Pochoir: A Compiler for Stencil Computations

Yuan Tang — Fudan University
Rezaul Chowdhury — Stony Brook University
Bradley C. Kuszmaul — MIT
Chi-Keung Luk — Intel Corporation
Charles E. Leiserson — MIT

Abstract

A stencil computation repeatedly updates each point of a d-dimensional grid as a function of itself and its near neighbors. Parallel cache-efficient stencil algorithms based on "trapezoidal decompositions" are known, but most programmers find them difficult to write. The Pochoir stencil compiler allows one to write a simple specification of a stencil in a domain-specific stencil language embedded in C++ which the Pochoir compiler then translates into high-performing Cilk code that employs an efficient parallel cache-oblivious algorithm. Pochoir supports general d-dimensional stencils and handles both periodic and aperiodic boundary conditions in one unified algorithm. The Pochoir system provides a C++ template library that allows the user's stencil specification to be executed directly in C++ without the Pochoir compiler (albeit more slowly), which simplifies user debugging and greatly simplified the implementation of the Pochoir compiler itself. A host of stencil benchmarks run on a modern multicore machine demonstrates that Pochoir outperforms standard parallel-loop implementations, typically running 2–10 times faster. The algorithm behind Pochoir improves on prior cache-efficient algorithms on multidimensional grids by making "hyperspace" cuts, which yield asymptotically more parallelism for the same cache efficiency.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming — Parallel programming; D.3.2 [*Programming Languages*]: Language Classifications — Specialized application languages; G.4 [*Mathematical Software*]: Algorithm design and analysis.

Keywords C++, cache-oblivious algorithm, Cilk, compiler, embedded domain-specific language, multicore, parallel computation, stencil, trapezoidal decomposition.

1. Introduction

Pochoir (pronounced "PO-shwar") is a compiler and runtime system for implementing stencil computations on multicore processors. A ***stencil*** defines the value of a grid point in a d-dimensional spatial grid at time t as a function of neighboring grid points at recent times before t. A ***stencil computation*** [2, 6, 8, 9, 13, 14, 22–24, 29–31, 33, 35] computes the stencil for each grid point over many time steps.

Stencil computations are conceptually simple to implement using nested loops, but looping implementations suffer from poor cache performance. Cache-oblivious [12] divide-and-conquer stencil codes [13, 14] are much more efficient, but they are difficult to write, and when parallelism is factored into the mix, most application programmers do not have the programming skills or patience to produce efficient multithreaded codes.

As an example, consider how the 2D ***heat equation*** [10]

$$\frac{\partial u_t(x,y)}{\partial t} = \alpha\left(\frac{\partial^2 u_t(x,y)}{\partial x^2} + \frac{\partial^2 u_t(x,y)}{\partial y^2}\right)$$

on an $X \times Y$ grid, where $u_t(x,y)$ is the heat at a point (x,y) at time t and α is the thermal diffusivity, might be solved using a stencil computation. By discretizing space and time, this partial differential equation can be solved approximately by using the following Jacobi-style update equation:

$$\begin{aligned} u_{t+1}(x,y) &= u_t(x,y) \\ &\quad + \frac{\alpha\Delta t}{\Delta x^2}\left(u_t(x-1,y) + u_t(x+1,y) - 2u_t(x,y)\right) \\ &\quad + \frac{\alpha\Delta t}{\Delta y^2}\left(u_t(x,y-1) + u_t(x,y+1) - 2u_t(x,y)\right) . \end{aligned}$$

One simple parallel program to implement a stencil computation based on this update equation is with a triply nested loop, as shown in Figure 1. The code is invoked as LOOPS$(u; 0, T; 0, X; 0, Y)$ to perform the stencil computation over T time steps. Although the loop indexing the time dimension is serial, the loops indexing the spatial dimensions can be parallelized, although as a practical matter, only the outer loop needs to be parallelized. There is generally no need to store the entire space-time grid, and so the code uses two copies of the spatial grid, swapping their roles on alternate time steps. This code assumes that the boundary conditions are ***periodic***, meaning that the spatial grid wraps around to form a torus, and hence the index calculations for x and y are performed modulo X and Y, respectively.

This loop nest is simple and fairly easy to understand, but its performance may suffer from poor cache locality. Let $\mathcal{M}$ be the number of grid points that fit in cache, and let $\mathcal{B}$ be the number of grid points that fit on a cache line. If the space grid does not fit in cache — that is, $XY \gg \mathcal{M}$ — then this simple computation incurs $\Theta(TXY/\mathcal{B})$ cache misses in the ideal-cache model [12].

Figure 2 shows the pseudocode for a more efficient cache-oblivious algorithm called TRAP, which is the basis of the algorithm used by the Pochoir compiler. We shall explain this algorithm in Section 3. It achieves $\Theta(TXY/\mathcal{B}\sqrt{\mathcal{M}})$ cache misses, assuming that $X \approx Y$ and $T = \Omega(X)$. TRAP easily outperforms LOOPS on large data sets. For example, we ran both algorithms on a 5000×5000 spatial grid iterated for 5000 time steps using the Intel C++ version 12.0.0 compiler with Intel Cilk Plus [20] on a 12-core Intel Core i7 (Nehalem) machine with a private 32-KB

This manuscript is an abridged version of a paper [34] that originally appeared in the proceedings of the *23nd Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA) in 2011.

This work was supported in part by a grant from Intel Corporation and in part by NSF grants CCF-0937860 and CNS-1017058.

Yuan Tang is Assistant Professor of Computer Science at Fudan University in China. Bradley C. Kuszmaul is Research Scientist at MIT CSAIL and Chief Architect at Tokutek, Inc. Chi-Keung Luk is Senior Staff Engineer at Intel Corporation and a Research Affiliate at MIT CSAIL. Rezaul Chowdhury is an Assistant Professor of Computer Science at Stony Brook University and a Research Affiliate at MIT CSAIL. Charles E. Leiserson is Professor of Computer Science and Engineering at MIT CSAIL.

