A Domain-Specific Language and Compiler for Stencil Computations on Short-Vector SIMD and GPU Architectures

Tom Henretty, Justin Holewinski, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, P. Sadayappan

1 Ohio State University {henretty,holewinski,rountev,saday}@cse.ohio-state.edu
2 Carnegie Mellon University rveras@cmu.edu, franzf@ece.cmu.edu
3 University of California, Los Angeles pouchet@cs.ucla.edu
4 Louisiana State University jxr@ece.lsu.edu

Abstract—Stencil computations are an integral part of applications in a number of scientific computing domains, such as image processing and partial differential equations. 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, that generates optimized code for multi-core processors with short-vector SIMD engines, as well as GPUs. The hardware differences between these two types of architecture prompt different optimization strategies for the compiler. A data layout transformation along with split tiling is used for multi-core CPUs, while overlapped tiling is used for GPUs. We evaluate our domain-specific compiler for a number of benchmarks on CPU and GPU platforms.

I. INTRODUCTION

Stencils represent an important computational pattern used in scientific applications in a variety of domains including computational electromagnetics [23], solution of PDEs using finite difference or finite volume discretizations [21], and image processing for CT and MRI imaging [6], [7]. A number of recent studies have focused on optimizing stencil computations for multicore CPUs [5], [10], [12], [22], [24] and GPUs [18]–[20].

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 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 recently proposed [13], this paper describes a set of compiler transformations that are needed to generate efficient code for (i) GPUs such as NVIDIA GTX580 and AMD A8-3850, and (ii) multicore processors with short-vector SIMD ISAs such as SSE, AVX, VSX, LRBNI etc., for stencil computations.

Stencil computations considered in this paper involve the repeated updating of values associated with points on a multi-dimensional regular grid, using only the values at a set of neighboring points. For multi-core processors, stencil computations are often memory bandwidth bound when the collective data for all grid points exceeds cache size, since each grid point is accessed in each time step, i.e., iteration of the enclosing time loop. Time-tiling, i.e., tiling along the time dimension, is useful in enhancing data locality. The standard approach to time-tiling of stencil computations requires loop skewing to make tiling legal and this results in loss of inter-tile concurrency [17], since inter-tiling dependences are introduced in the spatial directions due to the skewing. The approach of “overlapped tiling” and “split tiling” [17], also called “ghost zone” optimization [6], [18], has been used for preserving concurrency in parallel time-tiling execution of stencil computations. However, we are not aware of any fully automated compiler approach for the generation of overlapped-tiling code for execution on GPUs. In one part of this paper, we describe compiler algorithms for automated GPU code generation for stencil computations and demonstrate effectiveness through experimental evaluation using a number of stencils written in our proposed stencil domain-specific language [13].

Vector operations, with ISAs like SSE or AVX, 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. As stencil computations involve arithmetic operations on physically contiguous data elements, e.g., \( c0 \ast (A[i-1]+A[i]+A[i+1]) \), they pose challenges to efficient implementation on these architectures. It requires the use of redundant and unaligned loads of data elements from memory into different slots. We had previously addressed this issue through a dimension-lifting-transpose (DLT) data layout transformation [14]. However, only sequential, L1-resident execution was addressed. In the second part of this paper, we describe 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 [24] and Pluto [2], [4] for several benchmarks on multiple target multicore processors, demonstrating strong performance benefits for 1D and 2D stencils [15].

This paper provides a brief overview of a domain-specific language dedicated to stencil computations (called SDSL) applied on a rectangular grid, with support of arbitrary rectangular sub-regions for the application of stencils, and an arbitrary number of stencil functions to be applied (that is, supporting multi stencils). It then presents target-specific compilation schemes to map this Stencil-DSL to (i) GPU devices, achieving data reuse and parallelism along all dimensions, both spatial and time, and (ii) short-vector SIMD devices, achieving data reuse and parallelism along all dimensions, both spatial and time.

