# A Framework for Enhancing Data Reuse via Associative Reordering

Kevin Stock[1] Martin Kong[1] Tobias Grosser[2]
Louis-Noël Pouchet[3] Fabrice Rastello[4] J. Ramanujam[5] P. Sadayappan[1]

[1] The Ohio State University {stockk,kongm,saday}@cse.ohio-state.edu
[2] École Normale Supérieure tobias.grosser@inria.fr
[3] University of California Los Angeles pouchet@cs.ucla.edu
[4] Institut National de Recherche en Informatique fabrice.rastello@inria.fr
[5] Louisiana State University jxr@ece.lsu.edu

## Abstract

**The freedom to reorder computations involving associative operators has been widely recognized and exploited in designing parallel algorithms and to a more limited extent in optimizing compilers.**

**In this paper, we develop a novel framework utilizing the associativity and commutativity of operations in regular loop computations to enhance register reuse. Stencils represent a particular class of important computations where the optimization framework can be applied to enhance performance. We show how stencil operations can be implemented to better exploit register reuse and reduce load/stores. We develop a multi-dimensional retiming formalism to characterize the space of valid implementations in conjunction with other program transformations. Experimental results demonstrate the effectiveness of the framework on a collection of high-order stencils.**

## 1. Introduction

It is well known that the associativity of operations like addition and multiplication offer opportunities for reordering of operations to enable better parallelization. It forms the fundamental basis of many parallel algorithms, e.g., efficient implementation of scan operations on GPUs [13, 39]. Associativity is also exploited by optimizing compilers for parallelization, e.g., work-shared loops with the reduction clause in OpenMP, and for vectorization, e.g., to utilize vector-SIMD instruction sets like SSE, AVX, Altivec, etc. with reduction operations. In this paper, we show how excessive data traffic caused by poor register reuse in executing repetitive stencil operations on a multi-dimensional array can be dramatically reduced through reordering of associative/commutative operations.

The transformation framework we develop is of particular significance for high-order stencil computations. Stencils are a key computational pattern arising in numerous application domains where weighted sums of values at a set of neighboring points are computed over a regular multi-dimensional grid. High-order stencils involve weighted averages over multiple neighboring points along each dimension. They are at the core of several large-scale scientific codes, such as those using Lattice Boltzmann methods (e.g., fluid flow simulation) [43], Finite-Difference Time Domain methods (e.g., seismic wave propagations, electromagnetic radiations) [24], image processing (e.g., edge detection) [35], and others. Overture [26] is a toolkit for solving partial differential equations over complex geometry, and uses high-order approximations for increased accuracy, leading to high-order stencil computations. Similarly, the Chombo library [8] uses high-order stencil operations in discretizing high-order derivatives. Additionally, low-order stencils can be "unrolled" along a time dimension to create high-order stencils. As seen in Sec. 5, when using our techniques the rate of stencil applications is sustained as the order increases, which can enable efficient time tiling.

Previous work has shown that pattern-specific compilation strategies for stencils are needed to address a variety of stencil specific performance bottlenecks, including parallelization, communication, data reuse, etc. [11, 18–20, 35, 38]. However no previous work on compiler optimization has addressed higher-order stencils and the unique opportunity for data locality optimization from associativity and commutativity in a systematic way.

Unlike simple low-order stencils, high-order stencils feature a very high arithmetic intensity, i.e., the ratio of arithmetic operations to the number of distinct data elements accessed. It is generally the case that computations with a very low arithmetic intensity are memory bandwidth bound and achieve low compute performance, while codes with high arithmetic intensity and potential for parallel execution achieve high performance. Although high-order stencils satisfy this property, performance surprisingly decreases as the order of the stencil, and correspondingly the arithmetic intensity, increase. This apparently counter-intuitive behavior is a consequence of increasing register pressure that leads to excessive spilling. The paper makes the following contributions:

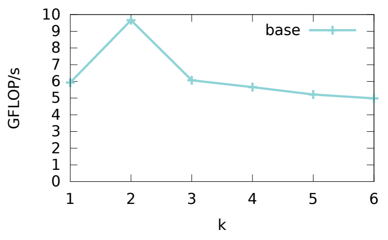
- It identifies a performance issue with the register reuse for an important class of stencil computations and develops an effective and general solution to this problem.
- It develops the first compiler framework, to our knowledge, to exploit associativity/commutativity of operations to enhance data locality in a class of iterative loop computations.
- It develops a compile-time approach to prune the space of possible associative/commutative reordering of operations.
- It demonstrates the benefits of our proposed framework, showing substantial performance improvement over a range of stencil benchmarks.

The paper is organized as follows: Sec. 2 uses an illustrative example to show the impact of associative reordering on register pressure and I/O operations for high-order stencils. Sec. 3 formalizes the program transformation framework and Sec. 4 presents the register optimization approach. Experimental results are presented in Sec. 5, before discussing related work.

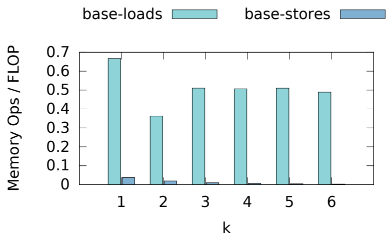
*PLDI '14*, June 9-11 2014, Edinburgh, UK.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594342

```
for (i=k; i<N-k; i++)
  for (j=k; j<N-k; j++) {
    OUT[i][j] = 0;
    // Compact representation shown below.
    // Loops (ii,jj) are fully unrolled for
    // each value of k generated in Fig. 1(b)
    for (ii=-k; ii<=k; ii++)
      for (jj=-k; jj<=k; jj++)
        OUT[i][j] +=
              IN[i+ii][j+jj]*W[k+ii][k+jj]; }
```

(a) 2D stencil prototype



(b) Performance of the base implementations

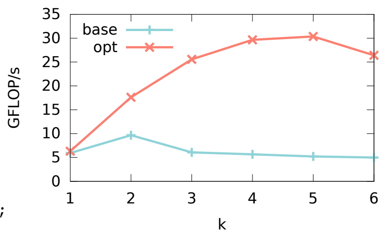


(c) Hardware counted loads and stores

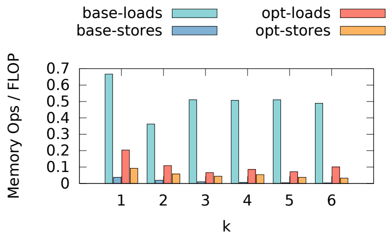
Figure 1: Implementation and performance of the base codes

```
for (i=k; i<N-k; i++)
  for (j=0; j<2*k; j++)
    { OUT[i][j+k] = 0; STMT(-k, -k+j) }
  for (j=2*k; j<N-2*k; j++)
    { OUT[i][j+k] = 0; STMT(-k, k) }
  for (j=N-2*k; j<N; j++)
    STMT(j-N+k+1, k)
where STMT(lb,ub) is:
  for (ii=-k; ii<=k; ii++)
    for (jj=lb; jj<=ub; jj++)
      OUT[i][j-jj] += IN[i+ii][j]*W[k+ii][k+jj];
```

(a) Reordered 2D stencil prototype



(b) Base vs. reordered (opt) implementations



(c) Hardware counted loads and stores

Figure 2: Implementation and performance of the base and optimized codes

## 2. Motivating Example

We use the example in Fig. 1 to illustrate the fundamental issues addressed in this paper. The code in Fig. 1(a) is a generic convolution stencil that sweeps over a 2D array *OUT*, where at each point $(i, j)$, a weighted sum of a $n \times n$ ($n = 2 \times k + 1$) neighborhood around $(i, j)$ in array *IN* is computed using a weight matrix $W$ of size $n \times n$. Stencil computations are generally considered to be memory-bandwidth bound since their arithmetic intensity is not usually sufficiently greater than the machine balance parameter, i.e., the ratio of peak main memory bandwidth to peak computational performance [44]. However, the arithmetic intensity of a stencil is directly related to its order $k$.

A $3 \times 3$ 2D stencil, that is $k = 1$ in Fig. 1(a), involves nine multiplications and eight additions at each data point $OUT[i][j]$ assuming all weight coefficients are distinct, i.e., 17 floating-point operations. Each data element $IN[i][j]$ is used in computing nine neighboring points of *OUT* (excluding the boundary). Thus if full reuse of data elements is achieved in the last level cache, i.e., the cache capacity is greater than approximately $2 \times k \times N$ words, the total bandwidth requirement per floating-point computation would correspond to an average of one word loaded from main memory and one word stored to memory per 17 floating-point operations, i.e., 16 bytes of data transfer across the memory bus per 17 operations, giving a bytes/flop requirement below 1. The machine balance parameter for most multicore systems today is much lower, e.g., around 20 GB/s bandwidth and upwards of 100 GFLOPs peak performance giving a bytes/flop ratio below 0.25.

