Highly Efficient Performance Portable Tracking of Evolving Surfaces

Wei Yu
Citadel Investment Group
Chicago, USA
Email: yuwei.emily@gmail.com

Franz Franchetti, James C. Hoe
ECE, Carnegie Mellon University
Pittsburgh, USA
Email: {franzf, jhoe}@ece.cmu.edu

Tsuhan Chen
ECE, Cornell University
Ithaca, USA
Email: tsuhan@ece.cornell.edu

***Abstract*—In this paper we present a framework to obtain highly efficient implementations for the narrow band level set method on commercial off-the-shelf (COTS) multicore CPU systems with a cache-based memory hierarchy such as Intel Xeon and Atom processors. The narrow-band level set algorithm tracks wave-fronts in discretized volumes (for instance, explosion shock waves), and is computationally very demanding. At the core of our optimization framework is a novel projection-based approach to enhance data locality and enable reuse for sparse surfaces in dense discretized volumes. The method reduces stencil operations on sparse and changing sets of pixels belonging to an evolving surface into dense stencil operations on meta-pixels in a lower-dimensional projection of the pixel space. These meta-pixels are then amenable to standard techniques like time tiling. However, the complexity introduced by ever-changing meta-pixels requires us to revisit and adapt all other necessary optimizations. We apply adapted versions of SIMDization, multi-threading, DAG scheduling for basic tiles, and specialization through code generation to extract maximum performance. The system is implemented as highly parameterized code skeleton that is auto-tuned and uses program generation.**

We evaluated our framework on a dual-socket 2.8 GHz Xeon 5560 and a 1.6 GHz Atom N270. Our single-core performance reaches 26%–35% of the machine peak on the Xeon, and 12%–20% on the Atom across a range of image sizes. We see up to 6.5x speedup on 8 cores of the dual-socket Xeon. For cache-resident sizes our code outperforms the best available third-party code (C pre-compiled into a DLL) by about 10x and for the largest out-of-cache sizes the speedup approaches around 200x. Experiments fully explain the high speedup numbers.

I. Introduction

Tracking the continuous evolution of surfaces such as shock wavefront or flame disturbance in the wind (so-called interfaces) has a wide range of applications in image processing, computer graphics, computational geometry, computational fluid mechanics, and many other fields. It is further used in image segmentation of noisy data (e.g., computed tomography), and in automatic target recognition. The level set algorithm is a widely used tool for tracking evolving interfaces, especially when the interface undergoes extreme topological changes like merging or splitting.

The *level set method* embeds the interface into a higher dimensional function defined on a regular grid discretizing the volume. It then evolves the level set function under a partial differential equation (PDE). The zero level set of a fix-point function under the PDE then corresponds to an interface while the time-dependent evolution of the function captures the movement of the interface. The core computational operation is applying a stencil at all grid points to iteratively solve the PDE: in each iteration the level set function gets updated on every grid point based on the values of the level set function on neighboring points in the previous iteration. The *narrow band level set algorithm* [1] restricts the computation to points in the neighborhood of the interface (the zero level set), which form a lower-dimensional (and thus sparse) subset of grid points. Since the function evolves, the zero level set evolves, and thus the narrow band around the zero level set must be updated frequently. In summary, the grid points to be computed upon change from iteration to iteration. The (narrow band) level set method is known to produce highly accurate results, but its major limitation in real applications is the slow processing speed [2]. Developing a framework to obtain highly optimized implementations that can fully utilize current multicore CPUs is an major step for applications that require robust and highly accurate real-time processing.

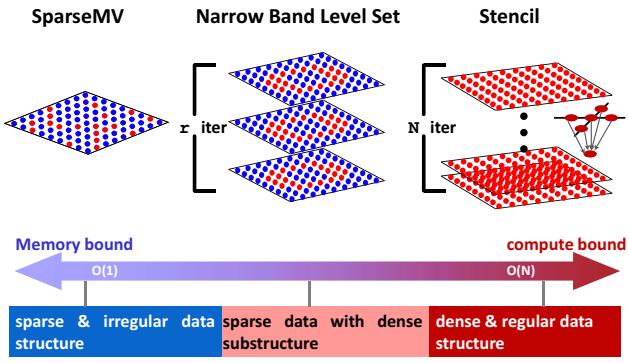


Figure 1. Arithmetic intensity of narrow band level set algorithm is in between sparse linear algebra and iterative stencil computation (view in color).

The main difficulty in obtaining highly efficient narrow band level set implementations comes from irregular data structures tracking the evolving region of interest. This irregularity prevents the direct application of well-known standard optimization techniques both by human program-

mers and by optimizing compilers. However, the algorithm has the potential for reuse and features data parallelism. Its arithmetic intensity lies halfway between sparse matrix-vector product and dense stencil computation, as depicted in Fig 1. Thus there is a chance to obtain better solutions than applying standard sparse linear algebra techniques. Unlocking this performance potential requires aggressive optimization and complicated code transformations. Further, the narrow-band level set method has a wide range of algorithmic options and application-specific formulations and tweaks that make it a sizable and important algorithm class that constitute a code pattern. In this paper we pick image segmentation as target application, however, our framework is designed to in principle support the whole class of narrow band level set applications, since the major difference across applications is the actual stencil operation.

Contributions. The main contribution of the paper is a framework to obtain highly efficient implementations of the narrow-band level set code pattern on commercial off the shelf (COTS) multicore CPU based systems.

The first and core contribution of our framework is a novel time skewing technique for surfaces embedded in discretized volumes which we call *projective time skewing.* With projective time skewing we achieve locality and reuse for an evolving sparse but contiguous set of grid points that describes a surface which evolves every time step. The method relies on the contiguousness of the surface and enables better performance than achievable with methods addressing stencil operations for general sparse grid point clouds.

The second contribution is the design of a parameterized code skeleton for the narrow band level set algorithm. To fully utilize modern multicore CPUs for this application, many optimization techniques need to be combined: approximation of transcendental functions, SIMDization, multi-threading, and low-level code optimization methods like instruction reordering (DAG scheduling) and unrolling. While all these techniques are well-known, their actual application in this particular algorithm is a hard problem. Since the evolving band introduces irregularity and unpredictability, traditionally orthogonal methods become dependent or inapplicable.

The third contribution is an autotuning system that provides performance portability for the narrow-band level set based image segmentation and achieves high performance on modern multicore CPU based systems. Our single-core performance reaches 26%–35% of the machine peak on a single core of an Intel Xeon and up to 6.5x speedup on 8 cores of a full dual-socket Quad-core Xeon system. For cache-resident sizes, our code outperforms the best available third-party code (C pre-compiled into a DLL) by about 10x and for the largest out-of-cache sizes the speedup approaches around 200x.

II. Narrow Band Level Set Algorithm

In this paper, we use 2-D image segmentation [3] as an illustrative example to explain the level set and its narrow band variation. However, our optimization framework is by no means restricted to this specific application. Other applications with a similar computational pattern but using a different stencil computational kernel or computing on a higher dimensional grid can be adapted to our framework.

Level set method. 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: $\{(y,x)|\phi(y,x)=0\}$. In this example, 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. Fig 2 shows the evolution process of the level set function ϕ. The lower row shows the evolution of ϕ,and the upper row shows the corresponding zero level set.

