(B) Project summary: SHF: AF: Medium: Collaborative Research: The Pochoir Stencil Compiler

Many high-end scientific applications perform ***stencil computations*** in their inner loops. 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. Stencil computations are conceptually simple to implement using nested loops, but looping implementations suffer from poor cache performance on multicore processors. Cache-oblivious divide-and-conquer stencil codes can achieve an order of magnitude improvement in cache efficiency over looping implementations, but most programmers find it difficult to write cache-oblivious stencil codes. Moreover, open problems remain in adapting these algorithms to realistic applications that lack the perfect regularity of simple examples. This project's investigation of cache-oblivious stencil compilation will enable ordinary programmers of stencil computations to enjoy the benefits of multicore technology without requiring them to write code any more complex than naive nested loops.

This research will develop a language embedded in C++ that can express stencil computations concisely and can be compiled automatically into highly efficient algorithmic code for multicore processors and other platforms. The ***Pochoir stencil compiler*** will compile stencil computations that exhibit

- complex boundary conditions, such as periodic, constant, Dirichlet, Neumann, mirrored, and phase factors;
- irregularities, including macroscopic and microscopic inhomogeneities, as well as irregular shapes;
- general complex dependencies, such as push dependencies, horizontal dependencies, and dynamic dependencies.

To achieve these goals, the researchers will

- develop provably good algorithms for complex stencil computations;
- explore how domain-specific compiler technology can achieve speedups from efficient cache management, processor-pipeline scheduling, and parallel computation.
- investigate how to run stencils efficiently on a wide variety of architectures such as multicore, distributed-memory clusters, graphical processing units, FPGA's, and future exascale machines;
- demonstrate the effectiveness of their research by developing a production-quality stencil compiler;
- develop a benchmark suite and benchmarking system for evaluating Pochoir.

Intellectual merit: Real stencil applications often exhibit complex irregularities and dependencies, which makes it difficult for programmers to produce efficient multicore code for them or to migrate them to other modern hardware platforms. Even simple stencils are hard to code for performance. This research proposes to attack the difficult problem of generating high-efficiency cache-oblivious code for stencil computations that make good use of the memory hierarchy and processor pipelines, starting with simple-to-write linguistic specifications. This effort requires cross-domain technical expertise, including an understanding of multicore programming, strong theoretical skills to develop efficient parallel algorithms and data structures, systems experience to build and tune a compiler and runtime system, knowledge of real applications this technology will benefit, and an aesthetics for language design.

Broad impact: If successful, this research will enable scientific researchers and others to easily produce highly efficient codes for complex stencil computations. The codes will make good use of the memory hierarchy and processor pipelines endemic to multicore processors and will run fast on a diverse set of hardware platforms. A wide variety of stencil-based applications — ranging across physics, biology, chemistry, energy, climate, mechanical and electrical engineering, finance, and other areas — will become easier to develop and maintain, benefiting these application areas, as well as society at large.

Key words: stencil computation; multicore; domain-specific compilation; cache efficiency; parallel computing.

1 Introduction

Many high-end scientific applications — in diverse areas including physics, biology, chemistry, energy, climate, mechanical and electrical engineering, finance, and recreational mathematics — perform ***stencil*** computations [3, 9, 10, 12, 17, 20, 21, 23, 27–29, 34, 35, 47–49, 51, 61, 64, 66, 68, 76, 82, 85] in their inner loops. 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. Stencil computations are conceptually simple to implement using nested loops, but looping implementations suffer from poor cache performance. Divide-and-conquer "cache-oblivious" [25,78] stencil codes [27,28] are much more cache-efficient, but they are difficult to write. When parallelism is factored into the mix, most application programmers do not have the programming skills, time, or patience to produce efficient multithreaded codes. Moreover, real applications include many types of spatiotemporal irregularities and other complications for which open problems remain even at the algorithmic level. This research project aims to understand and develop parallel cache-oblivious stencil compilation, which will allow ordinary programmers to enjoy the performance benefits of multicore and other parallel platforms while paying only for the coding complexity of naive nested loops.

We propose to develop Pochoir (pronounced "PO-shwar"), a compiler and runtime system for implementing stencil computations on multicore processors. The Pochoir language will allow the ***kernel*** (inner loop) of a stencil to be specified separately from the ***boundary conditions*** (along the edge of the space-time computing domain). Pochoir will employ sophisticated algorithmic optimizations to speed up the compiled code. We propose to investigate many domain-specific optimizations, including base-case coarsening, hoisting of boundary checks out of inner loops, recursion unrolling, overlapped time skewing, efficient divide-and-conquer strategies, automatic halo generation, heuristic autotuning, zigzag traversals, and bit tricks to avoid unpredictable branches. In addition, we plan to study how stencil arrays should be laid out in memory to promote cache-friendly accessing. We propose to investigate how complex boundary conditions can be programmed, including periodic, constant, Dirichlet, Neumann, mirrored, and phase conditions.

We propose to investigate how irregularities in stencil computations can be cleanly specified, compiled, and efficiently executed. We shall address ***macroscopic inhomogeneity***, in which different computations are performed in different regions of space-time; ***microscopic inhomogeneity***, in which each point of space-time comprises several arrays that are interlinked by different stencil operations; and ***irregular shapes***, in which the shape of space-time is not rectangular. We shall investigate new kinds of complex dependencies, such as ***push*** dependencies, in which one specifies where data flows *to* rather than where it flows from; ***dynamic*** dependencies that change over time; and ***horizontal*** dependencies that do not run strictly backwards in time.

To be widely useful, Pochoir will target a variety of platforms. Our principal focus will be multicore computers, since they are already in the hands of computational scientists (and ordinary people) everywhere, and they are the building blocks for heterogeneous and scalable systems. Today's multicore computers contain CPU's with a few complex processing cores, but we also propose to explore how to compile for CPU's with many simple cores, such as the Intel MIC architecture [44] and the Intel Polaris chip [43]. We shall also explore graphics-processing units (GPU's) [50, 69], distributed-memory clusters [75], field-programmable gate arrays (FPGA's), and future exascale computers.

To promote iterative improvements during the engineering of Pochoir, we propose to establish a benchmark suite of stencil computations drawn from a variety of disciplines such as physics [23, 61, 64, 82], computational biology [3,12,34], computational finance [47], mechanical engineering [10], adaptive statistical design [70], weather forecasting [1], clinical medicine [13,77], and image processing [38]. We propose to build a benchmarking system that can run benchmarks automatically to compare different architectures, algorithms, and compiler implementations.

We have created an early prototype of Pochoir [83] that compiles simple stencils into parallel cache-oblivious code for multicore systems. On the stencils it can handle, the prototype achieves outstanding speedups that compare favorably with the best hand-tuned or autotuned codes. This experience gives us confidence in our research direction and provides a good starting point for our two-year research effort.

If successful, this research will enable ordinary programmers to express complex stencil codes with little or no code complexity beyond what it would take to write naive nested loops. The Pochoir compiler and runtime system, which we shall release as open-source software, will produce efficient stencil codes for a variety of platforms, making good use of the architectural features these platforms provide for performance without requiring any platform-specific knowledge of the programmer.

The rest of this proposal is organized as follows. Section 2 explains stencils in more detail and describes how they can be executed in a cache-oblivious fashion. Section 3 outlines our proposed compiler design and stencil-specification language. Section 4 explains the foundational algorithms. The next three sections explain complications that arise in real applications which our proposal will address: boundary conditions in Section 5, inhomogeneities in Section 6, and complex dependencies in Section 7. Section 8 describes the issues for targeting stencils to platforms such as GPU's, FPGA's, clusters, and exascale. Section 9 describes the benchmarking system we propose to develop. Sections 10 and 11 describe the intellectual merit and broad impact, respectively, of the proposed activity. Section 12 summarizes our prior NSF support. Section 13 concludes with a collaboration plan.

2 Stencil computations

As a simple example of a stencil computation, consider a 2D ***heat equation*** [23]

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

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. By discretizing space and time, this partial differential equation can be solved approximately by using the following Jacobi-style update equation:

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

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

Computational scientists find such loop nests fairly simple and easy to understand, but looping codes suffer in performance from poor cache locality. Let $\mathcal{M}$ be the number of grid points that fit in cache, and let $\mathcal{B}$ be the number of grid points that fit on a cache line. If the spatial grid does not fit in cache — that is, $XY \gg \mathcal{M}$ — then this simple computation incurs $\Theta(TXY/\mathcal{B})$ cache misses in the ideal-cache model [25].

