RC24035 (W0608-077) August 24, 2006
Computer Science

IBM Research Report

Software Engineering Aspects of Cache Oblivious Stencil Computations

Volker Strumpen, Matteo Frigo
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Software Engineering Aspects of Cache Oblivious Stencil Computations

Volker Strumpen and Matteo Frigo*
IBM Austin Research Laboratory
11501 Burnet Road, Austin, TX 78758

July 13, 2006

Abstract

We discuss the design of cache oblivious stencil computations for finite-difference methods in 1-dimensional space and time. We employ our cache oblivious spacetime traversal [1] to obtain several fully cache oblivious algorithms. We focus on the practical aspects of the software engineering chores that today's superscalar architectures impose on high performance programming.

Contents

1 Introduction **2**

2 The 1-Dimensional Stencil Problem **3**

3 Impact of Processor Architecture on Performance **5**

4 1D Stencils with Toggle Arrays **7**
4.1 Nonperiodic 1D Stencils with Toggle Arrays 7
4.2 Periodic 1D Stencils with Toggle Arrays 10

5 1D Stencils with Boundary Passing **13**
5.1 Nonperiodic 1D Stencils with Boundary Passing 16
5.2 Periodic 1D Stencils with Boundary Passing 17

6 Performance Analysis **21**

*This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

1 Introduction

This paper discusses the design of high-performance programs for cache-oblivious stencil computations with small kernels, i.e. kernels containing just a few floating-point operations. Today's microarchitectures require substantial programming efforts to obtain acceptable levels of processor utilization. Applications benefit from cache oblivious formulations if they are memory bound. This implies in particular that an application must not be processor bound. With adequate programming techniques, we can guarantee that even relatively small computational kernels achieve decent processor utilization, as a prerequisite for observing performance improvements due to the reduction of memory traffic by means of cache oblivious formulations. This report was motivated by an inquiry of David Hensinger and Chris Luchini of Sandia National Labs, and uses their Lax-Wendroff kernel as a running example [3].

Table 1 summarizes the performance measurements comparing iterative and cache oblivious implementations of different variants of the Lax-Wendroff stencil computation. The speedup is the ratio of the runtime of the iterative version and the runtime of the corresponding cache oblivious version. A more detailed performance analysis appears in Section 6 below.

processor	970				Power5			
problem	periodic		nonperiodic		periodic		nonperiodic	
storage scheme	toggle	pass	toggle	pass	toggle	pass	toggle	pass
iterative [*sec*]	9.39	9.30	9.28	9.28	9.07	8.89	8.98	8.87
cache oblivious [*sec*]	3.32	3.37	3.35	3.34	4.19	3.89	4.04	3.93
speedup	2.83	2.76	2.77	2.78	2.17	2.29	2.22	2.26

Table 1: Runtimes and speedups of cache oblivious Lax-Wendroff codes versus the iterative programs for $N = 10,000,000$ space points and $T = 100$ time steps.

The results in Table 1 were generated on an Apple Power Mac G5 with 2 GHz PowerPC 970 processors and an IBM Power5 system with 1.65 GHz processors. We experiment with two different problem classes, *periodic* initial-value problems and boundary-value, initial-value (*nonperiodic*) problems. For both classes, we implemented programs with two different storage schemes, toggle arrays and boundary passing, as discussed in Section 2. Table 1 shows that for each of the two processor architectures, the speedups are roughly the same for the four different programs, and amount to about 2.8 on the 970 and about 2.2 on the Power5 processor. Furthermore, the absolute runtimes across the different problem classes and storage schemes are approximately equal as well.

The cache oblivious versions were optimized in two ways: (1) algorithmic restructuring for cache oblivious memory use, and (2) code optimization of the kernel computation, such as loop unrolling. We shall discuss these optimizations in detail in the rest of the paper. In addition, for the periodic boundary-passing program, we analyze the contribution of each kind of optimization to the total speedup in Section 6.

We conclude that the boundary-passing storage scheme, which saves a factor of 2 in working set requirements over the toggle-array scheme, should be the preferred storage scheme for implementing periodic as well as nonperiodic 1D stencil computations. Furthermore, the optimized, cache oblivious versions achieve the highest performance in all four cases.

This report is structured as follows. In Section 2, we introduce the stencil problem under investigation. Section 3 describes several first-order effects of the processor architecture which

impact program performance. Then, we discuss the software engineering aspects to arrive at efficient cache oblivious programs for the periodic initial value problems and boundary-value, initial value problems. For both classes of problems, we present programs with two different storage schemes, a toggle-array version in Section 4 and a boundary-passing version in Section 5. Finally, we provide more detailed insights by means of a performance analysis in Section 6.

2 The 1-Dimensional Stencil Problem

In this section we discuss several ways to implement 1D stencil computations. We assume that the computation produces values $u(t, x)$ in a discrete space domain with uniform grid spacings $\Delta x = 1/(N-1)$ and a discrete time domain with uniform intervals $\Delta t = 1/T$. Hence, the spacetime grid forms a rectangle with N points in the space dimension and $T+1$ points in the time dimension. Given initial values $u(0, x)$ at time $t = 0$ for all x in the space domain, we are interested in computing $u(T, x)$ for $t = T$. The intermediate values $u(t, x)$ for $0 < t < T$ are typically ignored. Thus, there is no need to retain these values until the end of the computation. If the structure of the computation of $u(t, x)$ is independent of t and x, we call the computation a ***kernel computation***, and the structure of the dependencies a ***stencil***. In this paper, we are concerned with *3-point stencils* in 1D space of the form:

$$u(t+1, x) = f\big(u(t, x-1), u(t, x), u(t, x+1)\big)\,. \tag{1}$$

The stencil is a 3-point stencil, because value $u(t+1, x)$ at space point x and time step $t+1$ depends on three values at the previous time step t and space points $x-1$, x, and $x+1$. In case of a boundary-value problem, the stencil and the kernel computation at the boundaries $x = 0$ and $x = (N-1)\Delta x$ differ from Equation 1. If the problem is a periodic initial-value problem, the same kernel computation may be applied to each spacetime point, however.

The design of a program for a stencil computation involves the choice of the underlying data structure. In principle, we may store the data associated with each spacetime point in one of three ways:

1. One $N \times T$ array stores all spacetime points of the domain.

2. Two 1D arrays hold N space points each. Two arrays can store two time slices t and $t+1$. When advancing time, we can read values from array $t \bmod 2$ and write values into array $(t+1) \bmod 2$. Since the arrays are toggled between time slices, we call this organization with two arrays ***toggle arrays***.

3. One 1D array holds N space points, and a relatively small number of temporary variables passes values across space boundaries. The number of temporary variables depends on the organization of traversal of spacetime, and may be as small as one spacetime point up to T spacetime points. We call this organization with one array and temporary variables for passing points across space boundaries the ***boundary passing*** method.

The toggle array and boundary passing methods are successive refinements of the naive method that uses an $N \times T$ array. Not only do these refinements minimize the storage requirements, they also enable ***in-place*** computations by reusing spacetime points. The reuse of memory cells is a crucial programming technique for exploiting hierarchical memory architectures, and is often

```
double u[2][N];

void kernel(int t, int x)
{
   u[(t+1)%2][x] = f(u[t%2][x-1], u[t%2][x], u[t%2][x+1]);
}

void traverse_spacetime(void)
{
   int t, x;
   for (t = 0; t < T; t++) {
      leftboundary(t);
      for (x = 1; x < N-1; x++)
         kernel(t, x);
      rightboundary(t);
   }
}
```

Figure 1: Toggle-array program for boundary-value, initial-value problem with 3-point stencil.

referred to as *temporal locality* in the context of performance analysis of cache memories. In-place computations are a prerequisite for cache oblivious algorithms.

In the following, we introduce 1D stencil computations with a 3-point stencil by comparing programs with toggle arrays and boundary passing. We use an initial-value, boundary-value problem to illustrate the program structures. Figure 1 shows the toggle-array version and Figure 2 the boundary-passing version. Both programs use the same procedure `traverse_spacetime` to visit all space points within each time slice. The left and right boundary functions compute points $u(t, 0)$ and $u(t, N-1)$, respectively. The differences appear in the choice of the data structure for array `u` and the corresponding `kernel` procedures.

