Making computer vision computationally efficient

Narayanan Sundaram

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-106
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-106.html

May 11, 2012

Copyright © 2012, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Making Computer Vision Computationally Efficient

by

Narayanan Sundaram

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt Keutzer, Chair
Professor Jitendra Malik
Professor Alper Atamturk
Professor James Demmel

Spring 2012

Making Computer Vision Computationally Efficient

Copyright 2012
by
Narayanan Sundaram

Abstract

Making Computer Vision Computationally Efficient

by

Narayanan Sundaram

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt Keutzer, Chair

Computational requirements for computer vision algorithms have been increasing dramatically at a rate of several orders of magnitude per decade. In fact, the growth in the size of datasets and computational demands for computer vision algorithms are outpacing Moore's law scaling. Meanwhile, parallelism has become the major driver of improvements in hardware performance. Even in such a scenario, there has been a lack of interest in parallelizing computer vision applications from vision domain experts whose main concern has been productivity. We believe that ignoring parallelism is no longer an option. Numerical optimization and parallelization are essential for current and future generation of vision applications to run on efficiently on parallel hardware.

In this thesis, we describe the computational characteristics of computer vision workloads using patterns. We show examples of how careful numerical optimization has led to increased speedups on many computer vision and machine learning algorithms including support vector machines, optical flow, point tracking, image contour detection and video segmentation. Together, these application kernels appear in about 50% of computer vision applications, making them excellent targets for focusing our attention. We focus our efforts on GPUs, as they are the most parallel commodity hardware available today. GPUs also have high memory bandwidth which is useful as most computer vision algorithms are bandwidth bound. In conjunction with the advantages of GPUs for parallel processing, our optimizations (both numeric and low-level) have made the above mentioned algorithms practical to run on large data sets. We will also describe how these optimizations have enabled new, more accurate algorithms that previously would have been considered impractical. In addition to implementing computer vision algorithms on parallel platforms (GPUs and multicore CPUs), we propose tools to optimize the movement of data between the CPU and GPU.

We have achieved speedups of $4 - 76\times$ for support vector machine training, $4 - 372\times$ for SVM classification, $37\times$ for large displacement optical flow and $130\times$ for image contour detection compared to serial implementations. A significant portion of the speedups in each case was obtained from algorithmic changes and numerical optimization. Better algorithms have improved performance not only on manycore platforms like GPUs, but also on multicore CPUs. In addition to achieving speedups on existing applications, our tool for helping

manage data movement between CPU and GPU has reduced runtime by a factor of $1.7-7.8\times$ and the amount of data transferred by over $100\times$. Taken together, these tools and techniques serve as important guidelines for analyzing and parallelizing computer vision algorithms.

Dedicated to Amma, Appa and Kavitha

Contents

List of Figures v

List of Tables viii

1 Introduction 1
1.1 Contributions 6
1.1.1 Numerical optimizations 7
1.1.2 Memory management 8
1.2 Outline of thesis 8

2 Background and Motivation 10
2.1 Trends in Computer vision 11
2.1.1 Size of unit data 11
2.1.2 Dataset sizes 13
2.1.3 Algorithmic complexity 13
2.2 Hardware parallelism 15
2.3 Summary 16

3 Understanding Computer Vision Workloads 18
3.1 Patterns and OPL 19
3.2 Workload collection & Analysis 19
3.2.1 Parallelism 21
3.2.2 Linear Algebra 22
3.2.3 Graph algorithms 23
3.3 Summary 24

4 Parallelizing Computer Vision 25
4.1 Numerical optimizations for image contour detection 26
4.1.1 Background 26
4.1.2 Problem description 27
4.1.3 Related work 28
4.1.4 Our approach 29
4.1.5 Results 29
4.2 Numerical optimizations in Optical flow 29

4.2.1 Background 30
4.2.2 Problem description 31
4.2.3 Related work 32
4.2.4 Our approach 32
4.2.5 Results 33
4.3 Implementation issues 33
4.3.1 Background 33
4.3.2 Problem Description 35
4.3.3 Related work 36
4.3.4 Our approach 40
4.3.5 Results 41
4.4 Summary 41

5 Parallelizing Support Vector Machine 42
5.1 SVM background 42
5.1.1 SVM Training 42
5.1.2 SVM Classification 44
5.2 Problem 44
5.3 Related work 45
5.3.1 Training 45
5.3.2 Classification 46
5.4 Our approach 46
5.4.1 Training 46
5.4.2 Classification 51
5.5 Results 52
5.5.1 Training 52
5.5.2 Classification 54
5.6 Summary 55

6 Optical Flow and Point Tracking 57
6.1 Background 58
6.2 Problem 60
6.3 Related work 61
6.4 Our approach 62
6.4.1 Positive semi-definiteness of the fixed point matrix. 62
6.4.2 Linear solvers. 66
6.4.3 Point tracking with large displacement optical flow 67
6.5 Results 68
6.5.1 Large displacement optical flow 69
6.5.2 Tracking 74
6.6 Summary 79

7 Image and Video Segmentation **80**
7.1 Background and Motivation . . . 80
7.2 The *gPb* Detector . . . 81
7.3 Problem . . . 83
7.4 Related Work . . . 84
7.4.1 Image segmentation . . . 84
7.4.2 Video segmentation . . . 85
7.5 Our approach . . . 86
7.5.1 Calculating Local cues . . . 86
7.5.2 Memory management in eigensolver . . . 88
7.5.3 Extension to video segmentation . . . 91
7.5.4 Postprocessing . . . 94
7.6 Results . . . 96
7.6.1 Image segmentation . . . 96
7.6.2 Video Segmentation . . . 99
7.7 Summary . . . 103