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

Figure 1: A parallel looping implementation of a stencil computation for the 2D heat equation with periodic boundary conditions.

Figure 2 shows the pseudocode for a more efficient cache-oblivious algorithm called TRAP, which is the basis of the algorithm we propose to use for Pochoir. Unlike the looping code, this algorithm steps time *nonuniformly* over the grid points, which vastly improves cache locality. As we shall describe in Section 4, Figure 2 recursively decomposes space-time into smaller and smaller regions so that completely in-cache time-stepping is attained for sufficiently small subregions. This code achieves $\Theta(TXY/\mathcal{B}\sqrt{\mathcal{M}})$ cache misses, assuming that $X \approx Y$ and $T = \Omega(X)$. TRAP easily outperforms LOOPS on large data sets. For example, we ran both algorithms on a 5000×5000 spatial grid iterated for 5000 time steps using the Intel C++ version 12.0.0 compiler with Intel Cilk Plus [42] on a 12-core Intel Core i7 (Nehalem) machine with a private 32-KB L1-data-cache, a private 256-KB L2-cache, and a shared 12-MB L3-cache. The code based on LOOPS ran in 248 seconds, whereas the code based on TRAP required about 24 seconds, more than a factor of 10 performance advantage and more than a factor of 100 times faster than a naive serial loop nest.

Unfortunately, the performance advantage of this cache-oblivious algorithm is inaccessible to most programmers of scientific computations, because the code is simply too hard for them to write, debug, and tune. Many stencil computations for scientific codes are vastly more complex than this simple heat equation, exacerbating the difficulty. The Pochoir stencil compiler, which is the focus of our proposed research, will provide programmers access to these complex but efficient algorithms through a simple linguistic interface.

It is natural to wonder whether a domain-specific compiler is overkill and if a highly tuned library would suffice. The answers is no. Libraries are generally useful for encapsulating high-performing computations where the inner loops have fixed functionality or can be parametrized using a few numerical parameters. For stencil computations, the key repetitive step, or ***stencil kernel***, residing in the innermost loop has functionality that must be specified by the programmer. The kernel cannot simply be passed as a function pointer to a library, for example, because the ensuing overhead would be prohibitive. Although C++ templates can overcome some overheads, they do not enable an ordinary compiler to divine that certain important optimizations may be applicable. For example, one key optimization specific to stencil computations is to specialize an optimized version, or ***clone***, of the kernel for the common case of grid points in the interior of the space-time grid, as opposed to on the boundary.

3 The Pochoir system

This section describes how we plan to build the Pochoir system and design the basic Pochoir stencil-specification language. We plan to embed the Pochoir stencil-specification language in C++ [80], because C++ is well recognized as the language of choice for object-oriented high-performance coding (see, for example, [40]), and we wish to give programmers the full flexibility of C++ in describing their stencil computations. C++ is a notoriously hard language to parse and type-check, however, and it might seem that

```
TRAP(u; ta, tb; xa, xb, dxa, dxb; ya, yb, dya, dyb)
 1  Δt = tb − ta
 2  Δx = max{xb − xa, (xb + dxbΔt) − (xa + dxaΔt)} // Longer x-base
 3  Δy = max{yb − ya, (yb + dybΔt) − (ya + dyaΔt)} // Longer y-base
 4  k = 0 // Try hyperspace cut
 5  if Δx ≥ 2σ_x Δt
 6      Trisect the zoid with x-cuts
 7      k += 1
 8  if Δy ≥ 2σ_y Δt
 9      Trisect the zoid with y-cuts
10      k += 1
11  if k > 0
12      Assign dependency levels 0, 1, ..., k to subzoids
13      for i = 0 to k // for each dependency level i
14          parallel for all subzoids (ta, tb; xa', xb', dxa', dxb'; ya', yb', dya', dyb') with dependency level i
15              TRAP(ta, tb; xa', xb', dxa', dxb'; ya', yb', dya', dyb')
16  elseif Δt > 1 // time cut: Recursively walk the lower zoid and then the upper
17      TRAP(ta, ta + Δt/2; xa, xb, dxa, dxb; ya, yb, dya, dyb)
18      TRAP(ta + Δt/2, tb; xa + dxaΔt/2, xb + dxbΔt/2, dxa, dxb; ya + dyaΔt/2, yb + dybΔt/2, dya, dyb)
19  else // base case
20      for t = ta to tb − 1, for x = xa to xb − 1, for y = ya to yb − 1
21          u((t+1) mod 2, x, y) = u(t mod 2, x, y)
                + CX · (u(t mod 2, (x−1) mod X, y) + u(t mod 2, (x+1) mod X, y) − 2u(t mod 2, x, y))
                + CY · (u(t mod 2, x, (y−1) mod Y) + u(t mod 2, x, (y+1) mod Y) − 2u(t mod 2, x, y))
22      xa += dxa,   xb += dxb,   ya += dya,   yb += dyb
```

Figure 2: A cache-oblivious algorithm that implements a stencil computation to solve the 2D heat equation.

the Pochoir compiler would need to understand all of C++ in order to compile stencil programs. Indeed, of the few C++ compiler front-ends available (e.g., [14,32,36]), none seem simple enough to base the Pochoir compiler on. Instead, Pochoir will use a novel two-phase compilation scheme that avoids complicated language processing while allowing the programmer full access to C++ language features. We are confident in this approach, because our prototype compiler [83] implements it and attains high performance on simple stencils. Figure 3 illustrates how the two-phase compilation process will work.

For the first phase, which does not actually use the Pochoir compiler at all, the programmer compiles the source program with the ordinary C++ compiler using the Pochoir template library, which implements Pochoir's linguistic constructs using unoptimized but functionally correct algorithms — in fact, it uses serial nested loops. This phase will ensure that the source program is ***Pochoir-compliant***, and it will allow the programmer to debug the functional correctness of the stencil computation using ordinary software-engineering tools.

For the second phase, the programmer runs the source through the Pochoir compiler, which acts as a preprocessor to a Cilk Plus compiler,[1] performing a source-to-source translation into a postsource Cilk Plus program. The postsource is then compiled with the Cilk Plus compiler to produce the optimized binary executable. The Pochoir compiler will make the following promise:

[1]The Pochoir prototype uses the Intel compiler, which supports both vanilla C++ and the Cilk Plus extensions for multithreading, but now that these extensions are available in the GNU Compiler Collection (GCC) [45], we shall attempt to make Pochoir as compiler-independent as possible.

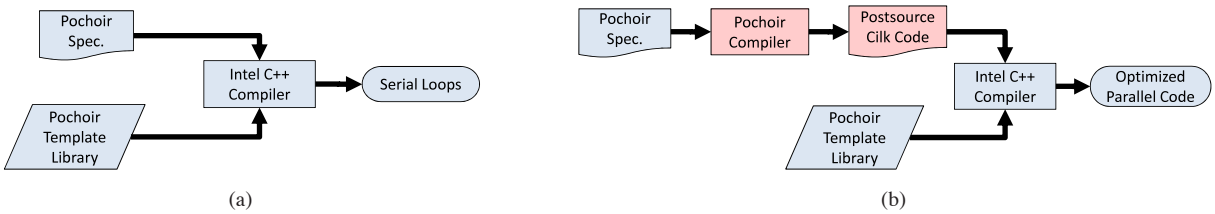

Figure 3: Pochoir's two-phase compilation strategy.

> **The Pochoir Guarantee:** If the stencil program compiles and runs correctly 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 novel two-phase compilation strategy will allow us to build significant domain-specific optimizations into Pochoir without taking on the massive job of parsing and type-checking the full C++ language. Knowing that the source program is Pochoir-compliant will allow the Pochoir compiler in Phase 2 to treat portions of the source as uninterpreted C++ text, confident that the Intel compiler will compile it correctly in the optimized postsource. Moreover, the Pochoir template library will allow the programmer to debug his or her code using a comfortable native C++ tool chain without the complications of the Pochoir compiler.

