appears in } E \text{ with offset } \vec{o}\} \tag{1}$$

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 $\vec{\delta_i}$ to the offset of the corresponding term. This shift, $\vec{\delta_i}$, 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 $\Delta = \{\vec{\delta_i}\}$.

$$R(\mathbb{E}^\Delta) = \bigcup_i \{\langle g, \vec{o} + \vec{\delta_i}\rangle | \langle g, \vec{o}\rangle \in R(E_i)\} \tag{2}$$

Using the cardinality of the two sets from Equation 1 and Equation 2, we can evaluate the profitability P of a reordered expression E^Δ 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 $t_1 \geq 1$.

$$P(E, \mathbb{E}^\Delta) = \frac{|R(E)|}{|R(\mathbb{E}^\Delta)|} \geq t_1 \tag{3}$$

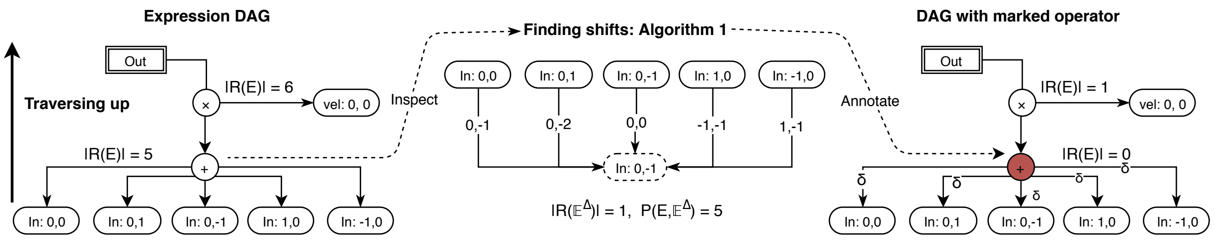


Figure 5: Illustration of profitability analysis and operator marking for the 5-point stencil in Figure 2. During DAG traversal (dashed arrow), Algorithm 1 is used to try to minimize distinct references by adding shifts to each term (numbers on edge). If the operator should be reordered (per Equation 3), the DAG is marked with shift δ_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, Δ, that minimize $|R(\mathbb{E}^{\Delta})|$.

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, δ, 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.

$$\forall E_i,\ \text{either}$$
$$\begin{cases} \forall \langle g, \vec{o} \rangle \in R(E_i), g \notin R(\mathbb{E} - \{E_i\}),\ \text{or} \\ \exists \langle g, \vec{o} \rangle \in R(E_i), \langle g, \vec{o} + \vec{\delta_i} \rangle \in R((\mathbb{E} - \{E_i\})^{\Delta - \{\delta_i\}}) \end{cases} \tag{4}$$

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

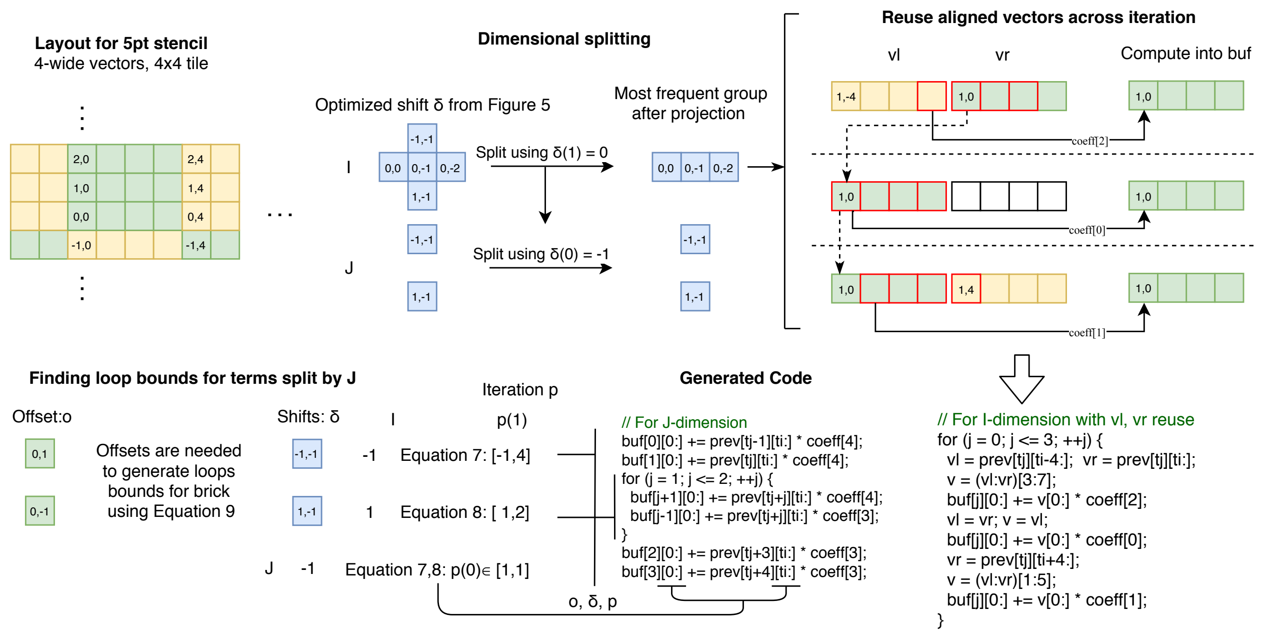
```
Initialize $\Delta = \{\vec{\delta_i}\} = \emptyset$
repeat
    for all child subexpressions $E_i$ do
        for all $\langle g, \vec{o} \rangle \in R(E_i)$ do
            for all $\langle g', \vec{o'} \rangle \in R(E_{j \neq i}), g = g'$ do
                $\vec{\delta}'_i = \vec{o'} + \vec{\delta_j} - \vec{o}$            ▷ Compute new shift
                If obtains a lower $|R(E^{\Delta})|$, $\vec{\delta_i} = \vec{\delta}'_i$
                                                                 ▷ Update $\Delta$