The paper is organized as follows. Sec. II summarizes the stencil DSL, describing programs input to the domain-specific compilers we developed. Sec. III discusses some of the challenges in compiling regular stencil computations on GPUs and CPUs. Sec. IV provides an overview of our stencil GPU compiler and stencil CPU compiler. Sec. V presents a static analysis of the stencil DSL to extract the necessary information to enable semantically correct code generation. Experimental results are presented in Sec. VI and related work is discussed in Sec. VII.
II. STENCIL DOMAIN-SPECIFIC LANGUAGE

A. SDSL – Stencil Domain-Specific Language

In this section, we summarize the features of a Domain-Specific Language (DSL) to describe stencil computations, such that any program written in this language can be processed automatically by our compiler. Our stencil DSL models iterative methods operating on (dense) fields atop a Cartesian grid. In addition to the the data space, we describe the stencil function which is applied iteratively to each point of the grid.

More details can be found in [13].

We enforce some constraints on the DSL to facilitate effective code generation for different target architectures. First, to enable compile-time generation of overlapped/split tiled code, we need to model the halo of the stencil, and it needs to be exactly computable at compile time. This implies that the neighboring relationship remains constant during the computation. Second, in order to perform tiling along the time dimension, the computation must iterate a constant number of time steps before any field-spanning operation is performed (e.g., a convergence check). Therefore, we introduce a construct check every that let the programmer specify a constant number of time steps before any convergence check is performed. This enables time-tiling along this number of uninterrupted time iterations.

1) Program Description: As illustrative example, the computation shown in Fig. 1 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.

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

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

Fig. 1: A simple Jacobi 2D example in SDSL

**Structural Mesh (grid):** The first line of Fig. 1 defines g, the grid where stencil computations may be defined. It is an n-dimensional Cartesian coordinate space (a subset of Z^n 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. 1 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. 1 define a stencil computation. Three key concepts are defined: (1) number of time iterations, (2) subgrid(s) over which to apply a stencil function and (3) stencil function(s).

The number of time iterations 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:diml-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 results 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.

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 shown in Fig. 2. The program is constrained to be a collection of $M \times K$-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. Finally, a convergence check may be specified to execute once every $I$ iterations.

```
g [dimN] ... [diml];

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(...);
    }
}
```

check (CONVERGENCE) every I iterations

Fig. 2: General form of an SDSL program

B. Discussions

Our stencil DSL offers several features making it a good fit to describe programs that can be optimized by our compilers, as well as other compilation techniques.

First, by focusing on regular sub-grids that are are subsets of Z^n, one can generate a sequence of affine loops that exactly scan each domain of each stencil function. In addition, the separation of grid from its associated field data gives full control to the compiler regarding the data layout. This enables (1) the generation of totally disjoint arrays with the guarantee of absence of pointer aliasing; and (2) the implementation of aggressive data layout transformations such as the dimension-lift-and-transpose implemented in our CPU compiler. Regular loops, constant stencil offsets in a stencil function, and the guarantee of non-overlapping array regions enables the generation of a guaranteed correct affine program for the corresponding stencil program. This is a very strong advantage over C, where complex analysis (especially pointer analysis and inter-procedural analysis) does not provide, in the general case, the guarantee to succeed in uncovering affine control parts for a stencil region.

Second, by design, our DSL is embedded in the original program. This allows full integration of an optimized stencil region with a full-blown program that requires the constructs available in general-purpose languages. We have experimented with its integration in C/C++, as presented in this work, and are currently integrating our Stencil DSL with Matlab. One of the issue when integrating within a program is the memory representation of the input/output fields, we currently
require the corresponding arrays to be implemented using a contiguous, flat memory layout. While static analysis may fail to discover this mandatory property, using a DSL depts to the programmer the task of fulfilling this property. We believe it is a sustainable approach for scientific codes, leveraging information known to the programmer.