8 Memory Management in CPU-GPU systems **105**
8.1 Background and Motivation . . . 105
8.2 Problem . . . 106
8.2.1 Challenge 1: Scaling to data sizes larger than the GPU memory . . . 107
8.2.2 Challenge 2: Minimizing data transfers for efficient GPU execution . 109
8.3 Related Work . . . 112
8.4 Our approach . . . 113
8.4.1 Problem specific approaches . . . 113
8.4.2 Operator splitting: Achieving scalability with data size . . . 114
8.4.3 Operator and data transfer scheduling . . . 115
8.5 Results . . . 119
8.5.1 Description of Applications . . . 119
8.5.2 Performance improvement and data transfer reduction . . . 121
8.5.3 Scalability . . . 122
8.6 Summary . . . 123

9 Conclusions **125**
9.1 Contributions . . . 125
9.1.1 Pattern analysis of computer vision workloads . . . 125
9.1.2 Improvements to numerical algorithms . . . 126
9.1.3 Memory management in CPU-GPU systems . . . 127
9.2 Future Work and Extensions . . . 128
9.2.1 Integration with frameworks . . . 128
9.2.2 More coverage . . . 128
9.3 Summary . . . 128

Bibliography **130**

List of Figures

1.1 Clock frequencies of processors and feature sizes of transistors from different manufacturers from 1970 to 2010. Data obtained from Danowitz et al [62]. . . 2
1.2 The implementation gap 4
1.3 Serial vs parallel versions of prefix scan. 5

2.1 Improvements in digital camera resolution in consumer cameras from 1990 to 2009. Each of the cameras shown was (for its time) the consumer grade camera with the most "Megapixels". 12
2.2 Increase in size of popular face recognition data sets from 1997 to 2006. . . . 13
2.3 Increase in algorithmic complexity of image contour detection algorithms. Numbers shown are estimates based on the amount of computation performed. 14
2.4 Increase in the algorithmic complexity of optical flow algorithms. Numbers shown are estimates based on the amount of computation performed. 15
2.5 Hardware scaling - (a) Improvement in peak theoretical floating point operations per second for CPUs and GPUs from 2003 to 2010. (b) Improvement in memory bandwidth for CPUs and GPUs from 2003 to 2010. 17

3.1 Organization of OPL 20

4.1 Illustration of normalized cut. For the graph cut denoted by the black dotted line, the normalized cut is given by $\frac{\text{Sum of black edges}}{\text{Sum of red and black edges}} + \frac{\text{Sum of black edges}}{\text{Sum of blue and black edges}}$ 27
4.2 Example image from the Berkeley Segmentation Dataset and its first 2 eigenvectors for the affinity matrix derived as in [119]. 28
4.3 Two consecutive frames (a) and (b) from the Middlebury dataset [14] and the calculated optical flow using [159] (c). 30
4.4 Logical organization of Nvidia GPUs 34
4.5 Logical organization of multicore CPUs 35

5.1 Examples of separating hyperplanes in 2 dimensions. 43
5.2 Effect of working set size on SVM training. 47

6.1 High level structure of the large displacement optical flow solver. 60

6.2 Sparse matrix showing the decomposition into dense 2 × 2 blocks. Filled dots represent non-zeroes and hollow dots & unspecified elements represent zeroes. 63
6.3 Diagonal blocks for pixel i. Ψ'_1, Ψ'_2 and Ψ'_S are defined as follows: $\Psi'_1 := \Psi'((I^k_x du^k + I^k_y dv^k + I^k_z)^2)$, $\Psi'_2 := \Psi'((I^k_{xx} du^k + I^k_{xy} dv^k + I^k_{xz})^2 + (I^k_{xy} du^k + I^k_{yy} dv^k + I^k_{yz})^2))$, $\Psi'_3 = \Psi'((u^k + du^k - u_1)^2 + (v^k + dv^k - v_1)^2)$ $\Psi'_S := \Psi'(|\nabla(u^k + du^k)|^2 + |\nabla(v^k + dv^k)|^2)$. $N(i)$ denotes the neighborhood of pixel i. 64
6.4 **Left: (a)** Initial points in the first frame using a fixed subsampling grid. **Middle: (b)** Frame number 15 **Right: (c)** Frame number 30 of the camer-amotion sequence. Figure best viewed in color. 68
6.5 Rates of convergence for different techniques considered. Y-axis shows the value of the residual normalized to the initial residual value averaged over 8 different matrices. Figure best viewed in color 70
6.6 Concurrency in downsampling 72
6.7 Comparison of different downsampling strategies. The crossover point is at a scale of 4.5% (0.2% of pixels). 73
6.8 Breakdown of execution times for serial and parallel variational optical flow solvers. Both solvers are run at a scale factor of 0.95, with 5 fixed point iterations and 25 Gauss-Seidel iterations/10 CG iterations to achieve similar AAE. Figure best viewed in color. 74
6.9 **(Top)** Frame 490 of the tennis sequence with **(left)** actual image, **(middle)** KLT points and **(right)** LDOF points. **(Bottom)** Frame 495 of the sequence with **(left)** actual image, **(middle)** KLT points and **(right)** LDOF points. Only points on the player are marked. KLT tracker points are marked larger for easy visual detection. Figure best viewed in color. 78

7.1 The gPb detector 81
7.2 Bresenham rotation: Rotated image with $\theta = 18°$ clockwise, showing "empty pixels" in the rotated image. 88
7.3 The Lanczos algorithm. 89
7.4 The Lanczos algorithm modified to use less memory. 90
7.5 Calculating inter frame pixel affinities. Frames 45 and 46 of the Tennis sequence [35] are shown. In order to calculate the affinity between pixels P and Q, we calculate the forward projection of P (Q') and the backward projection of Q (P'). Affinity between P & Q is calculated using equation (7.8). 92
7.6 Illustration of "leakage" in eigenvectors for long range video sequences. This sequence is one of the sequences in the Motion Segmentation data set [35] and is a scene from the movie "Miss Marple : Murder at the vicarage". **Top** Frames 1, 65, 120, 200 from the sequence. **Bottom** Corresponding portions of the second eigenvector. The vectors are normalized independently for better visualization. However, note the scale difference in the eigenvectors for each of the frames [0.899, 1.00], [0.430, 0.941] , [0.156, 0.293], [0.004, 0.066]. 95