WOSC'13, Oct 27, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/

```
LOOPS(u; ta, tb; xa, xb; ya, yb)
for t = ta to tb − 1
    parallel for x = xa to xb − 1
        for y = ya to ya − 1
            u((t+1) mod 2, x, y) = u(t mod 2, x, y)
                + CX · (u(t mod 2, (x−1) mod X, y)
                + u(t mod 2, (x+1) mod X, y) − 2u(t mod 2, x, y))
                + CY · (u(t mod 2, x, (y−1) mod Y)
                + u(t mod 2, x, (y+1) mod Y) − 2u(t mod 2, x, y))
```

Figure 1: A parallel looping implementation of a stencil computation for the 2D heat equation with periodic boundary conditions. The array u keeps two copies of an $X \times Y$ array of grid points, one for time t and one for time $t+1$. The parameters ta and tb are the beginning and ending time steps, and xa, xb, ya, and yb are the coordinates defining the region of the array u on which to perform the stencil computation. The constants $CX = \alpha\Delta t/\Delta x^2$ and $CY = \alpha\Delta t/\Delta y^2$ are precomputed. The call LOOPS$(u; 0, T; 0, X; 0, Y)$ performs the stencil computation over the whole 2D array for T time steps.

```
TRAP(u; ta, tb; xa, xb, dxa, dxb; ya, yb, dya, dyb)
Δt = tb − ta
Δx = max{xb − xa, (xb + dxbΔt) − (xa + dxaΔt)} // Longer x-base
Δy = max{yb − ya, (yb + dybΔt) − (ya + dyaΔt)} // Longer y-base
k = 0 // Try hyperspace cut
if Δx ≥ 2σ_x Δt
    Trisect the zoid with x-cuts, and increment k by 1
if Δy ≥ 2σ_y Δt
    Trisect the zoid with y-cuts, and increment k by 1
if k > 0
    Assign dependency levels 0, 1, ..., k to subzoids
    for i = 0 to k // for each dependency level i
        parallel for all subzoids
                (ta, tb; xa′, xb′, dxa′, dxb′; ya′, yb′, dya′, dyb′)
                with dependency level i
            TRAP(ta, tb; xa′, xb′, dxa′, dxb′; ya′, yb′, dya′, dyb′)
elseif Δt > 1 // time cut
    // Recursively walk the lower zoid and then the upper
    TRAP(ta, ta + Δt/2; xa, xb, dxa, dxb; ya, yb, dya, dyb)
    TRAP(ta + Δt/2, tb; xa + dxaΔt/2, xb + dxbΔt/2, dxa, dxb;
        ya + dyaΔt/2, yb + dybΔt/2, dya, dyb)
else // base case
    for t = ta to tb − 1
        for x = xa to xb − 1
            for y = ya to yb − 1
                u((t+1) mod 2, x, y) = u(t mod 2, x, y)
                    + CX · (u(t mod 2, (x−1) mod X, y)
                    + u(t mod 2, (x+1) mod X, y) − 2u(t mod 2, x, y))
                    + CY · (u(t mod 2, x, (y−1) mod Y)
                    + u(t mod 2, x, (y+1) mod Y) − 2u(t mod 2, x, y))
        xa += dxa, xb += dxb, ya += dya, yb += dyb
```

Figure 2: The Pochoir cache-oblivious algorithm that implements a 2D stencil computation to solve the 2D heat equation using a trapezoidal decomposition with hyperspace cuts. The parameter u is an $X \times Y$ array of grid points. The remaining variables describe the hypertrapezoid, or "zoid," embedded in space-time that is being processed: ta and tb are the beginning and ending time steps; xa, xb, ya, and yb are the coordinates defining the base of the zoid; dxa, dxb, dya, and dyb are the inverse slopes of the sides of the zoid. The values σ_x and σ_y are the slopes of the stencil in the x- and y-dimensions, respectively, which are both 1 for the heat equation.

L1-data-cache, a private 256-KB L2-cache, and a shared 12-MB L3-cache. The code based on LOOPS ran in 248 seconds, whereas the Pochoir-generated code based on TRAP required about 24 seconds, more than a factor of 10 performance advantage.

Figure 3 shows Pochoir's performance on a wider range of benchmarks, including heat equation (Heat) [10] on a 2D grid, a 2D torus, and a 4D grid; Conway's game of Life (Life) [15]; 3D finite-difference wave equation (Wave) [28]; lattice Boltzmann method (LBM) [26]; RNA secondary structure prediction (RNA) [1, 3]; pairwise sequence alignment (PSA) [16]; longest common subsequence (LCS) [4]; and American put stock option pricing (APOP) [21]. Pochoir achieves a substantial performance improvement over a straightforward loop parallelization for typical stencil applications, such as Heat and Life. Even LBM, which is a complex stencil having many states, achieves good speedup. When Pochoir does not achieve as much speedup over the loop code, it is often due to the spatial grid being too small to yield good parallelism, the innermost loop containing many branch conditionals, or the benchmark containing a high ratio of memory accesses to floating-point operations. For example, RNA's small grid size of 300^2 yields a parallelism of just over 5 for both Pochoir and parallel loops, and its innermost loop contains many branch conditionals. PSA operates over a diamond-shaped domain, and so the application employs many conditional branches in the kernel in order to distinguish interior points from exterior points. These overheads can sometimes significantly mitigate a cache-efficient algorithm's advantage in incurring fewer cache misses.

The Berkeley autotuner [5, 22, 35] focuses on optimizing the performance of stencil kernels by automatically selecting tuning parameters. Their work serves as a good benchmark for the maximum possible speedup one can get on a stencil. K. Datta and S. Williams graciously gave us their code for computing a 7-point stencil and a 27-point stencil on a 258^3 grid with "ghost cells" (see Section 4) using their system. Unfortunately, we were unable to reproduce the reported results from [5] — presumably because there were too many differences in hardware, compilers, and operating system — and thus we are unable to offer a direct side-by-side comparison. Instead, we present in Figure 4 a comparison of our results to their reported results.

We tried to make the operating conditions of the Pochoir tests as similar as possible to the Berkeley environment reported in [5]. We compared Pochoir running 8 worker threads on a 12-core system to the reported numbers for the Berkeley autotuner running 8 threads on 8 cores. The comparison may result in a disadvantage to the Berkeley autotuner, because their reported numbers involve only a single time step, whereas the Pochoir code runs for 200 time steps. (It does not make sense to run Pochoir for only 1 time step, since its efficiency is in large measure due to the temporal locality of cache use.) Likewise, the Pochoir figures may exhibit a disadvantage compared with the Berkeley ones, because Pochoir had to cope with load imbalances due to the scheduling of 8 threads on 12 cores. Notwithstanding these issues, as can be seen from the figure, Pochoir's performance is generally comparable to that of the Berkeley autotuner on these two benchmarks.

The Pochoir-generated TRAP code is a cache-oblivious [12] divide-and-conquer algorithm based on the notion of ***trapezoidal decompositions*** introduced by Frigo and Strumpen [13, 14]. We improve on their code by using ***hyperspace*** cuts, which produce an asymptotic improvement in parallelism while attaining essentially the same cache efficiency. As can be seen from Figure 2, however, the resulting parallel code is far more complex than LOOPS, involving recursion over irregular geometric regions. Moreover, TRAP presents many opportunities for optimization, including coarsening the base case of the recursion and handling boundary conditions. We contend that one cannot expect average application programmers to be able to write such complex high-performing code for each stencil computation they wish to perform.

