Coding Stencil Computations Using the Pochoir Stencil-Specification Language

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

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 straightforward parallel loop code. 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 system automatically parallelizing and optimizing stencils. The stencil is specified using the Pochoir specification language, which is embedded in C++. The Pochoir package contains two major components: the Pochoir template library and the Pochoir compiler. Each of these components anchors one phase of a two-phase methodology for stencil compilation.

After specifying a stencil computation in the Pochoir specification language, the user first compiles the program with the C++ compiler using 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 using the Pochoir template library, 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 ordinary C++ compilation with the Pochoir template library to type-check and catch any inconsistencies in the specification. The two phases are linked se-

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.

mantically by the following promise:

The Pochoir Guarantee: *If the stencil program compiles and runs with the Pochoir template library, no errors will occur 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 stencil specification language can be found in [25]. Section 2 describes a stencil for a 3-dimensional wave equation for seismic imaging. Section 3 gives the example of pairwise sequence alignment for computational biology. Section 4 describes the lattice Boltzmann method for theoretical physics. Section 5 compares the performance of the Pochoir-generated code with serial loops and parallel loops. Section 6 offers some concluding remarks.

2 Three-dimensional wave equation

This section illustrates a Pochoir specification of a stencil computation through the example of solving a 3D wave equation. We use a 3D 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]. The example also illustrates how periodic and nonperiodic boundary conditions can be specified in Pochoir.

Figure 1 shows the Pochoir source code for the 3D wave equation with a nonperiodic boundary condition. Line 1 declares a ***Pochoir object*** named `fd_3D` which will contain all state necessary to perform the stencil computation. The first two arguments in the declaration specify that the object is associated with stencil computations with single-precision floating-point (C++ type `float`) entries on spatial grids with 3 dimensions. The last argument says that the stencil depends on 2 time steps: the current one and the previous one.

Line 2 declares `pa` as a `Nx×Ny×Nz` ***Pochoir array*** of floating-point numbers, which represents the spatial grid. The arguments have the same meanings as in the Pochoir object declaration. Line 3 declares `fd_shape_3D` as the ***Pochoir shape***, the "footprint" of the stencil. The shape is represented as an array of arrays, each of which has 4 integer numbers representing the offset of each grid point in the stencil computing kernel relative to a center point $(\mathtt{t},\mathtt{z},\mathtt{y},\mathtt{x})$. Line 4 registers the shape with `fd_3D`.

Lines 6–8 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 $(\mathtt{z},\mathtt{y},\mathtt{x}) \notin [0,\mathtt{Nz}) \times [0,\mathtt{Ny}) \times [0,\mathtt{Nx})$.

```
1   Pochoir<float, 3, 2> fd_3D;
2   Pochoir_Array<float, 3, 2> pa(Nz, Ny, Nx);
3   Pochoir_Shape<3> 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}};
4   fd_3D.registerShape(fd_shape_3D);

6   Pochoir_Boundary_3D(fd_bv_3D, arr, t, z, y, x)
7     return 0;
8   Pochoir_Boundary_End
9   fd_3D.registerBoundaryFn(pa, fd_bv_3D);

11  Pochoir_Kernel_3D(fd_3D_fn, t, z, y, x)
12    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)));
13    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;
14  Pochoir_Kernel_End

16  /* Initialize the Pochoir_Array pa */
17  for (int z = 0; z < Nz; ++z)
18    for (int y = 0; y < Ny; ++y)
19      for(int x = 0; x < Nx; ++x) {
20    float r = abs((float)(x - Nx/2 + y - Ny/2 + z
        - Nz/2) / 30);
21    r = max(1 - r, 0.0f) + 1;
22    pa(0, z, y, x) = r;
23  }

25  fd_3D.run(T, fd_3D_fn);
```

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

For this nonperiodic boundary condition, `fd_bv_3D` supplies a 0 value when an off-domain access occurs. Line 9 registers the boundary function with `fd_3D`.