            end for
        end for
    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



```
// For J-dimension
buf[0][0:] += prev[tj-1][ti:] * coeff[4];
buf[1][0:] += prev[tj][ti:] * coeff[4];
for (j = 1; j <= 2; ++j) {
  buf[j+1][0:] += prev[tj+j][ti:] * coeff[4];
  buf[j-1][0:] += prev[tj+j][ti:] * coeff[3];
}
buf[2][0:] += prev[tj+3][ti:] * coeff[3];
buf[3][0:] += prev[tj+4][ti:] * coeff[3];
```

```
// For I-dimension with vl, vr reuse
for (j = 0; j <= 3; ++j) {
  vl = prev[tj][ti-4:];  vr = prev[tj][ti:];
  v = (vl:vr)[3:7];
  buf[j][0:] += v[0:] * coeff[2];
  vl = vr; v = vl;
  buf[j][0:] += v[0:] * coeff[0];
  vr = prev[tj][ti+4:];
  v = (vl:vr)[1:5];
  buf[j][0:] += v[0:] * coeff[1];
}
```

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.

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 of the two measures after the grouping and compare against a global threshold t_2 representing the architectural constraint, as in Equation 5, where $\mathbb{E}$ represents the group of operators' expressions and $|B|$ is the number of buffers in the current group. Merging can be attempted at most twice for each buffer. This results in a similar complexity of Algorithm 1 where $n = N$, the number of reads in the original stencil code. Since $N > 1, m < N$, we have a total complexity of $O(N^4 \log N)$. As noted previously, for stencils whose associative operator has only one offset for each term, this complexity reduces to $O(m^2 + N)$.

$$|R(\mathbb{E}^{\Delta})| + k|B| \leq t_2 \tag{5}$$

Consider the example in Figure 1. Our approach recognizes that there is potential for reuse within the group of associative operators colored by blue. Note that adding the blue operator on the left to a group consisting of the blue operator on the right will not affect the first term of Equation 5. The equation only approaches the threshold by the extra buffer. However, when adding the associative operator colored red to this group, both terms increase and the threshold is reached faster, preventing aggressive grouping.

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, $\vec{\delta_i}$, 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 Δ 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.