7.7 Relative errors for computing χ^2-distances over rotated rectangular patches produced by Bresenham rotation vs nearest neighbor rotation (a) Relative error as a function of the angle of rotation (b) Relative error as a function of the area of the patch. 97
7.8 Performance scaling with parallelism (0.15 MP images) 99
7.9 Comparison of our technique vs motion segmentation by [35] and [86] on one of the sequences from [35]. **From top to bottom** Video frames(1, 50, 140, 200), Segmentation results from [35], Hierarchical graph segmentation [86], Our final segmentation result. Each tracked point in [35] is shown as 9 pixels large for illustration purposes. Notice that that boundary shifts identity in [35]. Hierarchical graph segmentation produces oversegmentation and non-smooth boundaries. Also note the confusion between the person and the background in the later frames. Figure best viewed in color. 100
7.10 Flowergarden sequence (30 frames). **Far Left** Frames 1 and 30 of the sequence. **Left Center** Segmentation results from Hierarchical Graph segmentation [86]. Note the noise in the tree edges and the flower bed. **Right Center** Results from [30]. Notice the identity shift of the house and background. **Far Right** Our results. The segmentation edges are smooth and tracking is consistent. 101
7.11 Rocking horse sequence (10 frames) [156] **Far Left** Center frame of the sequence with ground truth superimposed. **Left Center** Results from [96]. This requires knowledge of number of segments. **Right Center** Our segmentation results. Note that the ground truth has open edges whereas our method only produces closed edges. **Far Right** Our results with increased weights to motion features. Notice the absence of the edge along the mane and the presence of the occlusion edge along the wall. Figure best viewed in color. 102
7.12 Breakdown of the video segmentation runtime. 103
7.13 Breakdown of the runtime of the Sparse Matrix Vector Multiply routine. . . 104

8.1 Schematic of a typical CPU-GPU system. Both CPU and GPU DRAM are shaded. 106
8.2 Edge detection in histological micrograph images (a) input/output images, (b) Algorithm used for edge detection, and (c) Memory requirements for the edge detection algorithm as a function of the input image size 108
8.3 Execution time breakdown for executing image convolution operations with varying kernel matrix sizes on a GPU 110
8.4 Two alternative schedules for edge detection that illustrate the impact of operator scheduling on data transfers 111
8.5 Pseudo-Boolean Formulation of Offload and Data Transfer Scheduling 116
8.6 Convolutional neural network - layer transformation 121
8.7 Performance of the edge detection template for scaling input data size 123

List of Tables

3.1 Papers referring to parallelism. 21
3.2 Papers referring to linear algebra (out of 50 randomly chosen papers in ICCV 2011). 22
3.3 Papers referring to graphs/trees (out of 50 randomly chosen papers in ICCV 2011). 24

4.1 Different parallelization techniques and their relative performance on various parameters. 40

5.1 Common Kernel Functions for Support Vector Machines 44
5.2 SVM Datasets - References and training parameters 53
5.3 Dataset Sizes 53
5.4 SVM Training Convergence Comparison 53
5.5 SVM Training Results 54
5.6 Accuracy of GPU SVM classification vs. LIBSVM 55
5.7 Performance of GPU SVM classifier compared to LIBSVM and Optimized CPU classifier 55

6.1 Average time taken by the linear solvers for achieving residual norm $< 10^{-2}$ of initial norm 71
6.2 Average Angular Error (in degrees) for images in the Middlebury dataset. . . 73
6.3 Tracking accuracy of LDOF and KLT over 10 frames of the MIT sequences . 75
6.4 Tracking accuracy of LDOF and KLT over all the frames of the MIT sequences 75
6.5 Tracking accuracy of LDOF and Sand-Teller tracker over the sequences used in [146] 76
6.6 Occlusion handling by KLT and LDOF trackers based on region annotation from the MIT data set. Occluded tracks indicate tracks that are occluded according to the ground truth data, but not identified as such by the trackers. 77
6.7 Tracking accuracy of LDOF and KLT for large displacements in the tennis sequence with manually marked correspondences. Numbers in parentheses indicate the number of annotated points that were tracked. 78

7.1 Local Cues Runtimes on GTX 280 98
7.2 GPU Processor Specifications 98

7.3 Comparison of our video segmentation with other approaches. 100

8.1 Reduction in data transfers between the host and GPU memory 122
8.2 Improvements in execution time from optimized data transfer scheduling . . . 122

Acknowledgments

This dissertation would not have been possible without the help and support of a large number of people.

First, I would like to thank my advisor Kurt Keutzer for his immense support throughout the duration of my PhD. His encouragement to forge my own path and his focus on the big picture have been a source of inspiration and have been crucial for this work. Working with him has made me much more aware of the technological and business aspects of the electronics industry and I am very thankful for it. I must thank the other members of my dissertation committee - Prof. James Demmel, Prof. Jitendra Malik and Prof. Alper Atamturk for their help and feedback on the dissertation.

I must thank all the members of the PALLAS group at Berkeley for making research collaborative and fun. Thanks to Bryan Catanzaro, Jike Chong, Nadathur Satish, Bor-Yiing Su, Michael Anderson, David Sheffield, Mark Murphy, Katya Gonina and Chao-Yue Lai for making my time in the group really enjoyable and productive (Special thanks to Bryan, Michael and David for all those interesting late afternoon discussions). I must thank my colleagues and fellow researchers for motivating discussions and technical insights. In particular, I must thank Prof. Thomas Brox, Prof. Anand Raghunathan, Prof. Per-Olof Persson for their help in guiding my research. I would like to thank John Shalf at LBNL for providing access to a GPU cluster which enabled a significant portion of my research. Thanks to everyone in Parlab and DOP center for making research more enjoyable.

My sincere thanks go to Parlab staff Roxana Infante and Tammy Johnson, EECS staff Ruth Gjerde, Dana Jantz, Shirley Salanio and Carol Zalon for their patient help with all the paperwork I had to deal with through the last 6 years.

