Coding Stencil Computations Using the Pochoir Stencil-Specification Language

Yuan Tang Rezaul Chowdhury Chi-Keung Luk Charles E. Leiserson

MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139

Abstract

Pochoir is a compiler for a domain-specific language embedded in C++ which produces excellent code from a simple specification of a desired stencil computation. Pochoir allows a wide variety of boundary conditions to be specified, and it automatically parallelizes and optimizes cache performance. Benchmarks of Pochoir-generated code demonstrate a performance advantage of 2–10 times over standard parallel loop implementations. This paper describes the Pochoir specification language and shows how a wide range of stencil computations can be easily specified.

1. INTRODUCTION

Stencil computations [2, 4–6, 8, 9, 14–16, 20–22, 24, 27] are frequently used in scientific computing, image processing, and geometric modeling. 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*** computes the stencil repeatedly for each grid point over many time steps.

It is hard to write efficient stencil computations. Programmers typically implement them as loop nests. This popular method results in poor performance on modern multicore architectures, however, because it is not cache-friendly and fails to use multiple processing cores. Frigo and Strumpen [8] introduced "trapezoidal decompositions" as a way of coding efficient cache-oblivious [7] algorithms for stencil computations. In later work [9], they showed how 1D stencils could be parallelized by cutting the spatial dimension into certain number of black and gray subtrapezoids, where subtrapezoids of the same color can be executed in parallel. They also indicated how their methodology might be extended to arbitrary d-dimensional stencils. Unfortunately, although their method can substantially reduce cache-miss ratios, it is complicated, and as with other cache-oblivious algorithms, good performance can be hard to achieve due to unpredictable branches [4, 14, 15, 21].

In [25], we introduced Pochoir (pronounced "PO-shwar"), a domain-specific compiler automatically parallelizing and optimizing stencils. The stencil is specified using the Pochoir specification language, which is embedded in C++. After specifying a stencil computation in the Pochoir specification language, the user first compiles the program any native C++ compiler with the Pochoir template library. The point of this first phase of the Pochoir methodology is to check the functional correctness of the stencil specification. The code produced using the Pochoir template library is not intended to be fast. Rather, it allows the programmer to debug the program using a comfortable native C++ tool chain without the complications of the Pochoir compiler. In addition, the Pochoir template library tests for inconsistencies in the specification. After validating the correctness of the specification in the first phase, the programmer recompiles the program with the Pochoir compiler. This second phase of the Pochoir methodology produces a highly efficient Cilk Plus [13] parallel code which typically performs at least as well as an expert hand-optimized stencil code. The Pochoir compiler automatically tunes the code without requiring the programmer to make any manual annotations or to insert any compiler-specific pragmas.

The Pochoir methodology greatly simplifies the implementation of the Pochoir compiler. The compiler need not type-check or even parse much of the C++ code that makes up a user's stencil specification. Instead, it relies on the first-phase native C++ compilation with the Pochoir template library to type-check and catch any inconsistencies in the specification. The two phases are linked semantically by 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.*

This paper illustrates how the Pochoir language can be used to specify a variety of stencil computations. A full description of the Pochoir compiler and the algorithm behind it can be found in [25]. Section 2 provides a specification of the Pochoir embedded language. Section 3 describes a stencil for a 3-dimensional wave equation for seismic imaging. Section 4 gives the example of pairwise sequence alignment for computational biology. Section 5 describes the lattice Boltzmann method for theoretical physics. Section 6 compares the performance of the Pochoir-generated code with serial loops and parallel loops. Section 7 offers some concluding remarks.

This work was supported in part by a grant from Intel Corporation and in part by the National Science Foundation under Grants CCF-0937860 and CNS-1017058.

Yuan Tang is an Assistant Professor of Computer Science at Fudan University in China and a Visiting Scientist at MIT CSAIL. Chi-Keung Luk is a Senior Staff Engineer at Intel Corporation and a Research Affiliate at MIT CSAIL. Rezaul Chowdhury is a Research Scientist at Boston University and a Research Affiliate at MIT CSAIL. Charles E. Leiserson is a Professor of Computer Science and Engineering at MIT CSAIL.

USENIX HotPar'11 May 26–27, 2011, Berkeley, California, USA.

