Performance Portable Tracking of Evolving Surfaces

Wei Yu

A Dissertation
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical and Computer Engineering
Advisers: Franz Franchetti, James C. Hoe

May 2011

Abstract

Our goal is to deliver high performance portable implementations across mainstream multicore machines for an important application: tracking evolving surfaces. Level set is a widely used numerical algorithm for tracking evolving surfaces, which embeds the surface in a regularly discretized volume and performs numerical computation in the volume. The narrow band level set algorithm is a variation of the level set method that reduces the computational cost by tracking the evolution in a narrow band, which is a neighborhood region around the evolving surface. Computationally, it performs numerical stencil computation on the points in the narrow band, and tracks the motion of the narrow band in the meantime. The narrow band level set algorithm is featured by abundant data parallelism and temporal data reuse. However, code optimization for the algorithm is complicated by the irregularity of the dynamically evolving sparse band and the data dependant control flow inherent in the algorithm.

We propose a novel code transformation for the narrow band level set algorithm, namely projective time skewing. This technique effectively extracts data reuse with low overhead. It is essentially different from applying the existing time skewing technique to the algorithm, and is much more efficient. In addition, we applied a set of other code transformations to fully utilize state-of-the-art multicore CPUs. These include in-core stencil optimization, lower level memory optimizations and parallelization, with focus on utilizing in-core resources, reducing memory access pressure, and achieving scalable performance across multicores. We incorporate all these optimizations in a parameterized framework, and build an auto-tuner on top of it to automatically search for good parameter values on different machines. The auto-tuning approach enables us to deliver performance portable code on different machines without manual re-optimization.

We did experiments on two Intel x86 cache-based multicore CPUs: a Intel dual-socket 2.8 GHz Xeon 5560 and a 1.6 GHz Atom N270. They exhibit significant variance in their core micro-architectures, DRAM bandwidth, peak flop rates, and the number of hardware threads. Fully tuned code shows up to 195x speedup over a straight forward C code implementation on the dual-quadcore Xeon system. The computational rate reaches 26% to 35% of the machine peak flop rate on Xeon, and 12% to 20% on Atom, across a wide range of problem sizes. The optimal parameters found by our auto-tuner averagely provide a performance gain of 12% on Xeon and 17% on Atom over an educated guess of good parameterizations.

Acknowledgements

I would like to thank my advisers, James C Hoe, Franz Franchetti, and Tsuhan Chen. Tsuhan was my advisor in the first half of my Phd life, and continues to support and advise me on research in the second half, when I work closely with James and Franz. I am very fortunate to have more than the usual number of advisors. All of them have always been very encouraging and supportive, to which I'd be ever thankful. They not only taught me how to become a better researcher, but also gave me wonderful guidance on shaping my mode of thinking, writing, presenting and communicating, from which I will benefit all my lifetime.

I also thank Markus Puesuel, whose incisive comments and broad knowledge have provided valuable insights on my research. He also set up a great example on how to improve the quality of writing and presentations. I appreciate that Marios Savvides and Ken Mai served on my qualifying exam committee as well, and gave me helpful suggestions on my early work on GPU.

Many thanks to my committee members David O'Hallaron and Nick Nystrom for their valuable feedback and comments, which have helped to improve the overall quality of my work and the thesis.

I would also like to thank Dr. Chunming Li for the enlightening discussions and suggestions on understanding the algorithms.

I feel very fortunate to get to know and work with members from three research groups: Ahmed Ashraf, Dhruv Batra, Kevin Chang, Andrew Gallagher, Zhaoyin Jia, Adarsh Kowdle, Hung-Chi Lai, Congcong Li, Edward Lin, David Liu, Mei-Hsuan Lu, Devi Parikh, Hyunjung Shim, Qi Wu, Wende Zhang, Yimeng Zhang, from Advanced Multimedia Processing group; Berkin Akin, Christos Angelopoulos, Volodymyr Arbatov, Christian Berger, Srinivas Chellappa, Tao Cui, Robert Koutsoyannis, Daniel McFarlin, Frédéric de Mesmay, Peter Milder, Marek Telgarsky, Qian Yu, from SPIRAL group; and Eric Chung, Yoongu Kim, Yongjun Jeon, Peter Klemperer, Eriko Nurvi-

tadhi, Michael PapaMichael, Stephen Somogyi, Evangelos Vlachos, Roland Wunderlich from the CALCM group. Thank you for the discussions and collaborations, as well as many pleasant memories we shared in group activities. I am especially thankful to Volodymyr Arbatov, Kevin Chang, Eric Chung, Daniel Macfarlin and Peter Milder for their hands-on help in my research, and many detailed suggestions in sharpening the quality of my work.

I would also like to thank my friends, many of them from CSSA and badminton club, Yun Gu, Fan Guo, Jinyuan Huang, Xiaoqian Jiang, Hongwen Kang, Lei Li, Qiao Li, Yanlin Li, Di Liu, Ni Lao, Bincheng Wang, Xiaohui Wang, Yiming Wang, Wanhong Xu, Hui Yang, Tianjun Ye, Jinyin Zhang, Xin Zhang, Zongwei Zhou and many more. We've spent a lot of fun times together playing games and having parties, making my life in the past five years much more joyous and colorful.

I thank the Department of Electrical and Computer Engineering and its staff, especially Elaine Lawrence, Reenie Kirby, Carolyn Patterson and Claire Bauerle.

Lastly, I would like to thank my family members. Thank you to my parents Xinxin Yu and Zhiran Zhan, my husband Weizhe An, for your continuous love and support, and my little one George An, for all the trouble and happiness you brought to us.

This work was supported by Industrial Technology Research Institute, and ONR (Office of Naval Research) grant N000141110112.

To my family.

Contents

List of Tables

List of Figures

List of Symbols

PDE	Partial Differential Equation
ATLAS	Automatically Tuned Linear Algebra Software
AVX	Advanced Vector Extensions
CSE	Common Sub-expression Elimination
CSR	Compressed Sparse Row
DDG	Data Dependency Graph
DRAM	Dynamic Random Access Memory
FLAME	Formal Linear Algebra Methods Environment
FP	Floating Point
FSB	Front Side Bus
GAC	Geodesic Active Contours
HT	Hyper-threading
ICC	Intel C/C++ Compiler
ILP	Instruction Level Parallelism
IR	Intensity Re-weighting
LBF	Local Binary Fitting
LLC	Last Level Cache
MPI	Message Passing Interface
NUMA	non-uniform memory access
OoO	Out-of-Order

OpenMP	Open Multi-Processing
OSKI	Optimized Sparse Kernel Interface
PETSc	Portable, Extensible Toolkit for Scientific Computation
Pthreads	POSIX threads
QPI	QuickPath Interconnect
SMT	Simultaneous Multithreading
SPARSEKIT	Sparse Matrix Utility Package
SPIRAL	Signal Processing Implementation Research for Adaptable Libraries

Chapter 1

Introduction

1.1 Tracking Evolving Surfaces

Tracking the continuous evolution of surfaces such as a 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. Usually the surfaces undergo deformations governed by some force field which can be modeled using PDEs (partial differential equations).

The level set is a powerful and widely used numerical method to track the complicated motion of 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 structural discretized grid volume, and performs numerical computation on the fixed grid instead of directly parameterizing the interface, which allows easy and robust handling of topological changes.

Fig 1.1 shows a simple example of a 2D evolving surface in a 3D regular data cube. In the level set method, the 3D data cube is discretized into a regular grid. A level set function is defined on the 3D cube, and the 2D evolving interface corresponds to the subset of points whose level set function values are zero. The level set tracks the

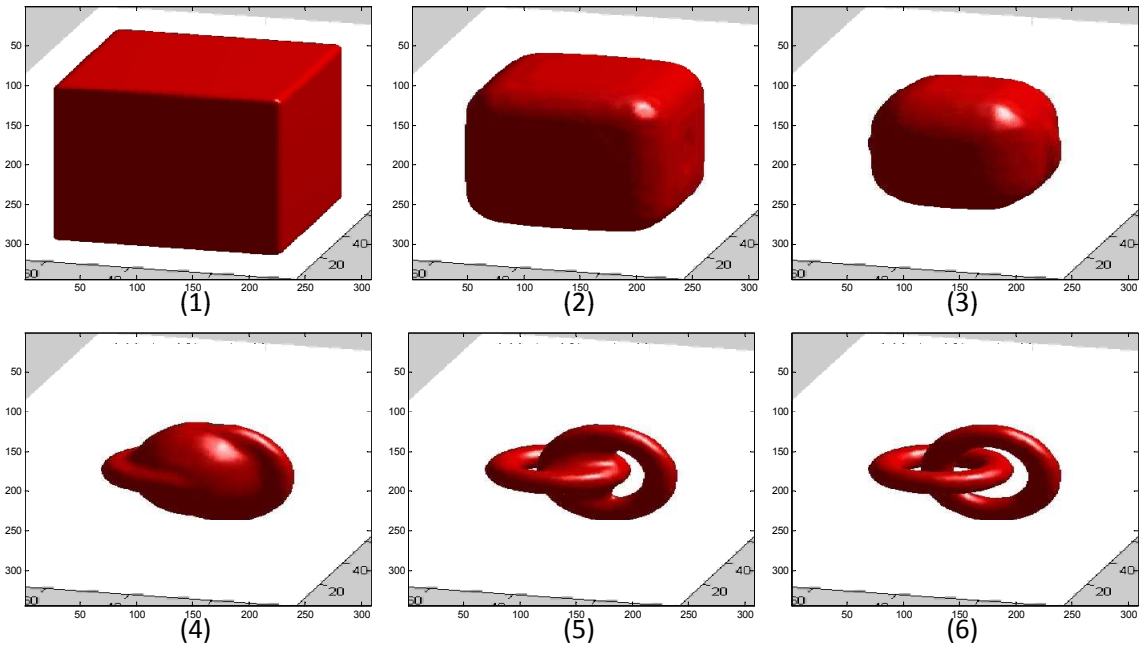

Figure 1.1: A simple synthesized example of an evolving surface in 3D cube, which converges at the boundary of two tori. (Images are extracted from a video created by Dr. Chunming Li)

motion of the evolving interface through an iterative process of updating the level set function value.

From a computational perspective, the level set algorithm performs a computation similar to iteratively solving PDEs with nearest neighbor stencil computation. 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. Performing computation on a fixed Cartesian grid is one important benefit of the level set method, but at the price of increasing computational cost.

The narrow band level set [44] is a variation of the level set method that can significantly reduce the computational cost without noticeable change in quality. The key observation is that the region of interest is the interface rather than the whole level set function. Therefore, the narrow band level set algorithm restricts the computation to points in the neighborhood region of the interface which are held inside a narrow

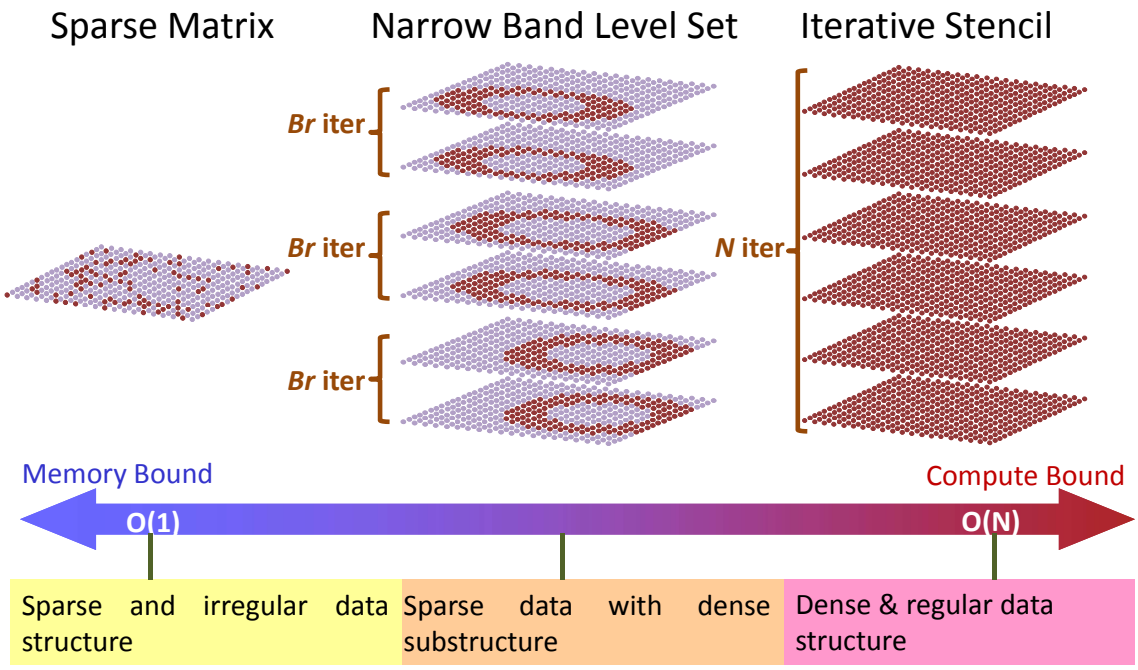

Figure 1.2: Arithmetic Intensity of the narrow band level set algorithm is in between sparse linear algebra and iterative stencil on dense grid. Red points are active grid points involved in computation. Note that the narrow band gradually evolves during the computational process, and gets updated every B_r iterations, where r is the narrow band radius.

band around the interface, and updates the band during the iterative computation process.

The goal of this work is to develop a system that enable efficient mapping of the narrow band level set algorithm to the mainstream multicore platforms. The narrow band level set algorithm is computationally intensive, and has abundant data parallelism and data reuse. In each iteration, the stencil computation for every point in the sparse band can be parallelized. Usually, the interface motion is tracked for a large number of iterations, thus having the potential for a high data reuse rate. Realizing the reuse is a challenging problem because the data is scattered on the continuously evolving narrow band, making code optimization on mainstream CPUs with multicores, deep memory hierarchies, diverse core micro-architectures and SIMD instructions a difficult task.

This work is closely related to prior works on optimization of stencils and sparse linear algebra. Sparse linear algebra usually has low data reuse, therefore it is likely to be memory-bound and the optimization effort focused on improving bandwidth utilization. Iterative stencil computation is highly data parallel with abundant reuse. Most prior works focused on stencil computation on the dense grid. The relationship of narrow band level set algorithm to the two well investigated areas are illustrated in Fig 1.2.

We will combine ideas developed for these prior works, as well as ideas from auto-tuning and code generation, to deliver highly efficient implementations of the narrow band level set algorithm on mainstream multicore platforms.

1.2 Performance Portability Challenges

The computer industry had been pushing the performance on single core chips through frequency-scaling and exploring instruction level parallelism, until single-threaded performance stopped to increase due to the heat density and power constraints in 2005. Since then, the microprocessor industry shifted to a new design paradigm of building power-efficient multicore platforms that have multiple simpler and lower frequency cores on a chip. This leads to massive thread-level parallelism and tremendous potential performance, but requires efficient parallel programming as well as extensive optimization on single-threaded performance to attain high machine utilization.

The goal of this work is to deliver performance *portable* implementations of tracking evolving surfaces. *Portability* means the ability to develop code once and achieve good performance on diverse multicore computers. Current general purpose compilers are not capable of automatically producing highly efficiency parallel code on multicore machines, even for very simple computational kernels [59, 40, 21]. The reason is that the complexity of current hardware and software is beyond what general purpose com-

pilers can handle. General purpose compilers perform well on handling basic blocks and simple loop transformations of independent data. But the tight budget in compilation time constrains the compilers to rely on simple heuristics instead of thorough experiments to find appropriate order and parameters of the code transformations. Furthermore, they usually lack the ability to identify complex but important domain-specific algorithmic transformations from software written in high-level languages like C/C++.

To achieve performance *portability* in front of the diversity and complexity of current multicore platforms, we adopted the auto-tuning approach, which allows us to achieve high performance code across different machines in a productive way. The auto-tuning method been successfully applied to many linear algebra kernels and numerical applications, including SPIRAL [40], FFTW [29, 28], ATLAS [55], Stencil [21, 22, 23], sparse linear algebra [53, 52, 57, 56].

To build an auto-tuning framework, we first need to thoroughly explore useful code transformations, and integrate them into a parameterized code framework. Auto-tuner is one layer on top of the parameterized framework, that automatically search for good parameter values on different machines using empirical methods.

1.3 Thesis Contributions

In this thesis, we propose a general recipe to generate high performance code for an important computational pattern: iterative stencil computation over a dynamically evolving region of interest, which is the neighborhood of a dynamically evolving sub-manifold inside a regularly discretized grid volume. This computational pattern is typical in the narrow band level set algorithm, which is a numerical method widely adopted for tracking evolving surfaces. We demonstrated the effectiveness of our methodology by going through an example of applying the narrow band level

set algorithm to the image segmentation problem. More specifically, our primary contributions include:

1. *A novel code transformation called projective time skewing.* This technique effectively improves locality when input data size cannot fit into cache. It is essentially different from applying the existing time skewing technique to the algorithm, and is much more efficient. 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 time skewing space and the narrow band manipulation (representation and tracking) are decoupled through the projection, which is the key for achieving high efficiency.

2. *A parameterized code framework and an autotuner to deliver performance portable code.* To deliver highly efficient code, we build a parameterized code framework that integrates the projective time skewing as well as a set of lower level optimizations to address different system bottlenecks. These optimizations include SIMDization, approximating transcendental functions, instruction ordering, using a code generator to unroll short loops with unpredictable trip counts into jump tables, and low overhead parallelization and synchronization on multicores. On top of this parameterized framework, we build an autotuner that can automatically search for good parameter values in the optimization parameter space. This enables performance *portability* of generating high performance code across different machines without the need for manual re-optimization.

3. *Others: Intensity Re-weighting Level Set Model.* We also propose a new energy model for the image segmentation problem: the intensity re-weighting level set model. This model allows fast and robust image segmentation when an image has inhomogeneity in regions.

1.4 Thesis Outline

The rest of the thesis is organized as following:

Chapter 2 gives an overview of the level set algorithm and its narrow band variation, along with an example of applying level set for the image segmentation problem. We also discuss various models that govern the evolution process in the level set method.

Chapter 3 reviews the two topics closely related to our work, optimizations of stencil computation on dense regular grid, and sparse linear algebra. These two topics are extensively investigated in the literature. We also discuss about the relationship and difference of our work with the prior work.

Chapter 4 describes the computational model used in the narrow band level set algorithm. We will formally present the computational flow of the algorithm, and detail the data structures.

Chapter 5 elaborates the cache-based multicore machines we performed experiments on. We also introduce important details in our experimental setup, including compilers, parallel programming models, and the performance evaluation metrics.

Chapter 6 introduces the full space of optimizations we explored for the computational pattern of the narrow band level set algorithm. The optimizations are introduced in the order of in-core optimization, memory-level optimization, band update optimization, and parallelization.

Chapter 7 explains how we incorporate all optimizations into a parameterized auto-tuner. We will detail the parameter search space, code generation, and empirical search strategies.

Chapter 8 discusses the performance results on the image segmentation example. We will show the speedup of auto-tuner generated code over a straight forward baseline implementation. We will also report the computational flop rate, and discuss architectural impacts on performance. The image segmentation example is represen-

tative of many other applications with a similar computational pattern. By understanding how optimizations and auto-tuning technique contribute to the performance results as well as the their limitations, we can learn how to apply these techniques to other applications.

Chapter 9 discusses future research directions, and Chapter 10 concludes our work.

Chapter 2

Level Set Algorithm

2.1 Level Set Overview

The level set is a widely used numerical method for tracking evolution of surfaces, for example, flame motion in the wind or a shock wave-front. The level set method was developed in the 1980s by the American mathematicians Stanley Osher and James Sethian [39, 44], and widely applied in image processing, computer graphics, computational geometry, computational fluid mechanics, and manufacturing of computer chips.

The level set method embeds the irregularly shaped surface into the so-called level set function, which is defined on a higher dimensional regular dense grid. Tracking evolution of the level set function is simply implemented by performing stencil computation on the fixed Cartesian grid, like in a partial differential equation (PDE) solver. The major advantage of the level set method is that it can easily handle topological changes such as splitting and merging. It has also been proven to be more accurate in handling complex shapes such as sharp corners and cusps than tracking surface motion in the lower dimension directly.

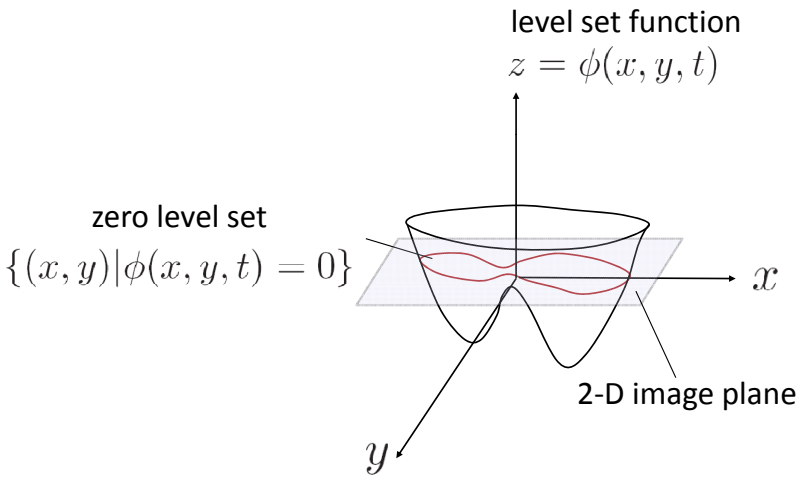

Figure 2.1: Illustration of the level set function defined on a 2-D image plane, and its corresponding *zero level set*.

In the thesis, we use an image segmentation problem as an illustrative example of how to apply the level set method to real world problems. This by no means restrains our work to the image processing application. In fact, the techniques developed for this problem are well applicable to other applications as long as the computational pattern is similar. For example, we can replace the stencil kernel in image segmentation with other stencil computational kernels, or work on higher dimensional data.

In the image segmentation example, 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: $\{(y, x)|\phi(y, x) = 0\}$, as illustrated in Fig 2.1. By tracking the evolution of the higher dimensional function ϕ instead of the interface itself, the level set method has the benefit of easy numerical implementation on fixed Cartesian grid as well as the capability to handle topological changes.

Ideally there is an one-to-one correspondence between the *zero level set* and the level set function ϕ [44]. ϕ should be a signed distance function, whose value characterizes the distance of a point to the boundary specified by the *zero level set*. The points inside the boundary have positive values, which gradually decrease to 0 as

approaching the boundary. The points outside the boundary has negative values. Having negative values for inside points and positive values for outside points is also valid. The signed distance function f has a property of $|\nabla f| = 1$. Fig 2.2 shows an example of the relationship.

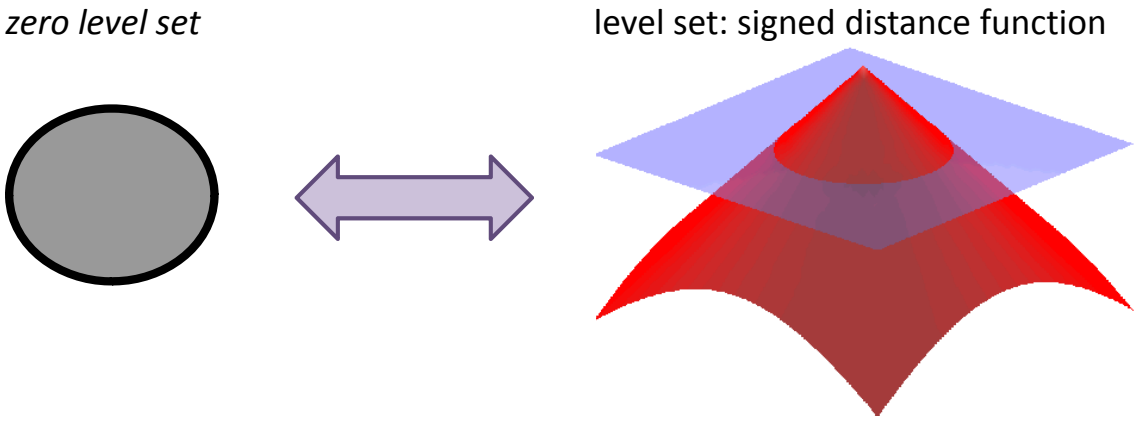

Figure 2.2: The level set function is a signed distance function of the *zero level set*. Figure is from [4].

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. There are many different choices in designing the force field, which will be discussed in more details in section 2.3. Here we briefly explain one edge-based model used in medical image segmentation [16].

The force field should attract the *zero level set* to evolve and converge on the object boundary. This is achieved by minimizing an energy target in the variational level set method. In [16], the energy term has three components:

1. The geodesic length term, which measures the weighted length of the *zero level set* in Euclidean space. The weighted curve length is defined as a function of the edge strength along the curve. The weight is large for smooth areas with weak edges and small for boundary areas with strong edges.

2. The balloon force term, which always drives the contour to expand or contract. This force can accelerate motion of the *zero level set*, but requires prior knowledge of whether the object is inside of outside the initial contour.

3. The regularization term, which regularizes ϕ to be close to signed distance function. This guarantees numerical stability without manual re-initialization to signed distance function during the evolution process.

The definition of the target energy function in [16] is

$$\int_{\Omega} \left(g\delta(\phi)|\nabla\phi| + gH(-\phi) + \frac{\lambda}{2}|\nabla\phi| - 1)^2 \right) dxdy. \tag{2.1}$$

g is the edge indicator function, which acts as the weight for curve length. $\int_{\Omega} (g\delta(\phi)|\nabla\phi|)\, dxdy$ is the geodesic length of the contour. $\int_{\Omega} (gH(-\phi))\, dxdy$ is the balloon force. And $\int_{\Omega} \left(\frac{\lambda}{2}|\nabla\phi| - 1)^2)\right) dxdy$ is the regularization term. More details of Eq 2.1 and other algorithmic design issues of the level set method can be found in section 2.3.

The evolution process of the level set function ϕ for the image segmentation problem is illustrated in Fig 2.3. The lower row shows the evolution of ϕ, and the upper row shows the corresponding *zero level set*.

Taking the derivative of Eq 2.1 gives the evolution function of the level set function ϕ.

$$\frac{\partial\phi}{\partial t} = \mu \left(\Delta\phi - \text{div}\left(\frac{\nabla\phi}{|\nabla\phi|} \right) \right) + \lambda\delta(\phi)\text{div}\left(g\frac{\nabla\phi}{|\nabla\phi|} \right) + \nu g\delta(\phi) \tag{2.2}$$

In real implementation, all terms in Eq 2.2 are computed using their numerical approximations. For example, first-order derivatives are estimated using a simple three-point estimation, like $\phi_x = (\phi(x+1, y) - \phi(x-1, y))/2$.

