A Stencil Compiler for Short-Vector SIMD Architectures

Tom Henretty
The Ohio State University
henretty@cse.ohio-state.edu

Richard Veras
Carnegie Mellon University
rveras@cmu.edu

Franz Franchetti
Carnegie Mellon University
franzf@ece.cmu.edu

Louis-Noël Pouchet
Univ. California, Los Angeles
pouchet@cs.ucla.edu

J. Ramanujam
Louisiana State University
jxr@ece.lsu.edu

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

ABSTRACT

Stencil computations are an integral component of applications in a number of scientific computing domains. Short-vector SIMD instruction sets are ubiquitous on modern processors and can be used to significantly increase the performance of stencil computations. Traditional approaches to optimizing stencils on these platforms have focused on either short-vector SIMD or data locality optimizations. In this paper, we propose a domain-specific language and compiler for stencil computations that allows specification of stencils in a concise manner and automates both locality and short-vector SIMD optimizations, along with effective utilization of multi-core parallelism. Loop transformations to enhance data locality and enable load-balanced parallelism are combined with a data layout transformation to effectively increase the performance of stencil computations. Performance increases are demonstrated for a number of stencils on several modern SIMD architectures.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Compilers*

Keywords

DSL, Multicore, SIMD, Split Tiling, Stencils

1. INTRODUCTION

There is increasing interest in developing domain-specific frameworks for high-performance scientific computing due to the diversity of current/emerging parallel architectures. In addition to the benefit of a DSL (Domain Specific Language) on user productivity, a significant advantage is that semantic properties derived from the high-level abstractions can be utilized to develop powerful specialized compiler transformations that can be tailored to the characteristics of different architectural platforms. Using the Stencil Domain Specific Language (SDSL) [11], this paper describes a set of compiler transformations that are needed to generate efficient code for multicore processors with short-vector SIMD ISAs such as SSE, AVX, VSX, LRBNi etc.

Stencil computations involve arithmetic operations on physically contiguous data elements, e.g., c0*(A[i-1]+A[i]+A[i+1]). Since 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. We [12] had previously addressed this issue through a *dimension-lifting-transpose (DLT)* data layout transformation. However, only sequential execution was addressed. Further, the approach was only evaluated on data sets that fit in L1 cache. Tiling over spatial and the time dimensions is essential in conjunction with DLT for high performance on large data sets. However, as elaborated in detail in the next section, standard time-tiling of stencil codes via skewing introduces inter-tile dependences that are incompatible with the form of vector parallelism used by the DLT transformation. In this paper, we develop an integrated approach to perform tiling in conjunction with DLT transformation to generate efficient parallel code for stencil computations over large data sets on shared memory multiprocessors. We compare performance with code generated by the Pochoir stencil compiler [18] and Pluto [2, 4] for several benchmarks on multiple target multicore processors, demonstrating strong performance benefits for 1D and 2D stencils. The paper makes the following contributions:

- It presents a stencil DSL compiler that integrates data layout transformation for short-vector SIMD ISAs with load-balanced tiled parallel execution for multi-statement stencils.
- It demonstrates significant performance improvement on several multi-core platforms for a number of benchmarks, over Intel's ICC compiler and state-of-the-art research compilers like Pochoir and Pluto.

The paper is organized as follows. In Sec. 2 we use an illustrative example to explain the main problem to be addressed in integrating DLT with tiling. Sec. 3.1 describes the stencil DSL and Sec. 3.4 provides a high-level overview of the compiler algorithms developed. Sections 4 and 5 provide details of the compiler algorithms. Experimental results are presented in Sec. 6 and related work is covered in Sec. 7.

2. PROBLEM DESCRIPTION

In this section we first provide some background on the DLT data layout transformation of Henretty et al. [12] that was developed to overcome the fundamental data access inefficiency on current

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

short-vector SIMD architectures with stencil computations. We then describe why standard time-tiling is infeasible in conjunction with the DLT transformation and a different form of tiling – split-tiling – can be used effectively in conjunction with DLT.

2.1 Overview of DLT Transformation

Fig. 1 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.

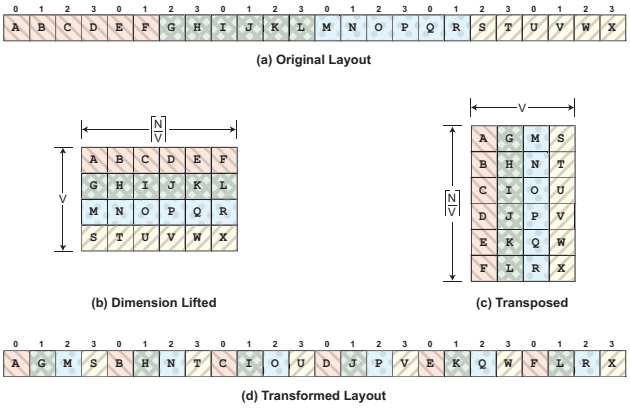


Figure 1: Data layout transformation for SIMD vector length of 4

2.2 Standard Tiling and DLT Transformation

We next use a Jacobi 1D stencil example to explain the problem with the use of standard time-tiling in conjunction with DLT layout transformation. Although the input to our stencil compiler uses a special DSL language (described in Sec. 3.1), we use standard loop notation in C to motivate the problem since this lower-level view makes it easier to discuss issues pertaining to loop fusion and time-tiling when compiling general multi-statement stencils for high performance – something that to the best of our knowledge is not addressed by other stencil compilers such as PATUS [5] and Pochoir [18].

Fig. 2(a) shows code for a 1D Jacobi 3-point stencil with a sequence of two spatial loops within an outer time loop, where S1 performs the stencil computation over the spatial domain and S2 copies the output array into the input array for use in the next time step. In order to enhance data locality, time-tiling may be employed, but will first require some transformations in order to create atomic tiles that compute forward for several time steps over a subset of the spatial domain that is small enough to fit within cache. Fig. 2(b) shows a fused form that creates a unified 2D iteration

```
for (t=0; t<T; t++) {
  for (i=1; i<N-1; i++) {
    B[i] = 0.33*(A[i-1] + A[i] + A[i+1]); // S1
  for (i=1; i<N-1; i++)
    A[i] = B[i]; // S2
}
```

(a) Unfused

```
for (t=0; t<T; t++) {
  B[1]=0.33*(A[0]+A[1]+A[2]);
  for (i=2; i<N-1; i++) {
    B[i] = 0.33*(A[i-1] + A[i] + A[i+1]);
    A[i-1] = B[i-1];
  }
  A[N-1] = B[N-1];
}
```

(b) Fused

Figure 2: Jacobi 1D stencil

space with a statement body including both S1 and S2 (along with peeling of an iteration at the boundaries of the *i* loop).

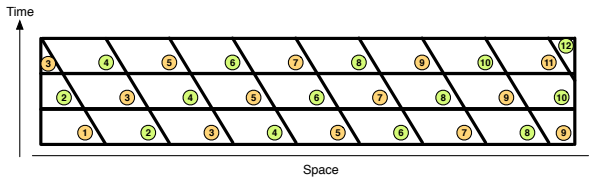
Further skewing of this unified iteration space will be required to create valid "rectangular" tiles, which can equivalently be viewed as parallelogram-shaped tiles in an unskewed iteration space, as shown in Fig. 3(a). Because of the shape of valid tiles (they cannot be rectangular in an unskewed iteration space due to forward and backward dependences along the spatial dimension), there are inter-tile dependences between adjacent tiles along both the time and spatial dimensions. This inter-tile dependence along the spatial dimension makes it infeasible to use DLT because DLT causes spatially separated data elements (for example, B[0], B[6], B[12], B[18] in Fig. (1)) to be gathered together in a single vector and therefore must be operated upon concurrently. The circled value in each tile of Fig. 3(a) represents the logical time at which the tile can be executed, such that all tiles it depends on have been previously computed.