The force field that derives the evolution of ϕ is derived from minimizing the following energy function:

$$\int_\Omega \left(\frac{\lambda}{2}(|\nabla\phi|-1)^2) + g\delta(\phi)|\nabla\phi| + gH(-\phi)\right) dxdy \quad (1)$$

The terms in the energy definition attract the zero level set contour to be close to pixels of large gradient, regularizes the smoothness of the contour and regularizes ϕ to guarantee numerical stability. Here we are not going to detail the physical meaning of each term above. Interested readers may refer to [3] for more details.

Taking the derivative of Eq (1) gives the evolution function of the level set function ϕ.

$$\frac{\partial\phi}{\partial t} = \mu\left(\Delta\phi - \text{div}\left(\frac{\nabla\phi}{|\nabla\phi|}\right)\right) + \lambda\delta(\phi)\text{div}\left(g\frac{\nabla\phi}{|\nabla\phi|}\right) + \nu g\delta(\phi) \quad (2)$$

In real implementation, all terms in Eq (2) are computed using their numerical approximations. For example, first-order derivatives are estimated using a simple three-point estimation, like $\phi_x = (\phi(x+1,y) - \phi(x-1,y))/2$.

From a computational perspective, updating the level set evolution function following Eq (2) can be viewed as nearest-neighbor stencil computation. In this example,

$$\phi^{(t+1)} = \mathcal{F}(\phi^t(x \pm \Delta_x, y \pm \Delta_y)), \Delta_x, \Delta_y \in \{0,1,2\}$$

Narrow band level set. Essentially, the level set method tracks a n-dimensional propagating interface (zero level set) in $(n+1)$-dimensional space. For example, in the image segmentation case, we track a 1-D contour by evolving ϕ defined on the 2-D image plane. The computational complexity is $O(N^{n+1} \cdot T)$, assuming N is the length along each dimension and T is the total number of iterations.

In the level set method, what we are interested in is the evolution of the interface (zero level set) rather than

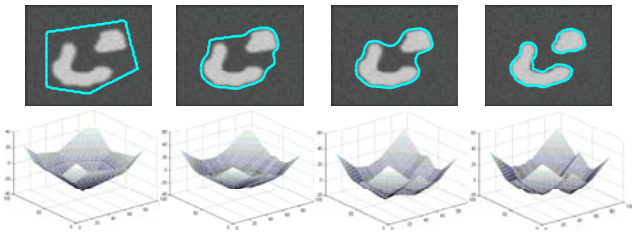


Figure 2. An example of level set evolution for image segmentation. First row shows evolution of the zero level set; second row shows evolution of the level set function.

the complete level set function. This leads to the lower complexity narrow band level set method, in which the computation is restricted to a narrow band around the zero level set.

The narrow band level set algorithm shown in Fig 3 has two basic components: 1) `CompLS` performs the stencil computation for all points in the narrow band following Eq (2), and 2) `UpdateB` rebuilds the band based on the current level set function ϕ. Conceptually, we need to re-detect the zero level set given the updated ϕ, and then construct a neighborhood region around the updated zero level set as the new band. More details of the computational model will be given in Section III.

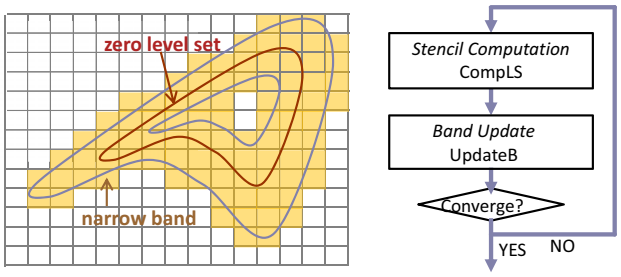


Figure 3. The narrow band level set and its algorithm flow.

III. Basic Implementation

In this section we describe a basic implementation of the narrow band level set method and discuss implementation choices like data structures, PDE discretization, and algebraic simplifications. As discussed in Section II, the narrow band level set has two major components: stencil computation `CompLS` and band update `UpdateB`. The narrow band is a set of sparse pixels in the image and can be represented using a data structure similar to CSR (Compressed Sparse Row) format in the sparse matrix solver.Following the sparse matrix approach, we tile the band using tile size $T_h \times T_w$. Tiles included in the band are called active tiles. Similar to tiling in the sparse matrix solver, choosing the appropriate tile size can lead to better instruction level parallelism (ILP), register reuse, and save storage for the indices. Given that band is represented in the tile granularity, the cost of both `CompLS` and `UpdateB` is closely related to the tile size.

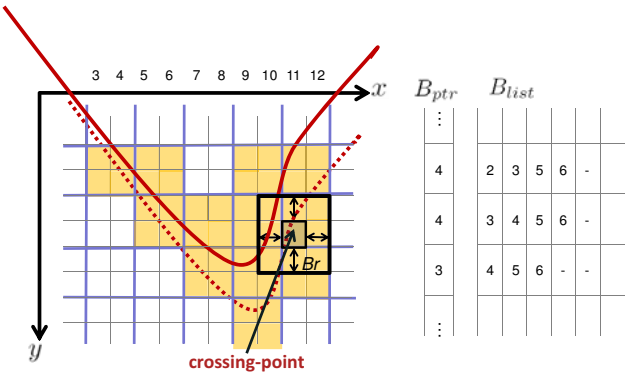


Figure 4. Illustration of the band update process and data structures. Solid and dashed lines are the old zero level set and the updated one. Band is tiled using 2×2 tile size.

Level set evolution time-step. In `CompLS`, we perform stencil computation on every pixel in the active tiles, following Eq (2). To compute Eq (2) at position (y, x), we need to first compute a normal vector N at its nearest neighbors $(y \pm 1, x \pm 1)$. The normal vector is computed as

$$\mathrm{N} = [\mathrm{Nx}, \mathrm{Ny}] = \frac{[\phi_x, \phi_y]}{\sqrt{\phi_x^2 + \phi_y^2}}. \tag{3}$$

Nx and Ny are x and y component of the normal vector N. Computing the normal vector is expensive. To save redundant computation, `CompLS` can be decomposed into two steps:

1) `CompN` step computes normal vector N=[Nx,Ny] for all pixels in the band following (3).
2) `CompL` step computes the updated level set function following (2).

After decomposition, `CompN` at (y, x) depends on ϕ at $(y \pm 1, x \pm 1)$, and `CompL` at (y, x) depends on ϕ and normal vector N at $(y \pm \Delta, x \pm \Delta)$, where $\Delta \in \{0, 1\}$. Evaluating (2) is expensive and involves the evaluation of $\cos(x)$. The whole stencil required to compute `CompLS` requires about 50 operations, including square root and cosine.

Update of the narrow band. In `UpdateB`, we need to check every point in the current band if it is a *crossing-point* using this condition:

$$\phi(y-1, x) \cdot \phi(y+1, x) \leq 0 \text{ or } \phi(y, x-1) \cdot \phi(y, x+1) \leq 0$$

The set of crossing-points form the new zero level set. Each crossing-point is expanded in four directions (up, down, left, right) of B_r pixels, whose union forms the updated band. This process is shown in Fig 4. To guarantee numerical stability, we need to do `UpdateB` once after every B_r iterations of stencil computation. This ensures

newly generated crossing-points will not overstep the current band.