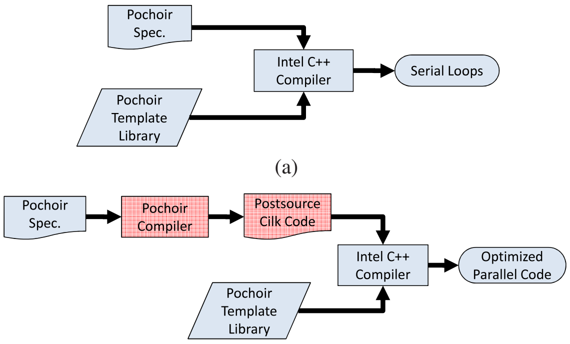
The Pochoir stencil compiler allows programmers to write simple functional specification for arbitrary d-dimensional stencils, and then it automatically produces a highly optimized, cache-efficient, parallel implementation. The Pochoir language can be viewed as a domain-specific language [7, 18, 27] embedded in the base language C++ with the Cilk multithreading extensions [20].

Bench -mark	Dims	Grid size	Time steps	Pochoir 1 core	Pochoir 12 cores	Pochoir speedup	Serial loops time	Serial loops ratio	12-core loops time	12-core loops ratio
Heat	2	$16,000^2$	500	277s	24s	115	612s	25.5	149s	6.2
Heat	2p	$16,000^2$	500	281s	24s	117	1,647s	68.6	248s	10.3
Heat	4	150^4	100	154s	54s	29	433s	8.0	104s	1.9
Life	2p	$16,000^2$	500	345s	28s	123	2,419s	86.4	332s	11.9
Wave	3	$1,000^3$	500	3,082s	447s	69	3,170s	7.1	1,071s	2.4
LBM	3	$100^2 \times 130$	3,000	345s	68s	51	304s	4.5	220s	3.2
RNA	2	300^2	900	90s	20s	45	121s	6.1	26s	1.3
PSA	1	100,000	200,000	105s	18s	58	432s	24.0	77s	4.3
LCS	1	100,000	200,000	57s	9s	63	105s	11.7	27s	3.0
APOP	1	2,000,000	10,000	43s	4s	107	515s	128.8	48s	12.0

Figure 3: Pochoir performance on an Intel Core i7 (Nehalem) machine. The stencils are nonperiodic unless the *Dims* column contains a "p." The header *Serial loops* means a serial `for` loop implementation running on one core, whereas 12-*core loops* means a parallel `cilk_for` loop implementation running on 12 cores. The header *ratio* indicates how much slower the looping implementation is than the 12-core Pochoir implementation. For nonperiodic stencils, the looping implementations employ ghost cells [5] to avoid boundary processing.

	Berkeley	*Pochoir*
CPU (2.66GHz)	Xeon X5550	Xeon X5650
#sockets, cores/socket	2, 4 (8 cores)	2, 6 (12 cores)
Hyperthreading	Enabled	Disabled
L1 data cache/core	32KB	32KB
L2 cache/core	256KB	256KB
L3 cache/socket	8MB	12 MB
Peak computation	85 GFLOPS	120 GFLOPS
Compiler	icc 10.0.0	icc 12.0.0
Linux kernel		2.6.32
Threading model	Pthreads	Cilk Plus
3D 7-point 8 cores	2.0 GStencil/s 15.8 GFLOPS	2.49 GStencil/s 19.92 GFLOPS
3D 27-point 8 cores	0.95 GStencil/s 28.5 GFLOPS	0.88 GStencil/s 26.4 GFLOPS

Figure 4: A comparison of Pochoir to the reported results from [5]. The 7-point stencil requires 8 floating-point operations per grid point, whereas the 27-point stencil requires 30 floating-point operations per grid point.



(b)

Figure 5: Pochoir's two-phase compilation strategy. (a) During Phase 1 the programmer uses the normal Intel C++ compiler to compile his or her code with the Pochoir template library. Phase 1 verifies that the programmer's stencil specification is Pochoir compliant. (b) During Phase 2 the programmer uses the Pochoir compiler, which acts as a preprocessor to the Intel C++ compiler, to generate optimized multithreaded Cilk code.

```
1   #define mod(r,m) ((r)%(m) + ((r)<0)? (m):0)

2   Pochoir_Boundary_2D(heat_bv, a, t, x, y)
3     return a.get(t,mod(x,a.size(1)),mod(y,a.size(0)));
4   Pochoir_Boundary_End

5   int main(void) {
6     const int X = 1000, Y = 1000, T = 1000;

7     Pochoir_Shape_2D 2D_five_pt[] = {{1,0,0}, {0,0,0},
            {0,1,0}, {0,-1,0}, {0,0,-1}, {0,0,1}};
8     Pochoir_2D heat(2D_five_pt);

9     Pochoir_Array_2D(double) u(X, Y);
10    u.Register_Boundary(heat_bv);
11    heat.Register_Array(u);

12    Pochoir_Kernel_2D(heat_fn, t, x, y)
13      u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x,
           y) + u(t, x-1, y)) + CY * (u(t, x, y+1) - 2
            * u(t, x, y) + u(t, x, y-1)) + u(t, x, y);
14    Pochoir_Kernel_End

15    for (int x = 0; x < X; ++x)
16      for (int y = 0; y < Y; ++y)
17        u(0, x, y) = rand();

18    heat.Run(T, heat_fn);

19    for (int x = 0; x < X; ++x)
20      for (int y = 0; y < Y; ++y)
21        cout << u(T, x, y);


23    return 0;
24  }
```

Figure 6: The Pochoir stencil source code for a periodic 2D heat equation. Pochoir keywords are boldfaced.

As shown in Figure 5, the Pochoir system operates in two phases, only the second of which involves the Pochoir compiler itself. For the first phase, the programmer compiles the source program with the ordinary Intel C++ compiler using the Pochoir template library, which implements Pochoir's linguistic constructs using unoptimized but functionally correct algorithms. This phase ensures that the source program is ***Pochoir-compliant***. For the second phase, the programmer runs the source through the Pochoir compiler, which acts as a preprocessor to the Intel C++ compiler, performing a source-to-source translation into a postsource C++ program that employs the Cilk extensions. The postsource is then compiled with the Intel compiler to produce the optimized binary executable. The Pochoir compiler makes the following promise:

> **The Pochoir Guarantee:** If the stencil program compiles and runs with the Pochoir template library during Phase 1, no errors will occur during Phase 2 when it is compiled with the Pochoir compiler or during the subsequent running of the optimized binary.

Pochoir's novel two-phase compilation strategy allowed us to build significant domain-specific optimizations into the Pochoir compiler without taking on the massive job of parsing and type-checking the full C++ language. Knowing that the source program compiles error-free with the Pochoir template library during Phase 1 allows the Pochoir compiler in Phase 2 to treat portions of the source as uninterpreted text, confident that the Intel compiler will compile it correctly in the optimized postsource. Moreover, the Pochoir template library allows the programmer to debug his or her code using a comfortable native C++ tool chain without the complications of the Pochoir compiler.

Figure 6 shows the Pochoir source code for the periodic 2D heat equation. We leave the specification of the Pochoir language to Section 2, but outline the salient features of the language using this code as an example.