Fig. 3(b) shows a different form of tiling – split-tiles. Here, upright and inverted tiles alternate and the inter-tile dependences are only from an upright tile to its two neighboring inverted tiles. As a result, concurrent execution of all upright tiles over a given time range is feasible, followed by concurrent execution of the inverted tiles over the same time range. Again, the circled values within the tiles indicate the sequence of execution of the tiles, where tiles with the same sequence number can be executed concurrently. With such a tiling strategy, it is now feasible to use DLT, as long as all data elements grouped into each vector are all within upright tiles or all within inverted tiles. Further, unlike execution required with standard tiling, the schedule for parallel tile execution with split-tiles is fully load balanced and does not have a sequential start and gradual build up of inter-tile parallelism as required for wavefront-parallel standard tiling.

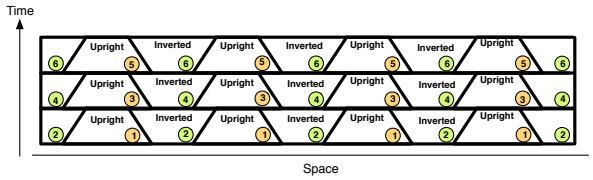
In the next section, we provide a description of the stencil DSL and a high-level overview of the compiler algorithm for code generation.

3. OVERVIEW OF APPROACH

Before delving into the details of the algorithms, in this section we first describe the stencil DSL we translate. Next, we provide overviews of two methods used to tile SDSL codes, nested and hybrid split-tiling. Finally, a high-level overview of the approach to



(a) Standard Tiling



(b) Split-Tiling

Figure 3: Tiled Iteration Space for 1D Space, 1D Time

transform an input SDSL program into an semantically equivalent split-tiled DLT program is provided.

3.1 SDSL – Stencil Domain Specific Language

A stencil computation can be summarized as one or more functions being identically applied to points on a regular grid, where the values of some groups of neighboring elements are used for each function, and this process is repeated multiple times. The Stencil Domain Specific Language (SDSL) enables the concise description of stencil computations and is briefly described in the following sections.

3.1.1 Program Description

The computation shown in Fig. 4 is a standard Jacobi 2D computation, which averages the value of the 5 neighboring points (up, down, left, right, and center) to compute the new value of the center point.

```
grid g[dim1][dim0];
double griddata a on g at 0,1;

iterate 100 {
  stencil five_pt {
    [1:dim1-2][1:dim0-2] : [1]a[0][0] =
                      0.2*([0]a[-1][0]+
              [0]a[0][-1]+[0]a[ 0][0]+[0]a[0][1]+
                          [0]a[ 1][0]);
  }
}
```

Figure 4: A simple Jacobi 2D example in SDSL

Structural Mesh (`grid`*).* The first line of Fig. 4 defines `g`, the grid where stencil computations may be defined. It is an n-dimensional Cartesian coordinate space (a subset of $\mathbb{Z}^2$ here), and the computations operate on a subset of this grid. We note that grid size can be a parameter, that is a program constant whose value is not known at compile-time.

Data Elements (`griddata`*).* The second line of Fig. 4 defines `a`, a double precision data field with the same structure as grid `g`. This field holds data values used in stencil functions, and multiple fields may be defined over a grid. The grid `g` is used to define the size of `a` and sets limits on field indices. The `at` clause specifies that there should be two copies of the field, one associated with the current outermost loop iteration and another at the next outermost loop iteration.

Computation (`iterate` *and* `stencil`*).* The last eight lines of Fig. 4 define a stencil computation. Three key concepts are defined: (1) outer loop trip count, (2) subgrid(s) over which to apply a stencil function and (3) stencil function(s).

The outer loop trip count is defined in the `iterate` construct, and is 100 in the example. The stencil construct is given a unique identifier, `five_pt`, and contains the definition of a subgrid over which to apply the stencil function definition that follows. In the example the subdomain `[1:dim1-2][1:dim0-2]` defines a subset of the grid `g` that contains all elements except a single cell border on all four sides.

A stencil function is defined after the subdomain definition. This function averages the current point and four of its neighbors in `a` at the current timestep and places the resluts in `a` at the next timestep. References to `griddata` consist of the offset from the current iteration in brackets, followed by the the name of the referenced field, followed by offsets from the current point in each spatial dimension in brackets.

3.1.2 General Form of an SDSL Program

In general, an SDSL program contains one `grid`, one or more `griddata`, one `iterate`, and one or more `stencil` definitions, where each `stencil` may define one or more subdomains and the stencil functions that operate upon them.

The abstract form of an SDSL program is represented in Fig. 5. The program is constrained to be a collection of M K-dimensional grid data and N stencils, with each stencil applying some stencil function f on one or more grid data. Each stencil function is executed on a rectangular subdomain $Z \subset \mathbb{Z}^K | 0 \leq lb_Z^k, ub_Z^k < dim_k \forall k \in \{1..K\}$. While an SDSL program may have multiple subdomains and stencil functions defined inside one `stencil`, the abstract version in Fig. 5 is semantically equivalent.

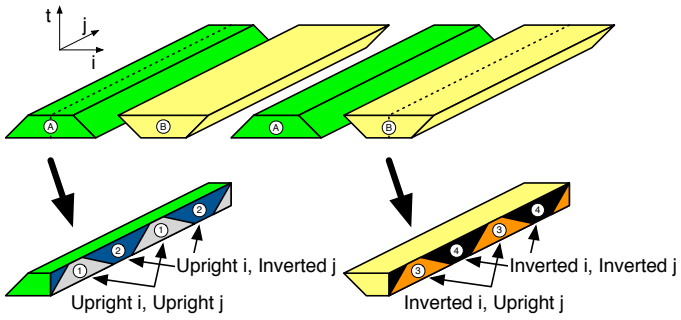
```
grid g[dimK]...[dim1];

griddata g1,g2...,gM on g;

iterate T {
  stencil s1 {
    [lb_s1_K:ub_s1_K]...[lb_s1_1:ub_s1_1] : f1(...);
  }
  stencil s2 {
    [lb_s2_K:ub_s2_K]...[lb_s2_1:ub_s2_1] : f2(...);
  }
  ...
  stencil sN {
    [lb_sN_K:ub_sN_K]...[lb_sN_1:ub_sN_1] : fN(...);
  }
}
```

Figure 5: General Form of an SDSL Program

SDSL programs are parallelized and optimized for data locality using nested and hybrid split-tiling, described in the next two sections.

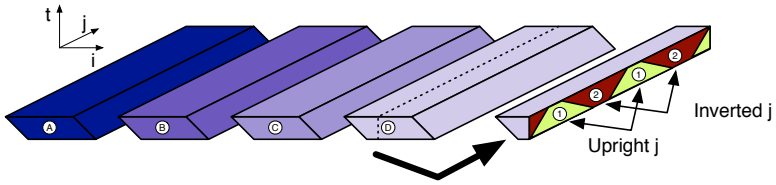
3.2 Nested Split-Tiling

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




```
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 6: 2D nested and hybrid split-tiling.

$d-1$ 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. 6(a).

Fig. 6(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. 6(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 *j j* 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 *j j* 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, as in Fig. 3(b). 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.

Given an upright tile with a base size of T_U^d, maximum absolute value of slopes in d s_{max}^d, maximum offset of all statements o_{max}^d, and time tile size T_T the following constraint can be stated:
$T_U^d \geq 2 * T_T * s_{max}^d + 2 * o_{max}^d$

For higher dimensional problems, the lower bound on upright tile size causes tiles to overflow cache for even small time tile sizes. Consider a 3-dimensional stencil with, for all dimensions, maximum slope $s_{max}^d = 2$, maximum offset $o_{max}^d = 0$, and $T_T = 8$. This requires $T_U^d \geq 32$. For an upright tile in all dimensions, including the innermost vector dimension, this is at least 32K vector elements, enough to overflow L1 and L2 cache on most modern architectures.

3.3 Hybrid Split-Tiling

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 stencil is illustrated in Fig. 6(b). 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 *j j* 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 the example at the end of Sec. 3.2, we may reduce the tile size of the outermost dimension to 2, thereby reducing the the tile size to 2K elements. Since inner loops are split-tiled, we retain adequate parallelism.

3.4 Overview of the Optimization Algorithm

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. The overall transformation flow is summarized in Fig. 7.