The toggle-array version in Figure 1 allocates two time slices of N space points by defining array `u[2][N]`. In contrast, the boundary-passing version in Figure 2 allocates only one array of N space points, `u[N]`, plus temporary variable `w`. The choice of the data structures determines the computations within function `kernel`, that computes $u(t+1, x)$ as a function f of values $u(t, x-1)$, $u(t, x)$, and $u(t, x+1)$. In the toggle-array version, we traverse the entire range of space points at time t before we can enter the next time step $t+1$. Function `traversal_spacetime` implements this traversal as a two-fold nested loop, where time is traversed in the outer loop. When writing values $u(t+1, x)$ into array `u`, we reuse the memory cells that stored the values $u(t, x)$. More succinctly, the toggle-array version stores the even time steps $t = 0, 2, 4, \ldots$ in `u[0][x]` and the odd time steps $t = 1, 3, 5, \ldots$ in `u[1][x]` for $x \in [1, \ldots, N-1)$.

The boundary-passing program in Figure 2 saves half of the working set compared to the toggle-array version, and reuses array `u` to store in `u[x]` the values of all time steps $t = 0, 1, 2, \ldots$. Variable `w` is used to pass the "left" value $u(t, x-1)$ to the right, to compute $u(t+1, x)$. This program requires that the space points be visited left-to-right in the inner loop of procedure `traverse_spacetime`. The temporary variable `w` is needed because the kernel computation of $u(t+1, x-1)$ overwrites array element `u[x-1]`, while the previously stored value $u(t, x-1)$ is

```
double u[N], w;

void kernel(int t, int x)
{
   double umid = u[x];
   double uleft = (x == 1) ? u[0] : w;
   u[x] = f(uleft, umid, u[x+1]);
   w = umid;
}

void traverse_spacetime(void)
{
   int t, x;
   for (t = 0; t < T; t++) {
      leftboundary(t);
      for (x = 1; x < N-1; x++)
         kernel(t, x);
      rightboundary(t);
   }
}
```

Figure 2: Boundary-passing program for boundary-value, initial-value problem with 3-point stencil.

needed for the kernel computation of the right neighbor $u(t+1, x)$.

The kernel computation under investigation is what we call the ***Lax-Wendroff kernel*** [3]:

$$f\big(u(t,x-1), u(t,x), u(t,x+1)\big) = \\ u(t,x) - c_0\big(u(t,x+1) - u(t,x-1)\big) + c_1\big(u(t,x+1) - 2u(t,x) + u(t,x-1)\big)\,, \quad (2)$$

with constant values c_0 and c_1. This kernel has 8 flops and its data dependencies form a 3-point stencil. With three floating point loads and one store, the ratio of computation to memory accesses is merely $8/4 = 2.0$. Programs with such small kernels are particularly difficult to optimize, because they leave very little room to amortize any overheads that arise when restructuring the simple loop into a cache friendly traversal.

In the remainder of this paper, we will develop cache oblivious versions for both program variants. Furthermore, we extend the problem domain from boundary-value, initial-value problems to periodic initial-value problems.

3 Impact of Processor Architecture on Performance

In this section, we discuss those features of modern processors that have the largest impact on the performance of our Lax-Wendroff programs. Some of these features may catch programmers by surprise; some have software remedies, others do not.

Pipeline depth Superscalar processors such as the IBM Power4, Power5 and 970 have two floating point units. Each unit is pipelined and has a latency of six clock cycles—i.e., the result can

be used six cycles after the operation is issued. In order to use both units at full capacity, a program must therefore make 12 independent floating-point operations available for execution at all times. Conversely, a simple dot-product such as

```
for (i = 0; i < N; ++i)
  s += a[i] * b[i];
```

runs at most at 1/12th of the peak performance of the machine.

A common technique for increasing instruction-level parallelism is *loop unrolling*, which is particularly useful for small kernels such as our Lax-Wendroff computation. By unrolling the Lax-Wendroff kernel manually, we are able to achieve about 80 % utilization of the Power5 floating-point units.

Recursion overhead Our cache oblivious algorithm for spacetime traversals is naturally formulated as a recursive function. The associated function calls introduce an overhead that is not present in the naive iterative code, which uses a two-fold nested loop.

For complicated stencil computations with a relatively large number of floating-point operations in the kernel computation, the recursion overhead is usually negligible. However, for the simple 3-point stencil in the Lax-Wendroff code, the overhead of the recursion is significant, and a high-performance implementation must amortize this overhead if it cannot avoid it. We discuss a methodology to coarsen the base case of the recursion in order to reduce the function call overhead.

Index computations For problems with periodic boundary conditions, our algorithm involves computations of array indices that require modular arithmetic. In general, modular arithmetic is several times slower than ordinary integer arithmetic.

While the overhead due to modular arithmetic is negligible for complicated stencils with large numbers of floating-point operations, it is significant for the Lax-Wendroff code in which the kernel consists of just 8 floating-point operations. Throughout in this paper, we show how to reduce the amount of modular arithmetic in index computations.

Prefetching Today's processors, including most members of IBM's PowerPC series, incorporate a hardware prefetcher for data, which detects accesses to consecutive memory locations at addresses x, $x+1$, $x+2$, ..., and speculatively starts fetching location $x+L$ into the cache, for some value L. The danger of prefetching is cache pollution.

Prefetching can be useful for 1D stencil codes such as the Lax-Wendroff code, because the access pattern of the code is detectable by the hardware. On the processors we experimented with, the prefetcher is active by default.

It should be noted that the prefetcher has minimal or no performance impact on our cache oblivious algorithms.

Data-dependent floating-point performance The performance of the Power4, Power5 and 970 floating-point units is data dependent. Specifically, the processor stalls when subtracting two floating-point numbers if the result is nonzero but much smaller than the two operands.

The impact of this feature on the Lax-Wendroff code can be substantial, because finite-difference methods are designed to subtract values, whose difference approaches zero as the iteration converges. For our cache oblivious Lax-Wendroff codes, we observe a 40% reduction in execution time when we initialize the input array to 0.0 instead of using realistic initial values.

For performance tuning, we suggest to apply the principle of exclusion, and initialize all values to zero to exclude the performance degradation due to the floating point unit while analyzing orthogonal effects like memory latencies.

We will refer to the features listed above during the remainder of this paper. More detailed information about effects like prefetching and performance degradation due to the floating point unit can be found in Section 6.

4 1D Stencils with Toggle Arrays

In this section, we introduce cache oblivious versions of the programs introduced in Section 2, and extend the problem domain from boundary-value, initial-value problems to periodic initial-value problems.

4.1 Nonperiodic 1D Stencils with Toggle Arrays

The program in Figure 1 of Section 2 implements the toggle-array version of a nonperiodic stencil computation in 1D space. Although this implementation reuses array `u`, its performance is limited by the access latencies of the level in the memory hierarchy that is large enough to store the entire array. We can circumvent this problem by using a different traversal order of the spacetime domain that respects the data dependencies of the 3-point stencil but maximizes the reuse of memory cells. We have developed a generic traversal routine for spacetime [1], which is independent of the data structures used and applies to arbitrary n-dimensional problems. This routine traverses spacetime such that the number of data movements between each level of a memory hierarchy is minimized. Furthermore, our traversal routine does not have any voodoo parameters to characterize the memory hierarchy. Figure 3 shows our cache oblivious spacetime traversal function for 1D space.

Procedure `walk1Dspace` cuts the spacetime domain recursively into trapezoid-shaped regions. Once it reaches the base case, that is a trapezoid with a height of one spacetime point, it applies function `kernel` to all spacetime points within the trapezoid. The traversal generated by `walk1Dspace` has good temporal locality because it splits the spacetime domain into smaller and smaller trapezoids until it reaches the base case. As soon as the trapezoid fits into the cache at some level of the memory hierarchy, the subproblems operate at the speed of this cache. However, the recursion itself is unaware of when the trapezoid fits into a cache. Because of this property, we say that `walk1Dspace` is cache oblivious: it operates without the aid of any fudge factors, and it exploits all levels of the memory hierarchy.

We present two implementations of the cache oblivious version of the program in Figure 1. We begin with a simple version that is relatively easy to comprehend. Then, we include the Lax-Wendroff kernel of Equation 2, and optimize this version to achieve high performance. Figure 4 is a straightforward modification of the program in Figure 1. Rather than traversing spacetime by means of a two-fold nested loop, it uses our cache oblivious traversal procedure `walk1Dspace`.

```
void walk1Dspace(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1)
{
    int Δt = t1 - t0;

    if (Δt == 1) {                                          /* base case */
        int x;
        for (x = x0; x < x1; ++x)
            kernel(t0, x);
    } else if (Δt > 1) {                                    /* recursion */
        if (2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * Δt) {      /* space cut */
            int xm = (2 * (x0 + x1) + (2 + ẋ0 + ẋ1) * Δt) / 4;
            walk1Dspace(t1, t1, x0, ẋ0, xm, -ds);
            walk1Dspace(t1, t1, xm, -ds, x1, ẋ1);
        } else {                                             /* time cut */
            int s = Δt / 2;
            walk1Dspace(t1, t1 + s, x0, ẋ0, x1, ẋ1);
            walk1Dspace(t1 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1);
        }
    }
}
```

Figure 3: Procedure `walk1` for traversing a 2D spacetime spanned by 1D space and time.

