Performance Portable Tracking of Evolving Surfaces [*]

Wei Yu[1], Franz Franchetti[1], James C Hoe[1], José M. F. Moura[1], Tsuhan Chen[2]
{wy,franzf,jhoe,moura}@ece.cmu.edu, tsuhan@ece.cornell.edu
[1] Carnegie Mellon University,[2] Cornell University

Introduction

Tracking the continuous evolution of surfaces such as a shock wavefront has a wide range of real world applications. The level set algorithm is a widely used tool for tracking evolving surfaces[4]. It embeds the surface into a higher dimensional function defined on a structural grid discretized volume, and performs numerical computation on the fixed Cartesian grid. The narrow band level set[4] is a variation of the level set method that can significantly reduce the computational cost without noticeable change in quality, by constraining computation to a neighborhood region around the interface which is called narrow band.

The narrow band level set algorithm performs a computation similar to iteratively solving partial differential equations with stencils, but on a irregular shaped and dynamically evolving narrow band. Usually, the interface motion is tracked for a large number of iterations, thus having the potential for a high data reuse rate. However, extracting data reuse on the highly irregular computational pattern is difficult. The major contribution of the paper is that we develop a framework to generate highly efficient code for this algorithm on mainstream CPUs with multicore, deep memory hierarchies and SIMD instructions.

Related work. This work is closely related to prior works on optimization of stencils [3, 2] and sparse linear algebra[5, 6]. Prior work on stencils has been primarily focusing on the dense structural grid. Sparse matrix solvers are usually memory-bound because of the low data reuse rate. We combined concepts from both areas, and develop a novel framework for the narrow band level set algorithm.

Surface Tracking Algorithm

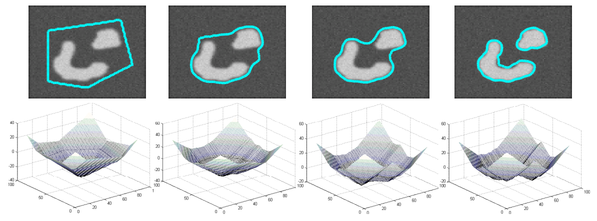

Figure 1: An example of level set evolution for image segmentation. First row shows evolution of the *zero level set*; second row shows evolution of the level set function.

We use a 2-D image segmentation example in Fig 1 to illustrate the surface tracking process by using the level set narrow band algorithm. The level set is a function ϕ defined on the 2-D image plane, whose *zero level set* corresponds to the evolving interface. The *zero level set* is the intersection of ϕ and the zero plane: $\{(y, x)|\phi(y, x) = 0\}$. The evolution of ϕ is driven by some force field such that at convergence, the *zero level set* forms a smooth contour on the object boundary. Computationally, updating the level set evolution function ϕ can be viewed as nearest-neighbor stencil computation. In this example, $\phi^{(t+1)}$ is a function of $\phi^t(x \pm \Delta_x, y \pm \Delta_y))$, where $\Delta_x, \Delta_y \in \{0, 1, 2\}$.

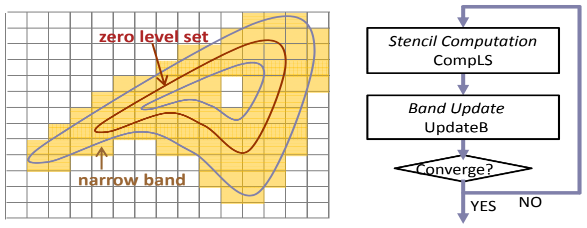

Figure 2: The narrow band level set and its algorithm flow.

In the level set method, what we are interested in is the evolution of the *zero level set* interface rather than the complete level set function. This leads to the lower complexity narrow band level set method in Fig 2, in which the computation is restricted to a narrow band around the zero level set. It is an iterative process of two basic components: 1) stencil computation for all points in the narrow band and 2) band update process based on the current level set.

Surface Tracking Framework

Our surface tracking framework thoroughly explores optimization opportunities of the narrow band level set algorithm, and integrates them in an auto-tuner to deliver performance portable code across different machines. The techniques proposed are grouped in the following six categories, addressing different aspects of the algorithm and hardware systems.

A fundamental algorithmic tradeoff. There is a fundamental tradeoff between the cost for stencil computation and the cost for band update. The tradeoff is controlled by band radius B_r and tile size. Using larger B_r or tile size reduces the band update cost, but incurs higher computational cost. These two parameters are chosen in the auto-tuning process.

In-Core stencil. In-core stencil explores efficient utilizations of on-chip resources. The major techniques include SIMDization, approximate transcendental functions, and optimizing instruction scheduling in the basic tile. For example, we approximate `cos` with a quadratic function that is simply vectorizable. We construct a search space of instruction schedules. Empirically we find that there is a 30%-70% performance difference between the best scheduling and the worst one.

[*]This work was supported by ITRI and ONR grant N000141110112.

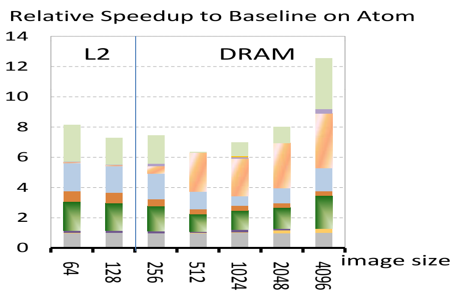

Figure 3: Single-threaded speed-up on Atom.

