User Manual for the Pochoir Stencil Compiler

Charles E. Leiserson Yuan Tang

MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

1. INTRODUCTION

Pochoir (pronounced "PO-shwar") [**?**, 22] 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*** [1, 2, 4, 5, 9, 10, 13–15, 17–19, 21, 23] 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 [8, 20] divide-and-conquer stencil codes [9, 10] 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.

The Pochoir stencil compiler achieves a substantial performance improvement over a straightforward loop parallelization for typical stencil applications. Pochoir 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 [3,11,16] embedded in the base language C++ with the Cilk multithreading extensions [12].

The remainder of this manual is organized as follows. Section 2 describes how the Pochoir system can be installed. Section 3 illustrates how a 2D heat equation can be specified using the Pochoir specification language. Section 4 provides a full specification of the Pochoir embedded language. Section A contains a complete list of error messages generated by the Pochoir system.

If you encounter bugs in the Pochoir system, please email `pochoir@csail.mit.edu`.

2. INSTALLATION AND USE

This section describes how to acquire, install, and use Pochoir on your Linux system. The Pochoir compiler has been mainly tested under Ubuntu 10.04 and 11.04. Other Linux systems should also work but have not yet been tested. If you wish to port Pochoir to another operating system, please let us know by emailing `pochoir@csail.mit.edu`.

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.

Preliminaries

Before you start, you need following tool suite in your environment:

- Intel C++ Compiler (Available with C++ Composer XE 2011 for Linux), version 12.0.0 or later, with Cilk Plus extension.
- The Glasgow Haskell Compilation System, version 6.12.1 or later, Parsec-2.1.0.1 or later, if you want to compile the Pochoir compiler for your system. (By default, the system carries a Pochoir compiler for Intel 64 architecture for your convenience.)

Acquire the software

Please send email to `pochoir@csail.mit.edu` requesting a copy of the Pochoir system. You will receive a tarball with the software.

Set up the Pochoir environment

Suppose that the home directory for Pochoir package is `$pochoir`, which is the directory containing all the `pochoir_xxx.hpp` and `*.hs` files. Typing

```
% make pochoir
```

in directory `$pochoir`, provided the Haskell compilation system and Parsec package are correctly installed, will produce the Pochoir compiler named `./pochoir`.

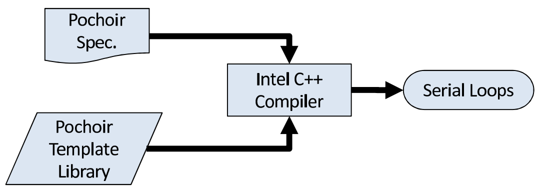
Compiling a program using Pochoir

By default, the Pochoir compiler assumes the Intel C++ compiler can be accessed via the name `icpc`. Before compilation, set up the environment variable by typing:

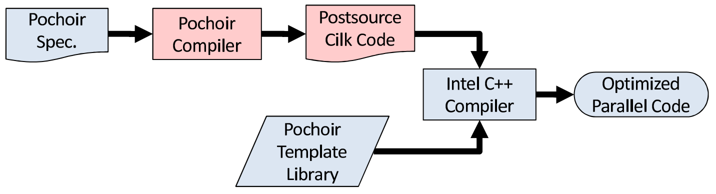
```
% export POCHOIR_LIB_PATH=$pochoir
```

You can also add this line to your start-up shell script.

As shown in Figure 1, 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++ pro-



(a)



(b)

Figure 1: 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.

gram 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.