Next let us consider a higher order stencil. Higher order stencils arise when higher order differences are used to discretize high order derivatives in PDE solvers, for example Overture from LLNL [26]. For a convolution with a $5 \times 5$ stencil (corresponding to $k = 2$), the arithmetic intensity increases, giving a machine balance requirement of 16/50, probably still memory-bandwidth bound on current multi-core systems. However, a $7 \times 7$ stencil's machine balance requirement will be roughly half of that for the $5 \times 5$ stencil. So we can expect that as the order of the stencil increases, the computation becomes less memory-bandwidth bound. We might therefore expect that the achieved performance of the stencil code should monotonically increase with the order of the stencil. However the measured performance shown in Fig. 1(b) shows a different trend. While performance does indeed increase from a $3 \times 3$ ($k = 1$) stencil to a $5 \times 5$ ($k = 2$) stencil, there is a drop in performance as we further increase the order of the stencil. Performance was tested on an Intel i7-4770k processor using code compiled with ICC -O3, using $N = 12000$. For each value of $k$, a distinct C code is generated and compiled. This C code is obtained by fully unrolling the $ii$ and $jj$ loops so as to have the standard implementation with all neighbor points accumulated in a single statement. The same approach is used to generate Fig. 2(b) from the template in Fig. 2(a).

The problem is that while the burden on the memory subsystem is reduced for higher order stencils, register pressure worsens. For a $3 \times 3$ stencil, as explained in greater detail later in the paper, six registers are needed to achieve three-way register reuse in the direction of stencil movement (the $j$ loop). For a $5 \times 5$ stencil, there is an opportunity to achieve a 5-way register reuse, but 20 registers are required to implement this reuse. Greater reuse is achieved at the cost of some register spilling and the overall performance improves. Hardware counters in Fig. 1(c) show the total number of load instructions executed per FLOP decreases when we go from $k = 1$ ($3 \times 3$ stencil) to $k = 2$ ($5 \times 5$ stencil).

A $7 \times 7$ stencil offers the potential for 7-way register reuse, but the register pressure is over 42. The net result is that the code generated by the Intel ICC compiler for this case is less effective in exploiting register reuse, as shown by the hardware counter measurements in Fig. 1(c). Performance continues to drop as we further increase the stencil order, while greater arithmetic intensity implies performance should be improving.

In this paper, we develop a solution to the increased register pressure for higher order stencils, by exploiting the freedom to reorder the associative/commutative operations of the stencil computations. The weighted contributions from the neighboring points can be accumulated in any order. However, changing just the order of operations among the set of accumulations to a single element of *OUT* is not useful. Instead, we need to judiciously interleave sten-

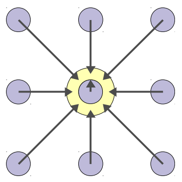
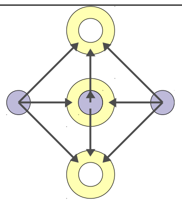
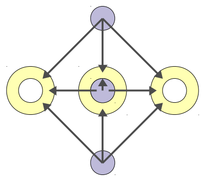
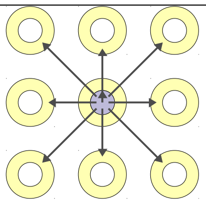
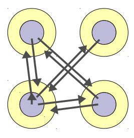
| Variant | Gather-Gather | Gather-Scatter | Scatter-Gather | Scatter-Scatter | Compact |
|---|---|---|---|---|---|
| Diagram |  |  |  |  |  |
| $IN_{loads}$ | $n$ | 1 | $n$ | 1 | $\lceil n/2 \rceil$ |
| $OUT_{loads}$ | 0 | $n-1$ | 0 | $n-1$ | $\lfloor n/2 \rfloor$ |
| $OUT_{stores}$ | 1 | $n$ | 1 | $n$ | $\lceil n/2 \rceil$ |
| $REGS$ | $n^2-n+2$ | $n+2$ | $n+2$ | $n^2-n+2$ | $2 \cdot (\lceil n/2 \rceil)^2+2$ |

Table 1: Expected IO and register pressure of different retiming variants for the 2D ($n \times n$) stencil of Fig. 1 ($n = 2k+1$)

cil accumulations to multiple target elements. A transformed code template, representative of the kind of operation reordering generated by our framework, is shown in Fig. 2(a).

In contrast to the original code in Fig. 1(a), which may be seen as an *all-gather* stencil (all contributions to a target element are gathered together in a single set of operations), the code in Fig. 2(a) may be viewed as a *scatter-gather* stencil. The code shown in Fig. 2(a) performs exactly the same set of operations as the code in Fig. 1(a), but in a different order of interleaving initialization and accumulation to elements of $OUT$. Within the loop over rows ($i$), the code contains a prologue loop that performs updates to some of the left columns of $OUT$, the main middle loop that performs the bulk of updates, and a final epilogue loop that performs updates to some of the right columns of $OUT$. Considering a $3 \times 3$ stencil, for a given point $(i, j)$ of the outer two loops, here we have a $3 \times 1$ "read-set" of three elements from $IN$ each making contributions to each element of a $1 \times 3$ "write-set" of $OUT$. For a $n \times n$ stencil, the transformed version involves a $n \times 1$ read-set updating a $1 \times n$ write-set in an all-to-all fashion. The main benefit is that now the register pressure is approximately $n$ registers instead of $n^2$. The performance of the modified stencil is shown in Fig. 2(b), and is compared with the base code over which it shows substantial performance improvement. Fig. 2(c) shows hardware counters for the modified code. It can be seen that the loads/flop ratio is considerably lower than the original code, while the ratio of stores/flop is slightly higher. In essence, a highly asymmetric *all-gather* stencil with minimal stores but many more loads has been transformed into a more balanced stencil that performs more stores, but is able to achieve a substantial reduction in the number of loads.

Consider again the stencil code in Fig. 1(a). A rectangular iteration space over the range $[k : N-k-1][k : N-k-1]$ is traversed, applying a stencil operation at each point in that space. The stencil can be characterized by a read-set and a write-set. For the version of code in Fig. 1(a), the read-set has an offset range of $[-k : k][-k : k]$ around $[i][j]$, while the write-set is a single point, with offset range $[0 : 0][0 : 0]$. In general, the stencil can be viewed as a many-to-many set of edges from points in the read-set to points in the write set. The stencil in Fig. 1(a) is an *all-gather* or *gather-gather* (gather in both dimensions) stencil, i.e., at iteration point $[i, j]$, we read from $IN[i-k : i+k][j-k : j+k]$ and write to $OUT[i][j]$. For the *all-scatter* or *scatter-scatter* stencil, at iteration $[i, j]$, we read from $IN[i][j]$ and write to all points in $OUT[i-k : i+k][j-k : j+k]$.

For the *gather-gather* stencil, the total computation may be viewed as a set of edges in a bipartite graph from $IN[0 : N-1][0 : N-1]$ to $OUT[k : N-k-1][k : N-k-1]$. Any order of execution of the set of computation edges in this bipartite graph is valid. This can be done by creating an arbitrary modified stencil that has exactly the same set of edges as the original stencil, but is moved around in the Cartesian space. Consider a bipartite graph with the read-set vertices on one side and the write-set vertices on the other. Initially, for an *all-gather* stencil, we have $n \times n$ points of $IN[-k : k][k : k]$ and a single output point $OUT[0][0]$ with an edge from each input point to the single output point. The edges can be moved around as long as the orientation is not changed, i.e., the shift between the source point on $OUT$ and the sink point on $IN$ is preserved. For example, the edge from $IN[-1][-1]$ to $OUT[0][0]$ can be shifted to go from $IN[0][-1]$ to $OUT[1][0]$ or from $IN[0][0]$ to $OUT[1][1]$ or from $IN[1][0]$ to $OUT[2][1]$, etc.

A *gather-scatter* stencil is formed by shifting the edges so that the footprint on $IN$ is only $[0][-k : k]$ but this changes the footprint in $OUT$ to $[-k : k][0]$. Many other configurations are possible; the only constraint is that all stencil edges are retained with their original orientations. Table 1 shows different stencils equivalent to the 9-point *gather-gather* stencil. The read-set vertices are as the solid purple circles and the write-set elements are the beige annuli.

The different stencil shapes differ in their register requirements as well as the number of loads and stores from memory required assuming REGS registers are available. For the *gather-gather* stencil, the write-set is a single element, all of whose updates happen in a single step. Thus a single register is needed for the write-set, and the IO cost is one store per iteration space point. The read-set has $n^2$ elements of which $n^2 - n$ will be reused at the next iteration point $[i][j+1]$. In order to achieve this reuse, $n^2 - n$ registers will be needed. At each iteration space point, a new set of $n$ input values of $IN$ will be loaded. The register requirement and the number of loads and stores are summarized in Table 1 for various equivalent stencils, including *scatter-scatter*, *gather-scatter*, *scatter-gather*, and a non-symmetric compact stencil with a read-set and write-set of four elements in a $2 \times 2$ configuration.

***Overview of the approach*** Our end-to-end optimization process involves the following steps:

1. Extract an internal representation of the input code using polyhedral compilation concepts; see Sec. 3.1.
2. Create a space of abstract scatter/gather alternatives along with different unrolling factors for the program; see Sec. 3.
3. For each point in the space, analytically compute the expected I/O per loop iteration and the expected register count needed to exploit full reuse along the loop; see Sec. 4.
4. Prune the space of candidate variants based on their arithmetic intensity relative to the original code using our analytical model; see Sec. 4.
5. For each point remaining, scatter/gather the appropriate dimension, unroll, generate C code, and perform complementary optimizations for vectorization; see Sec. 4.
6. Perform auto-tuning to find the best performing variant on the target machine; see Sec. 5.