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 Pochoir 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 following update equation:

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

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. The remaining cells must have time offsets that are 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 size of the ith spatial dimension, where $i \in \{0,1,\ldots,dim\}$, is given by $size_i$. 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 to compute the stencil. 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 $T+T'+k-1$.

3. 3D WAVE EQUATION

This section illustrates a Pochoir specification of a stencil computation through the example of solving a 3D wave equation. We use a finite-difference (3DFD) discretization of the wave equation, which gives rise to a 3-dimensional, 4th-order,

25-point stencil [19]. This stencil has practical applications in seismic imaging [1,17], and it illustrates how periodic and nonperiodic boundary conditions can be specified in Pochoir.

```
1   Pochoir_Boundary_3D(fd_bv_3D, arr, t, z, y, x)
2       return 0.0;
3   Pochoir_Boundary_End
4   Pochoir_Shape_3D fd_shape_3D[] = {{1,0,0,0},
        {0,0,0,0}, {0,0,0,1}, {0,0,0,-1},
        {0,0,1,0}, {0,0,-1,0}, {0,1,0,0},
        {0,-1,0,0}, {0,0,0,2}, {0,0,0,-2},
        {0,0,2,0}, {0,0,-2,0}, {0,2,0,0},
        {0,-2,0,0}, {0,0,0,3}, {0,0,0,-3},
        {0,0,3,0}, {0,0,-3,0}, {0,3,0,0},
        {0,-3,0,0}, {0,0,0,4}, {0,0,0,-4},
        {0,0,4,0}, {0,0,-4,0}, {0,4,0,0},
        {0,-4,0,0}};
5   Pochoir_3D fd_3D(fd_shape_3D);
6   Pochoir_Array_3D(float) pa(Nz, Ny, Nx);
7   pa.Register_Boundary(fd_bv_3D);
8   fd_3D.Register_Array(pa);
9   Pochoir_Kernel_3D(fd_3D_fn, t, z, y, x)
10    float div = c0*pa(t,z,y,x) + c1*(pa(t,z,y,x
          +1) + pa(t,z,y,x-1) + pa(t,z,y+1,x) + pa
          (t,z,y-1,x) + (pa(t,z+1,y,x) + pa(t,z-1,
          y,x)) + c2*(pa(t,z,y,x+2) + pa(t,z,y,x
          -2) + pa(t,z,y+2,x) + pa(t,z,y-2,x) + pa
          (t,z+2,y,x) + pa(t,z-2,y,x)) + c3*(pa(t,
          z,y,x+3) + pa(t,z,y,x-3) + pa(t,z,y+3,x)
           + pa(t,z,y-3,x) + pa(t,z+3,y,x) + pa(t,
          z-3,y,x)) + c4*(pa(t,z,y,x+4) + pa(t,z,y
          ,x-4) + pa(t,z,y+4,x) + pa(t,z,y-4,x) +
          pa(t,z+4,y,x) + pa(t,z-4,y,x));
11    pa(t+1,z,y,x) = 2*pa(t,z,y,x) - pa(t+1,z,y,x)
          + vsq[z*Nxy + y*Nx + x]*div;
12  Pochoir_Kernel_End
13  /* Initialize the Pochoir array pa */
14  for (int z = 0; z < Nz; ++z)
15    for (int y = 0; y < Ny; ++y)
16      for(int x = 0; x < Nx; ++x) {
17    float r = abs((float)(x - Nx/2 + y - Ny/2 + z
          - Nz/2) / 30);
18    r = max(1 - r, 0.0f) + 1;
19    pa(0, z, y, x) = r;
20  }
21  fd_3D.Run(T, fd_3D_fn);
22  /* Output the final results */
23  for (int z = 0; z < Nz; ++z)
24    for (int y = 0; y < Ny; ++y)
25      for(int x = 0; x < Nx; ++x) {
26      cout << pa(T, z, y, x);
27  }
```

Figure 1: A Pochoir specification for solving a wave equation on a 3D spatial grid with a nonperiodic boundary condition.

Figure 1 shows the Pochoir source code for the 3D wave equation with a nonperiodic boundary condition. Line 4 is the computation shape of the 3D wave equation. Line 5 declares a Pochoir object named `fd_3D` having that shape. The Pochoir object will contain all states necessary to perform the stencil computation. Each triple in the array `fd_shape_3D` corresponds to a relative offset from the space-time grid point (t,z,y,x) that the stencil kernel (declared in lines 9–12) 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.