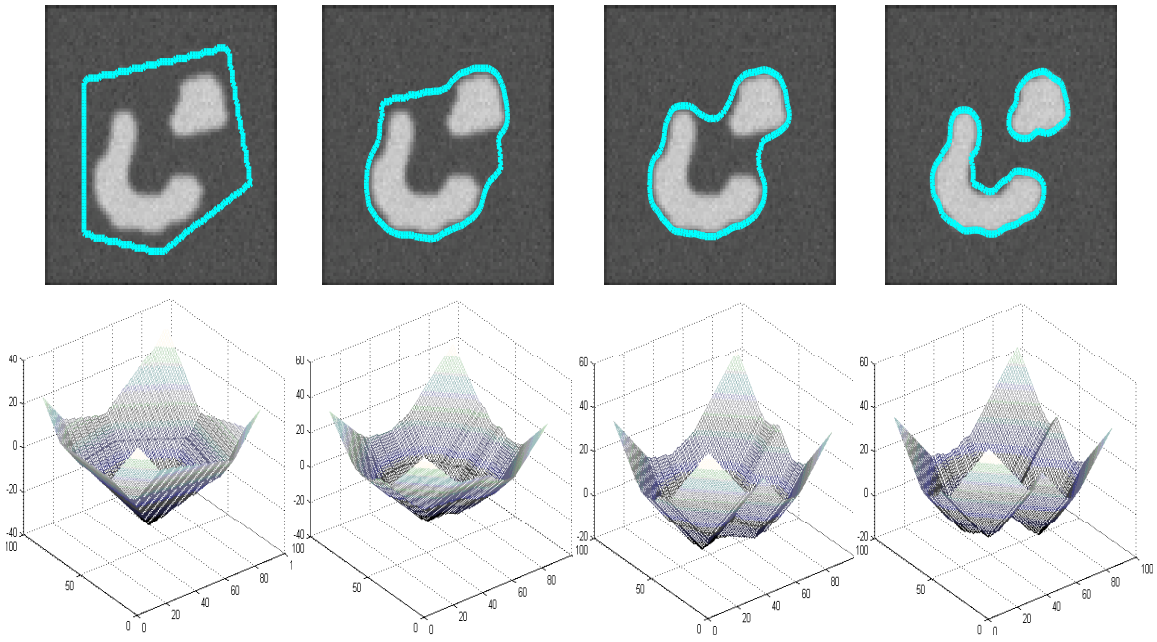

Figure 2.3: An example of level set evolution for image segmentation. First row shows evolution of the *zero level set*; second row shows evolution of the level set function.

From a computational perspective, updating the level set evolution function following Eq 2.2 can be viewed as nearest-neighbor stencil computation. In this example,

$$\phi^{(t+1)} = \mathcal{F}(\phi^t(x \pm \Delta_x, y \pm \Delta_y)), \Delta_x, \Delta_y \in \{0, 1, 2\}. \tag{2.3}$$

2.2 Narrow Band Level Set

Essentially, the level set method tracks a n-dim propagating interface (*zero level set*) in $(n+1)$-dim 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 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*, as illustrated in Fig 2.4.

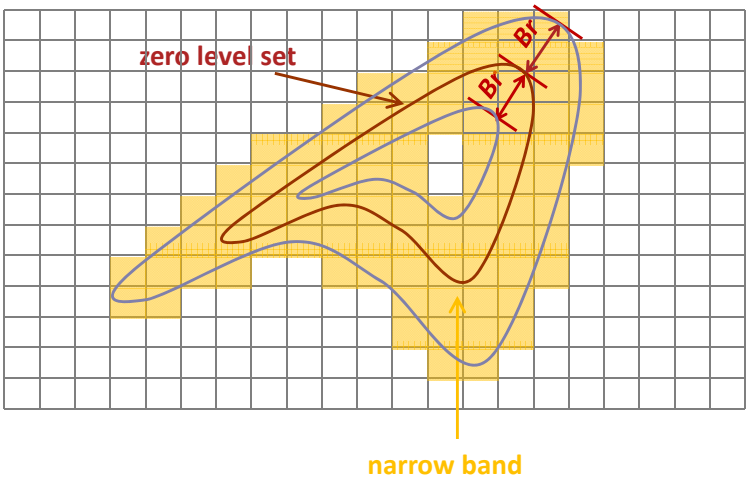

Figure 2.4: The narrow band is a neighborhood region of radius B_r around the *zero level set*.

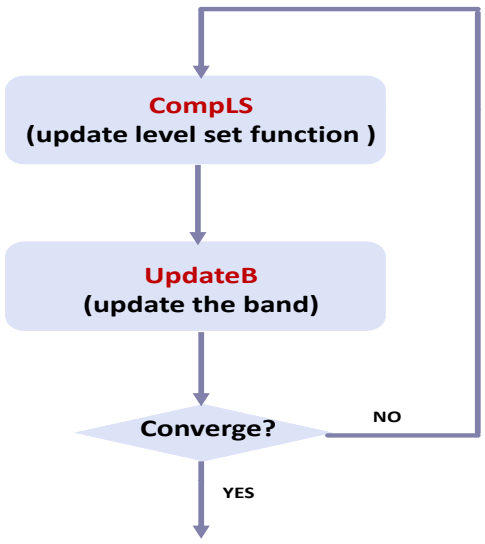

Figure 2.5: Algorithm flow of the narrow band level set: it repeats the `CompLS` and `UpdateB` till convergence.

The narrow band level set algorithm has two basic components: 1) `CompLS` performs the stencil computation for all points in the narrow band following Eq 2.2, and 2) `UpdateB` rebuilds the band based on the current level set function ϕ. It iterates over these two steps until the level set function value gets converged, as shown in Fig 2.5. 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 Chapter 4.

2.3 Level Set Algorithm Design Issues

There are many different ways to define the target energy. The definition in Eq 2.1 is one example of edge based level set that formulates the energy term based on edge information of the image. Different definitions involves intricate design considerations, including robustness to initialization, sensitivity to input data, computational complexity, etc. Given the complexity and diversity of real world images, there is no energy definition that can work well for all cases. Usually people define the energy according to specific properties of the application domain. Finding appropriate energy formulations is an active research topic in image processing and computer vision. We discuss about two mainstream categories of energy formulations that address different issues in the image segmentation problem: edge based and region based level set. We also discuss about some recent advances in level set models that can handle inhomogeneous regions.

2.3.1 Edge Based Level Set

In the edge based level set method, the active contour evolves according to some intrinsic geometric measure related to the edge strength of the image. Usually edge based models [49, 48, 41] have good localization property, but they are sensitive to

initialization and image content. A widely used edge based model is the geodesic active contours (GAC) model proposed by Caselles et al. [48]. The GAC model aims at finding a contour corresponding to the minimal curve length in a Riemannian space, whose distance metric is defined as a function of the edge strength. The length of the curve is a weighted version of the original curve length in Euclidean space, defined as

$$\mathcal{L}_g(\phi) = \int_\Omega g\delta(\phi)|\nabla\phi|dxdy. \tag{2.4}$$

g in the above equation is the *edge indicator function* defined as

$$g(x,y) = \frac{1}{1+|\nabla G_\sigma(x,y) * I(x,y)|^2}. \tag{2.5}$$

$\nabla G_\sigma * I$ is the convolution of the gradient of the Gaussian kernel and the input image, which is essentially the smoothed image gradient. g has small value for points where the gradient is large, and large value in smooth areas. δ in Eq 2.4 is the univariate Dirac function, which is close to 1 at *zero level set*, and close to 0 when getting further away from the *zero level set*. By minimizing the weighted length term $\mathcal{L}_g(\phi)$ in Eq 2.4, the contour is expected to converge at desired object boundary.

Since the level set is a numerical method, the contour advances a small step each time. When the contour position is far from the real object boundary, the contour has no ideas in which direction it should advance. This is why the edge based model is sensitive to initialization. Usually to guide the motion of contour in the smooth area, there is an additional balloon force term, that always drives the contour to expand or contract, depending on whether the initial contour is inside or outside the object. The balloon force term is defined as

$$\mathcal{A}_g(\phi) = \int_\Omega gH(-\phi)dxdy. \tag{2.6}$$

In Eq 2.6, $H(x)$ is the Heaviside function, which is integral of δ. It is close to 1 for positive x; and close to 0 for negative x.

Traditionally, the level set function needs to be re-initialized to be close to signed distance function shown in Fig 2.2, for numerical stability [44, 38, 34]. The traditional re-initialization process is performed in an *adhoc* manner and may result in undesirable effects. For example, the *zero level set* may be moved from the expected positions as a result of the regularization process [44, 38].

Li et al. proposed a better solution to regularize the level set function without the need of re-initialization [17, 16]. They integrate a term in the energy target in the variational level set, which penalizes the level set function when it deviates from the desired shape.

A simple penalty term was introduced by Li et al. in [16], which regularizes $\nabla\phi$ to be close to 1. It is defined as

$$\mathcal{P}(\phi) = \int_{\Omega} \frac{1}{2}(|\nabla\phi| - 1)^2 dxdy. \tag{2.7}$$

A more complicated penalty term is recently proposed in [17], which is an extension of the above idea. The complicated penalty term regularizes $\nabla\phi$ to be close to 1 around the *zero level set*, and close to 0 in smooth areas.

$$\mathcal{P}_2(\phi) = \begin{cases} \frac{1}{(2\pi)^2}\left(1 - \cos(2\pi|\nabla\phi|)\right) & \text{if } |\nabla\phi| \leq 1 \\ \frac{1}{2}(|\nabla\phi| - 1)^2 & \text{if } |\nabla\phi| > 1 \end{cases} \tag{2.8}$$

Incorporating the penalty term allows automatic adjustment of the level set function during the evolution process.

Combining the geodesic length term, the balloon force, and the penalty term, we obtained the energy target used in the edge-based level set method,

$$\mathcal{E}(\phi) = \mu\mathcal{P}(\phi) + \lambda\mathcal{L}_g(\phi) + \nu\mathcal{A}_g(\phi). \tag{2.9}$$

This is exactly the same as the energy target defined in Eq 2.1 when we introduced the level set method in Section 2.1.

2.3.2 Region Based Level Set

The region based level set algorithm assumes that the image should be segmented into different regions, where certain image properties like intensity are homogeneous or smoothly varying within each region, and change abruptly across the region boundary. It has two major advantages over the edge based model. First, it performs better on weak boundaries, because it does not rely on gradient information. Second, it is significantly more robust to the initial position of the contour. However, region based level set is usually much more expensive in computation because it iteratively collects regional statistics of certain image properties.

In 1989, Mumford and Shah proposed a generic model that serves as the basis for many practical region based segmentation models. The segments the image into disjoint regions, where in reach region the Mumford-Shah model assumes smoothly varying image intensity, and defines the energy target as

$$F^{MS}(u, C) = \int_{\Omega}(I-u)^2 dxdy + \mu\int_{\Omega\setminus C}|\nabla u|^2 dxdy + \lambda|C|. \tag{2.10}$$

$\Omega \subset \mathcal{R}^2$ is the 2-D image domain, and $I : \Omega \to \mathcal{R}$ is the image intensity. C is the contour that divides the image into disjoint regions. $|C|$ is the length of the contour. It is difficult to minimize the above energy term, because of the unknown contour position C and function u, as well as the non-convexity of the target function.

Later on, people developed many practical models based on the Mumford-Shah model. Two popular region based models are the piecewise constant model [20] and a more general piecewise smooth model [50]. The piecewise smooth model [20], also called Chan-Vese model, is a special case of Mumford-Shah model. It assumes constant image intensity in each region. The target energy in the Chan-Vese model is defined as

$$F^{PC}(C, c_1, c_2) = \mu \left(\int_{interior} (I - c_1)^2 dxdy + \int_{exterior} (I - c_2)^2 dxdy \right) + \lambda |C|. \quad (2.11)$$

Later Chan and Vese extend the above piecewise constant model to handle smoothly varying regions, and proposed a piecewise smooth model.

$$F^{PS}(I_1, I_2, \phi) = \sum_{i=1,2} \int_\Omega |\nabla I_i|^2 H_i(\phi) dxdy + \mu \sum_{i=1,2} \int_\Omega |I_i - I|^2 H_i(\phi) dxdy + \lambda |C| \quad (2.12)$$

In the above equation, I_1 and I_2 are the piecewise smooth functions that approximate the original image. ϕ is the level set function. $H_1(\phi)$ is the Heaviside function, which is close to 1 when ϕ is positive, and close to 0 when ϕ is negative. $H_2(\phi)$ is defined as $1 - H_1(\phi)$. $H_i(\phi), (i = 1, 2)$ can be viewed as the probability estimate of a pixel belonging to region i. The optimization process for this model is much more complex. In each iteration, it requires solving a PDE for ϕ while fixing I_1 and I_2, and solving two PDEs for I_1 and I_2 while fixing ϕ.

2.3.3 Level Set Models Designed for Inhomogeneity

Localized region based model.

Recently, people have proposed several region based models with an emphasis on local image information [18, 19, 31]. This model can handle inhomogeneous intensity within each region, which overcomes the limitation of models that utilize global image statistics. Fig 2.6 shows a synthetic example of foreground and background with

heterogeneous intensity, where piecewise constant Chan-Vese model fails to obtain the desired segmentation.

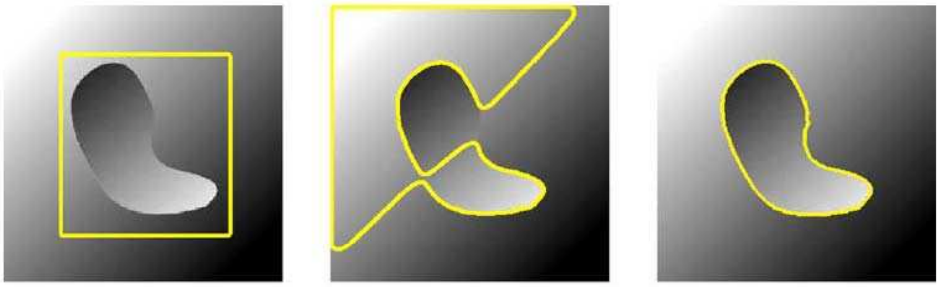

Figure 2.6: A synthetic example of a blob with heterogeneous intensity in foreground and background. (a) initial contour. (b). segmentation results using piecewise constant model [20]. (c) segmentation results using edge-based model or localized region based model. This figure is cited from [31].

Here we first briefly introduce a popular localized region based model, the local binary fitting (LBF) model proposed by Li. et al [18, 19]. Next, we discuss an extension of the LBF model we proposed, which is more robust to initialization and computationally more efficient [54].

LBF model.

In the LBF model, the energy term $\mathcal{E}^{LBF}$ is defined for each point $\mathbf{x}$ in the image as following

$$\mathcal{E}_{\mathbf{x}}^{LBF}(\phi, f_1(\mathbf{x}), f_2(\mathbf{x})) = \lambda_1 \int K_\sigma(\mathbf{y}-\mathbf{x})|I(\mathbf{y}) - f_1(\mathbf{x})|^2 H_1(\phi(\mathbf{y}))d\mathbf{y} + \lambda_2 \int K_\sigma(\mathbf{y}-\mathbf{x})|I(\mathbf{y}) - f_2(\mathbf{x})|^2 H_2(\phi(\mathbf{y}))d\mathbf{y}. \quad (2.13)$$

$\mathcal{E}_{\mathbf{x}}^{LBF}$ measures the summation of intensity variation within a local neighborhood of point $\mathbf{x}$ for two regions. $K_\sigma(\mathbf{y}-\mathbf{x})$ is a Gaussian kernel that diminishes with distance from $\mathbf{x}$. S_1 and S_2 refers to segment of $\phi > 0$ and $\phi < 0$ respectively. $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ are spatial fitting functions, capturing weighted average intensities of S_1

and S_2 from a local view of $\mathbf{x}$. $I(\mathbf{y})$ is the intensity at $\mathbf{y}$. $H_1(\phi(\mathbf{y}))$ and $H_2(\phi(\mathbf{y}))$ capture the probability of point $\mathbf{y}$ in S_1 and S_2. $H_1(\phi)$ is a Heaviside function.

$$H_1(\phi) = \frac{1}{2}\left(1 + \frac{2}{\pi}\arctan(\frac{\phi}{\epsilon})\right) \tag{2.14}$$

$$H_2(\phi) = 1 - H_1(\phi) \tag{2.15}$$

An intuitive explanation of minimizing $\mathcal{E}_{\mathbf{x}}^{LBF}$ is that $I(\mathbf{y})$ should be close to $f_1(\mathbf{x})$ if $H_1(\phi(\mathbf{y}))$ is high, and close to $f_2(\mathbf{x})$ otherwise. The complete LBF energy is defined as

$$\mathcal{E}^{LBF}(\phi, f_1, f_2) = \int_{\mathbf{x}} \mathcal{E}_{\mathbf{x}}^{LBF}(\phi, f_1(\mathbf{x}), f_2(\mathbf{x}))\, d\mathbf{x}. \tag{2.16}$$

Keeping ϕ fixed, minimizing $\mathcal{E}^{LBF}$ with respect to $f_1(\mathbf{x}), f_2(\mathbf{x})$ gives

$$f_i(\mathbf{x}) = \frac{K_\sigma(\mathbf{x}) * [H_i(\phi(\mathbf{x}))I(\mathbf{x})]}{K_\sigma(\mathbf{x}) * [H_i(\phi(\mathbf{x})]}, i = 1, 2. \tag{2.17}$$

Keeping f_1 and f_2 fixed, minimizing $\mathcal{E}^{LBF}$ with respect to ϕ, we derive the gradient descent flow.

$$\frac{\partial \phi}{\partial t} = -\frac{\partial \mathcal{E}^{LBF}}{\partial \phi} = -\delta(\phi)(\lambda_1 e_1 - \lambda_2 e_2) \tag{2.18}$$

$$e_i(\mathbf{x}) = \int K_\sigma(\mathbf{x} - \mathbf{y})|I(\mathbf{x}) - f_i(\mathbf{y})|^2 d\mathbf{y} \tag{2.19}$$

In Eq 2.19, $e_i(\mathbf{x})$ measures the intensity coherence of $I(\mathbf{x})$ with the average intensity of S_i near $\mathbf{x}$. Larger e_i means less coherent. Eq 2.18 can be interpreted as increasing the belief of $\mathbf{x}$ in S_1 if $e_1 < e2$, and vice versa.

The complete energy definition has two extra regularization terms: $\mathcal{E} = \mathcal{E}_{\mathbf{x}}^{LBF} + \mu\mathcal{P}(\phi) + \nu\mathcal{L}(\phi)$. $\mathcal{P}(\phi)$ is the penalty term and $\mathcal{L}(\phi)$ is the geodesic length term, as in the edge based level set model in Eq 2.9.

Intensity re-weighting model.

The motivation of the intensity re-weighting model is that the LBF model can easily get stuck in local minima for some initializations. Fig 2.7 shows such an example. In the following, R_1, R_2 refer to the brighter and darker region in an image respectively. S_1, S_2 refer to two segmented areas: S_1 corresponds to the segment of $\phi > 0$, and S_2 corresponds to the segment of $\phi < 0$. The goal of LBF model is to evolve ϕ such that ϕ has different signs in R_1 and R_2 to minimize energy. For the example in Fig 2.7, the contour is initialized to be close to boundary B_r. After a few iterations, the contour is formed around boundary B_l to reflect the intensity contrast, and the contour around boundary B_r is attracted towards the true boundary. However, evolution of ϕ around B_l and B_r are independent in the first few iterations. This results $\phi < 0$ around B_l and $\phi > 0$ around B_r in R_1, and leads to region 1 separated into two at convergence. The reason is that the initial contour is far from B_l, leading to two ambiguous directions to evolve ϕ around B_l: either $\phi > 0$ for R_1 and $\phi < 0$ for R_2, or the other way round. Small perturbations will cause ϕ to evolve in one of the two directions, because having the same sign of ϕ along both sides of the true boundary is an unstable state of high energy. But in which direction ϕ evolves is hard to predict when the initial contour position is far away.

The goal of segmentation is to make S_i and R_i $(i = 1, 2)$ paired, either $S_1 = R_1, S_2 = R_2$ or $S_1 = R_2, S_2 = R_1$ at convergence. Without loss of generality, we assume the goal is $S_i = R_i$, i.e. $\phi > 0$ in the brighter region and $\phi < 0$ in the darker region at convergence. We propose to incorporate bias in the level set function evolution process by adjusting the intensity weight when computing f_1 and f_2 in Eq 2.17. $f_i(\mathbf{y})$ measures the average intensity of S_i around point $\mathbf{y}$. Given the assumption that R_1 is brighter than R_2 along the boundary, we can put higher weights on brighter pixels when computing f_1 and higher weights on darker pixels when

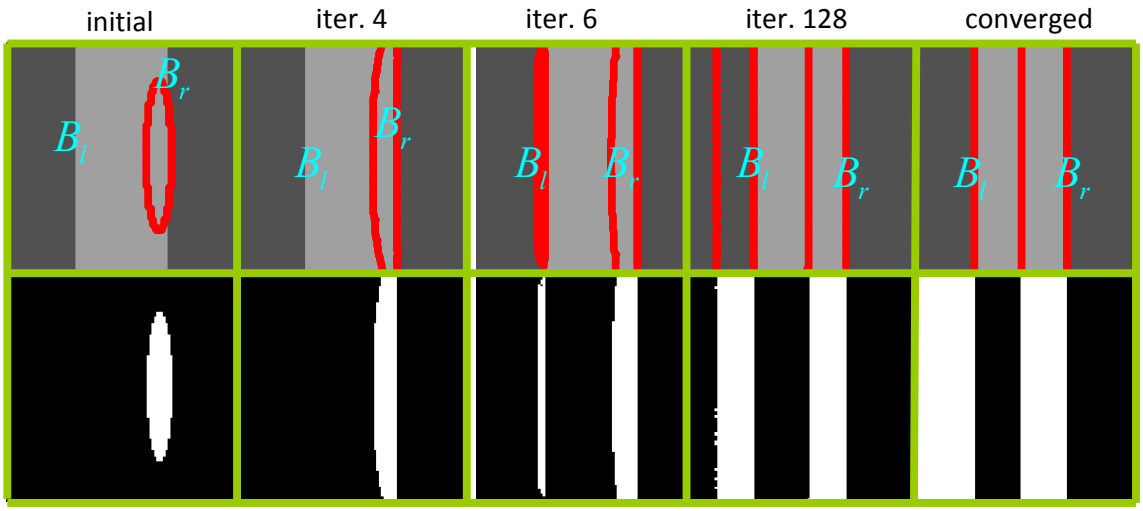

Figure 2.7: An example for which LBF converges to local minimums. First row shows contour evolution, second row illustrates areas where $\phi > 0$ (white) and $\phi < 0$ (black). B_l and B_r marked true object boundary discussed in the text.

computing f_2.

$$f_i = \frac{K_\sigma(\mathbf{x}) * [H_i(\phi(\mathbf{x}))I(\mathbf{x})W_i(I(\mathbf{x}))]}{K_\sigma(\mathbf{x}) * [H_i(\phi(\mathbf{x})W_i(I(\mathbf{x}))]}, i = 1, 2 \tag{2.20}$$

$$W_1(I(\mathbf{x})) = I(\mathbf{x}) \tag{2.21}$$

$$W_2(I(\mathbf{x})) = 255 - I(\mathbf{x}) \tag{2.22}$$

We call LBF with intensity re-weighting the IR model, which naturally encourages ϕ of brighter pixels to increase and ϕ of darker pixels to decrease along boundary. Fig 2.8 shows for the first example in Fig 2.7, why LBF gets stuck at a local minimum and IR converges to the desired global minimum. We initialize the sign of ϕ to be the convergence state in Fig 2.7, with absolute value of 0.1. Here we just show values for the middle row in the image (other rows are similar given no vertical intensity variation in the image). Eq 2.19 can be approximated as $e_i(\mathbf{x}) = |I(\mathbf{x}) - f_i(\mathbf{x})|^2$ because $f_i(\mathbf{y}) \approx f_i(\mathbf{x})$ when $\mathbf{y}$ is close to $\mathbf{x}$. Eq 2.18 can be interpreted as increasing ϕ if $|I(\mathbf{x}) - f_1(\mathbf{x})| < |I(\mathbf{x}) - f_2(\mathbf{x})|$, and decreasing ϕ otherwise. In LBF around

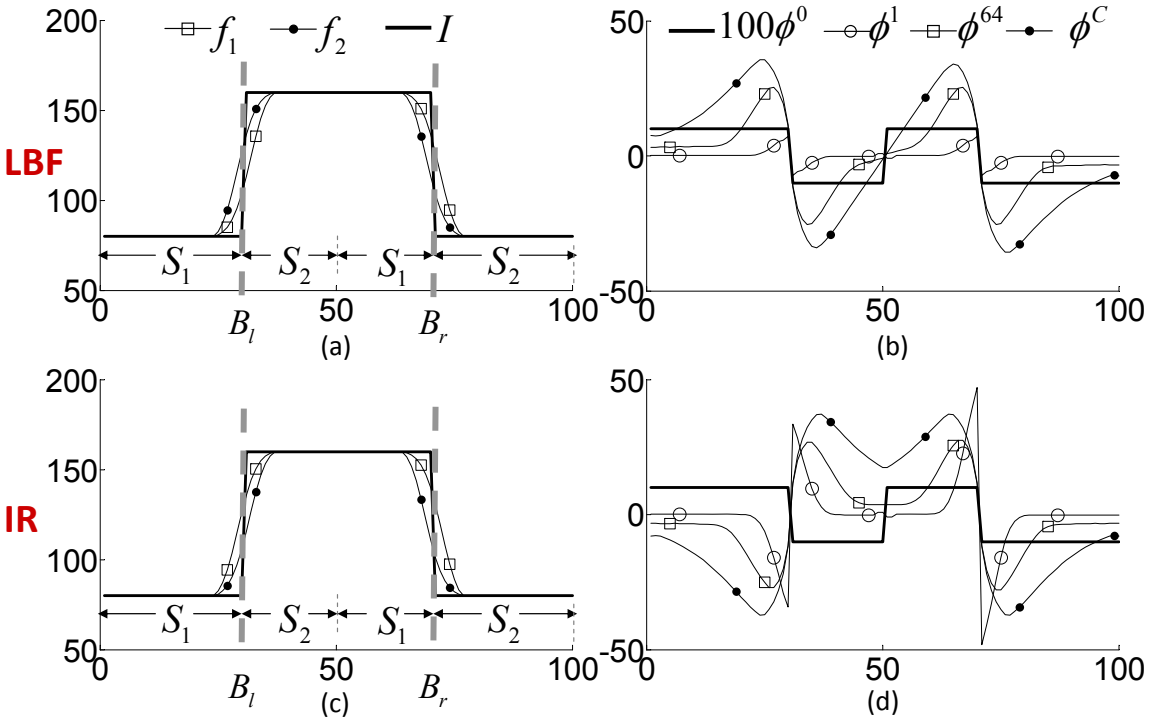

Figure 2.8: Comparison of LBF and IR for the first example in Fig 2.7. Here we only show values for the middle row of the image. ϕ is initialized to the convergence state in Fig 2.7. One the left, we show f_1, f_2 and I in the first iteration. On the right, we show the evolution of ϕ. ϕ^t is ϕ at iteration t. ϕ^0 is the initialization, and ϕ^C is ϕ at convergence. LBF does not change sign of ϕ in the evolution process, so gets stuck in local minimum. IR drives the sign of ϕ to flip in iteration 1, and converges to a global minimum.

boundary B_l, $\phi > 0$ for darker pixels and $\phi < 0$ for brighter pixels when initialized, resulting in $f_1 < f_2$. So near B_l, Eq 2.18 drives ϕ to increase in $S_1(\phi > 0)$ and decrease in in $S_2(\phi < 0)$, as shown in Fig 2.8 (b). In IR, around B_l, $f_1 > f_2$ because brighter pixels are weighted more in f_1, and darker pixels are weighted more in f_2. So near B_l, Eq 2.18 drives ϕ to decrease in $S_1(\phi > 0)$ and increase in $S_2(\phi < 0)$, causing the sign of ϕ to flip around B_l. At convergence, ϕ has consistent sign in each region.