## 3. Framework for Program Optimization

We use the Jacobi 1D stencil in Fig. 4 as the illustrating example throughout this section. It is equivalent to the code in Fig. 3 when assuming left associativity of + to evaluate the full expression, and shows a typical program input to our transformation framework.

```
    for (i = 1; i < N - 1; ++i)
S1:   OUT[i] = W[0]*IN[i-1] +
                 W[1]*IN[i] + W[2]*IN[i+1];
```

Figure 3: Jacobi 1D using a weight array `W`

```
    for (i = 1; i < N - 1; ++i) {
S1:   OUT[i]  = W[0] * IN[i-1];
S2:   OUT[i] += W[1] * IN[i];
S3:   OUT[i] += W[2] * IN[i+1];
    }
```

Figure 4: Jacobi 1D after statement splitting

```
S1: OUT[1]  = W[0] * IN[0];
S1: OUT[2]  = W[0] * IN[1];
S2: OUT[1] += W[1] * IN[1];
    for (i = 2; i < N - 2; ++i) {
S1:   OUT[i+1]  = W[0] * IN[i];
S2:   OUT[i]   += W[1] * IN[i];
S3:   OUT[i-1] += W[2] * IN[i];
    }
S2: OUT[N-1] += W[1] * IN[N-1];
S3: OUT[N-2] += W[2] * IN[N-1];
S3: OUT[N-1] += W[2] * IN[N];
```

Figure 5: Jacobi 1D after retiming (all-scatter)

Fig. 5 shows a transformed version of the code, where *multidimensional retiming* has been applied to "realign" the accesses to `IN` so that inside an iteration of loop `i`, the same element of `IN` is being accessed. This corresponds to the all-scatter version of the code. Retiming is the key concept we use to model the generalized scatter/gather transformation, and we show below some key properties to assess the legality of retiming by leveraging the associativity of the + operation.

### 3.1 Program Representation

In this work we focus on loop-based programs whose control-flow can be statically determined and represented using only affine inequalities. This class of programs is a superset of *affine programs* [15] as we do not require array index expressions to be limited to affine functions of the surrounding loop iterators. The main motivation for requiring that the control-flow can be captured using a polyhedral representation is the ease of expression and implementation of multidimensional retiming of statements, so as to achieve different locality and register pressure trade-offs.

The very first step of our framework is to convert an input program into an internal representation that is amenable to effective retiming. For maximal flexibility it is best to split a single statement as in Fig. 3 that contains multiple associative accumulation operations + into distinct statements so that we have only one accumulation operation per statement, as in Fig. 4. This enables different retiming for the operands of each accumulation. We remark that some = operation may need to be changed to +=, and vice-versa, to ensure that the first statement that updates an array element in the retimed program uses = and not +=, this is discussed in later Sec. 4. We use the following concepts to represent the program.

***Statement iteration set*** Each statement in the program, e.g., `S1`, `S2` and `S3` in Fig. 4, is associated with a set of integer points such that (1) there is one point in this set for each run-time instance of the statement; and (2) the coordinates of each point correspond exactly to the value taken by the surrounding loop iterators when that statement instance is executed. To ensure the ability to leverage polyhedral code generators such as CLooG [5] to implement the program transformation, we restrict this to a subset of $\mathbb{Z}^p$, if `S` is surrounded by $p$ loops, and this subset is bounded only using affine inequalities. For example the iteration set of `S1` in Fig. 4 is:

$$I_S : \{i \in \mathbb{Z} \mid 1 \leq i < N-1\}.$$

If in the original code the loop bound expressions are constants but not affine expressions, e.g., `i < sqrt(42) + pow(x, 42)`, they can safely be replaced by a unique parametric constant $Cst \in \mathbb{Z}$ and the equivalent loop bound in the iteration set would be `i < Cst`, which is an affine expression.

***Data accessed by a statement*** The data accessed by a particular statement *instance* is represented by a collection of access functions, one for each array access in the statement. Scalars are viewed as 0-dimensional arrays, and pointers are forbidden. Given a $n$-dimensional array reference, the associated access function is represented with a vector $\vec{f}$ with $n$ components. For example, the access `IN[i-1]` of statement `S1` is represented as: $f^{IN}_{S1} : (i-1)$ and for an access `A[i][j-2]` it is $f^{A}_{S} : (i, j-2)$. There is no restriction imposed on the components of this vector, but the precision of data dependence analysis and the computation of the data space touched by a loop iteration is impacted by the form of expressions used in $f$: if only affine expressions of the surrounding loop iterators and parametric constants are used, then exact polyhedral dataflow analysis [15] can be achieved. On the other hand, arbitrary expressions may lead to over-approximating the data dependences of the program, limiting the freedom to retime the code and the accuracy of our analytical model to compute the register pressure.

***Program execution order*** The order in which the dynamic instances of the statements are executed is specified by a scheduling function that is applied to each iteration set to obtain the timestamp at which every instance is executed. The instances will be executed according to the lexicographic ordering of the timestamps. Given a statement $S$ surrounded by $p$ iterators, the schedule $T_S$ is a vector with $2p+1$ components, such that: (1) odd components are scalars modeling the interleaving of loops and statements; and (2) even components are affine expressions of the surrounding loop iterators. In this work, we further restrict the even components to be of the form $i+\alpha$ where $i$ is the loop iterator at that particular nesting depth and $\alpha \in \mathbb{Z}$. For example, the schedule of statement `S1` in Fig. 4 is $T_{S1} : (0, i, 0)$; the schedule of `S2` is $T_{S2} : (0, i, 1)$; and for `S3` it is $T_{S3} : (0, i, 2)$. The original schedule can be constructed from a simple AST representation of only loops and statements as nodes, where alternating components of the schedule represent the surrounding loop iterators and the relative position of the statement within those loops.

***Applying loop transformations*** As we have restricted the control-flow to be static and exactly captured through the iteration sets, and have restricted the execution order to be exactly captured through multidimensional affine functions, loop transformations can be applied by using *polyhedral code generation*. In essence, polyhedral code generation emits a C code that implements the order specified by the schedule functions when applied on each point of the iteration sets. By carefully managing the overlap between "transformed" iteration sets for different statements through polyhedral separation [5, 34], polyhedral code generation seamlessly produces the code in Fig. 5 simply by using the schedules $T_{S1} : (0, i-1, 0)$; $T_{S2} : (0, i, 1)$; $T_{S3} : (0, i+1, 2)$, i.e., the original schedule with only the second component updated from $i$ to either $i-1$, $i$ or $i+1$, and the iteration sets $I_{S1} = I_{S2} = I_{S3} : \{i \mid 1 \leq i < N-1\}$.

## 3.2 Multidimensional Retiming

Multidimensional retiming has been previously studied in the literature, mostly for the purpose of parallelism exposure [6, 16, 36, 45] or to fix transformations for legality [30, 41]. We take a different approach, seeking a characterization of different data reuse patterns and register pressure when applying arbitrary multidimensional retiming. In particular, we show in Sec. 4 how the formalism below can be used to analyze and optimize the register pressure for a class of stencil computations. Multidimensional retiming is in essence a shift of an iteration set by a constant quantity along one or more of its dimensions. Def. 1 captures these factors in a vector, one for each statement in the program, which together uniquely represent a generalized scatter/gather combination across all dimensions.

DEFINITION 1 (Retiming vector). *Given a statement S surrounded by p loops, and its associated schedule $T_S$. A retiming vector $\vec{r^S} \in \mathbb{Z}^p$ defines the offset quantity, for each surrounding loop, to be applied on S. The new schedule of operations for S is $T'_S = T_S + even(\vec{r^S})$ where $even(\vec{r^S}) = (0, r^S_1, 0, r^S_2, 0, ..., r^S_p, 0)$.*

By applying $\vec{r^{S1}} = (-1)$, $\vec{r^{S2}} = (0)$ and $\vec{r^{S3}} = (1)$ to the example code in Fig. 4, one obtains the code in Fig. 5. A candidate program transformation is represented by the retiming vectors associated with each statement. If unrolling is to be applied, we replicate the statements to be unrolled, properly updating their schedule to capture the expected final order of statements. In our framework all replications of the same statement will use the same retiming vector, this is only a convenience, not a requirement.

One key observation about multidimensional retiming is that one can compute seamlessly the updated access functions for each memory reference in the code after retiming *without having to generate the code*. This is because we have constrained the retiming vectors to constant offsets. This feature is key for designing efficient analytical solutions to capture data reuse patterns after transformation, without having to explicitly generate any transformed code. The updated access functions are defined as follows.

DEFINITION 2 (Access function after retiming). *Given an access function $f^S_A$, and a retiming vector $\vec{r^S}$ of dimension p. After retiming, each loop iterator $i_1, i_2, ..., i_p$ in $f^S_A$ will be replaced by $i_1 - r^S_1, i_2 - r^S_2, ..., i_p - r^S_p$, respectively, in the updated access function.*

For example, for `IN[i-1]` with access function $f^{IN}_{S1} : (i-1)$, after retiming with $\vec{r^{S1}} = (-1)$ the updated access function will become $f^{IN}_{S1} = ((i - (-1)) - 1) = (i)$.