The data structure used for the band maintenance includes three arrays: B_I, B_{ptr} and B_{list}. Assuming the image size is $h \times w$, B_I is a 2D `char` array of size $\frac{h \times w}{T_h \times T_w}$, with each element taking 0/1, indicating if the tile is in the updated band. B_{ptr} is a 1D `int` array of size $\frac{h}{T_h}$, and B_{list} is a 2D `int` array of size $\frac{h \times w}{T_h \times T_w}$. Tiles in the narrow band are recorded using B_{ptr} and B_{list}, in a way similar to the CSR format. As shown in Fig 4, $B_{list}[j][]$ records tile indices in each row, B_{ptr} tracks how many tiles there are in each row. Unlike the exact CSR, we do not re-organize ϕ value on the sparse band into a continuous array. This is because the band is dynamically evolving, and the cost of re-organizing data is too high compared to its benefit. Instead, ϕ, Nx and Ny are stored in separate 2D `float` arrays of size $h \times w$. `UpdateB` is also decomposed into two steps:

1) `scatter` step checks all points in the band. If a point is a crossing-point, then B_I is updated accordingly,by setting 1 for all tiles covered by the square neighborhood around the point. Here a tile is covered if at least one point in it is covered.
2) `gather` step rebuilds the new band by scanning B_I for entry of 1s, and updates B_{list} and B_{ptr} accordingly.

Algorithmic trade-off. There is a fundamental algorithmic tradeoff in the narrow band level set: the cost for `CompLS` and `UpdateB`. The tradeoff is controlled by band radius B_r and tile size $T_h \times T_w$. Increasing B_r leads to fewer band update passes because the band update is performed every B_r iterations, but higher computational cost because more pixels are computed. A similar relationship exists for the tile size: increasing the tile size reduces the cost of band updates because fewer tiles are needed to track the band, but the number of pixels in the band is increased. The best choice of B_r, T_h, and T_w depends on the optimization level of `CompLS` and `UpdateB` and is best determined experimentally, e.g., through autotuning. The complete computation and band update process is summarized in the pseudo code in Fig. 5.

```
//Computation part (CompLS)
for (int iter=0; iter<Br; iter++)
 for (int j=0; j<h/Th; j++)
  for (int k=0; k<B_ptr[j]; k++)
   {do CompN for tile at (j, B_list[j][k]).}

for (int iter=0; iter<Br; iter++)
 for (int j=0; j<h/Th; j++)
  for (int k=0; k<B_ptr[j]; k++)
   {do CompL for tile at (j, B_list[j][k]).}

//Band update part (UpdateB)
//scatter
for (int j=0; j<h/Th; j++)
 for (int k=0; k<B_ptr[j]; k++){
  int i = B_list[j][k];
  int y0 = j*Th, x0 = i*Tw;
  for (int y=y0; y<y0+Th; y++)
   for (int x=x0; x<x0+Tw; x++){
   //check crossing-point
   if(phi[y][x-1]*phi[y][x+1]<=0 ||
      phi[y-1][x]*phi[y+1][x]<=0){
    int y_l = (y-Br)/Th,  y_u = (y+Br)/Th;
    int x_l = (x-Br)/Tw,  x_u = (x+Br)/Tw;
    for (int ys=y_l; ys<=y_u; ys++)
     for (int xs=x_l; xs<=x_u; xs++)
      B_I[ys][xs]=1;
}}}

//gather
for (int j=0; j<h/Th; j++){
 int k=0;
 for (int i=0; i<w/Tw; i++){
  if (B_I[j][i]) B_list[j][k++] = i; }
 B_ptr[j] = k;
}
```

Figure 5. Pseudo code for the narrow band level set algorithm.

IV. In-Core Optimization

In this section we discuss the in-core optimization of the function evolution (`CompLS`) and band update step (`UpdateB`). The result of this section is highly efficient code for cache resident sizes, and we use the resulting code as kernels within memory optimization in Section V and multi-threading in Section VI.

A. In-core Stencil Optimizations

Achieving high in-core performance for narrow-band level set stencil operation requires register tiling and SIMDization, approximation of a transcendental function, and instruction ordering in the basic tile.

Register tiling and SIMDization. We use a small register tile of size $t_h \times t_w$ in which we completely unroll the code and all further in-core optimizations are performed on the register tiles. Multiple register tiles may be constituting a band tile of size size $T_h \times T_w$ used for tiling the band, i.e., t_w divides T_w and t_h divides T_h. SSE instructions are 4-way SIMD vector instructions and naturally imply a basic tile size that is a multiple of 1×4. Register tiles must contain multiple SIMD tiles, thus 4 must divide t_w. Since the register tile is built from multiple of the basic SIMD register tiles, we trivially obtain SSE code for the stencil computation part by taking the scalar stencil code and replacing all operations with the corresponding SSE instructions for every basic SIMD register tile. When computation depends on the data not aligned to the 4-pixel boundary (for example, when computing the first-order derivatives), we use SIMD shuffle instructions.

Approximate transcendental functions. In `CompL`, notice there is a $\delta(\phi)$ in Eq (2), which is the smoothed *dirichlet* function defined as

$$\delta_\epsilon(\phi) = \begin{cases} 0 & |\phi| > \epsilon \\ \frac{1}{2\epsilon}\left[1 + \cos(\frac{\pi\phi}{\epsilon})\right] & |\phi| < \epsilon \end{cases}$$

We approximate $\frac{1}{2}[1 + \cos(\pi x)]$ for $x \in [-1, 1]$ by $1 - x^2$ and thus replace a transcendental function costing hundreds

of cycles by an addition and multiplication. In `CompN`, we compute an approximation of $1/\sqrt{(u_x^2+u_y^2)}$ in Eq (3) using the fast but low precision `_mm_rsqrt_ps` (we do not use its result within the usual Newton Raphson iteration). Both approximations together yield slightly less accurate results per iteration (and thus more but cheaper iterations) but do improve the overall runtime and yield the same segmentation result.

Instruction ordering. The stencil that operates on every pixel in a tile performs the exactly same operation, and thus the same piece of code could be replicated for every pixel in a tile. We unroll the entire register tile and thus have a long basic block of $t_w \times t_h$ many stencil code fragments that are glued together. The whole block is implemented in single static assignment (SSA) style, as Spiral and FFTW's program generators successfully do [4], [5].

Ideally the C compiler will optimize instruction scheduling and register allocation of the basic block for the best performance, regardless of the order of input instructions. However, we observe a large variance on performance when using different sequences of instructions that perform the same computation. Our solution is to build a stencil tile code generator that generates a set of C instruction sequences implementing the basic block and empirically find the best one, performing autotuning or super-optimization on the C instruction sequence. We observed up to 70% performance difference between the best and the worst input instruction sequences.

Our tile instruction sequence generator is implemented in Perl and takes the instruction sequence in the form of a directed acyclic graph (DAG) of a single stencil operation and the tile size as input parameter. It tries multiple DAG schedules and two cross-pixel scheduling schemes. Fig 6 shows a simple example to illustrate these two schemes. In the *replicate* scheme it picks a instruction sequence and simply replicates it for every pixel. This saves register spills, and varying the tile size produces different instruction sequences. The second scheme is *interleaving* in which the first instruction for each pixel is executed, followed by the second instruction for every pixel and so forth. Each pixel has its own instruction sequence to get better instruction mix and instruction level parallelism (ILP). In both schemes we perform common subexpression elimination in the code generator. While this only covers a small subset of all schedules for a full tile, it is sufficient to contain code for which the C compiler can produce a good basic block.