To ensure correct computation of the boundary conditions, we include the left and right boundary computations into the kernel and guard their application by means of an `if`-statement. The kernel computation itself remains unchanged. Note that procedure `kernel` can be designed without knowledge about the particular spacetime traversal generated by procedure `walk1Dspace`.

The perhaps more magical portion of the program in Figure 4 involves the arguments to procedure `walk1Dspace` and the choice of parameter `ds`. The latter is the slope of the spacetime cut used in `walk1Dspace` to split the spacetime region into trapezoids. The value of `ds` must be the maximum absolute value of the slopes of the left and rightmost edges of the stencil. In case of our 3-point stencil, this slope is $ds = 1$. The 3-point stencil has three dependency edges $e_{-1} = u(t, x-1) \rightarrow u(t+1, x)$, $e_0 = u(t, x) \rightarrow u(t+1, x)$, and $e_1 = u(t, x+1) \rightarrow u(t+1, x)$. The slopes of the leftmost edge e_{-1} and rightmost edge e_1 are $\texttt{ds}(e_{-1}) = 1$ and $\texttt{ds}(e_1) = -1$. Thus, the maximum absolute value is $\texttt{ds} = \max(|1|, |-1|) = 1$, so that $ds = 1$.

The first two arguments of procedure `walk1Dspace` are the inclusive lower bound of the time dimension $t_0 = 0$ and the exclusive upper bound $t_1 = T$. Note that the kernel computes $u(t+1, x)$ for all x and t, so that we obtain $u(T, x)$ for $t = T - 1$. The next two arguments specify the inclusive lower bound of the space dimension $x_0 = 0$ and the slope of the left boundary of the traversal domain $\dot{x}_0 = 0$. The last two arguments specify the exclusive upper bound of the space dimension $x_1 = N$, and the slope of the right boundary of the traversal domain $\dot{x}_1 = 0$. Among two possible choices, we pick the left and right boundaries of the traversal to be vertical by assigning $\dot{x}_0 = \dot{x}_1 = 0$ to produce a rectangular traversal domain; see [1] for more details.

Figure 5 shows the Lax-Wendroff kernel that replaces the generic kernel in Figure 4 to implement a cache oblivious Lax-Wendroff solver. For simplicity, we assume constant boundary conditions for

```
double u[2][N];

void kernel(int t, int x)
{
   if (x == 0)
      leftboundary(t);
   else if (x == N-1)
      rightboundary(t);
   else
      u[(t+1)%2][x] = f(u[t%2][x-1], u[t%2][x], u[t%2][x+1]);
}

const int ds = 1;  /* slope of spacetime cut */

void traverse_spacetime(void)
{
   walk1Dspace(0, T, 0, 0, N, 0);
}
```

Figure 4: Cache oblivious toggle-array program for boundary-value, initial-value problem.

$x = 0$ and $x = N-1$. While this implementation is correct and cache oblivious, it is very inefficient, because it incurs the overhead of one function call, one conditional branch, a few redundant memory accesses, and several redundant index computations in order to execute 8 floating-point operations. In the following, we show one way to optimize this implementation so as to minimize the overhead. We wish to emphasize, however, that the "code optimizations" we introduce here are necessary only because a kernel of 8 flops is so small. For applications with sufficiently many floating-point operations in the kernel, the overheads may be negligible already. For example, the kernel of the LBMHD application [2] has more than 300 floating-point operations, which amortizes not only the recursive function calls but all other overheads as well.

The dominant overhead of the kernel procedure in Figure 5 is due to the restriction to a single spacetime point (t, x), which prevents loop unrolling and similar optimizations. Our first step is then to modify the kernel to operate on a trapezoidal region of space, and to adopt procedure `walk1Dspace` to enter the base case of the recursion and call the kernel on a coarser problem. We call this transformation ***leaf coarsening***. The leaf-coarsened version of procedure `walk1Dspace` is shown in Figure 6. Rather than entering the base case when $\Delta t = 1$, the coarsened base case permits trapezoids whose base $x_1 - x_0$ is as wide as a tunable fudge factor `XBASE_FUDGEFACTOR`. Note that we introduce this fudge factor for the purpose of increasing the number of floating point operations in the base case, which enables us to apply code optimization techniques to improve performance. This fudge factor does not characterize any aspect of the memory hierarchy. However, the choice of the fudge factor is not entirely independent of the memory hierarchy. The working set of the coarsened base case should still fit into the L1-cache, and should even be relatively small compared to the L1-cache. We choose value 1,000, because 1,000 double-precision floating point numbers fit easily into the 32 KB L1 caches of our processors, yet is large enough to provide us with thousands of floating-point operations in the coarsened base case.

```
void kernel(int t, int x)
{
  const double C    = 0.45;
  const double HC   = C/2.0;
  const double CSQ  = C*C;
  const double HCSQ = CSQ/2.0;
  double *new = u[(t+1)%2];
  double *old = u[t%2];

  if (x == 0 || x == N-1)
    new[x] = old[x]; /* assume constant boundary conditions */
  else
    new[x] = old[x] - HC*(old[x+1]-old[x-1]) + HCSQ*(old[x+1]-2.0*old[x]+old[x-1]);
}
```

Figure 5: Unoptimized Lax-Wendroff kernel for boundary-value problems with toggle arrays.

Note that the base case of `walk1Dspace` requires a different kernel that visits not only one spacetime point but an entire trapezoid. Figure 6 includes the code of a coarsened pseudo-kernel without the actual kernel computation. This kernel is, strictly speaking, not cache oblivious, because it traverses the coarsened leaf trapezoid iteratively. To arrive at a fully cache oblivious implementation, the spacetime points within the coarsened leaf had to be visited in the sequence prescribed the the recursion of procedure `walk1Dspace`. Nevertheless, for all practical purposes, the solution in Figure 6 presents a reasonable compromise for two reasons: (1) unfolding the coarsened leaf trapezoid in a cache oblivious fashion is so tedious that it is impractical without compiler support. Unfortunately, such support does not exist in today's compilers. (2) The two-fold nested loop in procedure `kernel` in can be optimized manually so as to deliver peak performance. Having moved the loops on x and t inside the kernel, we can unroll the x-loop manually, and check for boundary conditions outside the x-loop. The resulting kernel is shown in Figure 7. We choose to unroll the x loop nine times because this is the largest unrolling factor that does not cause register spills on the PowerPC processors. We conclude that leaf coarsening, although not fully cache oblivious, presents a practical software engineering compromise for today's computer technology.

4.2 Periodic 1D Stencils with Toggle Arrays

Figure 8 shows the simple iterative version of the periodic initial-value problem with toggle arrays. Since the iteration due to the two-fold nested loop in procedure `traverse_spacetime` visits the spacetime domain time step by time step, toggle arrays provide sufficient storage to handle the wrap-around. We use modulo arithmetic for the index computations to access the points on the left and right boundaries of the toggle array. In particular, to compute $u(t+1,0)$, we need to access the left neighbor on the right boundary at spacetime point $(t, N-1)$. Using modulo arithmetic, this point is $(t, (x-1) \bmod N)$ for $x = 0$, because $(-1) \bmod N = N-1$. For all other $x \in \{1, \ldots, N-1\}$, the left neighbor is point $(t, x-1)$, which is equal to $(t, (x-1) \bmod N)$ as well. A similar relation holds for the right neighbors. To compute $u(t+1, N-1)$ on the right boundary, we need the value from the right neighbor on the left boundary $(t, 0) = (t, (x+1) \bmod N)$ for $x = N-1$. Thus, we may use modulo arithmetic to access all points of the spacetime domain, independent

```
void kernel(t0, t1, x0, ẋ0, x1, ẋ1)
{
   int x, t;

   for (t = t0; t < t1; t++) {
      for (x = x0; x < x1; x += 1)
         u[(t+1)%2][x] = ... ;
      x0 += ẋ0;  x1 += ẋ1;
   }
}

#define XBASE_FUDGEFACTOR  1000   /* fudge factor for coarsened base case */

void walk1Dspace(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1)
{
    int Δt = t1 - t0;

    if (Δt <= 1 || (x1-x0) < XBASE_FUDGEFACTOR) {    /* coarsened base case */
        kernel(t0, t1, x0, ẋ0, x1, ẋ1);
    } else {                                                /* recursion */
        if (2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * Δt) {
                                        /* space cut */
            int xm = (2 * (x0 + x1) + (2 + ẋ0 + ẋ1) * Δt) / 4;
            walk1Dspace(t1, t1, x0, ẋ0, xm, -ds);
            walk1Dspace(t1, t1, xm, -ds, x1, ẋ1);
        } else {                                             /* time cut */
            int s = Δt / 2;
            walk1Dspace(t1, t1 + s, x0, ẋ0, x1, ẋ1);
            walk1Dspace(t1 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1);
        }
    }
}
```