## 3.3 Unleashing the Power of Retiming

So far in our framework we have not discussed the legality of retiming, i.e., the constraints on the retiming vectors to ensure the semantics of the program are preserved. In the general case data dependence analysis is required to ensure a retiming preserves the semantics, however, we show that by leveraging associativity of certain operations, and/or focusing on specific classes of programs, we can prove that any retiming preserves the program semantics without resorting to dependence analysis.

***Legality of retiming for affine programs*** If the entire program region to optimize fits the polyhedral compilation model, i.e., all loops and access functions are affine, determining the legality of a particular retiming can be done using polyhedral dependence analysis, which captures exactly the set of all pairs of dynamic instances which are in dependence. This dependence information can be used to test if a particular set of retiming vectors preserves the relative order of all dependent instances.

To illustrate this point, consider the example of Fig. 4. Array dataflow analysis would tell that there are only two flow dependences, one from $S1(i)$ to $S2(i)$ and one from $S2(i)$ to $S3(i)$. As a consequence, any retiming vectors $\vec{r^{S1}}$, $\vec{r^{S2}}$ and $\vec{r^{S3}}$ such that $r^{S1} \leq r^{S2} \leq r^{S3}$ lead to a legal retiming. Interestingly, for purely affine programs, one can build the convex space of all semantics-preserving affine multidimensional schedules [22, 31], fix some particular coefficients to capture the schedule shape constraints (odd dimensions are scalars, even dimensions are of the form $i + \alpha$), and project out most dimensions keeping only those corresponding to the schedule coefficients associated with the implementation of shifting. The resulting set contains all legal retiming factors for each statement and each loop.

***Non-affine programs*** In the present work we consider a superset of affine programs, i.e., we allow for the array access functions to be non-affine. This may prevent *accurate* dataflow analysis from being performed using classical polyhedral analysis techniques, but it does not prevent a naïve or conservative dependence analysis, and then using the polyhedral framework to represent the computed dependences. Indeed, because retiming is an affine transformation, legality can still be checked using existing polyhedral model based techniques. To illustrate this point, consider again the example of Fig. 4. Suppose the most conservative analysis has been used to find dependences, by looking only at array names being read and written. In that case, in addition to the actual ones, a loop-carried dependence would appear. As a consequence, any retiming vectors $\vec{r^{S1}}$, $\vec{r^{S2}}$, and $\vec{r^{S3}}$ such that $r^{S1} = r^{S2} = r^{S3}$ lead to a legal retiming.

***Associative reordering for reductions*** Reductions amount to "accumulating" the result of a set of operations to a single memory location. An example is shown in Fig. 6.

```
    for (i = 0; i < N; ++i)
      for (j = 0; j < N; ++j)
R:      A[i] += C[i][j];
```

Figure 6: Reduction code

In this code, classical dependence analysis captures a strict sequential order between each instance of statement `R`; every iteration depends on all the previous ones because the same value is being read and written at each iteration of the $j$ loop. However, in practice, reduction operators are typically associative and commutative operations. While compilers are often limited in their ability to exploit this associativity, typically because of the inherent limitations of the IEEE 754 Floating Point standard, numerous previous works have established the benefit of exploiting associative reordering of reduction operators for parallelism [6, 16, 36, 45]. For instance, OpenMP supports parallelization of reduction via user-provided information about the reduction operator and accumulator location.

Interestingly, if we remove the "artificial" dependence on the reduction loop in Fig. 6 by allowing operations to be reordered based on the associative and commutative properties of the + operator, then any sequential order for the loop iterations becomes valid. That is, we can execute the dynamic instances of the statement in the order we want, provided they are executed one at a time. This leads to defining a class of dependences that can be safely ignored in the context of multidimensional retiming of reduction operations.

DEFINITION 3 (Commutative dependences). *Let Q and R be two statements of the form $B_Q = B_Q \odot f(A_Q)$ and $B_R = B_R \odot g(A_R)$ where: $\odot$ is an associative and commutative operator; $A_*$ do not alias with $B_*$; $B_R$ and $B_Q$ cannot overlap partially, i.e., either they represent the same variable or they do not alias. Then any dependence between Q and R is said to be commutative.*

Clearly, if other dependences allow it, the order of execution of $R$ and $Q$ can be permuted. Going back to the example in Fig. 4, all dependences turn out to be commutative. In other words, exploiting associativity and commutativity of the addition implies that any set of retiming vectors leads to a valid retiming, assuming that $A$ and $B$ do not alias, therefore any scatter/gather combination is also valid for this code.

***Stencil computations as reductions*** Convolution stencils are of special interest regarding our technique because all dependences are commutative. In other words, any arbitrary retiming is valid. Stencil computations implementing convolutions typically use two arrays at each time iteration. Such computation accumulates weighted neighboring pixels to form the updated value of the current pixel. This observation is key for the application of our framework: by exploiting the associativity of the operation used to accumulate the value from the different neighbor pixels, one can rewrite certain stencil computations to fit the reduction pattern described above, and therefore enable arbitrary retimings on such computations. Fig. 5 illustrates this concept by performing a retiming that implements the all-scatter variant of the kernel. The order in which updates are performed has changed from the original code; the new order does not preserve the constraints imposed by the commutative dependences, but is valid if such dependences are ignored.

***Scatter/Gather as retiming*** The code in Fig. 5 shows the all-scatter version of a Jacobi 1D code which explicitly exploits associative reordering. In general, the scatter/gather principle is seamlessly applied on convolution stencils provided they are written in a form where each update operation is in a distinct syntactic statement. This is required because in our framework the granularity at which retiming operates is limited to syntactic statements. For example, moving to a 2D Jacobi, one can implement a scatter/gather combination by retiming the various statements along one of the spatial loop, while not retiming along the other spatial loops, as illustrated in Fig. 1.

## 4. Register Optimization Framework

The previous section provides a framework to implement any multidimensional scatter/gather as a retiming of the statements. We now use these results to create an analytical estimate of the memory traffic for a transformed code, without actually generating that code. We then describe our overall code generation framework.

### 4.1 Computing Data Reuse Across Retimed Iterations

One key objective of our framework is to exploit retiming to change the data reuse pattern across loop iterations, and implicitly the register pressure. Indeed, assuming we aim to exploit all the available reuse between consecutive loop iterations in registers, one register per data element to be reused is needed. In the following we define a framework to compute analytically the data reused between sets of loop iterations given an arbitrary retiming.

To enable a generic approach to compute the set of data elements touched by consecutive loop iterations, and therefore the set of data elements to be reused between iterations, we first define the notion of an iteration set slice. This is a generalized form of the parametric domain slice defined by Pouchet et al. [32].

DEFINITION 4 (Iteration set slice). *Given an iteration set $I_s$ modeling a statement surrounded by d loops, the slice $I_S^{\vec{p}}$ is:*

$$I_S^{\vec{p}} = \{I_S \mid i_1 = p_1 + \alpha_1, ..., i_d = p_d + \alpha_d\}$$

*where $p_i$ is a parametric constant unrestricted on $\mathbb{Z}$, and $\alpha_i \in \mathbb{Z}$.*

For example, taking $I_{S1}$ from Fig. 4 and $\vec{p} = (p_1)$, $I_{S1}^{\vec{p}} = \{(i) \mid 1 \leq i < N-1 \wedge i = p_1 \wedge p_1 \in \mathbb{Z}\}$. This set contains only one point. Two arbitrary but consecutive iterations of $i$ are modeled using $\vec{p}^1 = (p_1)$ and $\vec{p}^2 = (p_1 + 1)$ in the two slices $I_{S1}^{\vec{p}^2}$ and $I_{S1}^{\vec{p}^2}$. Each set obtained contains exactly one point, and they necessarily capture two consecutive iterations (that is, $i$ and $i+1$ for any $i$).

We can now define the data space of a reference for a particular set of iterations as follows.

DEFINITION 5 (Data space of a reference for a set of iterations). *Given an access function $f_A^S$ of dimension n in a statement S, and $I'_S \subseteq I_S$ a subset of the iteration set of S. The data space touched by this reference is:*

$$\mathcal{D}_{f_A^S, I'_S} = \{\vec{x} \in \mathbb{Z}^n \mid \vec{x} = f_A^S(\vec{p}),\ \forall \vec{p} \in I'_S\}$$

The definition of the data set that is being reused between two consecutive iterations is therefore the intersection of the data spaces at iteration $i$ and $i+1$.

DEFINITION 6 (Data reused between consecutive loop iterations). *Given a collection of k references $f_A^k$ on array A inside the same loop nest of depth d, $I_l$ the iteration set of the inner-most loop body, $\vec{p}^1 = (p_1, ..., p_d)$, $\vec{p}^2 = (p_1, ..., p_{d-1}, p_d + 1)$ with $p_k \in \mathbb{Z}$, we have:*

$$DataSpace(A, I_l^{\vec{p}^1}) = \bigcup_i \mathcal{D}_{f_A^i, I_l^{\vec{p}^1}}$$
$$DataSpace(A, I_l^{\vec{p}^2}) = \bigcup_i \mathcal{D}_{f_A^i, I_l^{\vec{p}^2}}$$
$$Reuse(A) = DataSpace(A, I_l^{\vec{p}^1}) \bigcap DataSpace(A, I_l^{\vec{p}^2})$$

*And we have $Access(A) = DataSpace(A, I_l^{\vec{p}^1})$.*

It follows that the number of distinct elements reused across consecutive inner-loop iterations is simply $\#Reuse(A)$ where # is the cardinality of a set of integer points.

