SDSLc: a Multi-Target Domain-Specific Compiler for Stencil Computations

Prashant Rawat[1] Martin Kong[1] Tom Henretty[1] Justin Holewinski[1] Kevin Stock[1]
Louis-Noël Pouchet[1] J. Ramanujam[2] Atanas Rountev[1] P. Sadayappan[1]
[1] Ohio State University
[2] Louisiana State University

ABSTRACT

Stencil computations are at the core of applications in a number of scientific computing domains. We describe a domain-specific language for regular stencil computations that allows specification of the computations in a concise manner. We describe a multi-target compiler for this DSL, which generates optimized code for GPUa, FPGAs, and multi-core processors with short-vector SIMD instruction sets, considering both low-order and high-order stencil computations. The hardware differences between these three types of architecture prompt different optimization strategies for the compiler. We evaluate the domain-specific compiler using a number of benchmarks on CPU, GPU and FPGA platforms.

1. INTRODUCTION

Stencils represent an important computational pattern used in scientific applications in a variety of domains including computational electromagnetics [33], solution of PDEs using finite difference or finite volume discretizations [30], and image processing for CT and MRI imaging [8, 10]. A number of recent studies have focused on optimizing stencil computations for multicore CPUs [7, 12, 13, 32, 34] and GPUs [22–24]. 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) [37], Finite-Difference Time Domain methods (e.g., seismic wave propagations, electromagnetic radiations) [20], image processing (e.g., edge detection) [28], 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 [6] uses high-order stencil operations in discretizing high-order derivatives.

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, 15–17, 28, 29] and several domain-specific optimization frameworks have been provided [4, 7, 28, 34]. There is increasing interest in developing domain-specific frameworks for high-performance scientific computing due to the diversity of current and emerging parallel architectures, as exemplified by large-scale projects such as the Center for Domain-Specific Computing [2] and the ExaStencils project [1, 19]. In addition to the benefit of a DSL (Domain Specific Language) for user productivity, a significant advantage is that semantic properties derivable from the high-level abstractions can be utilized to develop powerful specialized compiler optimizations that can be tailored to the characteristics of different target architectural platforms. Using the Stencil Domain Specific Language (SDSL) we proposed [14] to provide a target-independent description of the computation. This paper presents an overview of target-specific optimization strategies for multi-core CPUs with short-vector SIMD instructions [16, 31]; for GPUs [17]; and for FPGAs [27, 39].

WOLFHPC2015, November 15-20, 2015, Austin, TX, USA
 ISBN 978-1-4503-4016-8/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2830018.2830025

The rest of the paper is organized as follows. Sec. 2 presents a brief overview of the features of the SDSL multi-target domain-specific compiler. Sec. 3 introduces the SDSL language. We then present optimization challenges and possible optimization strategies to overcome them in Sec. 4 for multi-core CPUs, in Sec. 5 for GPUs, and in Sec. 6 for FPGAs.

2. OVERVIEW OF THE SDSL COMPILER

SDSL programs can be automatically optimized for CPUs, GPUs and FPGA execution. Fig. 1 outlines the SDSL compiler workflow and its various backends.

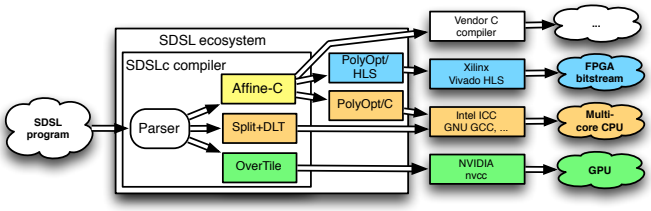


Figure 1: The SDSL Ecosystem

We have implemented target-specific backends within the SDSL compiler: one for CPUs and one for GPUs. To enable seamless interoperability with vendor compilers for other architectures, and to enable optimization using source-to-source compilers, a third backend produces *affine C* code describing the computation. This backend exploits the regular nature of programs that can be expressed in SDSL to generate regular (affine) C code which can then be analyzed and optimized by polyhedral compilers such as PolyOpt or Pluto. Two optimization strategies discussed below, supporting high-order stencil for CPUs [31] and general stencils for FPGAs, are implemented by using the affine-C backend and invoking PolyOpt on the generated source code.

The SDSLc compiler can produce three code variants. (1) The unoptimized affine C code is C99 compliant and is meant to be processed further by polyhedral optimization tools such as Pluto [35] and PolyOpt [25]. (2) The optimized CPU code is C99 compliant and uses vector intrinsic functions for x86 and 32-bit ARM CPU. The code is optimized with nested/hybrid split-tiling [16] in conjunction with dimension-lift-and-transpose [15] data layout transformations. (3) The optimized GPU code is generated in CUDA C by the OverTile backend [17]. OverTile generated code can be autotuned using a simple script included with the sdslc distribution.

SDSL code can be embedded in C, C++, and MATLAB, as discussed below. Optimized MATLAB code is generated as C functions called via MEX.

3. THE SDSL LANGUAGE

SDSL (Stencil Domain Specific Language) is a domain-specific language for expressing stencil computations. SDSL is loosely based on the RNPL [21] and SNPL languages used for rapid prototyping of partial differential equation solvers, although the resemblance is mostly cosmetic and no code is shared between the projects. The purpose of SDSL is to provide a programming language that allows for the specification of non-trivial stencil computations in a form that enables the generation of high- performance implementations that can be obtained in a performance-portable manner on multiple platforms.

Implementing high-performance stencil computations require both domain-specific and target-specific optimization strategies. For a compiler to automatically apply these strategies, the entire computation of interest must be analyzable without ambiguity. Traditional compiler analysis issues such as memory aliasing, loop trip count computation, induction variable analysis, etc. can prevent aggressive optimizations from being performed for safety/correctness reasons. Domain-specific languages can be designed so that programs can be successfully analyzed and transformed by the compiler. In SDSL, to achieve portable high-performance on multi-core CPUs, GPUs and FPGAs, program and data layout transformations must be applied by the compiler. Consequently we have designed the SDSL language to ensure that:

- the entire stencil computation (this includes time iterations, boundary conditions, and the operations applied on each voxel) can be modeled in the polyhedral program representation;
- the data space accessed by stencils is fully described at the SDSL level, enabling data layout transformations;
- general-purpose programming is allowed *outside* the stencil regions, that is SDSL is an embedded DSL;
- time-iterated stencils using data-dependent convergence checking are supported.