Figure 6: Leaf-coarsened version of procedure `walk1Dspace` with pseudo kernel.

```
void kernel(int t0, int t1, int x0, int dx0, int x1, int dx1)
{
  const double C    = 0.45;
  const double HC   = C/2.0;
  const double CSQ  = C*C;
  const double HCSQ = CSQ/2.0;
  int x, t;

  for (t = t0; t < t1; ++t) {
    double *new = u[(t+1)%2];
    double *old = u[t%2];
    int nx0 = x0, nx1 = x1;

    if (x0 == 0) { /* left boundary */
      new[x0] = old[x0]; ++nx0;
    }

    if (x1 == N) { /* right boundary */
      --nx1; new[nx1] = old[nx1];
    }

    for (x = nx0; x < nx1 - 8; x += 9) {
      double n0, n1, n2, n3, n4, n5, n6, n7, n8;

#define UPDATE(result, x) \
 result = old[x] - HC*(old[x+1]-old[x-1]) + HCSQ*(old[x+1]-2.0*old[x]+old[x-1])

      UPDATE(n0, x+0); UPDATE(n1, x+1); UPDATE(n2, x+2);
      UPDATE(n3, x+3); UPDATE(n4, x+4); UPDATE(n5, x+5);
      UPDATE(n6, x+6); UPDATE(n7, x+7); UPDATE(n8, x+8);

      new[x+0] = n0; new[x+1] = n1; new[x+2] = n2;
      new[x+3] = n3; new[x+4] = n4; new[x+5] = n5;
      new[x+6] = n6; new[x+7] = n7; new[x+8] = n8;
    }

    for (; x < nx1; x += 1)
      UPDATE(new[x], x);

    x0 += dx0; x1 += dx1;
  }
}
```

Figure 7: Optimized Lax-Wendroff kernel for nonperiodic space, using toggle arrays.

of their location as boundary points or inner points. Note that the modulo operator % of the C programming language is defined according to the mathematical definition for positive numbers only. Therefore, we use the C expression $(x + N - 1)\%N$ for $x > -N$ to compute $(x - 1) \bmod N$.

```
double u[2][N];

void kernel(int t, int x)
{
   u[(t+1)%2][x] = f(u[t%2][(x+N-1)%N], u[t%2][x], u[t%2][(x+1)%N]);
}

void traverse_spacetime(void)
{
   int t, x;
   for (t = 0; t < T; t++) {
      for (x = 0; x < N; x++)
         kernel(t, x);
   }
}
```

Figure 8: Toggle-array program for periodic initial-value problem with 3-point stencil.

The use of modulo arithmetic enables us avoid two `if`-statements in the kernel procedure to test for the left and right boundaries, at the expense of several modulo operations. Furthermore, modulo arithmetic enables us to use a cache oblivious spacetime traversal by generating a traversal domain that is different from the spacetime domain, but can be mapped into the spacetime domain by means of modulo arithmetic. For periodic initial-value problems, we tilt the rectangular spacetime domain into a parallelogram-shaped traversal domain with boundary slopes $\dot{x}_0 = \dot{x}_1 = ds$. Figure 5 in [1] illustrates a parallelogram-shaped traversal domain. Figure 9 below shows the cache oblivious version of the program. Compared to the iterative version in Figure 8, we need to perform additional modulo operations by replacing index x with $x\%N$.

For small kernel computations like the Lax-Wendroff kernel in Equation 2, the index computations using modulo arithmetic constitute a non-negligible overhead. Thus, optimizing such a kernel includes eliminating as many modulo computations as possible for the common case. Figure 10 shows our optimized kernel for use with a leaf-coarsened version of procedure `walk1Dspace`.

5 1D Stencils with Boundary Passing

In this section, we introduce cache oblivious versions of the boundary-passing program introduced in Figure 2 of Section 2. In Section 5.1, we transform the program in Figure 2 into a cache oblivious version. Then in Section 5.2, we extend the problem domain from boundary-value, initial-value problems to periodic initial-value problems.

We begin by incorporating the boundary passing method into our cache oblivious spacetime traversal by using temporary variables to pass the boundary values from the left to the right trapezoid. Figure 11 illustrates our strategy. When we perform a space cut, we use a ***boundary***

```
double u[2][N];

void kernel(int t, int x)
{
   u[(t+1)%2][x%N] = f(u[t%2][(x+N-1)%N], u[t%2][x%N], u[t%2][(x+1)%N]);
}

const int ds = 1;        /* slope of walk1Dspace cuts */

void traverse_spacetime(void)
{
   walk1Dspace(0, T, 0, ds, N, ds);
}
```

Figure 9: Cache oblivious toggle-array program for periodic initial-value problem.

array to pass the spacetime points on the right boundary of the left trapezoid $\mathcal{T}_1$ to the right trapezoid $\mathcal{T}_2$. Since we use slope -1 to cut the trapezoid of a 3-point stencil into a left and a right part, the data dependencies of the stencil demand that the left trapezoid $\mathcal{T}_1$ must be traversed before the right trapezoid $\mathcal{T}_2$.

To preserve the working-set savings of the boundary-passing method over the use of toggle arrays, we wish to keep the size of the boundary array small. However, to maximize the reuse of spacetime points, we wish to maximize the height of the trapezoid. Note that the maximum height of the trapezoid is limited by the size of the boundary array NB, if NB is less than $(X_1 - X_0)/2$, the maximum height of the trapezoid determined by the stencil structure. We satisfy these conflicting goals by choosing the size of the boundary array and, therefore, the maximum height of the trapezoid to be asymptotically insignificant compared to working-set array u, yet large enough to permit an effective reuse of spacetime points. For example, we may pick the size of the boundary array NB to be $NB = \sqrt{N}$.

To implement the restriction of the height of the trapezoid to the size of the boundary array, we perform a minor change to procedure `walk1Dspace`. Figure 12 shows the modified version of procedure `walk1Dspace`. Compared to the leaf-coarsened version of Section 4.1, it includes a conjunction of the original condition for the space cut with the new condition $\Delta t < NB$, which ensures that the height of the trapezoid does not exceed the size of the boundary array NB.

A given time step t contains the set of spacetime points at time t for all x of the space domain. During the cache-oblivious spacetime traversal, time steps are computed only partially: some points of the time step have been computed while others have not. For each time step, we need to allocate an entry in the boundary array to communicate values from the left to the right. In the naive iterative traversal from Figure 2, at most one level can be partially complete at any point during the execution, and therefore the boundary array degenerates to a single variable `w`. In the cache oblivious case, the "height check" in Figure 12 guarantees that at most NB partially computed time steps can exist simultaneously. The choice of NB is a tradeoff between the amount of cache reuse (at most NB) and the amount of extra storage required (at least NB). As a compromise, we choose $NB = \min(T, \lfloor\sqrt{N}\rfloor)$, for which cache reuse grows with N but requires an asymptotically negligible amount of additional storage only.

```
void kernel(int t0, int t1, int x0, int dx0, int x1, int dx1)
{
  const double C    = 0.45;
  const double HC   = C/2.0;
  const double CSQ  = C*C;
  const double HCSQ = CSQ/2.0;
  int x, t;

  for (t = t0; t < t1; ++t) {
    double *old = u[t%2];
    double *new = u[(t+1)%2];

    if ((N + x0 - 1) / N == (N + x1 + 1) / N) {
      int nx0 = x0 % N;
      int nx1 = x1 % N;

      for (x = nx0; x < nx1 - 8; x += 9) {
        double n0, n1, n2, n3, n4, n5, n6, n7, n8;

#define UPDATE(result, x) \
 result = old[x] - HC*(old[x+1]-old[x-1]) + HCSQ*(old[x+1]-2.0*old[x]+old[x-1])

        UPDATE(n0, x+0); UPDATE(n1, x+1); UPDATE(n2, x+2);
        UPDATE(n3, x+3); UPDATE(n4, x+4); UPDATE(n5, x+5);
        UPDATE(n6, x+6); UPDATE(n7, x+7); UPDATE(n8, x+8);

        new[x+0] = n0; new[x+1] = n1; new[x+2] = n2;
        new[x+3] = n3; new[x+4] = n4; new[x+5] = n5;
        new[x+6] = n6; new[x+7] = n7; new[x+8] = n8;
      }

      for (; x < nx1; x += 1)
        UPDATE(new[x], x);
    } else {
      for (x = x0; x < x1; ++x)
        new[x%N] = old[x%N] - HC*(old[(x+1)%N]-old[(x-1+N)%N])
          + HCSQ*(old[(x+1)%N]-2.0*old[x%N]+old[(x-1+N)%N]);
    }

    x0 += dx0;
    x1 += dx1;
  }
}
```

Figure 10: Optimized kernel for cache oblivious toggle-array version for the periodic initial-value problem.