Fig 2.9 shows a real example. The evolution direction of ϕ is visualized in the last column: increase ϕ for bright pixels ($\frac{\partial \phi}{\partial t} > 0$) and decrease ϕ for dark pixels ($\frac{\partial \phi}{\partial t} < 0$). In both LBF and IR, if a pixel is close to initial contour and true boundary (like A, B), then $f_1 > f_2$. So Eq 2.18 drives ϕ to increase for bright pixel B and decrease for dark

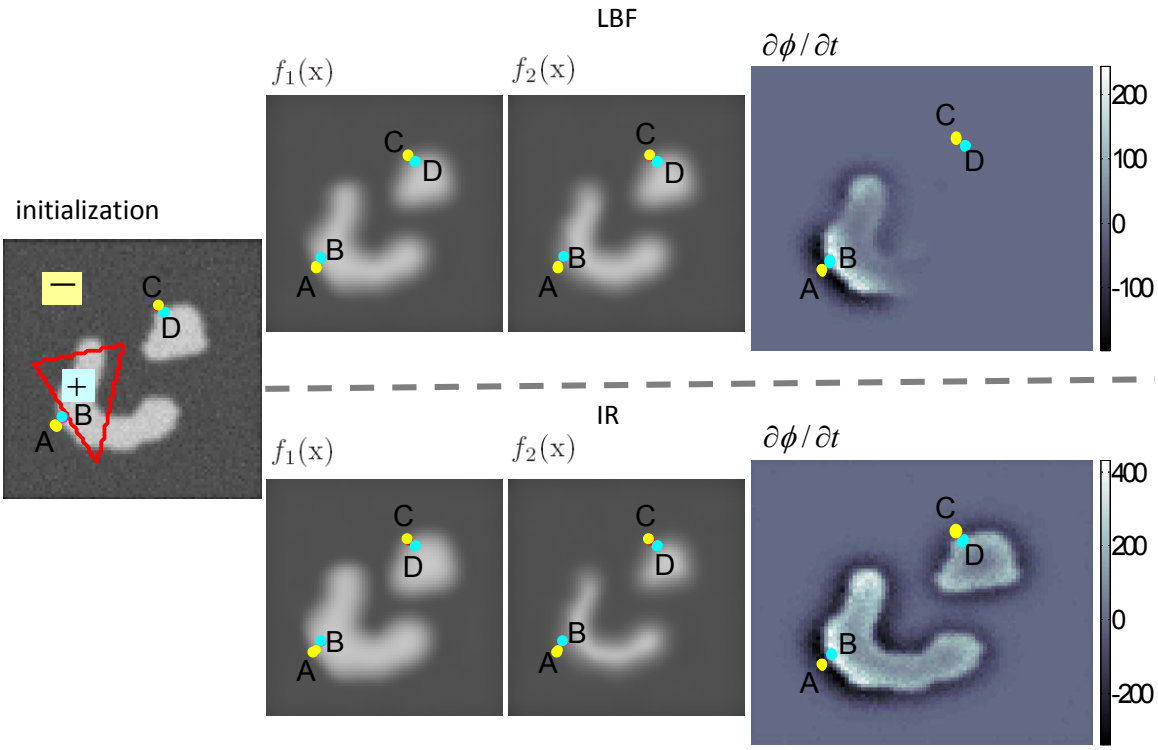

Figure 2.9: Comparing $f_1(\mathbf{x}), f_2(\mathbf{x})$ and $\frac{\partial\phi}{\partial t}$ in the LBF and the IR models. '+' and '-' indicates sign of the level set function ϕ.

pixel A. However, if a pixel is close to true boundary but far from initialization (like C, D), in LBF $f_1 \approx f_2$. This makes it unclear how ϕ evolves to reflect the intensity contrast on that boundary. Either increasing ϕ for C and decreasing ϕ for D or vice versa can decrease the energy. Such uncertainty of which direction to go is exactly the reason that can lead to local minima (convergence shown in Fig. 6 first column). In IR model, $f_1 > f_2$ near C and D, driving ϕ to evolve in the desired direction.

Another advantage of the IR model is that it enables faster convergence. LBF relies on the closeness of the current contour to the true boundary to guide evolution in the right direction. Whereas IR provides additional driving force from intensity contrast to guide evolution even when current contour is far from the true boundary. Fig 2.10 gives such an example.

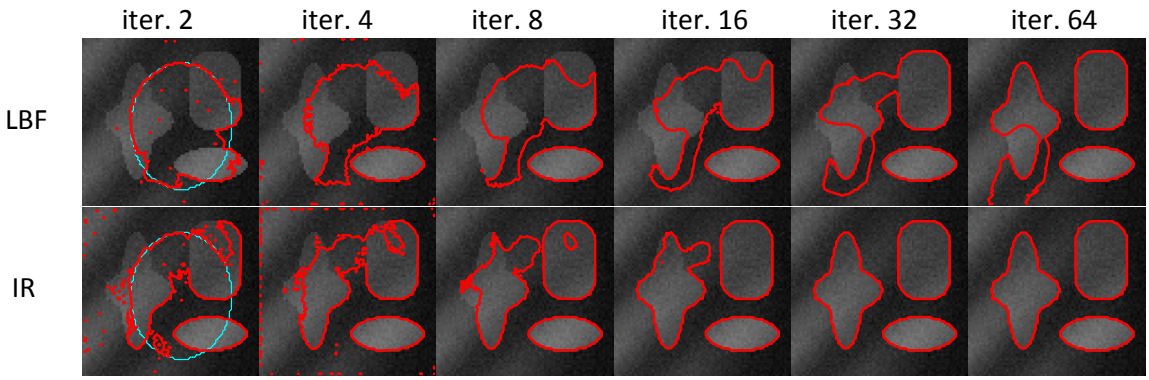

Figure 2.10: Comparing evolution contour (in red) at iteration 2–64 for the same initialization (in cyan) for the LBF and the IR model.

All experimental results use the same parameter setting $\lambda_1 = \lambda_2 = 1$, $\nu = 0.001 \times 255^2$, $\mu = 1$, $\sigma = 3.0$, $\epsilon = 1$, except for the first example in Fig 2.10, we set $\nu = 0.003 \times 255^2$.

We first show some examples for which LBF gets stuck in local minima, but IR successfully converges to the desired boundary in Fig 2.11. The last column shows an example where IR model does not work. This is an example when the basic assumption is violated. The two regions are of completely symmetric intensity. Unsurprisingly, at convergence $\phi > 0$ maps to brighter areas. Preprocessing the image (like computing the gradient image) can meet the basic assumption.

In Fig 2.11, we show both real and synthetic examples, and compare CPU time using LBF and IR in Table 1. To make fair comparison of computing time, we choose examples in which both models converge (or closely) to desired region boundaries. Table 2.3.3 lists CPU time for Matlab code on a Dell XPS 720 machine with 2.66 GHz Intel Core 2 Extreme QX6700 CPU and 2GB memory. We check convergence every 25 iterations. If the average percentage of pixels that change the sign of ϕ is less than 0.2%, then the model converges. It is clear that the IR model converges faster than the LBF model.

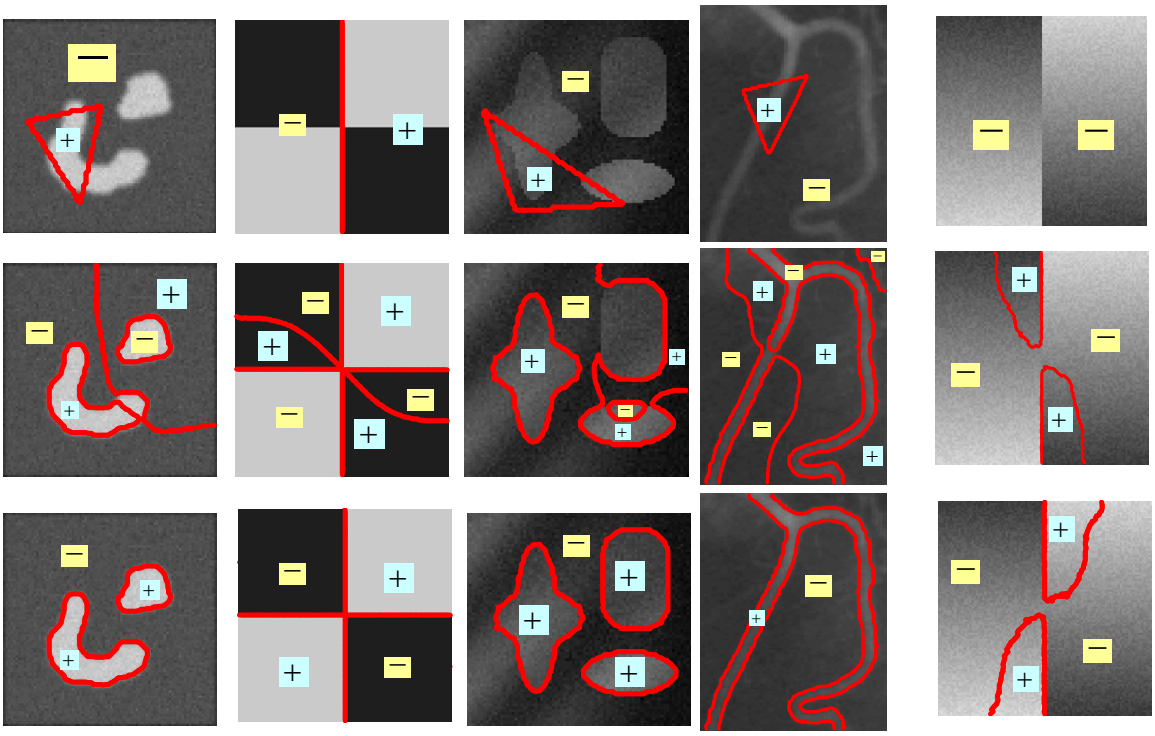

Figure 2.11: Top to bottom: initialization, final contours in LBF model, final contours in IR model. '+' and '-' indicates sign of the level set function ϕ. First four columns show examples in which LBF model gets stuck in local minima and IR model converges to the desired boundary. The last column shows an example where the IR model fails.

ImgID	1	2	3	4	5	6
LBF(Runtime in secs)	3.16	4.23	7.47	0.80	7.02	2.33
number of iterations	350	225	350	75	275	175
IR(Runtime in secs)	0.78	2.14	1.80	0.64	5.18	1.47
number of iterations	75	100	75	50	175	100

Table 2.1: CPU time (in second) and iteration number at convergence for LBF and IR model in Fig 2.11.

Chapter 3

Related Work

From a computational perspective, tracking evolving surfaces using the narrow band level set algorithm consists of two components, as described in chapter 2. First is the stencil computation on the narrow band, second is to track the motion of the narrow band. The stencil computation is closely related to prior works on optimization of stencils, and the narrow band tracking process is closely related to prior works on sparse linear algebra. Some prior works also explored improving data reuse on sparse data. Our work is also influenced by past work on auto-tuning, which is a generally adopted approach when performance *portability* is desirable. The auto-tuning technique has also been successfully applied to both stencil computation and sparse linear algebra, and to many other computational kernels.

3.1 Stencil Computation

Prior work on stencils has been primarily focusing on dense structured grids. Datta et al. investigate one-pass dense stencil computation over a wide range of multicore platforms [23, 21]. For a one-pass dense stencil, data reuse is relatively low because there is only one iteration in the temporal iteration space. There is some level of data reuse among stencil computation of neighboring grid points. Depending on the

computational complexity of the stencil kernel and the machine balance (CPU computational rate over memory bandwidth), the one-pass stencil can be either compute bound or memory bound. For example, in their work, the 7-point stencil is memory bound on all platforms, and optimization effort is focusing on utilizing memory bandwidth more effectively; while the 27-point stencil is compute intensive, and common sub-expression elimination (CSE) turns out to be important to reduce the flops/point and improve the overall performance.

There are many works in the literature investigating multiple-pass dense stencils. Given the abundant temporal reuse of data, the most important technique is performing time skewing in the iteration space, to effectively remove the bandwidth bottleneck [22, 58]. Time skewing is a special case of the polyhedral model, which is a nice mathematical framework for loop nest optimization in compiler theory [15, 14]. Some commonly used libraries for polyhedral transformation include Omega Library [26], PolyLib [33], and PPL [12]. The polyhedral model performs affine transformations on the original data structure, which is often modeled as mathematical objects called *polytopes*, and converts the original nested loops into more efficient loops in terms of locality and parallelism. In the following, we will use "time skewing" and "polyhedral transform" interchangeably, because the meaning will be clear in the context.

Time skewing on dense stencil.

In the following, we will briefly review the time skewing technique for dense stencil computation. This technique will be extended to work under the dynamic sparse narrow band setting in section 6.3 .

Time skewing is a general loop optimization technique that can effectively remove the memory bandwidth bottleneck or remove data dependency in the innermost loop for data parallelism. We first briefly review the basic polyhedral transform using examples of stencil computation on a dense 1-D grid (Fig 3.1) and a 2-D grid (Fig 3.2).

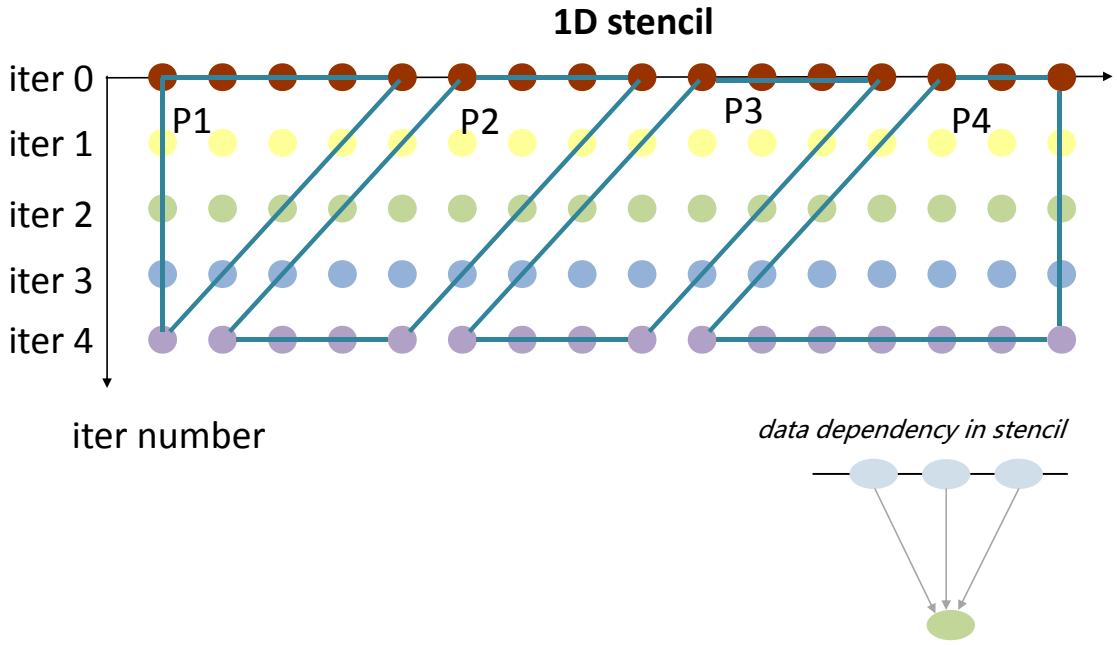

Figure 3.1: Polyhedral transform for 1-D and 2-D dense stencil. Data dependency assumes every point depends on three nearest neighbors in the previous iteration.

In the 1-D example, in each iteration, $\phi(x)$ depends on $\phi(x-1), \phi(x), \phi(x+1)$ in the previous iteration. The naïve approach is to sequentially finish each iteration on the complete array. The problem with this approach is that when the array size is too large to fit into cache, then each iteration it will load all source data from the main memory and store the updated value to the main memory. Given the poor machine balance (CPU computational rate over memory bandwidth) on modern cache-based architectures, the naïve approach easily becomes memory-bound.

The polyhedral transform reorganizes the computation order to improve data reuse. In the 1-D example, the computation in the space of 4 iterations is divided into four blocks (also called polytopes): P1–P4. Instead of finishing computation iteration by iteration, here we finish polytope by polytope, and in each polytope, computation is done iteration by iteration. If the polytope size is chosen appropriately to fit into the last level cache, all intermediate variables (points computed in iteration 1–3) will

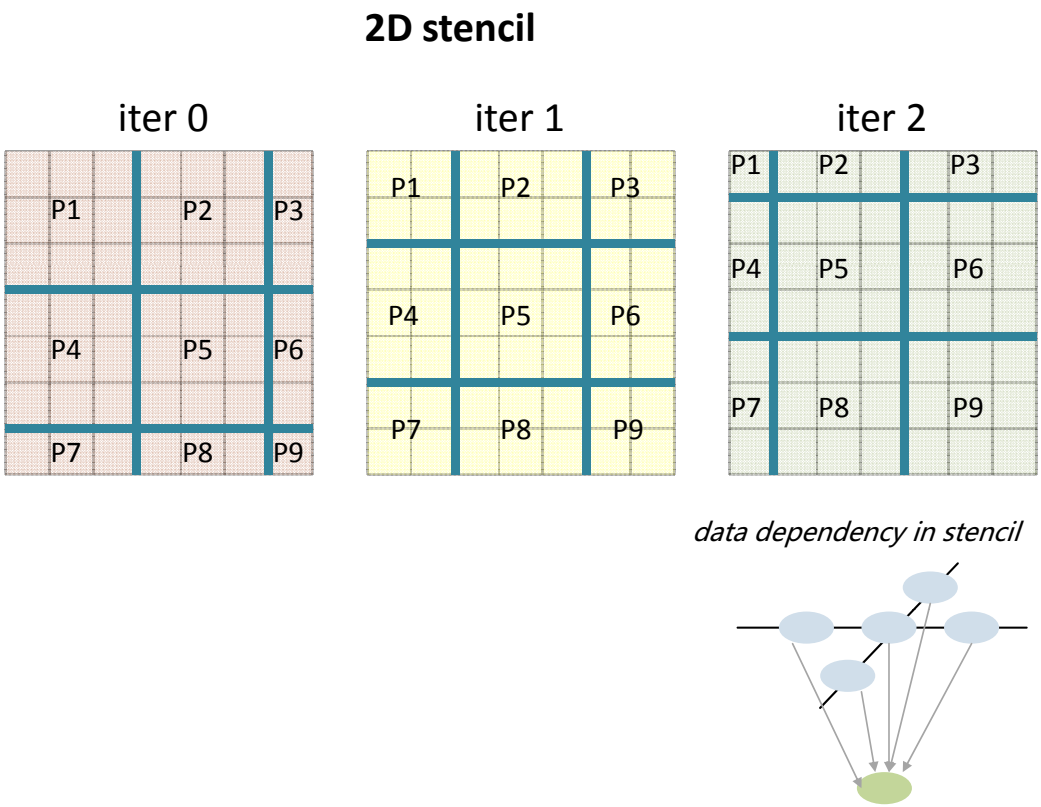

Figure 3.2: Polyhedral transform for 2-D dense stencil. Data dependency assumes every point depends on five nearest neighbors in the previous iteration.

reside in cache, and memory accesses only happen to points on the polytope borders (load points computed in iteration 0 and store points computed in iteration 4). The shape of the polytope depends on the exact data dependency of stencil computation. The blocking idea can be generalized to higher dimensional grid.

Fig 3.2 shows an example of 2-D stencil, where in each iteration, $\phi(y, x)$ depends on five nearest neighbors in the previous iteration: $\phi(y-1, x), \phi(y, x-1), \phi(y, x), \phi(y, x+1), \phi(y+1, x)$. Similarly, after blocking, computation is finished polytope by polytope (from P1 to P9). Within each polytope, computation is done iteration by iteration (from iteration 0 to iteration 2). With time skewing transformation, the data reuse is significantly improved by blocking in the temporal iteration space, and the application is likely to be compute bound.

3.2 Sparse Linear Algebra

Performance optimization on linear algebra is an extensively investigated topic. Many of the fundamental understanding and optimization methods are developed from research in optimizing linear algebra kernels, and extended to more complicated applications. Among the large family of linear algebra algorithms, there are numerous works on sparse matrix solvers. Some popular libraries developed for sparse matrix solvers include OSKI [53, 42], SPARSEKIT [43], and PETSc [13].

Most prior works on sparse linear algebra are memory intensive, and they focus on improving data reuse to better utilize the memory bandwidth. Earlier work by Im et al. [27] explored blocking strategies to save storage for indices, and register and cache level optimizations to improve locality. Recently, Vuduc *et al.* did thorough research in data structures, optimization spaces, and performance models to deliver high performance implementations and understand their efficiency [52]. Williams *et al.* extended Vuduc *et al.*'s work to thoroughly explore optimization opportunities on cache-based multicore platforms. Belgin *et al.* [35] pushed further in the direction of data representation of the sparse matrices, and proposed more efficient coding strategy for non-zero blocks. Their key observation is that for some types of the sparse matrices, patterns of how non-zero points are distributed inside non-zero blocks occur repetitively, and a small number of patterns are enough to capture the majority of the blocks.

The narrow band is naturally represented in sparse matrix format. We used a format similar to the compressed sparse row (CSR) format, which is an efficient format for storage of the sparse matrix data [13, 52]. In the CSR format, the sparse matrix is first tiled, and positions and data values of non-empty tiles are stored. Tiling can save storage for indices and better utilize computational resources on-chip. Conceptually, there is a 1-D array recording horizontal positions of non-empty tiles in every row, and another 1-D array per row recording the starting position of each row

in the first array. Similar to the sparse matrix solver, we also unroll the computation inside the tile, and explore opportunities to improve instruction level parallelism and register reuse. In a sparse matrix solver, the data values of non-empty tiles are also recorded in a compact array, when repeated computation on the same sparse matrix is performed. We are not compacting the data values as in the case of the sparse matrix solver, because the narrow band position is constantly changing to capture the motion of evolving surfaces.

3.3 Improving Reuse on Sparse Data

Sparse tiling [46, 45] and cache blocking [25] are techniques to improve locality for unstructured grids. In both methods, they assume a large number of iterations is performed on a stationary irregular mesh graph. Iteration space is tiled or blocked for better locality. The preprocessing stage to construct and partition the graph has a significant overhead, which hopefully can be amortized through a large number of iterations on the same mesh. For our case, we have a dynamic narrow band, and only a small number (B_r) of iterations are performed on every fixed band position, therefore the overhead cannot be well amortized. Also, constructing the mesh graph is unnecessary for our case. In other words, narrow band level set is a special case of iterative computation on an unstructured grid, but has a far more efficient solution for locality than the general sparse tiling or cache blocking technique.

3.4 Auto-tuning

Auto-tuning provides a productive solution to *portability* — the ability to write program once and deliver good performance across diverse multicore platforms. An auto-tuner requires a nontrivial amount of time to build. However, that time is amortized

through the performance *portability* that the auto-tuner offers. Auto-tuning typically consists of four basic steps.

1. Start with an application or a kernel of interest.

2. Identify all optimization opportunities and their parameters. For example, the block size used in blocking algorithms, the degree of loop unrolling, instruction scheduling, whether to use special instructions (like fused multiply-add), and how to vectorize code with SIMD instructions. Some choices are architecture-dependent, like SIMD instruction intrinsics and memory affinity. In such cases, an auto-tuner needs to generate specific code variants on different architectures.

3. Integrate all optimizations into an unified, parameterized auto-tuner. Different implementations are controlled simply by adjusting the parameter configurations. In some cases, the parameter choices lead to very different code that cannot be handled solely with the C preprocessor macros. In such cases, we either manually write code constructs, or we design a code generator to automatically synthesize the code parts according to the parameters if the code has a regular structure and highly repetitive pattern.

4. Find the best code by models or empirical search. The full search space typically grows factorially with the number of implementation choices. Current library generators usually incorporate some practical search strategies (like "hill climbing") or performance models to prune the search space such that the optimal or near-optimal parameter values can be found within reasonable amount of time.

Auto-tuning has demonstrated its effectiveness and advantages in a number of successful examples, including ATLAS [55, 24] for linear algebra, FFT libraries [29, 28, 51, 47, 37] for computing the discrete Fourier transform (DFT), SPIRAL [40, 36, 51] for a wide range of digital signal processing (DSP) transforms, OSKI [42, 53] for

computational kernels on sparse matrices, stencil computation on dense and regular grids [23, 21, 22], and many other applications.

Chapter 4

Computational Model

In this chapter, we describe the computational model used for the narrow band level set algorithm, including details of the stencil computation involved in the image segmentation example described in Chapter 2, and the data structures used in the narrow band level set implementation.

4.1 Algorithm in Details

We detail the stencil computation involved in edge-based level set for image segmentation in Eq 2.2, because later we will show how to optimize this stencil kernel based on its computational properties. For a different stencil kernel, for example, the stencil used in region based models, a similar optimization procedure should be applied, though details might differ.

In Eq 2.2, we need to compute following terms,

1. $\Delta\phi$, which is the divergence of ϕ. It is computed using its numerical approximation as

$$\Delta\phi = \phi(x, y+1) + \phi(x, y-1) + \phi(x-1, y) + \phi(x+1, y) - 4\phi(x, y). \quad (4.1)$$

2. ϕ_x, ϕ_y, which are first order derivatives of ϕ in x and y directions.

3. normal vector $\mathbf{N} = [\mathrm{Nx}, \mathrm{Ny}]$. $[\mathrm{Nx}, \mathrm{Ny}]$ are x, y components of the normal vector.

$$\mathbf{N} = \frac{[\phi_x, \phi_y]}{\sqrt{\phi_x^2 + \phi_y^2}} \tag{4.2}$$

4. curvature $k = \mathrm{div}(\frac{\nabla\phi}{|\nabla\phi|})$, computed as

$$k = \frac{\partial \mathrm{Nx}}{\partial x} + \frac{\partial \mathrm{Ny}}{\partial y}. \tag{4.3}$$

5. g, which is the *edge indicator function* in Eq 2.5.

6. g_x, g_y, which are first order derivative of g in x and y directions.

7. δ, which is the univariate Dirac function.

$$\delta_\epsilon(\phi) = \begin{cases} 0 & |\phi| > \epsilon \\ \frac{1}{2\epsilon}\left[1 + \cos(\frac{\pi\phi}{\epsilon})\right] & |\phi| \leq \epsilon \end{cases} \tag{4.4}$$

8. $\frac{\partial\phi}{\partial t}$, which is the update of level set function at each time step. Derived from Eq 2.2, it is computed as a function of the above terms

$$\frac{\partial\phi}{\partial t} = \mu\left(\Delta\phi - k\right) + \lambda\delta(\phi)\left(g_x \mathrm{Nx} + g_y \mathrm{Ny} + gk\right) + \nu g\delta(\phi). \tag{4.5}$$

In the real implementation, all first-order derivatives are computed using a simple three-point numerical approximation. For example, ϕ_x, ϕ_y are computed as

$$\phi_x = \frac{1}{2}\left(\phi(x+1, y) - \phi(x-1, y)\right), \tag{4.6}$$

$$\phi_y = \frac{1}{2}\left(\phi(x, y+1) - \phi(x, y-1)\right). \tag{4.7}$$

Among all the terms, g, g_x, and g_y are not dependent on ϕ, therefore they can be pre-computed. The other terms change as the surface evolves and must be updated in each iteration.