An illustrative example. We now illustrate the main concepts of the SDSL language using the example in Fig. 2. The program begins with a declaration of two int parameters, `dim0` and `dim1` on lines 1 and 2. These parameters are used to define a 2-dimensional grid of size `dim1 x dim0` on line 3. Line 4 declares floating point `griddata` in the shape of the `grid` defined on line 3. This data is also declared to exist at two different offsets from the current timestep, `0` and `1`. Lines 5-9 define a 5-point stencil as a `pointfunction` named `five_pt_avg`. This function takes an argument `p`, the grid data used to calculate the stencil. `five_pt_avg` averages 5 neighboring points in one timestep of `p` and writes the result to a point in the next timestep of `p`. Lines 11-23 define an iterative loop with a convergence check. The `iterate` construct on line 11 specifies that the iterative loop should run, at most, 1000 times. Lines 12-28 define a stencil computation over subsets of the problem domain. Lines 13-16 specify the value of the edges of grid data a to be the same at both timesteps at which a is defined. Line 17 specifies that the point function `five_pt_avg` be applied at every point over the interior of `a`. Lines 20-22 define a `reduction` to be performed at every point on `a`. The reduction variable is `max_diff`, and it contains the largest difference between a value of an element of `a` at two successive timesteps. Finally, line 23 defines a convergence condition for the iterate and specifies how frequently this check is to be performed. In this program the condition is that the largest difference between successive time values of an element of `a` is less than .00001 and that this condition should be checked every 4 iterations. The `check every` clause is essential to forward algorithmic knowledge to the compiler: if the convergence check needs only be computed every 4 iterations, then time-tiling techniques can be safely employed. The algorithm developer is in charge of ensuring the computation has enough numerical stability to possibly compute more time iterations than strictly needed, which in turn leads to powerful optimizations being applicable by the compiler.

```
1  int dim0;
2  int dim1;
3  grid g [dim1][dim0];
4  float griddata a on g at 0,1;
5  pointfunction five_point_avg(p) {
6    float ONE_FIFTH = 0.2f;
7    [1]p[0][0] = ONE_FIFTH*([0]p[-1][0]
8      + [0]p[0][-1]+[0]p[0][0]+[0]p[0][1]+[0]p[1][0]);
9  }
10
11 iterate 1000 {
12   stencil jacobi_2d {
13     [0][0: dim0 -1] : [1]a[0][0] = [0]a[0][0];
14     [dim1 -1][0:dim0 -1] : [1]a[0][0] = [0]a[0][0];
15     [0: dim1 -1][0] : [1]a[0][0] = [0]a[0][0];
16     [0: dim1 -1][ dim0 -1] : [1]a[0][0] = [0]a[0][0];
17     [1: dim1 -2][1: dim0 -2] : five_point_avg(a);
18   }
19
20   reduction max_diff max {
21     [0:dim1-1][0:dim0-1] : fabs([1]a[0][0]-[0]a[0][0]);
22   }
23 } check (max_diff < .00001f) every 4 iterations
```

Figure 2: Jacobi 2D in SDSL

Embedding SDSL in C/C++. Every embedded section of SDSL begins with `#pragma sdsl begin` and is terminated using `#pragma sdsl end`. Optionally, this pragma may contain a gpu() clause specifying parameters used by the OverTile [4] backend for CUDA C code generation. The main arguments are:

- `block:<comma separated list of integers>`: A comma separated list of integers specifying the GPU thread block size for generated code. There must be one integer in the list for each spatial dimension of the grid in the SDSL code.
- `tile:<comma separated list of integers>`: A comma separated list of integers specifying the number of elements to be computed per thread. There must be one integer in the list for each spatial dimension of the grid in the SDSL code.
- `time:<integer>`: A single literal integer value specifying the time tile size in the generated code.

Embedding SDSL in MATLAB. Many programs for scientific computation and visualization are written in MATLAB, but running MATLAB code is often time-consuming. A general technique to accelerate MATLAB is using the MEX interface to execute optimized native code. Compute-intensive stencil code in a MATLAB program can be rewritten in SDSL. The resulting MATLAB

code can be compiled by sdslc to generate a MEX source file containing a C or CUDA implementation of the SDSL functions and necessary MEX glue code. The generated MEX function can be independently optimized for different architectures. SDSL code embedded in MATLAB is defined by the same syntax and semantics as SDSL code embedded in C/C++.

In order to allow easy integration of eSDSL into C/C++ programs and MATLAB programs, we do impose restrictions on how the data structures interfacing between the rest of the program and the eSDSL segment are declared. Typically, contiguous memory regions of scalar types (e.g., `float`, `double`, etc.) are required for the fields storing the data; and integer types are required for parameters such as the grid size or the number of time iterations. It is also expected that the implementation of the stencil pointfunctions do not contain any side-effect generating instructions nor any function calls.

4. OPTIMIZATIONS FOR CPUS

4.1 General Stencils

Challenges Faced. Vector operations with ISAs like SSE require the loading of physically contiguous data elements from memory into vector registers and the execution of identical and independent operations on the components of vector registers. Stencil computations pose challenges to efficient implementation on these architectures, requiring the use of redundant and unaligned loads of data elements from memory into different slots in different vector registers. The DLT data layout transformation of Henretty et al. [15] was developed to overcome the fundamental data access inefficiency on current short-vector SIMD architectures with stencil computations, but coupling it with time-tiling techniques poses challenges on the acceptable tile shapes and overall strategy, as summarized below.