III. OPTIMIZATION OF STENCILS FOR CPUs AND GPUs

A. Optimizing Stencils on GPUs

Graphics Processing Units (GPUs) are massively-threaded, many-core architectures with peak floating-point throughput of over 1 TFLOPs. GPUs contain hundreds of cores (streaming processors) arranged in tightly coupled groups of 8–32 scalar processors per streaming multi-processor. Threads are grouped into thread blocks that are scheduled on a streaming multi-processor. Parallelism is exposed both across thread blocks and within thread blocks. Threads within a block are cooperative and can synchronize with each other, but threads in different blocks cannot synchronize, even if they are scheduled on the same streaming multi-processor. Each thread has access to global, off-chip memory and a shared scratch-pad memory that is shared among all threads within a block. Threads within a block can communicate and exchange data through shared memory.

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. GPUs 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 scratchpad 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 [16]. An overview is presented in the following Sec. IV-A.

B. Optimizing Stencils on Short-Vector SIMD Cores

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 provide a brief background on the DLT data layout transformation of Henretty et al. [14] that was developed to overcome the fundamental data access inefficiency on current short-vector SIMD architectures with stencil computations. This transformation is a key enabler to high-performance on short-vector SIMD architectures.


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 [14], including issues such as how elements at the DLT boundaries are handled.

IV. TILING OF STENCIL COMPUTATIONS FOR CPUs AND GPUs

In this section we present an overview of the two tiling techniques implemented in our target-specific compilers for the stencil DSL: overlapped tiling for GPUs, and split-tiling for CPUs.

A. Tiling Stencil Computations for GPUs

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 scratchpad memory before having to...
Fig. 4: 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.

Fig. 5: Overlapped Jacobi 9-Point Stencil for \(T = 2\).

<table>
<thead>
<tr>
<th>Logical Computation</th>
<th>Actual Computation at time (t)</th>
<th>Actual Computation at time (t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements needed at time (t+1)</td>
<td>Useless computation</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6: Jacobi 1D stencil

Fig. 6(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. 6(b) shows a fused form that creates a unified 2D iteration space with a statement body including both S1 and S2 (along with peeling of an iteration at the boundaries of the \(j\) 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. 7(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. (3)) to be gathered together in a single vector and therefore must be operated upon concurrently. The circled value in each tile of Fig. 7(a) represents the logical time at which the tile can be executed, such that all tiles it depends on have been previously computed.

Fig. 7(b) shows a different form of tiling – split-tiles. Upright and inverted tiles alternate and the inter-tile dependencies 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. 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 nested split-tiling, upright tiles must be sized such that they retain their characteristic trapezoidal shape, as in Fig. 7(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 \( T_i^d \), maximum absolute value of slopes in \( d s_{\text{max}}^d \), maximum offset of all statements \( o_{\text{max}}^d \), and time tile size \( T_T \) the following constraint can be stated: \( T_i^d \geq 2 * T_T * s_{\text{max}}^d + 2 * o_{\text{max}}^d \).

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 \( s_{\text{max}}^d = 2 \), maximum offset \( o_{\text{max}}^d = 0 \), and \( T_T = 8 \). This requires \( T_i^d \geq 32 \). 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) 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. 8(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 \( jj \) 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 the example at the end of Sec. IV-B2, 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.

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 combined data layout transformations for SIMD vectorization with parallel tiling for data locality. 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. 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 V. Function performSplitTiling uses the split-tile shape information computed to generate split-tile code for the \( d \) split inner spatial dimensions. Finally, function finalizeTiling completes code generation, by applying standard tiling on the remaining dimensions, if any. For additional details
A. Split-Tiling Jacobi 1D

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). The slopes of tile boundaries are identical for overlapped tiling too, the only difference being that we only use upright tiles with a different inter-tile spacing than inverted tiles.

V. 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). The slopes of tile boundaries are identical for overlapped tiling too, the only difference being that we only use upright tiles with a different inter-tile spacing than inverted tiles.

A. Split-Tiling Jacobi 1D

We illustrate the main ideas behind split-tiling using a Jacobi 1D example. Fig. 10 shows the corresponding input SDSL program.

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. 11.

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.

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]);
    }
}

Fig. 10: A simple Jacobi 1D example in SDSL

Examine 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. 12.

Fig. 12 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.

