Performance Characteristics of Cache Oblivious Implementation Strategies for Hyperbolic Equations on Opteron Based Super Computers

David Hensinger, *Sandia National Laboratories;*
Chris Luchini, *Sci Tac LLC;*
Matteo Frigo, *IBM Austin Research Laboratory;*
Volker Strumpen, *IBM Austin Research Laboratory.*

ABSTRACT: *For scientific computations, moving data from main memory into the processor cache incurs significant latency and comes at the expense of useful calculation. Cache oblivious strategies attempt to amortize the costs of movement of high latency data by carrying out as many operations as possible using in-cache data. For explicit methods applied to hyperbolic equations, cache oblivious algorithms advance regions of the solution domain through multiple time steps. Success of these strategies hinges on whether the performance advantage associated with computing on in-cache data offsets the overhead associated with managing multiple solution states. The performance characteristics of several computational kernels implemented using cache oblivious strategies were tested.*

KEYWORDS: Cache oblivious stencil computations, Performance analysis

1. Introduction

Since scientific data sets tend to consume a significant fraction of main memory (DRAM, Gigabytes in size), calculation performance is frequently limited by the processor cycles expended moving data from DRAM to caches (Megabytes in size) and thence to registers (Kilobytes in size). For this reason taking into consideration the locality of reference of memory access and the efficiency of caching can be a very important part of application performance. Cache oblivious strategies seek to improve processing performance by arranging operations on spatially coherent problem sub-domains to allow re-use of cached data. The re-use of in-cache data in these sub-domains is achieved by performing multiple iterations on each sub-domain during each visit. In the case of the transient solution of hyperbolic equations, this amounts to moving a sub-domain several steps forward in time during each visit. This approach is significantly different from using worksets in which data is copied explicitly into and out of a contiguous buffer area in main memory on the assumption that the copying overhead will be amortized by subsequent cache efficiency.

Cache oblivious strategies for stencil computations will be outlined, next the computational kernels and implementation strategies used will be introduced, and finally performance results will be presented and discussed.

2. Cache Oblivious Strategy

Kernels

The cache oblivious approach for stencil computations requires that the calculation can move forward by repeated application of a computational kernel to a subset of the computational domain. The effect that cache oblivious strategies have on application performance is dependent upon the efficiency of the kernel. If the kernel is inefficient, then the application of cache oblivious algorithms will have little effect.

Recursive Partitioning

The cache oblivious strategies tested here were developed by Frigo and Strumpen [2], based on the work of Frigo et al. [1]. The cache oblivious strategy has a parameter K representing the maximum number of steps for which a sub-domain may be evolved independently in time, and another parameter SIGMA that characterizes the stencil of dependency between time steps. For example, if in moving a computational cell forward from time step t to time step t+1the cell depends on its left and right neighbors, then SIGMA would be 1.

In the context of a cache oblivious strategy an n-dimensional solution domain for a transient problem is considered as n+1 dimensional, with time as additional dimension. This n+1-dimensional solution space is recursively bisected, observing the data dependencies of the stencil characterized by SIGMA. The spatial domain is bisected as long as it is larger than that required to support K forward iterations. If the spatial domain is not divided, then the temporal domain is divided. The recursion terminates when the size of the temporal domain is 1. The recursion is structured such that the kernel will then be called on the resulting spatial domain for K steps. An implementation of this recursive function *walk* in the C language appears in Appendix A.

Putting it Together

A cache oblivious implementation consists of repeated top level calls to the recursive partitioning function *walk.* Each of these top level calls moves the entire domain forward by K time steps. The leaves of the recursive decomposition call the computational kernel at each point of the spatial domain and for each time step.

The choice of the number of time steps K in the top level calls to *walk* affects the performance, because larger values of K increase the temporal locality created by procedure *walk.* It is implied that the size of each time step is pre-determined for each of the K steps. Should the size of the time step be adapted as the problem evolves during those K steps, then the solution may become unstable due to violation of Courant or other limits. In the extreme case, when K is reduced to 1, the benefits of cache oblivious strategies are negated because each cell is advanced one step during each sweep across the entire domain.