Usage of the Pochoir compiler

The shell command `pochoir` without arguments causes the Pochoir compiler to output the basic usage of Pochoir compiler as follows:

```
Usage: pochoir [OPTION] [filename]

Try 'pochoir --help' for more options.
```

```
Usage: pochoir [OPTION] [filename]

Run the Pochoir stencil compiler on [filename].

-auto-optimize
        Let the Pochoir compiler automatically
        choose the best optimizing level.  (Default)

-debug
        Perform Phase-1 compilation by running the
        ordinary C++ compiler with the Pochoir template
        library.

-split-macro-shadow
        Use macro tricks to split the interior
        and boundary regions.

-split-pointer
        Split the interior and boundary regions,
        and use ordinary C-style pointers to optimize
        the base case.
```

Phase-1 compilation

To compile a stencil application named `stencil.cpp` for debugging, you need to supply a `-debug` option to the Pochoir compiler as following:

```
CC = pochoir
stencil: stencil.cpp
    ${CC} -o stencil -O0 -g -debug stencil.cpp
```

In this phase, the code will be compiled against a template library which provides a functionally correct serial looping implementation of stencil algorithm. Also, in this phase I, the template library will try to capture as many bugs as possible as shown in Appendix Section A. The command line option `-debug` tells the `pochoir` compiler that it's the Phase 1 compilation.

Phase-2 compilation

To compile your stencil specification for the Phase-2 optimization, you just need to eliminate the `-debug` option from the command line and supply any other optimizing option you might supply to `icpc` compiler to Pochoir :

```
CC = pochoir
stencil: stencil.cpp
    ${CC} -o stencil -O3 stencil.cpp
```

Examples

The directory `Examples` in the distribution contains many examples illustrating the usage of Pochoir, including Conway's Game of Life, 2D heat equation, 3D wave equation, RNA secondary structure alignment, etc.

3. TUTORIAL

To illustrate how to use Pochoir, consider the 2D ***heat equation*** [6]

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

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:

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

Figure 2 shows the Pochoir source code for the periodic 2D heat equation. 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

```
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}, {0,1,0},
             {0,-1,0}, {0,0,0}, {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 2: The Pochoir stencil source code for a periodic 2D heat equation. Pochoir keywords are boldfaced.

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. Figure 3 shows how nonperiodic stencils can be specified, including how to specify Dirichlet and Neumann boundary conditions [7]. 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;`".

4. 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 have avoided to the extent possible making the language too specific to the Pochoir compiler, the Intel C++ compiler, and

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

Pochoir_Boundary_2D(zero_bdry, a, t, x, y)
  return 0;
Pochoir_Boundary_End
```

(a)

```
Pochoir_Boundary_2D(toroidal, a, t, x, y)
  return a.get( t,
                mod(x, a.size(1)),
                mod(y, a.size(0)) );
Pochoir_Boundary_End
```

(b)

```
Pochoir_Boundary_2D(cylindrical, a, t, x, y)
  if (x < 0) || (x >= a.size(1))
    return 0;
  return a.get( t, x, mod(y, a.size(0)) );
Pochoir_Boundary_End
```

(c)

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

(d)

```
Pochoir_Boundary_2D(neumann, a, t, x, y)
  int xx(x), yy(y);
  if (x<0) xx = 0;
  if (x>=a.size(1)) xx = a.size(1);
  if (y<0) yy = 0;
  if (y>=a.size(0)) yy = a.size(0);
  return a.get(t, xx, yy);
Pochoir_Boundary_End
```

(e)

Figure 3: Pochoir specifications for a variety of boundary conditions. (a) ***Grid***: Nonperiodic with constant value 0. (b) ***Toroidal***: periodic in both x and y. (c) ***Cylindrical***: nonperiodic in x with constant value 0 and periodic in y. (d) ***Dirichlet***: varying with time. (e) ***Neumann***: constrained first derivative (0).

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:

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

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`**($name, array, idx_t, idx_{dim-1}, \ldots, idx_1, idx_0$)
  ⟨*definition*⟩
  `Pochoir_Boundary_End`

This construct defines a ***boundary function*** called *name* that will be invoked 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 ⟨*definition*⟩) 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`**($name, array, idx_t, idx_{dim-1}, \ldots, idx_1, idx_0$)
  ⟨*definition*⟩
  `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 ⟨*definition*⟩ 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 $T+T'+k-1$.