When the function $f$ is affine, $\mathcal{D}_{f_A^S, I_S}$ can be computed analytically as the image of $f$ over the polyhedron $I_S$, and as $Reuse(A)$ is a union of convex sets of integer points. $\#Reuse(A)$ can be computed using tools such as the Integer Set Library [42]. However, more interestingly, when $f$ is limited to the form $f_k : i_k + \alpha_k$ where $i_k$ is a loop iterator, for all dimensions $k$ of $f$ as it is in typical stencil computations, the data space of one loop iteration can be simply computed from the values of $\alpha_k$. When focusing on affine stencil codes, we never need to resort to costly polyhedral operations to build the sets and count their number of points: the data space of the stencil for $i$ and $i+1$ and the resulting intersection can be computed via a simple enumeration of the updated array access functions.

In the general case of programs with arbitrary access functions the data space may not be computable analytically. This is not a problem for our transformation framework, but only for the analytical model we use to prune the space of possible transformations and accelerate the search for a good transformed variant. We remark that computing the data space only for *some* loops in the program, instead of all loops, can be helpful to expose affine array index expressions. Intuitively, when computing the data space of one iteration of the inner-most loop, all surrounding loop iterators become *constants*. Consequently, non-affine expressions involving only those iterators can be replaced by a parameter, in a manner similar to non-affine loop bounds as shown earlier.

Finally, we remark that to compute the data reused across iterations in the presence of retiming, one simply needs to use updated access functions according to Def. 2 in Def. 6 instead of the original access functions. As the updated access functions can be computed analytically for arbitrary retiming vectors, the computability of Def. 6 is not affected by retiming.

## 4.2 Estimating Register Pressure and Data Movements

To estimate the register pressure of a computation, we first add several hypothesis: (1) we exploit all reuse only between consecutive iterations of a loop; (2) we implement reuse between consecutive iterations through rotating registers; and (3) we ignore the register cost of computing the access functions.

With these properties in mind, we define the register pressure as the count of all reused data elements, plus two registers to ensure any 3-address operation can be performed.

DEFINITION 7 (Register count for a loop iteration). *The number of registers needed to execute a loop iteration while exploiting reuse across consecutive iterations of this loop through rotating registers is:*

$$rc = \sum_{A \in Arrays} \#Reuse(A) + 2$$

Technically, more than two registers may be needed in order to also exploit intra-iteration reuse between data elements that are not reused in the next iteration. Accounting for such cases can be done by adding one register for each set $s$ of access functions which are identical but do not refer to a data element reused at the next iteration when $\#s > 1$.

The number of memory loads required is computed with a similar reasoning in mind. The number of loads corresponds to the number of data elements which are not reused between consecutive iterations, considering only access functions related to reads.

DEFINITION 8 (Load count for a loop iteration). *The number of memory loads needed to execute a loop iteration while exploiting reuse across consecutive iterations of this loop through rotating registers is:*

$$lc = \sum_{A \in ReadArrays} (\#Access(A) - \#Reuse(A))$$

Finally, the number of memory stores required is computed directly from the set of distinct memory locations written to at a particular loop iteration that are not also written during the next loop iteration.

DEFINITION 9 (Store count for a loop iteration). *The number of memory writes needed to execute a loop iteration while exploiting reuse across consecutive iterations of this loop through rotating registers is:*

$$sc = \sum_{A \in WriteArrays} (\#Access(A) - \#Reuse(A))$$

## 4.3 Optimization Search Space

The definitions of $rc$, $lc$ and $sc$ gives analytical estimates of the memory behavior of a program, under the simplification hypothesis mentioned above. To optimize an input program, we consider numerous program transformations, compute the cost in terms of memory movement and registers needed for each candidate, and prune the set of candidate transformations using these metrics.

In this work we consider multidimensional retiming as a key optimization to enable better register reuse, especially for high-order stencil computations. But to effectively exploit the available reuse, retiming must be complemented by a set of transformations impacting data reuse patterns. A strength of our approach is that for all transformations considered we can compute the value of $rc$, $lc$ and $sc$ analytically without having to generate the transformed code variants. These additional transformations are listed below.

***Loop permutation*** Loop permutation is critical to change the direction of reuse. In our implementation we only consider loop permutations of perfectly nested loops, however more powerful loop permutations can be achieved using CLooG. By limiting to perfectly nested loops we can analytically compute the effect of permuting loops $i$ and $j$ by simply replacing $i$ by $j$ and vice-versa in all access functions.

***Loop unrolling and code motion*** Unroll-and-jam is key to exploiting inter-iteration data reuse opportunities, but comes at the cost of increased register pressure. We perform a customized loop-unrolling-and-code-motion transformation, which produces the same innermost loop body for the "steady state" loop as unroll-and-jam. However, additional code motion is performed on the prologue and epilogue loops created after unroll-and-jam to further improve locality.

## 4.4 Building Program Variants and Pruning the Space

Algorithm 1 summarizes our analytical approach to compute an estimate of the cost of various candidate transformations for a program $P$, removing from the search space variants with excessive register usage or excessive memory traffic.

**Algorithm 1** Explore search space

```
function EXPLORESPACE($P$)
    $U \leftarrow$BuildAllUCMVectors($P$)
    $R \leftarrow$BuildAllRetimingVectorSets($P$)
    $P \leftarrow$BuildAllLoopPermutations($P$)
    $variants \leftarrow \{\}$
    for all $\vec{u} \in U$ do
        for all $\vec{r} \in R$ do
            for all $\vec{p} \in P$ do
                $P\prime \leftarrow$BuildUpdatedRepresentation($P$,$\vec{u}$, $\vec{r}$, $\vec{p}$)
                $AI \leftarrow$ComputeCost($P'$)
                if AboveThresold($AI$)
                    continue
                $variants \leftarrow \{variants, (\vec{u},\vec{r},\vec{p})\}$
            end for
        end for
    end for
    return $variants$
end function
```

Function `buildAllUCMVectors` builds a set of all possible unrolling factors to be evaluated, based on a user-defined range for each dimension. Function `buildAllRetimingVectorSets` builds the retiming vectors for all statements based on user-defined ranges in each dimension. Function `buildAllLoopPermutations` builds the set of all possible loop permutations. These three functions are restricted by the program dependences in the general case: not all combinations of possible unroll and code motion, permutations and retiming are legal. However as shown in Sec. 3, for convolutions when leveraging associative reordering, all such combinations are necessarily valid and therefore no dependence analysis is required.

Function `buildUpdatedRepresentation` modifies the program representation, i.e., the iteration sets and access functions, as needed to emulate the effect of the program transformation. Function `computeCost` applies Def. 7, Def. 8 and Def. 9 to the updated program representation and analytically computes the desired values. The arithmetic intensity is then computed from the number of floating point operations executed in one iteration of the inner-most loop using the updated representation, and dividing it by $lc + sc$. Function `aboveThresold` checks if the arithmetic intensity is above a user-defined threshold, which in our framework is a function of the arithmetic intensity of the original program, and if so the variant is discarded, otherwise the variant is stored in the list of candidates to evaluate using auto-tuning. We report in Sec. 5 extensive experimental results on a collection of high-order stencils. Actual parameter values used for the search space evaluation are detailed in Sec. 5.

***Pruning the space of candidate variants*** Table 2 shows the search space statistics for several 3D benchmarks for different pruning factors. We keep only the candidates with arithmetic intensity (AI) greater or equal to the original program (1×), with AI greater than 1.5× the AI of the original program, and 2× greater. Perf shows the fraction of the performance of the space-best without pruning which is achieved by auto-tuning only on the pruned space. A value of 1× means the candidate achieving the best overall GF/s is still in the space after pruning. Column #*Space* reports the search space size and #*Left* its size after pruning. The benchmark names and experimental setup is detailed in later Sec. 5.

| | | $AI = 1\times$ | | $AI = 1.5\times$ | | $AI = 2\times$ | |
|---|---|---|---|---|---|---|---|
| Bench | #Space | #Left | Perf | #Left | Perf | #Left | Perf |
| 3-d-2 | 144 | 120 | 1x | 68 | 0.96x | 28 | 0.94x |
| 3-d-2-dlt | 144 | 120 | 1x | 68 | 1x | 28 | 0.97x |
| 3-f-1 | 144 | 132 | 1x | 80 | 1x | 28 | 0.95x |
| 3-f-1-dlt | 144 | 132 | 1x | 82 | 1x | 28 | 0.99x |
| 3-f-2 | 144 | 132 | 1x | 116 | 1x | 100 | 0.94x |
| 3-f-2-dlt | 144 | 132 | 1x | 116 | 1x | 100 | 1x |

Table 2: Search space statistics for different pruning factors

Our approach for pruning is empirical. Our objective is to keep the auto-tuning time tractable, by evaluating on the target machine only a reasonably small number of variants. For instance, when considering 2D stencils from our test suite, the number of variants without pruning is 24, making pruning unnecessary. On the other hand, for 3D codes the number of variants quickly grows because of the additional possible unrolling and combinations of scatter/gather along the third dimension. Keeping only variants which achieve an estimated AI two times better than the original code leads to a performance loss of 6% over the space-optimal point, but reduces the number of variants to be evaluated during auto-tuning by up to 5×. Finally we remark that arithmetic intensity alone is not a good performance predictor: the quality of vectorization is key for performance, and machine-specific metrics relating to the SIMD execution engine must be taken into account to approximate performance. To the best of our knowledge, accurate performance predictors are extremely difficult to build, so we instead take a conservative pruning approach while still relying on auto-tuning to find the best variant.