```
Input
   P: input SDSL program
   dsplit: number of dimensions to split-tile
Output
   O: optimized C program

   O ← performDLT(P)
   (α⃗,β⃗,o⃗_L,u⃗) ← backslice(P)
   O ← performSplitTiling(O, α⃗, β⃗, o⃗_L, u⃗, dsplit)
   O ← finalizeTiling(O, α⃗, β⃗, o⃗_L, u⃗, dsplit)
   return O
```

Figure 7: Overview algorithm

Function `performDLT` applies DLT on all inner-most vectorizable loops, following the method presented in Henretty et al. [12]. We remark that programs that can be modeled in SDSL all have vectorizable inner-loops, so that DLT can be applied for all stencil functions. Details of this function are provided later in Sec. 5. Function `backslice` performs backslicing analysis to compute the exact shape of the split-tiles (that is, computing for each stencil function the offsets and slopes of a split-tile, to be translated on the entire spatial domain). This is detailed in Section 4. Function `performSplitTiling` uses the split-tile shape information computed to generate split-tile code for the *d* split inner spatial dimensions. This is detailed in Section 5. Finally, function `finalizeTiling`completes code generation, by applying standard tiling on the remaining dimensions, if any. The integration as well as the complete algorithm is discussed in Section 5.

4. BACKSLICING ANALYSIS

Split-tiling requires the computation of sets of iteration space points that can be executed atomically – that is a valid tiling – while preserving parallelism between tiles of the same category (i.e., upright tiles have to be parallel with each other). In order to compute the *shape* of the split-tile that satisfies those properties, we first highlight the main ideas using a Jacobi 1D stencil, before describing the general algorithm for higher dimensional stencils with an arbitrary number of stencil functions.

4.1 Split-Tiling Jacobi 1D

We illustrate the main ideas behind split-tiling using a Jacobi 1D example. Fig. 8 shows the corresponding input SDSL program.

```
grid g[1000];
double griddata a on g at 0,1;

iterate 100 {
  stencil f1 {
    [1:998] : [1]a[0] =
           0.33*([0]a[-1]+[0]a[0]+[0]a[1]);
  }
}
```

Figure 8: A simple Jacobi 1D example in SDSL

In the SDSL intermediate representation, an explicit copy of the field `a1` into the field `a0` is added after each time iteration, leading to a program equivalent to the C code shown in Fig. 9.

Statement `f1` performs the actual stencil computation, producing the `a1` field, statement `f2` copies the `a1` field to the `a0` field. This sequence is repeated 100 times.

```
for (t = 0; t < 100; t++) {
  for (i = 1; i <= 998; i++) {
    f1: a1[i] = 0.33*(a0[i-1] + a0[i] + a0[i+1]);
  }
  for (i = 0; i <= 999; i++) {
    f2: a0[i] = a1[i];
  }
}
```

Figure 9: Jacobi-1D example with explicit copy

4.1.1 Examining an Upright Tile

Let us consider the top segment of an upright tile for `f2`, over a span $[P,Q]$, that corresponds to the iterations of `f2` performed at time T. In order to correctly compute those iterations, we need the values $[P-1,Q+1]$ of `a1` that were computed by executing `f1` on the segment $[P-1,Q+1]$ time T, which in turn depends on the values of `a0` over $[P-2,Q+2]$ copied by `f1` at time $T-1$. Based on data dependences between the statements `f1` and `f2`, we can compute precisely which iterations must have been computed at previous time steps for each of the statements in order to compute the segment $[P,Q]$ of `f2` at time step T. This is illustrated in Fig. 10.

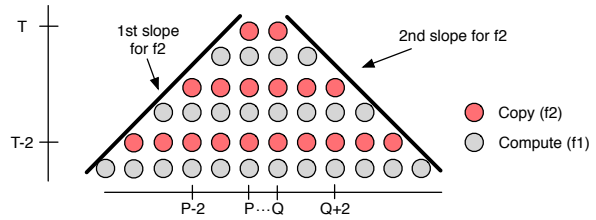


Figure 10: Upright split-tile for Jacobi 1D

Fig. 10 shows the set of preceding iterations, for both `f1` and `f2`, that must be computed in order to obtain the segment $[P,Q]$ at time T. We show here a time tile size of 3, that is, we build a split-tile that computes over three time steps.

The dependences are analyzed from the SDSL representation. Due to the restriction on stencil shapes to be constant integer offsets (e.g., -1, 2, etc.), the dependences are simple integer relations between time and the access functions. In the next sections we show how data dependences are used to construct a *dependence summary graph* and formulate validity constraints on the split-tiles for the slopes and statement offsets.

4.1.2 Building the Dependence Summary Graph

We begin by creating the dependence summary graph (DSG), a multigraph with vertices for each stencil function and edges that summarize flow and anti dependence information between stencil functions. In general, a vector of 2*d+1 components is used to model data dependence in an imperfectly nested loop with maximum loop depth d, with d components representing the distances along the loops, and the other d+1 components being used to mark the relative textual ordering within a loop level. However, the structure of an SDSL program always has the form of an outer time loop surrounding a sequence of perfectly nested loops. For generation of valid split-tiled code, the exact textual position of a sequence of statements is not significant, but only whether a dependence is from a textually preceding or succeeding statement within the time loop. Further, when several dependences exist between a pair of statements due to multiple array read references it is only necessary to

identify the maximal spatial extent of dependences along the different directions at each time step. Therefore, instead of using the standard general representation of dependence vectors, we separate out the distance vector component along the time (outermost) dimension and the components along the spatial dimensions.

For the Jacobi 1D example, we have the following dependences:

$$\mathcal{D}_{\text{f1}\to\text{f2}} = \begin{cases} \text{flow: f1}(t,i) & \to & \text{f2}(t,i) \\ \text{anti: f1}(t,i) & \to & \text{f2}(t,i-1) \\ \text{anti: f1}(t,i) & \to & \text{f2}(t,i) \\ \text{anti: f1}(t,i) & \to & \text{f2}(t,i+1) \end{cases}$$

$$\mathcal{D}_{\text{f2}\to\text{f1}} = \begin{cases} \text{flow: f2}(t,i) & \to & \text{f1}(t+1,i-1) \\ \text{flow: f2}(t,i) & \to & \text{f1}(t+1,i) \\ \text{flow: f2}(t,i) & \to & \text{f1}(t+1,i+1) \\ \text{anti: f2}(t,i) & \to & \text{f1}(t+1,i) \end{cases}$$

The spatial components of the dependence vectors between dependent statements are computed by subtracting the target iteration from the source iteration, yielding the following vectors:

$$d_{\text{f1}\to\text{f2}} = \begin{cases} \delta_T = 0, \delta_i =< 0 > \\ \delta_T = 0, \delta_i =< -1 > \\ \delta_T = 0, \delta_i =< 0 > \\ \delta_T = 0, \delta_i =< 1 > \end{cases} \qquad d_{\text{f2}\to\text{f1}} = \begin{cases} \delta_T = 1, \delta_i =< -1 > \\ \delta_T = 1, \delta_i =< 0 > \\ \delta_T = 1, \delta_i =< 1 > \\ \delta_T = 1, \delta_i =< 0 > \end{cases}$$

The spatial components of the distance vectors are then coalesced into a tuple for each dependence such that the coalesced tuple $C^{fs\to ft} =< \delta_L, \delta_U >$ where δ_L is the maximum spatial distance and δ_U is the minimum spatial distance between two dependent statements fs and ft. For the Jacobi 1D example the tuples for each dependence are identical, $C^{\text{f1}\to\text{f2}} = C^{\text{f2}\to\text{f1}} =< 1, -1 >$. These tuples are used to label edges in the DSG, along with a separate label for the time distance δ_T. Assembling the coalesced tuples, time distances, dependences, and statements leads to the DSG shown in Fig. 11 for the Jacobi 1D example.

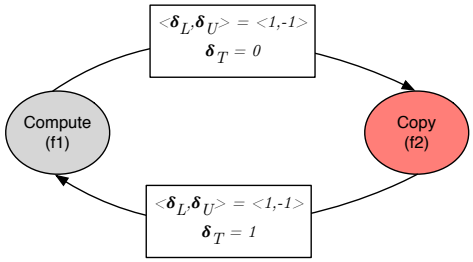


Figure 11: Dependence Summary Graph (DSG) for Jacobi 1D

This DSG is subsequently used in Sec. 4.1.3 to build validity constraints for split-tiles and in Sec. 4.1.4 to compute slopes and statement offsets.

4.1.3 Building Validity Constraints

We seek to constrain the legal values of tile slope and statement offsets by assembling a system of linear inequalities based upon the DSG and the loop bounds of the split-tiled code we will generate. Pseudocode for the loop nests of Jacobi 1D upright tile is shown in Fig. 12. Informally, the validity constraints state that, for any pair of dependent statements, given a region over which the target