B. Band Update Optimizations

The band update process has two distinct phases (scatter and gather). Both phases perform only a few operations per pixel and have very short loops with data-dependent unpredictable control flow in the innermost loop. This is the worst case scenario for out-of-order processors with speculative execution as well as simple in-order processors.

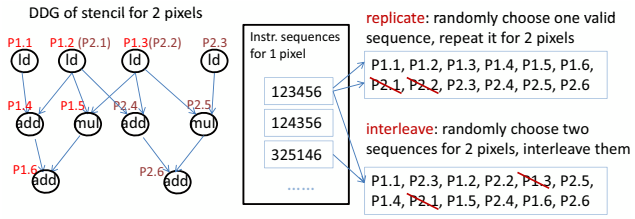
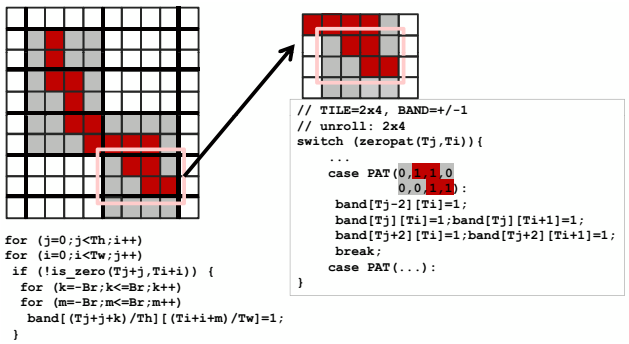


Figure 6. Illustration of replicate and interleave ordering schemes for an unrolled basic tile of 1×2 pixels. The two pixels use the same DAG ($Pi.j$ means the j-th instruction of pixel i). In the example, they share two inputs, $P1.2 = P2.1, P1.3 = P2.2$. We generate a set of instruction sequences for one pixel. The replicate scheme concatenates one sequence pixel by pixel; the interleave scheme mix two randomly chosen sequences of two pixels.

To remedy the problem we perform a specialized form of unrolling by generating a jump table that branches into automatically generated partially evaluated code snippets. This converts a short loop into hundreds of lines of code but substantially improves performance.



```
for (j=0;j<Th;i++)
for (i=0;i<Tw;j++)
 if (!is_zero(Tj+j,Ti+i)) {
  for (k=-Br;k<=Br;k++)
  for (m=-Br;m<=Br;m++)
   band[(Tj+j+k)/Th][(Ti+i+m)/Tw]=1;
 }
```

```
// TILE=2x4, BAND=+/-1
// unroll: 2x4
switch (zeropat(Tj,Ti)){
    ...
    case PAT(0,1,1,0
             0,0,1,1):
     band[Tj-2][Ti]=1;
     band[Tj][Ti]=1;band[Tj][Ti+1]=1;
     band[Tj+2][Ti]=1;band[Tj+2][Ti+1]=1;
     break;
    case PAT(...):
}
```

Figure 7. Code generation for band update.

Scatter optimization. The `Scatter` process searches for crossing-points of the level set function and updates B_I at the same time to mark up the neighborhood of crossing points to belong to the band. First we SIMDizes the search process and check a SIMD vector of pixels at a time. Using SSE instructions, we load the superpixels (SIMD vectors of pixels) up, down, left and right to the current pixel (`m_t`, `m_d`, `m_l`, `m_r`). Next a vector comparison produces an integer `pattern` in 0–15, where groups of 4 bits each indicate if a pixel is a crossing-point or not, using the following SSE instructions.

```
__m128 ud = _mm_xor_ps(m_u, m_d);
__m128 lr = _mm_xor_ps(m_l, m_r);
int pattern = _mm_movemask_ps(_mm_or_ps(lr, ud));
```

The next step is to update the membership indicator for pixels being in the narrow band (stored in B_I). A simple implementation would just loop over all pixels, check if they are a crossing point, and if so mark up all their neighbors

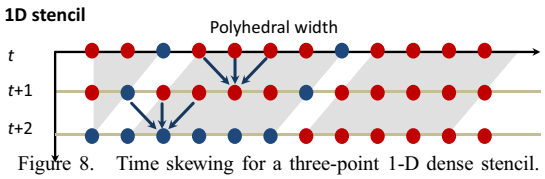


Figure 8. Time skewing for a three-point 1-D dense stencil.

in the band radius as being part of the narrow band (see the code in Fig 7 left). This is highly inefficient. However, there are $2^4 = 16$ possible patterns of crossing points in a 4-way vector and each of those results in a distinct pattern of updates that is known at compile time. We thus build a switch statement that implements all 16 versions as straight line code (unrolling all the small loops and pre-evaluating the if statement), and a switch statement that selects the correct code piece based on the vector result. An example for one zero-crossing pattern is shown in Fig 7, right.

Gather optimization. The `gather` builds up the new compact representation of the narrow band. It scans B_I for 1s, and collects the tile column indices into B_{list}. Performing SIMDization, we load 4 `chars` (data type `int`) or 16 `chars` (a SSE vector) each time, and build a big `switch` statement for each possible case of zero/one pattern the vector can hold (2^4 or 2^{16} cases, respectively). This is again a specialized form of unrolling requiring code generation and reduces load and branching overhead.

V. Projective Time Skewing

In this section we present our novel program transformation that provides reuse for stencil computation on a lower-dimensional surface that is embedded in a higher-dimensional regular grid. This is a special case of sparse stencil computation, since the lower-dimensional surface is a contiguous manifold. Further, the method is optimized for evolving surfaces, i.e., the overhead of pixels being included or dropping out of the surface is low. We first review 1D time skewing, and then explain how we translate a stencil computation on a 1D surface embedded in a 2D grid into a 1D meta-stencil operation. Higher-dimensional projections can be done analogously. Finally, we discuss lower-level memory optimizations that are required to achieve high performance on a cache-based multicore CPU system.

1D time skewing. Time skewing is a general loop optimization technique that can effectively remove the memory bandwidth bottleneck or remove data dependency in the inner-most loop for data parallelism [6], [7]. We show an example 1D 3-point stencil in Fig 8. Every grid point at time $t+1$ depends on itself and its both neighbors at time t. This produces few operations per data element and if done naively the whole data set is completely traversed at every time step. The result is a high memory bandwidth requirement and low arithmetic intensity, in particular if the whole data set does not fit into the cache.

Time skewing time-tiles the iteration and performs for a fixed set of grid points (4 in our example) multiple time steps (2 in our example) before moving to the next tile. The dependencies of the 3 point stencil (shown as arrows) require the tile to be rhomboid-shaped, and require a triangular initial tile. In this setup, no additional operations have to be performed. Intermediate results at every time step at the tile border need to be saved (blue grid points) so they can be reused by the neighboring tile or the next tile in the time direction. A tile is also called *polytope* and the generalization of time tiling and other similar loop transformations is formalized in the polyhedral framework [6], [7].

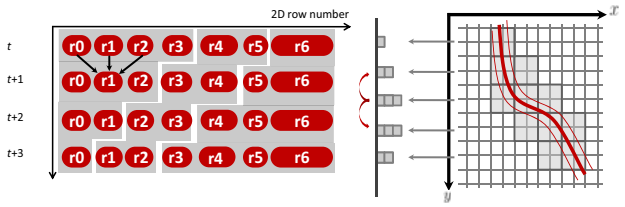


Figure 9. Projective time skewing for 1D surface embedded in a 2D image.