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 $\vec{p}$ within the tile, we can determine the shifted loop index, the source of the scatter, by subtracting $\vec{\delta_i}$ as in Equation 6. Each of the sources will scatter the corresponding subexpression that is shifted by adding $\vec{\delta_i}$ to all the original array indices. The scatter can then walk through all such $\vec{p'}$ and compute the value for each term to the destination with $\vec{p'} + \vec{\delta_i}$.

$$\vec{p'} = \vec{p} - \vec{\delta_i} \tag{6}$$

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 destination whose offset within the tile is a multiple of the vector length on all dimensions. With this reduced set of destinations, the meaning of the calculation is preserved since every location in the block is in one of the aligned vectors.

We notice that we can generate some loops to reduce the code size, since $\vec{\delta_i}$ reflects the amount of loop shifting applied to the term. $T(d)$ represents the tile size on dimension d. We can formulate the loop bounds as in Equations 7-9. Equation 7 represents the total range of the shifted iteration space. All or part of this range can use loops while some iterations might need to be peeled from either side of this range.

$$\begin{cases} 0 - \max_i(\delta_i(d)), \\ T(d) - v(d) - \min_i(\delta_i(d)) \end{cases} \tag{7}$$

Equation 8 represents when the writes for all terms are inside the original tiled space. A loop can be generated on the tiled code for all iterations in this range.

$$\begin{cases} 0 - \min_i(\delta_i(d)), \\ T(d) - v(d) - \max_i(\delta_i(d)) \end{cases} \tag{8}$$

Equation 9 represents when all reads are inside the blocked data region. For bricks, a loop can be generated by combining Equation 8 and Equation 9.

$$\begin{cases} 0 - \min_i(o_i(d) + \delta_i(d)), \\ T(d) - v(d) - \max_i(o_i(d) + \delta_i(d)) \end{cases} \tag{9}$$

The lower left of Figure 6 shows how loop bounds are inferred for two of the terms that are split from the original stencil using Equations 7 and 8.

4.2 Optimizations

We employ two additional optimizations when generating scatter for one stage, (1) further data reuse within vector registers using aligned vector load and vector align instructions; and, (2) dimensional splitting which divides the stencil expression and calculates it along one dimension at a time [18] to further reduce peeling. These optimizations are illustrated in the upper half of Figure 6.