The proposed Pochoir stencil-specification language will be a domain-specific language [19, 39, 62] embedded in the base language C++ with the Cilk multithreading extensions [42], what Intel calls "Cilk Plus." The basic idea behind the language is to use easily identifiable keywords to mark the various portions of the specification. During Phase 1, these keywords will be treated as macros that expand into C++-template code to implement the functional behavior of the specification, albeit unoptimized. During Phase 2, however, the Pochoir compiler will pick up on these keywords to identify the relevant code sections. The Pochoir compiler will then be free to "understand" as much of the code as it can, knowing that any code that it cannot understand can still be treated as correct uninterpreted C++ text, because it can assume that the code is Pochoir-compliant.

Figure 4 suggests a possible specification of the periodic 2D heat equation in Pochoir. The key linguistic features are explained at the right. The particulars of the specification language are tentative, but the code presented does satisfy the important property that it can be executed directly in C++ using macros and a template library to implement the linguistic constructs. This property allows Phase 1 to assure Pochoir-compliance, but it does limit the language's expressiblity. In later sections of this proposal, we suggest variations on this basic language to handle complications that arise in practice for real applications.

We shall also explore other linguistic and compilation strategies. For example, rather than having the stencil shape declared explicitly, we could infer it from the kernel function. Since this enhancement would entail significant compiler complexity and is not doable in general because array accesses can be obscured through function calls and the like, we plan to investigate this enhancement after other functionality has been implemented.

Even more ambitious would be to embed the Pochoir functionality directly in a C++ compiler, which potentially would yield a much cleaner language. Our thoughts along that line run as follows. We would create a special stencil class that the programmer inherits from in order to define his or her own stencil computation. The programmer would then override methods to the class to implement the kernel function, the boundary function, and other parts of the specification. An advantage of this approach is that compilers that do not recognize the special class can use a library definition for the class to obtain the same functionality, albeit with less performance. We plan to work with our collaborator C.K. Luk of Intel and software

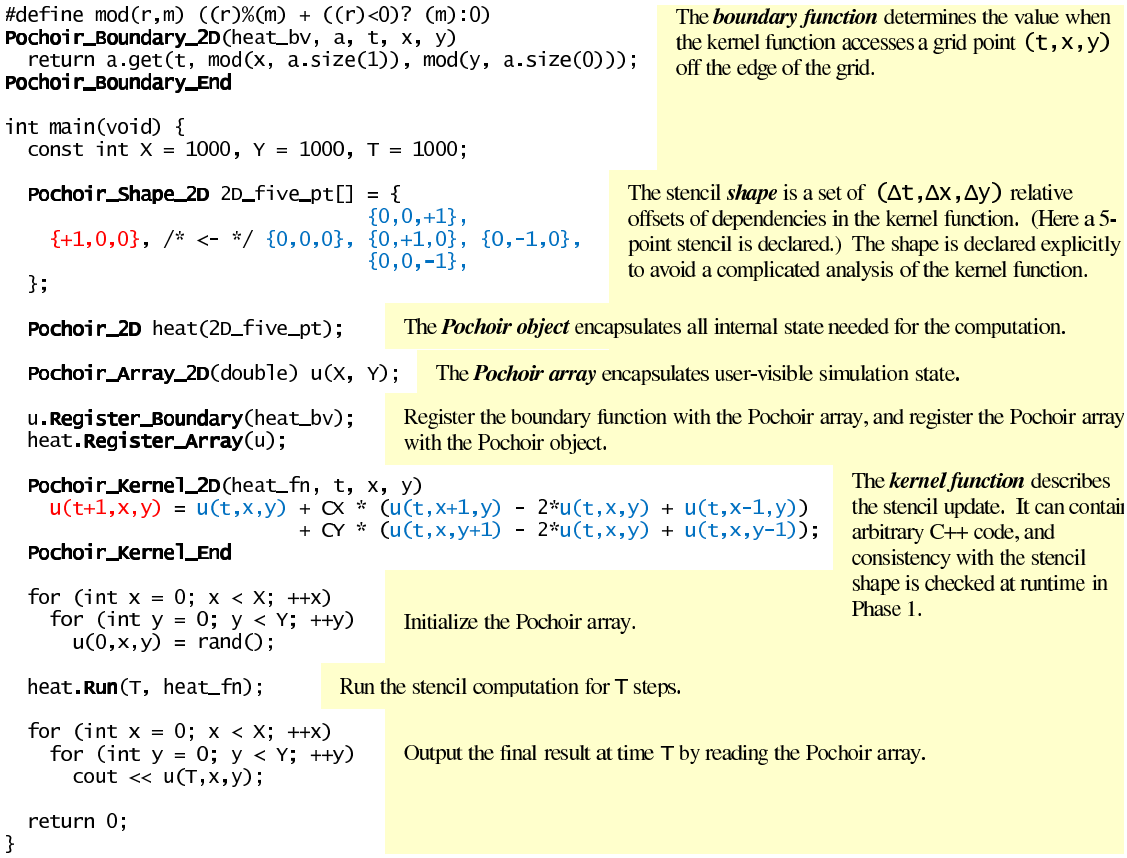

Figure 4: A possible Pochoir specification of a stencil computation that solves a periodic 2D heat equation. Pochoir keywords are boldfaced.

developers from Intel's compiler team to investigate how Pochoir functionality might be directly embedded into the GCC and Intel compilers.

4 Cache-oblivious multithreaded stencil algorithms

This section describes the cache-oblivious multithreaded algorithm we intend to use as the basis of Pochoir-generated stencil computations. We also describe several optimizations of this algorithm that we will investigate, which we suspect will contribute to high performance, including effective task decomposition, coarsening of recursion, code cloning, autotuning, and the use of cache-oblivious layouts for arrays.

The principal focus of the Pochoir compilation technology will be multicore processors. They are already in the hands of computational scientists (and ordinary people) everywhere, and they are the building block for heterogeneous and scalable systems. For example, the nodes of distributed-memory clusters are multicore chips, and GPU's are attached to multicore processors. To obtain good stencil performance on these systems requires efficient use of multicore processors. Multicore processors are characterized both by parallelism and memory hierarchy, and thus efficient codes for them must be both multithreaded and cache friendly. Pochoir will employ a provably efficient cache-oblivious multithreaded algorithm similar to the one shown in Figure 2. This divide-and-conquer code was inspired by the ***trapezoidal decomposition*** method of Frigo and Strumpen [27, 28], but it incorporates ***hyperspace cuts*** [83] to enhance parallelism.

This algorithm divides the space-time iteration space of the stencil computation into ***zoids***, which are multi-dimensional analogs of a trapezoid. At each recursive step, the current zoid is subdivided into three subzoids depending on the shape of the zoid.

We have proved [83] that the hyperspace-cutting strategy delivers asymptotically optimal cache-efficient performance, but the analysis of parallelism has been more problematic. For a well-formed $(d+1)$-dimensional space-time grid with side w operated on by a stencil with constant slopes, we can prove that the parallelism is $\Theta(w^{d-\lg(d+2)+1}/d^2)$. This bound theoretically beats the original Frigo-Strumpen method by a factor that is exponential in d. We plan to investigate how to extend this bound beyond well-formed grids.

In addition, our experience with the prototype compiler indicates that the practical advantage of hyperspace cutting over the Frigo-Strumpen algorithm is significant, but small, because d is effectively constant (2 or 3 for most applications). We will explore how additional parallelism can be elicited in practice using better task-decomposition strategies. One idea is to divide a zoid by cutting into the smaller of the two faces normal to time, rather than the face normal to time having the smaller time index as the algorithms in the literature propose. Another is to treat the dependency structure as task graph, and use a tool such as the Nabbit library [2] to execute it. Both of these approaches would generate more parallelism, but the Nabbit approach also has the potential for more asymptotic parallelism. We propose to explore the theoretical basis of stencil algorithms extensively, as well as to study the practical effectiveness of different strategies.

Coarsening is a fundamental optimization for recursive programs. The idea of coarsening is to terminate the recursion at some point above the leaves of the recursion tree and revert to a highly optimized base case that avoids function-call overhead. Usually the base case is implemented as a simple nest of loops, because this strategy is generally good for prefetching and the processor pipeline. Traversing in the recursive order, however, is generally good for both caching and register assignment by the compiler. We plan to investigate how base cases can be generated that use good orders for traversal of base cases.

Code cloning — the automatic generation of specialized versions of the stencil kernels — will be a central technique in the efficient implementation of both boundary conditions and other forms of inhomogeneity, as described in the subsequent sections.