Line 7 declares the ***Pochoir shape*** of the stencil, and line 8 creates the 2-dimensional ***Pochoir object*** `heat` having that shape. The Pochoir object will contain all the state necessary to perform the computation. Each triple in the array `2D_five_pt` corresponds to a relative offset from the space-time grid point (t,x,y) that the stencil kernel (declared in lines 12–14) will access. The compiler cannot infer the stencil shape from the kernel, because the kernel can be arbitrary code, and accesses to the grid points can be hidden in subroutines. The Pochoir template library complains during Phase 1, however, if an access to a grid point during the kernel computation falls outside the region specified by the shape declaration.

Line 9 declares `u` as an `X`×`Y` ***Pochoir array*** of double-precision floating-point numbers representing the spatial grid. Lines 2–4 define a ***boundary function*** that will be called when the kernel function accesses grid points outside the computing domain, that is, if it tries to access `u(t, x, y)` with `x` < 0, `x` $\geq$ `X`, `y` < 0, or `y` $\geq$ `Y`. The boundary function for this periodic stencil performs calculations modulo the dimensions of the spatial grid. (Section 2 shows how nonperiodic stencils can be specified, including how to specify Dirichlet and Neumann boundary conditions [11].) Line 10 associates the boundary function `heat_bv` with the Pochoir array `u`. Each Pochoir array has exactly one boundary function to supply a value when the computation accesses grid points outside of the computing domain. Line 11 registers the Pochoir array `u` with the `heat` Pochoir object. A Pochoir array can be registered with more than one Pochoir object, and a Pochoir object can have multiple Pochoir arrays registered.

Lines 12–14 define a ***kernel function*** `heat_fn`, which specifies how the stencil is computed for every grid point. This kernel can be an arbitrary piece of code, but accesses to the registered Pochoir arrays must respect the declared shape(s).

Lines 15–17 initialize the Pochoir array `u` with values for time step 0. If a stencil depends on more than one prior step as indicated by the Pochoir shape, multiple time steps may need to be initialized. Line 18 executes the stencil object `heat` for T time steps using kernel function `heat_fn`. Lines 19–21 prints the result of the computation by reading the elements `u(T, x, y)` of the Pochoir array. In fact, Pochoir overloads the "<<" operator so that the Pochoir array can be pretty-printed by simply writing "`cout << u;`".

The remainder of this paper is organized as follows. Section 2 provides a full specification of the Pochoir embedded language. Section 3 describes the cache-oblivious parallel algorithm used by the compiled code and analyzes its theoretical performance. Section 4 describes four important optimizations employed by the Pochoir compiler. Section 5 offers some concluding remarks.

2. The Pochoir Specification Language

This section describes the formal syntax and semantics of the Pochoir language, which was designed with a view to offer as much expressiveness as possible without violating the Pochoir Guarantee. Since we wanted to allow third-party developers to implement their own stencil compilers that could use the Pochoir specification language, we avoided to the extent possible making the language too specific to the Pochoir compiler, the Intel C++ compiler, and the multicore machines we used for benchmarking.

The static information about a stencil computation, such as the computing kernel, the boundary conditions, and the stencil shape, is stored in a ***Pochoir object***, which is declared as follows:

- **`Pochoir_`*dim*`D`** *name* `(` *shape* `);`

This statement declares *name* as a Pochoir object with *dim* spatial dimensions and computing shape *shape*, where *dim* is a small positive integer and *shape* is an array of arrays which describes the shape of the stencil as elaborated below.

We now itemize the remaining Pochoir constructs and explain the semantics of each.

- **`Pochoir_Shape_`*dim*`D`** *name* `[] = {`*cells*`}`

This statement declares *name* as a ***Pochoir shape*** that can hold shape information for *dim* spatial dimensions. The Pochoir shape is equivalent to an array of arrays, each of which contains $dim+1$ integer numbers. These numbers represent the offset of each memory footprint in the stencil kernel relative to the space-time grid point $\langle t,x,y,\cdots\rangle$. For example, suppose that the computing kernel employs the Jacobi-style update equation for 2D heat diffusion from Section 1. The shape of this stencil is $\{\{0,0,0\}, \{-1,1,0\}, \{-1,0,0\}, \{-1,-1,0\}, \{-1,0,1\}, \{-1,0,-1\}\}$.

The first cell in the shape is the ***home*** cell, whose spatial coordinates must all be 0. During the computation, this cell corresponds to the grid point being updated. Time offsets of the remaining cells must be smaller than the time coordinate of the home cell, and the corresponding grid points during the computation are read-only.

The ***depth*** of a shape is the time coordinate of the home cell minus the minimum time coordinate of any cell in the shape. The depth corresponds to the number of time steps on which a grid point depends. For our example stencil, the depth of the shape is 1, since a point at time t depends on points at time $t-1$.. If a stencil shape has depth k, the programmer must initialize all Pochoir arrays for time steps $0,1,\ldots,k-1$ before running the computation.

- **`Pochoir_Array_`*dim*`D`**`(`*type*, *depth*`)` *name* `(`$size_{dim-1},\ldots, size_1, size_0$`)`

This statement declares *name* as a ***Pochoir array*** of type *type* with *dim* spatial dimensions and a temporal dimension. The ith spatial dimension has size $size_i$, where $i \in \{0,1,\ldots,dim\}$. The temporal dimension has size $k+1$, where k is the depth of the Pochoir shape, and are reused modulo $k+1$ as the computation proceeds. The user may not obtain an alias to the Pochoir array or its elements.

- **`Pochoir_Boundary_`*dim*`D`**$(name, array, idx_t, idx_{dim-1},\ldots, idx_1, idx_0)$
  $\langle definition \rangle$
  `Pochoir_Boundary_End`

This construct defines a ***boundary function*** called *name* that will be invokeda to supply a value when the stencil computation accesses a point outside the domain of the Pochoir array *array*. The Pochoir array *array* has *dim* spatial dimensions, and $\langle idx_{dim-1},\ldots,idx_1,idx_0\rangle$ are the spatial coordinates of the given point outside the domain of *array*. The coordinate in the time dimension is given by idx_t. The function body $\langle definition \rangle$) is C++ code that defines the values of *array* on its boundary. A current restriction is that this construct must be declared outside of any function, that is, the boundary function is declared global.

- **`Pochoir_Kernel_`*dim*`D`**$(name, array, idx_t, idx_{dim-1},\ldots, idx_1, idx_0)$
  $\langle definition \rangle$
  `Pochoir_Kernel_End`

This construct defines a ***kernel function*** named *name* for updating a stencil on a spatial grid with *dim* spatial dimensions. The spatial coordinates of the point to update are $\langle idx_{dim-1},\ldots,idx_1,idx_0\rangle$, and idx_t is the coordinate in time dimension. The function body $\langle definition \rangle$ may contain arbitrary C++ code. Unlike boundary functions, this construct can be defined in any context.