Data layout transformation and tiling. The DLT transformation is a key enabler to high-performance on short-vector SIMD architectures. Fig. 3 illustrates the DLT transformation for a one-dimensional vector of 24 elements for an ISA with a vector length of 4. Whereas B[0:3] form an aligned vector before transformation, after the DLT transformation, B[0], B[6], B[12], and B[18] form the first four elements Bdlt[0:3] in the transformed layout. The next four contiguous elements Bdlt[4:7] in the transformed layout correspond to B[1], B[7], B[13], and B[19], etc. Thus the sum of aligned vectors, Bdlt[0:3]+Bdlt[4:7]+Bdlt[8:11], computes $< B[0]+B[1]+B[2], B[6]+B[7]+B[8], B[12]+B[13]+B[14], B[18]+B[19]+B[20] >$.

Thus the fundamental problem with vectorized addition of contiguously located elements in memory is overcome in the transformed layout where operands that need to be combined are located in the same slot of different vectors rather than in different slots of the same vector. Details on the code generation technique for DLT-transformed arrays may be found in [15], including issues such as how elements at the DLT boundaries are handled.

Nested Split-Tiling. Inter-tile dependences between adjacent tiles along both the time and spatial dimensions makes it infeasible to use DLT with classical parallelogram-shaped time-tiling because DLT causes spatially separated data elements (for example, B[0], B[6], B[12], B[18] in Fig. (3)) to be gathered together in a single vector and therefore must be operated upon concurrently.

In nested split-tiling, a d-dimensional loop spatial loop nest is recursively split-tiled along each dimension. The outermost spatial loop at level d is split-tiled, producing a loop over upright tiles and a loop over inverted tiles. Inside each of these loops, loop level $d-1$ is split-tiled, giving four tile loop nests. Split-tiling is performed recursively in each new loop nest until the base loop level 1 is reached and there are 2^d total loop nests corresponding to all possible combinations of upright and inverted tiles on each dimension. Nested split-tiling of a 2D code is illustrated in Fig. 4(a)

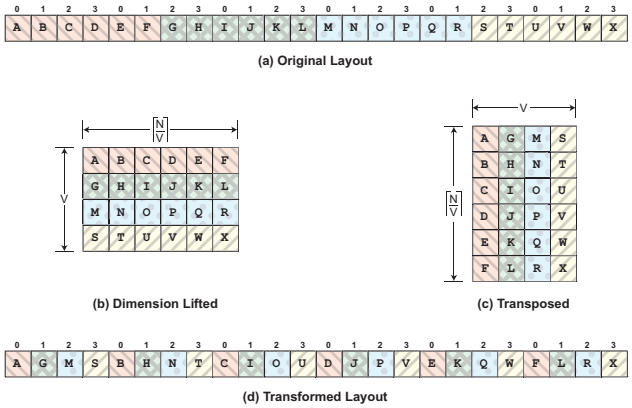


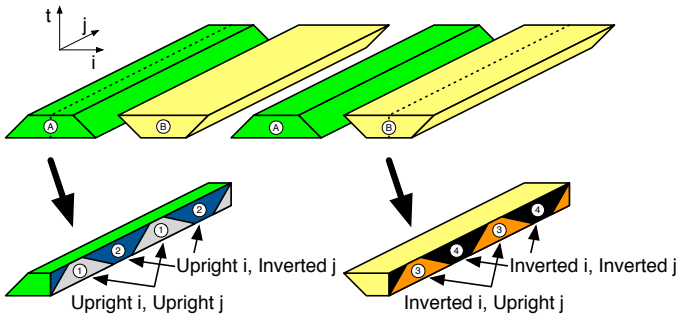
Figure 3: Data layout transformation for SIMD vector length of 4

Fig. 4(a) depicts, on the left, a series of upright ('A') and inverted ('B') tiles in the i dimension. All upright 'A' tiles may be executed concurrently, followed by all inverted 'B' tiles. Below these tiles are representative cross-sections of an upright and inverted tile showing the nested split-tiles in the j dimension. These tiles are labeled such that all tiles with the same number, ('1', '2', '3', or '4') may be executed concurrently, and tiles with a lower number must be executed before tiles with a higher number.

The pseudocode in Fig. 4(a) shows the loop nests responsible for producing the diagram. Nested inside the sequential *tt* loop are two parallel *ii* loops corresponding to the 'A' and 'B' tiles shown in the diagram. Nested inside the 'A' loop are parallel *jj* upright ('1') and inverted ('2') tile loops corresponding to the tiles shown in the left cross-section, Similarly, the 'B' loop contains nested parallel '3' and '4' loops corresponding to the right cross-section. A barrier follows each *jj* tile loop to enforce tile execution order and ensure that no dependences are violated.

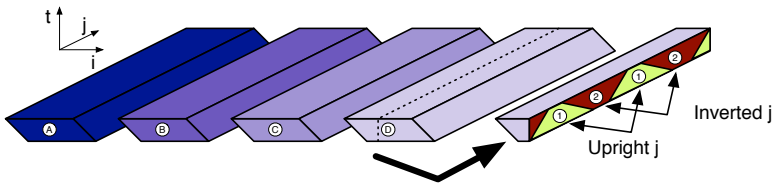
Nested split-tiling enables parallelization of all spatial loop nests in a stencil, however (1) it imposes a lower bound on the size of upright tiles for a given time tile size, or equivalently, (2) it imposes an upper bound on the time tile size given an upright tile's size. In nested split-tiling, upright tiles must be sized such that they retain their characteristic trapezoidal shape. If the base of the upright tile is not large enough for a given time tile size, the sloping lines will eventually form a tip. At this point tile execution cannot extend any further in time.