The code that the Pochoir compiler will generate will contain tuning parameters that the compiler must select to maximize performance. We have interfaced the Pochoir prototype to the Intel Software Autotuning Tool (ISAT), which can adjust these parameters using an exhaustive search. Recently, ***heuristic autotuners*** [41] have been developed which provide good tuning of parameters using heuristic search instead of exhaustive search, and we plan to investigate how these ideas can be incorporated into Pochoir.

The Pochoir prototype stores the spatial grid in row-major order, which is consistent with how C++ stores them. Since all accesses to the spatial grid are mediated by the Pochoir object, however, the layout of arrays could be different. In particular, we plan to investigate storing arrays using Z-Morton orders [65] and other divide-and-conquer cache-oblivious layouts.

5 Boundary conditions

Real applications require a diverse set of possible ***boundary conditions***, which determine how the stencil should update edge values that depend on points that "fall off" the grid. In the context of cache-optimized time-stepping, where different spatial regions are not updated in synchrony, an *algorithmic challenge* is posed by the prevalence of ***nonlocal*** boundary conditions (such as periodic boundary conditions) in which boundary values at one point depend on interior values at other points far away. In such a case, the cache-oblivious algorithm must be modified to ensure that the necessary synchronization is maintained. The common technique of ***ghost cells*** that form a ***halo*** around the periphery of the grid, which are updated with

the boundary conditions separately from the stencil computations, no longer suffices because of the finer-grained synchronization that is required compared to naive loop code. Second, a *performance challenge* is posed by the necessity of ensuring that handling of boundary conditions does not dominate the runtime, and in particular one must avoid both repeated computation of boundary values and minimize the overhead of testing whether a given point requires boundary values. Third, a *linguistic challenge* is posed by the question of how to concisely express general boundary conditions in such a way that the most important cases can still be efficiently analyzed and implemented by the compiler.

A simple case is that of Dirichlet boundary conditions, where the array is set to predetermined values at the edges, independent of the interior values. Nonlocal dependencies can be introduced, on the other hand, by periodic boundary conditions, which specify a type of symmetry in the problem. Indeed, there are many other types of symmetry-based boundary conditions [74]. For example, if a problem has a mirror symmetry, then the domain can be halved by imposing mirror conditions along the mid-plane. Neumann boundary conditions, in which the boundary-normal slope of the function is specified, can be thought of as a generalization of mirror boundaries (zero slope). Other examples include four-fold rotational symmetry, where one simulates a quarter of the computational domain, and "Bloch-periodicity," periodicity with a complex phase factor, which arises for waves in periodic media [6,46]. Besides symmetry, other generalizations include the absorbing boundary conditions commonly used for wave equations [82], for which the boundary values are determined by some extrapolation of nearby interior values.

Pochoir's compiler support will afford much more flexibility in how one can express and implement boundary conditions than is practical in traditional loop-based code. Pochoir will employ a ***two-clone*** strategy for executing its trapezoidal-decomposition algorithm. A fast ***interior*** clone can be used for zoids that contain only interior points, thereby avoiding any boundary processing at all. A slower ***boundary*** clone can be used for zoids that require extra processing for off-grid values. Since the work by the interior clones asymptotically dominates the work by boundary clones, the extra overhead in the boundary clones becomes insignificant for large grids. To enforce synchronization for nonlocal boundary conditions, the compiler can modify the way that the top level of the trapezoidal decomposition is performed. Our prototype compiler demonstrates that this approach is effective for Dirichlet and periodic boundary conditions on simple stencils. We plan to investigate how more general boundary conditions can be handled.

We propose to explore how the two-clone approach can be extended to handle irregular shapes, as opposed to simple regular grids. As we envisage it, the user will provide an arbitrary predicate coded in C++ that indicates whether a given grid point lies inside or outside the shape. The compiled code will perform a precomputation that executes a divide-and-conquer algorithm on a bounding box of the irregular shape, recursively dividing the box into subboxes along the longest dimension. This precomputation will label each subbox with one of three labels: ***interior*** (all points within the box lie within the irregular shape), ***boundary*** (at least one point is on the boundary), and ***exterior*** (all points are outside the shape). It will then save a ***plan*** [24] consisting of this divide-and-conquer tree, but pruning the subtrees underneath each interior and exterior node, since those can be determined implicitly at runtime. The resulting plan representation can be shown to be at most logarithmically larger than the perimeter of the irregular shape, which for irregular shapes occurring in many applications will be asymptotically less than the storage for the grid points. During the actual execution, the plan can be used to determine which clone to employ during the trapezoidal decomposition. This strategy allows the programmer good flexibility in specifying the irregular shape — an arbitrary predicate — while also permitting an efficient algorithm in practice for executing the stencil computation. We also plan to explore provably efficient algorithms and strategies for handling irregularities in space-time, as opposed to just space.

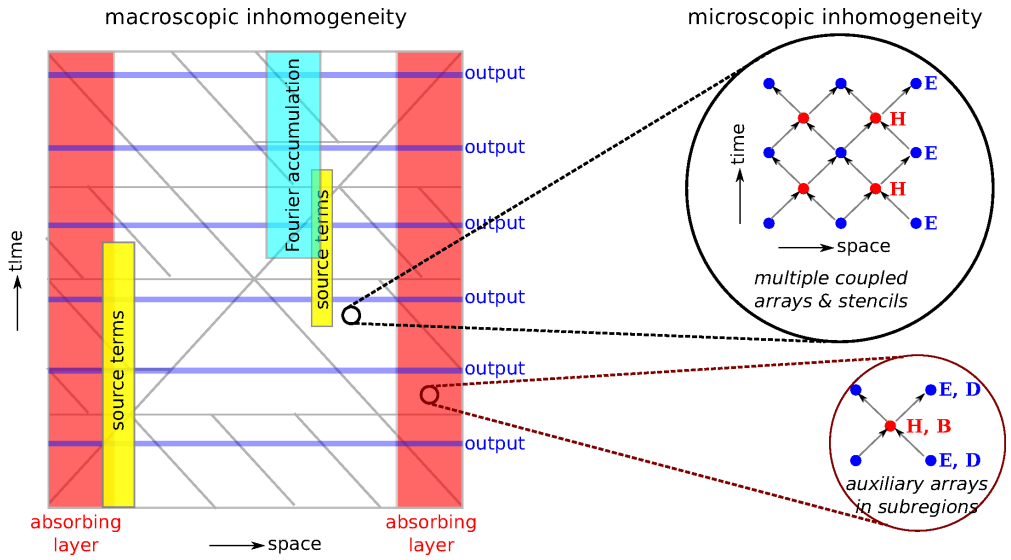

Figure 5: Schematic of inhomogeneities present in a typical application: wave propagation in electromagnetism. *Left:* At the macroscopic level, the stencil kernel is modified in various (possibly overlapping) subregions to output results, include source terms, implement absorbing boundary layers, or perform other analyses such as accumulating Fourier transforms. Gray lines indicate a cache-oblivious decomposition of a 1+1-dimensional space-time [27]. *Right:* At the "microscopic" level, each time step actually involves a sequence of interleaved stencils to update coupled solutions (e.g., electric and magnetic fields **E** and **H**). There may be different numbers of unknowns and stencils in subregions implementing additional physical processes (e.g., auxiliary arrays are typically needed in absorbers [82]).

6 Inhomogeneous stencil computations

A major complication in real-world stencil computations is the presence of inhomogeneities dictated by the underlying application. We have identified two kinds of inhomogeneities. A ***macroscopic inhomogeneity*** occurs when different computations are performed in different regions of space-time, often due to different materials or data analyses in those regions. A ***microscopic inhomogeneity*** occurs when each grid point of space-time comprises several arrays that are interlinked by different stencil operations, often because multiple physical unknowns are coupled. Handling these application inhomogeneities in the context of a parallel program that uses the memory hierarchy efficiently represents a daunting challenge to even the most capable stencil programmer. We propose to address this challenge by extending the basic Pochoir specification language discussed in Section 3 to allow the programmer to flexibly specify these inhomogeneities, perhaps even more simply than with existing loop-based code. The Pochoir compiler will generate cache-efficient multithreaded code automatically.