```
for (tt = ...)
  for (ii = ...)
    for (t = 0; t < T_T; t++)
      for (i = ii + o_L^f1 + α*t; i ≤ ii + T_U + o_U^f1 + β*t; i++)
        f1: a1[i] = 0.33*(a0[i-1] + a0[i] + a0[i+1]);
      end for
      for (i = ii + o_L^f2 + α*t; i ≤ ii + T_U + o_U^f2 + β*t; i++)
        f2: a0[i] = a1[i];
      end for
    end for
  end for
end for
```

Figure 12: Loop Nests for Jacobi 1D Upright Tile. Time tile size is T_T; upright space tile size is T_U. Offsets from lower bound are o_L^{f1} and o_L^{f2}; offsets from upper bound are o_U^{f1} and o_U^{f2}. Slope of lower bound is α; slope of upper bound is β.

statement is executed, the source statement will be executed over a region that is, *at minimum*, large enough to satisfy the dependence.

From the DSG, we note that to compute statement `f2` over some range $[A,B]$ at timestep T we require that statement `f1` be executed over the range $[A-1, B-(-1)]$ at timestep T. Similarly, to compute statement `f1` over some range $[C,D]$ at timestep T we require that statement `f1` be executed over the range $[C-1, D-(-1)]$ at timestep $T-1$.

Combining the dependence information from the DSG with the loop bounds of Fig. 12 gives us validity constraints on values the loop bounds may take, and results in the following system of inequalities for lower bounds:

$$\begin{aligned} ii + o_L^{\text{f1}} + \alpha * t &\leq ii + o_L^{\text{f2}} + \alpha * t - 1 \\ ii + o_L^{\text{f2}} + \alpha * (t-1) &\leq ii + o_L^{\text{f1}} + \alpha * t - 1 \end{aligned}$$

The following system constrains the upper bounds:

$$\begin{aligned} ii + T_U + o_U^{\text{f1}} + \beta * t &\geq ii + T_U + o_U^{\text{f2}} + \beta * t + 1 \\ ii + T_U + o_U^{\text{f2}} + \beta * (t-1) &\geq ii + T_U + o_U^{\text{f1}} + \beta * t + 1 \end{aligned}$$

Simplifying and rearranging these systems of inequalities yields the following system of difference constraints for lower bounds:

$$\begin{aligned} o_L^{\text{f1}} - o_L^{\text{f2}} &\leq -1 \\ o_L^{\text{f2}} - o_L^{\text{f1}} &\leq \alpha - 1; \end{aligned}$$

the corresponding constraints for upper bounds are shown below:

$$\begin{aligned} o_U^{\text{f2}} - o_U^{\text{f1}} &\leq -1 \\ o_U^{\text{f1}} - o_U^{\text{f2}} &\leq -\beta - 1. \end{aligned}$$

These systems are used in Sec. 4.1.4 to compute valid offsets for all statements.

4.1.4 Computing Slopes and Offsets

To determine legal values for slopes α and β we compute, respectively, maximum and minimum cycle ratios $[1,7]$ on the DSG. A cycle ratio $\rho_L(C)$ on the DSG is computed by finding a cycle C, summing δ_L values over the cycle, and dividing by the sum of δ_T values. A cycle ratio $\rho_U(C)$ is calculated in a similar fashion with δ_U values. We set $\alpha = max(\rho_L(C))$ and $\beta = min(\rho_U(C))$ for all cycles C in the DSG.

We examine the only cycle in our example DSG, C_0, between `f1` and `f2`. The DSG tells us that computing `f1` on some interval $[A,B]$ at time T allows us to compute `f2` on the interval $[A + \delta_L^{f1\to f2}, B + \delta_U^{f1\to f2}]$ at time $T + \delta_T^{f1\to f2}$ without violating any dependences. Continuing along the cycle, computing `f2` on the in-

terval $[A+\delta_L^{f1\to f2}, B+\delta_U^{f1\to f2}]$ at time $T+\delta_T^{f1\to f2}$ allows us to compute `f1` on the interval $[A+\delta_L^{f1\to f2}+\delta_L^{f2\to f1}, B+\delta_U^{f1\to f2}+\delta_U^{f2\to f1}]$ at time $T+\delta_T^{f1\to f2}+\delta_T^{f2\to f1}$.

Substituting in known values for the various δ variables shows us that computing `f1` over the interval $[A,B]$ at time T allows us to compute `f1` over the interval $[A+2, B-2]$ at time $T+1$. Thus, we see a slope of 2 on the lower bound and a slope of -2 on the upper bound.

Equivalently, summing δ_L values and dividing by the sum of δ_T values gives us $\rho_L(C_0)=2$; a similar calculation gives us $\rho_U(C_0)=-2$. Since there is only one cycle in the DSG, these values are $max(\rho_L(C))$ and $\min(\rho_U(C))$, and we set $\alpha = max(\rho_L(C))=2$ and $\beta=\min(\rho_U(C))=-2$.

These values for α and β are substituted into the systems of difference constraints, and a solution to each of these systems is obtained using the Bellman-Ford algorithm [1, 6]. For the Jacobi 1D example we obtain $o_L^{f1}=-1$, $o_U^{f1}=1$, and $o_L^{f2}=o_U^{f2}=0$.

4.2 General Method

Our general algorithm closely follows the principles we have explained for Jacobi 1D. Since stencils may be multidimensional slopes and offsets are separately computed for each dimension. Further, different stencil functions may apply to different subdomains, which may be disjoint, overlapping, or identical. Because of this we conservatively assume all stencil functions are executed at all points in the problem domain when calculating dependences. This is an over-approximation of data dependences and has no impact on the final correctness of the generated code.

We present in Fig. 13 a general algorithm for computing slopes and offsets for a given input program. This algorithm produces lower/upper slope vectors with one slope per dimension, and lower/upper offset vectors with one offset per stencil statement.

```
Input
  P: input SDSL program
Output
  (α⃗, β⃗, o⃗_L, o⃗_U): Vectors of slopes alpha, beta.
                Vectors of lower and upper offsets in each
                spatial dimension for each stencil statement