Projective time skewing. We now explain our projection from the sparse stencil operation on a 1D surface embedded in the 2D grid to a 1D dense meta-stencil operation. We assume a reasonably smooth 1D surface (curve), and a thin region of interest (the narrow band) around the curve is embedded into a regular 2D grid. The time direction is a third dimension, and in every time step on all points in the region of interest, a 2D stencil is applied. Further, we tile the 2D space in a way that fully covers the region of interest. This situation is depicted on the right side of Fig 9; the 2×2 tiles are depicted in grey and the curve/narrow band in red.

We observe that there usually are a small number of tiles per row needed to cover the narrow band, given a typical relatively smooth curve that is not exactly horizontal. We collect all the tiles needed to cover the narrow band and project them to the left. In our example this leads to 5 stacks (one per row) of 1, 2, 3, 3, and 2 tiles each, as shown in the middle of Fig 9. We make each stack of tiles a meta-grid point in 1D. Each meta-grid point collects all the tiles needed to cover the narrow band in its respective row. Thus, the meta-grid points are of varying size. Performing all stencil operations in one row in the 2D space becomes a single meta-pixel stencil operation in the projected 1D space, depicted by the red arrows pointing a row up and down.

Abstracting the fact that we are facing meta-grid points and a meta-stencil operation, we have reduced the special 2D sparse case to a 1D dense case. This allows us to apply 1D time skewing to the projected 1D dense stencil, as shown on the left side of Fig 9. The meta-grid points labeled r_0 to r_6 are collecting multiple tiles for rows 0 to 6, respectively, and thus are drawn with varying width. We depict the rhomboid

1D time tiles in grey again, and they mirror the standard 1D case, except for the varying width of the meta-grid points. A meta-pixel has many more operations than a standard 1D stencil, and the level set stencil already has more operations than a standard stencil, so a single tiling step, relatively small tiles, and few time steps per tile are sufficient to make the stencil on the region of interest compute bound. For instance the central tile consists only of 2 rows per time step. The first time step collects 3 2×2 tiles from row 4 and 2 2×2 tiles from row 5. The full tile includes 19 grid points in the 2D space.

Handling the evolving band. Now let us consider the complication introduced by band evolution specific to the narrow band level set method. The narrow band needs to get updated every B_r iterations, therefore we need to incorporate band updates in the projective time skewing. This is done by rebuilding the band as far as possible with currently available information. With ϕ known for rows $j \in [j_s, j_e)$, we have enough information to update B_I for row $j \in [j_s + 1, j_e - 1)$, and update B_{list} for rows $j \in [j_s + \Delta_s, j_e - \Delta_s)$, where $\Delta_s = \lfloor \frac{B_r - 1}{T_h} \rfloor + 2$. This is possible since a crossing-point in row j can only change B_I for rows $j \in [j - \lfloor \frac{B_r-1}{T_h} \rfloor - 1, j + \lfloor \frac{B_r-1}{T_h} \rfloor + 1]$. The content of sparse rows may change in the band update process, but continuity of the band evolution process guarantees that consecutive band positions have good overlap with each other.

Implementation details. The implementation overhead of the projective time skewing is low. All the index shifts in the projective time skewing can be pre-computed and defined as preprocessor macros. Below, P_h and P_w are the polytope height and width, and Δ_p is the row index shifts between polytopes and Δ_s is the band update index shift. Storage of B_I can be saved by using it in a cyclic way. Height of B_I can be reduced to $\Delta_p + P_w + 2\Delta_s$, which is the upper bound of the number of rows needed to be held at the same time.

With projective time skewing alone, we cannot completely hide the memory access penalty. Using large memory page size can almost completely remove the page miss penalty responsible for extra overhead. For some (2-power) image sizes appropriate padding is required. Projective time skewing together with these lower-level optimizations provides the ability to achieve near fully compute bound curve evolution. We summarize pseudo code for computing one polytope in the steady state in Fig 10.

Analysis. We now provide an analytical evidence that our approach is effective and for large enough cache size removes the memory bottleneck. We name all points in consecutive P_h iterations as one polyhedral segment. There are three segments: current segment, previous and next one. We assume perfect LRU cache replacement policy, and miss rate without polyhedral transform being M. There are two conditions under which cache miss rate is reduced: (I) when two consecutive polytopes can fit into cache, that is, $\Delta_p + 2P_w$ rows can fit, cache misses happen only on the polyhedral segment boundary, so total miss rate is roughly $\frac{M}{P_h}$; (II) when condition I is false and P_w rows can fit, cache miss happens to nodes on polytope boundary. So total miss rate is roughly $M(\frac{1}{P_h} + \frac{1}{P_w})$. Given fixed cache size and parameters B_r, T_h, T_w, if condition (I) holds, miss rate decreases as P_h increases. When condition (II) holds, a low miss rate corresponds to relatively large P_w and P_h. When both conditions do not hold, the miss rate is high as if there is no time skewing, and the cache is too small. Further, the degree of data reuse is roughly proportional to the cache size, and therefore, when the cache size is big enough, cache misses will no longer be the bottleneck. We observe both effects in our experiments detailed in Section IX.

```
int Bcnt=Br;
for (int iter=0; iter<Ph; iter++){
 for (int j=js; j<je; j++) {CompN for tiles in row j;}
 js--; je--;
 for (int j=js; j<je; j++) {CompL for tiles in row j;}
 js--; je--;

 Bcnt--;
 if (Bcnt==0){
  for (int j=js; j<je; j++) {
   scatter: check crossing-point and update B_I.}
  for (int j=js-Delta_s+1; j<je-Delta_s+1; j++) {
   gather: rebuild band by updating B_ptr and B_list.}
  Bcnt=Br; js = js-Delta_s+1; je = je-Delta_s+1;
 }
}
//update js and je for the next polytope
js = je + Delta_p; je = js + Pw;
```

Figure 10. Band update for one polytope in steady state.

VI. Multi-Threading

To scale our single-core implementation to multiple threads, we employ the idea of [8] given its low communication cost, and tailor the original method to the narrow band setting. Our tailored variant provides load balancing and is robust to non-uniform memory (NUMA) found in multi-socket multicore systems and scales well to the 8 cores of our dual quad-core Xeon system.

Software pipeline. Our method schedules time tiles across multiple processors (note that this are tiles in the projected space), breaks every tile into prologue, steady state, and epilogue, and implements a software pipeline. This is shown in Fig 11. For two cores, the dependency between tiles is between the tile prologue on processor 2 and the epilogue of the neighboring tile on processor 1. Given a big enough steady state and an execution begin of both tiles roughly at the same time, the prologue on processor 2 is guaranteed to be finished before the epilogue on processor 1 needs its data. This scheme needs very little synchronization and scales well to many cores.