To illustrate how inhomogeneities can be handled by Pochoir, let us focus on the example of simulating Maxwell's equations of electrodynamics using the popular finite-difference time-domain (FDTD) method [82]. Common macroscopic inhomogeneities that arise in such a stencil application are shown on the left in Figure 5. The user may wish to output the electromagnetic fields to disk at selected times (e.g., every 100 time steps). Current sources, which generate the electromagnetic fields by modifying the stencil equations to include additional "source" terms, are typically localized to small regions of space-time. Adjacent to the boundaries of the spatial domain are often "perfectly matched" absorbing layers (PML's), artificial regions in which a more complicated stencil is used that absorbs outgoing waves so as to eliminate unwanted reflections from the boundaries [82]. To obtain the entire spectrum of a response to a

short-pulse excitation (e.g., a reflection spectrum), one may also wish to accumulate Fourier transforms — $\tilde{f}(x,\omega) \sim \sum_t f(x,t)e^{i\omega t}\Delta t$ for a field $f(x,t)$ — in a small region of space for some set ω of frequencies [74].

The algorithmic complexity greatly increases when this inhomogeneity is combined with a cache-oblivious decomposition of space-time into trapezoidal regions (gray lines in Figure 5). For example, whereas in a loop-based code the output of fields at some time T is a simple matter of inserting a call to an output routine between time-steps T and $T+1$, in a cache-oblivious implementation the fields at different spatial points may never be synchronized at time T. Instead, the stencil kernel must be modified to include an additional computation: whenever a point (x,T) is traversed, the fields at that point must be exported, for example, by saving it to some auxiliary array, which is written to disk after all x have been reached for the time T. A naive implementation would therefore have a conditional check for whether $t = T$ in the inner loop. Worse, the existence of multiple overlapping inhomogeneities means that the inner loop might naively require many such ***guard conditions***. The cost of mispredicted branches from a even few such conditionals has been observed [74] to slow down a loop-based FDTD program significantly. Instead, an efficient cache-oblivious implementation should hoist the guard conditions out of the inner loops: it should check whether an entire trapezoid of space-time intersects a given subregion (e.g., the output time T), so that the costs of the guard conditions are incurred only an asymptotically negligible number of times. This kind of code transformation is difficult for programmers to do by hand. We propose to provide compiler support in Pochoir that uses code cloning to generate specialized kernels for each possible homogeneous region.

In addition to the algorithmic questions, there is also the linguistic question: how can the user conveniently specify complicated irregular kernels, possibly including overlapping irregularities, as in the example from Figure 5. In our prospective design, the user denotes either a general ***guard*** condition or a simple ***geometric primitive*** (e.g., box, circle, convex polyhedron, etc.), which describes the shape and scope of subregion in space-time for which a certain set of corresponding kernel functions should be applied. Based on the user's specification, the Pochoir system will select an efficient algorithm to determine the scope of all subregions at runtime by a geometric intersection method (such as [22]) or at compile-time by symbolic precomputation. After determination of the scope of each subregion, for nonoverlapped cases, every point in each subregion can be computed without any conditional evaluation. Conditionals must be evaluated in regions where different inhomogeneities overlap, but the overhead can be made small as long as these overlap regions are asymptotically small (e.g., lower dimensional) than the homogeneous regions. Ideally, user knowledge about which types of kernels occupy large subregions could be expressed in the form of hints that could be used to prune the combinatorial explosion of possible code clones in the case of many overlapping inhomogeneities.

The example of electromagnetism also illustrates ***microscopic*** inhomogeneity: each time-step is divided into two coupled stencils: first the electric field $\mathbf{E}$ is updated from a stencil ($\approx \nabla \times$) of the magnetic field $\mathbf{H}$, and then $\mathbf{H}$ is updated from a stencil of $\mathbf{E}$. This is sometimes referred to as a "staggered-grid" or "leap-frog" discretization, and more specifically as the "Yee grid" in electromagnetism [82]. In certain subregions or for certain materials, the time step may be further subdivided into steps that update auxiliary polarization fields from $\vec{E}$ to implement anisotropy or frequency-dependent materials [74, 82]. (The same is true for many other physical problems in which multiple physical quantities interact over time. The general idea of splitting each time step into several interacting substeps is sometimes known as ***operator splitting*** [60].) Because the time-stepping consists of multiple interleaved stencils, substantial modifications are required to the basic cache-oblivious trapezoidal-decomposition algorithm. One approach might be to alternate between two or more kernels depending on the value of the time coordinate t: if t is even, update $\mathbf{E}(x,t)$ from $\mathbf{H}$ using kernel k_E, and if t is odd update $\mathbf{H}(x,t)$ from $\mathbf{E}$ using kernel k_H. Some form of loop unrolling would be required, however, to avoid the overhead of executing the conditional in the inner loop.

Rather than asking the compiler to divine the dependency information needed to generate the space-time cuts of the trapezoidal decomposition, we shall investigate ways by which the user can specify this kind of microscopic inhomogeneity linguistically. For example, we might consider generalizing the concept of stencil algorithms to include arbitrarily ***tiled kernels***, which specify locally which subkernel to execute. The electromagnetic simulation, for example, would specify a single tiled kernel $k = \binom{k_H}{k_E}$ that is executed at every time t, with the subkernels executing at "fractional" time steps. Pochoir could then analyze the tiled kernel as a unit. From the dependencies of k_H and k_E, Pochoir would compute the dependencies of k and hence the nature of the allowed space-time cuts. For subregions in which additional physical quantities are updated, any number of components could be added to k in those regions while still conceptually allowing k to apply at every integer time t. Consequently, subregions with and without additional variables will time step **E** and **H** consistently at integer times. More generally, one could allow kernels that have subcomponents tiled in both space and time to express other sorts of interleaved dependencies, such as checkerboards.

7 Complex dependencies

Basic stencil computations update the value at a grid point at a given time t as a function of neighboring grid points at recent times before t. Some real-world applications exhibit more complicated dependency structures, however, including "push" dependencies, "horizontal" dependencies, and "dynamic" dependencies. We propose to investigate how general complex dependencies can be incorporated into Pochoir.

One phase of the Lattice Boltzmann Method (LBM) inverts the normal structure of a basic stencil. Instead of each grid point being updated at time t as a function of grid points at times $t-1$ and before — ***pull*** dependencies — this phase of LBM uses the value of the grid point at time t to update neighboring points at times $t+1$ and after — ***push*** dependencies. Although the programmer can refactor the computation to convert push dependencies into pull dependencies, this process is onerous and error prone. We plan to investigate how the expressiveness of Pochoir can be enhanced to deal with push stencils directly.

Another generalization is ***horizontal*** dependencies. Many dynamic-programming problems, including longest common subsequence [15], pairwise sequence alignment [34], and adaptive statistical design [70], exhibit a structure in which each grid point (x,y) depends on its nearest neighbors in two dimensions, e.g., on $(x-1,y)$ and $(x,y-1)$. Neither x nor y provides a suitable time dimension to perform a basic stencil computation, because grid points at time t would depend on other points also at time t. Nevertheless, using $t = x+y$ as a "virtual" time dimension allows the computation to be viewed as a stencil, albeit by rotating axes by 45 degrees and handling a diamond-shaped boundary. In principle, however, this kind of computation can be performed cache-efficiently in recursive fashion even without a trapezoidal decomposition. We propose to enable Pochoir to handle this kind of dependency. The basic idea is to analyze the set of dependencies as defined by the shape, which resemble distance vectors [33] from the domain of vectorizing compilers. From the shape, we can determine a vector direction for "virtual" time (assuming the induced dag is acyclic) by taking the maximum of each coordinate, which allows values at a given virtual time step to be updated based on earlier virtual time steps. The slopes in each dimension of the sides of the zoids used by the divide-and-conquer can then be determined from the minimum and maximum slope of the dependencies with respect to the virtual-time vector, from which the cache-oblivious divide-and-conquer strategy itself ensues. As it turns out, however, several different vector directions may serve for virtual time. We propose to explore theoretically how to choose the best such vector direction to minimize cache misses.