  𝒟 ← calculateDependences(P)
  s ← calculateDependenceDistances(𝒟)
  DSG ← buildDSG(s,P)
  foreach dimension d of P do
    α_d ← computeMaxCycleRatio(DSG,d)
    β_d ← computeMinCycleRatio(DSG,d)
    α⃗[d] ← α_d
    β⃗[d] ← β_d
    (v⃗_L, v⃗_U) ← buildValidityConstraints(DSG,α_d,β_d)
    o⃗_L[d] ← solveForOffsets(v⃗_L)
    o⃗_U[d] ← solveForOffsets(v⃗_U)
  end do
  return (α⃗, β⃗, o⃗_L, o⃗_U)
```

Figure 13: Algorithm `backslice`

The algorithm in Fig. 13 takes an arbitrary SDSL program as input. Dependence analysis is performed on this program to determine all flow and anti dependences, and dependence distance vectors are calculated. The dependence distance vectors are used to build the DSG, where each vertex is an SDSL statement and each edge is labeled with a time distance. Edges are also labeled with coalesced distances, computed for each spatial dimension as in Sec. 4.1.2 and placed in vectors $(\vec{\delta_L}, \vec{\delta_U})$.

After the DSG is built, slopes and offsets are computed along each dimension d. For each dimension of the DSG we compute α and β using maximum and minimum cycle ratios. The α_d and β_d values are then added to $\vec{\alpha}$ and $\vec{\beta}$ as the slopes for d. Once the slopes have been computed for a dimension, validity constraints are constructed as shown in Sec. 4.1.3. Once the slopes and validity constraints are known for a given dimension, `solveForOffsets()` rearranges the validity constraints into a system of differences and solves for offset values using the Bellman-Ford algorithm. Offsets for each statement in the current dimension are appended to the offset vectors. This process is repeated until slopes and offsets have been computed for all dimensions.

4.3 Proof of Correctness

Upright Tiles: The validity constraints guarantee that no dependences in an upright tile will be violated. In this section we prove that lower bound slopes calculated with the maximum cycle ratio always lead to a system of constraints that has a solution; the proof for the upper bound and minimum cycle ratio is identical.

A system of difference constraints not solvable by the Bellman-Ford algorithm would contain a negative weight cycle. We show that this is not possible.

We begin by constructing a graph isomorphic to the DSG where each vertex is a statement and each directed edge is labeled with its corresponding validity constraint for the lower bound v_L^e on some dimension d.

All validity constraints take one of the following forms:

$$o^s \le o^t - \delta_e \quad \Rightarrow \quad o^s - o^t \le -\delta_e \tag{1}$$

$$o^s \le o^t - \delta_e + \alpha \quad \Rightarrow \quad o^s - o^t \le -\delta_e + \alpha \tag{2}$$

In (1) the dependence associated with the constraint has source and target statements at the same timestep; in (2) the statements are 1 timestep apart. The unknown source statement offset is o^s, the unknown target statement offset is o^t, the dependence distance/edge weight is δ_e, and α is the unknown slope.

We next show that any cycle of length k on this graph restricts the possible values of α to be of the following form:

$$\alpha \ge \frac{1}{t}\sum_{i=1}^{k}\delta_i. \tag{3}$$

Note that the maximum cycle ratio simply maximizes α subject to the constraint in (3). In (3), δ_i is the weight of an edge in the cycle and t is the number of edges where source and target are separated by 1 timestep. Every vertex n in the cycle is entered through an edge f and exited through an edge g. On edge e, vertex n is the target of the dependence; on edge g, vertex n is the source of the dependence. Summing validity constraints $v_L^f + v_L^g$ eliminates the offset o^n. The accumulated sum $\sum_{i=1}^{k} v_L^i$ eliminates all offsets and leaves $0 \ge \sum_{i=1}^{k}\delta_i - t*\alpha$, which is equivalent to (3).

We now show that a negative weight cycle in the graph used by the Bellman-Ford algorithm to solve the system of difference constraints requires a value of α that violates (3).

All constraints in the system of differences we are solving take one of two forms, Equation 1 or Equation 2. The graph used in Bellman-Ford, after removing the start vertex and zero-weight edges emanating from it, is isomorphic to the DSG and the validity constraint graph described above. Each vertex is a statement offset, and each edge is weighted by the difference between the source and the target offset. The weight of any length k cycle is shown in (4).

$$\sum_{i=1}^{k} -\delta_i + t*\alpha \tag{4}$$

Assume there is a negative weight cycle of length k. This requires (5) to be true:

$$\sum_{i=1}^{k} -\delta_i + t * \alpha < 0 \tag{5}$$

Rearranging (5), we have α constrained by (6):

$$\alpha < \frac{1}{t} \sum_{i=1}^{k} \delta_i \tag{6}$$

The same cycle, on the validity graph, shows that α is constrained by (3). This is a contradiction as we know α was constructed by maximized subject to (3) thus our assumption of a negative weight cycle in the graph used to compute offsets was wrong and the system of differences constraints used to solve for offsets will always have a solution.

Inverted Tiles: The loop bounds for each statement in an inverted tile are set to fully span the gap between adjacent upright tiles (or a domain boundary). The proof of satisfaction of all dependences during execution of an inverted tile is a direct consequence of this "completeness" property of the span of spatial loops of an inverted tile. Consider all instances at time t, of an arbitrary statement S_q in an inverted tile. Any dependence on instances of any statement S_p mean that S_p textually precedes S_q, or that the dependence is from instances of S_q at some earlier time step. There are three possibilities for any such statement instance from which there exists a dependence: i) it belongs in some upright tile, ii) it belongs in the same inverted tile, iii) it belongs in some other inverted tile. If we have case (i), the dependence is satisfied since all upright tiles are executed before any inverted tiles. If we have case (ii), again the dependence is satisfied since statements are executed in textual order within the inverted tile for a time step, and in increasing time order across time steps. Finally, case (iii) is impossible since other inverted tiles are separated from the current inverted tile by at least one upright tile, whose slopes are guaranteed to prevent any dependence edge crossing their boundaries.

Multiple Spatial Dimensions: In the proof of correctness, so far we have focused on the case of a single spatial dimension. For multi-dimensional stencils, as explained in Sec. 3, tiles may have different characteristics along the different spatial dimensions. For example, with two spatial dimensions and nested split tiling, there are four possibilities: Upright-Upright, Upright-Inverted, Inverted-Upright, Inverted-Inverted. With hybrid split tiling, one or more spatial dimensions may be tiled using standard parallel tiles (using the slope β computed for the right boundary in that dimension by the back-slicing analysis). The proof of validity of execution in the multi-dimensional case is a consequence of the fact that the loop bounds along each spatial dimension are independent of iterators of other spatial dimensions. Consider, for example, a stencil with three spatial dimensions i, j, and k, where standard tiling is used along k, and split tiling along i, and j. Let a particular tile be upright along i, and inverted along j. At some time step t, the set of all instances of a statement S_q will necessarily correspond to a cross product of ranges along the three spatial extents: [kl:ku,jl:ju,il:iu]. The instances of any statement S_p that S_q depends upon, can be expressed as $[kl+\delta_L^k : ku+\delta_U^k, jl+\delta_L^j : ju+\delta_U^j, il+\delta_L^i : iu+\delta_U^i]$. We can prove that all such dependences are satisfied by proving that the extents along each dimension are satisfied. For upright or inverted tiles along any dimension, we have proved that the required extents will be covered by the generated slopes/offsets. For a standard tiled dimension too, the dependences will be satisfied: on the lower side, the instances will either lie in the same tile or in an earlier numbered tile that would have been executed before this tile; on the upper side, the instance will necessarily lie within the same tile because the slope and offset from the back-slicing computation is used.

5. CODE GENERATION

Before detailing the code generation algorithm, the structure of generated code is shown in Figure 14. The example shows code for a single-statement 1D stencil. Upright and inverted inter-tile loops can be found, respectively, at lines 11 and 26. Note that the upright tile loop covers an extent from the lowest bound of any stencil function to the highest bound of any stencil function, and the inverted tile tile loop extends beyond this on both sides. This is needed to ensure that boundary cells are computed as upright tiles slope away from them. It is enabled by the `max()` and `min()` expressions that prevent intra-tile loop bounds from taking values outside of the original loop's domain (lines 17 and 32). Constraints on tile size values such as `UPR_TILE_SIZE` and `INV_TILE_SIZE` are imposed as needed to ensure correctness of generated code; this is explained below.

```
1   // Time tile loop
2   for (tt = 0; tt < T; tt += TT_SIZE) {
3     // Upright tile loop
4     // F1 is a stencil function, lb_F1 (ub_F1) is the lower (upper) bound of
5     // the grid coordinate where F1 is applied
6     upr_lb = lb_F1; // for multiple stencil functions F1, F2, ..., it is
7                     // actually  min(lb_F1,lb_F2,...)
8     upr_ub = ub_F1; // Similarly max(lb_F1, lb_F2...)
9     // UPR_TILE_SIZE (INV_TILE_SIZE) is the size of the upright (inverted)
10    // tile base
11    for (ii = upr_lb; ii < upr_ub; ii += UPR_TILE_SIZE + INV_TILE_SIZE) {
12      // Time loop
13      for (t = tt; t < min(tt+TT_SIZE,T); t++) {
14        tile_lb_F1 = ii + offset_F1_lb + upr_alpha*(t-tt);
15        tile_ub_F1 = ii + UPR_TILE_SIZE +
16        offset_F1_ub + upr_beta*(t-tt) - 1;
17        for (i = max(tile_lb_F1, lb_F1); i <= min(tile_ub_F1, ub_F1); i++) {
18            // 1D stencil function code.
19        }
20      }
21    }
22    // Inverted tile loop
23    inv_lb = upr_lb - INV_TILE_SIZE;
24    inv_ub = ub_F1 + INV_TILE_SIZE; // for multiple stencil functions
25                                    // F1, F2, ... it is max(ub_F1,ub_F2,...)
26    for (ii = inv_lb; jj <  inv_ub; ii += UPR_TILE_SIZE + INV_TILE_SIZE) {
27      // Time loop
28      for (t = tt; t < min(tt+TT_SIZE,T); t++) {
29        tile_lb_F1 = ii + inv_offset_F1_lb + inv_alpha*(t-tt);
30        tile_ub_F1 = ii + INV_TILE_SIZE +
31        inv_offset_F1_ub + inv_beta*(t-tt) - 1;
32        for (i = max(tile_lb_F1, lb_F1); i <= min(tile_ub_F1, ub_F1); i++) {
33           // 1D stencil function code
34        }
35      }
36    }
37  }
```

Figure 14: Illustration of structure of generated code for 1D case

Algorithm 15 is the overall algorithm to perform DLT and split-tiling on a program that can be expressed in SDSL. It is a more detailed version of the overview algorithm shown earlier.

Functions `GetXXX` retrieve properties of the SDSL program required for code generation (e.g., name of arrays, number of dimensions, etc.). Functions `GenCopyXXXLoop` copies the data from the original layout to the DLT layout. Function `vectorize` generates a sequence of vector intrinsics call (SSE or AVX in our experiments) to compute the same arithmetic operation as the corresponding statement in the stencil function, using SIMD vectors. Function `ChangeBoundsToDLT` adapts/creates the loop structure scanning the dimension-lifted data arrays. Functions `GenMaskedXXX` use SIMD masked write operations to cope with vectors in the generated code where one of the vector slot (necessarily the first or last slot only) contains ghost data. The reader may refer to [12] for