- *name*.**`Register_Array`**(*array*)

A call to this member function of a Pochoir object *name* informs *name* that the Pochoir array *array* will participate in its stencil computation.

- *name*.**`Register_Boundary`**(*bdry*)

A call to this member function of a Pochoir array *name* associates the declared boundary function *bdry* with *name*. The boundary function is invoked to supply a value whenever an off-domain memory access occurs. Each Pochoir array is associated with exactly one boundary function at any given time, but the programmer can change boundary functions by registering a new one.

- *name*.**`Run`**$(T, kern)$

This function call runs the stencil computation on the Pochoir object *name* for T time steps using computing kernel function *kern*.

After running the computation for T steps, the results of the computation can be accessed by indexing its Pochoir arrays at time $T+k-1$, where k is the depth of the stencil shape. The programmer may resume the running of the stencil after examining the result of the computation by calling *name*.Run(T', *kern*), where T' is the number of additional steps to execute. The result of the computation is then in the computation's Pochoir arrays indexed by time $T+T'+k-1$.

Rationale

The Pochoir language is a product of many design decisions, some of which were influenced by the current capabilities of the Intel 12.0.0 C++ compiler. We discuss some of the more important design decisions in [34].

3. Pochoir's cache-oblivious parallel algorithm

This section describes the parallel algorithm at the core of Pochoir's efficiency. TRAP is a cache-oblivious algorithm based on "trapezoidal decompositions" [13, 14], but which employs a novel "hyperspace-cut" strategy to improve parallelism without sacrificing cache-efficiency. On a d-dimensional spatial grid with all "normalized" spatial dimensions equal to w and the time dimension a power-of-2 multiple of w, TRAP achieves $\Theta(w^{d-\lg(d+2)+1}/d^2)$ parallelism, whereas Frigo and Strumpen's original parallel trapezoidal decomposition algorithm [14] achieves $\Theta(w^{d-\lg(2^d+1)+1}/2^d) = O(w)$ parallelism. Both algorithms exhibit the same asymptotic cache complexity of $\Theta(hw^d/\mathcal{M}^{1/d}\mathcal{B})$ proved by Frigo and Strumpen, where h is the height of the time dimension, $\mathcal{M}$ is the cache size, and $\mathcal{B}$ is the cache-block size.

TRAP uses a cache-oblivious [12] divide-and-conquer strategy based on a recursive trapezoidal decomposition of the space-time grid, which was introduced by Frigo and Strumpen [13, 14]. Whereas Frigo and Strumpen's parallel algorithm [14] cuts the spatial dimensions of a hypertrapezoid, or "zoid," one at a time with "parallel space cuts," TRAP performs a ***hyperspace cut*** where it applies parallel space cuts simultaneously to as many dimensions as possible, yielding asymptotically more parallelism when the number of spatial dimensions is 2 or greater. As we will argue later in this section, TRAP achieves this improvement in parallelism while attaining the same cache complexity as Frigo and Strumpen's original parallel algorithm.

TRAP operates as follows. Line 5 of Figure 2 determines whether the x-dimension of the zoid can be cut with a parallel space cut, and if so, line 6 trisects the zoid, as we shall describe later in this section and in Figure 7, but it does not immediately spawn recursive tasks to process the subzoids, as Frigo and Strumpen's algorithm would. Instead, the code attempts to make a "hyperspace cut" by proceeding to the y-dimension, and if there were more dimensions, to those, cutting as many dimensions as possible before spawning recursive tasks to handle the subzoids. The counter k keeps track of how many spatial dimensions are cut. If $k>0$ spatial dimensions are trisected, as tested for in line 9, then line 10 assigns each subzoid to one of $k+1$ dependency levels such that the subzoids assigned to the same level are independent and can be processed in parallel, as implied by Lemma 1 and described in its proof in [34]. Lines 11–13 recursively walk all subzoids level by level in parallel. Lines 15–17 perform a time cut if no space cut can be performed. Lines 18–23 perform the base-case computation if the zoid is sufficiently small that no space or time cut is productive.

We first introduce some notations and definitions, many of which have been borrowed or adapted from [13, 14]. A $(d+1)$-dimensional ***space-time hypertrapezoid***, or ***$(d+1)$-zoid***, $\mathcal{Z} = (ta, tb;\ xa_0, xb_0, dxa_0, dxb_0;\ xa_1, xb_1, dxa_1, dxb_1;\ \ldots;\ xa_{d-1}, xb_{d-1}, dxa_{d-1}, dxb_{d-1})$, where all variables are integers, is the set of integer grid points $\langle t, x_0, x_1, \ldots, x_{d-1}\rangle$ such that $ta \le t < tb$ and $xa_i + dxa_i(t-ta) \le x_i < xb_i + dxb_i(t-ta)$ for all $i \in \{0,1,\ldots,d-1\}$. The ***height*** of $\mathcal{Z}$ is $\Delta t = tb - ta$. Define the ***projection trapezoid*** $\mathcal{Z}_i$ of $\mathcal{Z}$ along spatial dimension i to be the 2D trapezoid that results from projecting the zoid $\mathcal{Z}$ onto the dimensions x_i and t. The projection trapezoid $\mathcal{Z}_i$ has two ***bases*** (sides parallel to the x_i axis) of lengths $\Delta x_i = xb_i - xa_i$ and $\nabla x_i = (xb_i + dxb_i\Delta t) - (xa_i + dxa_i\Delta t)$. We define the ***width***[1] w_i of $\mathcal{Z}_i$ to be the length of the longer of the two bases (parallel sides) of $\mathcal{Z}_i$, that is $w_i = \max\{\Delta x_i, \nabla x_i\}$. The value w_i is also called the ***width*** of $\mathcal{Z}$ along spatial dimension i. We say that $\mathcal{Z}_i$ is ***upright*** if $w_i = \Delta x_i$ — the longer base corresponds to time ta — and ***inverted*** otherwise. A zoid $\mathcal{Z}$ is ***well-defined*** if its height is positive, its widths along all spatial dimensions are positive, and the lengths of its bases along all spatial dimensions are nonnegative. A projection trapezoid $\mathcal{Z}_i$ is ***minimal*** if $\mathcal{Z}_i$ is upright and $\nabla x_i = 0$, or $\mathcal{Z}_i$ is inverted and $\Delta x_i = 0$. A zoid $\mathcal{Z}$ is ***minimal*** if all its $\mathcal{Z}_i$'s are minimal.

