Time Skewing Made Simple

Robert Strzodka Mohammed Shaheen
Max Planck Institut Informatik, Saarbrücken, Germany
{strzodka,mshaheen}@mpi-inf.mpg.de

Dawid Pająk
Pomeranian University of Technology, Szczecin, Poland
dpajak@mpi-inf.mpg.de

Abstract

Time skewing and loop tiling has been known for a long time to be a highly beneficial acceleration technique for nested loops especially on bandwidth hungry multi-core processors, but it is little used in practice because efficient implementations utilize complicated code and simple or abstract ones show much smaller gains over naive nested loops. We break this dilemma with an essential time skewing scheme that is both compact and fast.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming

General Terms Algorithms, Performance

Keywords time skewing, loop tiling, temporal blocking, data locality, bandwidth, memory wall, memory bound, stencil

1. Introduction

Time skewing [12] is a successful technique to accelerate multiple iterations of a memory-bound stencil computation. If we do not exploit the temporal locality between iterations, then the performance is necessarily limited by the system bandwidth. In this case, Kamil et al. [7] show that after careful code optimization 0.49 Gkernels/s (giga kernel executions per sec) can be achieved with a Laplace stencil kernel (8 flops) on a $256^3 \times 100$ domain on a quad-core Xeon X5550. With time skewing, Wittmann et al. [11] achieve already 1.75 Gkernels/s on a similar system with a Jacobi kernel (6 flops) on a $600^3 \times 100$ domain. So there is a large benefit to this technique, however, the authors admit that the code used to obtain this result is complicated, specific to this type of kernel and multiple parameters had to be tuned manually.

Frigo and Strumpen [3] introduced a *cache oblivious* time skewing scheme that can be cast into few lines of code and adapts automatically to the memory hierarchy without the need for parameter tuning. However, practical implementations do require some adaptation to hardware and even then it is difficult to beat an optimized naive scheme in case of constant boundary conditions [6]. Strzodka et al. [10] offer a high performance implementation of cache oblivious stencils but at the expense of more complicated code.

If we want to exploit the power of time skewing without the programming complexity, then PluTo [2] is a great tool that offers easy-to-use source-to-source transformations. Given a source file it generates the optimized transformed code that can be compiled instead of the original source, so the complexity of time skewing and other transformations is hidden from the programmer. Other state-of-art tiling schemes for (partially) nested loops are HiTLoG [8], PrimeTile [5], and PTile [1]. Unfortunately, the abstraction has its price in performance as Fig. 1 demonstrates in comparison to PluTo, and the newest results of PTile being up to 30% faster on such kernels do not change the picture significantly.

So the current dilemma is that efficient time skewing implementations operate with complicated code and simple or abstract ones show much smaller gains over naive nested loops that are highly optimized by compilers. We resolve this problem by selecting only few time skewing and tiling transformations based on their impact on performance and create a compact time skewing scheme that maintains high performance.

2. Essential Time Skewing (EssTS)

We go back to the essence of time skewing and rather than outlining all possible options identify few crucial transformations that have the largest impact on performance. In this way, the transformed code is not much longer than the naive implementation but performs much better.

Alg. 1 shows the transformation of a naive iterative stencil computation in 3D to a time-skewed, tiled and parallelized scheme. The transformation can be applied to any stencil computation as long as the slopes of the stencil are known a-priori. It occurs in three steps.

- Time-skewing: the `t`-loop and the `z`-loop change order, thus access to the `z`-coordinate has to be offset by `s*t`, where `s` is the slope of the stencil in the `z`-direction.
- Tiling: The inner `t`,`y`-loops are tiled to enable in-cache processing. This introduces an outer loop over all tiles.
- Parallelization: The `tileSet` is split into disjoint subsets such that each thread executes one subset.

The parallelization step is responsible for a big complication of the code if locally dependent tiles are chosen. Wittmann et al. [11] employ thread parallelization in time, which introduces multiple delay parameters and gives asymmetric roles to different thread functions. Strzodka et al. [10] use a hierarchical thread distribution that requires complex load-balancing. Liu and Li [9] use simple dependent tiles and then either the parallelization has to move into the tiles restricting its impact or the synchronization has to be relaxed restricting the applicability of the scheme.