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 \(2^d+d\) 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 along 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:

\[
D_{1\rightarrow t_2} = \begin{cases} 
\text{flow: } f_1(t,i) & \rightarrow f_2(t,i) \\
\text{anti: } f_1(t,i) & \rightarrow f_2(t,i-1) \\
\text{anti: } f_1(t,i) & \rightarrow f_2(t,i) \\
\text{anti: } f_1(t,i) & \rightarrow f_2(t,i+1) 
\end{cases}
\]

\[
D_{t_2\rightarrow f_1} = \begin{cases} 
\text{flow: } f_2(t,i) & \rightarrow f_1(t+1,i-1) \\
\text{flow: } f_2(t,i) & \rightarrow f_1(t+1,i) \\
\text{anti: } f_2(t,i) & \rightarrow f_1(t+1,i+1) \\
\text{anti: } f_2(t,i) & \rightarrow f_1(t+1,i) 
\end{cases}
\]

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:

\[
d_{f_1\rightarrow t_2} = \begin{cases} 
\delta_T = 0, \delta_i = < 0 > \\
\delta_T = 0, \delta_i = < -1 > \\
\delta_T = 0, \delta_i = < 0 > \\
\delta_T = 0, \delta_i = < 1 >
\end{cases} \quad d_{t_2\rightarrow f_1} = \begin{cases} 
\delta_T = 1, \delta_i = < -1 > \\
\delta_T = 1, \delta_i = < 0 > \\
\delta_T = 1, \delta_i = < 1 > \\
\delta_T = 1, \delta_i = < 0 >
\end{cases}
\]

The spatial components of the distance vectors are then coalesced into a tuple for each dependence such that the coalesced tuple \(C_{f_1\rightarrow f_1} = < \delta_T, \delta_i, \delta_U >\) where \(\delta_T\) is the maximum spatial distance and \(\delta_U\) is the minimum spatial distance between two dependent statements \(f_s\) and \(f_t\). For the Jacobi 1D example the tuples for each dependence are identical, \(C_{f_1\rightarrow f_1} = C_{f_2\rightarrow f_1} = < 1, -1 >\). These tuples are used to label edges in the DSG, along with a separate label for the time distance \(\delta_T\). Assembling the coalesced tuples, time distances, dependences, and statements leads to the DSG shown in Fig. 13 for the Jacobi 1D example.

This DSG is subsequently used in Sec. V-A3 to build validity constraints for split-tiles and in Sec. V-A4 to compute slopes and statement offsets.

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. 14. Informally, the validity constraints state that, for any pair of dependent statements, given a region over which the target statement is executed, the source statement will be executed over a region that is, at minimum, large enough to satisfy the dependence.

\[
\text{for } (t = \ldots) \\
\text{for } (i = \ldots) \\
\text{for } (i = t + T_0 + o^L_t + \alpha \ast t + 1, t + +) \\
\text{for } (i = t + T_0 + o^L_t + \beta \ast t + 1, i + +) \\
\text{for } (i = t + T_0 + o^L_t + \alpha \ast t + 1, i + +) \\
\text{for } (i = t + T_0 + o^L_t + \beta \ast t + 1, i + +) \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{end for}
\]

Fig. 13: Dependence Summary Graph (DSG) for Jacobi 1D

Fig. 14: Loop nests for Jacobi 1D upright tile. Time tile size is \(T_f\); upright space tile size is \(T_u\). Offsets from lower bound are \(o^L_t\) and \(o^U_t\); offsets from upper bound are \(o^L_u\) and \(o^U_u\). Slope of lower bound is \(\alpha\); slope of upper bound is \(\beta\).