4.2 Data Structures

As discussed in chapter 2, the narrow band level set has two major components: the stencil computation `CompLS` and the 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 [52]. We tile the band using tile size $T_h \times T_w$. 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 indices. Given that band is represented in the tile granularity, the cost of `UpdateB` is closely related to the tile size.

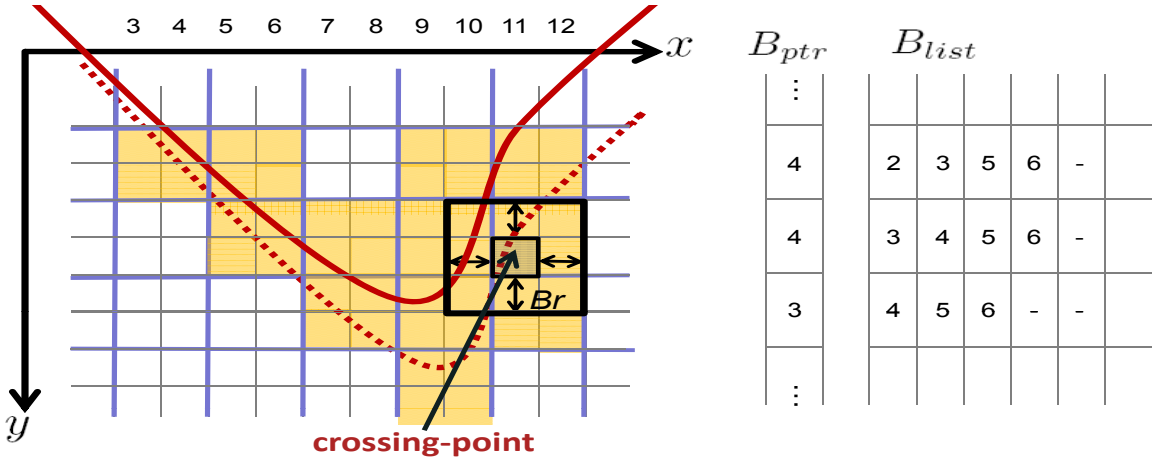

Figure 4.1: Illustration of the narrow band update process and data structures. Solid and dashed lines are the old *zero level set* and the updated one. In this example, the narrow band is tiled using 2×2 tile size.

In `CompLS`, we perform stencil computation on every pixel in each tile in the band, following Eq 4.5. To compute Eq 4.5 at position (y, x), we need to first compute a

normal vector N at its nearest neighbors $(y \pm 1, x \pm 1)$, following Eq 4.2. Computing the normal vector is expensive, since it requires square root and division. 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 Eq 4.2.

2. `CompL` step computes the updated level set function following Eq 4.5.

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 $\Delta \in \{0, 1\}$.

In `UpdateB`, we need to check every point in the current band if it is a *crossing-point* using the following 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 illustrated in Fig 4.1. 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 trespass 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 4.1, $B_{list}[j][]$ 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, each 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 entries of 1s, and updates B_{list} and B_{ptr} accordingly.

The complete computation and band update process is summarized in the following pseudo code.

```
//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 4.2: Pseudo code of the narrow band level set algorithm.

Chapter 5

Experimental Setup

In this chapter, we discuss details of the hardware development platforms, the parallel programming model we used in our experiments, and the compiler setup. We also explain how the experiment is designed and what kinds of performance metrics are used to understand the efficiency of the delivered code.

5.1 Hardware Platforms

We perform experiments 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 micro-architecture, memory system hierarchy, the number of hardware threads, peak flop rates and DRAM bandwidth, which are summarized in Table 5.1. Performing experiments on the two platforms allow us to understand the impact of architectural differences on the performance. Also it will demonstrate how to generate performance *portable* code using the auto-tuning approach.

Hardware configuration diagrams of both machines used in our study are shown in Fig 5.1.

Intel Dell T410.

Table 5.1: Summary of Hardware Platforms

Processor	Intel Xeon 5560	Intel Atom N270
Core Microarchitecture	Nehalem	Atom
Type	superscalar OoO	in-order
Threads/Core	2	2
Clk (GHz)	2.8	1.6
SP Gflop/s	22.4	6.4
L1 D-cache	32kB	32kB
L2(private)	256kB	512kB
L3(shared)	8M	–
System	**Dell T410**	**Atom N270**
Cores/Socket	4	1
Sockets	2	1
SP Gflop/s	179.2	6.4
DRAM size (GB)	12	1
peak DRAM BW (GB/s)	63.98	4.26
Compiler	ICC 11.0	ICC 11.0

Dell T410 is a 2.8 GHz dual-processor machine. Each processor is a quad-core Nehalem-based Xeon 5560. The Nehalem used a modern multi-socket architecture. There is a 6.4 GT/s QuickPath Interconnect (QPI) on-chip, handling communication between the two sockets like fetching data from remote DRAM, cache coherency, and I/O accesses. In this dual socket system, accessing DRAM memory from a local socket has much higher bandwidth and lower latency than accessing DRAM in a remote socket. This type of architectural design is called non-uniform memory access (NUMA). To attain full memory bandwidth offered by the NUMA based multi-socket system, programmers are responsible to map memory pages to the appropriate DRAM to minimize memory accesses to remote sockets.

The Nehalem microarchitecture also supports Hyperthreading (HT) and Turbo Boost technology. HT is Intel's term for SMT (simultaneous multithreading). Xeon 5560 supports two-way HT, allowing two hardware threads to run simultaneously on the same physical core. The idea of HT is to let one thread fill in the pipeline holes

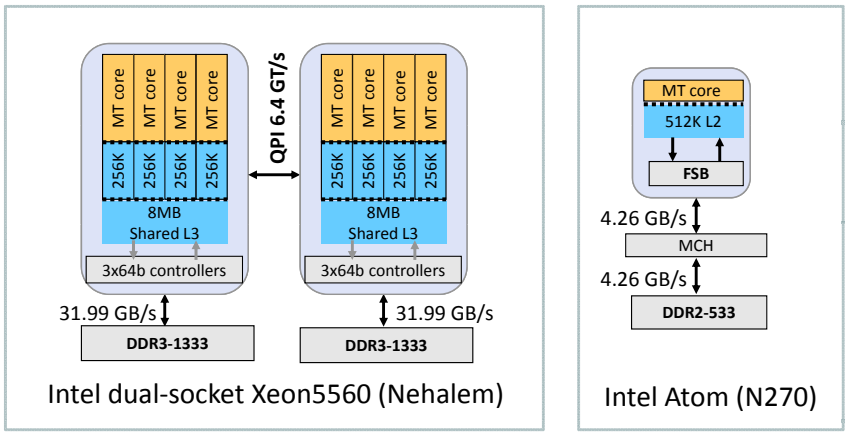

Figure 5.1: Hardware configuration diagrams for the machines in our study.

when the other thread is stalled, with little additional hardware cost. Physically, HT is implemented by replicating registers, statically partitioning load/store buffers between the two threads, and competitively sharing reservation station and caches [2]. The performance gain expected from HT technology depends on the exact performance bottleneck. When the bottleneck comes from some shared resource, there may be no performance improvement or even performance degradation from enabling HT technology. The Turbo Boost technology feature allows the core to run faster than its base operating frequency by dynamic over-clocking under certain workloads. However, it produces inconsistent timing results, therefore is disabled in our experiments.

Xeon 5560 supports x86-64 instruction set. In the dual-socket T410, each processor has four out-of-order (OoO) cores. Each core has its private 32kB L1 and 256kB L2 cache and the four cores share a 8M L3 cache. Each core can issue a 128-bit SIMD FP (floating point) multiplication and one 128-bit SIMD FP addition simultaneously per cycle. The theoretical peak of single precision FP arithmetic of this machine is $2 \times 4 \times 2.8 \times 8 = 179.2$ Gflop/s. Each socket integrates an on-chip DDR3 memory controller of three channels, providing 31.99GB/s DRAM bandwidth for each socket.

The integrated memory controller offers much higher memory bandwidth compared to the front side bus (FSB) used in the earlier Intel Core micro-architecture. The Nahelem microarchitecture also supports 2/4MB page size, which may be useful when TLB miss penalty has a noteworthy impact on overall performance. There is a single 32 entry DTLB for the large pages, according to the performance analysis guide for Nahelem based processors [32].

Intel Atom N270.

The Intel Atom N270 is a low-end processor designed for energy efficiency. It is mainly used for netbooks, mobile devices, smartphones, and consumer electronics (CE) devices. Our Atom N270 laptop is a 1.6GHz single processor machine, with a single core based on the Atom micro-architecture, which is highly optimized to reduce energy consumption. The Atom core has a two-issue wide, in-order pipeline that supports two-way Hyperthreading. It does not support Turbo Boost technology.

Atom implements the x86 (IA-32) instruction set. x86-64 is only activated for higher-end Atom processors designed for desktops, e.g. the Diamondville and Pineview cores. The Atom N2xx models cannot run x86-64 code [1].

The front end of Atom can issue up to two instructions (a 128-bit SIMD arithmetic operation and one memory operation) per cycle, delivering a theoretical peak of $1.6 \times 4 = 6.4$ Gflop/s for single precision FP arithmetic. The Atom processor has a 512kB L2 and a 32kB L1 data cache. The processor is attached to a memory controller via a 533MHz FSB, providing a memory bandwidth of 4.26 GB/s. Atom does not support large page size.

5.2 Software Environment

5.2.1 Parallel Programming Model

Three commonly used parallel programming models are OpenMP (Open Multi-Processing) [6], Pthreads (POSIX threads) [3], and MPI (Message Passing Interface) [5]. OpenMP and Pthreads are widely used for shared memory programming, and MPI is the dominant programming technique for distributed memory.

Pthreads are realized through an Application Programming Interface expressed in C programming language types and functions, to create and manipulate parallel threads. The threads created are fairly lightweight, unlike in MPI, where processes are much more heavyweight and require much information of programming resources and execution states. Programmers can explicitly control the number of threads created and how they are bound to physical cores (the so called "thread affinity") in Pthreads.

OpenMP supports multi-platform shared memory programming using a set of compiler directives, library routines, and environment variables that influence run-time behavior [6]. Similar to Pthreads, threads created using OpenMP are relatively lightweight. Additionally, OpenMP has the advantage of being fairly easy to use. Compiler handles most of the lower level details necessary for creating and running parallel threads. Programmers can easily create parallel regions and handle thread affinity through a simple and flexible interface.

Explicit and versatile thread affinity control in OpenMP becomes available recently. In earlier release of the OpenMP standard, compiler is decisive on controlling the parallelism. It could overwrite the number of threads specified by the programmer. The thread affinity is also handled completely by the compiler, therefore programmers had no control or knowledge over how the parallel threads got mapped to the multi-core system [21]. Recently, full control of thread affinity is provided though the Intel

thread affinity interface [7], which is a runtime library that can be used with Intel C/C++ compiler (ICC). Programmers can specify the binding of threads to physical cores using high-level, mid-level or low-level thread affinity interface, depending on how much details they want to control and specify. The low level thread affinity interface is very similar to the `sched_setaffinity` in Pthreads.

Given the above advantages of OpenMP programming model, we choose OpenMP in our experiments. We control the number of parallel threads by
`#pragma omp parallel num_threads(NUM_THREADS)`.
The thread affinity is simply controlled by setting an environmental variable named `KMP_AFFINITY`. For example,