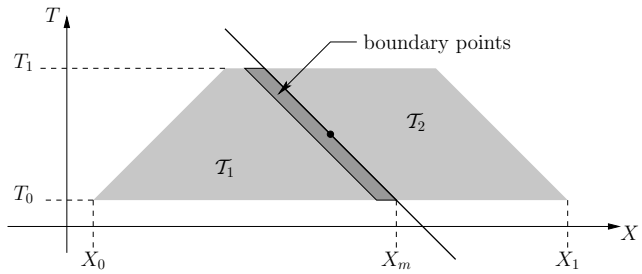

Figure 11: Illustration of a *space cut* with *boundary*. When the space dimension is large enough and the time slice is small enough such that the boundary points fit into the boundary array, procedure `walk1Dspace` cuts the trapezoid along the line of slope -1 through its center.

5.1 Nonperiodic 1D Stencils with Boundary Passing

Our nonperiodic boundary-passing version builds upon the simple program in Figure 2. We first introduce a simple cache oblivious version. Then, we present our optimized version of the Lax-Wendroff code. For simplicity, we assume that the boundary values $u(t, 0)$ on the left and $u(t, N-1)$ on the right boundary remain constant for all t. Figure 13 illustrates a simple cache oblivious program, that traverses the spacetime domain using a non-leaf-coarsened version of procedure `walk1Dspace`. Since we assume constant boundary values, we restrict our spacetime traversal to visiting space range $1 \leq x < N - 1$. For nonperiodic problems, we save the modulo arithmetic in the index computations by choosing arguments $\dot{x}_0 = \dot{x}_1 = 0$ for procedure `walk1Dspace`, so that the traversal domain is a rectangle with N space points and T time steps.

The kernel computation in Figure 13 uses array `u[`N`]` and boundary array `ulefties[`NB`]`. Setting $NB = \min(T, \sqrt{NB})$ ensures that $NB = o(N)$, so that contribution of the boundary array to the working-set is asymptotically insignificant. Taking the minimum ensures that the size of the boundary array does not exceed the maximum trapezoid height T for small values of T. We access the boundary array using index t mod NB, because a trapezoid spans the index range of time slice $T_0 \leq t < T_1$, where $T_1 - T_0 \leq NB$. The conditional assignment of variable `uleft` distinguishes between the left boundary and inner points. If $x = 1$, we compute spacetime point $u(t, 1)$, whose left neighbor is on the left boundary $u(t, 0)$. Otherwise, for $x > 1$, the value of the left neighbor is stored in boundary array `ulefties`. Since the traversal procedure visits spacetime point $(t, x - 1)$ before (t, x) for all x and t, and no other point (t, x') at time step t between those two points, the kernel finds the expected value in `ulefties[`$t\%NB$`]`. The right boundary at $x = N - 1$ does not require any special treatment, because we do not pass the constant boundary value `u[`$N-1$`]` across trapezoids. This boundary value is read only when visiting spacetime points $(t, N - 2)$ for all t.

Figure 14 presents the optimized version of the simple kernel shown in Figure 13 in Figure 14. Note that the optimized kernel preserves the loop structure of the leaf-coarsened kernel introduced in Figure 6. Since the nonperiodic boundary-value, initial-value problem does not require modulo arithmetic for the index computations, it suffices to unroll the inner x-loop to provide sufficiently many independent operations to the floating-point pipelines. We amortize the overhead of the `if`-statement testing for the left boundary by choosing a relatively large fudge factor `XBASE_FUDGEFACTOR` $= 1,000$. We note that optimizing this kernel is relatively easy.

```
#define XBASE_FUDGEFACTOR  1000     /* fudge factor for coarsened base case */

void walk1Dspace(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1)
{
    int Δt = t1 - t0;

    if (Δt <= 1 || (x1-x0) < XBASE_FUDGEFACTOR) {    /* coarsened base case */
        kernel(t0, t1, x0, ẋ0, x1, ẋ1);
    } else {                                                  /* recursion */
        if (Δt < NB && 2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * Δt) {
                                        /* space cut -- INCLUDES HEIGHT CHECK */
            int xm = (2 * (x0 + x1) + (2 + ẋ0 + ẋ1) * Δt) / 4;
            walk1Dspace(t1, t1, x0, ẋ0, xm, -ds);
            walk1Dspace(t1, t1, xm, -ds, x1, ẋ1);
        } else {                                                  /* time cut */
            int s = Δt / 2;
            walk1Dspace(t1, t1 + s, x0, ẋ0, x1, ẋ1);
            walk1Dspace(t1 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1);
        }
    }
}
```

Figure 12: Modified procedure `walk1Dspace` for boundary passing. The condition for the space cut includes a check that the height of the trapezoid Δt does not exceed the size of the boundary array *NB*.

5.2 Periodic 1D Stencils with Boundary Passing

We now turn to the periodic initial-value problem with a Lax-Wendroff kernel. We discuss the cache oblivious version with boundary passing. Since the problem is periodic, we must pass values across the cyclic wrap-around from the left boundary to the right of the spacetime domain and vice versa. The wrap-around requires trickier code optimizations than the programs presented in Section 5.1.

We begin our discussion with the relatively simple program for the non-leaf-coarsened spacetime traversal shown in Figure 15. Recall from Section 4.2 that we traverse the spacetime domain of periodic problems by generating a parallelogram-shaped traversal domain, and by using modulo arithmetic for the index computations to map the parallelogram-shaped traversal domain into the rectangular spacetime domain, see also Figure 16 below. Accordingly, we use arguments $\dot{x}_0 = \dot{x}_1 = ds$ for procedure `walk1Dspace` to generate the parallelogram with boundary slopes $ds = 1$ for our 3-point stencil. Furthermore, within the procedure `kernel`, we use modulo arithmetic to map the x index of the traversal domain into index $i = x \bmod N$ of our rectangular spacetime domain. Note that the mapping of t to $k = t \bmod NB$ has nothing to do with the parallelogram-shaped traversal domain. Its purpose is to access the boundary array of size *NB*, as already discussed in Section 5.1.

The periodicity of the problem requires that we pass values from the left boundary $u(t, 0)$ to

```
double u[N];
double ulefties[NB];    /* boundary array */

void kernel(int t, int x)
{
   double uleft, umid = u[x];

   uleft = (x == 1) ? u[0] : ulefties[t%NB];
   u[i] = f(uleft, umid, u[x+1]);
   ulefties[t%NB] = umid;
}

const int ds = 1;       /* slope of walk1Dspace cuts */

void traverse_spacetime(void)
{
   walk1Dspace(0, T, 1, 0, N-1, 0);
}
```

Figure 13: Cache oblivious boundary-passing program for boundary-value, initial-value problem.

compute $u(t+1, N-1)$ on the right boundary, and values from the right boundary $u(t, N-1)$ to compute $u(t+1, 0)$ on the left boundary of our spacetime domain. Since our spacetime traversal visits spacetime point $u(t, x-1)$ before $u(t, x)$, that is it traverses the space dimension from left to right, we can implement the cyclic wrap-around with one additional array `uwrap` as follows. Like the boundary array, array `uwrap` shall have size NB to hold as many values as required to obey the maximum-height limitation of our spacetime trapezoids.

To understand the use of array `uwrap`, we focus our attention on the wrap-around during a single time step t. Procedure `walk1Dspace` visits spacetime point $(t, 0)$ to compute $u(t+1, 0)$ on the left boundary before visiting point $(t, N-1)$ to compute $u(t+1, N-1)$ on the right boundary. Point $u(t+1, 0)$ depends on its left neighbor $u(t, N-1)$ on the right boundary, and point $u(t+1, N-1)$ depends on its right neighbor $u(t, 0)$ on the left boundary. Using modulo arithmetic illustrates that the left and right boundaries of the rectangular spacetime domain are immaterial for the periodic problem: due to the 3-point stencil, point $u(t+1, x \bmod N)$ depends on points $u(t, (x-1) \bmod N)$, $u(t, x \bmod N)$, and $u(t, (x+1) \bmod N)$. There are no boundaries due to the problem itself, but due to the alignment of the space dimension with the index range of array `u` only.