Accommodating Multiple Time States

The cache oblivious implementation requires different subdomains of a solution exist at different time states at the same stage of a simulation. Since all but the simplest stencils require information from the previous time step, care must be taken to provide access to the proper temporal and spatial data to evolve the solution.

Two data structures, toggle arrays and boundary passing variables, are often used to store the state of the computational domain. Here, toggle arrays were used in all programs. This approach duplicates the time dependent data for the entire domain so there is always a copy of the appropriate data available. It has the significant advantage of being simple to implement, although it increases the storage requirements for temporally dependent quantities by a factor of two compared to using a few temporary variables to pass values across spatial boundaries.

3. Kernel Tuning

Leaf Coarsening

The spatial and temporal decomposition produced by procedure *walk* is independent of both the data layout chosen for the simulation and work carried out in the kernel. In some applications, where the kernel produces only a handful of floating point operations (flops), the amount of computation can be dominated by the overhead due to the recursive function calls of procedure *walk.* In this case the cache oblivious algorithm can be refined by "leaf coarsening." To that end the time cut of procedure *walk* is modified by introducing a parameter V (for Voodoo) such that the kernel is called if delt = 1 or if the size of the spatial domain is less than the V parameter, see Appendix A. The kernel is modified accordingly to handle multiple time steps.

If the V parameter increases beyond the size of the entire problem, the strategy is no longer cache oblivious and reduces to the ordinary iterative algorithm.

Loop Unrolling

A leaf coarsened kernel should contain sufficiently many flops to amortize the overhead of the recursion. In particular, the leaf coarsened kernel traverses a convex subdomain of the problem. Floating point performance of these kernels can be significantly enhanced by loop unrolling. We found that manual loop unrolling is generally superior to automatic compiler optimizations, which rarely produce efficient instruction schedules for the processors under investigation.

4. Test Kernels

3D Arbitrary Work Kernel

This kernel was concocted to execute 48 flops, including one division, during each evaluation. The calculation consists of a summation of "P" from all adjacent cells, an accumulation of "RT" from face neighbours, and a calculation of "T" from the result.

This kernel uses only a single centering of quantities. Toggle arrays were used to manage quantities from alternating time steps. The original kernel program was programmed carefully with high-performance in mind.

3D Arbitrary Work Kernel the "Wrong Way"

This kernel was identical to that above, but the calculation progressed over the three dimensional (i,j,k) data array in the "wrong" order so that stride-one data access was never performed. This orchestrated mistake illustrates the resilience of cache oblivious strategies when data are poorly arranged.

2D Lagrangian Hydrodynamics

A two dimensional Lagrangian hydrodynamics kernel was produced by consolidating the ALEGRA hydrodynamics advance time function call sequence into a single file and then greatly simplifying it. The resulting kernel supports a single ideal gas material and artificial viscosity but no longer supports hourglass control. The kernel contains three main loops. First a loop over elements to accumulate forces to nodes. Next a loop over nodes to calculate new accelerations, velocities, and positions, and finally a second loop over elements to update material properties prior to the next time step.

The kernel contains more than 1,000 flops. As a consequence, the recursion overhead due to procedure *walk* is negligible compared to the kernel compuation, and leaf coarsening is not necessary.

The ALEGRA code uses variables centered at nodes (acceleration, velocity, force) as well as at the elements (internal energy, pressure). The stencil for ALEGRA is such that an element at time t+1 depends on all adjacent elements and their nodes at time t. For the purposes of cache oblivious implementation this requires a SIGMA of 2. The storage of multiple state values is organized in toggle arrays with two time steps for all time dependent element and nodal quantities.

2D Lax-Wendroff

A Lax-Wendroff kernel was programmed to solve a simple advection equation on a rectilinear grid with cell centered quantities. This kernel contains only 9 flops, and was extensively tuned using leaf coarsening and loop unrolling. Multiple time step quantities were managed using toggle arrays.

5. Results

We report millions of floating point operations per second (mflop/s) for all kernels described in the previous section. Measurements were performed on an Intel Xeon processor running at 2.8Ghz, an AMD Opteron Processor running at 2.1Ghz, and an IBM Power5 running at 1.66GHz. The Xeon and Power5 programs were compiled with gcc -O3 and on the Opteron we used the pgi compiler with -O3. The results tabulated below illustrate the range of performance improvements that can be expected from cache oblivious formulations. Performance varies significantly depending on the compiler/processor pair. The results also help to explain the causes of the variation in the performance of cache oblivious traversals. In short, if the kernel performance is too low, the program is not memory bound, and cache oblivious traversals cannot improve performance. In contrast, if the program is memory bound (which implies that the kernel performance is high), then we observe significant performance improvements with cache oblivious traversals.