Hybrid Split-Tiling. For higher dimensional problems, the lower bound on upright tile size causes tiles to overflow cache for even small time tile sizes. We overcome the tile size constraints of nested split-tiling with a hybrid of standard tiling on the outermost space loops and split-tiling on the inner loops. Hybrid split-tiling for a 2D




```
for tt
  parfor ii // (A) Upright i
    parfor jj // (1) Upright j
      for t { for i { for j {}}};
    barrier();
    parfor jj // (2) Inverted j
      for t { for i { for j {}}};
    barrier();
  parfor ii // (B) Upright j
    parfor jj // (3) Upright j
      for t { for i { for j {}}};
    barrier();
    parfor jj // (4) Inverted j
      for t { for i { for j {}}};
    barrier();
```

(a) 2D nested split-tiling




```
for tt
  for ii // (A) (B) (C) (D) Traditional i
    parfor jj // (1) Upright j
      for t { for i { for j {}}};
    barrier();
    parfor jj // (2) Inverted j
      for t { for i { for j {}}};
    barrier();
```

(b) 2D hybrid split-tiling

Figure 4: 2D nested and hybrid split-tiling.

stencil is illustrated in Fig. 4b. The pseudocode contains a single *ii* loop nested in the *tt* loop which corresponds to the four traditional *i* dimension tiles 'A', 'B', 'C', and 'D' shown in the diagram. These tiles must be executed in sequence from 'A'–'D'. Nested inside the *ii* loop is the split-tiled *jj* loop which has the same upright / inverted tile structure as the split-tiled inner loops described in the previous section.

Standard tiling does not impose any constraint on tile sizes along spatial dimensions as a function of the time tile size. Thus, standard tiles may be compacted to a much smaller size to compensate for the larger tile sizes required by split-tiled dimensions. This allows for a substantially reduced multidimensional tile footprint.

In order to perform combined data layout transformations for SIMD vectorization with parallel tiling for data locality, we use a multi-stage process to integrate dimension-lift-and-transpose (DLT) with multi-level split-tiling. Arguments to the transformation algorithm are computed using a static analysis on the stencil DSL input program, as described in [16].

Experimental validation. The effectiveness of the both nested split-tiling and hybrid split-tiling applied in conjunction with the dimension-lifting transformation was experimentally evaluated in detail in [16]. We compare performance to the diamond-tiling system used by Pluto [4], the cache-oblivious tiling system used by Pochoir [34], and the Intel C Compiler v13.0, all running on *Intel Core i7-2600K* (Sandy Bridge micro-architecture) which is a quad-core core x86-64 chip running at 3.4 GHz; double-precision peak performance is 27.2 GFlop/s/core (108.8 GFlop/s aggregate). The following stencil codes were used as benchmarks (with the names used to refer to them in parentheses): Jacobi 1D (jac-1d-3), Jacobi 2D (jac-2d-9), Jacobi 3D (jac-3d-7), Laplacian 2D (lapl-2d), Gradient 2D (grad-2d) Heat 1D/2D/3D (heat-*n*d) distributed with Pochoir [34]. All array dimensions were set to be significantly larger than last level cache on all micro-architectures. For all stencils, the footprint of each array was set to 488.28MB of double-precision data; this was achieved using 1D arrays with $64 * 10^7$ scalar elements, 2D arrays with 8000^2 elements, and 3D arrays with 400^3 elements. The number of time steps was set to 100 for all benchmarks. Tile sizes were autotuned for Pluto with diamond tiling, as well as for our split-tiling and overlapped tiling work.

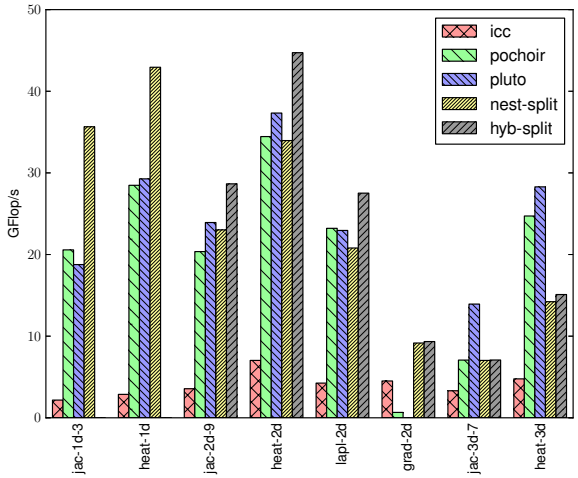


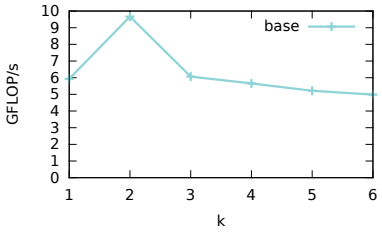
Figure 5: Intel Core i7-2600K AVX Performance

4.2 High-Order Convolution Stencils

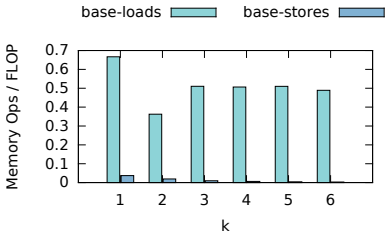
Challenges faced. We use the example in Fig. 6 to illustrate the fundamental issues with high-order stencils that we tackled in [31]. The code in Fig. 6(a) is a generic convolution stencil that sweeps over a 2D array *OUT*, where at each point (i, j), a weighted sum of a $n \times n$ ($n = 2 \times k + 1$) 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

```
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

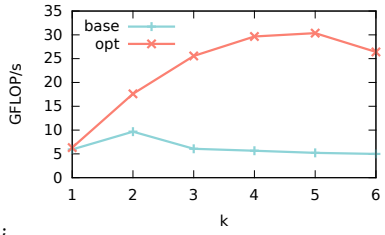


(c) Hardware counted loads and stores

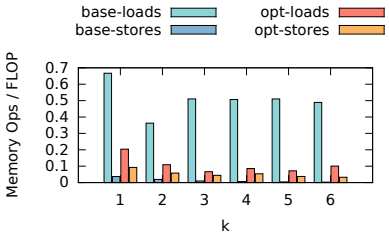
Figure 6: Implementation and performance of the base codes

```
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 7: Implementation and performance of the base and optimized codes

memory bandwidth to peak computational performance [38]. However, the arithmetic intensity of a stencil is directly related to its order k.