As indicated in Figure 16, we cut the cyclic dependency of the periodic problem in the parallelogram-shaped traversal domain to the left of spacetime point $(t, x = t)$ and to the right of point $(t, x = t + N - 1)$, the boundaries of the parallelogram. Now, consider time step t. When visiting point $(t, x = t)$ on the left boundary, we compute $u(t+1, x)$ by using values $u(t, x-1)$, $u(t, x)$, and $u(t, x+1)$. Since $u(t, x)$ is on the left boundary of the parallelogram-shaped traversal domain, value $u(t, x-1)$ is stored in point $(t, x+N-1)$ on the right boundary. Analogously, when visiting point $(t, x+N-1)$ on the right boundary, we compute $u(t+1, x+N-1)$, and need value $u(t, x+N)$ from point $u(t, x = t)$ on the left boundary. Furthermore, to compute value $u(t+1, x+1)$, we need value $u(t, x)$, which we pass via boundary array `ulefties`. Now, consider point $(t, x = t)$ on the

```
inline void kernel(int t0, int t1, int x0, int dx0, int x1, int dx1)
{
  const double C    = 0.45;
  const double HC   = C/2.0;
  const double CSQ  = C*C;
  const double HCSQ = CSQ/2.0;
  double uleft;
  int t, x;

  for (t=t0; t<t1; t++) {
    if (x0 == 1)                  /* left boundary */
      uleft = u[0];
    else
      uleft = ulefties[t%NB];

    for (x=x0; x<x1-8; x+=9) {  /* unroll 9 times */
      double u0=u[x],   u1=u[x+1], u2=u[x+2], u3=u[x+3], u4=u[x+4];
      double u5=u[x+5], u6=u[x+6], u7=u[x+7], u8=u[x+8], u9=u[x+9];
      u[x]   = u0 - HC*(u1-uleft) + HCSQ*(u1-2.0*u0+uleft);
      u[x+1] = u1 - HC*(u2-u0) + HCSQ*(u2-2.0*u1+u0);
      u[x+2] = u2 - HC*(u3-u1) + HCSQ*(u3-2.0*u2+u1);
      u[x+3] = u3 - HC*(u4-u2) + HCSQ*(u4-2.0*u3+u2);
      u[x+4] = u4 - HC*(u5-u3) + HCSQ*(u5-2.0*u4+u3);
      u[x+5] = u5 - HC*(u6-u4) + HCSQ*(u6-2.0*u5+u4);
      u[x+6] = u6 - HC*(u7-u5) + HCSQ*(u7-2.0*u6+u5);
      u[x+7] = u7 - HC*(u8-u6) + HCSQ*(u8-2.0*u7+u6);
      u[x+8] = u8 - HC*(u9-u7) + HCSQ*(u9-2.0*u8+u7);
      uleft = u8;
    }

    for ( ; x<x1; x++) {        /* remainder */
      double umid = u[x];
      u[x]  = umid - HC*(u[x+1]-uleft) + HCSQ*(u[x+1]-2.0*umid+uleft);
      uleft = umid;
    }

    ulefties[t%NB] = uleft;

    x0 += dx0;  x1 += dx1;      /* update loop bounds */
  }
}
```

Figure 14: Optimized cache oblivious kernel for nonperiodic problems with boundary passing.

```
double u[N];
double ulefties[NB];   /* boundary array */
double uwrap[NB];      /* cylic wrap-around */

void kernel(int t, int x)
{
   int i = x % N;
   int k = t % NB;
   double uleft, uright, umid = u[i];

   uleft  = (x == t) ? u[(x+N-1)%N] : ulefties[k];
   uright = (x == t+N-1) ? uwrap[k] : u[(x+1)%N];

   u[i] = f(uleft, umid, uright);

   if (x == t)
      uwrap[k] = umid;   /* wrap left boundary around to right */
   ulefties[k] = umid;
}

const int ds = 1;       /* slope of walk1Dspace cuts */

void traverse_spacetime(void)
{
   ulefties[0] = u[N-1];            /* initialization */
   walk1Dspace(0, T, 0, ds, N, ds);
}
```

Figure 15: Cache oblivious boundary-passing program for periodic initial-value problem.

left boundary of the parallelogram again. Upon visiting this spacetime point, we compute value $u(t+1,x)$. We need value $u(t,x)$ to update both the right neighbor and the value on the right boundary due to the cyclic wrap-around. Thus, we need to store $u(t,x)$ in two temporary places, boundary array `ulefties` to pass it to the right, and `uwrap` to pass it through the wrap-around to the left. None of the other points in time step t needs the wrap-around. Our traversal procedes from left to right passing value $u(t,x)$ via `ulefties` to compute its right neighbor $u(t+1,x+1)$, passing $u(t,x+1)$ via `ulefties` to compute its right neighbor $u(t+1,x+2)$, and so on, until we visit the right boundary, which uses value $u(t,x+N-2)$ from `ulefties` and value $u(t,x=t)$ via the wrap-around from `uwrap`. The `kernel` function in Figure 15 implements this strategy.

Next, we discuss the optimized version of the cache oblivious boundary-passing kernel in Figure 17. Like in Section 5.1, we unroll the inner x-loop of the leaf-coarsened kernel to cope with the deep floating-point pipelines. In addition, we wish to amortize the overhead of the modulo arithmetic, and the branches needed to distinguish the passing of values through the wrap-around via array `uwrap`. With only 8 flops, the Lax-Wendroff kernel poses a serious challenge to any code optimizer. Manual code optimization based on asymptotic amortization strategies are needed to

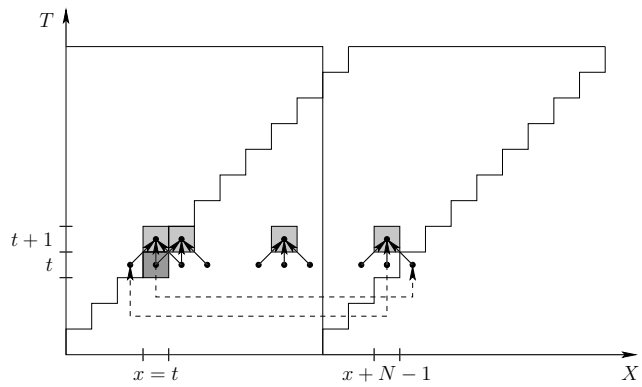

Figure 16: Illustration of *wrap-around* for periodic initial-value problems. The parallelogram-shaped traversal domain maps into the rectangular spacetime domain by means of modulo arithmetic.

obtain high performance. Besides unrolling the x-loop, we distinguish the case where the space range $[x_0, \ldots, x_1[$ excludes the boundaries of the parallelogram. If so, we need only two modulo operations to compute the left and right bounds of the unrolled loop. Otherwise, in the rare case, we include the modulo computation as part of every index computation. The rare case also covers the case where the trapezoid includes the parallelogram boundaries, where we have to pass values via array `uwrap`. Like in Section 5.1, this implementation can amortize the overhead of the rare case only if the base of the trapezoid is relatively large. Hence the relatively large value of 1,000 for `XBASE_FUDGEFACTOR`.

6 Performance Analysis

In this section, we analyze in detail the performance of our cache oblivious, periodic, optimized Lax-Wendroff code with boundary passing. By principle of exclusion, we study the effects of the memory hierarchy and kernel code optimizations separately. Furthermore, we analyze the performance impact of the floating-point unit and the hardware prefetcher empirically.

When refering to the *unrolled iterative version*, we mean unrolling the inner loop in Figure 1 as shown in Figure 7.

Performance vs. Array Size

Figure 18 compares the runtime per spacetime point of three different programs on a Power5, varying the number N of space points. The two iterative programs, the naive iterative version and the loop unrolled, optimized version, become slower as N increases, while the cache oblivious program becomes faster until its performance reaches an asymptotic limit.