APPENDIX

A. ERROR MESSAGES

The Pochoir template library used during Phase-1 compilation reports as many bugs as possible to assist debugging. This section documents the error messages. If your code compiles and runs correctly during Phase 1, it should not encounter any compilation or runtime problems during Phase 2. If it does, please report the bug to `pochoir@csail.mit.edu`.

List of Pochoir error messages

Compile-time errors

- Pochoir environment variable not set

Run-time errors

- Pochoir off-shape access error
- Pochoir array registration error
- Pochoir array access error
- Pochoir array size mismatch error
- Pochoir illegal access by boundary function error

Pochoir environment variable not set

When the Pochoir compiler is invoked, it will automatically check the value of environmental variable `POCHOIR_LIB_PATH` and include it as the path to the Pochoir template library. If the environmental variable is not set up properly, the Pochoir compiler reports:

```
Pochoir environment variable not set:
POCHOIR_LIB_PATH
```

Pochoir off-shape access error

This error message will try to capture the mismatch between the shape specified via `Pochoir_Shape` and registered in a Pochoir object declaration, and the shape the user really used in the `Pochoir_Kernel_`*dim*`D`. For example, suppose that the user writes the following piece of stencil code, as in Figure 4:

```
    Pochoir_Shape_1D heat_shape_1D[] = {{1, 0}, {0,
        1}, {0, -1}, {0, 0}};
    Pochoir_1D heat_1D(heat_shape_1D);
    Pochoir_Array_1D(double) a(N_SIZE);
    a.Register_Boundary(heat_bdry);
    heat_1D.Register_Array(a);
    Pochoir_Kernel_1D(heat_1D_fn, t, i)
       a(t+1, i) = 0.125 * (a(t, i+1) - 2.0 * a(t, i
           ) + a(t, i-2));
    Pochoir_Kernel_End
    ...... /* Initialization */
    heat_1D.Run(T_STEP, heat_1D_fn);
```

Figure 4: The Pochoir stencil source code for a periodic 1D heat equation.

After the user compiles and runs it with the normal C++ compiler, the Pochoir template library reports the following errors at runtime:

```
Pochoir off-shape access error:
Pochoir array index (0, 999)
Shape index {0, 2}
Input Pochoir_Shape<1> =
{{1, 0}, {0, 1}, {0, -1}, {0, 0}}
```

The Pochoir template library checks for shape mismatch errors on temporal as well as spatial dimensions.

Pochoir array registration error

If you run a Pochoir object without registering any Pochoir arrays, the Pochoir template library reports:

```
Pochoir registration error:
You forgot to register Pochoir array.
```

Pochoir array access error

Accessing a Pochoir array before registering it with a Pochoir object causes the Pochoir template library to report:

```
Pochoir array access error:
A Pochoir array is accessed without being registered
with a Pochoir object.
```

Pochoir array size mismatch error

There can be multiple Pochoir arrays participating in one stencil computation, that is, be registered with the same Pochoir object. If the user registers Pochoir arrays with different sizes with the same Pochoir object, the Pochoir template library reports:

```
Pochoir array size mismatch error:
Registered Pochoir arrays have different sizes.
```

Pochoir illegal access by boundary function error

Usually, the Pochoir compiler assumes that the access to a Pochoir array in the boundary function should be within the domain by using the `get` method. Suppose that an off-boundary access in boundary function occurs, such as

```
Pochoir_Boundary_2D(heat_bv_2D, arr, t, i, j)
    return arr.get(t, -1, -1);
Pochoir_Boundary_End
```

The Pochoir template library reports:

```
Pochoir illegal access by boundary function error:
Out-of-range access by boundary function at index
(0, -1, -1)
```

B. REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

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

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

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

[10] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[11] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, December 1996.

[12] 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`.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

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

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

[23] 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.