A 3×3 2D stencil, that is $k = 1$ in Fig. 6(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 5×5 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 7×7 stencil's machine balance requirement will be roughly half of that for the 5×5 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. 6(b) shows a different trend. While performance does indeed increase from a 3×3 ($k = 1$) stencil to a 5×5 ($k = 2$) 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. 7(b) from the template in Fig. 7(a).

The problem is that while the burden on the memory subsystem is reduced for higher order stencils, register pressure worsens. For a 3×3 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 5×5 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. 6(c) show the total number of load instructions executed per FLOP decreases when we go from $k = 1$ (3×3 stencil) to $k = 2$ (5×5 stencil).

A 7×7 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. 6(c). Performance continues to drop as we further increase the stencil order, while greater arithmetic intensity implies performance should be improving.

Register reuse framework for high-order stencils. We developed 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 [31]. 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 stencil accumulations to multiple target elements. A transformed code template,

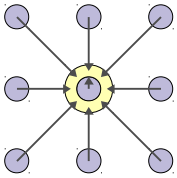
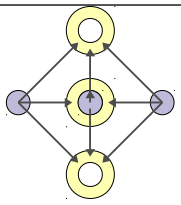
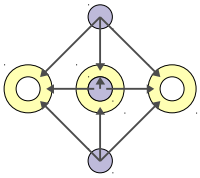
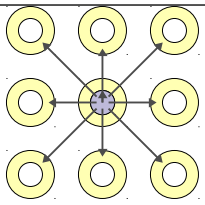
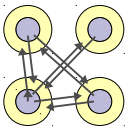
Variant	Gather-Gather	Gather-Scatter	Scatter-Gather	Scatter-Scatter	Compact
Diagram					
IN_{loads}	n	1	n	1	$\lceil n/2 \rceil$
OUT_{loads}	0	$n-1$	0	$n-1$	$\lfloor n/2 \rfloor$
OUT_{stores}	1	n	1	n	$\lceil n/2 \rceil$
$REGS$	n^2-n+2	$n+2$	$n+2$	n^2-n+2	$2\cdot(\lceil n/2 \rceil)^2+2$

Table 1: Expected IO and register pressure of different retiming variants for the 2D ($n \times n$) stencil of Fig. 6 ($n = 2k+1$)

representative of the kind of operation reordering generated by our framework, is shown in Fig. 7(a).

In contrast to the original code in Fig. 6(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. 7(a) may be viewed as a *scatter-gather* stencil. The code shown in Fig. 7(a) performs exactly the same set of operations as the code in Fig. 6(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 3×3 stencil, for a given point (i, j) of the outer two loops, here we have a 3×1 "read-set" of three elements from *IN* each making contributions to each element of a 1×3 "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. 7(b), and is compared with the base code over which it shows substantial performance improvement. Fig. 7(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. 6(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. 6(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. 6(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 2×2 configuration.

Overview of the approach. Our end-to-end optimization process involves the following steps, and is further described in [31]:

1. Extract an internal representation of the input code using polyhedral compilation concepts.
2. Create a space of abstract scatter/gather alternatives along with different unrolling factors for the program.
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.
4. Prune the space of candidate variants based on their arithmetic intensity relative to the original code using our analytical model.

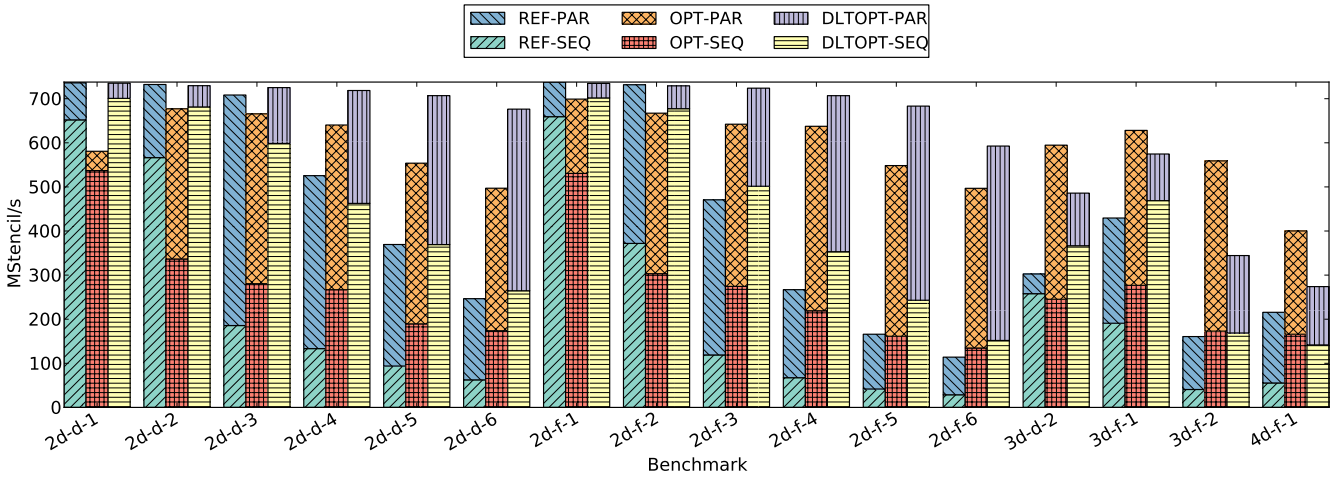


Figure 8: Performance as rate of stencil applications per second

5. For each point remaining, scatter/gather the appropriate dimension, unroll, generate C code, and perform complementary optimizations for vectorization.
6. Perform auto-tuning to find the best performing variant on the target machine.

Experimental validation. Comprehensive experimental evaluation is presented in [31], we summarize here results on a four-core Intel Core i7-4770K CPU (Haswell micro-architecture) running at 3.5GHz with AVX2 SIMD instruction set including fused-multiply-add (FMA) instruction. Its theoretical peak is 112 GF/s (224GF/s if using only FMA instructions). 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. We used a set of synthetic high-order stencil benchmarks with varying stencil size and number of neighbors with non-zero weight. 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.