Line 6 declares `pa` as a `Nx`×`Ny`×`Nz` Pochoir array of single-precision floating-point numbers representing the spatial grid. Lines 1–3 define a function `fd_bv_3D` which is called when the kernel function accesses grid points outside the computing domain, that is, if it tries to access `pa(t,z,y,x)`, where $(\mathtt{z},\mathtt{y},\mathtt{x}) \notin [0,\mathtt{Nz}) \times [0,\mathtt{Ny}) \times [0,\mathtt{Nx})$. For this nonperiodic boundary condition, `fd_bv_3D` supplies a value 0 when an off-domain access occurs. Line 7 associates the boundary function with the Pochoir array `pa`. Each Pochoir array has exactly one boundary

```
1  #define mod(r, m) ((r)%(m) + ((r) < 0) ? (m) : 0)
2  Pochoir_Boundary_3D(fd_bv_3D, a, t, z, y, x)
3    return a.get(t, mod(z, a.size(2)), mod(y, a.
         size(1)), mod(x, a.size(0)));
4  Pochoir_Boundary_End
```

Figure 2: Specifying the boundary function for a 3D torus.

function at any given time. Line 8 registers the Pochoir array `pa` with the `fd_3D` Pochoir object. A Pochoir array can be registered with more than one Pochoir object, and a Pochoir object can have multiple Pochoir array registered.

Lines 9–12 define `fd_3D_fn` as a kernel function, which specifies how the stencil is computed for every grid point. The kernel can be an arbitrary piece of C++ code, but accesses to the registered Pochoir arrays must respect the declared shape(s).

Finally, we are ready to initialize and run the computation. Lines 14–20 initialize the Pochoir array `pa` with values for time step 0. If more than 1 previous time step is needed for updating the current time step, the user is responsible for initializing the corresponding number of time steps before running the stencil. Finally, line 21 executes the stencil object `fd_3D` for T time steps, specifying the kernel function `fd_3D_fn`. Lines 23–27 is the example code of how to extract the value out of the Pochoir array after the computation is done.

Figure 2 shows how to specify the boundary function `fd_bf_3D` for a periodic boundary, causing the computation to operate on a 3D torus having "wrap-around," as opposed to a terminating boundary. Line 1 defines a modulo operation for indices, and line 3 obtains and returns the required entry based on the new indices. Due to the current limitation in Intel C++ compiler (as discussed in rationale of Section 2), the boundary function has to be declared as a global function and outside the scope of any other functions.

4. PAIRWISE SEQUENCE ALIGNMENT

We now consider an example from computational biology, namely, an algorithm for computing the optimal cost of aligning a pair of DNA or RNA sequences. Sequence alignments play a central role in biological-sequence comparison and can reveal important relationships among organisms [11, 26]. We use this example to show how to specify stencil computations in Pochoir when (1) grid cells in time step t depend on data points in time steps deeper than $t-1$, and/or (2) each grid cell consists of multiple fields. The example also demonstrates how Pochoir handles stencil computations with spatial grids whose size and shape may change with each time step.

Our example is specifically Gotoh's algorithm [10] for global pairwise sequence alignment with affine gap penalty. When two sequences are aligned they may become fragmented and gaps may arise. Given a ***gap open cost*** g_o and a ***gap extension cost*** g_e, a run of k gaps in either sequence incurs a total cost of $g_o + g_e \times k$. Moreover, each mismatched aligned character pair incurs a given ***mismatch cost*** m. Gotoh's algorithm finds an alignment of the given sequences such that the total cost of gaps and mismatches is minimized.