```
Input
    P: Stencil program
    L_vec: Integer length of a vector (in elements)
Output
    P_sdlt: C code for Split-tiled, DLT version of the input
          SDSL program

  P_sdlt ← Copy(P)
  // Convert arrays to DLT.
  A ← GetArrays(P_sdlt)
  l_time ← GetTimeLoop(P_sdlt)
  foreach array a in A do
    dims ← GetDimensions(a, P_sdlt)
    l_copyin ← GenCopyInLoop(a, dims, L_vec)
    l_copyout ← GenCopyOutLoop(a, dims, L_vec)
    l_time.Prepend(l_copyin)
    l_time.Append(l_copyout)
  end do
  // Vectorize statements.
  S ← GetStatements(P_sdlt)
  foreach statement s in S do
    s_vec ← Vectorize(s, L_vec)
    l_enc ← GetEnclosingLoop(s_vec, P_sdlt)
    ChangeBoundsToDLT(l_enc)
    s_vec^start ← GenMaskedWriteStartStmt(s_vec, L_vec)
    l_start ← GenMaskedStartLoop(s_vec^start, l_enc)
    l_enc.Prepend(l_start)
    s_vec^end ← GenMaskedWriteEndStmt(s_vec, L_vec)
    l_end ← GenMaskedEndLoop(s_vec^end, l_enc)
    l_enc.Append(l_end)
  end do
  // Backslice original code.
  (x⃗) ← computeOffsetSlopes(P)
  AddBacksliceResults(P_sdlt, x⃗)
  // Create nested split-tile loops
  l_tt ← CreateTimeTileLoop()
  d ← GetSpaceDimensionCount(P_sdlt)
  GenNestedSplitTileLoops(l_tt, d)
  GenTileBodies(l_tt, P_sdlt, d)
  ReplaceTimeLoop(l_tt, P_sdlt)
```

Figure 15: LayoutTransformAndSplitTile

more details. Functions `GenNestedSplitTileLoops` and `GenTileBodies` are described below.

Algorithm 16 provides detail for the code generation of split-tiled code, in the context of a program on which DLT has been performed first. We remark that in order for the code generation of the `masked writes` required by DLT to be correct, we impose a constraint relating the size of the innermost dimension to the tile sizes:

$$|d_{innermost}| \mod (upr\_tile\_size + inv\_tile\_size) = 0$$

Adding this constraint greatly simplifies the code generation, avoiding the need to handle complex corner cases at the tile boundaries. This limitation is not a problem in practice, since padding can be used to comply with this constraint.

Function `GenInvertedFullBndryTileCondition` generates a conditional to determine whether to execute a full inverted tile or an boundary inverted tile that includes two loop nests for the start and end boundaries of DLT codes. Function `AdjustTimeAndSpaceLoopBounds` alters the loop boundaries of the loops copied from the original code to tile loop boundaries using the results of backslicing. Function `TrimMaskedWriteLoops` removes the masked write loops at the start and end of the inverted boundary tile loops, leaving the masked writes between them.

We conclude by presenting the algorithm for generating split-tiled loop nests shown in Algorithm 17. Hybrid split-tiling code generation requires minimal changes to Algorithms 16 and 17. Algorithm 17 requires a conditional to check if it is generating code for the outermost loop. If so, a standard tile loop is generated instead of a split-tiled loop. Algorithm 16 requires a change to the `AdjustTimeAndSpaceLoopBounds()` procedure to correctly traverse the standard tile dimension.

```
Input
    l: A time tile loop with empty nested split-tile loops inside
    P_sdlt: Stencil program with DLT performed and backslicing
          results inserted
    d: Integer representing current loop depth (1 innermost)
Output
    l: with split-tile loop bodies generated
  if d == 1 then
    l_orig ← GetOriginalLoopBody(P_sdlt)
    //Generate upright tile body
    l_upr ← GetUprightTileLoop(l)
    CopyTimeAndSpaceLoops(P_sdlt, l_upr, 1)
    RemoveMaskedWriteLoops(l_upr)
    AdjustTimeAndSpaceLoopBounds(l_upr, P_sdlt)

    l_inv ← GetInvertedTileLoop(l)
    (l_inv^full, l_inv^bndry) ← l.GenInvertedFullBndryTileCondition()
    //Generate inverted full tile body
    CopyTimeAndSpaceLoops(P_sdlt, l_inv^full, 1)
    RemoveMaskedWriteLoops(l_inv^full)
    AdjustTimeAndSpaceLoopBounds(l_inv^full, P_sdlt)

    //Generate inverted boundary tile body
    CopyTimeAndSpaceLoops(P_sdlt, l_inv^bndry, 2)
    AdjustTimeAndSpaceLoopBounds(l_inv^bndry, P_sdlt)
    TrimMaskedWriteLoops(l_inv^bndry)
  else
    l_upr ← GetUprightTileLoop(l)
    GenTileBodies(l_upr, P_sdlt, d − 1)
    l_inv ← GetInvertedTileLoop(l)
    GenTileBodies(l_inv, P_sdlt, d − 1)
  end if
```

Figure 16: Algorithm GenTileBodies

```
Input
    l: Loop to add nested upright and inverted tile loops to
    d: Integer representing current loop depth (1 innermost)
Output
    l: with nested upright and inverted tile loops added