### 4.5 Putting it all together

Each program variant consists of a different set of values for loop permutations, retiming vectors for each program statement, and unrolling factors. The transformation system has been implemented[1] using PoCC [29]. Once the polyhedral structures are extracted from the internal representation, the loop permutation and retiming of all statements corresponding to a particular variant is embodied in the scheduling functions of the polyhedral representation. CLooG is used to generate the code structure, automatically handling all possible boundary cases induced by the retiming. A syntactic pass then applies loop unrolling and code motion, followed by stripmining the unit-stride dimension to enable vectorization. Finally, a post-processing pass was written to explicitly vectorize the stripmined loops using SIMD intrinsics, and apply a limited global value numbering pass to ensure rotating registers are implemented in the final code. Algorithm 2 shows the step-by-step transformations applied to a single program variant.

`ConvertToIR` translates the input program to the representation described in Sec. 3.1. `PermuteAndRetime` applies permutation and retiming, and selects the statement with the lexicographic largest retiming vector as initialize-and-accumulate in a single step, i.e., using = instead of += wherever needed. `PolyhedralCodegen` generates the transformed C code. `Prevectorize` applies loop unrolling, code motion and stripmining. `Vectorize` generates short-vector SIMD intrinsics for each stripmined loop using the vector ISA specified in *MACHINEINFO*, and applies global value numbering to enable register reuse across iterations. Finally, the transformed source is compiled and executed, and the best performing variant is kept.

[1] http://hpcrl.cse.ohio-state.edu/wiki/index.php/HOSTS

**Algorithm 2** End-to-end algorithm

```
function OptimizeProgram($SOURCE$, $MACHINEINFO$)
    $IR \leftarrow$ConvertToIR($SOURCE$)
    $\{bestVariant, bestTime\} \leftarrow \{SOURCE, \infty\}$
    for all $(\vec{u}, \vec{r}, \vec{p}) \in$ ExploreSpace($IR$) do
        $P \leftarrow$PermuteAndRetime($IR$, $\vec{r}, \vec{p}$)
        $P \leftarrow$PolyhedralCodegen($IR$, $\vec{r}, \vec{p}$)
        $P \leftarrow$Prevectorize($P$, $\vec{u}$)
        $P \leftarrow$Vectorize($P$, $MACHINEINFO$)
        $T \leftarrow$Compile-and-Run($P$)
        if $T < bestTime$ then
            $\{bestVariant, bestTime\} \leftarrow \{P, T\}$
        end if
    end for
    return $bestVariant$
end function
```

## 5. Experimental Results

### 5.1 Experimental Protocol

***Machine*** Experimental results presented in this paper, excluding hardware counters measurements, have been obtained using a four-core Intel Core i7-4770K CPU (Haswell micro-architecture) running at 3.5GHz. It features the AVX2 SIMD instruction set including fused-multiply-add (FMA) instruction. Its theoretical peak is 112 GF/s (224GF/s if using only FMA instructions). DDR3-1600 RAM was used, the STREAM benchmark [25] has a peak performance of 17.6 GB/s on our machine. Results based on hardware counters were collected on an Intel Core i7-2600k using Intel VTune. All benchmark variants, including the reference codes, were compiled with ICC 13.1.3 using the `-std=gnu99 -Wall -O3 -fp-model fast=2 -fma -override-limits -wd13383 -xHost -openmp` flags. The `-fast` flag did not improve performance in our test suite.

***Benchmarks*** We created a set of synthetic benchmarks with varying stencil size and number of neighbors with non-zero weight to evaluate the effectiveness of our framework on a variety of cases. We generated 2D, 3D and 4D stencils, with either a diamond-shaped set of non-zero coefficients in the weight matrix (others outside the diamond are necessarily zero), or the full weight matrix being with all non-zero coefficients within a *n*-cube. Benchmarks are named xD-{d,f}-yy where *x* is the dimension and *yy* size, i.e., the furthest non-zero weights in any direction. Diamond stencils are represented with d and full stencils with f. These synthetic benchmarks are not independent of practical applications; Table 3 shows applications which some of them are commonly used in.

We also report on a subset of stencils from the *Stencil Micro-Benchmarks* [12]; We only consider those which contain enough non-zero points for our technique to be potentially profitable: Dr-prj3 is a 19-point stencil from NAS MG Benchmark [3], and also corresponds to the D3Q19 Lattice Boltzmann method used in CFD applications such as [43]; Dresid is a 21-point stencil also from [3]; Dbigbiharm is a 25-point stencil for biharmonic operator as described in [1] and is used in areas of continuum mechanics; Inoise3 is a 49-point stencil for noise cleaning as described in [17] along with the next three, which can be used as part of image pipelines, e.g. [35]; Ibiglaplace is a 97-point stencil for gradient edge detection; Izerocross is a 25-point stencil for edge detection; Inevatia is a 20-point stencil for gradient edge detection.

| Benchmark | Application |
|---|---|
| 3d-f-1 | D3Q27 Lattice Boltzmann method [43] |
| 2d-d-3 | second-order-system Upwind Schemes [4] |
| 2d-d-1 | Jacobi 5 point |
| 2d-f-1 | Jacobi 9 point |
| Drprj3 | NAS MG [3] |
| Dresid | NAS MG [3] |
| Dbigbiharm | Biharmonic operator [1] |
| Inoise3 | Noise cleaning [17] |
| Ibiglaplace | Gradient edge detection [17] |
| Izerocross | Edge detection [17] |
| Inevatia | Gradient edge detection [17] |

Table 3: Practical applications of the benchmarks

***Code Generation*** For all benchmarks we generate the following implementations. Reference: A plain C implementation of the stencil using the standard notation; no code tested was written with points rolled together as in Fig. 1. An OpenMP pragma was added to the outer-most parallel loop to enable multicore parallelism. DLT-Reference: A plain C implementation of the DLT transformed code (described below) without any explicit reuse or vectorization. Optimized: The best of all variants generated by the algorithm shown in the previous section, operating on the original layout of the data, selected with auto-tuning. DLT-Optimized: The best variant using the DLT transformed data layout.

The rotating registers [21] transformation unrolls the inner loop by the stencil size and explicitly allocates registers to avoid register copies when transferring data to the next iteration of the inner loop. In combination with the decreased register pressure from the retimed code significant improvements to spilling are possible. This optimization can be done by global value numbering [40]; the LLVM compiler has been observed to automatically apply rotating registers when the inner-loop is a gathered dimension.

An issue with the rotating registers transformation is that it unnecessarily prevents auto-vectorization in ICC because of false dependences found by the compiler. For these cases outer loop vectorization using intrinsics was implemented in the code generator to ensure all the compute hardware was utilized. ICC's inability to vectorize the code also indicates that it may be avoiding other transformations which could further improve performance of the optimized kernels, but even with these limitations substantial improvements are possible as shown below. Outer-loop vectorization comes at the cost of forcing the transposition of the innermost two loops which results in a sweep of memory with a larger stride. This can degrade performance due to TLB misses. The DLT codes are able to utilize the original loop order with vectorization.

Tiling is implemented but not tuned to cache size, it currently only exists to enable OpenMP parallelization in tandem with the retiming. However, there is spatial locality which could be better exploited in some cases through more careful tiling.

For the unroll and code motion transformation, each loop except the inner most was unrolled by either $1\times$, $2\times$, or $4\times$, although the product of all unrolling was limited to at most $8\times$. For the retiming vectors, we considered all combinations leading to either scattering or gathering along a particular dimension, for every dimensions. We set the threshold in our cost model to eliminate all candidate variants with a lower arithmetic intensity than the original code. Finally, all problems are set to sizes such that each array is approximately $1GB$.

The Dimension-Lifted Transpose by Henretty et.al. is a transformation that enables effective vectorization of stencil codes by addressing the stream-alignment conflict problem [18]. It treats a dimension of the array as a 2d array with `vector-width` rows which is then transposed, ensuring that each vector operation in the "steady state" is aligned. In the standard layout it is not possible to implement rotating registers if the inner-most loop is the dimension of vectorization since neighboring points will be shifted by their distance from the center of the stencil, and not by the vector length. However, the DLT lets us treat vectors as scalars within the stencil, i.e., neighboring points in the stencil on the unit stride dimension are neighboring aligned vectors in memory. This is desirable since it enables rotating registers and vectorization along the unit-stride dimension, which is optimal for both register and cache reuse. The DLT is tested independently of the other optimization to show results on both the standard layout and the DLT.

The code generator produced each variant of a benchmark under 3 seconds. However some instances with multiple scattered dimensions resulted in exceptionally long C code being generated, and consequently, in ICC sometimes taking over 30 minutes to compile a single function. LLVM is able to compile the largest functions produced with `-O3 -ffast-math` in just over 5 minutes.