Our cache oblivious program outperforms the iterative versions for large values of N, because our cache oblivious memory reuse is effective for problems so large that they do not fit into fast caches, and because our code optimizations are based on amortizing the overheads by means of large leaf trapezoids. For $N < 10^5$, the number of spacetime points is too small for our optimizations to be effective. The shape of the curve for our cache oblivious program depends on the value of fudge factor `XBASE_FUDGEFACTOR` and on the implementation of the kernel. In our kernel implementation

```
double u[N];
double ulefties[NB];    /* boundary array */
double uwrap[NB];       /* periodic wrap-around */

void kernel(int t0, int t1, int x0, int dx0, int x1, int dx1)
{
  const double C = 0.45, HC = C/2.0, CSQ = C*C, HCSQ = CSQ/2.0;
  int t, x;
  for (t = t0; t < t1; ++t) {
    int tmod = t % NB;
    if (x0 > t && x1 < t + N - 1  /* no points on left and right boundary */
        && ((N + x0 - 1) / N == (N + x1 + 1) / N)) { /* no wrap-around */
      int nx0 = x0 % N,  nx1 = x1 % N;
      double um1 = ulefties[tmod];
#     define UPDATE(result, um1, u0, u1) result = u0 - HC*(u1-um1) + HCSQ*(u1-2.0*u0+um1)
      for (x = nx0; x < nx1 - 8; x += 9) {
        double u0 = u[x],   u1 = u[x+1], u2 = u[x+2], u3 = u[x+3], u4 = u[x+4];
        double u5 = u[x+5], u6 = u[x+6], u7 = u[x+7], u8 = u[x+8], u9 = u[x+9];
        UPDATE(u[x], um1, u0, u1);  UPDATE(u[x+1], u0, u1, u2); UPDATE(u[x+2], u1, u2, u3);
        UPDATE(u[x+3], u2, u3, u4); UPDATE(u[x+4], u3, u4, u5); UPDATE(u[x+5], u4, u5, u6);
        UPDATE(u[x+6], u5, u6, u7); UPDATE(u[x+7], u6, u7, u8); UPDATE(u[x+8], u7, u8, u9);
        um1 = u8;
      }
      for (; x < nx1; ++x) {
        double u0 = u[x], u1 = u[x+1];
        UPDATE(u[x], um1, u0, u1);
        um1 = u0;
      }
      ulefties[tmod] = um1;
    } else {
      for (x = x0; x < x1; ++x) {
        double uleft, uright, umid = u[x%N];
        if (x == t) {      /* left boundary */
          uleft = u[(x+N-1)%N];
          uwrap[tmod] = umid;
        } else
          uleft = ulefties[tmod];
        if (x == t + N - 1) /* right boundary */
          uright = uwrap[tmod];
        else
          uright = u[(x+1)%N];
        u[x%N] = umid - HC*(uright-uleft) + HCSQ*(uright-2.0*umid+uleft);
        ulefties[tmod] = umid;
      }
    }
    x0 += dx0;  x1 += dx1;
  }
}
```

Figure 17: Optimized cache oblivious boundary-passing kernel for the periodic initial-value Lax-Wendroff program.

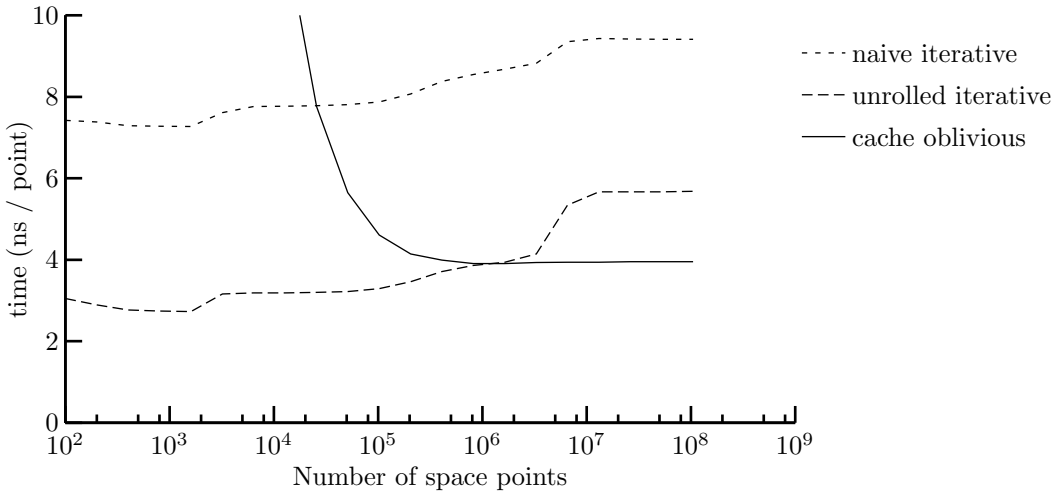

Figure 18: Performance of the naive iterative Lax-Wendroff program (*naive iterative*), our unrolled iterative Lax-Wendroff program (*unrolled iterative*), and our optimized cache oblivious program (*cache oblivious*) on a Power5 (1.66 GHz) with $T = 128$ time steps.

shown in Figure 17, we have chosen for the sake of simplicity to check whether the segment $[x_0, x_1)$ intersects the wrap-around boundary $\pmod N$. If so, we execute the whole segment in the `else` branch, paying the full cost of modular arithmetic for each point. Up to `XBASE_FUDGEFACTOR` points (1000 points in our case) are therefore computed in "slow mode." Our kernel can be refined to execute faster for small problems if desired. However, due to our focus on improving the performance of large problems, we do not pursue this issue any further.

Performance vs. Number of Time Steps

The effectiveness of our cache oblivious algorithm for stencil computations stems from reusing spacetime points as often as possible. In theory, the performance gain increases as the number of time steps T increases, because a larger number of time steps permits higher trapezoids and, therefore, improved reuse. We evaluate the lower bound on T, that is the smallest number time step where cache obliviousness is effective, empirically.

Figure 19 shows the performance of the same three programs used to analyze the effect of array size above on a Power5 for various values of the number of time steps T and a constant number of space points N. As expected, the runtime per spacetime point of the iterative programs is independent of T. Perhaps more surprisingly, the performance of the cache oblivious program degrades only marginally as T decreases. Just 3 time steps are sufficient for the runtime to be within 10 % of the asymptotic performance of $3.95\,ns$ per spacetime point. We conclude that the benefits of the cache oblivious program are not limited to large numbers of time steps T, but might well be suited as a core of an adaptive solver.

Effects of Data-dependent FPU Performance

An examination of the assembly code of our cache oblivious program reveals that the asymptotic performance of about $3.95\,ns$ per spacetime point misses the peak performance of this particular

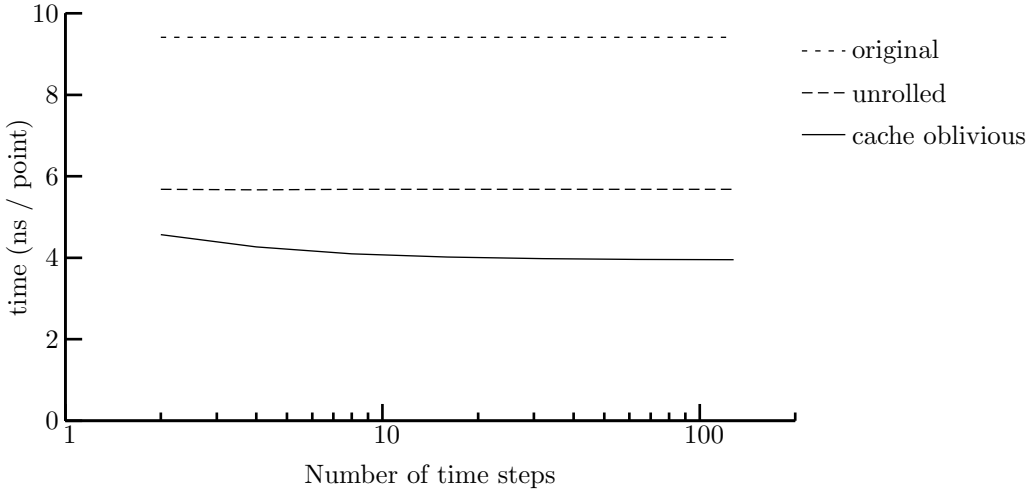

Figure 19: Performance of three program versions on Power5, for varying number T of time steps with $N = 104,857,600$ space points.

program on a Power5 by about 40 %. The reason for this performance degradation is the data-dependency of the performance of the floating-point unit.

