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 compiles 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-
nationally 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 [1, 17]. This section illustrates a Pochoir specification of a 3D wave equation. We use a 3D finite-difference (3DFD) computation through the example of solving a 3D wave equation. The example also illustrates how periodic and nonperiodic boundary conditions have practical applications in seismic imaging [1, 17]. Section 3 gives the example of pairwise sequence alignment in 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 `Nz x Ny x 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 `(t, z, y, x)`. Line 4 registers the shape with `fd_3D`.

Lines 6–8 define a function `fd_by_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 `(z, y, x) \not\in [0, Nz) \times [0, Ny) \times [0, Nz)`. 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

```cpp
Pochoir<float, 3, 2> fd_3D;
Pochoir_Array<float, 3, 2> pa(Nz, Ny, Nz);
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),
    (1, 0, 0, 0), (0, 0, -2, 0), (0, 0, 2, 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)};
fd_3D.registerShape(fd_shape_3D);
Pochoir_Boundary_3D(fd_by_3D, arr, t, z, y, x)
Pochoir_Kernel_3D(fd_3D_fn, t, z, y, x)
float div = c * 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+2, x) + pa(t, z, y, x+2) + pa(t, z, y, x-2) + pa(t, z+2, y, x) + pa(t, z-2, y, x)) + (pa(t, z+3, y, x) + pa(t, z-3, y, x)) + c3 * ((pa(t, z, y+3, x) + pa(t, z, y+4, x) + pa(t, z, y+4, x) + pa(t, z, y-4, x) + pa(t, z+4, y, x) + pa(t, z-4, y, x))));
Pochoir_Kernel_End

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_by_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

```cpp
Pochoir<float, 3, 2> fd_3D;
Pochoir_Array<float, 3, 2> pa(Nz, Ny, Nz);
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),
    (1, 0, 0, 0), (0, 0, -2, 0), (0, 0, 2, 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)};
fd_3D.registerShape(fd_shape_3D);
Pochoir_Boundary_3D(fd_by_3D, arr, t, z, y, x)
Pochoir_Kernel_3D(fd_3D_fn, t, z, y, x)
float div = c * 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+2, x) + pa(t, z, y, x+2) + pa(t, z, y, x-2) + pa(t, z+2, y, x) + pa(t, z-2, y, x)) + (pa(t, z+3, y, x) + pa(t, z-3, y, x)) + c3 * ((pa(t, z, y+3, x) + pa(t, z, y+4, x) + pa(t, z, y+4, x) + pa(t, z, y-4, x) + pa(t, z+4, y, x) + pa(t, z-4, y, x))));
Pochoir_Kernel_End

/* Initialize the Pochoir_Array pa */
  for (int y = 0; y < Ny; ++y)
    for (int x = 0; x < Nx; ++x) {
      float z = abs(float(x) - Nx/2 + y - Ny/2 + z - Nz/2) / 30);
      r = max(1 - r, 0.0f) + 1;
      pa(0, x, y, z) = r;
    }
fd_3D.run(T, fd_3D_fn);
```
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 sequences $X[1 \ldots n_x]$ and $Y[1 \ldots n_y]$:

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

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

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

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 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...
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 (lmb_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.

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 (lmb_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.
function \texttt{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 \texttt{PerformStreamCollide} and \texttt{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 outperform 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