Given the shape S of a d-dimensional stencil (as described in Section 2), define t_{home} be the time index of the home cell. We define the ***slope***[2] of a cell $c = (t, x_0, x_1, \ldots, x_{d-1}) \in S$ along dimension $i \in \{0,1,\ldots,d-1\}$ as $\sigma_i(c) = |x_i/(t_{\text{home}} - t)|$, and we define the ***slope*** of the stencil along spatial dimension i as $\sigma_i = \max_{c\in S}\lceil\sigma_i(c)\rceil$. (Pochoir assumes for simplicity that the stencil is symmetric in each dimension.) We define the ***normalized width*** of a zoid $\mathcal{Z}$ along dimension i by $\widehat{w_i} = w_i/2\sigma_i$.

Parallel space cuts

Our trapezoidal decomposition differs from that of Strumpen and Frigo in the way we do parallel space cuts. A ***parallel space cut*** can be applied along a given spatial dimension i of a well-defined zoid $\mathcal{Z}$ provided that the projection trapezoid $\mathcal{Z}_i$ can be trisected into 3 well-defined subtrapezoids, as shown in Figures 7(a) and 7(b). The triangle-shaped gray subtrapezoid that lies in the middle is a minimal trapezoid. The larger base of $\mathcal{Z}_i$ is split in half with each half forming the larger base of a black subtrapezoid. These three subtrapezoids of $\mathcal{Z}_i$ correspond to three subzoids of $\mathcal{Z}$. Since the two black subzoids have no interdependencies, they can be processed in parallel. As shown in Figure 7(a), for an upright projection trapezoid, the subzoids corresponding to the black trapezoids are processed first, after which the subzoid corresponding to the gray subtrapezoid can be processed. For an inverted projection trapezoid, as shown in Figure 7(b), the opposite is done. In either case, the 3 subzoids can be processed in parallel in the time to process 2 of them, what we shall call 2 ***parallel steps***. The following lemma which we proved in [34], describes the general case.

LEMMA 1. *All 3^k subzoids created by a hyperspace cut on $k \ge 1$ of the $d \ge k$ spatial dimensions of a $(d+1)$-zoid $\mathcal{Z}$ can be processed in $k+1$ parallel steps.*

Pochoir's cache-oblivious parallel algorithm

Given a well-defined zoid $\mathcal{Z}$, the algorithm TRAP from Figure 2 works by recursively decomposing $\mathcal{Z}$ into smaller well-defined zoids as follows.

Hyperspace cut. Lines 4–8 in Figure 2 apply a hyperspace cut involving all dimensions on which a parallel space cut can be applied, as shown in Figures 7(a) and 7(b). If the number k of dimensions of $\mathcal{Z}$ on which a space cut can be applied is at least 1, as tested for in line 9 of Figure 2, then as explained in the proof of Lemma 1 (given in [34]), dependency levels are computed for all resulting subzoids in line 10. Lines 11–13 recursively process all subjoids in order according to dependency level.

Time cut. If a hyperspace cut is not applicable and $\mathcal{Z}$ has height greater than 1, as tested for in line 14, then lines 15–17 cut $\mathcal{Z}$ in

[1] Frigo and Strumpen [13, 14] define width as the average of the two bases.

[2] Actually, the reciprocal of slope, but we follow Frigo and Strumpen's terminology.

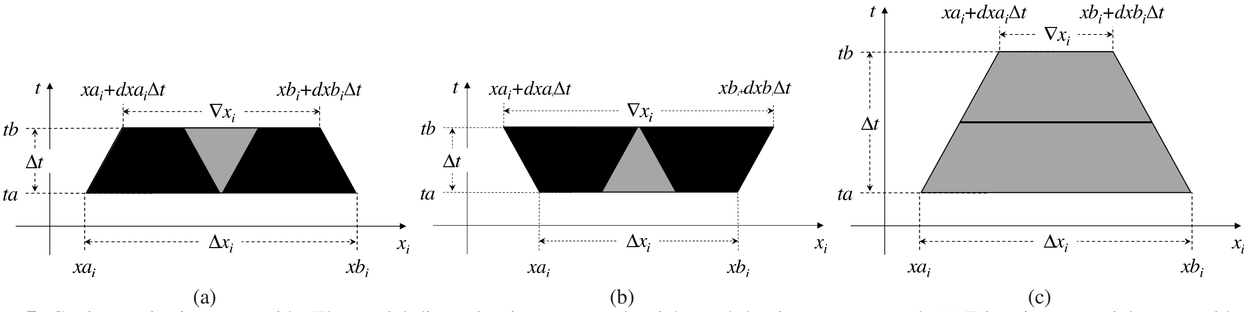


Figure 7: Cutting projection trapezoids. The spatial dimension increases to the right, and the time runs upward. (a) Trisecting an upright trapezoid using a parallel space cut produces two black trapezoids that can be processed in parallel and a gray trapezoid that must be processed after the black ones. (b) Trisecting an inverted trapezoid using a parallel space cut produces two black trapezoids that can be processed in parallel and a gray trapezoid that must be processed before the black ones. (c) A time cut produces a lower and an upper trapezoid where the lower trapezoid must be processed before the upper.

the middle of its time dimension and recursively process the lower subzoid followed by the upper subzoid, as shown in Figure 7(c).

Base case. If neither a hyperspace cut nor a time cut can be applied, lines 18–23 processes $\mathcal{Z}$ directly by invoking the stencil-specific kernel function. In practice, the base case is ***coarsened*** (see Section 4) by choosing a suitable threshold larger than 1 for Δt in line 14, which cuts down on overhead due to the recursion.

Analysis

We can analyze the parallelism using a work/span analysis [4, Ch. 27]. The ***work*** T_1 of a computation is its serial running time, and the ***span*** T_∞ is the longest path of dependencies, or equivalently, the running time on an infinite number of processors assuming no overheads for scheduling. The ***parallelism*** of a computation is the ratio T_1/T_∞ of work to span.

We proved the following bound on TRAP's parallelism in [34].

THEOREM 2. *Consider a $(d+1)$-dimensional grid $\mathcal{Z}$ with $\widehat{w}_i = w$ for $i \in \{0,1,\ldots,d-1\}$ and height $h = 2^r w$. Then the parallelism of* TRAP *when processing $\mathcal{Z}$ using a stencil with constant slopes is $\Theta(w^{d-\lg(d+2)+1}/d^2)$.*

We can compare TRAP with a version of Frigo and Strumpen's parallel stencil algorithm [14] we call STRAP, which performs the space cuts serially as in Figures 7(a) and 7(b). Each space cut results in one synchronization point, and hence a sequence of k space cuts applied by STRAP introduces 2^k parallel steps compared to the $k+1$ parallel steps generated by TRAP (see Lemma 1). Thus, each space cut virtually doubles STRAP's span. The next theorem analyzes STRAP, mimicking Theorem 2.