## 5.2 Performance Results

Fig. 7 compares the maximum performance obtained by both the original stencil C code or the naïve DLT implementation(REF), with the best performing variant we have found after auto-tuning (original layout – OPT, DLT – DLTOPT), for each benchmark. The shorter segment of each bar shows the sequential performance which illustrates the benefit of the approach in a best-case scenario for the original code: the entire bandwidth available is allocated to a single computation unit instead of being shared between all cores. The bar stacked on top represents the absolute performance of the same benchmark executing in parallel across all four cores.

Fig. 9 presents a similar performance comparison for the *Stencil Micro-Benchmarks*. As many of these benchmarks are still relatively small stencils or contain many zero weight points the benefit of the retiming optimization is less significant. However, `Ibiglaplace` is an excellent example of a real world computation which can substantially benefit from the retiming operation: in combination with the DLT the retiming achieves a $3\times$ improvement over the reference code.

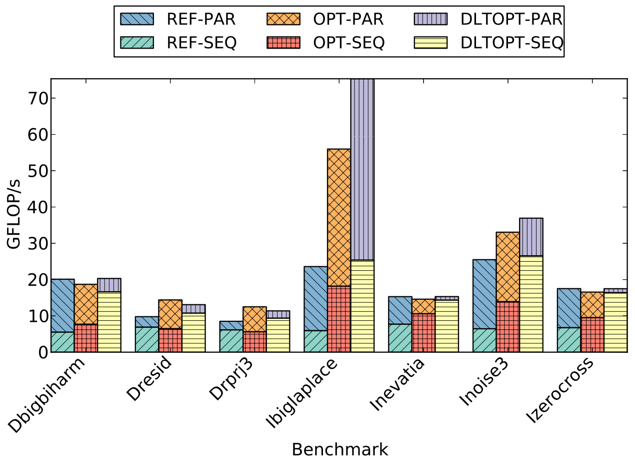


Figure 9: Maximum performance of different implementations on the *Stencil Micro-Benchmarks*

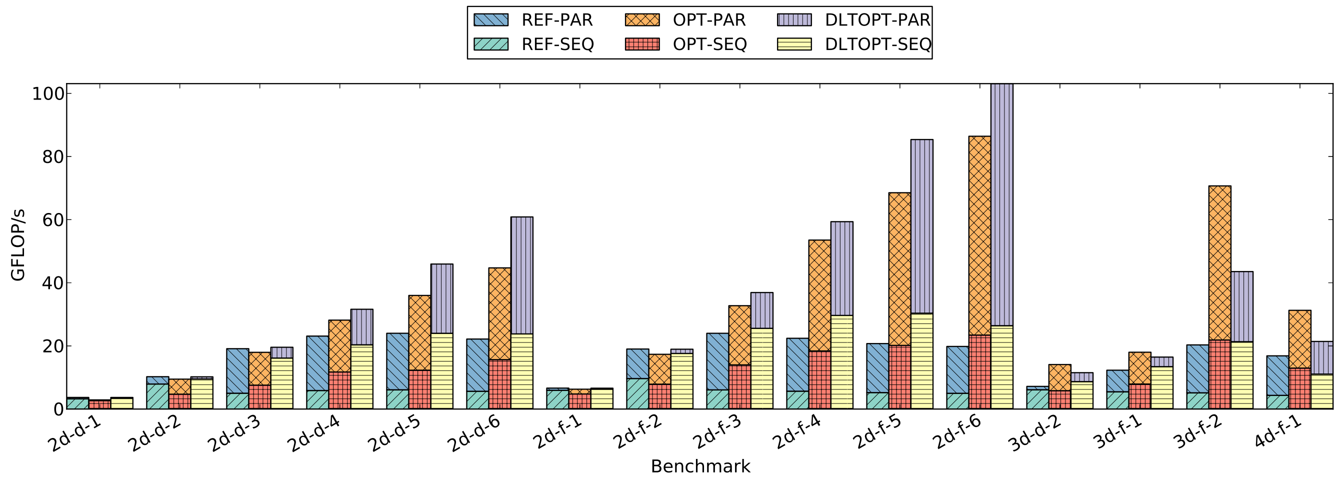


Figure 7: Maximum performance of the different implementations on the synthetic benchmarks

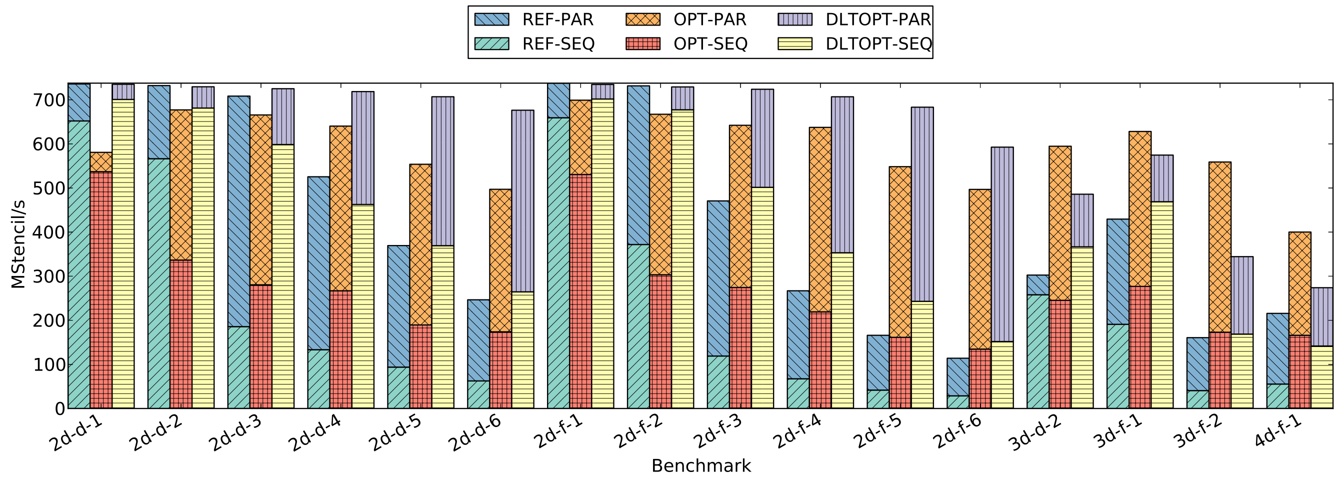


Figure 8: Performance as rate of stencil applications per second

On all problems we obtain comparable or better performance than either reference code compiled with ICC. Benchmarks with near identical performance are those with the highest proportion of zero weights and the lowest order, which is where our technique is least likely to be effective. A general trend is that the more complex the stencil is, i.e., the fewer zero weights in larger stencils, the better the performance of the OPT codes. ICC is able to very effectively optimize the smaller reference stencils, but the performance does not scale with the stencil order.

A direct comparison with the Array Subexpression Elimination (ASE) technique presented in [12] is not easy because the implementation is unavailable. However, a comparison of improvements over reference implementations for sequential codes can be made with Fig. 13 of Deitz's paper. Notable stencils to compare are Dbigbiharm which is 40% faster with ASE, whereas (using different hardware) we achieve over 3× improvement using retiming for sequential execution; and both the lbiglaplace and lnoise3 stencils which are 1.8× faster with ASE, but over 4× faster with retiming.

Fig. 8 shows performance in stencils computed per second. Based on results of the Stream benchmark, the practical peak rate for any stencil is 1100 MStencil/s. If perfect reuse of all data in the benchmarks was possible, we would expect the rate of stencils/s to remain flat until the problem becomes compute bound, however we observe that the reference codes demonstrates a rapid decrease as the stencil size increases, and since the GFLOP/s of these benchmarks is not near peak performance of the machine the codes have become artificially bound. This artificial bound is explained by Fig. 10 which shows the number of loads and stores retired for each benchmark as reported by Intel VTune.

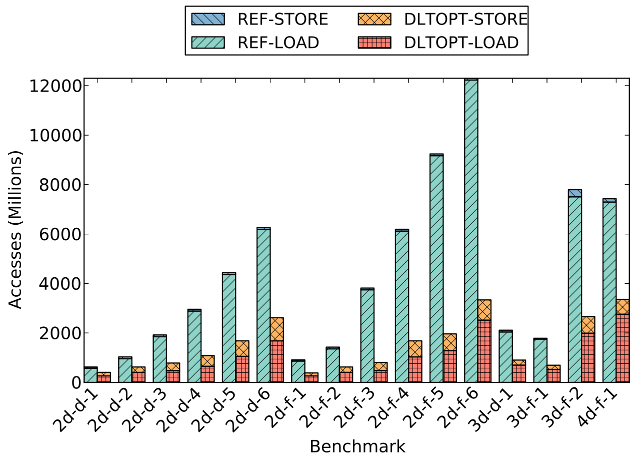


Figure 10: Number of loads and stores from hardware counters

In a perfect machine there would be about the same loads and stores for all the different stencil sizes across the given problem size, but as seen the reference codes show a steady increase of loads as stencil sizes increase. This is due to the result of high register pressure which forces each value to be reloaded many times.