We noticed that more reuse can be exploited with an aligned vector load. This optimization is especially useful for bricks where a logical vector may cross the brick boundary and must be merged using vector aligns. This merging, using `alignr` intrinsics on the CPU, is happening for almost every iteration 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 Figure 6 shows that 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` intrinsics on the CPU using either shared memory or shuffle instructions, `__shfl_up` and `__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 8×8×8 or 4×4×4.

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 [9] bandwidth of 586 GB/s.

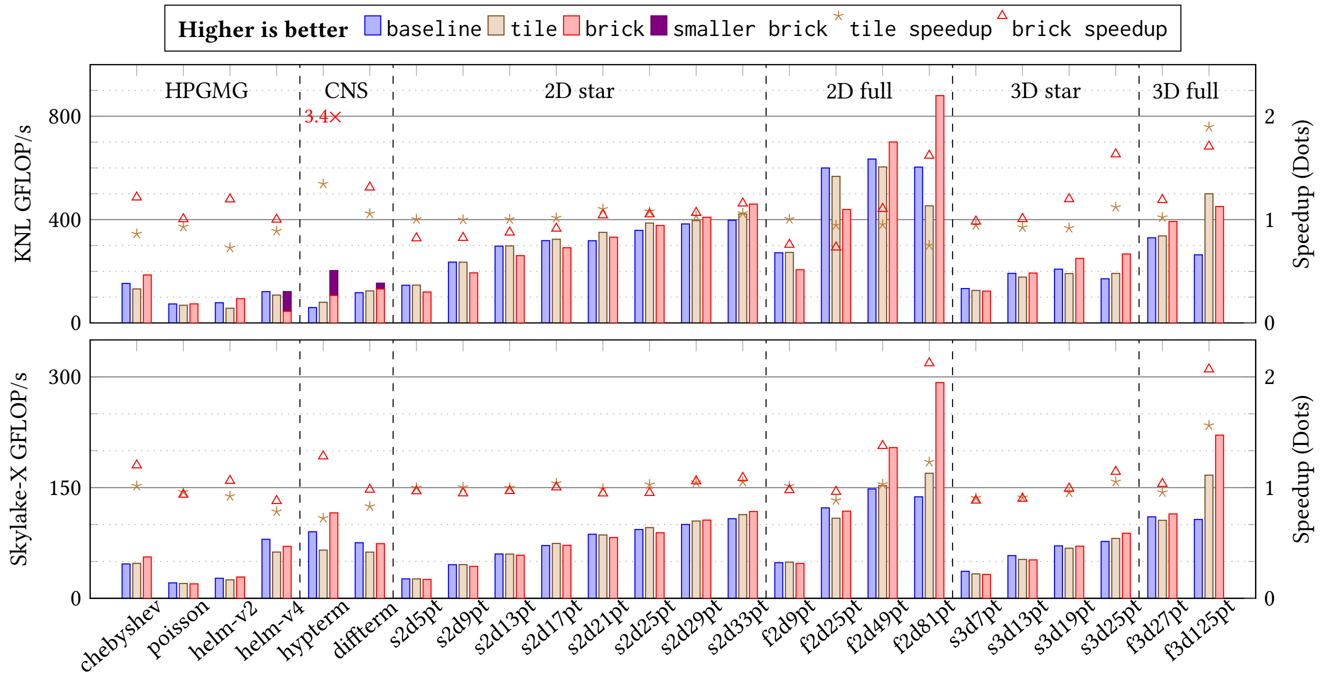


Figure 7: Performance on KNL and Skylake-X. For real-world stencil kernels, smaller blocking sizes may be beneficial because more input grids can put higher pressure on the cache. For higher-order and real-world stencils, the brick approach often provides the best speedup: On KNL, `hypterm` – 3.4× and `f3d125pt` – 1.9×. On Skylake-X, `hypterm` – 1.3× and `f3d125pt` – 2.1×.

Name	Grid	FLOPS	Name	Grid	FLOPS
Stencils from HPGMG					
chebyshev	6	39	poisson	2	21
helm-v2	7	22	helm-v4	7	115
Stencils from CNS					
hypterm	13	358	diffterm	11	415
2D Star-Shaped Stencils					
s2d5pt	2	9	s2d9pt	2	17
s2d13pt	2	25	s2d17pt	2	33
s2d21pt	2	41	s2d25pt	2	49
s2d29pt	2	49	s2d33pt	2	49
2D Full Stencils					
f2d9pt	2	17	f2d25pt	2	49
f2d49pt	2	97	f2d81pt	2	161
3D Star-Shaped Stencils					
s3d7pt	2	13	s3d13pt	2	25
s3d19pt	2	37	s3d25pt	2	49
3D Full Stencils					
f3d27pt	2	53	f3d125pt	2	249

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.

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

[1]We use 32 tiles for a total of 64 cores in all our experiments to isolate system overhead.

in under two seconds. Further discussion on the relationship between the brick library, the code generator, and our choices for these parameters is discussed in Section 5.5.

5.3 CPU Performance: KNL and Skylake-X

Figure 7 shows performance in GFLOP/s for the 24 stencils in our experiment, running on KNL and Skylake-X. We compare the performance of three code versions:

- a `baseline` version where the stencils are expressed as a gather. The stencils are tiled on all dimensions and parallelized using OpenMP. 2D stencils are parallelized using threads on the outermost tile control loop (J-dimension). 3D stencils are parallelized using threads on the two outermost tile control loops (K-, J-dimensions). The innermost tiled loop (I-dimension) is vectorized using `#pragma omp simd`. For 3D stencils we use three different tile sizes: 4×4×4, 4×4×8, and 8×8×8. For 2D stencils we use two sizes: 8×8, and 4×8. This version reports the best performance out of these tile sizes.
- a `tile` version, that uses the same thread schedule and tiling as `baseline` but applies our code generator. Only reusing aligned loads is not applied as it attempts to be more general by keeping the array-format references. Vectorization is done using `#pragma omp simd`.