As a third generalization, some dynamic-programming problems such as the ***subset sum*** and the ***integer knapsack*** [15] can be expressed as stencil computations in which the shape of the stencil changes with each time step. Consider the subset sum as an example. Given a list $X = \langle x_1, x_2, \ldots, x_N \rangle$ of positive integers and

an integer M, this problem asks if the elements in any subset of X sums to M. Define the function $S(n,m)$ to be 1 if some subset of $\{x_1,x_2,\ldots,x_n\}$ sums to m, and 0 otherwise. Our goal is to compute $S(N,M)$, which for $n = 0,1,\ldots,N$ and $m = 0,1,\ldots,M$ can be recursively computed as follows:

$$S(n,m) = \begin{cases} 1 & \text{if } m = 0 \text{ ,} \\ 0 & \text{if } n = 0 \text{ and } m > 0 \text{ ,} \\ \max\{S(n-1,m), S(n-1,m-x_n)\} & \text{otherwise .} \end{cases} \tag{1}$$

Observe that $S(n,m)$ depends on $S(n-1,m-x_n)$, where the spatial distance between these two points can be as large as M. Since the trapezoidal decomposition algorithm is based on a single slope, one can solve such stencils through the use of a single stencil based on the largest value in X, call it $x_{\max}$. But for large values of $x_{\max}$, this approach leads to zoids that are thin in the time dimension, and so cannot be solved in a cache-efficient fashion. A similar problem arises if a different slope is used for each time step. We plan to design Pochoir so that such time-varying stencils can be expressed precisely, extending the trapezoidal decomposition algorithm to generate zoids with good aspect ratios.

8 Beyond multicore

Although the principal focus of this research project targets standard multicore systems, we propose to explore how other architecture platforms and architectural features can be targeted, including distributed-memory clusters, graphical-processing units (GPU's), FPGA systems, and future exascale computers. The programmer will then be able to write a single Pochoir source program and rely on the Pochoir compiler to target multiple platforms. In this section, we briefly outline our research ideas for these platforms.

Clusters. Distributed-memory clusters require that programs operate on a collection processing nodes, each of which has an independent address space. Instead of communicating through shared memory, clusters employ message-passing using, for example, MPI [16,63,75] or Erlang [4,5]. Since the nodes of today's clusters are multicore processors, obtaining best-possible performance is considerably more complicated, since it involves parallelism at two levels: intranode (shared memory) and internode (distributed memory), a level of complexity that most programmers are not willing or able to address.

We propose to investigate a strategy by which the multicore Pochoir implementation can be employed for intranode parallelism, while extending it with message passing for internode communication. Conceptually, the message-passing Pochoir could generate multicore Pochoir code with embedded MPI or Erlang library calls, which would then be run through the multicore Pochoir compiler. The central research question is how Pochoir can minimize the overhead of internode data movement while retaining intranode cache-efficiency.

We propose to investigate novel distributed-memory approaches, such as the one depicted schematically in Figure 6. Whereas in most loop-based code, the data distribution is fixed, in this scheme the data distribution changes over time so as to maximize locality and minimize communication. If we start with a given data distribution of space across the nodes (bottom edge of Figure 6) and each node operates for as long as possible without communication, the result is that each node obtains the upper edge of the bottom-most triangles in Figure 6. The key idea is that at this point, we can shift the data distribution: each node transfers half its data (one upper edge of its triangle) to the neighboring node, so that each node now possesses the lower edges of a white diamond that it can proceed to operate on without communication. The process repeats when the upper edges of the white diamonds are reached. In comparison, a typical loop-based program uses a fixed decomposition of space, exchanging the boundary values on every time step [57].

Although this strategy communicates the same total amount of data as the shifting algorithm, it uses many small messages with a much larger total latency cost, rather than a few large messages. The research challenge is to validate this algorithm in practice and generalize it to arbitrary kernels and dimensions without adversely impacting intranode performance gained from effective parallelization and caching or the simplicity of the linguistic specification.

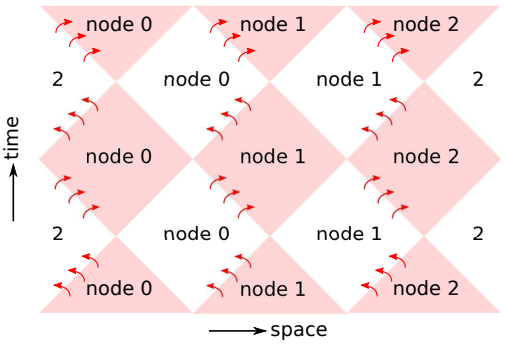

Figure 6: Schematic of a possible algorithm to minimize communication for stencil computation on a distributed-memory architecture, here with $1+1$-dimensional periodic space-time and three compute nodes. The data distribution changes with time: each node propagates for as long as possible with no communication (diamonds), then sends one upper edge of its diamond to the adjacent node (red arrows), and then continues propagating locally over a different region of space (white diamonds versus shaded diamonds).

GPU's and vector units. While the emergence of GPGPU languages like CUDA [69] and OpenCL [50] have substantially eased the job of GPU programming, writing high-performance GPU codes still requires significant effort. Moreover, the vector units in most multicore processors are underutilized. We believe Pochoir can produce specialized recursion base cases to exploit these architectural features.

Exploiting GPU's and vector units poses several difficult research challenges. First, to keep the Pochoir specification simple and universal across the range of targeted platforms, it will be necessary to produce vectorized code from a scalar description of the kernel. While this may not be hard for simple stencils, such as the heat equation, more complex kernels, such as LBM, may resist efficient vectorization. Second, vectorized loops consume significantly more memory bandwidth than divide-and-conquer recursion. Thus, although the code may go faster using vector technologies on one processing core, memory bottlenecks may cause it to run slower on multiple cores. An intriguing possibility is to adaptively enable vectorization depending on the number of cores.

FPGA systems. Field-Programmable Gate Arrays (FPGA's), which can provide a massive level of hardware parallelism with fast on-chip memories, seem eminently suitable for stencil computations. Indeed, companies like Maxeler [59] offer FPGA stencil solvers. The main research issue is that FPGA's are much more difficult to program efficiently than pure-software approaches. As with GPU's and vector units, a key challenge will be whether the Pochoir language can be kept simple while fully exploiting the hardware.

Exascale. For truly large machines, the issues of checkpointing and fault tolerance arise, since the mean time to failure of these machines can be much less than the duration of the stencil code. We propose to investigate how application-level automatic checkpointing and fault tolerance can be incorporated into Pochoir. The key issue that Pochoir will face is that its cache-oblivious algorithm moves though time nonuniformly, unlike the standard looping algorithm. Thus, resuming a computation after a checkpoint may require additional information that would not be required for less-efficient looping code.

9 Benchmarking

We propose to assemble a benchmark suite of stencil computations drawn from a wide range of application areas in order to track improvements during the engineering of Pochoir. The benchmark suite will include representative applications from a variety of disciplines including physics (e.g., heat equation [23], wave equation [64], Maxwell's equation [82], Lattice Boltzmann Method [61]), computational biology (e.g., RNA secondary structure prediction [3,12], pairwise sequence alignment [34]), computational finance (e.g., American put stock option pricing [47]), and mechanical engineering (e.g., compressible Euler flow [10]),

among others.

The benchmark suite will be organized hierarchically [30, 31] with at least the following three levels: (1) ***optimizations***, (2) ***kernels***, and (3) ***applications***. The performance of Pochoir at a given level will provide insights into its performance at the next higher level. The code snippets under optimizations will be designed to test the individual optimizations implemented in Pochoir. Each kernel will be a code fragment implementing only a single stencil kernel (e.g., regular/irregular, diamond-shaped, staggered) with (possibly) all optimizations enabled. Each application will be a real-world program that will include at least one stencil kernel along with other nonstencil computations. Although the lowest level of the hierarchy — optimizations — is intended solely for benchmarking Pochoir, the remaining two levels can be used for all stencil compilers.

A benchmarking system will also be developed that will allow automatic execution of the benchmark suite to compare the performance as the machine architecture, operating environment, and the host compiler vary. The system will be designed to provide a fair comparison among stencil algorithms, for example, by selectively generating both cache-oblivious and optimized traditional loop-based code from the same front-end description. The benchmarking system will be implemented in a scripting language. It will test correctness using both randomized inputs and explicit corner cases for stress-testing. It will export benchmark data and automate the build and execution for different environments (e.g., distributed-memory versus multicore). In addition to coarse-grained execution times, fine-grained performance data, such as cache-miss counts at various levels and data-communication costs (for cluster and GPU environments), will be collected with the aid of performance counters and simulation tools [67].