THEOREM 3. *Consider a $(d+1)$-dimensional grid $\mathcal{Z}$ with $\widehat{w}_i = w$ for $i \in \{0,1,\ldots,d-1\}$ and height $h = 2^r w$. Then the parallelism of* STRAP *when processing $\mathcal{Z}$ using a stencil with constant slopes is $\Theta(w^{d-\lg(2^d+1)+1}/2^d)$.*

Empirical results

Figure 8 shows the results of using the Cilkview scalability analyzer [17] to compare the parallelism of TRAP and STRAP on two typical benchmarks. We measured the two algorithms with uncoarsened base cases. As can be seen from the figure, TRAP's asymptotic advantage in parallelism is borne out in practice for these benchmarks.

We used the Linux perf tool [25] to verify that TRAP does not suffer any loss in cache efficiency compared to the STRAP algorithm. Figure 9 also plots the cache-miss ratio of the straightforward parallel loop algorithm, showing that it exhibits poorer cache performance than the two cache-oblivious algorithms.

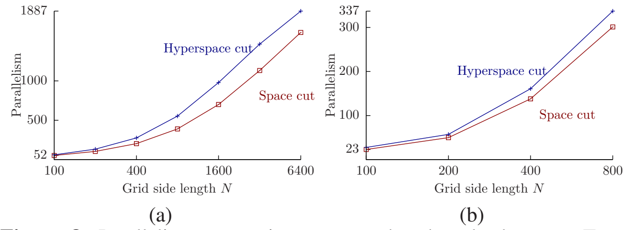


Figure 8: Parallelism comparison on two benchmarks between TRAP, which employs hyperspace cuts, and STRAP, which uses serial space cuts. Measurements are of code without base-case coarsening. (a) 2D nonperiodic heat equation. Space-time size is $1000N^2$. (b) 3D nonperiodic wave equation. Space-time size is $1000N^3$.

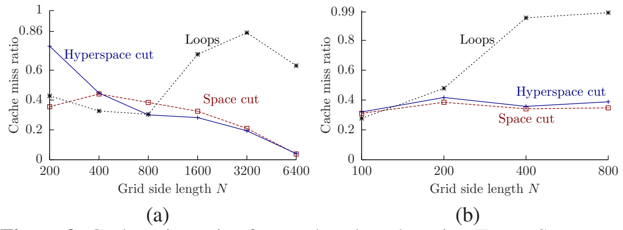


Figure 9: Cache-miss ratios for two benchmarks using TRAP, STRAP, and a parallel-loop algorithm. The cache-miss ratio is the ratio of the cache misses to the number of memory references. Measurements are of code without base-case coarsening. (a) 2D nonperiodic heat equation. Space-time is $1000N^2$. (b) 3D nonperiodic wave equation. Space-time is $1000N^3$.

4. Compiler optimizations

The Pochoir compiler transforms code written in the Pochoir specification language into optimized C++ code that employs the Intel Cilk multithreading extensions [20]. The Pochoir compiler is written in Haskell [32], and it performs numerous optimizations, the most important of which are code cloning, loop-index calculations, unifying periodic and nonperiodic boundary conditions, and coarsening the base case of recursion. This section describes how the Pochoir compiler implements these optimizations.

Before a programmer compiles a stencil code with the Pochoir compiler, he or she is expected to perform Phase 1 of Pochoir's two-phase methodology which requires that it be compiled using the Pochoir template library and debugged. This C++ template library is employed by both Phases 1 and 2 and includes both loop-based and trapezoidal algorithms. Differences between stencils, such as dimensionality or data structure, are incorporated into these generic algorithms at compile-time via C++ template metaprogramming.

Handling boundary conditions by code cloning

The handling of boundary conditions can easily dominate the runtime of a stencil computation. For example, we coded the 2D heat

```
Pochoir_Boundary_2D(dirichlet, arr, t, x, y)
     return 100 + 0.2*t;
Pochoir_Boundary_End
```
(a)

```
Pochoir_Boundary_2D(neumann, arr, t, x, y)
    int newx = x;
    if (x < 0) newx = 0;
    if (x >= arr.size(1)) newx = arr.size(1);
    int newy = y;
    if (y < 0) newy = 0;
    if (y >= arr.size(0)) newy = arr.size(0);
    return arr.get(t, newx, newy);
Pochoir_Boundary_End
```
(b)

Figure 10: Pochoir code for specifying nonperiodic boundary conditions. (a) A Dirichlet condition with constrained boundary value (set equal to a function of t). (b) A Neumann condition with constrained derivative at the boundary (set equal to 0).

equation on a periodic torus using Pochoir, and we compared it to a comparable code that simply employs a modulo operation on every array index. For a 5000^2 spatial grid over 5000 time steps, the runtime of the modular-indexing implementation degraded by a factor of 2.3.

For nonperiodic stencil computations, where a value must be provided on the boundary, performance can degrade even more if a test is made at every point to determine whether the index falls off the grid. Stencil implementers often handle constant nonperiodic boundary conditions with the simple trick of introducing ***ghost cells*** [5] that form a ***halo*** around the periphery of the grid. In practice, however, nonperiodic boundary conditions can be more complicated than simple constants, and we wanted to allow Pochoir users flexibility in the kinds of boundary conditions they could specify. For example, Dirichlet boundary conditions may specify boundary values that change with time, and Neumann boundary conditions may specify the value the derivative should take on the boundary [11]. Figure 10(a) shows a Pochoir specification of a Dirichlet boundary condition, and Figure 10(b) shows the Pochoir specification of a Neumann boundary condition.

To handle boundaries efficiently, the Pochoir compiler generates two code clones of the kernel function: a slower ***boundary*** clone and a faster ***interior*** clone. The boundary clone is used for ***boundary*** zoids: those that contain at least one point whose computation requires an off-grid access. The interior clone is used for ***interior*** zoids: those all of whose points can be updated without indexing off the edge of the grid. While the boundary clone must check whether each point is on the boundary or not, the interior clone completely avoid this expensive computation. Whether a zoid is interior or boundary is determined at runtime.

The trapezoidal-decomposition algorithm exploits the fact that all subzoids of an interior zoid remain interior. If all the dimensions of the grid are approximately the same size, the boundary of the grid is much smaller than its (hyper)volume. Consequently, the faster interior clones dominate the running time, and the slower boundary clones contribute little.

Loop indexing

Because the interior zoids asymptotically dominate the computing time, most of the optimizations performed by Pochoir compiler focus on the interior clone. Two important optimizations relate to loop indexing. The particular optimization is chosen automatically by the Pochoir compiler, or it can be mandated by user as a command-line option. Consistent with their command-line names, the optimizations are called `-split-macro-shadow` and `-split-pointer`.