From the DSG, we note that to compute statement \(f_2\) over some range \([A,B]\) at timestep \(T\) we require that statement \(f_1\) be executed over the range \([A-1,B-(-1)]\) at timestep \(T\). Similarly, to compute statement \(f_1\) over some range \([C,D]\) at timestep \(T\) we require that statement \(f_1\) 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. 14 gives us validity constraints on values the loop bounds may take, and results in the following system of inequalities for lower bounds:

\[
ii + o^L_t + \alpha \ast t \leq ii + o^U_t + \alpha \ast t - 1 \\
ii + o^L_t + \alpha \ast (t - 1) \leq ii + o^U_t + \alpha \ast t - 1
\]
The following system constrains the upper bounds:

\[ ii + T_U + \alpha f_U^1 + \beta \cdot o f_U^1 + \delta f_U^1 \cdot \rho \geq \alpha f_U^1 + \beta \cdot o f_U^1 + \delta f_U^1 \cdot \rho + 1 \]

\[ ii + T_U + \alpha f_U^2 + \beta \cdot o f_U^2 + \delta f_U^2 \cdot \rho \geq \alpha f_U^2 + \beta \cdot o f_U^2 + \delta f_U^2 \cdot \rho + 1 \]

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

\[ \alpha f_U^1 - \beta f_U^1 \leq -1 \]

\[ \alpha f_U^2 - \beta f_U^2 \leq -1 \]

\[ \alpha f_U^2 - \beta f_U^2 \leq -\alpha - 1 \]

the corresponding constraints for upper bounds are shown below:

\[ \alpha f_U^1 - \beta f_U^1 \leq -1 \]

\[ \alpha f_U^2 - \beta f_U^2 \leq -1 \]

These systems are used in Sec. V-A4 to compute valid offsets for all statements.

4) Computing Slopes and Offsets: To determine legal values for slopes \( \alpha \) and \( \beta \) we compute, respectively, maximum and minimum cycle ratios \( \rho \) from the following system of difference constraints for lower bounds:

\[ \alpha f_U^1 - \beta f_U^1 \leq -1 \]

\[ \alpha f_U^2 - \beta f_U^2 \leq -\alpha - 1 \]

A cycle ratio \( \rho \) on the DSG is computed by finding a cycle, \( \delta t \), over the cycle, and dividing by the sum of \( \delta T \) values. A cycle ratio \( \rho \) on the DSG is calculated in a similar fashion with \( \delta t \) values. We set \( \alpha = \max(\rho(C)) \) and \( \beta = \min(\rho(C)) \) for all cycles \( C \) in the DSG.

We examine the only cycle in our example DSG, \( C_0 \), between \( t_1 \) and \( t_2 \). The DSG tells us that \( \rho(C) + 1 \) on some interval \([A, B]\) at time \( T \) allows us to compute \( \delta f_U^1 \) on the interval \([A + \delta T f_U^1 - \delta T f_U^2, B + \delta T f_U^1 - \delta T f_U^2] \) at time \( T + \delta T f_U^1 - \delta T f_U^2 \) without violating any dependences. Continuing along the cycle, computing \( \delta f_U^2 \) on the interval \([A + \delta T f_U^1 - \delta T f_U^2, B + \delta T f_U^1 - \delta T f_U^2] \) at time \( T + \delta T f_U^1 - \delta T f_U^2 \) allows us to compute \( \delta f_U^1 \) on the interval \([A + \delta T f_U^1 - \delta T f_U^2, B + \delta T f_U^1 - \delta T f_U^2 + \delta T f_U^1 - \delta T f_U^2] \) at time \( T + \delta T f_U^1 - \delta T f_U^2 \) on some 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 \( \delta T \) values and dividing by the sum of \( \delta T \) values gives us \( \rho(C_0) = 2 \); a similar calculation gives us \( \rho(C_0) = -2 \). Since there is only one cycle in the DSG, these values are \( \max(\rho(C)) \) and \( \min(\rho(C)) \), and we set \( \alpha = \max(\rho(C)) = 2 \) and \( \beta = \min(\rho(C)) = -2 \).

These values for \( \alpha \) and \( \beta \) are substituted into the systems of difference constraints, and a solution to each of these systems is obtained using the Bellman-Ford algorithm [1], [8]. For the Jacobi 1D example we obtain \( \alpha f_U^1 = 1 \), \( \alpha f_U^2 = 0 \), and \( \beta f_U^1 = \beta f_U^2 = 0 \).