- a `brick` version, where fine-grained data blocking is used [44] with our code generator. The thread schedule is kept the same, and vectorization also uses the OpenMP `simd` directive. Aligned data loads and merge are implemented using instrinsics to load vectors across brick boundaries. Reusing aligned load is applied. Here we try to separate the effect of different brick sizes by using `brick` to denote the performance for sizes 8×8×8 and 8×8 and use `smaller brick` to represent other smaller sizes presents in the `baseline`.

The `baseline` and `tile` are compiled with either `-O2` or `-Ofast`, whichever achieves the best performance. The brick version is compiled using `-O2` only.

The results for the two CPU architectures in Figure 7 suggest several observations. The trend of performance relative to the shape and size of the stencil can be shown in the synthetic stencils. For star-shaped stencils, both 2D and 3D, the relative performance of `tile` to the `baseline` increases with larger stencil diameters due to more temporal reuse. For lower order stencils, `brick` is slower than both other versions due to indexing overhead. However, since higher order stencils exhibit much higher temporal reuse, `brick` becomes the fastest version due to both the data blocking from bricks and operator reordering.

For the full stencils, on KNL, `tile` is able to obtain speedup on 3D but not on 2D stencils. This is due to 3D stencils exhibiting much higher reuse for each loaded input. Due to the alignment constraints, for each element read, `f3d125pt` is able to scatter to at most 25 points, while for `f2d81pt` it can only have up to 9 points.

Although `tile` is often slower than `brick`, it is faster for `f3d125pt` on KNL. Here, the generated code size becomes the bottleneck for `brick` because Equation 9 limits the region that can use loops. For `tile`, the code size for the entire kernel including outer loops is around 34KB for 4×4×8, which is when `tile` gives the best performance. Due to the extra indexing calculation and much wider 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 `hypterm`. As noted previously, KNL threads 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 I 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 `warpsize`, 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