Gotoh's algorithm solves three interdependent recurrences that update three different fields — D, I, and G — on a 2D rectangular grid (see [3,10] for details). This grid cannot be directly evaluated as a stencil because of the dependence of each cell on cells in the same row/column. Nevertheless, it can be transformed as shown in Figure 3 to obtain a diamond-shaped

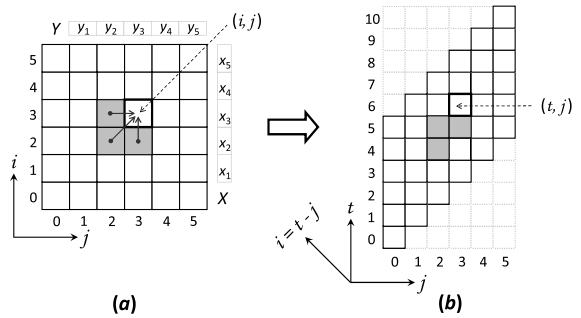

Figure 3: Transforming the PSA dynamic program [10] to a stencil. **(a)** The PSA grid, where each cell (i, j), where $i, j > 0$, depends on cells $(i, j-1)$, $(i-1, j)$ and $(i-1, j-1)$. **(b)** Sliding the columns of the PSA grid upward to form a staircase so that no cell depends on cells on the same row, and thus give it a proper stencil shape.

grid that can be evaluated as a stencil. We obtain the following set of transformed recurrences for computing the optimal alignment cost between sequences $X[1 \ldots n_X]$ and $Y[1 \ldots n_Y]$:

$$D(t,j) = \begin{cases} G(t,j)+g_e & \text{if } t=j>0\,, \\ \min\{G(t-1,j)+g_o, D(t-1,j)\}+g_e & \text{if } t>j>0\,; \end{cases}$$

$$I(t,j) = \begin{cases} G(t,j)+g_e & \text{if } t>j=0\,, \\ \min\{G(t-1,j-1)+g_o, I(t-1,j-1)\}+g_e & \text{if } t>j>0\,; \end{cases}$$

$$G(t,j) = \begin{cases} 0 & \text{if } t=j=0\,, \\ g_o+tg_e & \text{if } t=j>0 \\ & \quad \text{or } t>j=0\,, \\ \min\{G(t-2,j-1)+m\delta, D(t,j), I(t,j)\} & \text{if } t>j>0\,. \end{cases}$$

where $\delta = 1$ if $X[t-j] = Y[j]$ and $\delta = 0$ otherwise. The optimal alignment cost is given by $\min\{G(n_X+n_Y, n_Y), D(n_X+n_Y, n_Y), I(n_X+n_Y, n_Y)\}$. A Pochoir implementation of the stencil computation is shown in Figure 4.

Pochoir provides two ways of specifying stencil computations that update multiple fields. The method used in Figure 4 is to create a C++ structure that contains three fields, as is done in line 1. Another option (not shown) is to use three different arrays for the three different fields. In this case each array must be registered with the Pochoir object. Performance results indicate, however, that the first solution has better locality, because the three fields for the same index are packed together and loaded into the cache simultaneously. In our experiments the structure-based solution ran slightly faster than the solution based on multiple arrays.

Observe from the recurrence for G that $G(t,j)$ depends on $G(t-2, j-1)$. This depth of dependence in the time dimension is inferred from the Pochoir shape `psa_shape`, which is needed for a Pochoir-object declaration in line 7. When a Pochoir array is registered with a Pochoir object, as in line 9, all the shape information associated with the Pochoir object is transferred to the Pochoir array.

Finally, observe that Figure 4 does not define a boundary function, and all boundary conditions are checked inside the kernel function. This choice was made because currently Pochoir's boundary functions do not handle spatial boundaries that change with time. The kernel traverses a rectangular region, and the checks inside it ensure that the stencil is evaluated only inside the required diamond-shaped region within the rectangle. Future research should enable us to improve Pochoir's performance even further by eliminating the overhead of traversing outside the computing domain and reducing or eliminating the boundary checks inside the kernel.

```
1  typedef struct { int D, I, G; } OPT_COST;
2  Pochoir_Boundary_1D(psa_bdry, a, t, i)
3      printf("Access Pochoir_Array opt(%d, %d)\n",
           t, i);
4  Pochoir_Boundary_End
5  int PSA( int nX, char *X, int nY, char *Y,
           int go, int ge, int m ) {
6   Pochoir_Shape_1D psa_shape[]
        = { {0,0}, {-1,0}, {-2,-1}, {-1,-1} };
7   Pochoir_1D psa(psa_shape);
8   Pochoir_Array_1D( OPT_COST ) opt;
9   psa.Register_Array( opt );
10  opt(0,0).G = 0;
11  Pochoir_Kernel_1D( psa_fn, t, j )
12    if ( t >= j && t <= j + nX )
13     if ( t > j && j > 0 ) {
14          int c = (X[t - j] == Y[j]) ? 0 : m;
15          opt(t,j).D = min(opt(t-1,j).G + go,
                             opt(t-1,j).D) + ge;
16          opt(t,j).I = min(opt(t-1,j-1).G + go,
                             opt(t-1,j-1).I) + ge;
17          opt(t,j).G = min(opt(t-2,j-1).G + c,
                             opt(t,j).D, opt(t,j).I);
18      } else {
19          int G_tj = go + t * ge;
20          if ( t > j || j > 0 ) opt(t,j).G = G_tj;
21          if ( t > j ) opt(t,j).I = G_tj + ge;
22          if ( j > 0 ) opt(t,j).D = G_tj + ge;
23      }
24  Pochoir_Kernel_End
25  int t = nX + nY;
26  psa.Run( t, psa_fn );
27  return min(opt(t,nY).G,opt(t,nY).D,opt(t,nY).I);
28 }
```