```
set KMP_AFFINITY=verbose,granularity=fine,proclist=[0,1,2,3],explicit
```

explicitly binds four hardware threads to the processor specified in `proclist`, and `verbose` will enable display of the physical binding when the parallel threads are created.

In our experiments, we experiment with the number of threads which is a power of 2, and always exhaust resource on a given part of the chip before exploring new hardware, as Datta et al. did for stencil optimization [21]. For example, in Dell T410, we first experiment with 1,2, and 4 cores in the first socket, and finally 8 cores in both sockets. This approach allows us to understand the scaling efficiency of multicore systems. Also, we do not over-subscribe the number of threads to physical cores, because context switching between software threads is expensive and degrades the overall performance.

5.2.2 Compilers

The compiler we chose in our experiment is the Intel C/C++ compiler (`icc`), because it is considered to be the the best compiler today for Intel x86 platforms. `icc` has proven to generate superior code than `gcc` under most application scenarios [21, 9].

Also `icc` supports the Intel runtime library for controlling the thread affinity. `icc` can do some automatic SIMDization for simple loop structures, but can not handle complex data dependencies such as what we found in the stencil computation. For those complicated cases, we will manually SIMDize code, as will be explained in section 6.2.

5.3 Performance Measurement

Commonly used metrics to gauge machine utilization are delivered computational rate in Gflop/s and measured bandwidth in GB/s. We will show that these two metrics are not enough for our application.

For the narrow band level set algorithm, as explained in Section 2.2, it has a computational part `CompLS` and a band update part `UpdateB`. The computational part mainly consists of iterative stencil computations on the narrow band, which is computationally intensive. The band update part involves unpredictable control flows, for example, nested loops whose trip counts are hard to predict. We will show in Section 6.1, that the cost of `CompLS` and `UpdateB` are not fixed in terms the total number of arithmetic operations. By adjusting some algorithmic parameters, we can increase the cost on one part while decreasing the cost on the other part. Since there is no fixed arithmetic operation count, reporting Gflop/s does not reflect real runtime. However, reporting runtime itself is not very informative about the code efficiency, because it does not provide insight such as how much fraction of the machine peak is achieved.

To fully understand efficiency of the delivered code, we first build a stencil kernel code under an ideal setting, which uses a small image grid size such that all data can fit into the last level on-chip cache. This removes the memory bandwidth as a possible performance bottleneck. The kernel computation is performed on the whole

image grid as in the level set method, without the overhead of the narrow band representation. We also choose a relatively long unit-stride dimension to maximize memory access continuity. The stencil computational code is heavily optimized under such setting, as described in section 6.2. The measured performance serves as the "speed-of-the-light" upper bound of attainable performance for this specific stencil computation.

In the results section, we will report following performance metrics on each development machine.

1. Speedup over our baseline, which is a straight-forward implementation of the narrow band level set algorithm. We will compare our baseline with the best publicly available third-party code.

2. Delivered computational rate in Gflop/s for the computational part `CompLS`. We will compare this result with the kernel upper bound to understand the code efficiency.

3. Measured efficiency of control flows for the band update part `UpdateB`. After optimization, band update part mainly consists of big `switch` statements and memory writes. It is difficult to characterize the bottleneck for such control-intensive flows. We will report cycles/entry and cycles/write to the best effort.

4. Fraction of runtime spent in `CompLS` and `UpdateB`. This number reflects under proposed optimization techniques, the best tradeoff point found between the compute part and the band update part.

Chapter 6

Surface Tracking Framework

6.1 Overview

Surface tracking algorithm is closed related to stencil and sparse linear algebra, as discussed in Chapter 3. To deliver highly efficient code, we combine ideas developed for stencil and sparse linear algebra, and incorporate various code transformation techniques into an unified surface tracking framework.

We discuss about algorithmic design tradeoffs as well as code optimizations that address different possible system bottlenecks. The algorithmic design tradeoff explores balance between the stencil computational part and the band tracking part. The system utilization bottlenecks can be caused by on-chip resources, DRAM memory transfer, and multithreading. We elaborate them in more details in the following.

Algorithmic design tradeoff.

As introduced in section 2.2, the narrow band level set algorithm has a compute part `CompLS`, and a band update part `UpdateB`. There is a fundamental algorithmic tradeoff between the cost of these two parts. 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. In the extreme case when B_r is large enough to encompass the complete image, it degrades to the level set method on the complete grid. A similar relationship exists for the tile size: increasing the tile size reduces the cost of the band update because fewer tiles are needed to track the band, but the number of pixels in the band is increased. We will develop optimizations for both `CompLS` and `UpdateB`, and finally find out B_r, T_h, T_w that optimize the tradeoff in the auto-tuning process described in Chapter 7. The optimal B_r, T_h, T_w largely depends on how well each part is optimized, as illustrated in Fig 6.1.

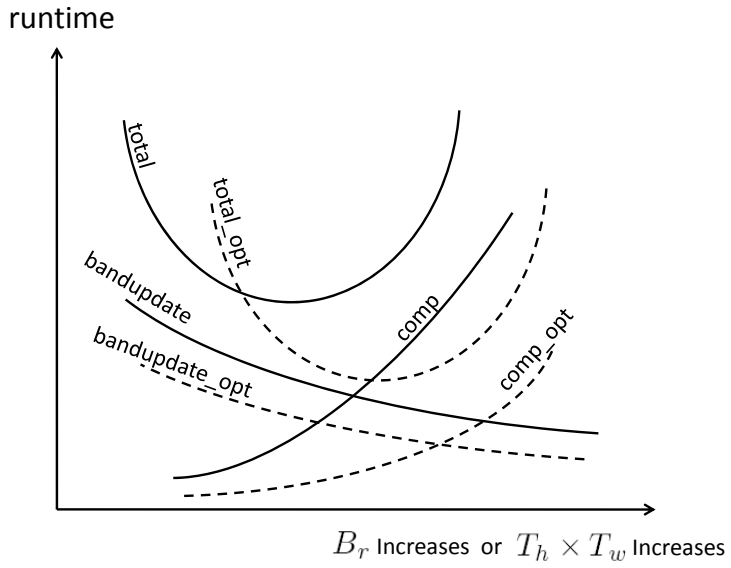

Figure 6.1: The fundamental tradeoff between computation and band update, controlled by B_r, T_h and T_w. Solid and dashed lines correspond to runtime before and after optimizations.

Code optimizations.

We propose a set of code optimization techniques for `CompLS` and `UpdateB`, organized into four categories:

1. In-core stencil optimizations that explore efficient utilization of on-chip resources;

2. Memory level optimizations that target at improving cache reuse;

3. Band update optimizations that reduce the cost of short loops with unpredictable trip counts;

4. Parallelization optimization that targets at achieving scalable performance with the number of cores on multicore system.

For the compute part `CompLS`, the in-core stencil optimizations are the most important when the data set size is small enough to fit into the last level on-chip cache. Memory level optimizations are most important when the data set size cannot fit into on-chip caches and the DRAM memory accesses become a bottleneck. Band update optimizations can effectively reduce the overhead of control intensive flows in `UpdateB`. With all these optimizations, we can achieve speedup of tens of times over varying input sizes for single-threaded performance. The parallelization scheme is specifically designed to minimize the computation and communication overhead when scaling to multiple cores. We observe close to linear speedup with the number of cores using our parallelization method.

6.2 In-Core Stencil Optimizations

We propose three optimizations aimed at maximizing utilization of in-core computational resources: SIMDization, approximate transcendental arithmetic, and instruction scheduling in the basic block of unrolled tile. In the following text, stencil computation of pixels in the same basic tile is completely unrolled and packed together to form a large basic block to explore ILP. The basic tile size $t_h \times t_w$ can be different from the tile size $T_h \times T_w$ used for tiling the band. $T_h \times T_w$ is chosen to balance the tradeoff as discussed in Section 6.1. For a set of basic tile sizes $t_h \in \{1, 2, 4\}$ and $t_w \in \{4, 8, 16\}$, we will explore the best instruction scheduling for the unrolled basic block formed using each of the basic tile sizes. In the auto-tuning search process, for any given T_h and T_w, we simply constrain that $T_h \times T_w$ can be divided into

multiple basic tiles of size $t_h \times t_w$, and choose the basic tile size that maximizes the computational rate.

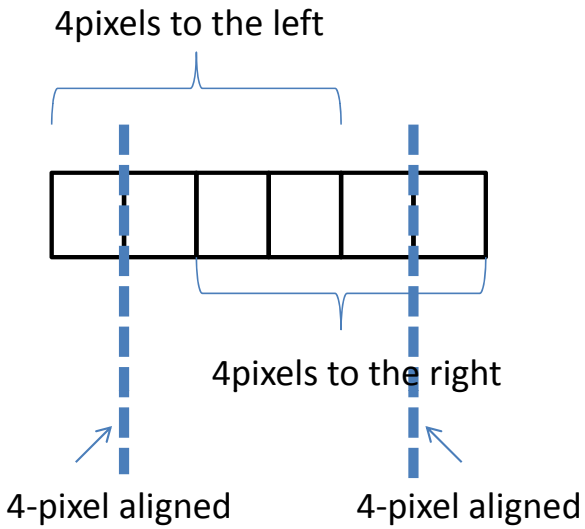

Figure 6.2: Illustration of four-pixel alignment. In computing first order derivatives, we need misaligned 4 pixels to the left and to the right. This is implemented by SIMD loads and shuffles to repack misaligned 4 pixels into 128-bit variables.

SIMDization.

Vector instructions naturally use a basic tile size that is a multiple of 1×4. Sometimes the stencil computation involves data dependency that is not 4-pixel aligned. For example, when computing the first-order derivatives of ϕ, we need $\phi(x)$ of 4 pixels to the left and to the right, as illustrated in Fig 6.2. In this case, we use SIMD loads and SIMD shuffle instructions to repack the 4 pixels to the left or to the right into a 128-bit variable. We also experimented with other tile sizes that are not 4-pixel aligned, but observed severe performance degradation. Therefore we constrain the basic tile width t_w to be a multiple of 4.

Approximate transcendental functions.

In `CompL`, notice there is the smoothed *dirichlet* function $\delta(\phi)$ computed as in Eq 4.4. In that equation, $\frac{1}{2}[1+\cos(\pi x)]$ for $x \in [-1, 1]$ can be well approximated by

$1 - x^2$, which can be easily vectorized. In `CompN`, we need to compute the magnitude of the norm, which is $1/\sqrt{(u_x^2 + u_y^2)}$ in Eq 4.2. Instead of using `_mm_sqrt_ps` and `_mm_div_ps`, we can use the fast and low precision `_mm_rsqrt_ps` without a noticeable impact on the final contour.

Instruction ordering.

Stencil computation on every pixel uses the same data dependency graph (DDG). We unroll the stencil computation instructions for all pixels in the same basic tile and pack them into a long basic block. Ideally compilers will perform instruction scheduling and register allocation to attain the best performance, no matter how C instructions are ordered in the basic block. However, we observe a large variance on performance when using different sequences of instructions that performs the same computation. Therefore, we generate a set of C instruction sequences and empirically find the best one. We do not change anything in the compiler, so this approach is compatible with future generation of compilers.

We generate a search space of instruction sequences using the following to two schemes: *replicate* and *interleave*, aimed at exploring a good balance between resultant register spills/reloads and ILP exposed. The replicate scheme is prone to placing dependent instructions close to each other to minimize additional memory transfers; while the interleave scheme is prone to interleaving independent instructions to maximize ILP. Fig 6.4 shows a simple example to illustrate these two schemes. We found that the assembly code for the interleave scheme typically has more spills than that of the replicate scheme, indicating that compilers re-schedule the input instruction sequences locally. All the instruction sequences are generated automatically by a code generator, which takes DDG of the stencil computation and the tile size as inputs. Empirically, we observed up to 70% performance difference between the best and the worst input instruction sequences.

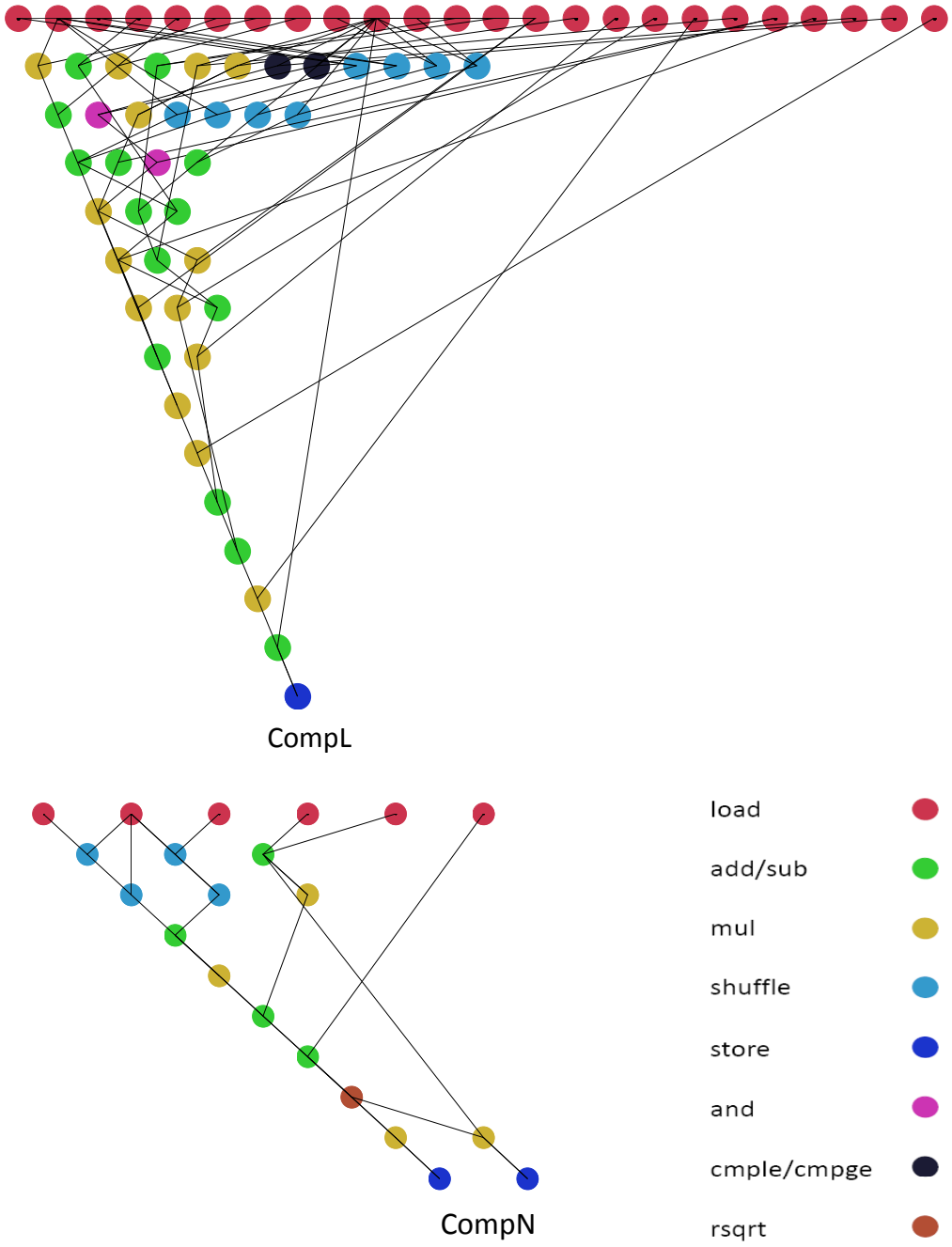

Figure 6.3: Data dependency graph (DDG) of stencil computation on one *superpixel* of 1×4 pixels. The upper plot shows DDG of `CompL`, the upper plot shows DDG of `CompN`.

Fig 6.4 shows a simple example to illustrate how these two schemes are implemented. In this example, assuming the DDG for each superpixel has 6 instructions:

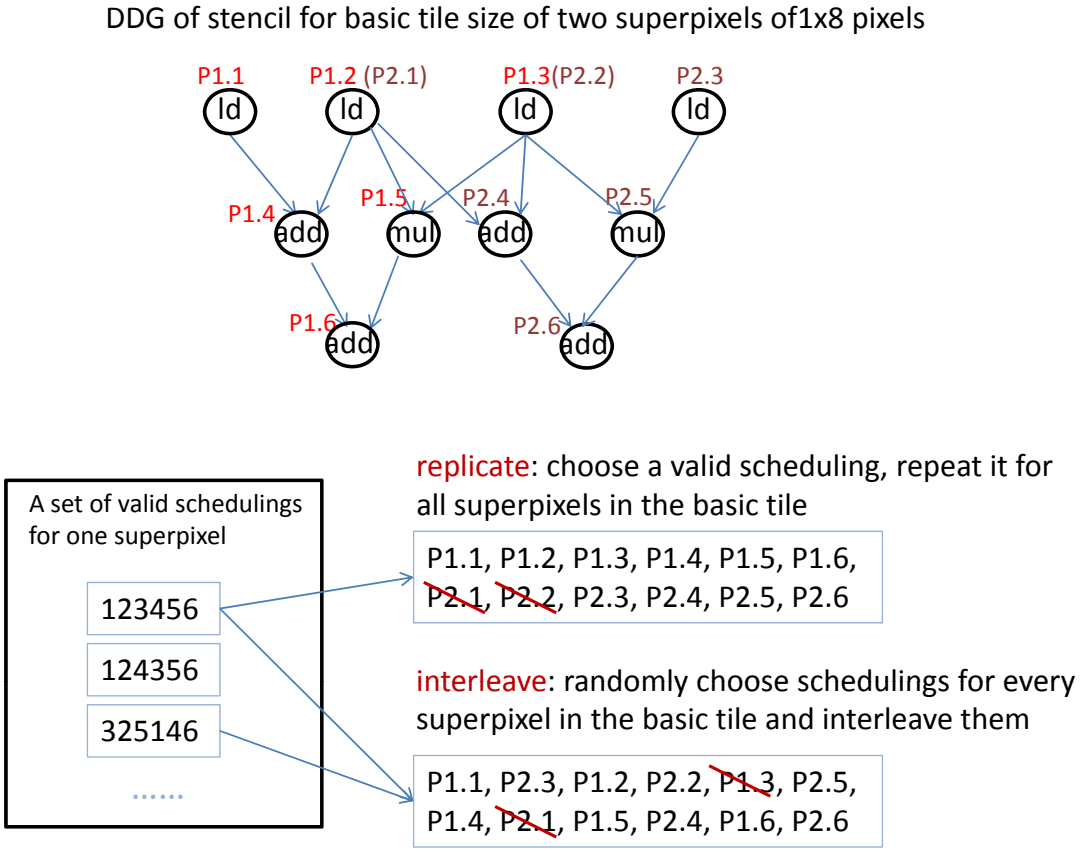

Figure 6.4: Illustration of replicate and interleave ordering schemes for an unrolled basic tile of 1×2 superpixels. A superpixel consists of 1×4 pixels. For SIMDization purpose, we only consider tile sizes which are multiples of 1×4. The two superpixels use the same DDG ($Pi.j$ means the j-th instruction 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 superpixel. The replicate scheme concatenates one sequence superpixel by superpixel; the interleave scheme mix two randomly chosen sequences of two superpixels. We use different sequences in interleave because usually a core can issue instructions of different types in the same cycle. In both schemes, redundant (shared) instructions of multiple superpixels will be removed.

3 loads, 2 additions, and 1 multiplication. The tile has two superpixels of 1×8 pixels, so DDG of two superpixels are packed into one basic block. The two superpixels share some input data, which is common in the nearest neighbor stencil computation, for example when computing first order derivatives. We first generate a set of valid sequences of 6 instructions of one superpixel, based on its DDG. The *replicate*

scheme simply concatenates one sequence superpixel by superpixel; while the *interleave* scheme interleave two randomly selected sequences from the set of valid ones. Note that we choose to interleave different sequences in *interleave* instead of the same one. This results in better balanced instruction types in a local window, and instructions of different types may be issued in the same cycle. In both schemes, redundant (shared) instructions of multiple superpixels will be removed. The same method can be applied to scalar code without SIMDization by replacing superpixel with pixel.

In real implementations, we randomly generate 200 prototype sequences of `CompL` on one superpixel of 1×4 pixels, based on its DDG. The *replicate* scheme simply replicates each as illustrated in the above example, when the basic tile has multiple superpixels. The *interleave* scheme randomly picks up $t_h \times t_w$ sequences from the 200 prototypes, and interleaves them into a new sequence. This process is repeated 200 hundred times to generate 200 *interleaved* sequences for each basic tile size. For `CompN`, we choose 50 instead of 200, because the DDG of `CompN` is much simpler than that of `CompL`, as illustrated in Fig 6.3. In `CompL`, a superpixel of 1×4 pixels performs 14 vector ADD and 14 vector MUL; while in `CompN`, a superpixel of 1×4 pixels performs 4 vector ADD and 4 vector MUL.

6.3 Memory Level Optimizations

6.3.1 Time skewing on Narrow Band

Why traditional time skewing does not work?

Fig 6.5 (a) shows applying the traditional 2-D time skewing to the narrow band. Here we assume a five nearest-neighbor stencil. Let us start from a fixed narrow band tiled with 2×2 tile size. The 2D grid is partitioned into blocks, and each block is skewed left-ward and up-ward by one pixel in the iteration space to respect the data dependency. Misalignment of the time skewing space with the tile size requires

programmers to handle corner cases when a tile is partially covered by the block. An alternative could be aligning the skewing space with the tile size, such as in Fig 6.5 (b). However, the traditional time skewing incurs considerable code overhead:

1. Identifying which tiles are active in the block has a non-trivial cost. It can be done either by checking an indication array like B_I for active tiles, or by checking the B_{list} in CSR format. Both methods consist of a number of unpredictable branches.

2. A broader region of the image needs to be considered, rather than the narrow band region. A block containing no active tiles at iteration t may contain active tiles at future time steps in the skewed space. The problem gets even more complicated when considering a dynamically evolving narrow band. The iteration space needs to be further skewed to respect the data dependency in the band update process, which leads to more complicated code and less efficient cache reuse.

The inefficiency is caused majorly by the interaction between the time skewing space and handling of the narrow band (representation and updating). In the following, we propose a novel technique called projective time skewing, which essentially decouples the time skewing space and the narrow band part. Nice properties of the traditional time skewing are maintained, and code overhead is kept very low.

Projective time skewing.

In our solution, we project the 2-D sparse grid into a 1-D array of sparse rows, where each node in the 1-D array corresponds to a sparse row in the original 2-D image, as illustrated in Fig 6.6. Let's consider a fixed narrow band first. After projection, updating each node depends on its three nearest neighbors in the previous iteration. Therefore, we can treat the problem in the same way as 1-D dense stencil computation, as illustrated in Fig 6.7. The only difference is that now each node corresponds to one sparse row in the original image, and may have a variable length of effective tiles in the narrow band. Given that a sparse row usually has a small

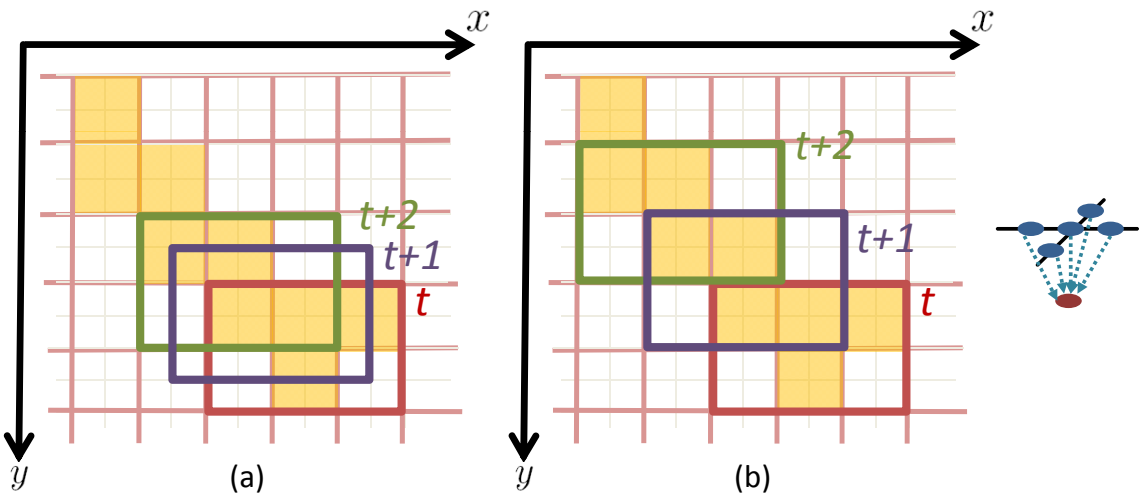

Figure 6.5: Applying traditional 2-D time skewing to 2-D sparse and fixed narrow band.

number of tiles, it is reasonable to assume a fairly large number of sparse rows can fit into caches on-chip, and time skewing can significantly reduce the DRAM accesses. In addition, the CSR representation of the sparse matrix naturally merges into this projective time skewing method, because it exactly contains the position information of "active" tiles in each sparse row.

Now let us consider the evolving band. The band needs to get updated every B_r iterations. If B_r is a multiple of the height of polytope, band update can be done after every B_r iterations on the whole image. However, usually B_r is small to keep the band "narrow" to reduce the number arithmetic operations. And the polytope height is usually large to increase data reuse. Typically, B_r is much smaller than polytope height. Under this case, we need to incorporate the band update process in the time skewing space.

The original time skewing space needs to be modified to accommodate the data dependency involved in the band update process. This is illustrated in Fig 6.8. Assuming polytope height is 4 and $B_r = 2$. Consider a polytope in steady state, starting from row 6 and 7 at time t. After two iterations, we can not proceed to compute row

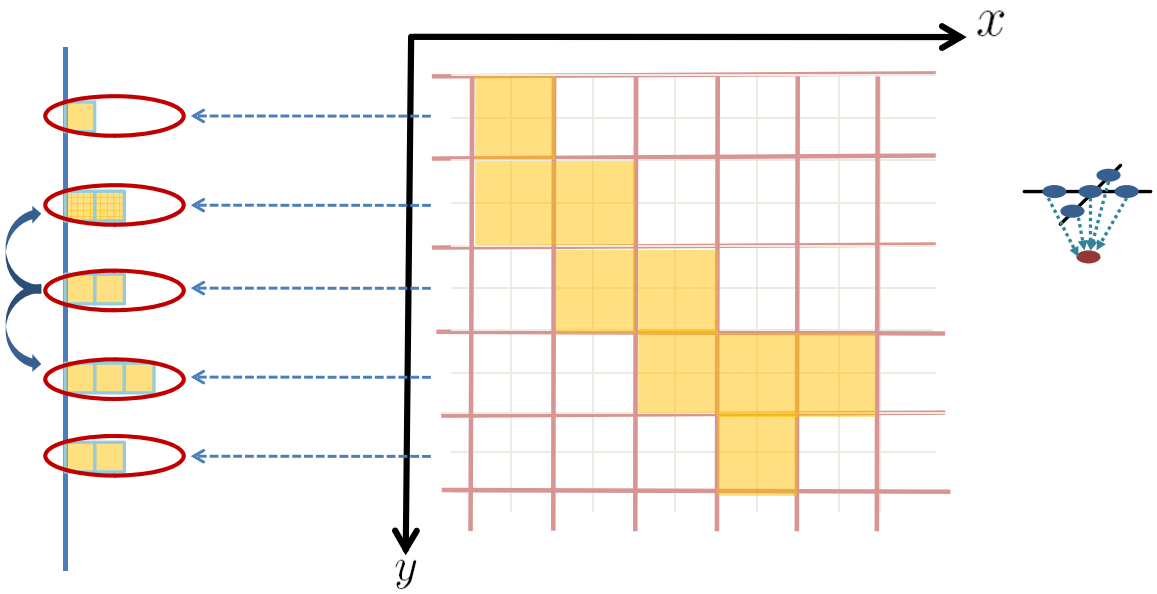

Figure 6.6: Projecting 2-D sparse grid to 1-D dense array, where each node in 1-D array corresponds to a sparse row in the 2-D grid.

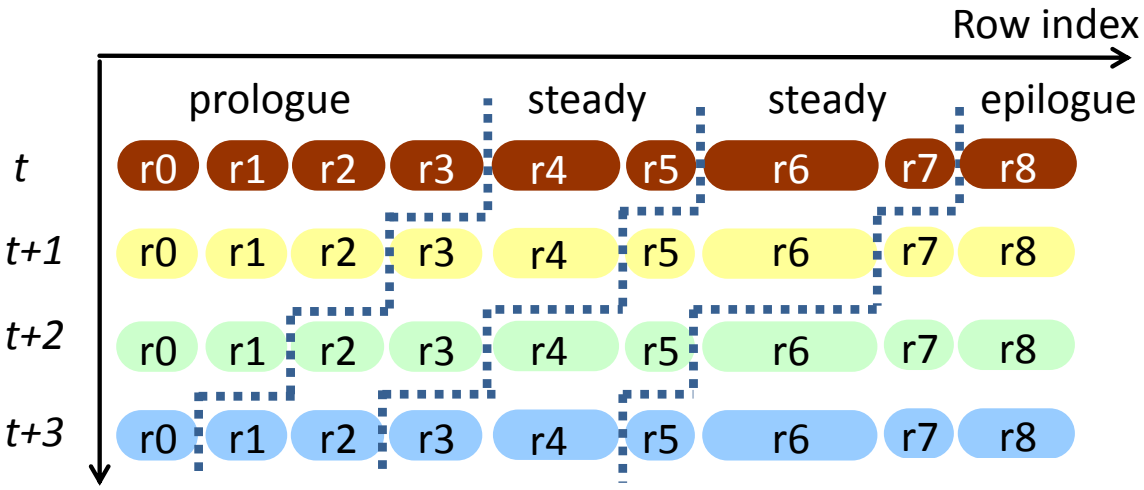

Figure 6.7: Time skewing of fixed 2-D narrow band, after projecting the 2-D sparse grid to 1-D array of sparse rows.

4 and 5 at time $t + 2$ because the narrow band shape changes and we do not have enough information to reconstruct the narrow band in row 4 and 5 yet. Additional skewing is needed to compensate the data dependency involved in the band reconstruction process, as shown in Fig 6.8. We reconstruct the narrow band as far as

possible with currently available information. When 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 because a crossing-point in row j can only change B_I for rows $j \in [j - \lfloor \frac{B_r-1}{T_h} \rfloor - 1, j + \lfloor \frac{B_r-1}{T_h} \rfloor + 1]$. After P_h iterations, the row index is left shift by $\Delta_p = \lfloor \frac{(2B_r+\Delta_s-1)P_h}{B_r} \rfloor$. Both Δ_s and Δ_p can be pre-computed and used as preprocessor macros, so index manipulations in the time-skewing is very cheap.

In this example, band is updated for row 2 and 3, and we can proceed with additional two time steps. Notice that the content of a row could change after the band update process. In this way, the interaction between the time skewing space and the narrow band part is kept minimal, and nice properties of the traditional time skewing on dense grid become reusable.

In real implementation of the image segmentation problem, update of ϕ defined in Eq 2.2 is broken down into `CompN` and `CompL`, as introduced in Section 4.2. Either `CompN` or `CompL` of row j depends on level set value ϕ and normal vector N for row $j-1, j$ and $j+1$ in the previous iteration. Please note that the row here is also in the tile granularity, that is, row index j is the original row index scaled by $\frac{1}{T_h}$. So each iteration actually consists of two small 1-D stencil computations: the first one for `CompN` and the second one for `CompL`. Note that we use two arrays to store the output of `CompN` and `CompL`, which essentially serve as ping-pong buffers for consecutive iterations. Otherwise, an additional buffer is needed to make sure data is not overwritten before it is needed again, which will incur an overhead of buffering data. The projective time skewing used for the real image segmentation problem is summarized in Fig 6.9.

Following pseudo code is for computing one polytope in steady state in Fig 6.10.

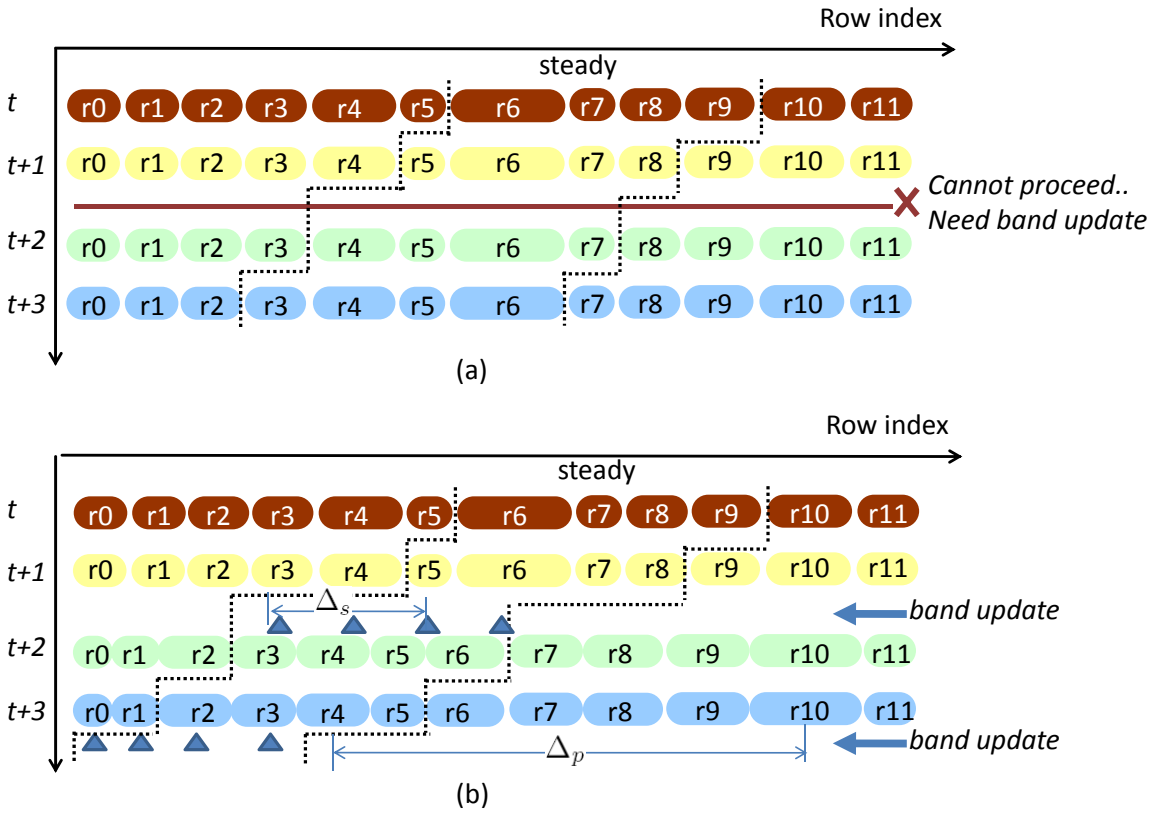

Figure 6.8: Projective time skewing on a dynamically evolving narrow band. In the example, $B_r = 2$, $P_h = 4$, $P_w = 4$. Band update happens every B_r iterations (indicated by triangles), and is integrated into the time skewing space.

With the polyhedral transform, we can save space for B_I by using it in a cyclic way. Height of B_I can be reduced to $\Delta_p + P_w + 2\Delta_s$, because this is an upper bound of the number of rows that are "active" at the same time.

Here we give a qualitative analysis of the choice of the polytope size. We name all points in consecutive P_h iterations as one polyhedral *segment*. In Fig 6.9, we show three *segments*: current segment, previous and next one. If 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, a cache miss happens to nodes on the polytope

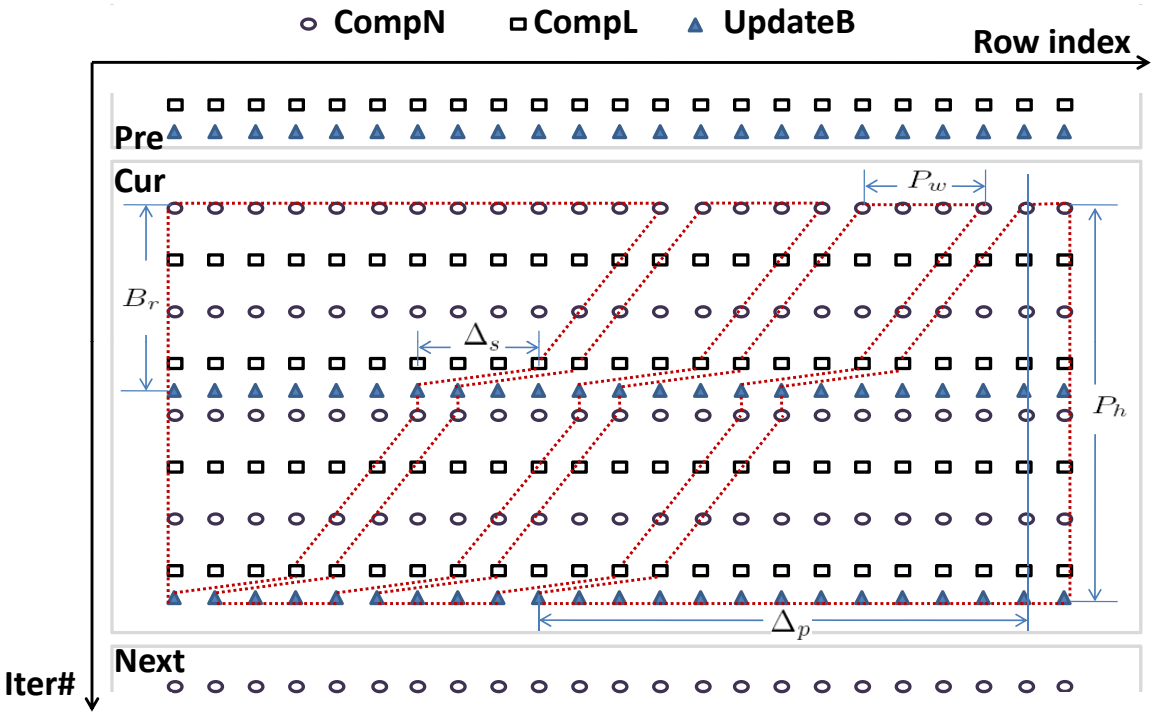

Figure 6.9: Polyhedral transform for narrow band level set. In the example, $B_r = 2$, $P_h = 4$, $P_w = 4$. Every P_h iterations form a polyhedral segment. Computation is finished in the order of segment by segment. Within each segment, computation is finished polytope by polytope. `UpdateB` is performed every B_r iterations in each polytope(here finishing a stripe of `CompN` and `CompL` counts as 1 iteration).

boundary. The total miss rate is then 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, the 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, miss rate is high as if there is no polyhedral transform. Therefore, when the cache size is big enough to hold a fairly large number of sparse rows, cache misses will no longer be the bottleneck since the miss rate changes roughly with $\frac{1}{P_h}, \frac{1}{P_w}$. This is observed on Xeon with a 8MB L3. When the cache size is small, it is possible the cache miss penalty can only be partially hidden, which is observed on Atom with a 512K L2.

```
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 6.10: One polytope in steady state.

6.3.2 Lower Level Optimizations

With the polyhedral transform only, we cannot completely hide the memory access penalty. Fig 6.11 shows the result on Xeon assuming a fixed band. After the polyhedral transform, performance slowly decays as the image size grows, with a few bumps at certain image sizes. By collecting hardware performance counters information using Intel Vtune [10], we determined that the slope is caused by page walk penalties. On Xeon, each core has an L1 TLB of 64 entries and a shared 512-entry L2 TLB for 4kB pages. There is a 32-entry DTLB for the large 2/4MB pages. Therefore, the large-page TLB can hold a maximum address span of 64MB/128MB, while 4kB TLB can only hold 2MB. Using the large page size can almost completely remove the page miss penalty. The bumps are caused by conflict misses in the last level cache, which can be removed with appropriate padding.

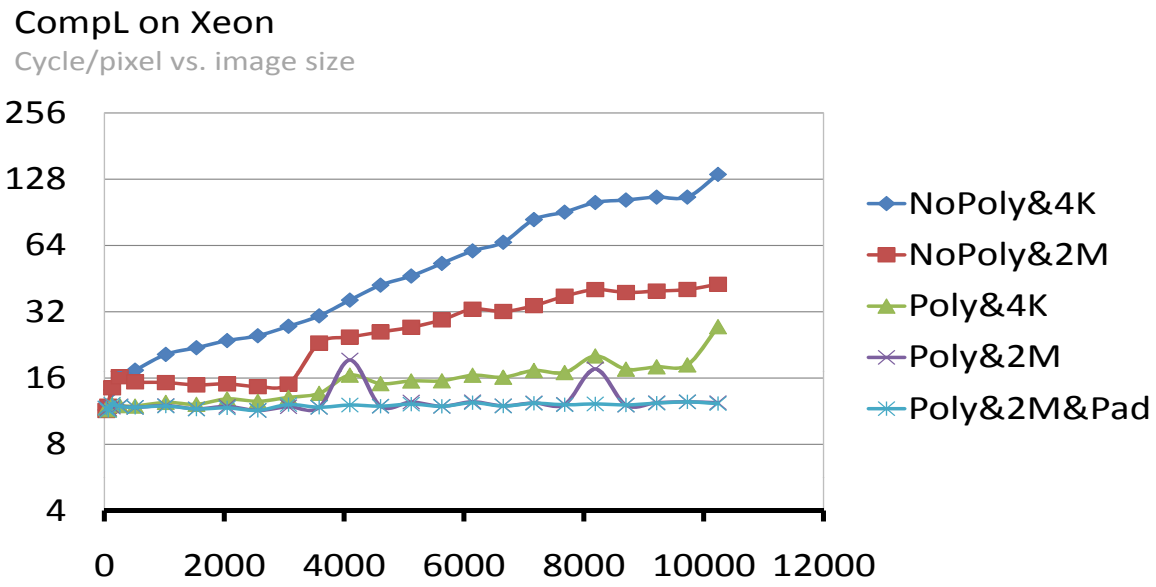

Figure 6.11: Performance (cycle/pixel) of `CompL` on fixed narrow band, with $B_r = 1$, $T_h \times T_w = 1 \times 4$. With polyhedral transform, large page support and padding, the application becomes completely compute bound.

6.4 Band Update Optimizations

The cost of the band update is largely dependent on the choice of the tile size, because the band is tracked in the tile granularity. Given fixed tile size and band radius, we propose two optimizations.

Optscatter.