  if d == 1 then
    l_upr ← l.AddUprightTileLoop()
    l_inv ← l.AddInvertedTileLoop()
  else
    l_upr ← l.AddUprightTileLoop()
    GenNestedSplitTileLoops(l_upr, d − 1)
    l_inv ← l.AddInvertedTileLoop()
    GenNestedSplitTileLoops(l_inv, d − 1)
  end if
```

Figure 17: Algorithm GenNestedSplitTileLoops

6. EXPERIMENTAL EVALUATION

The effectiveness of the both nested split-tiling and hybrid split-tiling applied in conjunction with the dimension-lifting transformation was experimentally evaluated on several hardware platforms using a variety of stencil kernels. We compare performance to the diamond-tiling system used by Pluto [2], the cache-oblivious tiling system used by Pochoir [18], and the Intel C Compiler v13.0.

6.1 Experimental Setup

Hardware: Experiments were performed on three hardware platforms with DVFS features disabled on all of them. *AMD Phenom II X6 1100T* (K10 micro-architecture) is a 6-core x86-64 chip, clocked at 3.3GHz; single-precision peak performance of 26.4 GFlop/s/-core (158.4 GFlop/s aggregate); double-precision peak performance

of 13.2 GFlop/s/core (79.2 GFlop/s aggregate). *Intel Core i7-920* (Nehalem micro-architecture) is a quad-core x86-64 chip running at 2.66 GHz; single-precision peak performance of 21.28 GFlop/s/-core (85.12 GFlop/s aggregate); double-precision peak performance of 10.64 GFlop/s/core (42.56 GFlop/s aggregate). *Intel Core i7-2600K* (Sandy Bridge micro-architecture) is a quad-core core x86-64 chip running at 3.4 GHz; single-precision peak performance of 54.4 GFlop/s/core (217.6 GFlop/s aggregate); double-precision peak performance of 27.2 GFlop/s/core (108.8 GFlop/s aggregate).

Programs were compiled using the Intel C Compiler v13.0 with the '`-O3 -ipo -xHost`' optimization flags was used for split-tiled, Pluto, and Pochoir codes on all machines. Auto-parallelization and auto-vectorization was enabled for ICC results with the '`-parallel -O3 -ipo -xHost`' optimization flags and appropriate vectorization pragmas.

Benchmarks: The following stencil codes were used (with the names used to refer to them in parentheses): Jacobi 1D (jac-1d-3), Jacobi 2D (jac-2d-5), Jacobi 3D (jac-3d-7); Laplacian, Gradient and Poisson 2D; FDTD 2D [16]; Heat 1D/2D/3D distributed with Pochoir [18].

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 244.14MB for single precision and 488.28MB for double precision; 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 work. The autotuning was done over a sampling of the set of tile size combinations that respect the various constraints (i.e., multiple of vector length) of each framework. Autotuning runs were performed for a maximum of 4 hours per SIMD unit / benchmark combination (how to speed up tile size autotuning, for instance using acceleration/search heuristics is beyond the scope of this paper, we simply tested all tile sizes in our subset). For split-tiled codes all threads (one thread per core) were assigned to the outermost parallel loop using OpenMP parallel for pragmas.

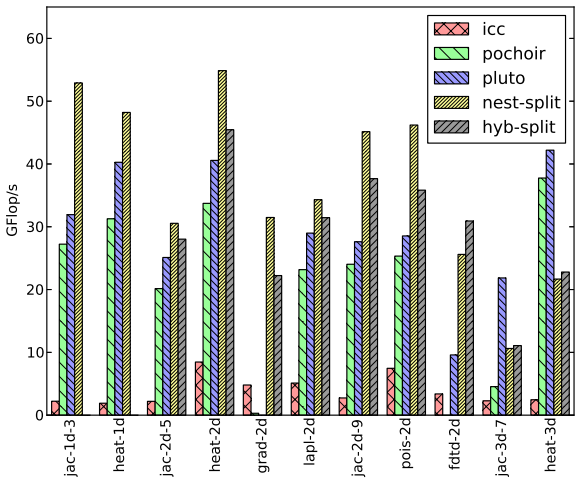
6.2 Experimental Results

Absolute performance for single and double precision experiments across all platforms and codes are given in Figures 18–20.

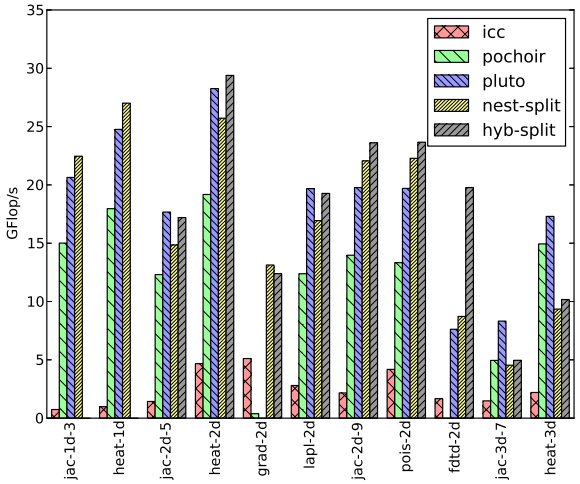
In all cases Pochoir, nested split-tiled, and hybrid split-tiled codes are significantly faster than sequential C code auto-parallelized and vectorized by Intel's ICC compiler. Not all benchmarks could be optimized by Pochoir, which can only generate optimized tiled parallel code for single statement stencil codes. Multi-statement, multi-loop stencil codes like FDTD, with three inter-related stencils updating Ex[t] using Ex[t-1] and Hz[t-1], Ey[t] using Ey[t-1] and Hz[t-1], and Hz[t] using Hz[t-1], Ex[t], and Ey[t]), cannot be expressed in Pochoir.

1D Benchmarks: Having only one split-tiled dimension allowed tiles to fit in cache and provided ample parallelism. High performance was expected of these codes and was observed. Nested split-tiling + DLT outperformed ICC, Pochoir, and Pluto on both 1D benchmarks across all platforms. Improvement over Pochoir ranged from a low of 1.27× for double precision Jacobi-1D on Nehalem to a high of 2.2× for single precision Heat-1D on Sandy Bridge. Hybrid split-tiling does not apply to 1D benchmarks as the inner loop is split-tiled for both fine grain vector parallelism and coarse grain thread-level parallelism.

2D Benchmarks: Across all 2D benchmarks hybrid split-tiling outperformed ICC, Pochoir, Pluto, and nested split-tiling on both quad core Intel platforms. For the hexacore AMD, performance



(a) Single-precision



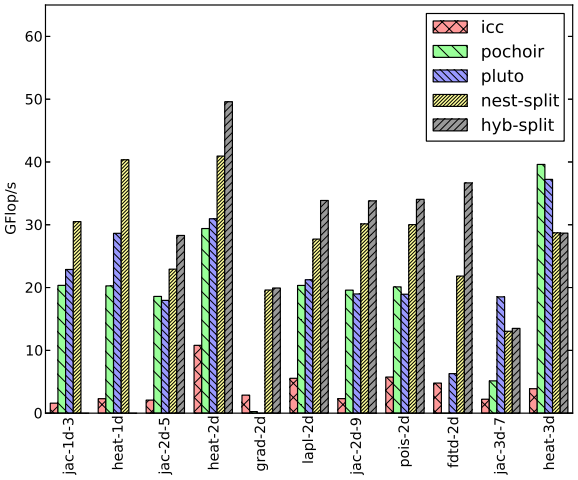
(b) Double-precision

Figure 18: AMD Phenom II X6 SSE2 Performance

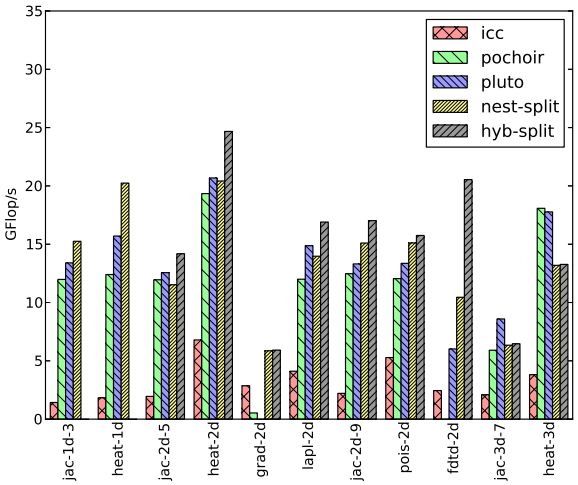
of hybrid split-tiling + DLT fell behind both nested split-tiling and Pluto on several benchmarks, both single and double precision.

This performance anomaly is due to load balancing issues. DLT effectively turned the dimension of the arrays involved from 8000 × 8000 float/double scalars to 8000 × 2000 float vectors and 8000 × 4000 double vectors. With hybrid split-tiling, tiles from the smaller dimension (subject to the constraints described in Sec. 3.2) had to be distributed across threads. Smaller tiles with better load balancing characteristics limited reuse, while larger tiles with significant reuse were not plentiful enough to adequately distribute load across cores.

Nested split-tiling exhibited better load balancing because only tiles along the larger dimension were distributed across cores. The outer dimension was large enough to allow both large tiles for significant reuse and a large quantity of tiles for load balancing. Hybrid split-tiling on quad core Intel platforms did not have any load balancing issues because the smallest dimension was divisible by 4, allowing for the same number of tiles to be distributed across all cores.

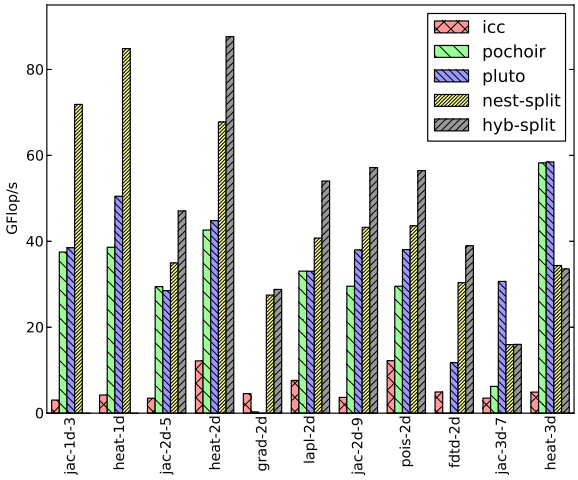


(a) Single-precision

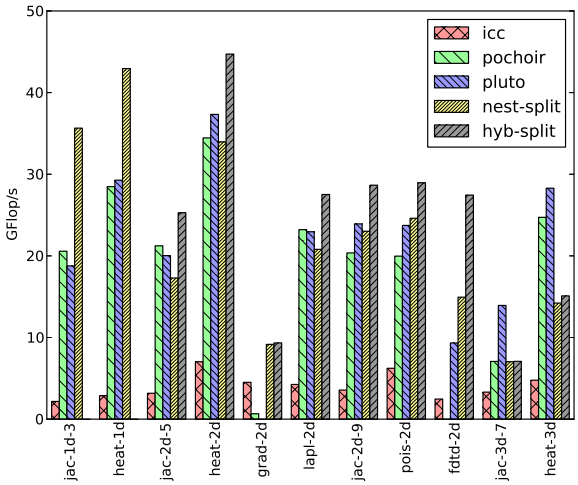


(b) Double-precision

Figure 19: Intel Core i7-920 SSE4 Performance



(a) Single-precision



(b) Double-precision

Figure 20: Intel Core i7-2600K AVX Performance

3D Benchmarks: Both hybrid and nested split-tiling fell behind Pochoir and Pluto on both 3D benchmarks across all platforms. While hybrid split-tiling was able to significantly reduce the pressure on the memory system for 2D benchmarks, its benefit was not seen when adding a third dimension. In 3D, both inner dimensions are split-tiled, thus spatial tile sizes in both inner dimensions must increase when time tile size is increased.

For both the Heat-3D and Jacobi-3D benchmarks, an increase of one in the time tile size leads to an upright tile size increase of four in each dimension. Tile sizes grow fast enough that by the time significant gains can be achieved from data reuse, the code is bound by memory latency and bandwidth consumption from spatial tiles that have grown significantly larger than L1 cache.