This section would be incomplete without a mention of my friends. I would also like to thank all the roommates I have had during my stay at Berkeley. Thanks Ankush Kheterpal, Jia Zou, Vinayak Nagpal, Luca Garre, Susmit Jha, Siddharth Dey, Rahul Rai, and Amik Singh for putting up with me and making the time I spent in Berkeley truly memorable. Thanks Subhransu Maji, Mohit Chawla, Avinash Varadarajan, Shashank Nawathe, Pallavi Joshi, Deepan Raj Prabhakar, Sudeep Juvekar, Chandrayee Basu, Pamela Bhattacharya, Varsha Desai and all my friends at Berkeley for all the time-pass. My PhD would surely not have been this interesting without all the trips and fun we had through the years. I would like to thank my "wingmates" from IIT for great times past, present and future.

Finally, I would like to thank my parents and my sister without whose support and encouragement my PhD would not have been possible at all. I cannot thank them enough for everything they have done for me so far. I would also like to thank my 2-month old nephew Nishanth for bringing more joy and happiness in my life.

To everyone who has been a part of my life, thank you for everything.

Chapter 1

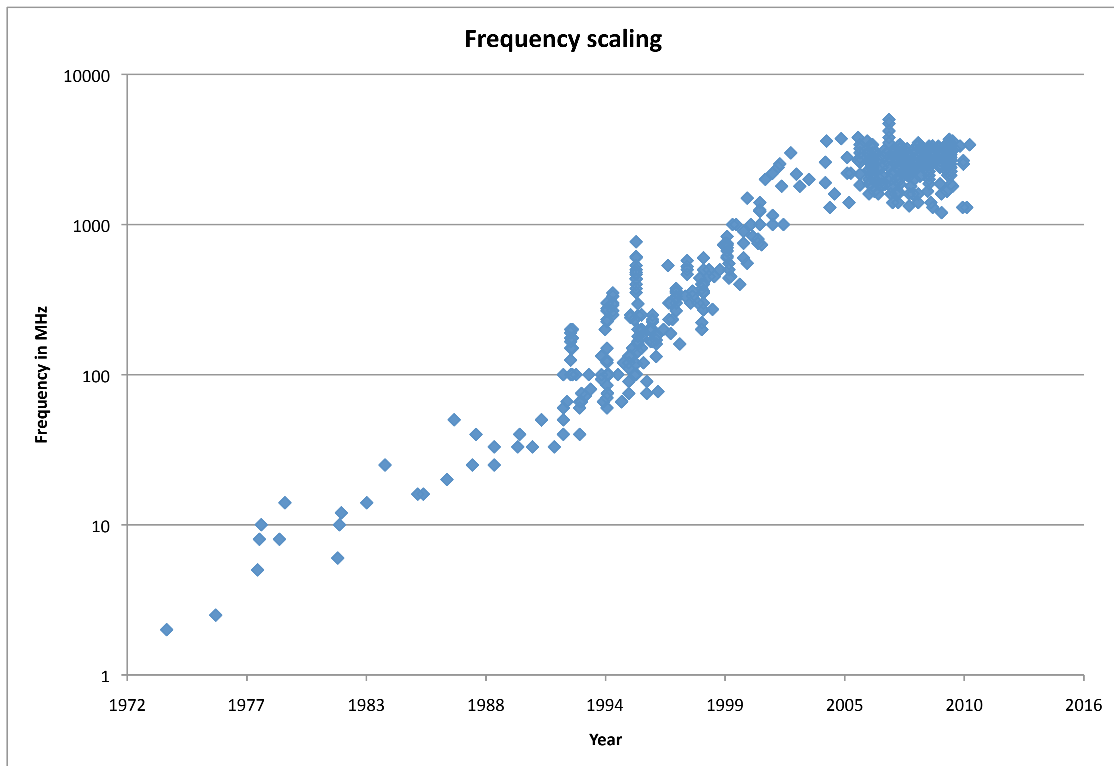
Introduction

Improvements to computing power of electronic devices have sustained our current technological progress and contributed to a marked improvement in the quality of our lives. The massive increase in the computational power of processors has come through our sustained ability to reduce the size of transistors and to pack more of them in a single die. The empirical observation that the number of transistors in a single chip doubles every 18 months is usually referred to as "Moore's law".

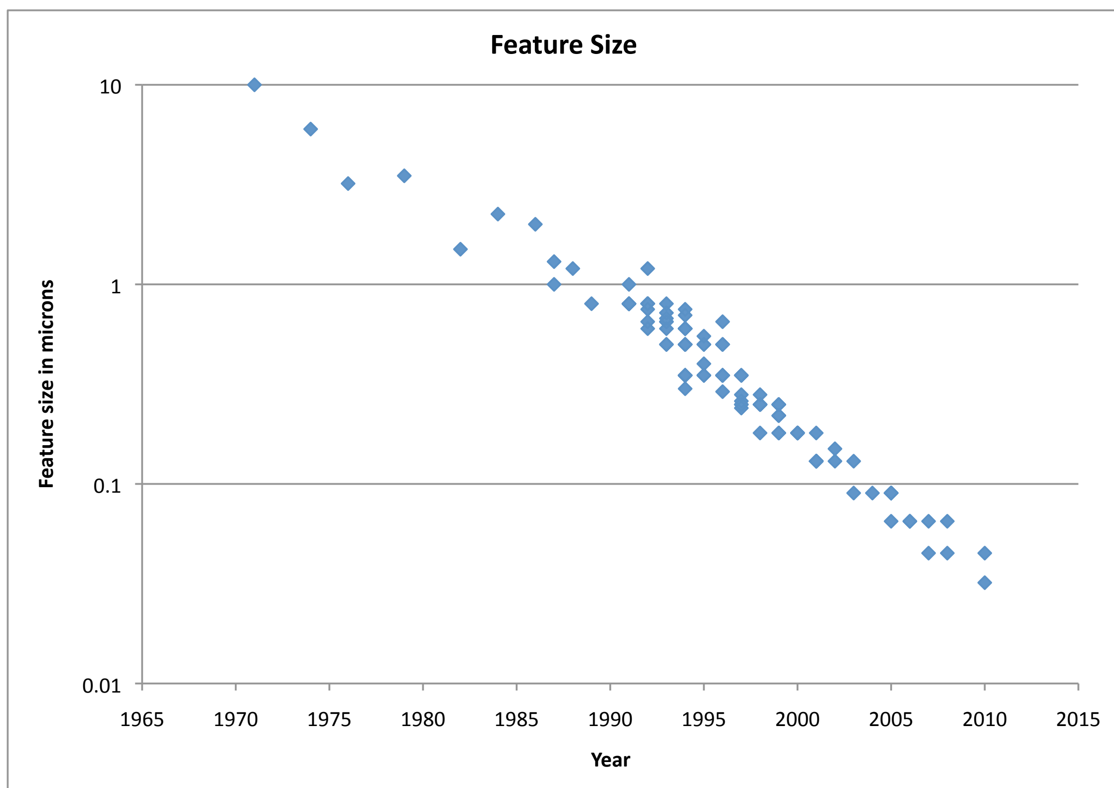
Computer industry has relied on Moore's law to improve performance while keeping the chip area constant by reducing the size of the transistors. Smaller transistors have smaller capacitances, which can charge/discharge faster. The total dynamic power dissipation of the chip varies as CV^2f where C is the total capacitance, V is the voltage and f is the frequency of switching. f can be increased without increasing the power density (power dissipated per unit area) even with increasing number of transistors by reducing the voltage V. All this changed when the voltage could no longer scale down with decreasing transistor sizes because of the increase in leakage power. This meant that improving performance within a constant power density using only voltage and frequency scaling became impossible. This is referred to as the "power wall" [10]. The power wall has forced us into the world of single socket parallel computing. With clock frequency no longer increasing, the other practical way to improve performance using increasing number of transistors was through parallelism. However, this has meant that software developers must now write code to explicitly take advantage of this parallelism. As the Berkeley View paper [10] said,