The `Scatter` step searches for crossing-points and updates B_I at the same time. We propose two level of optimizations on `Scatter`. Level 1 optimization SIMDizes the search process. To check a *superpixel* of 1×4 pixels, we need *superpixels* on the up, down, left and right: `m_t`, `m_d`, `m_l`, `m_r`. We compute an integer `pattern` in 0–15, with 4 bits each indicating one pixel being a crossing-point or not, using the following SIMD 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));
```

Level 2 optimization unrolls the B_I update part. Updating B_I in the square neighborhood of new *crossing-points* involves short nested loops with unpredictable trip counts. With known B_r, T_h, T_w, we wrote a Perl script enumerating all possible cases, unrolling the code and organizing them into a big `switch` statement.

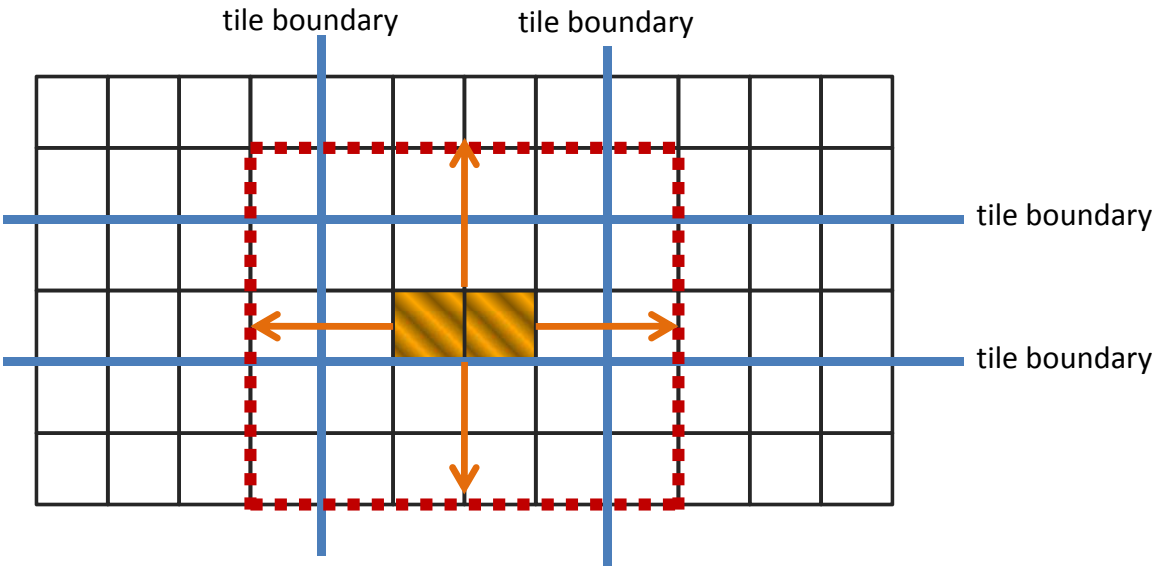

Figure 6.12: An example to illustrate the unrolled `scatter` process. In this example, the tile size is 2×4. The relative position of the *superpixel* within the tile is (1,0), meaning it is the lower half of the tile. The *superpixel* has a crossing-point pattern of '0110'. Band radius $B_r = 2$. The red dotted box illustrates expanded area of the crossing points. All tiles that have a non-empty intersections with the red dotted box should be included into the updated band.

We show a simple example in Fig 6.12. The original code and code after unrolling for this example is shown in Fig 6.13.

Optgather.

`gather` simply scans B_I for 1s, and collects tile column indices into B_{list}. Instead of loading `char` in B_I one by one, we load 4 `chars` or 16 `chars` each time, and build a big `switch` statement for each possible case. This reduces load and branching overhead. It helps most when the image width is large and the application is compute bound.

```
//Original code without unrolling
void scatter0110(int j,int i,int rj,int ri,char B_I[][BIw]){
   int j_l = (j*Th+rj-Br)/Th;
   int j_u = (j*Th+rj+Br)/Th;
   int i_l = (i*Tw+ri+1-Br)/Tw;
   int i_l = (i*Tw+ri+2+Br)/Tw;
   for (int j=j_l; j<=j_u; j++)
      for (int i=i_l; i<=i_u; i++)
         B_I[j][i] = 1;
}