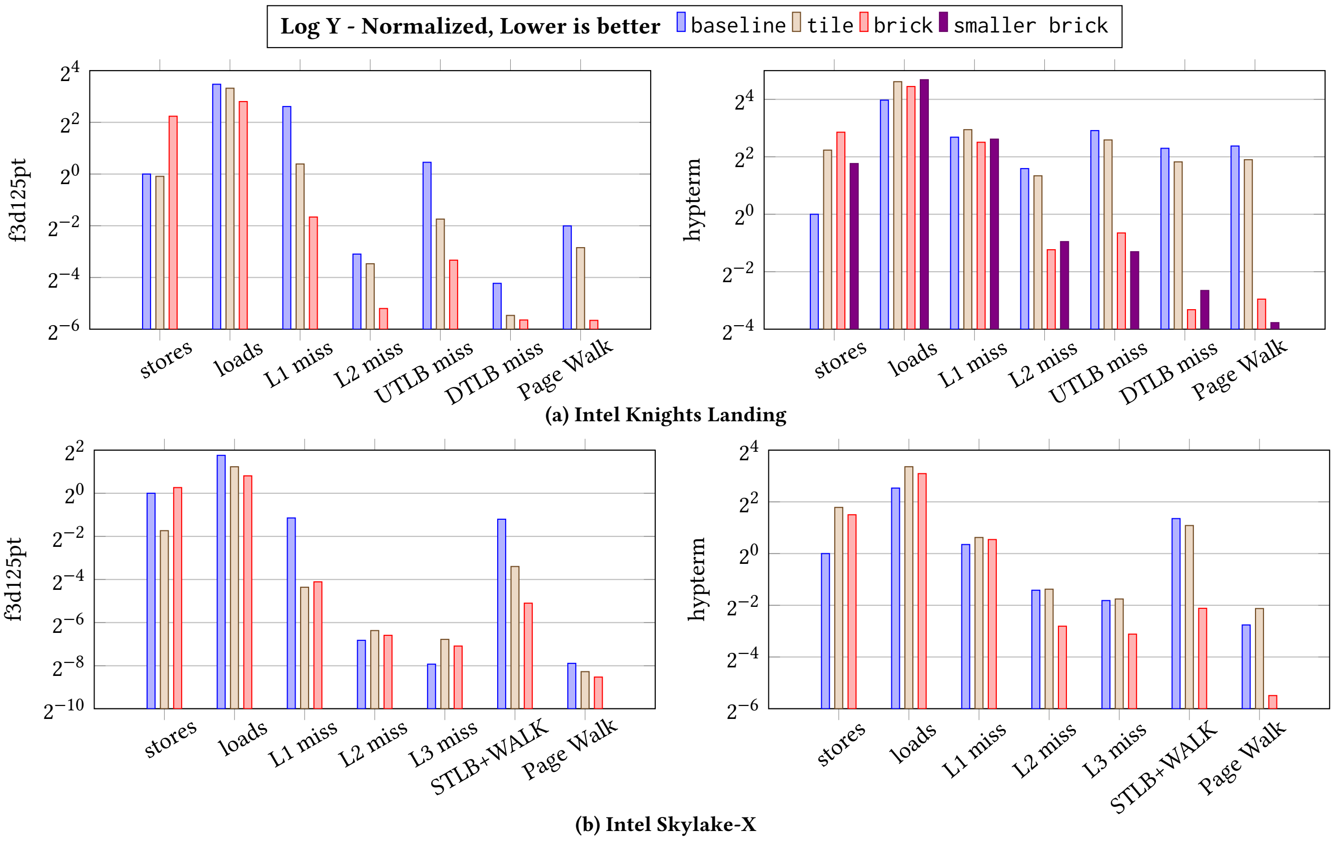


Figure 8: Profiling metrics on KNL and Skylake-X. Each bar is the raw counter value, normalized to the `baseline` total number of stores. The generated code provides a dramatic reduction in cache and TLB misses and page walks. On KNL, L1 L2 and TLB misses are reduced as much as 19×, 7×, and 49× respectively. 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.

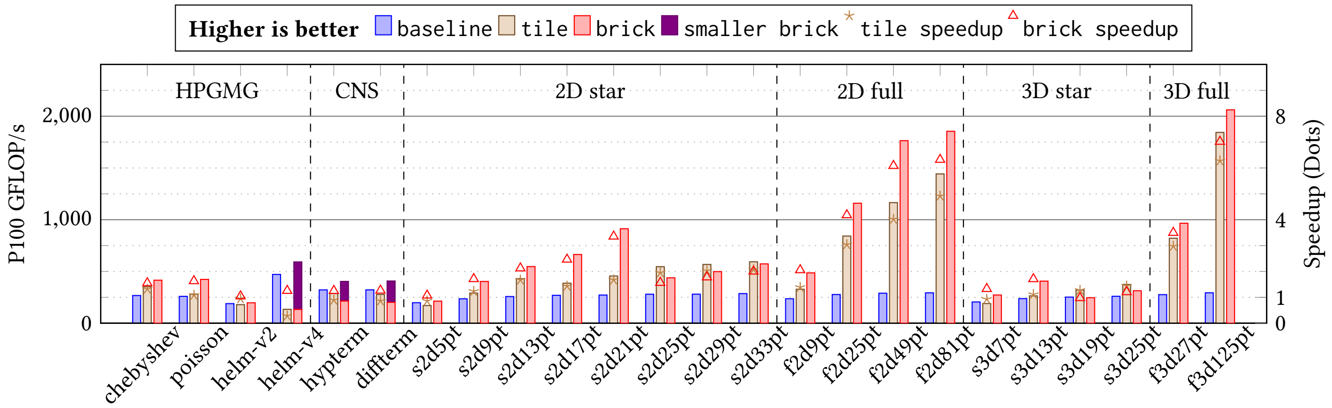


Figure 9: Performance on NVIDIA P100. Brick code generation produces the best performance for many of the stencils. For synthetic stencils, using either 32 threads or full block results has little effect, but smaller block sizes reduce cache pressure for real stencils. Bricks speedup many stencils, for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

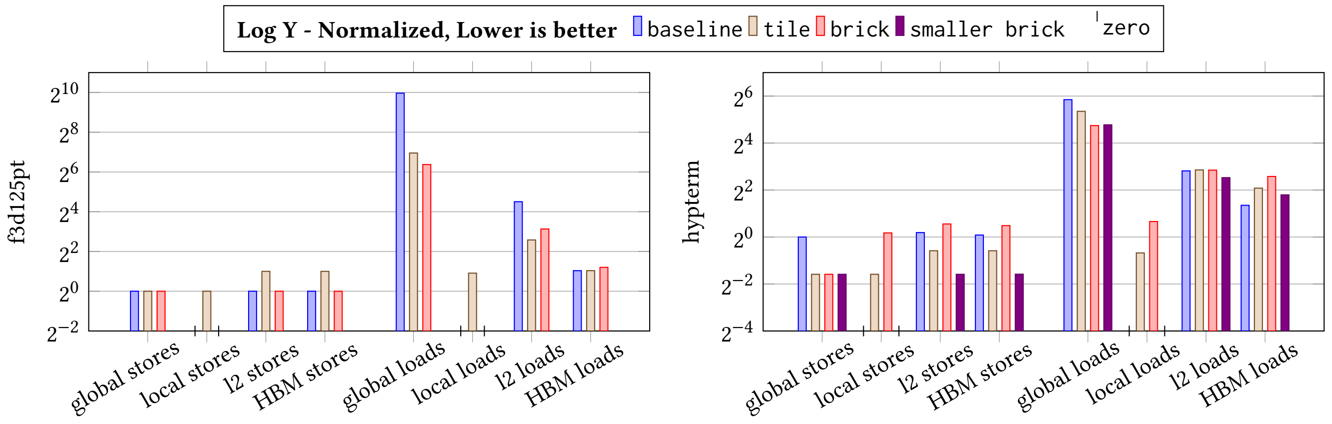


Figure 10: Profiling metrics on P100. Each bar represent transaction count normalized to the total number of global stores of `baseline`. `brick` version reduces global loads by up to 12× and L2 loads by up to 2.6× implying much better locality.

We used NVProf to obtain several metrics related to memory performance for the `f3d125pt` and `hypterm` stencils in Figure 10. Our code generator improves register and cache reuse significantly; the number of global loads are reduced by as much as 12×. Higher cache reuse results in lower L1 pressure that contributes to reduction in the volume of global load traffic seen at L2. Some of the generated code temporarily stages values in local memory, which can be identified by the local load and store metric (L-Load/Store). The `tile` code also shows higher L2 traffic. The performance impact of these increases is significantly outweighed by the higher global and L2 loads of the `baseline` code. `hypterm`, one of the more complex stencils, exhibits increased register and cache pressure, resulting in extra local load and stores and HBM traffic even when using bricks. However, with a reduced brick size and vector folding, this effect is completely eliminated. It is possible that TLB behavior is improved on the GPU as it was on the CPU, but such metrics are unavailable in NVProf.