"This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures".

We entered the multi-core era in the mid-2000's when single threaded processor performance tapered off and dual core processors become the norm. Figure 1.1(a) shows that the clock frequency scaling stopped around 2005. This data is obtained from Danowitz et al [62] and includes processors from Intel, AMD, IBM, DEC, HP, Motorola, SGI, Sun etc. The size of transistors, however has kept shrinking (Figure 1.1(b)). The number of cores kept increasing every generation keeping up with Moore's law. Graphics Processing Units



(a)



(b)

Figure 1.1: Clock frequencies of processors and feature sizes of transistors from different manufacturers from 1970 to 2010. Data obtained from Danowitz et al [62].

(GPUs) became more general purpose programmable. We also saw a massive increase in the amount of data and compute intensive applications from domains like computer vision and machine learning. However, software developers are still reluctant to rewrite code to take advantage of hardware parallelism and hence miss out on any performance improvements from hardware scaling.

Computer vision has grown into a large and important area in research and industry. Ever since multimedia (audio, images and video) started becoming digital, the amount of such data generated has increased exponentially. Several factors have enabled this development e.g. cheap storage, proliferation of media capture devices like digital cameras, smart phones, CCTVs etc., ease of sharing the data driven by improved wired & wireless bandwidth, and improvement in processing power in computing devices. It has been estimated that the global machine vision and vision guided robotics market will be worth $15.3 billion by 2015 [136]. Computer vision algorithms have been increasing in complexity and require tremendous amounts of computing resources (around 100,000 floating point operations per pixel for image segmentation or optical flow). This will be explained in more detail in Chapter 2.

Exploiting the concurrency in applications has become critical for getting better performance with time because computational requirements for computer vision are increasing and processors are becoming increasingly parallel. Computer vision domain experts are focused mostly on productivity i.e. they would like to get their algorithms running with the shortest amount of programmer effort while simultaneously taking advantage of the improvements in hardware to scale to future processors. This vision worked well during the era of serial processors and voltage-frequency scaling. That goal, however, has been put in jeopardy due to the parallelism in today's processors. Without explicitly writing code to take advantage of parallelism, it is neither possible to get good performance today nor is it possible to scale to tomorrow's even more parallel processors. Auto parallelizing compilers have not been as successful in finding and exploiting concurrency as had been hoped earlier, and hence domain experts have been forced to manually parallelize their applications. This has lead to an "implementation gap", a gap between application developer's and an expert parallel programmer's view of computing resources. This is illustrated in Figure 1.2.

As mentioned earlier, application developers focus on application level trade-offs without the knowledge of how this could affect parallelization opportunities in hardware. Expert parallel programmers, on the other hand, have an good understanding of the trade-offs at the hardware level, but have limited knowledge of the application domain. Work that bridges this gap either through the use of tools that can automate parallelization of applications from high-level specifications or through the demonstration of the trade-offs needed to achieve good performance and scalability on parallel hardware are useful to advance the state-of-the-art in the application domain by focusing the attention of application developers on the key bottlenecks. Given that GPUs are the most parallel commodity processors one can buy today, porting applications to GPUs has been of much interest lately. Interestingly, GPUs are becoming more and more general purpose, while CPUs are becoming more and more parallel, thus leading to convergence in how parallel programmers view these platforms. However with either platform, application parallelization needs to be explicit in order to