Lines 11–14 declare `fd_3D_fn` as the computing kernel for the stencil computation. The compiler cannot infer the stencil shape from the kernel, because the kernel can be arbitrary C++ code, and accesses to grid points can be hidden in subroutines. When the code is run with the Pochoir template library during the first phase of the Pochoir two-phase methodology, however, the system will complain if an access to a grid point deviates from the shape registered in Line 4.

Finally, we are ready to initialize and run the computation. Lines 17–23 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

```
1  Pochoir_Boundary_2D(fd_bv_3D, arr, t, z, y, x)
2    int new_z = (z + arr.size(2)) % arr.size(2);
3    int new_y = (y + arr.size(1)) % arr.size(1);
4    int new_x = (x + arr.size(0)) % arr.size(0);
5    return arr.get(t, new_z, new_y, new_x);
6  Pochoir_Boundary_End
```

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

time steps before executing the stencil. Finally, line 25 executes the stencil object `fd_3D` for T time steps, specifying the kernel function `fd_3D_fn`.

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. Lines 2–4 employ the modulo operator to compute the index on the 3D torus of each spatial coordinate. Then line 5 obtains and returns the required entry based on the new indices.

3 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 grid that can be evaluated as a stencil. We obtain the following set of transformed recurrences for computing the optimal alignment cost between

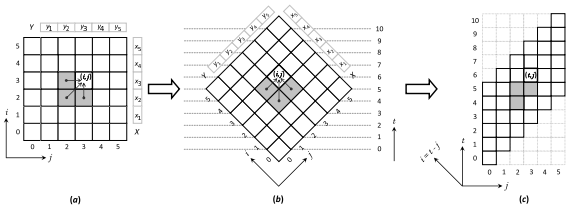

Figure 3: Transforming the PSA dynamic program [10] to a stencil. **(a)** The PSA grid, where each cell (i, j) $[i, j > 0]$ depends on cells $(i, j-1)$, $(i-1, j)$ and $(i-1, j-1)$. **(b)** Rotating the PSA grid from (a) by 45° in the counterclockwise direction so that no cell depends on cells on the same row (i.e., diagonals of (a)). **(c)** Sliding the rows of (b) to the right to give it a proper stencil shape.

sequences $X[1 \ldots n_X]$ and $Y[1 \ldots n_Y]$:

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

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

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

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 the 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[1] is specified in the Pochoir object and the Pochoir array declarations (last argument in both cases) in line 3 and line 4, respectively, as well as in the Pochoir shape declaration in line 7.

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

[1] If time step t depends on time steps down to $t-d$, then the depth of dependence is $d+1$.

```
1  typedef struct { int D, I, G; } OPT_COST;
2  int PSA( int nX, char *X, int nY, char *Y,
           int go, int ge, int m ) {
3   Pochoir< OPT_COST, 1, 3 > psa;
4   Pochoir_Array< OPT_COST, 1, 3 > opt;
5   Pochoir_Domain J(0, nY + 1);
6   Pochoir_Shape< 1 > psa_shape[]
        = { {0, 0}, {-1, 0}, {-2, -1}, {-1, -1} };
7   opt(0,0).G = 0;
8   Pochoir_Kernel_1D( psa_fn, t, j )
9     if ( t >= j && t <= j + nX )
10     if ( t > j && j > 0 ) {
11         int c = (X[t - j] == Y[j]) ? 0 : m;
12         opt(t,j).D = min(opt(t-1,j).G + go,
                            opt(t-1,j).D) + ge;
13         opt(t,j).I = min(opt(t-1,j-1).G + go,
                            opt(t-1,j-1).I) + ge;
14         opt(t,j).G = min(opt(t-2,j-1).G + c,
                            opt(t,j).D, opt(t,j).I);
15     } else {
16         int G_tj = go + t * ge;
17         if ( t > j || j > 0 ) opt(t,j).G = G_tj;
18         if ( t > j ) opt(t,j).I = G_tj + ge;
19         if ( j > 0 ) opt(t,j).D = G_tj + ge;
20     }
21  Pochoir_Kernel_End
22  psa.registerArray( opt );
23  psa.registerDomain( J );
24  psa.registerShape( psa_shape );
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 [10].