//Optimized code with unrolling
//we can pre-compute j_l, j_u, i_l, i_u
//j_l = j-1; j_u = j+1; i_l = i-1; i_u = i+1;
//The following code will be inserted in a big switch
//statement, with one entry for one possible
//combination of (rj, ri, pattern)
B_I[j-1][i-1]=1;  B_I[j-1][i]=1;  B_I[j-1][i+1]=1;
B_I[j][i-1]=1;    B_I[j][i]=1;    B_I[j][i+1]=1;
B_I[j+1][i-1]=1;  B_I[j+1][i]=1;  B_I[j+1][i+1]=1;
```

Figure 6.13: An example of unrolled `scatter`.

6.5 Parallelization on Multicores

There are multiple ways to explore data parallelism when scaling to multicores. We use the method in [58] due to its low communication cost, but tailor the original method to the narrow band setting. This is shown in Fig 6.14. Core n first finishes its prologue, then sets DONE[n] to 1, indicating core $n-1$ it has finished. Then core n makes progress in the steady state until entering the epilogue. It will wait until DONE[$n+1$] is set because it needs some information produced in the prologue of core $n+1$. 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 processor $n+1$ finishes its prologue, it will copy B_I information of certain rows (indicated by little triangles in Fig 6.14) to B'_I of core n. When core n finishes the last *polytope* in the steady state, it also copies B_I of certain rows (indicated by the little triangles in Fig 6.14) to its own B'_I. Size of B'_I is $2(\Delta_p + \Delta_s)\frac{w}{T_w}$ to make sure all "alive" rows at the same

time can fit in. For large image size, each core has enough work in the steady state, so when core n reaches the epilogue, core $n+1$ should have finished the prologue and set the DONE$[n+1]$. The overhead of this method is low since no redundant computation is needed near the partition borders. To balance the workload among the cores, after each polyhedral *segment*, we re-collect information of the number of tiles in the narrow band per row, and partition rows in a way such that the number of tiles per core is roughly the same.

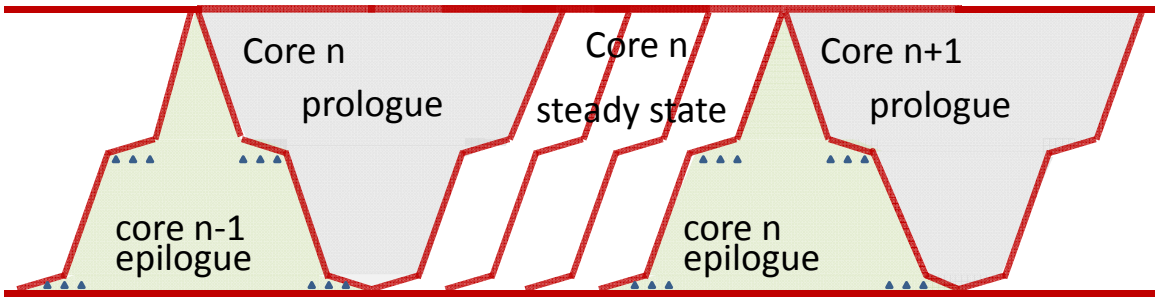

Figure 6.14: Parallelization on multicores (small triangles indicate rows whose B_I information needs to be copied to an additional buffer B'_I for the epilogue processing).

We also attempted a finer grain partition scheme so that processors share cache resources, that is, two cores cooperate to finish each row, with core 0 computing all even entries in $B_{list}[j][]$ and core 1 computing all odd entries. Experiments show a 1.5x speed-up when each core has its private L1. The main penalty comes from data sharing among the two cores on chip, incurring a lot of "modified data sharing" transactions (verified by Vtune hardware counters [10]). We also experimented with SMT (simultaneous multithreading) using this scheme, however observed no speed-up. This is likely due to some shared resource of SMT becoming the bottleneck.

Chapter 7

Auto-tuning

With all optimization techniques discussed in the surface tracking framework in Section 6, now we have a fully parameterized code framework. The remaining question is to generate code based on different parameter configurations and identify good parameter values. Naïvely, we can manually try out all possible combinations of parameter values and pick the best one. In practice, this process is very tedious and even unfeasible because of the two reasons. First, the total number of possible combinations usually grow factorially with the number of parameters, making it impossible to explore the complete parameter space. Second, even for a small number of tuning parameters, it will be very time consuming to manually tune parameters when porting code to a new machine.

Auto-tuning is a method that automatically finds out good parameter values using some empirical search schemes. It provides a desirable solution to performance *portability*. It takes non-trivial amount of effort to build an auto-tuner, but once the auto-tuner is built, it can be used to automatically generate high performance code on different machines.

The general auto-tuning methodology has been introduced in Section 3.4. In this Chapter, we will discuss the code generation process, the tuning parameter space

developed for our application, and the search strategy we used in this work. We found that auto-tuning technique can provide up to 50% performance gain over an educated guess for this surface tracking application.

7.1 Code Generation

We need to generate code variants based on different code optimizations and parameters proposed in Chapter 6. Some parameters can be specified through simple C preprocessor flags, for example, tile size $T_h \times T_w$, band radius B_r, polytope size P_h, P_w, and index increments/decrements in the projective time skewing. For more complicated optimizations, we develop specialized code generators in PERL scripting language to automatically generate code variants. Code generator is used to generate the search space of instruction sequences in Section 6.2, and enumerate all possible combinations in the `optscatter` process in the band update in Section 6.4. The code generator saves the painful and error prone process of manually developing C code for similar and repetitive patterns.

One important problem is to verify correctness of the full space of auto-tuner generated code. Our current auto-tuner is not yet capable of automatically checking the correctness of code, thus still relies on programmers to verify it manually. This is not a scalable approach in the long term. In the future, we hope to construct a set of semantic rules of the optimization space, and embed those rules in a domain specific compiler that can automatically generate code by exploring different code transformations and verify the correctness at the same time. Such approach is deployed in more mature auto-tuning systems like FLAME [30] and SPIRAL [40].

7.2 Parameter Space

All optimizations are summarized in Table 7.2. The primary tunable parameters are B_r, T_h, T_w, P_h and P_w. Parameters B_r, T_h, T_w control the tradeoff between the computational part and the band update part. Their optimal values largely depend on architectural properties. For example, Sandy Bridge processor family supports AVX (Advanced Vector Extensions) [8], which defines 256-bit registers that can be used as 8-way single-precision vectors or 16-way single-precision vectors. We would expect larger B_r, T_h, T_w values are preferred on those machines to take advantage of its strengthened SIMD units. The Polytope size P_h and P_w depend on edge distributions in the image as well as last level cache size. The goal is to choose a large enough polytope size to improve data reuse, but the sparse rows held in the Polytope should fit into cache. The nice thing of building an auto-tuner is that it automates the process of finding good parameter values when the underlying algorithm changes, or the input size changes, or the architecture changes.

Some optimizations are controlled with a Enable/Disable parameters. We observe from experiments that choosing Enable will not degrade the performance. The reason we put Enable/Disable tuning parameters here is twofold: first, to understand the effectiveness of the optimization choices; second, to allow easier future extension on the current version because we may want to replace or upgrade certain optimizations.

Empirically, we found for the surface tracking application, there is an up to 50% performance difference between the good parameter values found by the auto-tuner and a fixed educated guess of good parameter values. The educated guess can be taken by domain experts with prior knowledge of good ranges of parameters. However, provided with the diversity and complexity of the hardware architectures, it is extremely difficult to guess or build an analytical model to infer good parameter values on different machines. For example, the vector length in the SIMD instructions

Category	Optimization	Tunable paramters
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
	instruction 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	optscatter	0 (naïve) 1 (SIMDized search) 2 (unrolled `scatter`)
	optgather	0 (naïve) 1 (load 4 `chars` per time) 2 (load 16 `chars` per time)

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

Table 7.1: Summary of Optimizations and Tuning Parameters

will have an impact on the choice of band radius and tile size, which also depends on the effectiveness of other optimizations.

7.3 Search Strategy

Given that the total number of possible combinations of tuning parameters grows combinatorially, an exhaustive search is often not feasible in practice. Instead, heuristics are used to construct a reasonable search space, or to perform a limited search in the complete space, or to guess good parameter ranges learnt from past experience. We use heuristics in the instruction ordering optimization and an iterative greedy search method to sample the overall optimization space. We also discuss machine learning methods which are promising for future work.

Heuristics.

For instruction ordering discussed in Section 6.2, we use simple heuristics to construct a set of valid sequences. The goal is to construct a reasonably large space to explore good tradeoffs between ILP and additional load/store instructions, and hopefully the best sequence resides in this space or stays close. As discussed in Section 6.2, we generate sequences using *replicate* and *interleave* schemes for both stencil kernels: `CompL` and `CompN`. For a given basic tile size $t_h \times t_w$, we simply measure the Gflop/s of all generated sequences under ideal settings described in Section 5.3, and find out the best one for each stencil kernel on every development machine. In later experiments on real input data, a tile size of $T_h \times T_w$ is organized into multiple basic tiles. The basic tile size is chosen such that T_h is a multiple of t_h, T_w is a multiple of t_w, and $t_h \times t_w$ delivers the best Gflop/s under the previous constraint.

Iterative greedy search.

For the overall tuning process, we adopt the iterative greedy search method. The idea is perform greedy *line* search of one parameter while fixing the other parameters, and iterate this process for every parameter. The line search simply means searching for all possible values within the range of that parameter and choosing the best one. The order in which parameters are searched may incorporate some expert knowledge. In our case, we simply organize them category by category as shown in Table 7.2.

The iterative greedy search will find the global optimal configuration if the parameters are independent of each other. However, parameters are unavoidably coupled. For example, tuning parameters T_h, T_w and B_r in the first category largely depends on how well computation part and band update part are optimized, which are closely related to tuning parameters in the last three categories in Table 7.2. Therefore, iterative greedy search could converge to a local minimum. In practice, we run multiple passes of the iterative greedy search. In each pass, we iterate over all parameters once, and use the result as a starting configuration for the next pass. Given the nature of

the iterative greedy search process, the performance will not degrade after each iteration. Usually after 2-3 passes, performance no longer improves and converges to a stable point. The configuration found is used as the final tuned output.

Machine learning.

Some auto-tuning frameworks use machine learning to search the parameter space. Machine learning methods can learn from past experience, and predict for the future. The basic idea is that we could learn the relationship between some features (like the machine characteristics, problem sizes) and good choices of tuning parameters. The SPIRAL project used machine learning method to find the best DSP algorithms. Ganapathi et al. applied machine learning to stencil computation on multicores [11]. We believe that machine learning is a promising research direction for the application of tracking evolving surfaces. The optimization configurations are closely related to image contents, for example, the density of edges. Depending on the applications, tuning for every image may not be feasible. It would be very interesting to learn some useful image features to predict good parameterizations or reduce the size of the search range. For other applications of the level set algorithm, domain specific knowledge may provide useful insights to constrain the search space.

Chapter 8

Performance and Evaluation

8.1 Description

The experimental results are presented in the following order.

1. We construct a kernel code under the ideal settings as described in Section 5.3, and apply all in-core optimizations to the kernel code. The performance measured for the stencil kernel serves as the upper bound of the computational rate. In this experiment, we searches over a set of generated sequences for stencil computation and reports the best performance.

2. We test our auto-tuning framework on both Intel dual-socket Xeon 5560 and Intel Atom 270 machines. We add optimizations proposed in Chapter 6 one by one. After adding each optimization, we report the best delivered performance after tuning all optimizations added so far. This allows us to understand the effectiveness of optimizations under the experimental context. We also report the final computational rate delivered by our auto-tuner, as well as fraction of the runtime spent in the computation part and band update part. Architectural differences and their impact on performance are also analyzed.

3. We discuss the performance result of multithreading, and its implication for future scaling to even larger systems.

4. We discuss the performance gain from tuning, as well as the sensitivity of tuning to input images.

5. Finally, we compare our baseline code with the best publicly available third-party code.

8.2 In-Core Stencil Kernel Performance

As discussed in Section 5.3, the kernel code explores how to maximize utilization of the on-chip hardware resource. It is tested under the setting of full image grid without narrow band representation, and a small enough data set size that fits into the last level on-chip cache.

We apply all in-core optimizations discussed in Section 6.2, and report performance results using basic tile size $t_h \times t_w$, where $t_h \in \{1, 2, 4\}$ and $t_w \in \{4, 8\}$. In `CompN`, a superpixel of 1×4 pixels performs 4 vector `ADD` and 4 vector `MUL`, so the theoretical peak performance 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 14 vector `MUL`, so the theoretical peak performance is 3.5 cycle/pixel on Xeon, and 6.75 cycle/pixel on Atom.

Table 8.1 summarizes measured kernel performance for `CompN` and `CompL`, and the corresponding Gflop/s. The kernel code runs at 50% of 22.4 Gflop/s machine peak on Xeon and 25% of 6.4 Gflop/s machine peak on Atom. The gap between the achieved Gflop/s and peak Gflop/s is mainly due to pipeline stalls caused by the complex data dependencies in the stencil kernel, and additional memory transfer instructions caused by register spills/reloads.

There are two main reasons why a higher fraction of peak performance is attained on the Xeon. First, the Xeon uses aggressive out-of-order cores, and has many more physical 128-bit registers in the physical register file. Second, the compiler sees sixteen `xmm` registers on the 64-bit Xeon, but only 8 `xmm` registers on the 32-bit Atom. Therefore, the assembly code on the Atom has far more register spill/reload instructions than on the Xeon. This can be seen from the additional memory transfer ratio in Fig 8.1.

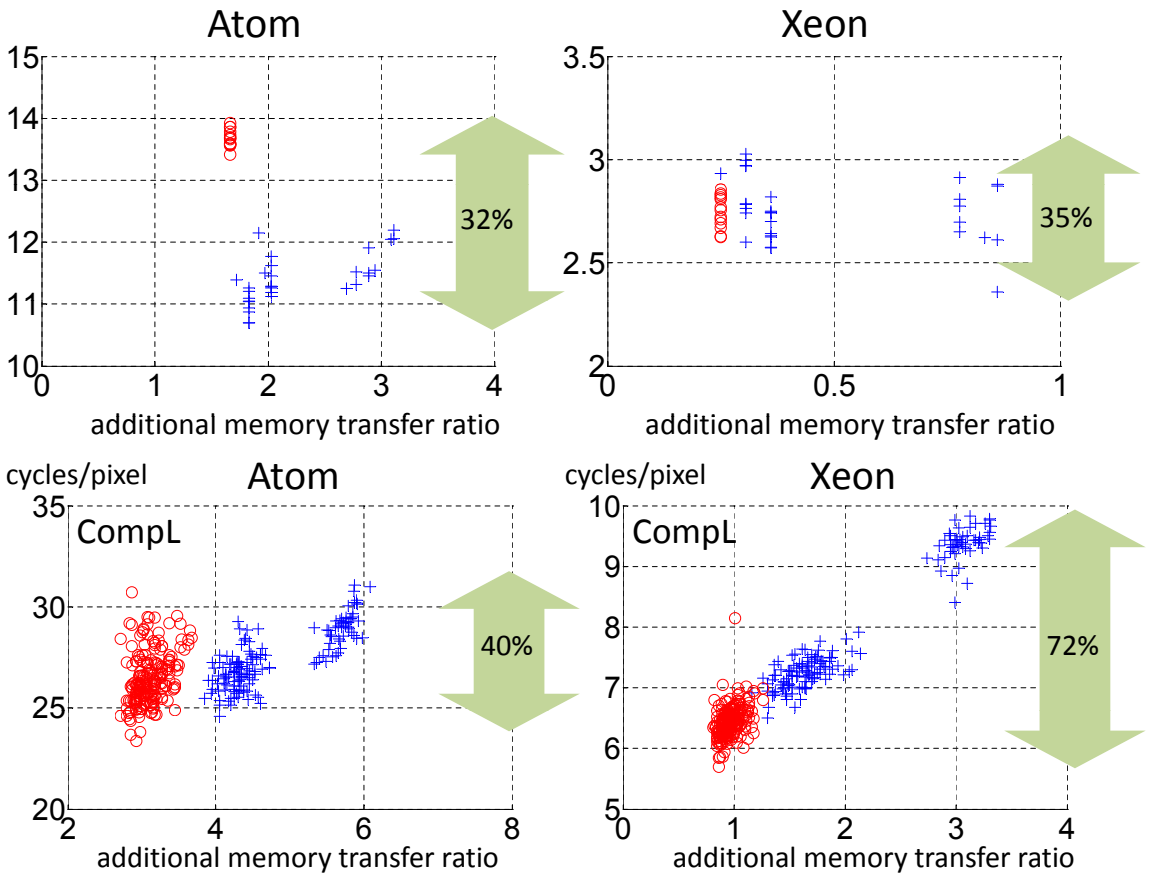

Figure 8.1: Performance of different instruction sequences vs. *additional memory transfer ratio* for basic tile size of 4×8.

	$t_h \times t_w$	1×4	2×4	4×4	1×8	2×8	4×8
Atom	`CompN` (cycle/pixel)	12.8	9.0	9.5	9.9	10.1	10.7
	`CompL` (cycle/pixel)	26.9	23.7	23.1	23.8	24.3	23.4
	Aggregate Gflop/s	1.4	1.7	1.7	1.7	1.6	1.6
Xeon	`CompN` (cycle/pixel)	3.0	2.7	2.5	2.6	2.5	2.4
	`CompL` (cycle/pixel)	6.7	6.2	5.8	5.91	5.79	5.70
	Aggregate Gflop/s	10.1	11.0	11.7	11.5	11.8	12.2

Table 8.1: Stencil Kernel Performance for `CompN` and `CompL`, and corresponding aggregate Gflop/s.

Fig 8.1 shows the performance versus *additional memory transfer ratio* of different instruction sequences. The *additional memory transfer ratio* is defined as

$$\left(\frac{\text{\# of loads and stores in the assembly code of the long basic block}}{\text{\# of loads and stores in the original C code in the long basic block}} - 1 \right).$$

As expected, the *interleave* scheme generates more register spills/reloads compared to the *replicate* scheme. In general, *interleave* performs better on Atom than on Xeon, because it explores some level of inter-pixel ILP. The optimal sequences for `CompN` use *interleave* on both machines, while `CompL` use *replicate*. This could be because DDG `CompL` is more complicated and has better intra-pixel ILP. Also, we can see from Table 8.1 that using larger basic tile size has about 10%–20% performance gain over using the basic tile size of 1×4.

8.3 Auto-tuning Performance

8.3.1 Xeon

Fig 8.2 shows the performance result delivered by our auto-tuner on Xeon for square image sizes varying from 64 to 8,192 (the input data is generated by scaling the example image in Fig 2.3). In this plot, we show the relative speed-up over our baseline code by adding the optimizations one by one, in the order of in-core stencil,

memory level, and band update optimizations. The baseline is a straight-forward implementation of the pseudo code described in Section 4.2. It is scalar C code, with tile size 1×1 and $B_r = 1$. Every time a new optimization is added, we perform three passes of iterative greedy search, as discussed in Section 7.3. Therefore, each bar in the figure should be interpreted as the best performance with all optimizations so far in the search space. The speedup numbers are summarized in Table 8.2.

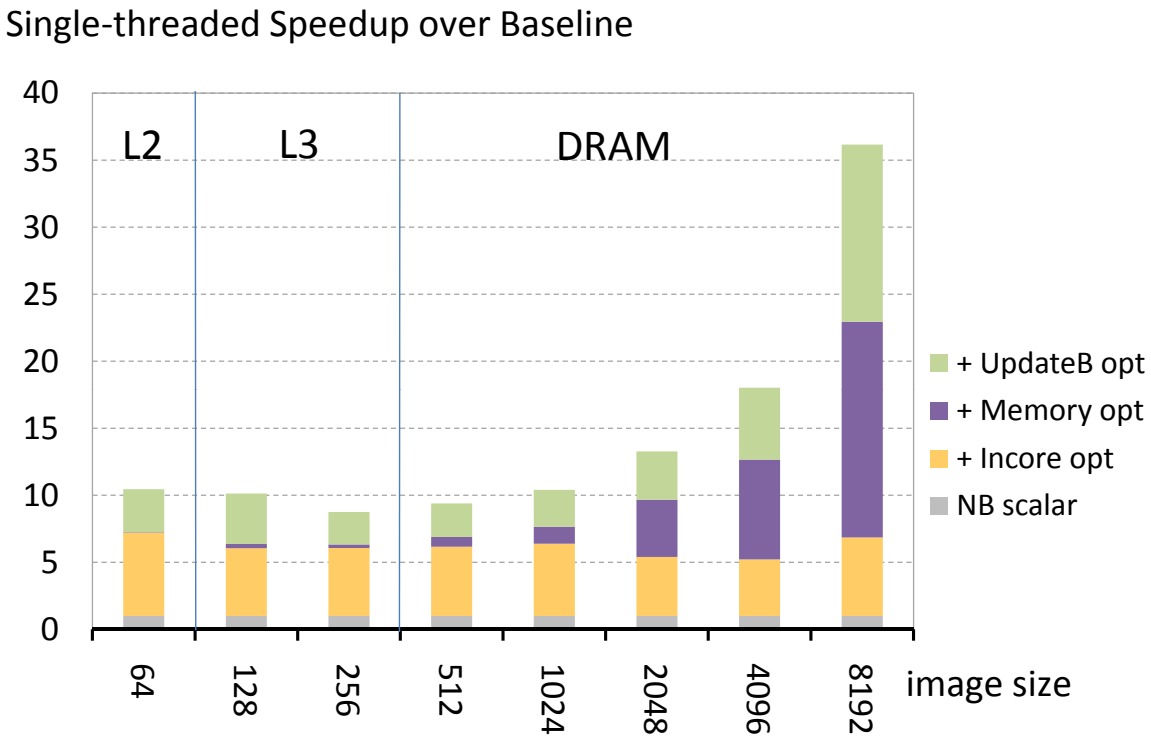

Figure 8.2: Single-threaded speedup over scalar C code baseline on Xeon. Each point in the figure should be interpreted as the optimal performance with all optimizations so far. Scalar code on full image grid is also shown for comparison. (Best view in color)

It can be observed from Fig 8.2 that 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 polyhedral transform plays an important role to remove the memory bottleneck.

Image Size	64	128	256	512	1024	2048	4096	8192
full grid	0.42	0.22	0.12	0.10	0.09	0.07	0.04	0.03
narrow band baseline	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
+ in-core opt	7.20	6.05	6.07	6.16	6.39	5.40	5.22	6.85
+ memory opt	7.24	6.36	6.34	6.89	7.66	9.67	12.66	22.95
+ band update opt	10.45	10.13	8.75	9.39	10.40	13.27	18.03	36.16

Table 8.2: Single-threaded speedup over scalar C code baseline on Xeon.

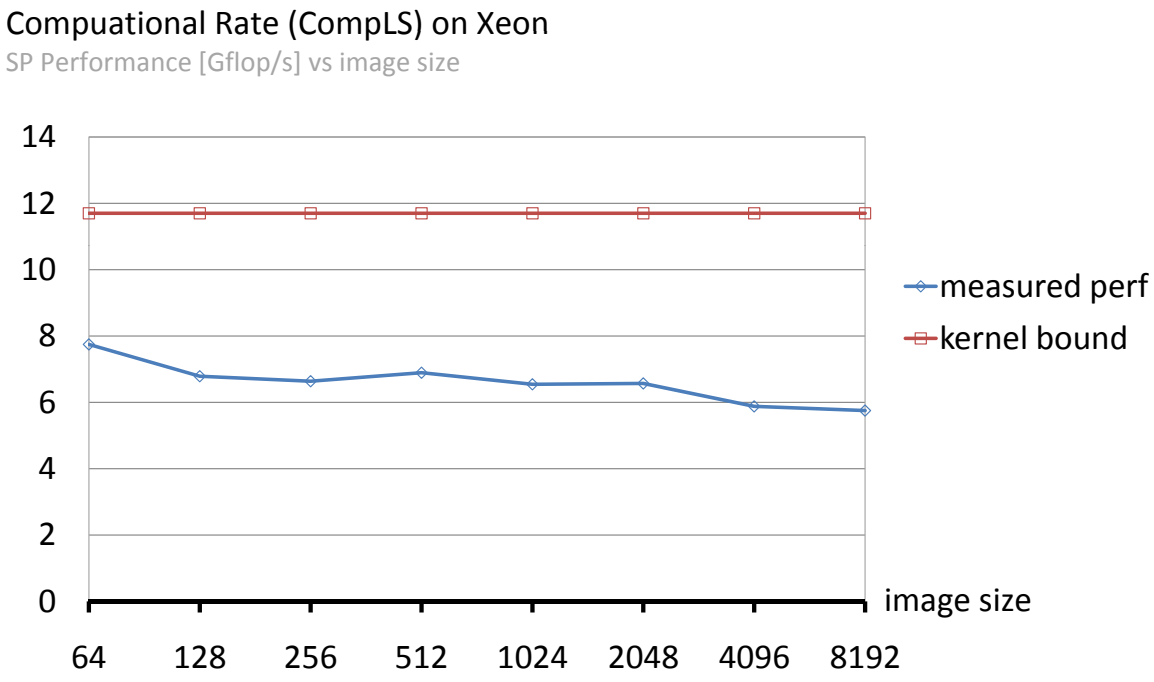

Figure 8.3: Auto-tuner delivered computational rate on Xeon. Kernel stencil performance is shown for comparison. The basic tile size used in kernel stencil is the same $t_h \times t_w$ found by the auto-tuner for any image size

Image Size	64	128	256	512	1024	2048	4096	8192
Kernel Computational Rate	11.7	11.7	11.7	11.7	11.7	11.7	11.7	11.7
Measured Computational Rate	7.75	6.79	6.64	6.89	6.55	6.57	5.88	5.75

Table 8.3: Auto-tuner delivered computational rate on Xeon (in Gflop/s).

Fig 8.3 shows the computational rate in Gflop/s measured for the computation part (`CompLS`), which is about 26%–35% of the machine peak for varying input sizes.

The real numbers are summarized in Table 8.3. The performance is almost flat when image sizes grow out of cache. So the application is close to compute-bound. The gap between the kernel performance and the measured performance is due to overhead of the indirect memory access using the CSR format. We test the kernel code using the CSR format for the complete image, and got almost the same computational rate.

It is difficult to characterize the code efficiency for `UpdateB`, because the main body in both `scatter` and `gather` consists 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. On average, `UpdateB` spends 80% of time in `scatter` and 20% of time in `gather`. In `scatter`, it takes about 29 cycle/entry in the `switch` statement, and writes to B_I takes about 13 cycle/write. In `gather`, on average it takes 30 cycle/entry in the `switch` statement, and writes to B_{list} takes 23 cycle/write. The performance is most likely limited by the unpredictable control flow.

8.3.2 Atom

Fig 8.4 shows the performance result delivered by our auto-tuner on Atom for square image size varying from 64 to 4,096. The plot is similar to Xeon, where for small image size that can fit into L2, in-core and band update optimizations are most important, and for large image size, the polyhedral transform plays a key role to reduce the memory pressure. The real numbers are summarized in Table 8.4.

Fig 8.5 also shows the computational rate on Atom, which is about 12%–20% of the machine peak for varying input sizes. The real numbers are summarized in Table 8.5. The performance degrades as the image size increases, indicating the polyhedral transform cannot completely remove the memory bottleneck.

For the band update, similar to Xeon, 80% of the time is spent in `scatter`, and 20% in `gather`. On average, in `scatter` it takes 50 cycle/entry, and writes to B_I

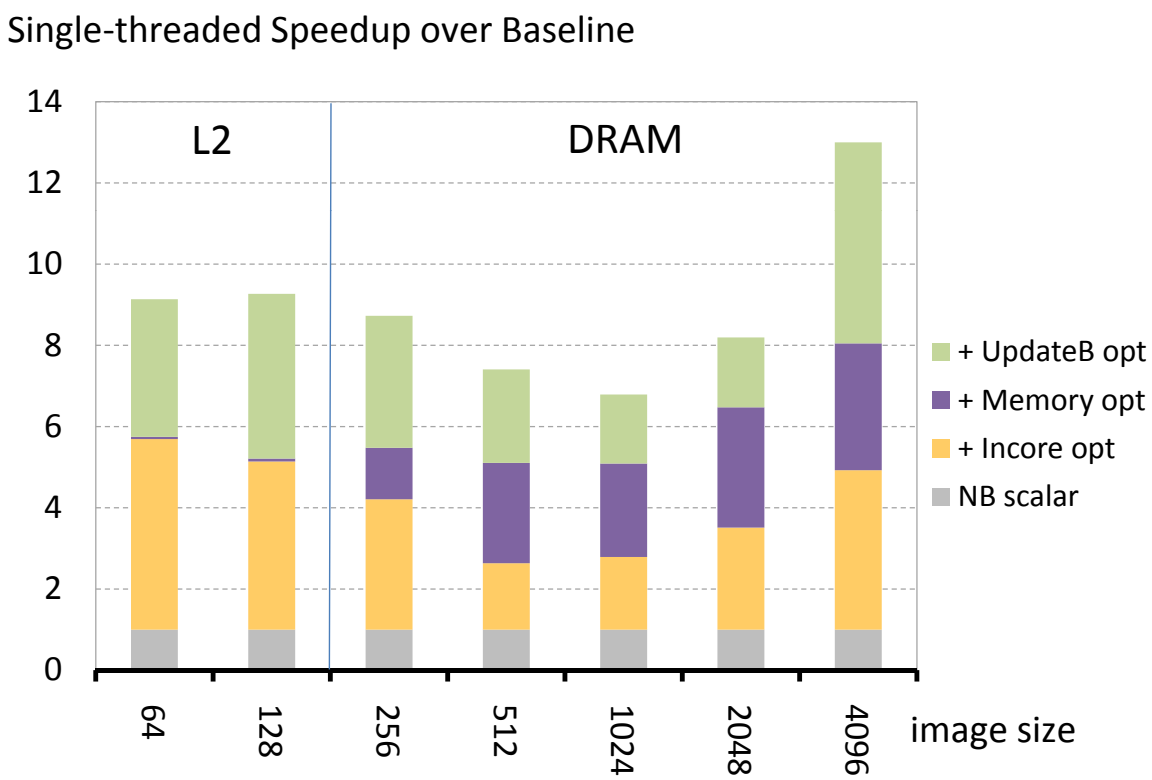

Figure 8.4: Single-threaded speedup over scalar C code baseline on Atom. (Best view in color)

Image Size	64	128	256	512	1024	2048	4096
full grid	0.37	0.20	0.13	0.10	0.07	0.04	0.04
narrow band baseline	1.00	1.00	1.00	1.00	1.00	1.00	1.00
+ in-core opt	5.69	5.14	4.21	2.64	2.79	3.51	4.93
+ memory opt	5.75	5.21	5.48	5.11	5.09	6.47	8.05
+ band update opt	9.14	9.27	8.73	7.41	6.79	8.19	13.00

Table 8.4: Single-threaded speedup over scalar C code baseline on Atom.

Image Size	64	128	256	512	1024	2048	4096
Kernel Computational Rate	1.7	1.7	1.7	1.7	1.7	1.7	1.7
Measured Computational Rate	1.26	1.19	0.94	0.82	0.90	0.82	0.79

Table 8.5: Auto-tuner delivered computational rate on Atom (in Gflop/s).

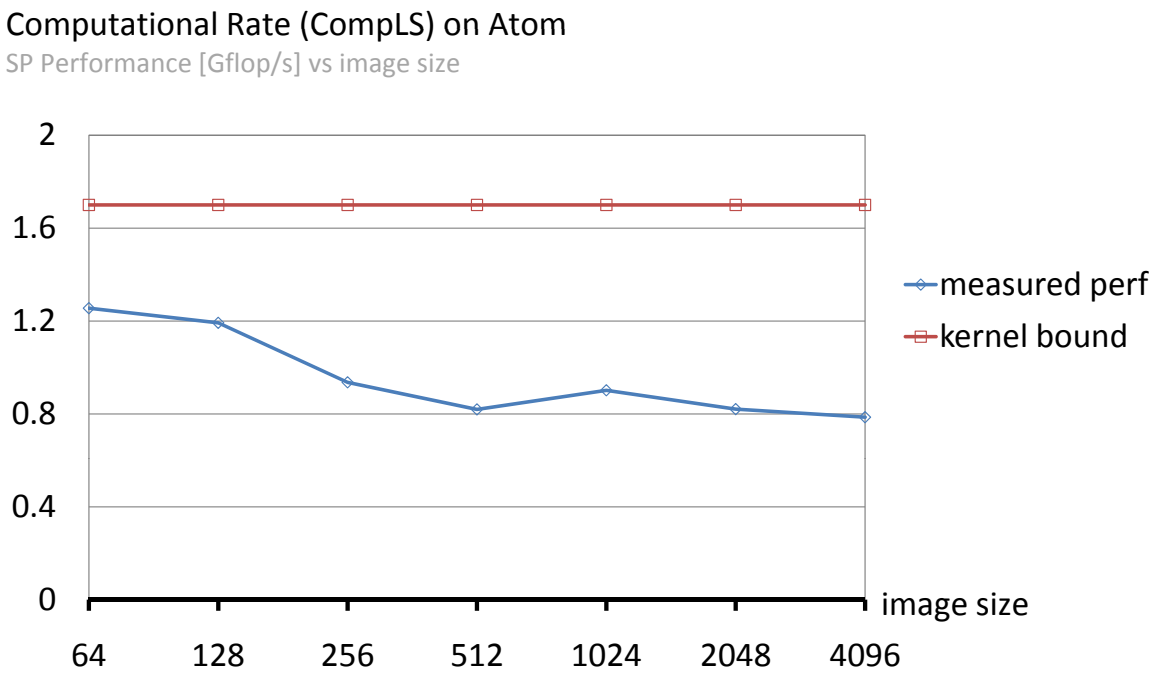

Figure 8.5: Auto-tuner delivered computational rate on Atom. Kernel stencil performance is shown for comparison. The basic tile size used in kernel stencil is the same $t_h \times t_w$ found by the auto-tuner for any image size.

takes 10 cycle/write. In `gather`, it takes 36 cycle/entry, and writes to B_{list} takes 28 cycle/write.

8.3.3 Fraction of Computation Part and Band Update Part

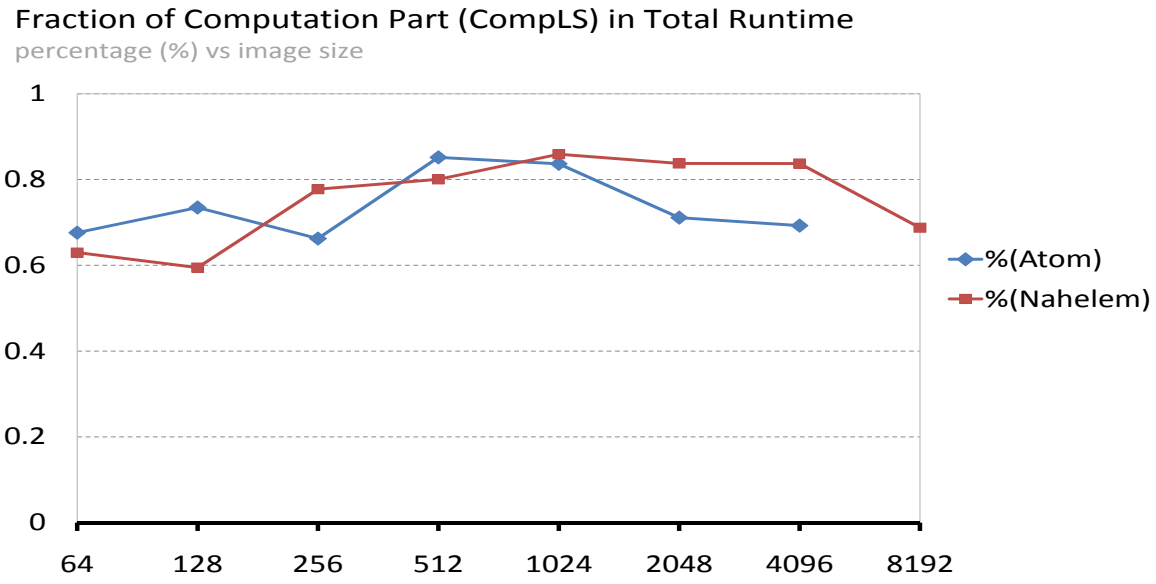

Figure 8.6: Fraction of `CompLS` in total runtime after tuning.

Fig 8.6 shows the fraction of `CompLS` and `UpdateB` after full tuning. For most of the cases, computation takes 80% of total runtime. Given the computational rate achieved, there's little headroom for further improvement. As discussed in section IV, the optimal tradeoff between `CompLS` and `UpdateB` depends on how well each part is optimized. Further optimization should be focusing on the band update part, if possible.

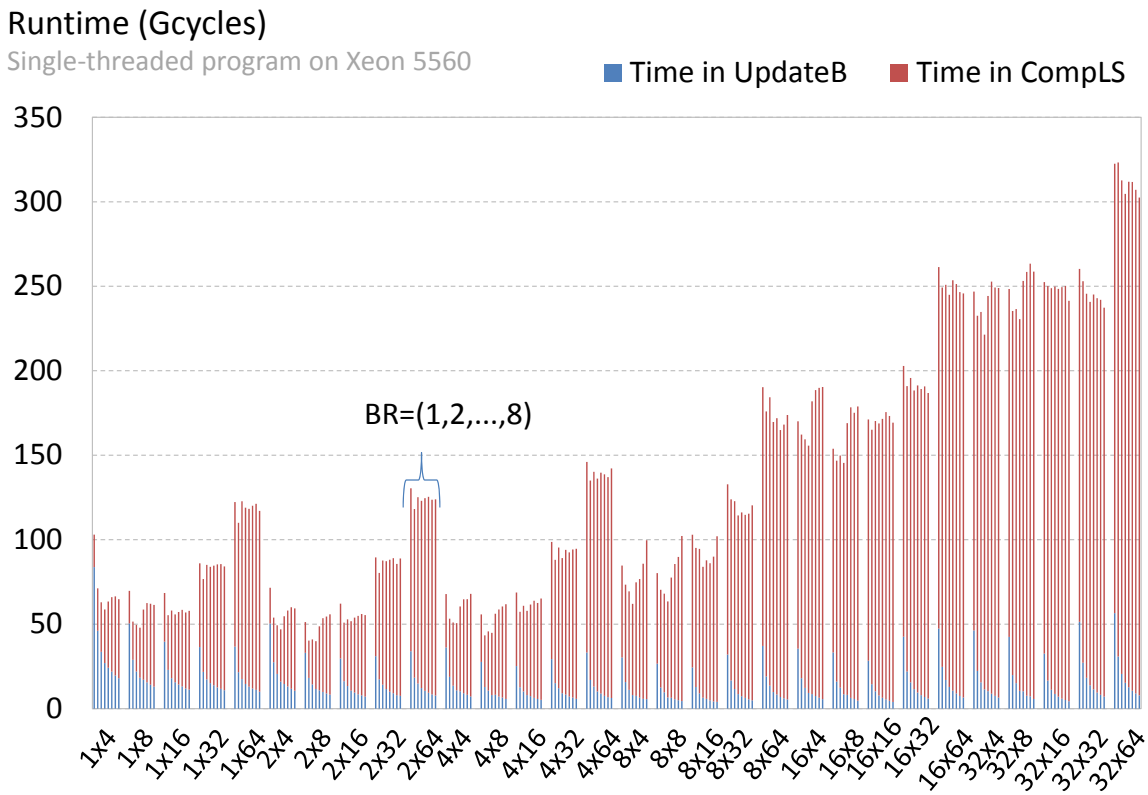

Figure 8.7: An example of decomposition of runtime in `CompLS` and `UpdateB`, with varying $T_h \times T_w$ and B_r. Each group corresponds to a fixed tile size while B_r varies in {1, 2, ... ,8}.