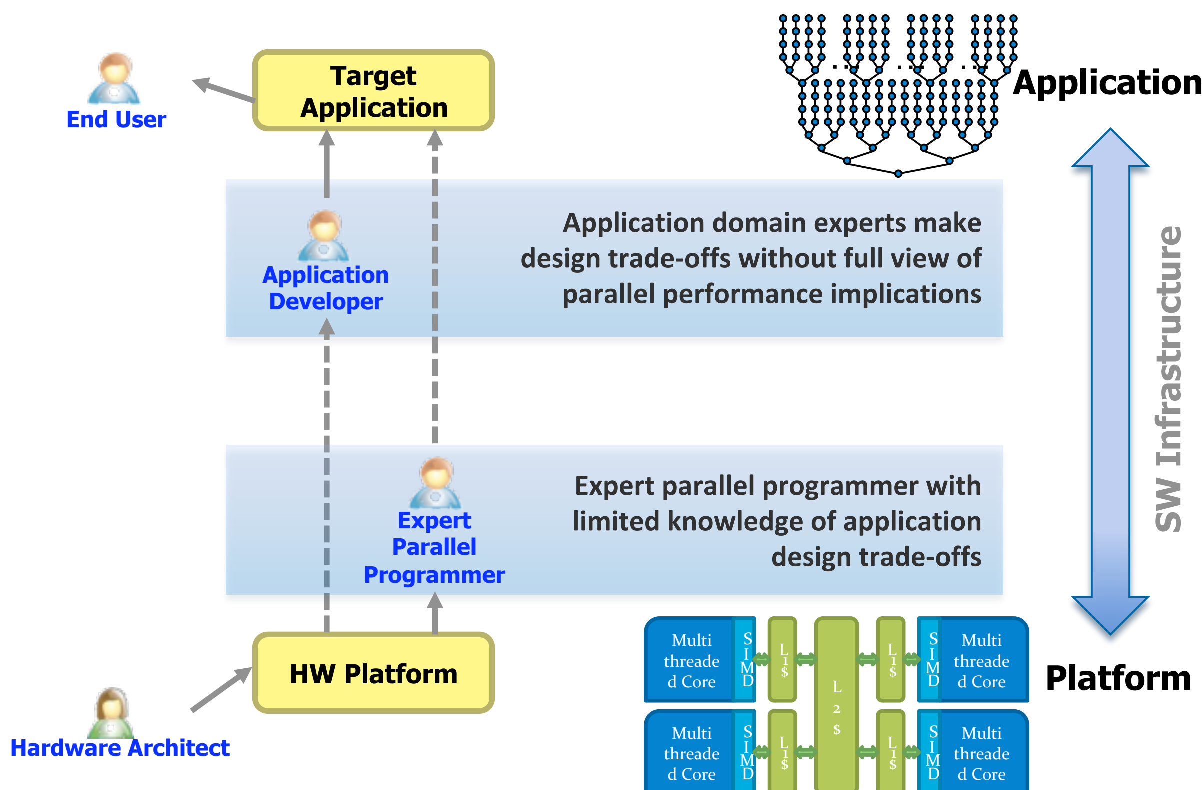


Figure 1.2: The implementation gap

achieve peak performance. While parallelizing existing serial algorithms is itself an enormous task, it is more complicated by the need to explore different algorithms in order to find the one that is most efficient on today's hardware and is scalable in the future. In addition to the need to parallelize applications, it is also important to note that there is an extra challenge. GPUs have memory bandwidth that is many times more than DRAM-CPU bandwidth and is helpful for bandwidth bound applications. However, GPUs have limited memory capacity which severely restricts productivity when the amount of data being handled is larger than the GPU memory capacity.

Parallelizing computer vision algorithms has been considered relatively easy due to the abundance of independent computations in many algorithms. In particular, computations that operate on every pixel independent of the others provide plenty of concurrency. This purported ease of exploiting concurrency, along with the absence of parallel hardware at their disposal earlier led to computer vision researchers not making special efforts to parallelize their applications. Before the advent of single socket parallel machines, most of the parallel hardware was distributed. The downside of distributed hardware is the high cost of communication between the nodes. Given that not all computer vision algorithms are pixel-parallel and many of them solve global optimization problems using iterative techniques, high communication costs lead to poor performance and consequently a lack of interest in parallelization. However, the arrival of multicore and manycore machines has occurred at

a time when there has also been a huge increase in the complexity of computer vision algorithms. The presence of single socket parallelism works in our favor here; communication costs that used to be prohibitive become quite manageable when it is within a single die, and iterative computations can be parallelized while still being scalable. Even though computer vision deals with a large variety of problems conceptually, there are only a finite number of computations that are recurrent in these problems. These computational patterns are neatly categorized in a pattern language OPL [104].

It is not sufficient to just naively parallelize existing computer vision algorithms without any changes in order to achieve performance and scalability. Some of the optimizations performed for runtime improvements in computer vision lead to poor parallelizability. For example, integral image [115] is a useful technique for reusing computations and improving runtimes. Consider integral images in 1-dimension i.e. the computation involved in scanning an array $I(x) = \sum_{i=0}^{x} f(i)$. This computation, when performed serially would lead to an implementation that cannot be parallelized directly (Figure 1.3). Every iteration of the loop is dependent on the previous iteration. However, this computation can be parallelized efficiently if we use a prefix scan [22] instead of serial summation. In fact, even though the parallel version performs 3 times as many additions as the serial version, it runs faster on parallel machines and is scalable. Both versions of the algorithm are shown in Figure 1.3.

Algorithm: Serial Scan
Input: f (Input Array)
Output: I (Output Array)
1 $I[0] \leftarrow f[0]$
2 **for** $i \leftarrow 1, 2, \ldots N-1$
3 $\quad I[i] \leftarrow I[i-1] + f[i]$
4 **end for**

Algorithm: Parallel Scan
Input: f (Input Array)
Output: I (Output Array)
1 $a \leftarrow f$
2 **for** $d \leftarrow 0, \ldots log_2(N) - 1$
3 $\quad$ **parallel for** $i \leftarrow 0, 1, \ldots N-1$ **step** 2^{d+1}
4 $\quad\quad a[i + 2^{d+1} - 1] \leftarrow a[i + 2^d - 1] + a[i + 2^{d+1} - 1]$
5 $\quad$ **end for**
6 **end for**
7 $a[N-1] \leftarrow 0$
8 **for** $d \leftarrow log_2(N) - 1, \ldots 0$
9 $\quad$ **parallel for** $i \leftarrow 0, 1, \ldots N-1$ **step** 2^{d+1}
10 $\quad\quad t \leftarrow a[i + 2^d - 1]$
11 $\quad\quad a[i + 2^d - 1] \leftarrow a[i + 2^{d+1} - 1]$
12 $\quad\quad a[i + 2^{d+1} - 1] \leftarrow t + a[i + 2^{d+1} - 1]$
13 $\quad$ **end for**
14 **end for**
15 **parallel for** $i \leftarrow 0, 1, \ldots N-1$
16 $\quad I[i] \leftarrow f[i] + a[i]$
17 **end for**

Figure 1.3: Serial vs parallel versions of prefix scan.