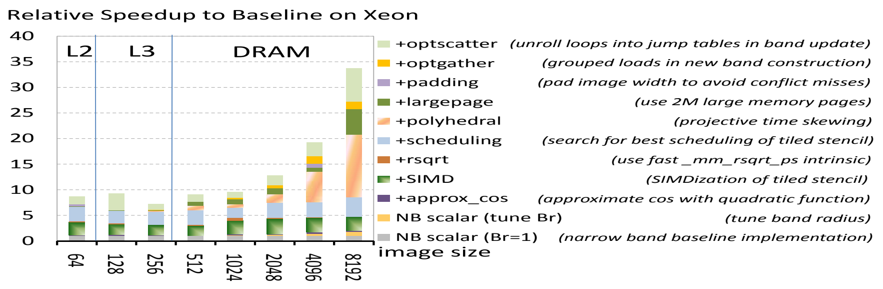

Figure 4: Single-threaded speed-up on Xeon.

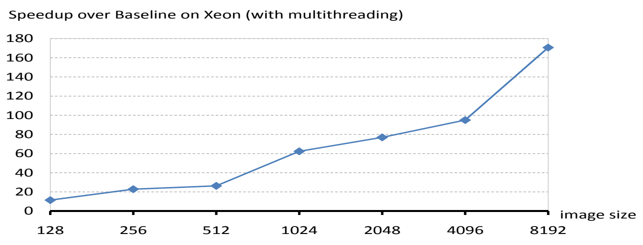

Figure 5: Full speedup on Xeon with multithreading (up to 8 cores).

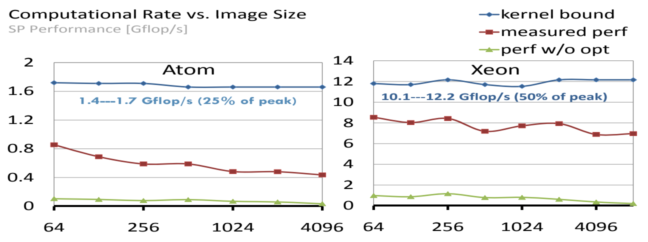

Figure 6: Computational rate and kernel stencil upper bound.

Memory. Memory level optimizations target removing the memory bottleneck by extracting the temporal reuse of data. Time skewing is a standard technique to remove the memory bottleneck for iterative stencil computation on dense stencils, by re-organizing the computational order. Directly applying time skewing transformation to the narrow band level set algorithm is not efficient, due to the high overhead incurred by the irregular shape and tracking process of the narrow band.

We propose a simple yet effective way to decompose the time skewing space and the narrow band part. The idea is to project a $n+1$-dim grid into n-dim, where each node in the n-dim space represents one sparse row in the original $n+1$-dim space. The continuous evolving interface which is sparse in the $n+1$-dim space now becomes dense in the n-dim space. The tracking process can be solved in a similar way as n-dim dense stencil computation, plus some special handling for the projected dimension and the narrow band tracking process. This technique can significantly reduce the memory pressure, or even make the application completely compute-bound.

Band update. The band update process has many short loops with unpredictable trip counts, due to the uncertainty in the updated band positions and interaction with tuning parameters like band radius and tile size. We build a big jump table using a code generator, and essentially unroll the short loops for each possible case.

Multithreading. We use a low overhead parallelization scheme when scaled to multiple cores. Naïvely partitioning the workload into independent sets will incur redundant computation along partition boundaries. We tailor the method in [7] to the narrow band setting, which has very low communication overhead and no redundancy for large data size. Performance scales almost linearly with the number of cores.

Auto-tuning. Now we have developed a parameterized framework that integrates all above techniques. The remaining problem is to identify good choice for our tunable parameters. On top of the framework, we build an auto-tuner that automates the search process for good parameter values, using some empirical search methods.

Experimental Results

Our test machines include a a Intel dual-socket 2.8 GHz Xeon 5560 and a 1.6 GHz Atom N270. Fig 3 and Fig 4 show the single-threaded speedup over a straight forward implementation baseline of the surface tracking algorithm, by incrementally adding one optimization technique a time. Our baseline runs at comparable efficiency to the best publicly available code [1]. On Xeon, single-threaded speedup ranges in 7x-34x for a wide range of image sizes. On Atom speedup ranges in 6x-13x. With multithreading, the speedup ranges in 11x-170x on Xeon, as shown in Fig 5.

We show the computational rate delivered in Fig 6. To understand the code efficiency, we setup an upper bound of the computational rate, which is the rate of dense stencil kernel after applying all in-core optimizations. The kernel runs at about 50% of peak on Xeon, and 25% of peak on Atom. The delivered computational rate is about 30%-40% of peak on Xeon, and 7%-13% of peak on Atom. The result indicates that the memory bottleneck becomes completely hidden on Xeon, but still has a penalty for large inputs on Atom, mainly due to the much smaller last level cache size and larger penalty for cache miss on Atom.

References

[1] http://www.engr.uconn.edu/ cmli/.

[2] K. Datta. Auto-tuning stencil codes for cache-based multicore platforms. *PhD thesis and University of California and Berkeley*, 2009.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *SuperComputing(SC)*, 2008.

[4] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry and Fluid Mechanics and Computer Vision and Material Science.* Cambridge University Press, 1999.

[5] R. Vuduc. Automatic performance tuning of sparse matrix kernels. *PhD thesis and University of California and Berkeley*, 2004.

[6] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *In Proc. SciDAC, J. Physics*, 2005.

[7] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. *International Parallel and Distributed Processing Symposium(IPDPS)*, 2000.