As an alternative view of the hardware counters, Fig. 11 presents the memory operations per floating point operation for each benchmark, i.e., the inverse arithmetic intensity. Because the available reuse is increasing as the stencil size increases, a corresponding increase in the arithmetic intensity would be expected, but as can be seen the reference codes generally peak at about $2FLOPS/MOP$, whereas the retimed code reaches $6FLOPS/MOP$.

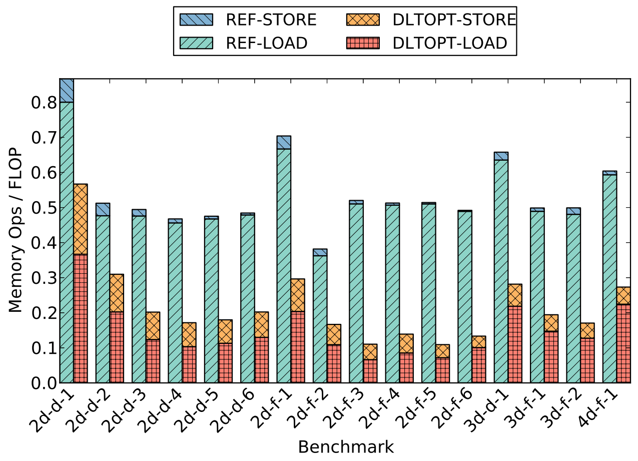


Figure 11: Loads and stores relative to total FLOPS executed

### 5.3 Impact of Transformations

Fig. 12 demonstrates the improvement gained by the various transformations for the synthetic benchmarks 2d-f-4, 3d-f-1, and 3d-f-2 as speedup over the corresponding reference implementation.

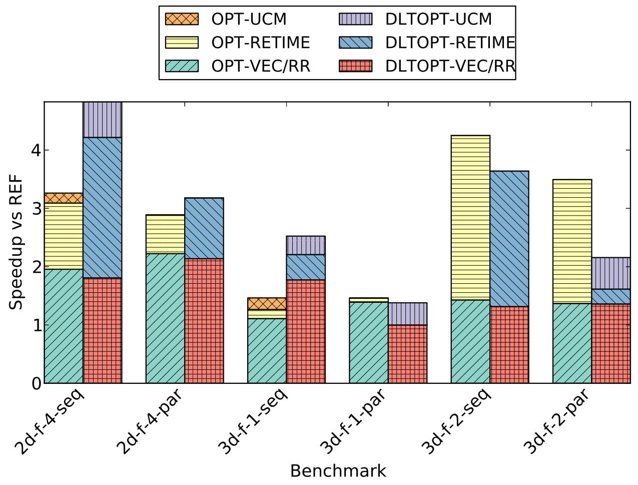


Figure 12: Speedup for different stages of optimization on synthetic benchmarks

For instance, for 2d-f-4 sequential using the standard layout, vectorization and application of rotating registers improves performance by 2× over reference, choosing the optimal retiming further improves performance to 3.5× better than reference, and allowing unroll and code motion (that is, when using all optimizations) brings the performance to 3.75× better. It is clear from this figure that high-order stencils can greatly benefit from the retiming optimization and effective reuse of registers, however unroll and code motion is less useful due to the already high register pressure in higher-order stencils. Comparing between 3d-f-1 and 3d-f-2 demonstrates that while the retiming can improve both stencils, its benefit increases as the order increases.

## 6. Related Work

A number of works have addressed the detection and automatic parallelization of scans and reductions [6, 16, 36, 45]. Commutativity analysis has been recognized as an approach to parallelizing computations [2, 23, 33, 37]. We are not aware of the use of commutativity to improve data locality. Multi-dimensional retiming has been been studied in the context of exploiting parallelism by a number of authors [7, 10, 27, 28].

In the context of stencil codes, Dursun et al. [14] describe hand-coded optimizations that compute contributions of data in registers instead of computing stencil elements one by one. We are not aware of any automatic approaches to this problem. Sedaghati et al. [38] propose hardware extensions to vector instructions to reduce the IO of stencil computations. Deitz et al. [12] and Datta [11] present techniques that exploit common subexpressions across different iterations in stencil codes. While this is orthogonal from the register reuse issues addressed in this paper, techniques such as Array Subexpression Elimination [12] are emulated by the combination of our technique with the compiler's common subexpression elimination. This is done automatically as a single plane's contributions to multiple points are considered in a single iteration; If the same coefficients are used for multiple points CSE removes the redundant work. Cruz et al. [9] present a technique for improving register reuse for a specific class of stencils which accumulates from neighbors along the axes. Their transformation is in fact a singular retiming, unique from the gather/scatter retiming, which can be represented by our framework.

## 7. Conclusion

This paper has addressed the use of associativity and commutativity in reordering operations with the goal of enhancing data locality and register reuse. This is of particular relevance in optimizing high-order stencil computations. A multi-dimensional retiming formalism was used to characterize the space of valid transformations and generate transformed code. Experimental results using a range of high-order stencils demonstrated the effectiveness of our transformation framework.

***Acknowledgements*** This work was supported in part by the U.S. National Science Foundation through awards 0811457, 0926127, 1059417 and 1321147, by the U.S. Department of Energy through award DE-SC0008844, and a Google European Fellowship in Efficient Computing.

## References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, 1964.

[2] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS*, pages 241–252, 2009.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks - summary and preliminary results. In *SC*, pages 158–165, 1991.

[4] J. W. Banks and W. D. Henshaw. Upwind schemes for the wave equation in second-order form. *J. Comput. Phys.*, 231(17):5854–5889, 2012.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, 2004.

[6] G. E. Blelloch. Scans as primitive parallel operations. *IEEE TC*, 38 (11):1526–1538, 1989.

[7] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *IEEE TPDS*, 9(1):24–35, 1998.

[8] Chombo. `https://commons.lbl.gov/display/chombo`.

[9] R. Cruz, M. Araya-Polo, and J. Cela. Introducing the semi-stencil algorithm. In *PPAM*, pages 496–506. 2010.

[10] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *PPL*, 7(4):379–392, 1997.

[11] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS, University of California, Berkeley, 2009.

[12] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Eliminating redundancies in sum-of-product array computations. In *ICS*, pages 65–77, 2001.

[13] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS*, pages 205–213, 2008.

[14] H. Dursun, M. Kunaseth, K. ichi Nomura, J. Chame, R. F. Lucas, C. Chen, M. W. Hall, R. K. Kalia, A. Nakano, and P. Vashishta. Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters. *The Journal of Supercomputing*, 62(2): 946–966, 2012.

[15] P. Feautrier. Dataflow analysis of scalar and array references. *IJPP*, 20(1):23–53, 1991.

[16] L. Han, W. Liu, and J. Tuck. Speculative parallelization of partial reduction variables. In *CGO*, pages 141–150, 2010.

[17] R. Haralick and L. Shapiro. *Computer and robot vision*. Computer and Robot Vision. Addison-Wesley, 1993.

[18] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short simd architectures. In *CC*, pages 225–245, 2011.

[19] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS*, 2013.

[20] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, 2012.

[21] S. Kim and S.-M. Moon. Rotating register allocation for enhanced pipeline scheduling. In *PACT*, pages 60–72, 2007.

[22] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *PLDI*, 2013.

[23] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *PLDI*, pages 542–555, 2011.

[24] T. Liebig. openEMS - Open Electromagnetic Field Solver. URL `http://openEMS.de`.

[25] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA*, pages 19–25, 1995.

[26] Overture. Overture: An Object-Oriented Toolkit for Solving Partial Differential Equations in Complex Geometry; version 25, 2012. http://www.overtureframework.org/.

[27] N. L. Passos and E. H.-M. Sha. Achieving full parallelism using multidimensional retiming. *IEEE TPDS*, 7(11):1150–1163, 1996.

[28] N. L. Passos, E. H.-M. Sha, and S. C. Bass. Optimizing dsp flow graphs via schedule-based multidimensional retiming. *IEEE TSP*, 44 (1):150–155, 1996.

[29] L.-N. Pouchet. PoCC 1.2: the Polyhedral Compiler Collection. `http://pocc.sourceforge.net`, 2012.

[30] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI*, pages 90–100, 2008.

[31] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *POPL*, pages 549–562, 2011.

[32] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *FPGA*, 2013.

[33] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *PLDI*, pages 1–11, 2011.

[34] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *IJPP*, 28(5):469–498, 2000.

[35] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, 2013.

[36] X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In *PARLE*, pages 132–145, 1993.

[37] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 19(6):942–991, 1997.

[38] N. Sedaghati, R. Thomas, L. Pouchet, R. Teodorescu, and P. Sadayappan. StVEC: A vector instruction extension for high performance stencil computation. In *PACT*, pages 276–287, 2011.

[39] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *GH*, pages 97–106, 2007.

[40] L. T. Simpson. *Value-driven Redundancy Elimination*. PhD thesis, Houston, TX, USA, 1996.

[41] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *PACT*, pages 292–304, 2007.

[42] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.

[43] H. Weller. OpenFOAM. URL `http://www.openfoam.org/`.

[44] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[45] Y. Zou and S. Rajopadhye. Scan detection and parallelization in ”inherently sequential” nested loop programs. In *CGO*, pages 74–83, 2012.