Such inefficiencies are not just restricted to prefix scans. For many computer vision applications, the majority of runtime is spent on numerical algorithms. For example, eigensolvers are used for image contour detection algorithms [119]. The space of different algorithms that can be used in these cases is usually large and the right choice depends on the specifics of the problem at hand. Even if there are sufficient parallelization opportunities in any of these algorithms, choosing the right one can give us an order of magnitude improvement compared to naive parallelization [47]. Sometimes, a reformulation of the problem can lead to the use of algorithms that can run much closer to the machine's peak throughput [48]. Picking the right algorithm is at least as important as performing all the low level optimizations in order to get good runtime performance. In such a scenario, parallelizing computer vision algorithms is not just restricted to porting them to a parallel architecture, but also developing innovative parallel algorithms or new applications of numerical algorithms for the specific problems that we deal with.

The main contribution of the thesis is exploring how to make computer vision algorithms run efficiently on modern parallel machines. In particular, we focusing on parallelizing computer vision algorithms to run efficiently on GPUs. We look at the challenges in terms of choosing the right algorithm, any reformulations to make the implementation more efficient, low level optimizations that are necessary to get good performance and automating memory management in CPU-GPU systems.

1.1 Contributions

This thesis makes the following main contributions -

- We propose numerical optimizations for improving the performance of algorithms in computer vision & machine learning. These optimizations are tailored for single socket parallel hardware, in particular GPUs. We perform algorithmic explorations and reformulations to improve performance of numerical algorithms for many applications including support vector machine training & classification, optical flow & tracking and image & video segmentation.

- We propose memory optimizations that need to be done for running large applications on CPU-GPU systems. We formulate the problem of managing data and controlling the transfers between CPU and GPU formally as a pseudo boolean optimization problem and propose heuristics that are shown to empirically perform well.

- We analyze computer vision workloads for computational patterns in order to quantitatively analyze the importance of various computational patterns. We pick a subset of research papers from recent literature in order to get a representative sample of the kinds of algorithms currently in use in the computer vision community. This is explained in Chapter 3.

In addition, we also produce efficient parallel implementations of multiple computer vision and machine learning applications which are being used by the research community in multiple application areas. The source code of these implementations is publicly available.

In the following section, we explain in more detail about the particular applications that we looked at and how algorithmic exploration helps in those cases. We also discuss how memory management in CPU-GPU systems can improve performance.

1.1.1 Numerical optimizations

Numerical algorithms are at the heart of computer vision, since most problems are formulated as continuous optimization problems that are then solved either directly or approximated using relaxations or heuristics. In such a scenario, attention to the numerical techniques employed becomes significantly important for improving performance.

We consider the following computer vision and machine learning algorithms in this dissertation - support vector machines, optical flow & tracking, image & video segmentation. Each of them have particular numerical properties that can be exploited for better performance. We explain the details in the respective chapters, but we give an overview here.

Support vector machine is a machine learning algorithm used for classification and regression. In the classic case, it is formulated as a two-class classification problem. Looking at the training portion of the algorithm, most of the time is spent on a matrix-vector-multiplication-like BLAS2 computation. GPUs have significantly higher memory bandwidth compared to CPUs, and hence mapping this computation to a GPU provides a significant speedup. In addition, we also map data structures and computations to take advantage of the particular features of the GPU architecture. SVM classification can also be written as a BLAS2 computation [48]. However, restructuring the computation gives us a way to implement it as a matrix-matrix-multiplication-like BLAS3 computation [48]. This reformulation leads to improved performance on both multi-core CPU and GPU platforms compared to the widely used LibSVM library [71]. Chapter 5 gives more details.

Optical flow is a video processing algorithm that analyzes consecutive frames of a video sequence and produces a displacement map for every pixel in the frame. This computation is important for performing motion analysis on video sequences and hence both accuracy and speed are essential. Most accurate optical flow techniques such as the one by Brox et al [36] require a significant amount of processing (about 100,000 floating point operations per pixel as mentioned in Chapter 2). Parallelizing this computation has its challenges. Algorithmic exploration in this case requires a careful evaluation of a number of linear solvers tailored to the problem at hand. In addition, we prove that the linear system being solved is positive-semidefinite even with non-convex penalty functions. Our use of the preconditioned conjugate gradient solver and efficient parallelization produces a speedup of over 37× compared to the original serial implementation of large displacement optical flow [159]. Of this 37× speedup, about 10× is attributable to the difference in memory bandwidth & hardware and the rest to algorithmic changes. Using this optical flow information to create point tracks leads to vastly improved accuracy and density compared to other existing point trackers. Details on the parallelization of optical flow and tracking are discussed in Chapter 6.

Segmentation is basic to image and video content analysis. Separating objects from surroundings and other objects is essential to understanding the scene and reconstructing

the model of the world where the picture/video was taken. We look at the parallelization and optimization of gPb [119], the state-of-the-art algorithm for image contour detection. In particular, we look at the complexities in mapping the eigensolver and local boundary detection computations on GPUs with limited memory capacity. Using the image contour detection as a driver, we propose a video segmentation algorithm that utilizes the same underlying mathematical framework. We show that this algorithm produces results that are comparable to or better than other video segmentation algorithms. Numerical optimization leads us to solve computations that would have been considered infeasible earlier and brings new algorithmic capabilities. Details on parallelizing segmentation problems are provided in Chapter 7.

1.1.2 Memory management

In all the applications that we parallelize, there always exists the problem of mapping a problem that does not fit in GPU memory to run on GPU hardware efficiently. Under such a scenario, the amount of data transfers between the CPU and GPU memory can become a bottleneck as this communication happens over PCI-Express which has limited bandwidth (8 GB/s for PCI-e v2.0). We propose to solve this problem for a subset of computations. The computations we consider are ones that can be expressed as operator graphs with statically analyzable dependencies. The ability to run applications on GPUs to take advantage of their high memory bandwidth is constrained by the limited memory capacity of modern GPUs. In particular, most desktop GPUs available today have memory capacity of a few hundred MB to 5GB. Frameworks that can help in simplifying and improving the productivity of domain experts who have to deal with artifacts like these are important. Chapter 8 shows how this problem is formulated and solved efficiently.