Implementation details. In our level set code, each core has a private B_I array as in the single-threaded case, and an additional buffer B'_I for processing the epilogue. When core

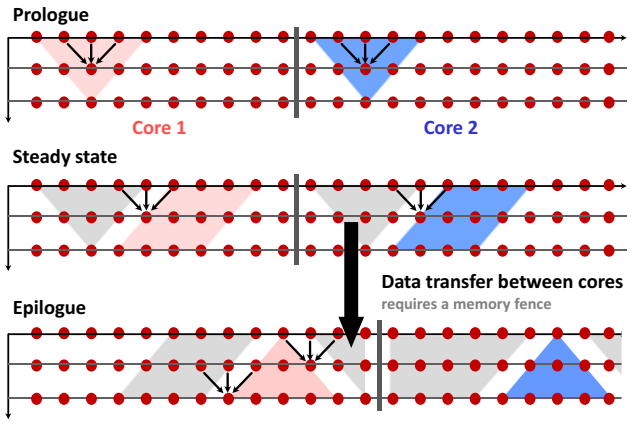


Figure 11. Parallelization on multicores Tiles are executed in a software pipeline that hides the latency of cross-core data transfer. Active computation is depicted in pink (core 1) and blue (core 2). The union of all pink shapes is core 1's tile, and all blue shapes constitute core 2's tile.

$n+1$ finishes its prologue, it will copy B_I information of shared rows to B'_I of core n. After this copying core $n+1$ signal its ready state to core n and then core n executes its epilogue. This scheme is repeated on all cores. To balance the workload among the cores during the evolution process, after each band update we adjust the partition to guarantee the number of active tiles assigned to each core is roughly the same.

VII. Autotuning and Code Generation

We developed a fully parameterized code framework that implements and combines all the optimization methods described so far. The remaining problem is to identify good parameter values that lead to high performance on a given machine. Some of these parameters need code generation to obtain specialized code fragments based on the parameters.

Autotuning. Using an autotuning approach allows us to find good parameters automatically and provides performance portability, i.e., good performance across a range of machines. Good parameterizations depend on both algorithmic choices and the target architectures. The search space of our parameters is summarized in Table I. The primary tuning parameters are B_r, T_h, T_w, P_h and P_w. Further options have only a Enable/Disable parameter or multiple choices (0, 1, and 2). Usually choosing Enable will not result in worse performance, however the parameters are handy to evaluate the impact of optimizations in isolation.

Since the search space is too big for exhaustive search, we use multiple iterations of the line search method. This standard search method in one step searches for the optimal choice of one parameter while fixing all other parameters, and within one iteration optimizes all parameters one at a time. Multiple iterations overcome the interdependence of

Table I
Summary of Optimizations and Tuning Parameters

Category	Optimization	Tunable parameter
Fundamental Tradeoff	opt. tradeoff between `CompLS` and `UpdateB`	$B_r \in (1..8)$, $T_h \in (1..8)$, $T_w \in (4, 8, 16)$
In-core	SIMD	Enable/Disable
	approx complex arithmetic	Enable/Disable
	instr. ordering	Enable/Disable
Memory	polyhedral transform	Enable/Disable, $P_h \in P_{hs}, P_w \in P_{ws}$
	padding	Enable/Disable
	large page	Enable/Disable
Band Update	optimized scatter	0 (naïve) 1 (vec search) 2 (scatter by pattern)
	optimized gather	0 (naïve) 1 (load 4 chars per time) 2 (load 16 chars per time)

note: $P_{hs} = (2, 4, 8, 16, 24, 32, 48, 64, 80, 96, 128, 160, 192)$, $P_{ws} = (2, 4, 8, 12, 16, 32)$

parameters and in our case we converge to good solutions usually after 2 to 3 iterations. We did not get stuck in local minima and thus did not investigate more powerful search methods.

Code generation. The autotuning requires the availability of code snippets for various parameter combinations, and these snippets need to be generated automatically as discussed in Section IV. Some parameters can be handled simple C preprocessor macros definition: for example, tile size $T_h \times T_w$, band radius B_r, polytope size P_h, P_w, and index displacements in the projective time skewing.

However, other parameters involve more complicated code variants, for which we developed code generators using the Perl scripting language to automatically generate these code parts. Beyond code specialization, our code generators implement simple compiler optimization techniques like common subexpression elimination and are used to generate instruction sequences in the long basic block in the in-core optimization, and to enumerate all possible cases in the big jump table in the band update optimization. Only a code generator provides the flexibility to handle combinations and many values of parameters as summarized in Table I. It would be very time consuming to provide a sufficient set of code blocks covering enough parameter combinations to perform reasonable autotuning.

Extension to other level set applications. Our code framework can be generalized for other surface tracking applications beyond image segmentation. The main difference is that they use different stencil kernels or work on higher dimensional data. Replacing a stencil kernel only requires extending or replacing the stencil scheduling code generator,

which is a localized change and easy to accomplish. Our system can be extended to higher dimensional data, however, this requires additional effort in the projective time skewing part of the framework which is currently specialized to two dimensions.

VIII. Related Work

This work targets an algorithm that is conceptually between sparse linear algebra and dense stencil computation. It takes inspiration from approaches in these two domains, and further adds ideas from autotuning and program generation into one integrated framework that is taking advantage of the specifics of the narrow band level set method. The unique properties of the targeted algorithm requires novel instantiations and adaptations of known concepts since the added complexity from the evolving band does not allow direct application of the well-known methods discussed below.

Stencils. Dense stencil computation on multicores and time skewing is investigated [9], [10]. Polyhedral compilers targeting imperfectly nested loops are discussed in [6], [7], and improve locality for dense stencils [8]. Dense methods are not directly applicable for the sparse narrow band in level set.

Sparse linear algebra. Optimizations for sparse linear algebra usually focused on maximizing bandwidth utilization due to the low data reuse [11], [12]. Sparse tiling [13], [14] and cache blocking [15] are techniques to improve locality for unstructured grid. In both methods, they assume a large number of iterations is performed on a stationary irregular mesh graph. This assumption does not hold for the evolving narrow band in the level set method.

Autotuning and program generation. Library generators such as ATLAS [16], FFTW [5], OSKI [12], Spiral [4] have proven extraordinarily effective at generating code that matches the performance of hand-written code. These systems employ both autotuning concepts and utilize various levels of program generation and domain-specific compilers but do not support the level set method.

IX. Performance Evaluation and Analysis

We examine the performance delivered by our framework on two Intel x86 multicore CPUs: a Intel dual-socket 2.8 GHz Xeon 5560 and a 1.6 GHz Atom N270. The two platforms represent two extremes on the power efficiency spectrum. They have a significant difference in the core microarchitecture, cache hierarchy, the number of hardware threads, peak flop rate and DRAM bandwidth, as summarized in Table II. All experiments are performed using the Intel C++ compiler 11.0 with best optimization flags.

Baseline code. Our baseline code for speedup is a straight-forward implementation of the pseudo code described in Section III. It is scalar single-threaded C code, with tile size 1×1 and $B_r = 1$. We turn on all compiler auto-optimizations including high level optimization, SIMDization and auto-parallelization when measuring our base line.

Table II
Summary of hardware platform features.

Processor	Intel Xeon 5560	Intel Atom N270
Microarchitecture	Nehalem	Atom
Type	superscalar OoO	in-order
Threads/core	2	2
Clock	2.8 GHz	1.6 GHz
Peak single precision	22.4 Gflop/s	6.4 Gflop/s
L1 D-cache	32 kB	32 kB
L2 (private)	256 kB	512 kB
L3 (shared)	8 MB	–
System	**Dell T410**	**Atom N270**
Cores/socket	4	1
Sockets	2	1
Peak single precision	179.2 Gflop/s	6.4 Gflop/s
DRAM size	12GB	1 GB
DRAM BW	63.98 GB/s	4.26 GB/s

Comparison to third-party code. The best publicly available code for comparison is the C code provided by Li. et at [3] in the form of a pre-compiled C program available as dynamic link library (DLL). Their code is close to a straight forward C implementation of the algorithm. While it is implemented in C the program is called through a Matlab interface. For the problem sizes we are considering, the invocation overhead of a C kernel from within Matlab should be negligible. Due to portability issues of the library file, we could only run their code on a Core 2 Extreme machine with 64-bit Windows Vista. We compared our C base line to their implementation on the same machine and measured a speedup of our scalar C code baseline over their code of about 1.3–2.0x when compiled using MS Visual Studio compiler 32-bit, and 2.0–3.0x when compiled with Intel C on 64-bit. This indicates that both their code and our base line code are of similar optimization grade, and our base line is comparable to the best available third-party code, both in implementation approaches and performance.