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. A key observation is that with our approach, the rate of stencil application is essentially maintained even when the order is increased, meaning more accurate discretizations (possibly converging faster) can be used without performance penalty.

5. OPTIMIZATIONS FOR GPUS

Challenges faced. Efficient GPU programs typically involve the scheduling of hundreds of threads per streaming multi-processor to hide memory latency. The streaming multi-processors schedule threads at the granularity of warps, which comprise 32 threads on previous and current generation architectures. The thread scheduler time-shares the streaming multi-processors between all currently active warps, and thread context switches incur no overhead.

Several sources of inefficiency can arise when developing GPU applications. GPU devices provide a very high off-chip memory bandwidth (up to 192 GB/sec for the GTX 580), but *this bandwidth is only achievable with coalesced access*. Data from the off-chip memory is transferred to the GPU device in contiguous blocks and therefore high bandwidth can be achieved only when requests by concurrent threads in a warp fall within such contiguous blocks. When non-contiguous memory locations are accessed by threads, the achieved bandwidth is much lower than the peak, leading to stalling and wasted compute cycles. Branch divergence is another source of inefficiency. Threads within a warp that follow different control paths are serialized, again leading to wasted compute cycles. Traditional approaches to time tiling of stencil computations to enhance data reuse for CPUs do not translate well to GPUs because they lead to uncoalesced memory access and divergent branching of threads. Another challenge comes from shared scratch-pad memory implemented as a banked memory system. If concurrently executing threads in a block make requests to shared memory locations in the same bank, a bank conflict occurs and the requests are serialized. Therefore, to achieve optimal usage of shared memory, *concurrently executing threads should access data from different banks*. We have described an automated code generation approach to overcome these challenges, for the class of stencil computations [17]. A brief overview is presented in the following.

Overview of the approach. Tiling for stencil computations is complicated by data sharing between neighboring tiles. Cells along the boundary of a tile are often needed by computations in surrounding tiles, requiring communication between tiles when neighboring tiles are computed by different processors. To compute a stencil on a cell of a grid, data from neighboring cells is required. These cells are often referred to as the *halo* region. In general, to compute an $N \times M$ block of cells on a grid, we need an $(N+n) \times (M+m)$ block of data to account for the cells we are computing as well as the surrounding halo region, where n and m are constants derived from the shape of the stencil. For GPU devices, the halo region needs to be re-read from global memory for every time step as surrounding tiles may update the values in these cells. This limits the amount of re-use we can achieve in scratch-

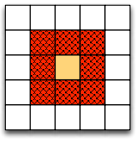
pad memory before having to go back to global memory for new data. The new cell values produced in each time step must also be re-written to global memory as other tiles may require the data in their halo regions. In addition to the cost of going to global memory for each time step, global synchronization is also required to ensure all surrounding tiles have completed their computation and written their results to global memory before new halo data is read for each tile.

To get around these issues, we use overlapped tiling [18] to reduce the data sharing requirements for stencil computations by introducing redundant computations. Instead of forcing tile synchronization after each time step to update the halo region for each tile, each tile instead redundantly computes the needed values for the halo region. This allows us to efficiently perform time tiling and achieve high performance on GPU targets.

Consider a simple 2-D Jacobi 9-point stencil like the one shown in Figure 2. In order to compute one time step of an $n \times n$ tile, we need to read $(n+2) \times (n+2)$ cells into scratchpad memory, compute the stencil operation on each of the $n \times n$ points, then write back $n \times n$ points to global memory. Now, let us consider the computation of two time steps of the stencils on a single block, without having to go back to global memory between the two time steps. If we read $(n+4) \times (n+4)$ cells into memory, we can compute a $(n+2) \times (n+2)$ tile of cells in the first time step, which includes the results for our original $n \times n$ tile as well as the halo region needed for the next time step. If we again apply the stencil operation to the $(n+2) \times (n+2)$ tile, we correctly compute the inner $n \times n$ tile for the second time step but the results computed in the halo region for the second time step are not correct. However, this is not a problem since we only care about the inner $n \times n$ region. An illustration of this computation is presented in Figure 10.

```
1  for(t = 0; t < T; ++t) {
2    for(i = 1; i < N-1; ++i) {
3      for(j = 1; j < N-1; ++j) {
4        A[i][j] = CNST * (B[i  ][j  ] +
5          B[i  ][j-1] + B[i  ][j+1] +
6          B[i-1][j  ] + B[i-1][j-1] +
7          B[i-1][j+1] + B[i+1][j  ] +
8          B[i+1][j-1] + B[i+1][j+1]);
9      }
10   }
11   for(i = 1; i < N-1; ++i) {
12     for(j = 1; j < N-1; ++j) {
13       B[i][j] = A[i][j];
14     }
15   }
16 }
```

(a)



(b)

Figure 9: Jacobi 9-Point Stencil. In (a), C code is shown for the stencil, and in (b), the accessed data space is shown for one grid point. The tan cell is the (i, j) point, and each red hashed cell is read during the computation of the (i, j) cell.

Experimental validation. A comprehensive evaluation of the proposed overlapped tiling technique can be found in [17]. Overlapped tiling was evaluated on an NVIDIA GTX 580 using the CUDA 5.0 SDK. In general, overlap tiled codes outperformed their CPU based counterparts, while not approaching the machine peak of 198 GFlop/s. The GTX 580 relies upon fused multiply-add instructions to achieve this rate, however all benchmarks were strongly biased towards add instructions, with a 3-10X ratio of adds to multiplies. Further, while GPU codes were optimized using overlapped

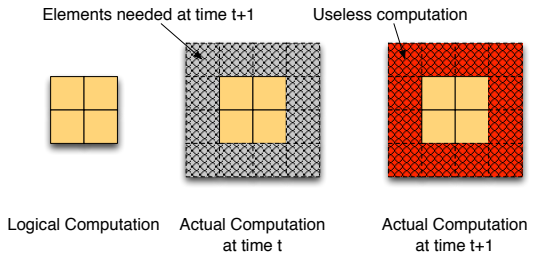


Figure 10: Overlapped Jacobi 9-Point Stencil for $T = 2$.

tiling, other optimizations were not systematically applied. Finally, 3D stencil codes on the GPU were not time-tiled because of the substantial overhead introduced by computing the multidimensional ghost regions.