3D Arbitrary Work Kernel

The performance of the arbitrary work kernel was examined by reducing the problem size until it would fit within the L1-cache. On the Opteron this in-cache problem producs 920 mflop/s, on the Xeon 801 mflop/s, and on the Power5 1188 mflop/s. Given the data dependencies of the kernel computation, these performance numbers are reasonable, although they are only in the range of 50% or less of processor peak. These performance numbers constitute an upper bound for larger problem sizes that do not fit into cache.

Table 1 reports mflop/s for measurments with an iterative traversal of the domain, and two cache oblivious versions, one without leaf coarsening (V=1) and the second with leaf coarsening (V=100). The fact that the performance of the iterative version drops by about 40% compared to the small kernel that fits into L1-cache indicates that the problem is memory bound rather than processor bound. Although the performance of the cache oblivious algorithm on large problems is somewhat lower than the in-cache performance, the cache oblivious algorithm is an improvement over the iterative implementation.

3D Arbitrary Work Kernel the "Wrong Way"

When the arbitrary work kernel problem was modified by requiring the memory access to abandon stride one, the performance of the non cache oblivious formulation was significantly degraded on all three processors. When cache oblivious traversal was turned on for this poor memory layout, performance on the Opteron returned to close to its best. This suggests that an application that has poor memory layout (perhaps allocated in an ad-hoc manner), but exhibits spatial data coherence can substantially benefit from cache oblivious strategies.

Table 1. Mflop/s for 3D Arbitrary Work Kernel, K= 8.

	Xeon	***Opteron***	***Power5***
Iterative/fits in L1	800	920	1188
Iterative	480	650	650
cache oblivious V=1	660	568	690
cache oblivious V=100	590	690	800
Wrong Way iterative	130	151	90
Wrong Way cache oblivious V=1	590	740	570
Wrong Way cache oblivious V=100	650	800	590

2D Lagrangian Hydrodynamics

The kernel performance of small in-cache problems on the Opteron is 810 mflop/s, on the Xeon 538 mflop/s, and on the Power5 with a peak performance of 6.6 Gflops, the kernel performance of 541 mflop/s constitutes only 8% of peak floating point performance. Consequently, we do not expect a performance gain from using a cache oblivious traversal for this application.

Mflop/s results comparing iterative with cache oblivious versions are shown in Table 2. The best performance on the Xeon was attained with the simple iterative version, without application of the cache oblivious traversal. On the Opteron and the Power5 the best performance resulted from the cache oblivious version with leaf coarsening (V=100). It should be noted that this kernel duplicates some computational effort on the boundaries of the computational subdomain. The number of floating point operations for the different traversals are included in Table 2.

Despite the lack of improvement on the Xeon and only modest gains on both the Opteron and the Power5, it should be noted that in creating this kernel the advance time step was simplified from several dozen cascading function calls to a single 700+ line kernel function. During this process, the performance of the advance time function increased by more than a factor of two.

Table 2. Mflop/s for 1,000,000 element 2D Lagrangian hydrodynamics problem with K=10.

	Xeon	***Opteron***	***Power5***
Iter/fits L1	538	810	541
Iterative 6.0×10^9 flops	463	605	494
CO V=1 7.3×10^9flops	428	690	520
CO V=100 6.8×10^9flops	446	739	539

2D Lax-Wendroff

Table 3 suggests the performance improvements possible when the cache oblivious formulation is paired with a tuned kernel. The Lax-Wendroff kernel exploits leaf coarsening as well as loop unrolling to marginalize the recursion overhead. For this kernel, the unoptimized performance for in-cache problem on the Xeon was extremely poor. After manual code tuning, performance improved significantly. On a 16 Million point problem the iterative traversal reduces performance to about 50% of the in-cache performance on the Opteron and to about 70% on the Xeon and Power5. The cache oblivious traversal restores the Opteron and Power5 performance surprisingly close to that of the in-cache performance. Thus, the cache oblivious version turns the program from being memory bound into one that is nearly independent of the memory.