```
1  void RunLbm(MAIN_SimType simType, LBM_Grid srcGrid
       , LBM_Grid dstGrid, int numTimeSteps)
2  {
3    Pochoir<CellEntry, 3> lbm;
4    Pochoir_Array<PoCellEntry, 3> pa((2*MARGIN_Z+
         SIZE_Z), (2*MARGIN_Y+SIZE_Y), (2*MARGIN_X+
         SIZE_X));
5    Pochoir_Domain X(0+MARGIN_X, SIZE_X+MARGIN_X), Y
         (0+MARGIN_Y, SIZE_Y+MARGIN_Y), Z(0+MARGIN_Z
         , SIZE_Z+MARGIN_Z);
6    Pochoir_Shape<3> lbm_shape[8] = {{1, 0, 0, 0},
         {0, 0, 0, 0}, {0, 0, 0, +MARGIN_X},  {0, 0,
          0, -MARGIN_X}, {0, 0, +MARGIN_Y, 0},  {0,
         0, -MARGIN_Y, 0}, {0, +MARGIN_Z, 0, 0},
         {0, -MARGIN_Z, 0, 0}};
8    Pochoir_Kernel_3D(lbm_kernel_0, t, z, y, x)
9      HandleInOutFlow(t, z, y, x);
10     PerformStreamCollide(t, z, y, x);
11   Pochoir_Kernel_End
13   Pochoir_Kernel_3D(lbm_kernel_1, t, z, y, x)
14     PerformStreamCollide(t, z, y, x);
15   Pochoir_Kernel_End
17   lbm.registerArray(pa);
18   lbm.registerShape(lbm_shape);
19   lbm.registerDomain(Z, Y, X);
21   CopyLbmGridToPochoirGrid(srcGrid, pa, 0);
22   CopyLbmGridToPochoirGrid(dstGrid, pa, 1);
24   if (simType == CHANNEL)
25     lbm.run(numTimeSteps, lbm_kernel_0);
26   else
27     lbm.run(numTimeSteps, lbm_kernel_1);
29   CopyPochoirGridToLbmGrid(srcGrid, pa, 0);
30   CopyPochoirGridToLbmGrid(dstGrid, pa, 1);
31 }
```

Figure 5: Pochoir specification for a lattice Boltzmann method.

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.

4 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 19-point 3D stencil with 19 floating-point fields per grid cell [18]. The kernel includes more than 250 FLOPs [21]. The example also illustrates the use of zero padding for improved performance, and 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 shows one way 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, since the user arrays `srcGrid` and `dstGrid` are already zero padded, the corresponding Pochoir array `pa` needs to be declared as padded (in line 4) as well. Moreover, the Pochoir domains must also be adjusted correspondingly (in line 5). If the computing domain is not exactly the same as that of the Pochoir array, such as the zero-padded case in LBM, the user is responsible for registering the adjusted computing domain with corresponding ***Pochoir*** object as in line 19. Second, the initial values of the Pochoir array are copied from the user arrays (in lines 21–22) and the final values are copied back to the user arrays (in lines 29–30). 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 the stream and collide phases, and so it is naturally to code these two phases as a