A largely overlooked solution that produces much simpler code is to use tiles that are locally independent. Clearly, large subsets of tiles cannot be completely independent, since the stencil computation couples all values in the domain. However, the space dimensions are large (otherwise the domain would fit into the cache), so it suffices to make them spatially independent, e.g. Frigo and Strumpen [4] use trapezoidal cuts. For highest code simplicity we recommend the diamond shape, because a diamond has exactly two

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

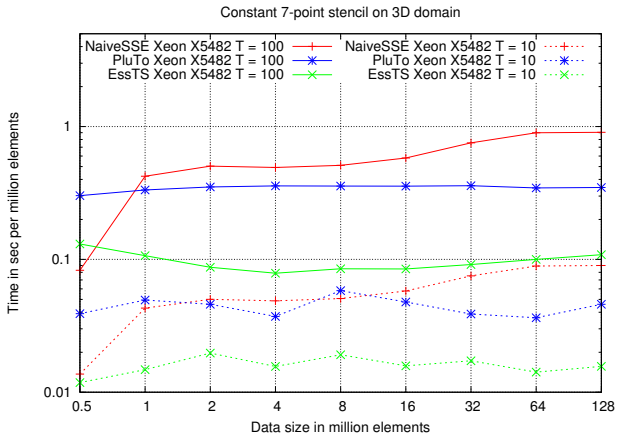

Figure 1. Performance of the Xeon X5482 on a constant 7-point stencil in 3D (13 flops). GFLOPS for 128 million elements with $T = 100$: NaiveSSE Xeon 1.4, PluTo Xeon 3.7, EssTS Xeon 13.

predecessors in time, so wait_on_dependencies() consists of two simple checks without the need to distinguish differently shaped trapezoids. In either case, it is easy to specify the tile boundaries in tstart(), tend(), ystart(), yend(). In the cache oblivious version tile sizes are determined through a series of hierarchical cuts, in the simplest version tile sizes correspond directly to the cache size available to each thread. In summary, the code from Alg. 1 concentrates in its simplicity on the most essential parts of an effective time skewing scheme and this suffices to obtain high performance.

Fig. 1 compares the vectorized naive scheme (NaiveSSE) with the automatic parallelizer and locality optimizer PluTo [2] and our essential time skewing code (EssTS) in the simplest variant. The timings have been obtained on a quad-core Xeon X5482 (Harpertown) 3.2 GHz system, benchmarked with 6.2 GB/s system and 64.2 GB/s L2 cache bandwidth (12 MiB) and 40.8 GFLOPS peak, running a 64-bit Linux and icpc-11.1 compiler with 4 pthreads in double precision. The naive scheme is only strong for the 0.5 mil. elements domain which is processed in-cache. PluTo compares well against the naive scheme, however, there is still much opportunity for improvement. EssTS performs on all domains similarly, as though the entire data could always fit into the cache.

3. Conclusions

Effective time skewing does not require complex code. With only three essential transformations we can turn a naive stencil computation into a parallel bandwidth optimized scheme of only slightly longer code but far superior performance.

Alg. 1 Essential time skewing example in 3D. Only few transformations are necessary to obtain much faster parallel C++ code from the naive implementation. Function stencil_SSE() contains the stencil computation vectorized along the x-axis from 0 to WIDTH. The remaining freedom lies in choosing the tile form and thus defining the tileSet, its dependencies in wait_on_dependencies() and the tile boundary functions tstart(), tend(), ystart(), yend().

```
void NaiveSSE_3D ()
{
  for(int t = 0; t < T; t++) {
    for(int z = 0; z < DEPTH; z++) {
      for(int y = 0; y < HEIGHT; y++) {
        stencil_SSE(t, z, y, 0, WIDTH);
      }//y
    }//z
  }//t
}

void EssTS_3D (int threadID)
{
  for( TileIt tile = tileSet[threadID].begin();
       tile != tileSet[threadID].end(); ++tile) {
  wait_on_dependencies(tile);

  for(int z = 0; z < DEPTH; z++) {
    for(int t = tstart(tile,z); t < tend(tile,z); t++) {
    for(int y = ystart(tile,z,t); y < yend(tile,z,t); y++) {
      stencil_SSE(t, z-s*t, y, 0, WIDTH);
    }}//t,y
  }//z

  }//tile
}
```

References

[1] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parametrized tiling revisited. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'10)*, 2010.

[2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[3] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 361–366. ACM, 2005.

[4] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

[5] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 147–157, 2009.

[6] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60. ACM, 2006.

[7] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[8] D. Kim, L. Renganarayanan, D. Rostron, S. V. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 51, 2007.

[9] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings ACM symposium on Principles and practice of parallel programming*, PPoPP '10, pages 213–222, 2010.

[10] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 49–59. ACM, 2010.

[11] M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *Proc. Workshop on Large-Scale Parallel Processing (LSPP'10) at IPDPS'10*, 2010.

[12] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2000.