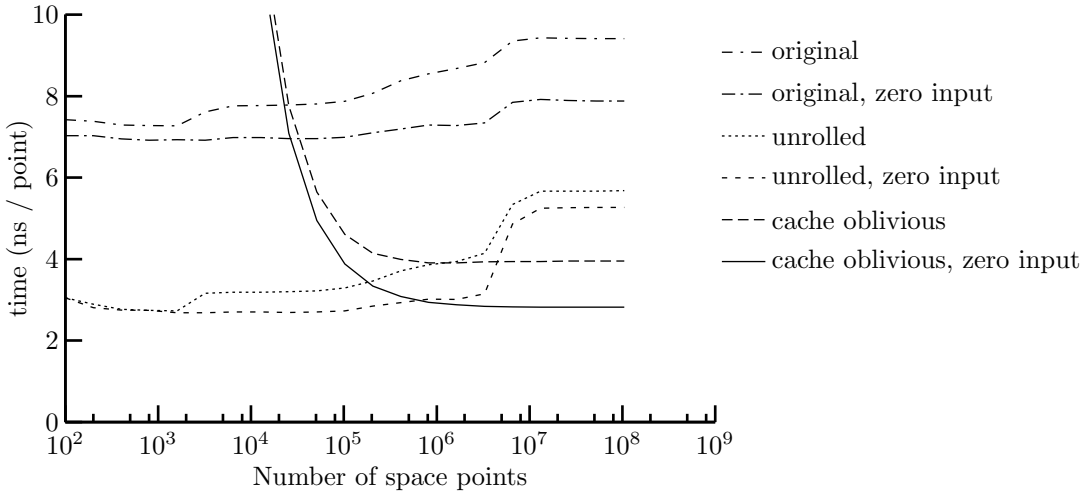

Figure 20: Performance of three program versions on Power5, for varying number N of space points and $T = 128$ time steps. For each program, we compare the runtime of a version with realistic initial values with the version that sets all values to zero.

To understand the effect of the data-dependent performance of the Power5 floating-point units, we compare the performance of our three programs with realistic initial values with the performance when forcing all values initially to zero. Figure 20 shows the corresponding runtimes per spacetime point. The runtimes with zero inputs are consistently lower than those with realistic initial values. In particular, note that the asymptotic runtimes ($2.8\,ns$) of our cache oblivious program with zero inputs are approximately equal to the runtimes of the optimized, loop unrolled iterative version for small problem sizes that fit into the L1-cache. This result indicates that we have succeeded with our cache oblivious version to eliminate all overheads due to and minimizing the performance impact

of the memory hierarchy. Unfortunately, there is no software remedy for the data-dependent FPU performance without changing the numerics.

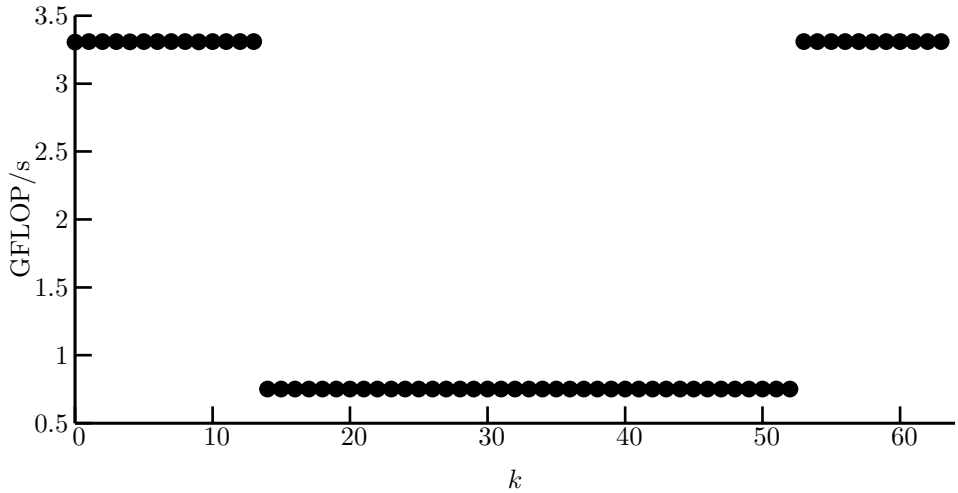

Figure 21: Performance of Power5 when adding the two double precision numbers $a = (1.0 + 2^{-k})$ and $b = -1.0$. The peak performance of the machine is 3.3 billion additions per seconds, which is achieved for small and large k. However, for $14 \leq k < 53$, floating-point performance drops by a factor of 4.4 to 0.75 Gflop/s.

In the following, we provide an empirical explanation for the performance degradation due to the FPU. The FPU of the Power4, Power5, and 970 processors incurs a penalty for subtracting two floating-point numbers that are approximately equal. The issue is neither due to cancellation nor due to denormalized numbers. Instead, we observe the performance degradation for normal floating-point numbers. The phenomenon is detailed in Figure 21, which plots the performance of a Power5 when adding the two double precision numbers $a = (1.0 + 2^{-k})$ and $b = -1.0$, for various values of k. For $k < 14$, the machine achieves its peak performance of two additions per cycle. However, for $14 \leq k < 53$, the machine incurs additional stalls that cause performance to degrade by a factor of 4.4. For $k \geq 53$, we have $1.0 + 2^{-k} = 1$ in double precision due to cancellation, and the machine executes again at full speed. Because it computes derivatives by subtracting two numbers that are approximately equal, the Lax-Wendroff code incurs this speed penalty in the common case. In general, most finite difference computations will suffer a performance degradation due to this effect.

Effects of Fused Multiply-add Instruction

We note that the Lax-Wendroff kernel in Equation 2 contains the subexpression

$$u(t,x) - c_0 * \big(u(t,x+1) - u(t,x-1)\big)\,. \tag{3}$$

This expression generates three flops: two subtractions and one multiplication.

We can take advantage of the PowerPC fused multiply-add instruction (FMA) by rewriting the expression as follows:

$$u(t,x) + c_0 * \big(u(t,x-1) - u(t,x+1)\big)\,. \tag{4}$$

This coding generates two floating-point instructions: one subtraction and one fused multiply-add.

The reason why the second expression is better suited for PowerPC's than the first is the following. The instruction set of the PowerPC includes an instruction that computes $-(ab - c)$, but not an instruction that computes $c - ab$. While mathematically equivalent, the two expressions are different in IEEE floating point—specifically, they may differ in sign when the result is 0. Consequently, a compiler is not allowed to use an FMA instruction for the original code, but it can use the FMA instruction in the second case.

The recommended code change reduces the complexity of the Lax-Wendroff kernel from 6 floating-point instructions (3 add, 2 fma, 1 mul) to 5 instructions (2 add, 3 fma). We have observed the corresponding performance increase of about 16.7 % in our experiments. However, we did not implement this code transformation in our programs, because introducing the FMA changes the numerical behavior of the program.

Effects of the Prefetcher

Finally, we show that the hardware prefetcher does not affect the performance of our cache oblivious codes, which already minimizes the number of data movements in the memory hierarchy. However, the prefetcher speeds up the iterative versions of the Lax-Wendroff code, although not enough to match the performance of our cache oblivious codes.

On both processors Power5 and 970 the hardware prefetcher for data is active by default. We examine the effects of the prefetcher by turning the prefetcher off on a 970 processor. On the Power5, the prefetcher can be disabled from within a privileged hypervisor mode only, over which ordinary users have no control.

problem	periodic		nonperiodic	
storage scheme	toggle	pass	toggle	pass
iterative [*sec*]	18.45	18.39	18.42	18.55
cache oblivious [*sec*]	3.40	3.54	3.44	3.50
speedup	5.42	5.19	5.36	5.30

Table 2: Runtimes and speedups of cache oblivious Lax-Wendroff codes versus the iterative programs ***without prefetcher*** on 970 for $N = 10,000,000$ space points and $T = 100$ time steps.

Table 2 shows the runtimes and speedups of our optimized cache oblivious program compared to the naive, iterative Lax-Wendroff codes on a 970 processor. These numbers should be compared with those in Table 1 of Section 1. We find that the runtimes of the cache oblivious version are essentially unchanged while those of the naive, iterative version increase by a factor of 2. We conclude that the hardware prefetcher is capable of speeding up the naive program by a factor of 2, yet does not eliminate the impact of the memory hierarchy entirely. Furthermore, since our cache oblivious program does not benefit from the hardware prefetcher at all, it would be desirable to disable the prefetcher, and save the associated power consumption.

References

[1] Matteo Frigo and Volker Strumpen. Cache Oblivious Stencil Computations. In *International Conference on Supercomputing*, pages 361–366, Boston, MA, June 2005. ACM Press.

[2] Matteo Frigo and Volker Strumpen. The Memory Behavior of Cache Oblivious Stencil Computations. *The Journal of Supercomputing*, 2006. Accepted for publication.

[3] David Hensinger and Chris Luchini. Private communication, January 2006.