Table 3. Mflop/s for 2D Lax-Wendroff solutions with K=100.

	Xeon	***Opteron***	***Power5***
Unoptimized Iterative fits in L1 Cache	70	520	420
Optimized Iterative fits in L1 Cache	1050	1470	1774
Optimized Iterative	758	747	1280
Optimized Cache Oblivious	559	1410	1621

6. Conclusions

Successful application of cache oblivious strategies for performance improvement depends on whether the kernel is processor-bound or memory-bound. If the floating point performance of the kernel is poor, the potential benefit of a cache oblivious strategy is reduced. If the computational kernel is efficient per se but its performance is limited by memory accesses, then a cache oblivious approach can significantly improve performance. Independently of the performance impact of a cache oblivious approach, reducing a calculation to a single computational kernel may already provide performance benefits. The impact cache oblivious strategies have on application performance is sensitive to the performance of the kernel computation, which depends on the code structure (leaf coarsened, loop unrolling, etc.) and the compiler/processor pair.

About the Authors

David Hensinger dmhensi@sandia.gov is a staff member at Sandia Nation Laboratories in the Computational Physics Research and Development group. P.O. Box 5800, Albuquerque NM, 87185-0378

Chris Luchini (cbluchi@sandia.gov) is the CEO of Sci Tac LLC a technology consultancy based in Los Alamos New Mexico, and a contractor for Sandia National Laboratories.

Volker Strumpen and Matteo Frigo are Research Staff Members at IBM's Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78758.

Appendix A: Cache Oblivious 2D Walk Function "C" Code with Leaf Coarsening for Toggle Arrays

```
void walk(int t0, int t1, cut * cuts, float *** tsa){
  const int SIGMA = 1;
  int delt = t1-t0;
  if((delt == 1) || (((cuts[0].x1 - cuts[0].x0) * (cuts[1].x1 - cuts[1].x0)) <
V)){
    basecase(t0,t1,cuts, tsa);
  }  else if (delt > 1){
       if  (2*(cuts[1].x1-cuts[1].x0)+(cuts[1].x1dot-cuts[1].x0dot)*delt  >=
4*SIGMA*delt){//dim1 cut
      cut save = cuts[1];
   int xm = (2*(save.x0+save.x1)+(2*SIGMA + save.x0dot + save.x1dot)
*delt)/4;
      cuts[1] = cut(save.x0,save.x0dot,xm,-SIGMA);walk(t0, t1, cuts, tsa);
      cuts[1] = cut(xm,-SIGMA,save.x1,save.x1dot);walk(t0, t1, cuts, tsa);
      cuts[1] = save;//restore configuration
     } else if(2*(cuts[0].x1-cuts[0].x0)+(cuts[0].x1dot-cuts[0].x0dot)*delt
>= 4*SIGMA*delt){//dim0 cut
      cut save = cuts[0];
            int  xm  =  (2*(save.x0+save.x1)+(2*SIGMA  +  save.x0dot  +
save.x1dot)*delt)/4;
      cuts[0] = cut(save.x0,save.x0dot,xm,-SIGMA);walk(t0, t1, cuts, tsa);
      cuts[0] = cut(xm,-SIGMA,save.x1,save.x1dot);walk(t0, t1, cuts, tsa);
      cuts[0] = save;//restore configuration
    } else { //cut time
      int s = delt/2;
      cut new_cuts[2];
      walk(t0, t0+s, cuts, tsa);
      for( int i = 0; i < 2; i ++){
        new_cuts[i] = cut(cuts[i].x0 + cuts[i].x0dot*s, cuts[i].x0dot,
                    cuts[i].x1 + cuts[i].x1dot*s, cuts[i].x1dot);
      }
      walk(t0+s,t1,new_cuts,tsa);
    }
  }
}
```

References:

1. Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache Oblivious Algorithms. In *Proc. 40th Ann. Symp. Foundations of Computer Science* (FOCS'99), New York, NY, October 1999.
2. Matteo Frigo, Volker Strumpen, Cache Oblivious Stencil Computations, *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05),* pp 361-366, Boston, MA, June 2005.