The diamond tiles used by Pluto overcome this limitation by halving the amount of data required for a given time tile size. For a time tile size of 16 and slopes of 2 on either side, an upright tile used in nested and hybrid split-tiling must be at least 64 elements at its base. A diamond tile with a time tile size of 16 begins at a point, expands to 32 elements at its widest and narrows to a point again. Pluto is able to reuse data significantly more than both Pochoir and nested and hybrid split-tiling.

Further exacerbating the problem is the fact that DLT causes tile sizes in the innermost dimension to be multiplied by vector size. Coupled with the constraints on tile size described in Sec. 3.2 this leads to split-tiling + DLT tile sizes (in KB) that are much larger than similar tile sizes (size of each dimension). In Heat-3D both Pluto and Pochoir are able to achieve high performance across all platforms. It is likely that the smaller tile sizes (in KB) enable Pochoir to achieve very high performance on this code.

7. RELATED WORK

A number of recent efforts have targeted the optimization of stencil computations for multicore CPUs and GPUs [2, 5, 8–10, 13, 15, 17–19]. Strzodka et al. [17] used time skewing and cache-size oblivious parallelograms to improve the memory system pressure and parallelism in stencils on CPUs. Micikevicius et al. [15] hand-tuned a 3-D finite difference computation stencil and achieved an order of magnitude performance increase over existing CPU implementations on GT200-based Tesla GPUs. Datta et al. [8] developed an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVidia GPUs, and Cell SPUs.

Tiling is a critical transformation for optimizing stencil computations since they typically perform repeated sweeps over large multi-dimensional arrays that are much too large to fit within cache. In order to benefit from both intra-step reuse as well as data reuse across several successive sweeps over the domains, so called "time tiling" is essential. The standard approach to time-tiling of stencil computations requires skewing of the iteration space and introduces inter-tile dependences along the spatial dimensions and thereby restricts parallelism to wavefront parallelism in the tile space. Alternate approaches to tiling using overlapped tiles [13], split-tiles [9] or "diamond" tiles [2, 14] enable a greater degree of inter-tile parallelism. In this paper we have developed compiler algorithms for enabling split-tiling in conjunction with a dimension-lifted-transpose data layout transformation. Grosser et al. [9] utilize a variant of split-tiling for GPGPU with substantially different tile shaping, analysis, and code generation techniques than this work.

Among the numerous research efforts to optimize stencil computations, a few of them have developed a specialized DSL compiler for stencils. PATUS [5] is a stencil compiler developed by Christen et al. that uses both a stencil description and a machine mapping description to generate efficient CPU and GPU code for stencil programs. A stencil-DSL compiler for GPUs that uses overlapped tiling for parallel execution was recently reported [13].

Tang et al. [18] have developed and publicly released the Pochoir stencil compiler that uses a DSL embedded in C++ to produce high-performance code for stencil computations using cache-oblivious parallelograms for parallel execution on shared-memory systems. In this paper, we compare performance on several multi-core systems of the code generated using DLT and split-tiling with the code generated by Pochoir, showing that we achieve comparable or better performance. But unlike the DSL compiler we have described in this paper, neither PATUS nor Pochoir can generate optimized time-tiled code for multi-statement stencil computations such as the FDTD (Finite Difference Time Domain) stencil.

The use of Chapel for the description of dense and sparse stencils was investigated by Barrett et al. [3]. The Chapel work enables automated distributed memory parallelization of stencils but does not address time-tiling or vectorization. Very recent work by Bandishti et al. [2] enhanced the Pluto compiler to incorporate a strategy for "diamond" tiling, that is particularly effective in parallelizing stencil computations. However, our approach to combining data layout transformation in conjunction with split-tiling provides significant performance advantages over Pluto for a range of stencil benchmarks, as seen from our experimental results.

8. CONCLUSIONS

In this paper, we have described compiler algorithms for a a stencil DSL incorporating two key transformations – dimension-lifted-transpose data layout transformation to optimize vector loads / stores, and split-tiling for enhanced inter-tile concurrency. Experimental results on a number of stencil benchmarks on multiple target multicore platforms demonstrate significant performance improvements achievable over state-of-the-art production compilers such as Intel's ICC, but also significant improvements over Pochoir and recent work on diamond tiling in Pluto. Our compiler has greater scope of applicability than Pochoir, effectively optimizing arbitrary multi-statement stencils.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation through awards 0926687 and 0926688, by the Center for Domain-Specific Computing (CDSC) funded by NSF "Expeditions in Computing" award 0926127, and the Department of Energy through award DE-SC0005033.

9. REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

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

[3] R. F. Barrett, P. C. Roth, and S. W. Poole. Finite difference stencils implemented using chapel. ORNL TM-2007/122, 2007.

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

[5] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, 2011.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2001.

[7] G. B. Dantzig, W. Blattner, and M. Rao. Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. 1966.

[8] 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*, 2008.

[9] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proc. GPGPU-6*, 2013.

[10] D. Han, S. Xu, L. Chen, and L. Huang. PADS: A pattern-driven stencil compiler-based tool for reuse of optimizations on GPGPUs. In *ICPADS*, 2011.

[11] T. Henretty, J. Holewinski, N. Sedaghati, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Stencil Domain Specific Language (SDSL) User Guide 0.2.1 draft. OSU TR OSU-CISRC-4/13-TR09, 2013.

[12] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector SIMD architectures. In *CC*, 2011.

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

[14] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, 2007.

[15] P. Micikevicius. 3d Finite difference computation on GPUs using CUDA. In *GPGPU-2*, 2009.

[16] D. Orozco and G. R. Gao. Mapping the FDTD Application to Many-Core Chip Architectures. In *ICPP*, 2009.

[17] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS*, 2010.

[18] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *SPAA*, 2011.

[19] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *ArXiv e-prints*, 2010.