10 Intellectual merit

Traditional loop-based implementations of stencil computations may suffer by well over an order of magnitude in performance compared to the optimal algorithms when run on modern multicore architectures. Exploiting the parallel processing capability of multicore architectures and their steep cache hierarchies, however, involves complicated programming. By attaining high performance without sacrificing programming simplicity, the Pochoir project will broadly advance knowledge across diverse fields. Pochoir will enable a dramatic increase in both computational and programmer efficiency. Even within computer science, the generalization of cache-oblivious algorithms to irregular stencils and the other innovations of this proposal, as well as their crystallization into a computer language, represents a significant advance in the understanding of stencil problems. Previous work on cache-optimization and load-balancing of stencil algorithms, aside from straightforward loop-based techniques, has been dominated by simplified examples involving purely homogeneous stencils with conventional dependencies and simple boundary conditions. This proposal represents an original as well as a practical direction of research, with a creative new approach in the form of two-phase compiler technology.

The deep and interdisciplinary technical expertise required for this effort is amply present within the proposing team. Their accomplishments include the invention of cache-oblivious algorithms [25], the development and deployment of innovative parallel languages and compilers (Cilk [81]), and world-leading computational software (Cilk and FFTW [24]). They also have great expertise in stencil applications such as computational biology [12] and electromagnetism (including deployment of production open-source stencil-based simulation code [74]). The proposers have computational facilities ranging from multicore clusters to supercomputers. The team members have a long experience of collaboration across disciplines and institutions, and in particular, they have already enjoyed a productive collegial relationship with one another.

11 Broad impact

The goal of this research is to make cache-efficient parallel stencil algorithms easy to use by the majority of users in the many stencil application areas, not just by expert computer scientists, while supporting the idiosyncrasies that arise in real-world problems. In addition to advancing discovery and understanding, it will broaden access to stencil simulations by bringing them to a wider audience of scientists, students, and educators, who could not previously afford the investment required to implement and experiment with these techniques on modern multicore processors.

As we have done in the past with our research projects, including Cilk [81] and FFTW [24], we will make our software freely available in open-source form. We plan to work with computer and software vendors, most notably Intel thanks to the explicit participation of C.K. Luk (unfunded) in this project, to make them aware of our Pochoir tool and the advantages of our new approach to common problems that it represents. We shall continue interacting with computational scientists at MIT (e.g., in physics, chemical engineering, biology, climatology), sharing infrastructure for high-performance computing, as we have for almost 25 years. We shall also interact with computational scientists at New York Center for Computational Sciences (NYCCS) which is jointly managed by Stony Brook University and Brookhaven National Laboratory, and share the high-performance computing infrastructure (e.g., IBM Blue Gene) available there.

The research team comprises 10 graduate students, including four women. The project will engage undergraduate students as part of MIT's Undergraduate Research Opportunities Program. Materials will additionally be made freely available to the public via the MIT OpenCourseWare initiative (`http://ocw.mit.edu`). The proposing faculty have a history of involvement in outreach to underrepresented minorities, including serving as faculty supervisor for the Undergraduate Society for Women in Mathematics at MIT, serving as director of the Undergraduate Practices Opportunity Program, and by supervising numerous graduate and undergraduate researchers from underrepresented minorities (six in the past two years).

12 Results under prior NSF support

Prof. Johnson's most recent prior NSF grant, "Multimaterial Multifunctional Nanostructured Fibers," is an ongoing interdisciplinary research group funded through the MRSEC Program of NSF under award number DMR-0819762. This six-year grant, which began September 1, 2008 and will run to August 31, 2014, focused on new classes of fiber-based technologies (optical devices, piezoelectric and photovoltaic sensors, and others); the total anticipated award is $212,305. Thus far, this award has provided funding in part for twelve published papers [8,11,18,37,52,56,58,71–74,79] of which Prof. Johnson is a coauthor, on topics such as the design of thermophotovoltaic devices, analysis of fluid-dynamical instabilities, computational methods in electromagnetism, and the fundamentals of electromagnetic waveguiding.

Prof. Leiserson's most recent NSF-funded project is NSF Grant CNS-1017058, "CSR: Small: Using Thread-Local Memory Mapping to Support Memory Abstractions for Dynamic Multithreading," a two-year grant for $500,000 which began on August 1, 2010 and will run to July 31, 2012. The objective of this project is to understand how operating-system support can enable linguistic memory abstractions for parallel programming. This award has provided funding in part for six published papers [7,53–55,83,84] on the topics of runtime systems for concurrency platforms, stencil computations, parallel algorithms, and cache-oblivious algorithms. In addition, the research support has allowed us to build a novel runtime system called Cilk-M which incorporates thread-local memory mapping and is a plug-in replacement for the runtime system provided by the Intel Cilk Plus compiler. The runtime system provides provably good scheduling, support for legacy binaries, and guaranteed space bounds.

13 Collaboration plan

The research project will be led by Professor Charles E. Leiserson of MIT CSAIL. The project will involve Professor Steven G. Johnson of MIT Mathematics Department and Professor Rezaul Chowdhury of Stony Brook University Department of Computer Science who is also affiliated with New York Center for Computational Sciences. Dr. Chi-Keung Luk of Intel Corporation, who will be funded by his company, will also collaborate. In addition, the project will involve Professor Yuan Tang of Fudan University Software School in Shanghai, China, who contributed to the Pochoir prototype while a Visiting Scientist in Professor Leiserson's group at MIT. Professor Tang will be funded independently and will visit MIT twice a year. This project will involve a postdoctoral researcher. The team will meet regularly using phone and video conferencing, as well as in person.

The project will be broken into six tasks:

1. ***Basic*** — Using the existing Pochoir prototype as a model, producing a new robust system that handles homogeneous stencil rules on rectangular shapes, simple boundary conditions, and simple dependency rules. Organizing the codebase to allow new algorithms and support for new platforms to be modularly incorporated. Investigating algorithms and data layouts.
2. ***Boundary*** — Augmenting the system to handle mirroring, phase factors, and other complex boundary conditions. Designing simple and intuitive linguistics that can be compiled to efficient codes. Benchmarking example codes.
3. ***Irregular*** — Augmenting the system to handle microscopic inhomogeneities, macroscopic inhomogeneities, and irregular shapes. Investigating algorithms and compilation strategies. Designing simple and intuitive linguistics that can be compiled to efficient codes. Benchmarking example codes.
4. ***Dependency*** — Augmenting the system to handle push, horizontal, and dynamic dependencies. Investigating algorithms and compilation strategies. Designing simple and intuitive linguistics that can be compiled to efficient codes. Benchmarking example codes.
5. ***Platform*** — Targeting the system to support platforms other than today's complex multicores, including simple multicores, GPU's, distributed-memory clusters, FPGA's, and future exascale computers. Investigating target-specific algorithms and data layouts. Benchmarking example codes.
6. ***Benchmark*** — Assembling a benchmark suite of stencil computations. Implementing a benchmarking system that can run benchmarks automatically to compare different architectures, algorithms, and compiler implementations.

As the research team is small and its members familiar with working with one another, management of tasks can largely be done informally. In addition to leading the overall research effort, Prof. Leiserson will lead the design effort for Tasks 1–4 and mentor the postdoctoral researcher. Prof. Leiserson and Prof. Tang (separately funded) will jointly lead the software-development effort for those tasks. Prof. Johnson will lead the fashioning of the application benchmark suite in Task 6. Prof. Chowdhury will lead the development of the benchmarking system in Task 6. All PI's and collaborators will work on Task 5 and otherwise contribute to all tasks, including helping to design the linguistics, developing the foundational algorithmic theory, and implementing the algorithms and benchmarks.

We plan to make a major release of Pochoir annually, as well as minor and beta releases during the year. The software will be made freely available on the World Wide Web under an open-source license.

References

[1] Core documentation of the COSMO-model. `http://cosmomodel.cscs.ch/content/model/documentation/core/`.

[2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing task graphs using work stealing. In *IPDPS*, pages 1–12. IEEE, April 2010.

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

[4] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[5] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[6] N. W. Ashcroft and N. D. Mermin. *Solid State Physics*. Holt Saunders, Philadelphia, 1976.

[7] Michael A. Bender, Bradley C. Kuszmaul, Shang-Hua Teng, and Kebin Wang. Optimal cache-oblivious mesh layouts. *Theory of Computing Systems*, pages 269–296, 2011.

[8] Peter Bermel, Michael Ghebrebrhan, Walker Chan, Yi Xiang Yeng, Mohammad Araghchini, Rafif Hamam, Christopher H. Marton, Klavs F. Jensen, Marin Soljačić, John D. Joannopoulos, Steven G. Johnson, and Ivan Celanovic. Design and global optimization of high-efficiency thermophotovoltaic systems. *Optics Express*, 18:A314–A334, 2010.