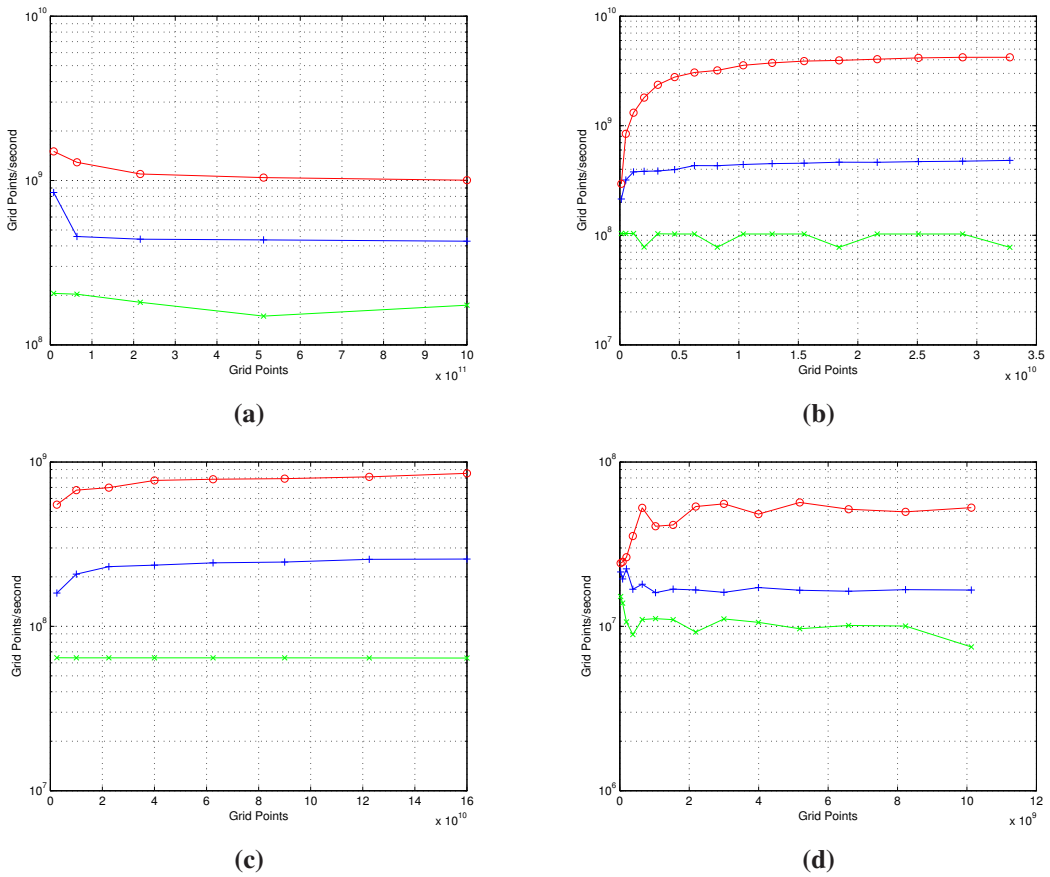

Figure 6: Comparing the number of grid points processed per second (semilogarithmic scale) for Pochoir-generated code on 12 cores versus serial- and parallel-loop implementations. In all figures, the top curve is for the Pochoir-generated code, the middle curve is for parallel loops, and the bottom curve 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.

function `PerformStreamCollide` called by both kernels.

LBM presents a challenge to the current Pochoir compiler. In order to optimize the base case, the Pochoir compiler must inspect the code in the kernel. 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. By declaring these two functions as macros instead, however, the Pochoir compiler can analyze their behavior and optimize the code effectively. Handling interprocedural analysis within Pochoir represents future research.

5 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.

6 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.

7 Acknowledgments

We are grateful to the Intel Cilk team for support during the development of Pochoir, and especially to Will Leiserson for his responsiveness. 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. Will Hasenplaugh of Intel and members of the MIT Supertech Research Group provided us with many helpful discussions.

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 *ACM/IEEE Conference on Supercomputing*, pages 4:1–4:12, 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 *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 533–538, 2009.

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

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

[9] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–280, 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.

[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 *ACM Workshop on Memory System Performance and Correctness*, pages 51–60, 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 *ACM Workshop on Memory System Performance*, pages 36–43, 2005.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 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 *ACM Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 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 *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 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. Submitted for publication, 2011.

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

[27] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, pages 1–14. IEEE, 2008.