A. In-core Performance.

Here evaluate the performance of our in-core optimizations as described in Section IV. We use a square image of a size that is cache resident for the last level cache on the respective machine.

Stencil. In `CompN`, a superpixel of 1×4 pixels performs 4 vector `ADD` and 4 vector `MUL`, so theoretical peak is 1 cycle/pixel on Xeon, and 2 cycle/pixel on Atom. In `CompL`, a superpixel of 1×4 pixels performs 14 vector `ADD` and 13 vector `MUL`, so theoretical peak is 3.5 cycle/pixel on Xeon, and 6.75 cycle/pixel on Atom. Using the best tile size, the `CompLS` kernel code runs at 50% of 22.4 Gflop/s machine peak on Xeon and 25% of 6.5 Gflop/s machine

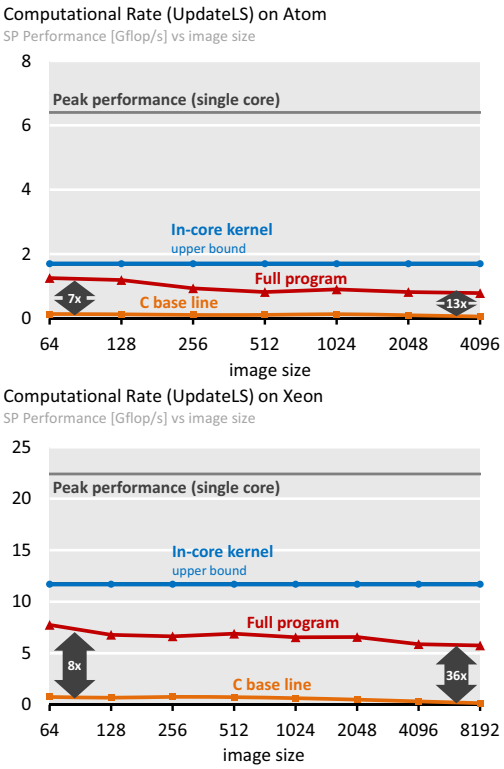


Figure 12. Computational rate of fully auto-tuned program on Atom and Xeon. Also show machine peak performance, stencil kernel performance, and baseline for comparison.

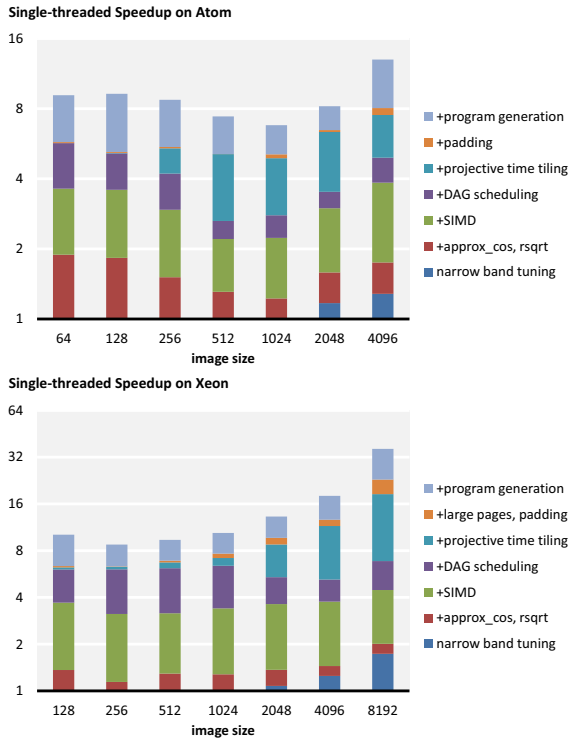


Figure 13. Speedup over scalar C code baseline on Atom and Xeon. Each point in the figure shows the performance with all optimizations so far. (Best view in color)

peak on Atom, which is good for such complicated stencil kernels. A higher fraction is delivered on Xeon because of its aggressive out-of-order core and more physical and architectural registers. This performance sets the upper limit for larger image sizes that do not fit into the cache.

Band update. It is difficult to characterize the code efficiency for `UpdateB`, because the main bodies of both `scatter` and `gather` consist of `switch` statements, which are highly unpredictable depending on the exact evolution pattern of the band. Here we report a lower bound by counting the total number of `switch` statement entries and memory writes, divided by CPU cycles. In `scatter`, executing an average `case` in the `switch` statement costs about 29 cycle, and writes to B_I takes about 13 cycle/write. In `gather`, the average `case` in the `switch` statement costs about 30 cycle/entry, and writes to B_{list} take about 23 cycle/write. The performance is most likely limited by the unpredictable control flow. On Atom, in `scatter` it takes on average 50 cycle per `case` and writes to B_I take about 10 cycle/write. In `gather`, it takes about 36 cycle/`case`, and writes to B_{list} takes 28 cycle/write.

B. Memory Optimization

Next we evaluate our memory optimizations including projective time skewing from Section V on a single core. We use square image sizes varying from 64 to 8,192 (the input data is generated by scaling the example image in Fig 2) on both machines. 8,192 is a large out-of-cache size that is too big to be held in memory addressable by the TLB on the Xeon.

Levelset evolution performance. Fig 12 shows the computational rate in Gflop/s measured for the computation part (`CompLS`), which is about 26%–35% of the machine peak on Xeon, and 12%–20% on Atom for varying input sizes. On Xeon, the performance remains almost flat when image size grows out of cache size. This means the application is close to compute bound. On Atom, the performance degrades as the input cannot be held in cache, indicating a considerable penalty from cache misses. This is because the L2 (512K) on Atom limits the search space of the polytope size, so that the memory bottleneck can only be partially hidden. On both machines, the gap between the kernel performance and the real application performance for small image sizes is mainly caused by the overhead of the indirect memory access using the CSR format. We tested the kernel code using the CSR format for the complete image, and got almost the same performance as in the real application.

Full program speedup. Fig 13 shows speedup of the best auto-tuned single-core code over our C base line. We show the relative speedup over our baseline code by adding the optimizations in the order of in-core, memory, and

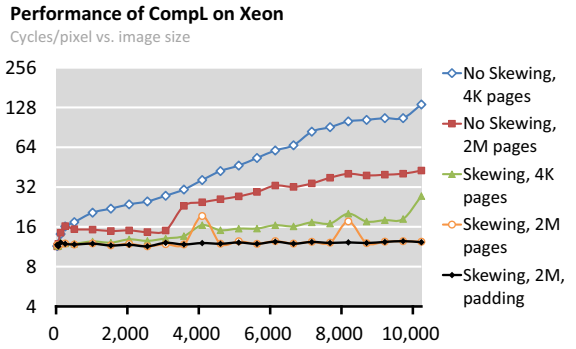


Figure 14. Performance (cycles/pixel) of `CompL` on fixed narrow band.

band update optimizations. For small image sizes fitting into the last level cache, in-core and band update optimizations are most effective, and for large image size, the memory level optimizations play an important role. We see a typical speedup around 8 on the Atom and on the Xeon for smaller sizes. Towards the larger sizes the memory optimizations show their impact and push the compound single core speedup to over 32 on the Xeon and around 14 on the Atom.