The `-split-macro-shadow` option causes the Pochoir compiler to employ macro tricks on the interior clone to eliminate the boundary-checking overhead. The `-split-pointer` command-line option causes the Pochoir compiler to transform the indexing

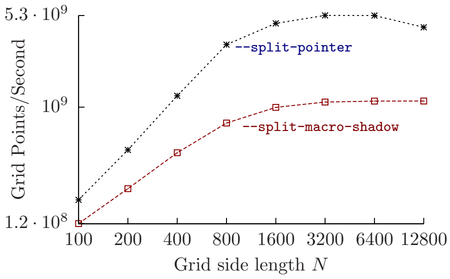


Figure 11: The performance of different loop-index optimizations on a 2D heat equation on torus. The grid is N^2 with 1000 time steps.

of Pochoir arrays in the interior clone into C-style pointer manipulation. The Pochoir compiler tries to use the `-split-pointer` optimization if possible. It can do so if it can parse and "understand" the C++ syntax of the user's specification. Because our prototype Haskell compiler does not contain a complete C++ front end, however, it sometimes may not understand unusually complex C++ code written by the user, in which case, it employs the `-split-macro-shadow` optimization, relying on Phase 1 to ensure that the code is Pochoir-compliant.

Figure 11 compares the performances of the two optimizing options for a 2D heat equation on a torus. Other benchmarks show similar relative performances.

Unifying periodic and nonperiodic boundary conditions

Typical stencil codes discriminate between periodic and nonperiodic stencils, implementing them in different ways. To make the specification of boundary functions as flexible as possible, we investigated how periodic and nonperiodic stencils could be implemented using the same algorithmic framework, leaving the choice of boundary function up to the user. Our unified algorithm allows the user to program boundary functions with arbitrary periodic/nonperiodic behavior, providing support, for example, for a 2D cylindrical domain, where one dimension is periodic and the other is nonperiodic. The key idea is to treat the entire computation as if it were periodic in all dimensions and handle nonperiodicity and other boundary conditions in the base case of the boundary clone where the kernel function is invoked.

Coarsening of base cases

Previous work [6, 22, 23, 30] has found that although trapezoidal decomposition dramatically reduces cache-miss rates, overall performance can suffer from function-call overhead unless the base case of the recursion is coarsened.

Since choosing the optimal size of the base case can be difficult, we integrated the ISAT autotuner [19] into Pochoir. Despite the advantage of finding the optimal coarsening factor on any specific platform, this autotuning process can take hours to find the optimal value, which may be unacceptable for some users.

In practice, Pochoir employs some heuristics to choose a reasonable coarsening. One principle is that to maximize data reuse, we want to make the spatial dimensions all about the same size. Another principle is that to exploit hardware prefetching, we want to avoid cutting the unit-stride spatial dimension and avoid odd-shaped base cases. We have found that for 3D problems, the effect of hardware prefetching can often be more important than cache efficiency for reasonably sized base cases. For 2D problems, Pochoir stops the recursion at 100×100 space chunks with 5 time steps. For 3D problems, the recursion stops at $1000 \times 3 \times 3$ with 3 time steps.

5. Concluding remarks

It is remarkable how complex a simple computation can be when performance is at stake. Parallelism and caching make stencil computations interesting. We see many ways to improve Pochoir by taking advantage of various other features of modern machines — such as prefetchers, graphical processing units, and clustering.

In addition, we see ample opportunity to enhance the linguistic features of the Pochoir specification language to provide more generality and flexibility to the user. For example, we are considering how to allow the user to specify irregularly shaped domains. As long as the boundary of a region, however irregular, is small compared to the region's interior, special-case code to handle the boundary should not adversely impact the overall performance. Even more challenging is coping with boundaries that change with time. We believe that such capabilities will dramatically speed up the PSA, RNA, and LCS benchmarks which operate on diamond-shaped space-time domains.

Pochoir's two-phase compilation strategy introduces a new method for building domain-specific languages embedded in C++. Historically, the complexity of parsing and type-checking C++ has impeded such separately compiled domain-specific languages. C++'s template programming does provide a good measure of expressiveness for describing special-purpose computations, but it provides no ability to perform the domain-specific optimizations such as those that Pochoir employs. Pochoir's compilation strategy offers a new way to build optimizing compilers for domain-specific languages embedded in C++ where the compiler can parse and "understand" only as much of the programmer's C++ code as it is able, confident that code it does not understand is nevertheless correct.

The Pochoir compiler can be downloaded from `http://supertech.csail.mit.edu/pochoir`.

6. Acknowledgments

The authors would like to thank Matteo Frigo, Volker Strumpen, Kaushik Datta, Sam Williams, Geoff Lowney, Robert Geva, the Intel Cilk team, Will Leiserson, Will Hasenplaugh, and the members of the MIT Supertech Research Group.

References

[1] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Appl. Math.*, 104:45–62, 2000.

[2] R. Bleck, C. Rooth, D. Hu, and L. T. Smith. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the North Atlantic. *J. Phys. Oceanogr.*, 22(12):1486–1505, 1992.

[3] R. Chowdhury, H.-S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *TCBB*, 7(3):495–510, 2010.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[5] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, UC Berkeley, Dec 2009.

[6] 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*, pages 4:1–4:12, Austin, TX, Nov. 15–18 2008.

[7] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[8] H. Dursun, K.-i. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par*, pages 642–653, Delft, The Netherlands, Aug. 25–28 2009.

[9] H. Dursun, K.-i. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, Las Vegas, NV, July13–16 2009.

[10] J. F. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.

[11] H. Feshbach and P. Morse. *Methods of Theoretical Physics*. Feshbach Publishing, 1981.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, New York, NY, Oct. 17–19 1999.

[13] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, pages 361–366, Cambridge, MA, June 20–22 2005.

[14] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theor. Comput. Syst.*, 45(2):203–233, 2009.

[15] M. Gardner. Mathematical Games. *Scientific American*, 223(4):120–123, 1970.

[16] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[17] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, Santorini, Greece, June 13–15 2010.

[18] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), December 1996.

[19] Intel software autotuning tool. `http://software.intel.com/en-us/articles/intel-software-autotuning-tool/`, 2010.

[20] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[21] C. John. *Options, Futures, and Other Derivatives*. Prentice Hall, 2006.

[22] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC*, pages 51–60, San Jose, CA, 2006.

[23] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP*, pages 36–43, Chicago, IL, June 12 2005.

[24] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, San Diego, CA, June 10–13 2007.

[25] `https://perf.wiki.kernel.org/index.php/Main_Page`.

[26] R. Mei, W. Shyy, D. Yu, and L. Luo. Lattice Boltzmann method for 3-D flows with curved boundary. *J. of Comput. Phys*, 161(2):680–699, 2000.

[27] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, December 2005.

[28] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPPGPU*, pages 79–84, Washington, DC, Mar. 8 2009.

[29] A. Nakano, R. Kalia, and P. Vashishta. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Computer Physics Communications*, 83(2-3):197–214, 1994.

[30] A. Nitsure. Implementation and optimization of a cache oblivious lattice Boltzmann algorithm. Master's thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2006.

[31] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11, Rome, Italy, May 23–29 2009.

[32] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 1998.

[33] A. Taflove and S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Norwood, MA, 2000.

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

[35] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *IPDPS*, pages 1–14, Miami, FL, Apr. 2008.