Figure 4: Pochoir specification for computing optimal pairwise sequence alignment cost with affine gap penalty.

5. LATTICE BOLTZMANN METHOD

In this section we use the lattice Boltzmann method (LBM) as an example of a stencil with a heavyweight computing kernel. We implemented LBM as a 3D 19-point stencil with 19 floating-point fields per grid cell [18]. The kernel requires more than 250 floating-point operations to update each point [21]. The example also illustrates the use of multiple kernels with the same Pochoir object. It also exposes a limitation of the current version of Pochoir to fully optimize kernels containing function calls and explains how macros can be used to work around this problem.

The lattice Boltzmann method (LBM) computes the finite-difference approximation of discrete velocity Boltzmann equation [23], where each cell in a uniform 3D grid is updated in each time step using information from a subset of neighboring cells. Each cell represents a volume element of the fluid, and consists of a collection of fluid particles. Each time step consists of two phases: the streaming/propagation phase and the collision phase. The baseline version of LBM we used is the one in SPEC'06 [12].

Figure 5 shows a Pochoir specification of LBM, which illustrates several interesting aspects of Pochoir. First, although the user arrays `srcGrid` and `dstGrid` are padded by ghost cells, the corresponding Pochoir array `pa` employs a boundary function `lbm_bdry`, defined in lines 4–11, to supply the value of the ghost cell whenever an access to the padded ghost cells occurs. Line 18 associates the boundary function `lbm_bdry` with the Pochoir array `pa`. Second, the initial values of the Pochoir array are copied from the user arrays (in lines 26–27) and the final values are copied back to the user arrays (in lines 32–33). Third, there are two kernels (`lbm_kernel_0` and `lbm_kernel_1` defined, and the one to be used depends on the input argument `simType` at runtime. Finally, both kernels share

```
1  #define DFL1 (1.0/3.0)
2  #define DFL2 (1.0/18.0)
3  #define DFL3 (1.0/36.0)
4  Pochoir_Boundary_3D(lbm_bdry, a, t, z, y, x)
5    PoCellEntry result;
6    result._C = DFL1;
7    result._N = result._S = result._E = result._W =
           result._T =  result._B = DFL2;
8    result._NE = result._NW = result._SE = result.
         _SW = result._NT = result._NB = result._ST
          = result._SB = result._ET = result._EB =
         result._WT = result._WB = DFL3;
9    result._FLAGS = 0.0;
10   return result;
11 Pochoir_Boundary_End

12 void RunLbm(MAIN_SimType simType, LBM_Grid
       srcGrid, LBM_Grid dstGrid, int numTimeSteps)
13 {
14   Pochoir_Shape_3D lbm_shape[] = { {1,0,0,0},
         {0,0,0,0}, {0,0,1,0}, {0,0,-1,0},
         {0,1,0,0}, {0,-1,0,0}, {0,0,0,1},
         {0,0,0,-1}, {0,1,1,0}, {0,-1,1,0},
         {0,1,-1,0}, {0,-1,-1,0}, {0,0,1,1},
         {0,0,1,-1}, {0,0,-1,1}, {0,0,-1,-1},
         {0,1,0,1}, {0,1,0,-1}, {0,-1,0,1},
         {0,-1,0,-1}};
15   Pochoir_3D lbm(lbm_shape);
16   Pochoir_Array_3D(PoCellEntry) pa(SIZE_Z, SIZE_Y
         , SIZE_X);
17   lbm.Register_Array(pa);
18   pa.Register_Boundary(lbm_bdry);
19   Pochoir_Kernel_3D(lbm_kernel_0, t, z, y, x)
20     HandleInOutFlow(t, z, y, x);
21     PerformStreamCollide(t, z, y, x);
22   Pochoir_Kernel_End
23   Pochoir_Kernel_3D(lbm_kernel_1, t, z, y, x)
24     PerformStreamCollide(t, z, y, x);
25   Pochoir_Kernel_End
26   CopyLbmGridToPochoirGrid(srcGrid, pa, 0);
27   CopyLbmGridToPochoirGrid(dstGrid, pa, 1);
28   if (simType == CHANNEL)
29     lbm.Run(numTimeSteps, lbm_kernel_0);
30   else
31     lbm.Run(numTimeSteps, lbm_kernel_1);
32   CopyPochoirGridToLbmGrid(srcGrid, pa, 0);
33   CopyPochoirGridToLbmGrid(dstGrid, pa, 1);
34 }
```