5.5 Discussion

In this subsection, we consider the performance impact of varying the configuration of the compiler and code generator to tease out contributions of different aspects of our approach. Suppose, for example, the brick library were used without the vector code generation. The brick library uses template expansion for address calculation and incorporates indirection to represent neighboring bricks; consequently, without code generation, backend compilers will introduce redundant index calculations and cannot vectorize on CPUs. In fact, stencils in the naive brick library version might even be slower than the `baseline`. For the more compute-intensive stencils such as the `125pt`, our code generator provides an 18× speedup when compared to the naive brick library version on KNL. Even on P100, where these issues are less pronounced, the fully optimized code is 7.5× faster.

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 using 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 `hypterm`, 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. 2×2×4 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× on a Intel Knights Landing (Xeon Phi) processor, up to 1.3× on Intel Xeon Skylake-X, and up to 1.6× 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

[1] 2016. High-Performance Geometric Multigrid. http://hpgmg.org
[2] Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. 2009. 3D Seismic Imaging Through Reverse-time Migration on Homogeneous and Heterogeneous Multi-core Processors. *Sci. Program.* 17, 1-2 (Jan. 2009), 185–198. https://doi.org/10.1155/2009/382638
[3] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.* IEEE, 313–323.
[4] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS).* https://doi.org/10.1109/IPDPS.2011.70
[5] Kaushik Datta. 2009. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms.* Ph.D. Dissertation. EECS Department, University of California, Berkeley.
[6] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Rev.* 51, 1 (2009), 129–159.