[9] R. Bleck, C. Rooth, Dingming Hu, and L. T. 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.

[10] T. Brandvik and G. Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[11] Jorge Bravo-Abad, Alejandro W. Rodriguez, John D. Joannopoulos, Peter T. Rakich, Steven G. Johnson, and Marin Soljačić. Efficient low-power terahertz generation via on-chip triply-resonant nonlinear frequency mixing. *Applied Physics Letters*, 96:101110, 2010.

[12] Rezaul A. Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *TCBB*, 7(3):495–510, July-September 2010.

[13] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel data-locality aware stencil computations on modern micro-architectures. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[14] Clang: A C language family frontend for LLVM. `http://clang.llvm.org`, 2011. Retrieved 9/19/2011.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[16] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*. Springer, 2011.

[17] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 4:1–4:12, Austin, TX, November 15–18 2008. (`http://portal.acm.org/citation.cfm?id=1413370.1413375`).

[18] D. S. Deng, J.-C. Nave, X. Liang, S. G. Johnson, and Y. Fink. Exploration of in-fiber nanostructures from capillary instability. *Optics Express*, 17:16273–16290, 2011.

[19] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[20] Hikmet Dursun, Ken-ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par*, pages 642–653, Delft, The Netherlands, August 25–28 2009.

[21] Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, Las Vegas, NV, July13–16 2009.

[22] David Eberly. Intersection of convex objects: The method of separating axes. `http://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf`, 2008. Retrieved 9/19/2011.

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

[24] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, Washington, 1998.

[25] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, New York, NY, October 17–19 1999.

[26] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

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

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

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

[30] Steven Gerding. The Extreme benchmark suite: measuring high-performance embedded systems. Master's thesis, Massachusetts Institute of Technology, 2005.

[31] V.S. Getov, A.J.G. Hey, R.W. Hockney, and I.C. Wolton. The Genesis benchmark suite: Current state and results. In *Proceedings of Workshop on Performance Evaluation of Parallel Systems-PEPS*, volume 93, pages 29–30.

[32] GCC, the GNU compiler collection. `http://gcc.gnu.org`, 2011. Retrieved 9/19/2011.

[33] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Conference on Programming Language Design and Implementation*, PLDI '91, pages 15–29. SIGPLAN, ACM, 1991.

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

[35] Stephen M. Griffies, Claus Böning, Frank O. Bryan, Eric P. Chassignet, Rüdiger Gerdes, Hiroyasu Hasumi, Anthony Hirst, Anne-Marie Treguier, and David Webb. Developments in ocean climate modelling. *Ocean Modelling*, 2:123–192, 2000.

[36] Edison Design Group. The EDG C++ front end. `http://www.edg.com`, 2011. Retrieved 9/19/2011.

[37] Rafif E. Hamam, Mihai Ibanescu, Steven G. Johnson, J. D. Joannopoulos, and Marin Soljačić. Broadband super-collimation in a hybrid photonic crystal structure. *Optics Express*, 17:8109–8118, 2009.

[38] M. Howison. Comparing GPU implementations of bilateral and anisotropic diffusion filters for 3D biomedical datasets. In *SIAM Conference on Imaging Science*, 2010.

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

[40] Robert Hundt. Loop recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*, 2011.

[41] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07)*, pages 1152–1157, 2007.

[42] 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`.

[43] Intel Corporation. Tera-scale Computing-A Parallel Path to the Future. `http://software.intel.com/en-us/articles/tera-scale-computing-a-parallel-path-to-the-future/`, 2011. Retrieved 9/19/2011.

[44] Intel Corporation. The Intel Many Integrated Core Architecture. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2011. Retrieved 9/19/2011.

[45] Balaji V. Iyer. Announcing the port of Intel(r) Cilk (TM) Plus into GCC. `http://gcc.gnu.org/ml/gcc/2011-08/msg00279.html`, August 2011. Retrieved 9/19/2011.

[46] John D. Joannopoulos, Steven G. Johnson, Joshua N. Winn, and Robert D. Meade. *Photonic Crystals: Molding the Flow of Light*. Princeton University Press, second edition, 2008.

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

[48] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC*, pages 51–60, San Jose, CA, 2006. (`http://doi.acm.org/10.1145/1178597.1178605`).

[49] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP*, pages 36–43, Chicago, IL, June 12 2005. (`http://doi.acm.org/10.1145/1111583.1111589`).

[50] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1*, June 2011. Retrieved 9/19/2011, (`http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`).

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

[52] André Kurs, John D. Joannopoulous, Marin Soljačić, and Steven G. Johnson. Abrupt coupling between strongly dissimilar waveguides with 100% transmission. *Optics Express*, 19:13714–13721, 2011.

[53] Edya Ladan-Mozes, I-Ting Angelina Lee, and Dmitry Vyukov. Location-based memory fences. In *SPAA*, pages 75–84. ACM, 2011.

[54] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT*, pages 411–420. ACM, 2010.

[55] Charles E. Leiserson, Liyun Li, Marc Moreno Maza, and Yuzhen Xie. Efficient evaluation of large polynomials. In *ICMS*, pages 342–353, 2010.

[56] X. Liang, D. S. Deng, J.-C. Nave, and Steven G. Johnson. Linear stability analysis of capillary instabilities for concentric cylindrical shells. *Journal of Fluid Mechanics*, 683:235–262, 2011.

[57] Calvin Lin and Larry Snyder. *Principles of Parallel Programming*. Addison Wesley, 2008.

[58] Po-Ru Loh, Ardavan F. Oskooi, Mihai Ibanescu, Maksim Skorobogatiy, and Steven G. Johnson. Fundamental relation between phase and group velocity, and application to the failure of perfectly matched layers in backward-wave structures. *Physical Review E*, 79:065601(R), 2009.

[59] Maxeler. The MaxGenFD Whitepaper. `http://www.maxeler.com/content/briefings/MaxelerWhitePaperMaxGenFD.pdf`.

[60] Robert I. McLachan and G. Reinout W. Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

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

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

[63] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009.

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

[65] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, March 1966.

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

[67] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, volume 42, New York, 2007. ACM.

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

[69] Nvidia. *CUDA C Programming Guide v 3.2*, October 2010.

[70] Robert Oehmke, Janis Hardwick, and Quentin F. Stout. Scalable algorithms for adaptive statistical designs. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000.

[71] Ardavan Oskooi and Steven G. Johnson. Distinguishing correct from incorrect PML proposals and a corrected unsplit PML for anisotropic, dispersive media. *Journal of Computational Physics*, 230:2369–2377, 2011.

[72] Ardavan F. Oskooi, J. D. Joannopoulos, and Steven G. Johnson. Zero–group-velocity modes in chalcogenide holey photonic-crystal fibers. *Optics Express*, 17:10082–10090, 2009.

[73] Ardavan F. Oskooi, Chris Kottke, and Steven G. Johnson. Accurate finite-difference time-domain simulation of anisotropic media by subpixel smoothing. *Optics Letters*, 34:2778–2780, 2009.

[74] Ardavan F. Oskooi, David Roundy, Mihai Ibanescu, Peter Bermel, J. D. Joannopoulos, and Steven G. Johnson. MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method. *Computer Physics Communications*, 181:687–702, 2010.

[75] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[76] Liu Peng, Richard Seymour, Ken-ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11, Rome, Italy, May 23–29 2009.

[77] H.H. Pennes. Analysis of tissue and arterial blood temperatures in the resting human forearm. *Journal of applied physiology*, 1(2):93, 1948.

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

[79] David Ramirez, Alejandro W. Rodriguez, Hila Hashemi, J.D. Joannopoulos, Marin Soljačić, and Steven G. Johnson. Degenerate four-wave mixing in triply-resonant Kerr cavities. *Physical Review A*, 83:033834, 2011.

[80] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, third edition, 2000.

[81] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.2 Reference Manual*, November 2005. (`http://supertech.csail.mit.edu/`).

[82] Allen Taflove and Susan C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech, Norwood, MA, 2000.

[83] Yuan Tang, Rezaul A. Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *SPAA*, San Jose, CA, USA, 2011.

[84] Yuan Tang, Rezaul A. Chowdhury, Chi-Keung Luk, and Charles E. Leiserson. Coding stencil computation using the Pochoir stencil-specification language. In *HotPar'11*, Berkeley, CA, USA, May 2011.

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