Figure 5: Pochoir specification for a lattice Boltzmann method.

the stream and collide phases, and so it is naturally to code these two phases as functions `PerformStreamCollide` which can be called by both kernels.

LBM presents a challenge to the current Pochoir compiler. In order to maximize the performance, the Pochoir compiler must inspect the code in the kernel (otherwise, Pochoir has to employ a more conservative strategy which doesn't always yields the best performance). Since Pochoir currently does not perform interprocedural analysis, however, it cannot understand how the accesses to the Pochoir array are performed within `PerformStreamCollide` and `HandleInOutFlow`, and so its performance is hindered. Instead, if these two functions are declared as macros, however, the Pochoir compiler can understand and optimize the code effectively. Handling interprocedural analysis within Pochoir represents future research.

6. EMPIRICAL RESULTS

The benchmark results shown in Figure 6 indicate that stencil codes generated by Pochoir outperforms serial- or parallel-loop implementations. All experiments were run on a 12-core Intel Core i7 (Nehalem) machine with a private 64 KB L1-cache, a private 256 KB L2-cache, and a shared 12 MB L3-cache. The C++ compiler was the Intel C++ version 12.0.0 compiler with Intel Cilk Plus [13]. The performance is measured in terms of space-time grid points per second, calculated by multiplying the volume of the spatial grid by the number of time steps and then dividing by the overall execution time.

7. CONCLUSION

We are currently improving Pochoir's expressiveness, finding new opportunities for optimization, and employing it in more applications. Since many users of multicore technology do not understand parallelism and caching well, we believe that tools such as Pochoir can allow them to exploit the capabilities of modern multicore architectures without undergoing a steep learning curve.

8. ACKNOWLEDGMENTS

We are grateful to the Intel Cilk team for software support during the development of Pochoir. Many thanks to Bradley Kuszmaul of MIT CSAIL for his contributions to Pochoir. Thanks to Geoff Lowney of Intel for his support and critical appraisal of the system and to Robert Geva of Intel for an enormously helpful discussion that led to a great simplification of the Pochoir specification language. Members of the MIT CSAIL Supertech Research Group provided us with many helpful discussions.

9. REFERENCES

[1] E. Baysal, D. Kosloff, and J. Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.

[2] R. Bleck, C. Rooth, H. Dingming, and L. Smith. Salinity-driven thermocline transients in a wind-and thermohaline-forced isopycnic coordinate model of the North Atlantic. *Journal of Physical Oceanography*, 22(12):1486–1505, 1992.

[3] R. Chowdhury, H. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(3):495–510, 2010.

[4] 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 *Supercomputing*, pages 4:1–4:12. ACM/IEEE, 2008.

[5] H. Dursun, K. 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 *International Euro-Par Conference on Parallel Processing*, pages 642–653, 2009.

[6] H. Dursun, K. 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, 2009.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297. IEEE, 1999.

[8] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, pages 361–366. ACM, 2005.

[9] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA*, pages 271–280. ACM, 2006.

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

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, New York, 1997.