[7] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *Supercomputing (SC).*
[8] Raúl De La Cruz, Mauricio Araya-Polo, and José María Cela. 2010. Introducing the Semi-stencil Algorithm. In *International Conference on Parallel Processing and Applied Mathematics: Part I (PPAM).* 11.
[9] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2. 0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *International Conference on High Performance Computing.* Springer, 489–507.
[10] Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th international conference on Supercomputing.* ACM, 65–77.
[11] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss. 2000. Cache Optimization for Structured and Unstructured Grid Multigrid. *Elect. Trans. Numer. Anal* 10 (2000), 21–40.
[12] Matthew Emmett, Weiqun Zhang, and John B Bell. 2014. High-order algorithms for compressible reacting flow with complex chemistry. *Combustion Theory and Modelling* 18, 3 (2014), 361–387.
[13] M. Frigo and V. Strumpen. 2005. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proc. ACM International Conference on Supercomputing (ICS).*
[14] P. Ghysels, P. Kosiewicz, and W. Vanroose. 2012. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra with Applications* 19, 2 (2012), 253–267.
[15] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. 2011. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction.* Springer, 225–245.
[16] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *International Conference on Supercomputing (ICS).*
[17] Jagan Jayaraj. 2013. *A strategy for high performance in computational fluid dynamics.* Ph.D. Dissertation. University of Minnesota.

[18] Jagan Jayaraj, Pei-Hung Lin, Paul R Woodward, and Pen-Chung Yew. 2014. CFD builder: A library builder for computational fluid dynamics. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 1029–1038.

[19] Markus Kowarschik and Christian WeiSS. 2001. DiMEPACK - A Cache-Optimized Multigrid Library. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), volume I*.

[20] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI)*.

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

[22] Paulius Micikevicius. 2009. 3D Finite Difference Computation on GPUs Using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. 6.

[23] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[24] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register Optimizations for Stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 168–182. https://doi.org/10.1145/3178487.3178500

[25] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 46, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291718

[26] G. Rivera and C. Tseng. 2000. Tiling Optimizations for 3D Scientific Computations. In *Supercomputing (SC)*.

[27] S. Sellappa and S. Chatterjee. 2004. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications* 18, 1 (2004), 115–133.

[28] Y. Song and Z. Li. 1999. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[29] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 65–76.

[30] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *ACM symposium on Parallelism in algorithms and architectures*.

[31] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. 2016. *TiDA: High-Level Programming Abstractions for Data Locality Management*. Springer International Publishing, Cham, 116–135.

[32] Jerry E. Watkins, Joshua Romero, and Antony Jameson. 2016. Multi-GPU, Implicit Time Stepping for High-order Methods on Unstructured Grids. In *46th AIAA Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics. https://doi.org/10.2514/6.2016-3965

[33] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. In *International Computer Software and Applications Conference*. https://doi.org/10.1109/COMPSAC.2009.82

[34] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. 2008. Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*.

[35] Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. 2011. Extracting ultra-scale Lattice Boltzmann performance via hierarchical and distributed auto-tuning. In *Supercomputing (SC)*.

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

[37] S. Williams, A. Watterman, and D. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Commun. ACM* (April 2009).

[38] D. Wonnacott. 2000. Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. In *Proc. Interational Conference on Parallel and Distributed Computing Systems*.

[39] C. Yount. 2015. Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 865–870.

[40] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK-yet Another Stencil Kernel: A Framework for HPC Stencil Code-generation and Tuning. In *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC (WOLFHPC '16)*. 10.

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

[42] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox. 2017. Snowflake: A Lightweight Portable Stencil DSL. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 795–804.

[43] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *International Symposium on Code Generation and Optimization (CGO)*.

[44] T. Zhao, S. Williams, M. Hall, and H. Johansen. 2018. Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 59–70. https://doi.org/10.1109/P3HPC.2018.00009

[45] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical overlapped tiling. In *Proc. International Symposium on Code Generation and Optimization (CGO)*.

[46] Gerhard Zumbusch. 2013. Vectorized Higher Order Finite Difference Kernels. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing (PARA'12)*. Springer-Verlag, Berlin, Heidelberg, 343–357. https://doi.org/10.1007/978-3-642-36803-5_25