Projective time skewing. Fig 14 shows the result on Xeon assuming a fixed band. Performance severely decays without projective time skewing ("No Skewing"), and large pages only help moderately ("No Skewing, 2M pages"). After projective time skewing still, performance slowly decays as the image size grows, with a few bumps at certain image sizes ("Skewing, 4K pages"). With large pages ("Skewing, 2M pages") and additional padding ("Skewing, 2M, padding") the cycles/pixel finally performance stays stable across all image sizes, showing success.

Autotuning gain. To understand the benefits of autotuning, we compare against a reasonably good choice of the parameters: $T_h \times T_w = 2 \times 4, Br = 2, P_h = \min(\frac{w}{64}, 48)$, with all other optimizations enabled or set to the highest level wherever applicable. This is used as default across all problem sizes and on both machines. We compare these parameters to the best found separately for every problem size and machine. Autotuning on average improves performance by 12% on Xeon and 17% on Atom. The maximum gain is 25%on Xeon and 40% on the Atom. This shows that the most impact (factors of speedup) is derived from the careful development of the optimized code skeleton, and autotunig can further improve the performance to gain an extra edge (10s of % of speedup).

C. Multicore Parallelization Result

Next we evaluate the scaling across multiple cores on our multicore Xeon system. Fig 15 shows the speedup results; image sizes too small for parallelization are omitted. For very small image sizes, the synchronization overhead is relatively high, because each core has little work in the steady state. For medium image sizes, we observe close-to-linear speedup. Degradation happens when scaling from 4 cores (single-socket) to 8 cores (dual-socket) for large image sizes. This is because all L2 misses go through the main memory attached to the first socket, and this doubles the memory pressure compared to the single socket case. The compound speedup over our C baseline with all optimizations and multithreading on Xeon is also shown in Fig 15, ranging in 14–195x. The lower speedup numbers are obtained for small images and mainly due to in-core optimization, and the highest speedup numbers show the benefit of the largest sizes from both threading and projective time skewing on top of the in-core optimizations.

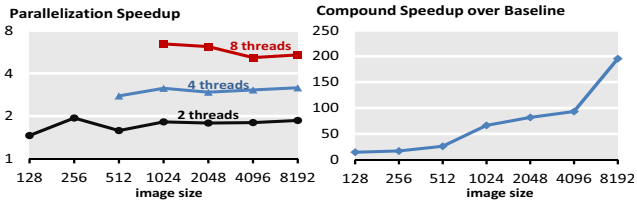


Figure 15. Parallelization (using 2, 4 and 8 cores) and compound speedup on Xeon.

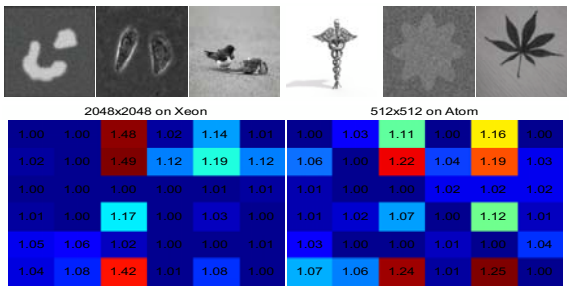


Figure 16. Sensitivity of tuning parameters to input images. The number in the j-th row i-th column shows the relative slowdown of applying the best tuning parameters of image i to image j compared to using the best tuning parameters of image j.

D. Sensitivity to Images

Level set based image segmentation is a highly data dependent algorithm, and as last experiment we evaluate the sensitivity of our tuning parameters to various images. In Fig 16, we shows how sensitive the performance is to six different images on both Xeon and Atom. For each image, we perform the autotuning process to find out the optimal parameter. Then we cross-test the optimal tuning parameters found for one by using it for another image. Let T_{ji} be the runtime when applying the parameters of image i to image j. The number in j-th row i-th column shows $\frac{T_{ji}}{T_{jj}}$, which is the slowdown for image j when using the (less than optimal) parameter from image i. We found that the maximum slowdown is 1.49, and average slowdown is 1.03x across all scales and both machines. We see clear evidence that in level set based image segmentation the

tuning parameters depend on the edge distribution of the image. For images of similar edge distributions like the first two images, the measured slowdown is within 13% on Xeon and 7% on Atom for all scales. The results suggest that in practice we can perform off-line tuning for a set of similar images to obtain performance portability within an image class.

X. Conclusion

Developing highly efficient computational code for modern multicore CPUs is a hard problem. Performance portable implementations that deliver high machine utilization across multiple machines usually require thorough exploration of application-dependent code optimizations, in conjunction with autotuning and program generation approaches. In this paper, we present a framework that delivers a highly efficient performance portable implementation of the narrow band level set method, which is an important tool for tracking evolving surfaces and in image segmentation.

To achieve high efficiency for this irregular algorithm based sparse stencil computation on an evolving set of pixels, we developed a projective time skewing technique to extract reuse for dynamically evolving contiguous lower-dimensional sub-sets of grid points of a regular dense grid. We further adapted standard optimization techniques to this dynamic use case and performed specialized in-core stencil optimization, lower level memory optimizations, band update optimization, and threading. We built an autotuning framework to find good parameterizations for our optimization techniques. For 2D image segmentation on a dual quad-core Xeon system, our fully optimized code shows between 10x and 200x speed-up over our C base line implementation, which has similar performance as the best third-party C implementation. Our code reaches 25%–35% of the machine peak performance across a wide range of problem sizes. This shows the effectiveness of our approach.

Acknowledgment

This work was supported by ONR grant N000141110112, NSF through award 0702386, Intel and the Industry Technology Research Institute Lab (ITRI) at Carnegie Mellon.

References

[1] 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.

[2] C. Li, R. Huang, Z. Ding, C. Gatenby, D. N. Metaxas, and J. C. Gore, "A level set method for image segmentation in the presence of intensity inhomogeneities with application to mri," *IEEE Trans. Image Process.*, vol. 20, no. 7, pp. 2007–2016, July 2011.

[3] C. Li, C. Xu, C. Gui, and M. D. Fox, "Level set evolution without re-initialization: A new variational formulation," *Proc. of Computer Vision and Pattern Recognition (CVPR)*, 2005.

[4] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005, special issue on *Program Generation, Optimization, and Adaptation*.

[5] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[7] C. Bastoul, "Code generation in the polyhedral model is easier than you think," *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

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

[9] 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.

[10] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.

[11] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, p. 2004, 2004.

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

[13] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *International Journal of High Performance Computing Applications*, vol. 18, pp. 95–113, 2004.

[14] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular gauss-seidel via sparse tiling," in *in 15th Workshop on Languages and Compilers for Parallel Computing (LCPC*, 2002.

[15] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, U. R. Ude, and C. Wei, "Cache optimization for structured and unstructured grid multigrid," *Elect. Trans. Numer. Anal*, vol. 10, pp. 21–40, 1999.

[16] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.