[12] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

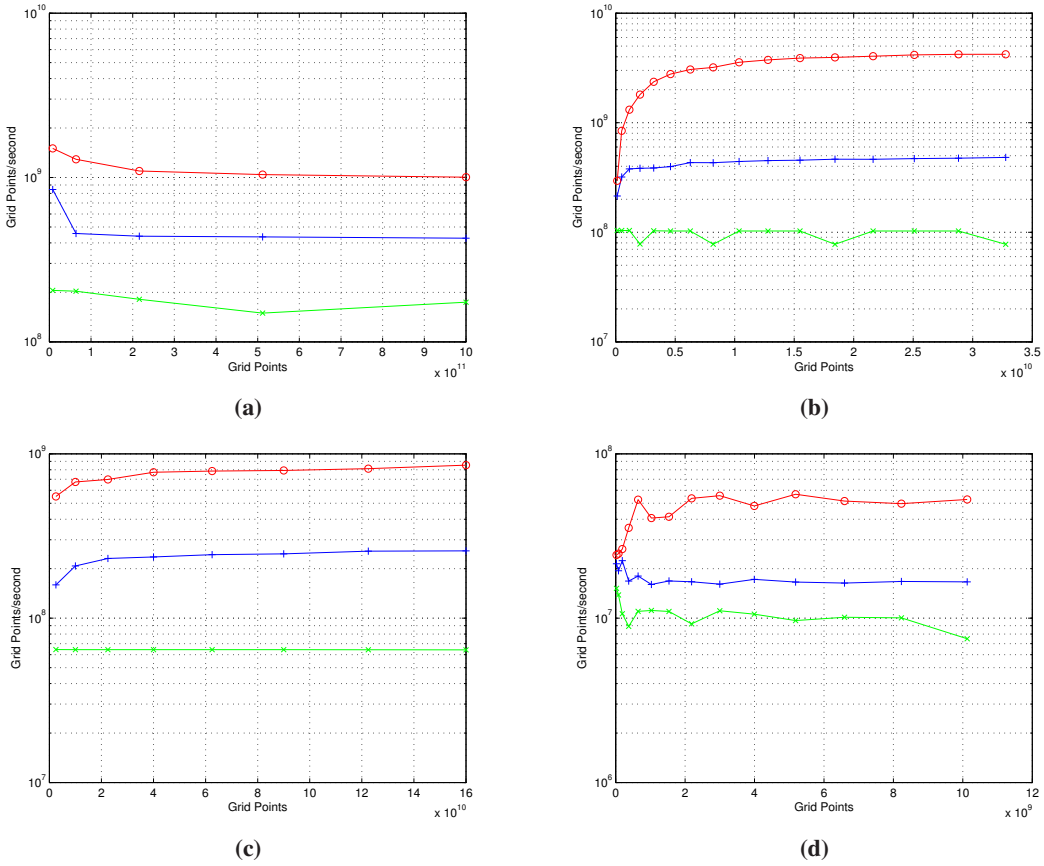

Figure 6: Comparing the number of grid points processed per second (semilogarithmic scale) for Pochoir-generated code on 12 cores versus serial- and parallel-loop implementations. In all figures, the top curve is for the Pochoir-generated code, the middle curve is for parallel loops, and the bottom curve is for serial loops. **(a)** A 3D wave equation with a nonperiodic boundary condition executing for 1000 time steps. **(b)** A 2D heat equation on a torus executing for 3200 time steps. **(c)** 1D pairwise sequence alignment (no time steps). **(d)** A lattice Boltzmann method with a nonperiodic boundary condition for 3000 time steps.

[13] 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`.

[14] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Workshop on Memory System Performance and Correctness*, pages 51–60. ACM, 2006.

[15] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Workshop on Memory System Performance*, pages 36–43. ACM, 2005.

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

[17] G. McMechan. Migration by extrapolation of time-dependent boundary values. *Geophysical Prospecting*, 31(3):413–420, 1983.

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

[19] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[20] 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.

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

[22] L. Peng, R. Seymour, K. 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. IEEE, 2009.

[23] T. Platkowski and R. Illner. Discrete velocity models of the Boltzmann equation: a survey on the mathematical aspects of the theory. *SIAM Review*, 30(2):213–255, 1988.

[24] A. Taflove and S. Hagness. *Computational electrodynamics: The finite-difference time-domain method*. Artech House, Norwood, MA, 2000.

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

[26] M. Waterman. *Introduction to Computational Biology*. Chapman & Hall, London, 1995.

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