6. OPTIMIZATIONS FOR FPGAS

Challenges faced. High level synthesis (HLS) tools for synthesizing designs specified in a behavioral programming language like C/C++ can dramatically reduce the design time especially for embedded systems. While the state-of-art HLS tools have made it possible to achieve performance close to hand coded RTL designs from designs specified completely in C/C++ [3], considerable manual design optimization is still often required from the designer [8]. To get a HLS friendly C/C++ specification, the user often needs to perform a number of explicit source-code transformations addressing several key issues such as on-chip buffer management, choice of degree of parallelism / pipelining, attention to prefetching, avoidance of memory port conflicts etc., before designs rivaling hand coded RTL can be synthesized by the HLS tool.

Most integrated circuits, especially for embedded systems, use on-chip buffer memories for fast and energy-efficient access to the most frequently used data. For FPGAs, the total data for the application is typically much larger than on-chip memory capacity. In contrast to general-purpose processors that use hardware-managed caches to hold frequently accessed data, the use of on-chip buffers with explicit copy-in and copy-out of data is a key optimization for embedded systems [9]. By storing frequently accessed data in the on-chip buffer, the bandwidth contention is decreased, and the overall performance increases significantly as the latency of accessing on-chip data is significantly faster than off-chip accesses. A fully automated approach for on-chip buffer management that consists of promoting to local memory (e.g., the on-chip buffer) memory references in the program is needed to achieve good performance in a productive manner.

Overview of the approach. To address these problems, we developed an automated compiler support based on the latest advances in polyhedral frameworks (e.g., [5,36]) to greatly reduce the human design effort currently required to create effectively synthesizable specification of designs using HLS tools. In particular, we developed compiler support for source-to-source transformations to optimize critical resources such as memory bandwidth to off-chip memory and on-chip buffer capacity [27].

In our framework, we use a multi-stage process to automatically optimize C programs for effective execution on a FPGA. Our approach uses design-space exploration to determine the best performing program variant. Specifically, we search for best perfor-

mance through the evaluation of different *rectangular* tile sizes. Our framework is built so that different tile sizes lead to different program candidates, with distinct features in terms of the communication schedule, buffer size, loop to be tiled (e.g. when a tile size of 1 is used for this loop), etc. Each candidate is built as follows.

1. We first transform the input program, using polyhedral loop transformations. The objective is to restructure the program so that data locality is maximized (e.g., the "time" between two accesses to the same memory cell is minimized), and at the same time the number of loops that can be tiled is maximized. Tilable loops are tiled using a tile size given as input.
2. We then promote all memory accesses to on-chip buffers in the transformed program, and automatically generate off-chip / on-chip communication code. Data reuse between consecutive iterations of a loop is automatically exploited. The hardware constraints on the maximal buffer size are automatically satisfied, using a lightweight search algorithm that trades off communication volume for buffer size.
3. We conclude the code transformation process by performing a set of HLS-specific optimizations, such as coarse-grain and fine-grain/task-level parallelism extraction.

Promoting the entire data accessed by a program to local memory is often infeasible, in particular for FPGA design where the on-chip buffer resource is limited. Therefore, we want to enable the promotion of all program references to an on-chip buffer, while still controlling its size. We chose to solve this problem by using the granularity of the loop iteration, for any of the loops in the program. That is, given an arbitrary loop in the program (which may very well be surrounded by other loops), our technique will compute the minimal on-chip buffer size requirement and associated communications to execute one iteration of this loop, while exploiting the reuse between consecutive iterations of said loop. This implicitly offers a lot of freedom for the on-chip buffer size. By considering the innermost loop, its size will be similar to the number of registers required to execute the computation. By considering the outermost loops it will be equivalent to the entire data space of the program. Any loop in-between will trade off communication count for on-chip buffer size (and its associated static energy).

For example, in Figure 2 if we put on-chip the data accessed by one full row i of data, we need to store the i^{th} row of A and B, as well as the $(i-1)^{th}$ and $(i+1)^{th}$ rows of A, leading to a buffer requirement of $4.N$. This buffer must be filled for each iteration of the i loop, that is roughly $T.N$ times (total communication volume is roughly $4.N^2.T$). Putting on-chip the full computation (that is, along the t loop on line 1) leads to a $2.N^2$ buffer requirement, but to be filled only once (total communication volume is reduced to $2.N^2$). So, the trade-off here is between a buffer size N times smaller versus a communication volume increase of $2.T$.

Our technique operates on each array individually, and promotes optimally (under the framework constraints) all references to this array into a dedicated on-chip buffer for this array. Our approach is based on the concept of *parametric polyhedral sets* to express the set of data elements being used at various specific points of the computation. Those sets correspond exactly to the data to be communicated, reused, or stored. We then use a polyhedral code generator to scan those sets, and properly modify the program by inserting the code that scans communications sets, and change main memory references in the modified source code to on-chip buffer references. To illustrate our approach, in Figure 11 we show the sets $DS(A,\vec{P}_{j,0})$ (left) and $DS(A,\vec{P}_{j,-1})$ (center), the data space of the immediately preceding iteration, for a pixel at position (i,j) of the Jacobi2D example. By computing the difference or intersection between those sets (right), we can capture naturally the data reused between two consecutive iterations, as well as the data that is not alive at the previous iteration and that needs to be brought from off-chip locations. In our compiler, we compute these sets for various situations (e.g., buffering a pixel, a row, a tile of pixels, etc.), then use static cost models to determine which buffering scheme leads to the best balance between on-chip usage and communication cost, so as to maximize performance.