VI. EXPERIMENTAL EVALUATION

The effectiveness of the nested split-tiling and hybrid split-tiling applied in conjunction with the dimension-lifting transformation was experimentally evaluated on two 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 [24], and the Intel C Compiler v13.0. Overlapped tiling was evaluated on an NVIDIA GTX 580 using the CUDA 5.0 SDK.

A. Experimental Setup

**Hardware:** CPU experiments were performed on two hardware platforms with DVFS features disabled. **AMD Phenom II X6 1100T** (K10 micro-architecture) is a 6-core x86-64 chip, clocked at 3.3GHz; double-precision peak performance is 13.2 GFlop/s/core (79.2 GFlop/s aggregate). **Intel Core i7-2600K** (Sandy Bridge micro-architecture) 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).

GPU experiments were performed on the **NVIDIA GTX 580** (Fermi microarchitecture) clocked at 1.54GHz with 512 cores; double-precision peak performance is 386 MFlop/s/core (198 GFlop/s aggregate).

**CPU programs:** were compiled using the Intel C Compiler v13.0 with the ‘-03 -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 -03 -ipo -xHost’ optimization flags and appropriate vectorization pragmas. GPU codes were compiled using NVCC and the ‘-03 -xcompiler -03’ optimization flags for CUDA compute capability 2.0.

**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-9), Jacobi 3D (jac-3d-7), Laplacian 2D (lapl-2d), Gradient 2D (grad-2d) Heat 1D/2D/3D (heat-2d) Heat 1D/2D/3D (heat-nd) distributed with Pochoir [24].

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-tiled and overlapped 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 microarchitecture / 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.

**B. Experimental Results**

Absolute performance for double precision experiments across all platforms and codes are given in Figures 15–17.

![Fig. 15: AMD Phenom II X6 SSE2 Performance](image-url)
effectively turned the dimension of the arrays involved from 8000 × 8000 scalars to 8000 × 4000 vectors. With hybrid split-tiling, tiles from the smaller dimension (subject to the constraints described in Sec. IV-B2) 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 Sandy Bridge 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.

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 upward 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. IV-B2 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.

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 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.

VII. Related Work

A number of recent efforts have targeted the optimization of stencil computations for multicore CPUs and GPUs [2], [5], [10]–[12], [16], [19], [22], [24], [25]. Strzodka et al. [22] used time skewing and cache-size oblivious parallelograms to improve the memory system pressure and parallelism in stencils on CPUs. Micikevicius et al. [19] 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. [10] developed an optimization and auto-tuning framework for stencil computations, targeting multicore systems, NVidia GPUs, and Cell SPUs.

Tiling is a critical transformation for optimizing stencil computations since they typically perform repeated sweeps over large multidimensional 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 \cite{16}, split-tiles \cite{11} or “diamond” tiles \cite{2, 17} 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. \cite{11} 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 \cite{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 \cite{16}.

Tang et al. \cite{24} 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 parallel-legalrams 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-tilded 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. \cite{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. \cite{2} enhanced the Plato 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 Plato for a range of stencil benchmarks, as seen from our experimental results.

VIII. CONCLUSIONS

In this paper, we provided a brief overview of a domain-specific language for regular stencil computations, that allows specification of the computations in a concise manner. We then described a multi-target compiler for this DSL, that generates optimized code for multi-core processors with short-vector SIMD engines, as well as GPUs. The hardware differences between these two types of architecture prompted different optimization strategies for the compiler. For short-vector SIMD architectures, we have described compiler algorithms incorporating two key transformations – dimension-lifted-transpose data layout transformation to optimize vector loads and stores, and split-tiling for enhanced inter-tile concurrency. For GPUs, we have used overlapped tiling. Experimental results on a number of stencil benchmarks on multiple target 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 Plato. 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 U.S. Army through contract W911NF-10-1-0004, 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.

References

\begin{thebibliography}{99}
\bibitem{13} 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.
\bibitem{19} P. Miciekiewicz. 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.
\bibitem{20} 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 \textit{IEEE Conference of Supercomputing}, SC ’10, pages 1–13, 2010.
\end{thebibliography}