Fig 8.7 shows an example of the decomposition of measured runtime with varying tunable parameters $T_h \times T_w$ and B_r. The result is tested for image size of 4,096 × 4,096, with full optimizations. As discussed in Section 6.1, the tradeoff between `CompLS` and `UpdateB` is controlled by the three parameters. It is observed that increasing the tile size quickly lengthen the runtime spent in `CompLS`. We found that on both machines, the optimal tradeoff usually corresponds to a relatively small tile size like $2 \times \{4, 8\}$ and $B_r \in \{2, 3, 4\}$. Therefore, we fix $T_h \times T_w$ to 2×4 and B_r to 2, as a reasonably good benchmark of performance without auto-tuning, which will be used in Section 8.5.

8.3.4 Multicore Parallelization Results

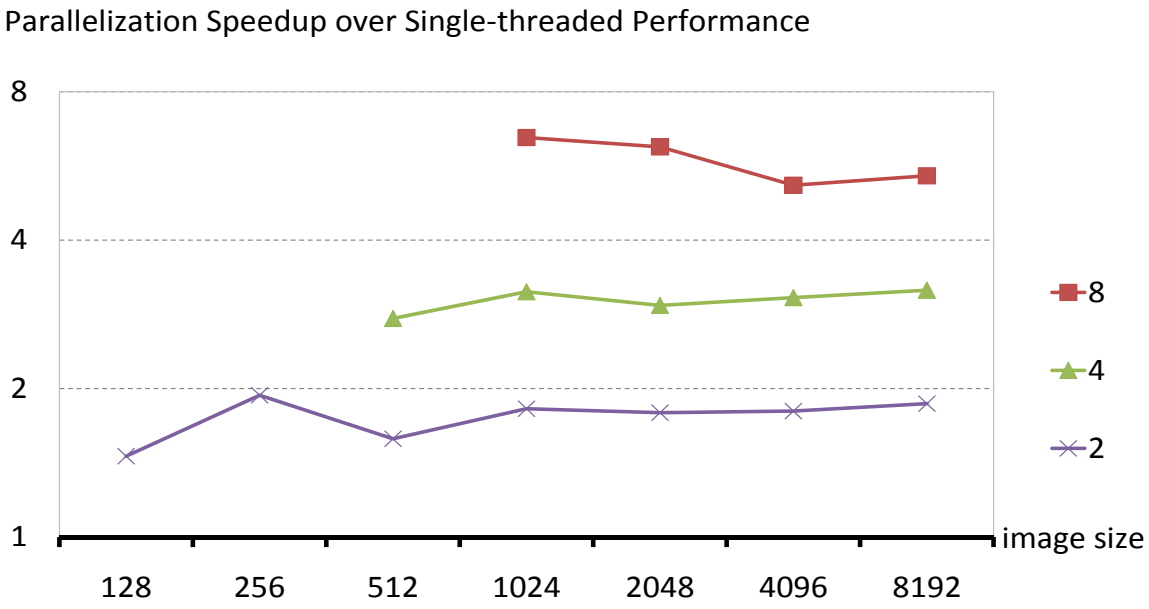

Figure 8.8: Parallelization speedup on Xeon using 2, 4 and 8 cores.

Fig 8.8 shows the parallelization result on the dual-socket Xeon. We constrain the number of rows processed by any core to be at least $2(\Delta_p + \Delta_s)$ to make sure prologues and epilogues do not overlap. For small image sizes, the number of rows per core may be too small to satisfy the constraint, therefore we do not report any performance number for this case.

For a very small image size such as 128, 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 speed-up for 2 and 4 cores. When scaling from 4 cores (single-socket) to 8 cores (dual-socket), we observe a degradation in speed-up 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. Given the high data transfer rate offered by the QPI link between the two sockets, latency of transferring data between DRAM attached to the first socket and processor in the second socket could be the major cause of performance penalty observed.

With NUMA-aware memory management, it is possible to relieve the pressure by utilizing DRAM bandwidth in the second socket. This could possibly lead to some further speed-up using 8 cores for large image sizes, but requires non-trivial effort to reduce the overhead along the partition boundary between the two sockets.

The compound speedup with single-threaded optimizations and multithreading over the baseline code is shown in Fig 8.9, and summarized in Table 8.6, ranging in 14–195x.

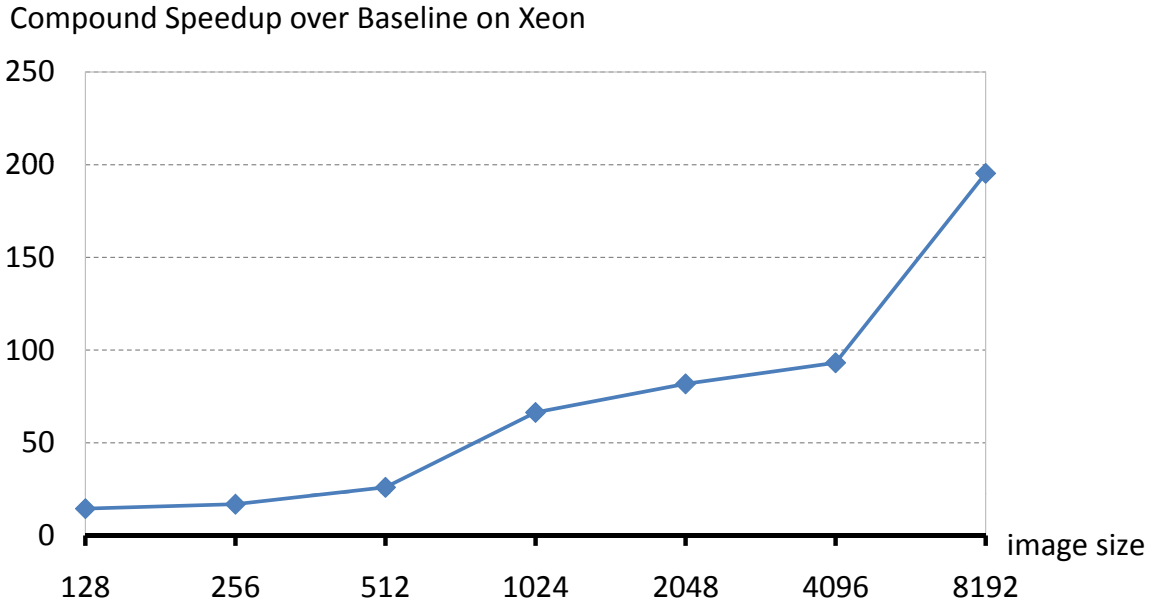

Figure 8.9: Full speedup over baseline code with multithreading up to 8 cores.

Image Size	128	256	512	1024	2048	4096	8192
Full speedup over Baseline	14.48	16.89	25.98	66.37	81.8	93.19	195.36

Table 8.6: Full speedup over baseline code with multithreading up to 8 cores.

We did some experiments to understand how performance varies with the polytope size when using 1, 2, 4 and 8 threads. Fig 8.10 shows the result for image size 8,192× 8,192, with fixed $T_h \times T_w = 1 \times 4, B_r = 2$, and $P_w = 4$. According to the analysis in Section 6.3, the miss rate of LLC is inversely proportional to P_h, as long as P_h

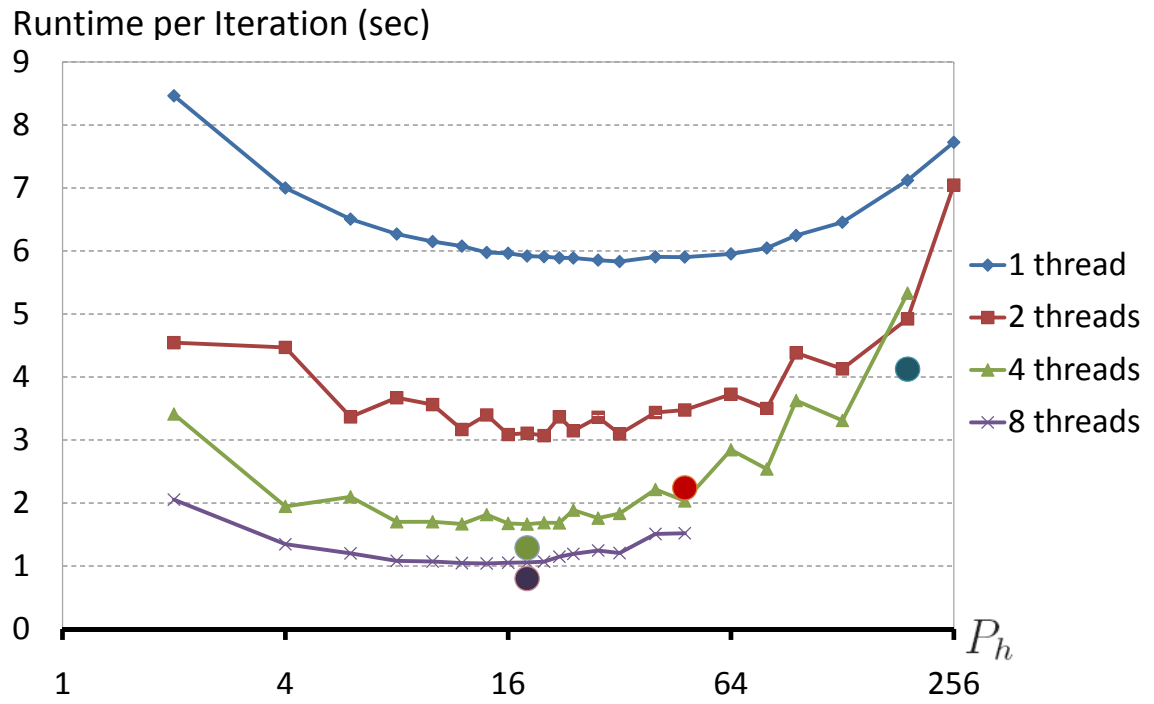

Figure 8.10: Performance sensitivity to P_h, with fixed $P_w = 4$. The result shows for image size 8,192× 8,192, with $T_h \times T_w = 1 \times 4$ and $B_r = 2$, the performance of using 1, 2, 4, and 8 threads. The fat dots show the optimal performance of using 1, 2, 4, and 8 threads, and corresponding P_h after full tuning.

sparse rows from each thread can fit into LLC. Therefore, when the number of threads doubles, the range of P_h delivering good performance is narrowed—roughly speaking, the lower end of P_h is doubled to halve the LLC miss rate per thread, and the upper end of P_h is halved for fitting into the LLC capacity. When scaling to two sockets, only the lower end gets doubled because LLC capacity doubles. This explains what we observed in Fig 8.10. The fat dots in the figure shows the optimal performance and its P_h after auto-tuning. For fewer number of threads, its larger range of the good polytope size indicates more flexibility in the search space, which explains why the performance gain from tuning is larger. From this result, we learned that when scaling to even larger systems with more processors, utilizing bandwidth of every processor is important. Simply replying on polyhedral transform to reduce memory traffic is not a scalable solution. For large systems, additional code transformations that handle

NUMA-aware data allocation and reduce communication overhead among processor will ne necessary for good performance.

8.4 Architectural Comparison

In general, for single-threaded performance Atom is about 4.7x slower then Xeon5560 for small data set sizes that fit into on-chip cache, and 5–6x slower for large data set sizes that cannot fit.

There are two reasons for the 6x slow down for small data size that fit into last level cache. First, Atom has a peak single-threaded Gflop/s rate of 6.4, which is 28.6% of the peak Gflop/s on Xeon. Second, Nehalem is capable to effectively hide data dependency by OoO engine, while the in-order Atom need to fill in much more register spill/reload instructions for higher ILP to avoid pipeline stalls. This explains why stencil kernel achieves about 50% of peak on Xeon, but only 20%–30% on Atom.

The performance gap enlarges as the image size increases. Xeon can maintain almost flat performance with increasing image size. It has a large enough (8M) L3 to decrease the LLC miss rate such that the application is compute-bound. On Atom, with the best polyhedral transform size, L2 miss rate can be effectively reduced, however not to the extent that can completely remove bandwidth bottleneck. In addition, the lack of large page support on Atom results in increasing page miss penalty as the input image size increases.

8.5 Performance Gain from Autotuning

To understand the benefits of auto-tuning, 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. Fig 8.11 shows

the speed-up of fully tuned code over the educated guess, which is 12% on Xeon and 17% on Atom in average.

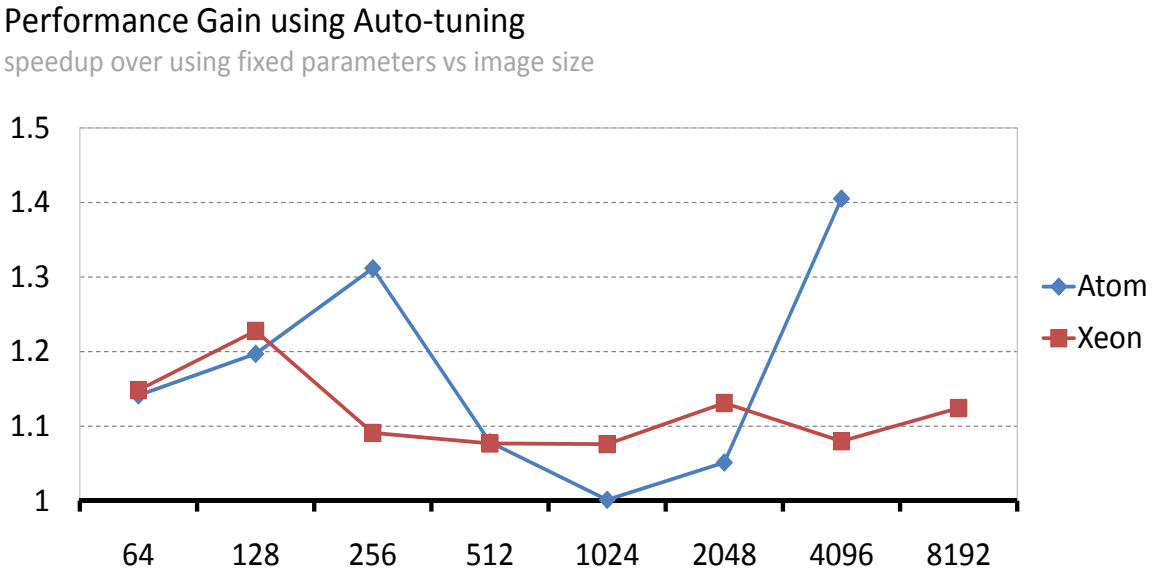

Figure 8.11: Speed-up of fully tuned code over using fixed optimization parameters from an educated guess.

8.6 Sensitivity to Input Data

Figure 8.12: Four input images used in sensitivity test of the auto-tuning parameters.

In Fig 8.13 and Fig 8.14, we shows how sensitive the performance result is to different image inputs on Xeon and Atom. We use four different images as our inputs, as shown in Fig 8.12. For each image, we perform the auto-tuning process to find out the optimal parameter for it. And we do a cross-test to apply the optimal

parameters found by one image to another. In both figures, the number in j-th row i-th column shows the runtime when using the optimal parameters of image i on image j, normalized to the optimal runtime of image j.

8.7 Comparison to Third-Party Code

The best publicly available code for comparison is the C code provided by Li. et at[16] in the form of a pre-compiled dll file called through a Matlab interface. Their code is close to a straight forward C implementation of the algorithm. Due to portability issues of the dll file, we only got their code to run on a Core 2 Extreme machine with 64-bit Windows Vista. The speed-up of our scalar C code baseline over their code is plotted in Fig 8.15. The speed-up is about 1.3–2.0x when compiled using MS Visual Studio compiler 32-bit, and 2.0–3.0x when compiler with ICC 32-bit or 64-bit.

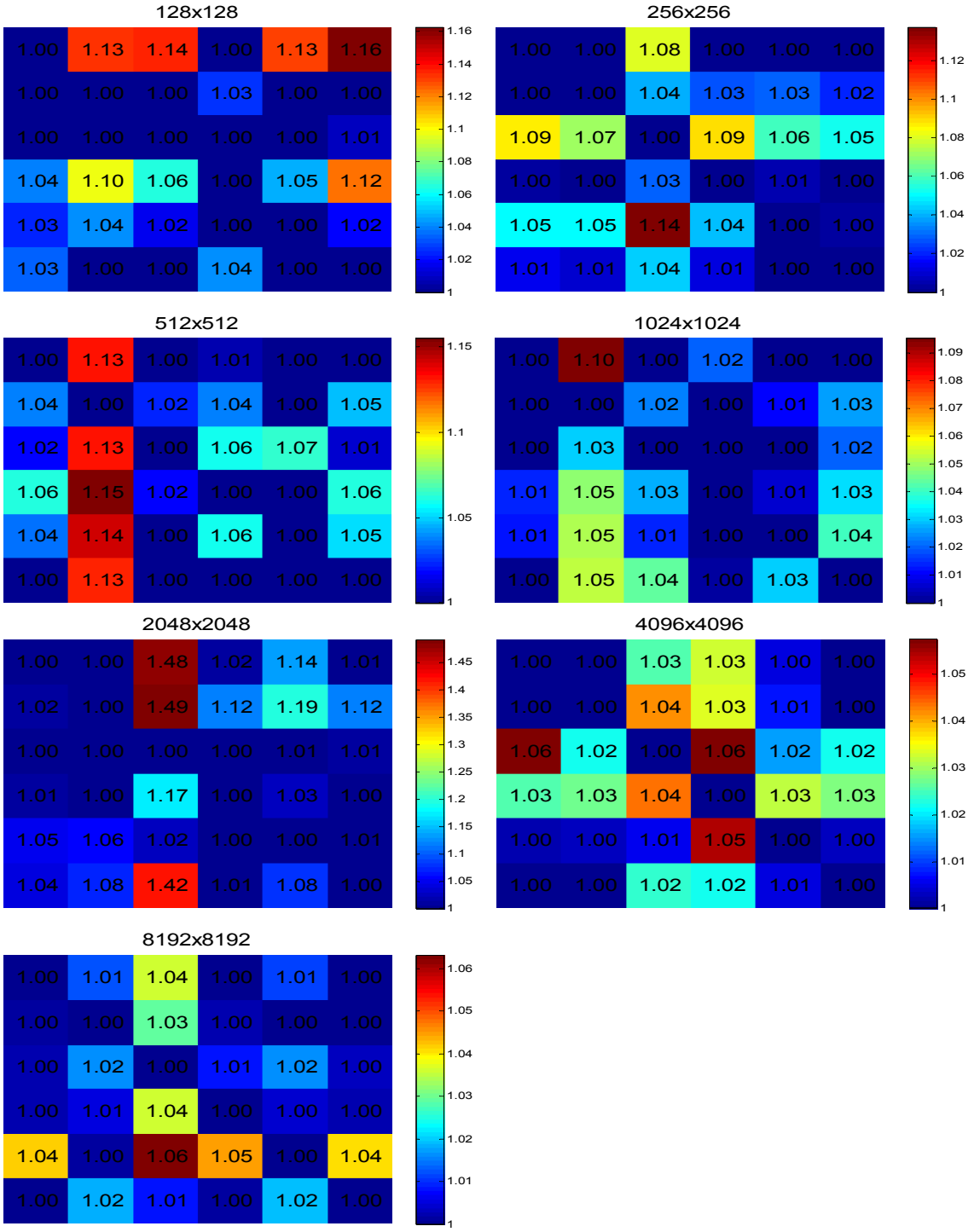

Figure 8.13: Sensitivity of applying the optimal tuning parameters from one image to other images on Xeon. The number in j-th row i-th column shows the runtime when using the optimal parameters of image i on image j, normalized to the optimal runtime of image j. Four test images are shown in Fig 8.12.

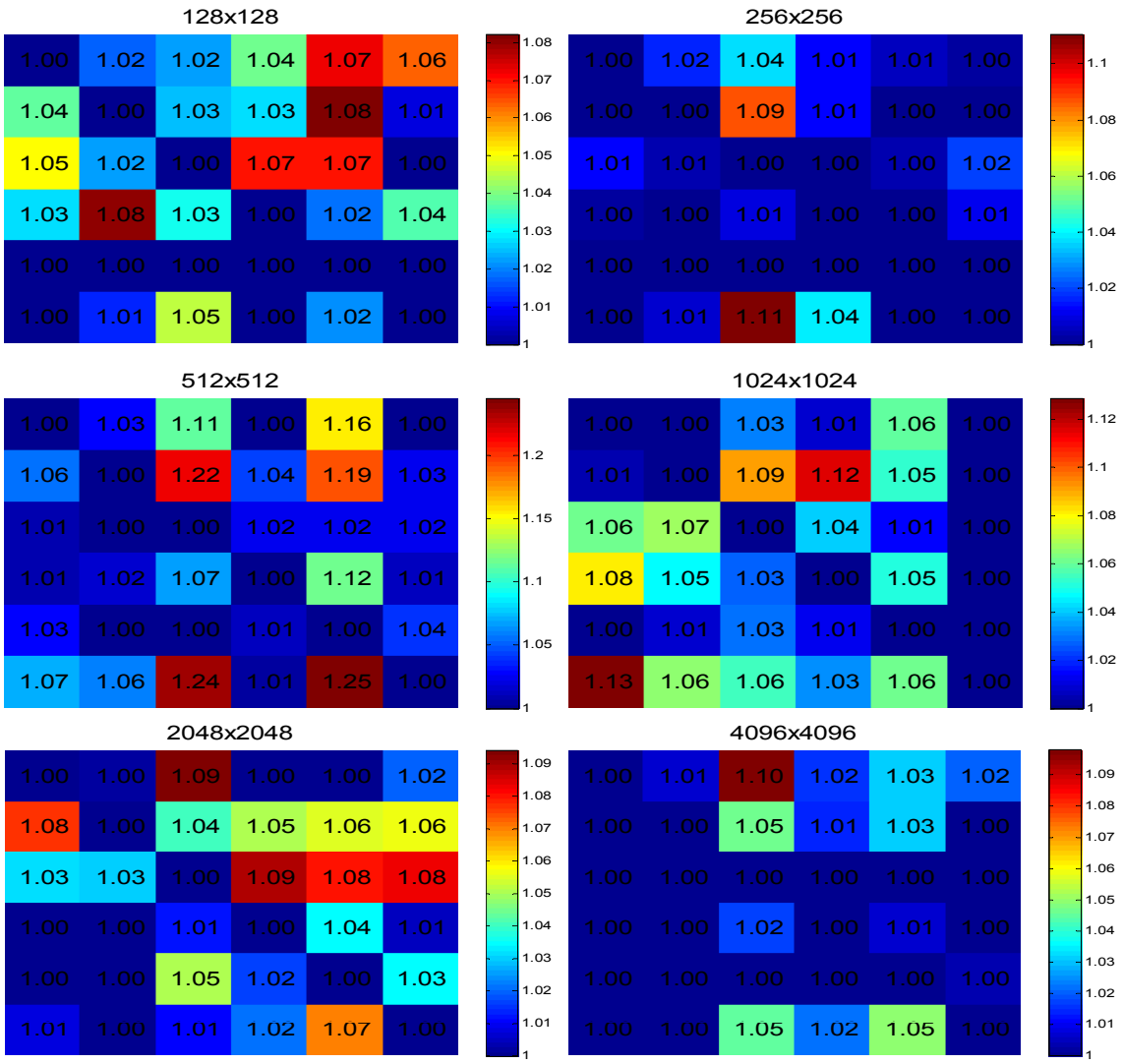

Figure 8.14: Sensitivity of applying the optimal tuning parameters from one image to other images on Atom. The number in j-th row i-th column shows the runtime when using the optimal parameters of image i on image j, normalized to the optimal runtime of image j. Four test images are shown in Fig 8.12.

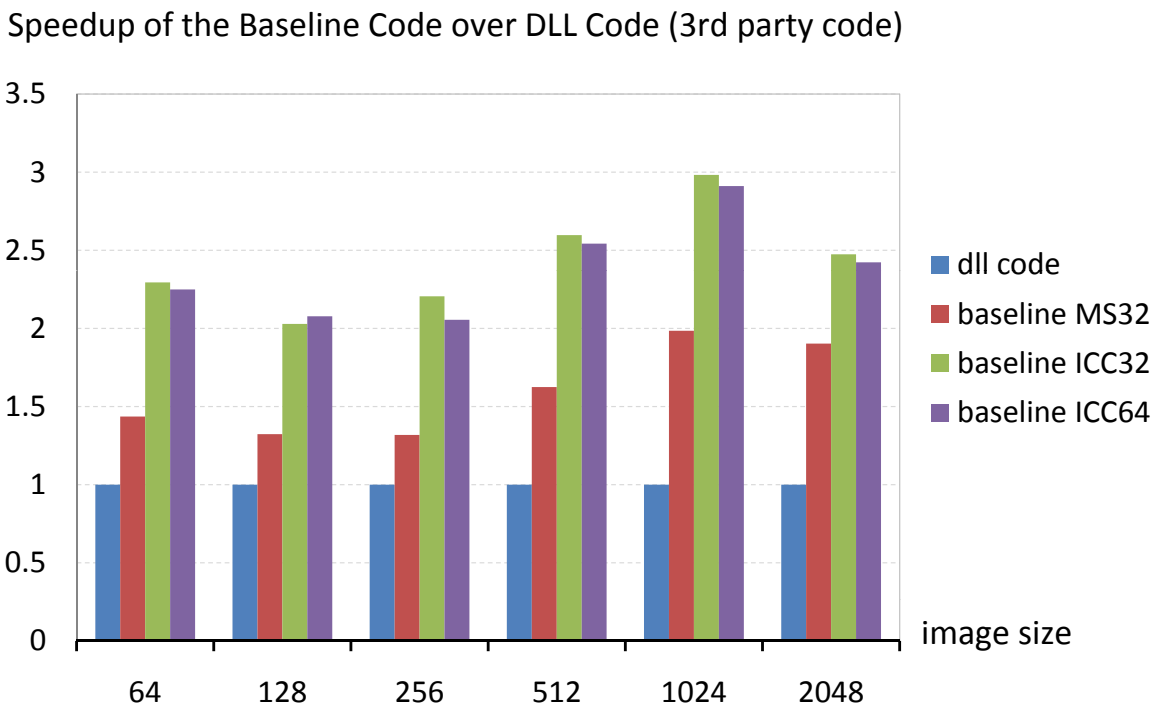

Figure 8.15: Speedup of our baseline C code over the best publicly available code. The result is shown when our baseline is compiled using three different compilers: MS Visual Studio 32-bit, ICC 32-bit and ICC 64-bit.

Chapter 9

Future Work

This thesis investigates performance optimization opportunities for an important real world application — tracking motion of irregular surfaces. This application represents an interesting computational pattern featured by massive data parallelism plus data dependent control flow. We proposed a set of effective optimizations for this computational pattern on current multicore platforms. The work presented in this thesis serves as a prelude to a set of related research problems, rather than a thorough solution.

In the future, we hope to generalize the ideas proposed in this thesis to a broader set of similar computational problems. Some open problems remained to be explored are listed as following.

1. The current auto-tuning framework is designed for the specific stencil kernel used in image segmentation. Changing the stencil kernel will involve manual effort in modifying the in-core stencil optimizations as well as the formulation of polytopes in the time-skewing technique. A better solution would be to extract semantic structures or rules as basic building blocks of the auto-tuning framework. This approach is taken in more developed auto-tuning systems. For example in SPIRAL, a specially designed compiler searches various transformations and generates target

code by manipulating a domain specific language called SPL (Signal Processing Language). The SPL formulas or structures can be pre-verified, thus it can automatically guarantee correctness of the generated code. Formalizing semantic rules also allows easier and more structural exploration of algorithmic transforms. For example, replacing stencil kernels with different algorithms discussed in Section 2.3 will be fully automatic. In the future, we hope the auto-tuning framework will be able to take in a stencil kernel in the form of a DDG (data dependency graph), and automatically generate and search code variants based on optimization rules.

2. Many interesting surface tracking problems requires computation in a higher dimensional space. By projecting the narrow banded grid into a lower dimensional space, we have shown the irregular stencil computation can be solved in a way similar to the stencil computation on a dense regular grid. Existing polyhedral transform frameworks can handle high dimensional regular grids well. We hope to combine the idea of projection and some components from the polyhedral transform library to build a general framework that can track surface motion in higher dimensions.

3. Depending on the application area, the range of good parameterizations may vary and relate closely to the property of the force field that drives the surface evolution. This currently is not a big issue for our image segmentation problem, but may be important when the input force field is diverse and no unique parameterization of the tuning parameters can work reasonably well in general case. It is usually infeasible to tune for each possible input data. Under such cases, machine learning methods can be helpful in learning relationship of some important features and good values of the tuning parameters. For example, edge density is important in choosing the polytope size in the time skewing technique. We can sample the image and compute the edge strength to generate some heuristics, learn how those

heuristics affect the choice of parameters through some examples, and predict good parameter values or their ranges when presented with new examples.

Chapter 10

Conclusion

Developing highly efficient computational code on modern multicore CPUs is a hard problem. Performance portable implementations that deliver high machine utilization across multiple machines usually require thorough exploration of useful code transformation techniques based on the application domain, in conjunction with auto-tuning and program generation approaches.

In this thesis, we present a methodology to deliver highly efficient performance portable implementations of the surface tracking application. The surface motion is tracked by the narrow band level set algorithm, characterized by iterative stencil computation on a dynamically evolving narrow band, which is a sparse sub-set of grid points held in a neighborhood region around the evolving surface in a regular dense grid. we developed a novel projective time skewing technique to extract data reuse for this irregular algorithm with data-dependent control flow. In addition, we applied other code transformations aggressively to address different system bottlenecks. These include in-core stencil optimization, lower level memory optimizations and parallelization, with focus on utilizing in-core resources, reducing page miss penalty and conflict misses, and achieving scalable performance across multicores, separately. We

built an auto-tuning framework to find good parameterizations for our code transformations.

We demonstrate the effectiveness of the proposed optimization methodology through a 2D image segmentation example. The fully optimized code shows up to 195x speed-up over our scalar C baseline code on a dual-quadcore Xeon system, reaching 26%–35% of the machine peak performance across a wide range of problem sizes. On the low-end power-efficient, single-core Atom, the fully tuned code achieves up to 13x speedup over the baseline code, reaching 12%–20% of the machine peak performance. Speed-up of our baseline code over the best publicly-available third-party implementation ranges in 1.3–2.0x for various input sizes. The primary reasons for the performance gap between Xeon and Atom are the power of the core microarchitectures, how efficiently they tolerate cache misses, and the last level cache sizes. On Xeon, with the projective time-skewing technique, the application becomes completely compute-bound, even for very large input sizes. Thus there is little head room for further performance improvement. On Atom, the memory bandwidth pressure cannot be fully hidden for large input sizes, because of the limited size of the last level cache, the less efficient tolerance scheme for cache miss, and the TLB miss penalty. Re-organizing the data structure could possibly lead to further performance gain by hiding TLB miss penalty. This can be implemented by copying a broader area of tiles that cover the current narrow band to a continuous region in memory. Whenever the narrow band evolves outside the area, we need to repeat this process of copying-to-continuous-memory. The tradeoff between the benefit and cost of this method is left to be explored.

The current architectural trend indicates the disparity between processor computational rate and memory bandwidth will continue to increase. More and more cores will be integrated onto one chip, delivering higher computational rate. However, the memory bandwidth grows much slower, indicating bandwidth per core will become

increasingly scarce. In the dual-quadcore system, we observed that the range of the polyhedral size that can completely hide the memory bottleneck is getting more and more constrained as the number of cores increases. The architectural trend implies there is a limit to which the projective time skewing technique can hide the memory bandwidth bottleneck. Most likely, performance will be memory bound and some cores will become effectively unused on many core platforms in the future.

Stencil kernel in other applications may exhibit a different ratio between arithmetic operations and memory transfers. So the application can be compute bound, or memory bound, or likely influenced by both. The nice thing of having an auto-tuned parameterized code framework is that tuning parameters can adapt themselves to the algorithmic and hardware properties.

The methodology proposed in the thesis is more like an optimization recipe of how to optimize code for a broader set of problems with similar computational pattern. In the future, we hope to automate the code transformations used in this work, so that code can be automatically generated for other stencil kernels or higher dimensional data grid. Building a domain specific compiler that automatically search and combine code transformations in a structural way will allow better productivity in the code generation as well as the validation process.

List of Notations

Bibliography

[1] Intel Atom Processor Specifications. http://www.intel.com/products/processor/atom/index.htm.

[2] Performance Insights to Intel Hyper-Threading Technology. http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/.

[3] POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/.

[4] Signed Distance Function. http://en.wikipedia.org/wiki/Signed_distance_function.

[5] The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

[6] The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[7] Thread Affinity Interface (Linux and Windows). http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/compiler_f/optaps/common/optaps_openmp_thread_affinity.htm.

[8] Intel Advanced Vector Extensions programming reference. 2008. http://software.intel.com/en-us/avx/.

[9] Performance of ICC, GCC and OpenMP. *Equalizer White Paper*, 2008. http://www.equalizergraphics.com/documents/WhitePapers/OpenMP_ICC.pdf.

[10] Intel VTune Amplifier XE 2011: Optimize Performance and Multicore Scalability on Windows and Linux. 2011. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[11] A. Fox A. Ganapathi, K. Datta and D. Patterson. A case for machine learning to optimize multicore performance. *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[12] R. Bagnara, P. M. Hill, and E. Zaffanella. PPL: The Parma Polyhedra Library. http://www.cs.unipr.it/ppl/.

[13] Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc: Portable, Extensible Toolkit for Scientific Computation. 2011. http://www.mcs.anl.gov/petsc.

[14] 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.

[15] 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.

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

[17] C. Gui C. Li, C. Xu and M. D. Fox. Distance regularized level set evolution and its application to image segmentation. *IEEE Transaction on Image Processing*, 19(12), 2010.

[18] J. C. Gore C. Li, C. Kao and Z. Ding. Implicit active contours driven by local binary fitting energy. *Proceeding of Computer Vision and Pattern Recognition (CVPR)*, 2007.

[19] J. C. Gore C. Li, C. Kao and Z. Ding. Minimization of region-scalable fitting energy for image segmentation. *IEEE Transaction on Image Processing*, 17(10):1940–1949, 2008.

[20] T. Chan and L. Vese. Active contours without edges. *IEEE Transaction on Image Processing*, 10:266–277, 2001.

[21] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, PhD thesis and University of California and Berkeley, 2009.

[22] 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*, 51(1):129–159, 2009.

[23] 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. *Proceeding of ACM/IEEE Conference on Supercomputing (SC)*, 2008.

[24] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of IEEE*, 93(2):293–312, 2005.

[25] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich R´ude, Ulrich R Ude, and Christian Wei? Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal*, 10:21–40, 1999.

[26] W. Pugh D. Wonnacott E. Rosser, W. Kelly and T. Shpeisman. The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs. http://www.cs.umd.edu/projects/omega/.

[27] Katherine Yelick Eun-Jin Im and Richard Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *International Journal on High Performance Computing Applications (IJHPCA)*, 18(1):135–158, 2004.

[28] M. Frigo and S. G. Johnson. A fast Fourier transform compiler. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.

[29] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[30] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[31] S. Lankton and A. Tannenbaum. Localizing region based active contours. *IEEE Transaction on Image Processing*, 17(11), 2008.

[32] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[33] V. Loechner. PolyLib: A library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.

[34] P. Smereka M. Sussman and S. Osher. A level set approach for computing solutions to incompressible two-phase flow. *Journal of Computational Physics*, 114(1):146–159, 1994.

[35] G. Back M.Belgin and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proceedings of the 23rd international conference on Supercomputing (ICS)*, 2009.

[36] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Puschel. Formal datapath representation and manipulation for implementing DSP transforms. *Proceedings of Design Automation Conference (DAC)*, pages 385–390, 2008.

[37] D. Mirković and S. L. Johnsson. Automatic performance tuning in the UHFFT library. *Proc. Intl Conf. Computational Science (ICCS)*, 2073:71–80, 2001.

[38] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. New York: Springer-Verlag, 2002.

[39] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton–Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[40] M. Puschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for dsp transforms. *Proceeding of IEEE*, 93(2):232–275, 2005.

[41] J. A. Sethian R. Malladi and B. C. Vemur. Shape modeling with front propagation: a level set approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:158–175, 1995.

[42] J. Demmel R. Vuduc and K. Yelick. OSKI: Optimized Sparse Kernel Interface. http://bebop.cs.berkeley.edu/oski/.

[43] Youcef Saad. SPARSEKIT: Sparse Matrix Utility Package. http://people.sc.fsu.edu/~jburkardt/f_src/sparsekit/sparsekit.html.

[44] 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.

[45] 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.

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

[47] D. Takahashi. An implementation of parallel 1-D FFT using SSE3 instructions on dual-core processors. *Proc. Intl Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, pages 1178–1187, 2006.

[48] R. Kimmel V. Caselles and G. Sapiro. Geodesic active contours. *International Journal of Computer Vision*, 22(1):61–79, 1997.

[49] T. Coll V. Caselles, F. Catte and F. Dibos. A geometric model for active contours in image processing. *Numerische Mathematik*, 66(1):1–31, 1993.

[50] L. Vese and T. Chan. A multiphase level set framework for image segmentation using the Mumford and Shah model. *International Journal on Computer Vision*, 50:271–293, 2002.

[51] Y. Voronenko. *Library Generation for Linear Transforms*. PhD thesis, PhD thesis of Electrical and Computer Engineering, Carnegie Mellon University, 2008.

[52] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, PhD thesis and University of California and Berkeley, 2004.

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

[54] Y.-J. Chang W. Yu, F. Franchetti and T. Chen. Fast and robust active contours for image segmentation. *IEEE International Conference on Image Processing (ICIP)*, 2010.

[55] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[56] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, PhD thesis and University of California and Berkeley, 2008.

[57] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Matrix-vector multiplication on emerging multicore platforms. *Proceeding of ACM/IEEE Conference on Supercomputing (SC)*, 2007.

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

[59] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 63–76, 2003.