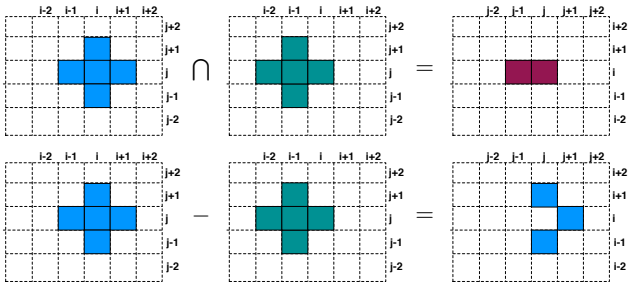


Figure 11: Computation of the *Reuse* (top) and *PerIterCommunication* (bottom) sets

Experimental validation. A comprehensive evaluation can be found in [27]. We summarize here our results for two complex stencil applications from the medical imaging domain, combining several Jacobi-like and Seidel-like 3D stencils. Table 2 reports the performance, in GigaFlops per second, of several different implementations of the same benchmark. out-of-the-box reports the performance of a basic manual off-chip-only implementation of the benchmark, without our framework, but using the Vivado HLS tool chain. PolyOpt/HLS reports the performance achieved with our automated framework used on top of Vivado. From a user standpoint, the same code is written as input to the system in the out-of-the-box and PolyOpt/HLS-E cases, but in the latter case the code compiled by Vivado is automatically generated by PolyOpt. Hand-tuned reports the performance of a manually hand-tuned version serving as our performance reference, from Cong et al. [8]. It has been designed through time-consuming source code level manual refinements, specifically for the Convey HC-1ex machine. It demonstrated that a 4-FPGA manual design for denoise and segmentation systematically outperforms a CPU-based implementation, both in terms of performance improvement (from 2× to 20×) and energy-delay product (up to 2000×), therefore showing the great potential of implementing such 3D image processing algorithms on FPGAs [8].

For segmentation, we outperform the manual design, despite the clear remaining room for improvement our framework still has, as shown by the denoise example. We mention that semi-automated manual design can be performed on top of our framework, to address optimizations we do not support, such as array partitioning.

Table 2: FPGA Performance on Imaging Benchmarks

Benchmark	out-of-the-box	PolyOpt/HLS	hand-tuned [8]
denoise	0.02 GF/s	4.58 GF/s	52.0 GF/s
segmentation	0.05 GF/s	24.91 GF/s	23.39 GF/s

7. CONCLUSION

In this paper, we have provided a brief overview of the Stencil Domain-Specific Language (SDSL) for regular stencil computa-

tions, which allows specification of the computations in a concise manner. We then provided an overview of the SDSL optimization ecosystem, a multi-target compiler for this DSL that generates optimized code for multi-core processors with short-vector SIMD engines, for GPUs, and for FPGAs. We have illustrated a selection of key challenges in optimizing stencils on each of these architectures, summarizing our domain-specific-target-specific strategies to generate portable high-performance for stencils from a single program source.

Acknowledgments. 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.

8. REFERENCES

[1] Advanced Stencil-Code Engineering (ExaStencils). http://www.exastencils.org.

[2] Center for Domain-Specific Computing. http://cdsc.ucla.edu.

[3] An independent evaluation of the autoesl autopilot high-level synthesis tool. Technical report, Berkeley Design Technology, Inc., 2010.

[4] V. Bandishti, I. Pananilath, , and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *SC'12*, 2012.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *Proc. PLDI*, June 2008.

[6] https://commons.lbl.gov/display/chombo.

[7] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IEEE Intl. Parallel & Distributed Processing Symp.*, IPDPS '11, pages 676–687, 2011.

[8] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-fpga platforms. In *Proc. of Intl. Conference on Field Programmable Logic and Applications*, FPL, 2011. to appear.

[9] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473 –491, april 2011.

[10] J. Cong and Y. Zou. Lithographic aerial image simulation with fpga-based hardwareacceleration. In *ACM/SIGDA symp. on Field programmable gate arrays*, FPGA, pages 67–76, 2008.

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

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08*, pages 4:1–4:12, 2008.

[13] D. Han, S. Xu, L. Chen, and L. Huang. Pads: A pattern-driven stencil compiler-based tool for reuse of optimizations on gpgpus. In *Parallel and Distributed Systems (ICPADS'11)*, pages 308 –315, dec. 2011.

[14] T. Henretty, J. Holewinski, N. Sedaghati, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Stencil domain specific language (sdsl) user guide. Technical Report OSU-CISRC-4/13-TR09, Department of Computer Science and Engineering, Ohio State University, Columbus, OH, April 2013.

[15] 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.

[16] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS'13*. ACM Press, June 2013.

[17] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS'12*. ACM Press, June 2012.

[18] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI '07*, pages 235–244, 2007.

[19] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *Euro-Par 2014: Parallel Processing Workshops*, pages 553–564. Springer, 2014.

[20] T. Liebig. openEMS - Open Electromagnetic Field Solver. http://openEMS.de.

[21] R. Marsa and M. Choptuik. The rnpl userâĂŹs guide. http://laplace.physics.ubc.ca/People/arman/files/RNPL_ref.pdf.

[22] J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. *Intl. Journal of Parallel Programming*, 39:115–142, 2011. 10.1007/s10766-010-0142-5.

[23] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, 2009.

[24] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *IEEE Conference of Supercomputing*, SC '10, pages 1–13, 2010.

[25] Ohio State University. the PolyOpt polyhedral compiler. http://hpcrl.cse.ohio-state.edu/wiki/index.php/Polyhedral_Compilation.

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

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

[28] 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.

[29] 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.

[30] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 2004.

[31] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, volume 49, pages 65–76. ACM, 2014.

[32] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS '10*, pages 49–59, New York, NY, USA, 2010. ACM.

[33] A. Taflove. *Computational electrodynamics: The Finite-difference time-domain method*. Artech House, 1995.

[34] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *ACM Symp. on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, 2011.

[35] Uday Bondhugula. the Pluto polyhedral compiler collection. http://pluto-compiler.sourceforge.net.

[36] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin / Heidelberg, 2010.

[37] H. Weller. OpenFOAM. http://www.openfoam.org/.

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

[39] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. Improving polyhedral code generation for high-level synthesis. In *CODES+ISSS'13*. IEEE Press, 2013.