1.2 Outline of thesis

The thesis is organized as follows:

Chapter 2 describes the background of increasing computational demands from applications in computer vision and machine learning. It shows how the hardware is scaling from the increase in the number of transistors from Moore's law and how the gap between the required and available computational power is widening.

Chapter 3 describes computational patterns in computer vision applications. By collecting applications from research papers in computer vision, we have been able to analyze the amount and type of computations needed.

Chapter 4 describes prior work in parallelizing computer vision applications. We look at previous techniques for optimizing problems like image segmentation and optical flow, and see discuss how our methods differ from them. We describe various methods for helping computer vision domain experts take advantage of parallelism and how they are deficient in terms of efficiency and portability.

Chapter 5 discusses the parallelization of support vector machine training and classification. We discuss the problem and show that parallelizing it on GPUs can give speedups of an order of magnitude or more compared to the standard LibSVM library. We show how transformations to convert BLAS2 to BLAS3 routines in SVM classification can produce massive improvements in performance.

Chapter 6 describes the parallelization of optical flow and point tracking. We discuss the algorithmic exploration of different linear solvers for solving large displacement optical flow on GPUs and compare against a previously published serial implementation. We describe how this efficient optical flow implementation can form the core of a point tracking system that is denser and more accurate than other previously known point trackers.

Chapter 7 discusses the parallelization of image and video segmentation using gPb. We describe how local histograms can be calculated efficiently with no loss of accuracy, how an eigensolver can be made to fit in limited GPU memory, how efficient image segmentation can lead to scalable video segmentation on a cluster of GPUs and how such a video segmentation technique is competitive with other state-of-the-art methods.

Chapter 8 describes how we can efficiently solve the problem of memory management in CPU-GPU systems. We use a combination of pseudo-boolean (or equivalently MILP) optimization problems and heuristics to minimize the amount of data transfers between the CPU and GPU. We show that this technique can produce speedups of over $7\times$ compared to unoptimized parallel GPU code.

Chapter 9 concludes the dissertation and discusses possible future work and extensions.

Chapter 2

Background and Motivation

Computer vision algorithms have been increasing their computational requirements over time (like most other areas in computer science). We see that as an increase in both data and computation. In general, the increase in data sizes has been due to the following reasons -

1. Generalization performance of computer vision algorithms improves with increase in training data. Therefore, in order to get good performance on tasks like people recognition [27] or object recognition [70], large datasets for training are a necessity.

2. Data acquisition is getting cheaper. The explosion of image and video capturing devices like digital cameras, mobile phones etc. has led to a tremendous increase in the amount of data generated.

3. Generated data is getting larger. The resolution of images that are being taken with digital cameras and phones has reached the order of 10 megapixels in 2011 (up from 0.15 megapixels in 1990). It is common for mobile phones to record at 1920×1080 reesolution at 30 fps.

4. Generated data is more accessible. The growth of online social networking in recent years has accelerated the generation and distribution of user-generated multimedia content. Growth of popular image and video sharing websites like flickr (http://flickr.com), youtube (http://youtube.com) and others has led to sharing (and public availability) of the generated data.

5. Data annotation is getting cheaper. Ability to get cheap human annotated data through tools like Amazon Mechanical Turk (https://www.mturk.com) has enabled the creation of large annotated data sets (e.g. ImageNet [64]).

6. Development of large scale machine learning techniques (like linear time training for SVMs [101]) has enabled us to learn from large amounts of training data.

We find that this increase in data is of the order of $1000\times$ per decade.

Computational requirements have always increased as researchers tend to more and more sophisticated algorithms over time. In general, algorithmic complexity for solving particular problems (like image contour detection or optical flow) has been increasing at a rate of about $10\times$ per decade. Combined with increasing data set sizes and image data resolution, this shows that the hardware requirements for computer vision algorithms are increasing at the rate of $10,000\times$ per decade.

On the other hand, hardware performance has also been going up. Moore's law dictates that the number of transistors on chip roughly doubles every 18 months. Through voltage and frequency scaling, we were able to get improved performance with more transistors. Unfortunately, voltage scaling has slowed down significantly by early 2000's and power density kept increasing until we could no longer cool the chip. With microprocessor architectures hitting the power wall, parallelism seemed to be the way out for keeping in line with Moore's law. However increased transistor count means increased performance only if we can take advantage of parallelism. This has meant that computer vision researchers and application developers must worry about their algorithms being amenable to parallelism. When Moore's law translates directly to performance, we get a $100\times$ benefit every decade from hardware improvements. This has led to the use of clusters and datacenters for doing large scale problems in computer vision. We claim that even with parallelism, hardware improvements are barely keeping up with the pace of algorithmic development. Therefore, parallelism and performance optimizations are not optional for scaling computer vision algorithms into the future, but are necessities. Without efficient parallelization, there is no hope of hardware improvements catching up to application requirements.

In the rest of the chapter, we will look at all of the factors driving computational requirements for computer vision in detail, and describe their scaling with specific examples.

2.1 Trends in Computer vision

This section discusses the scaling of computational requirements in computer vision algorithms. We look at the rise of computational requirements as the result of scaling in three distinct, but complementary axes - size of unit data (image or video), data set sizes and algorithmic complexity.

2.1.1 Size of unit data

The size of a single image is dependent on its resolution and content. Since storage has become extremely cheap and image sensors have become large, it is possible to take images of extremely high resolutions even on consumer grade digital cameras. Figure 2.1 shows the evolution of image resolution in consumer grade cameras. With the increasing presence of cameras in mobile phones, similar resolution improvements have been observed there.

From Figure 2.1, it is clear that the amount of data generated per image has been increasing at the rate of $1.29\times$ per year or approximately $12\times$ every decade.

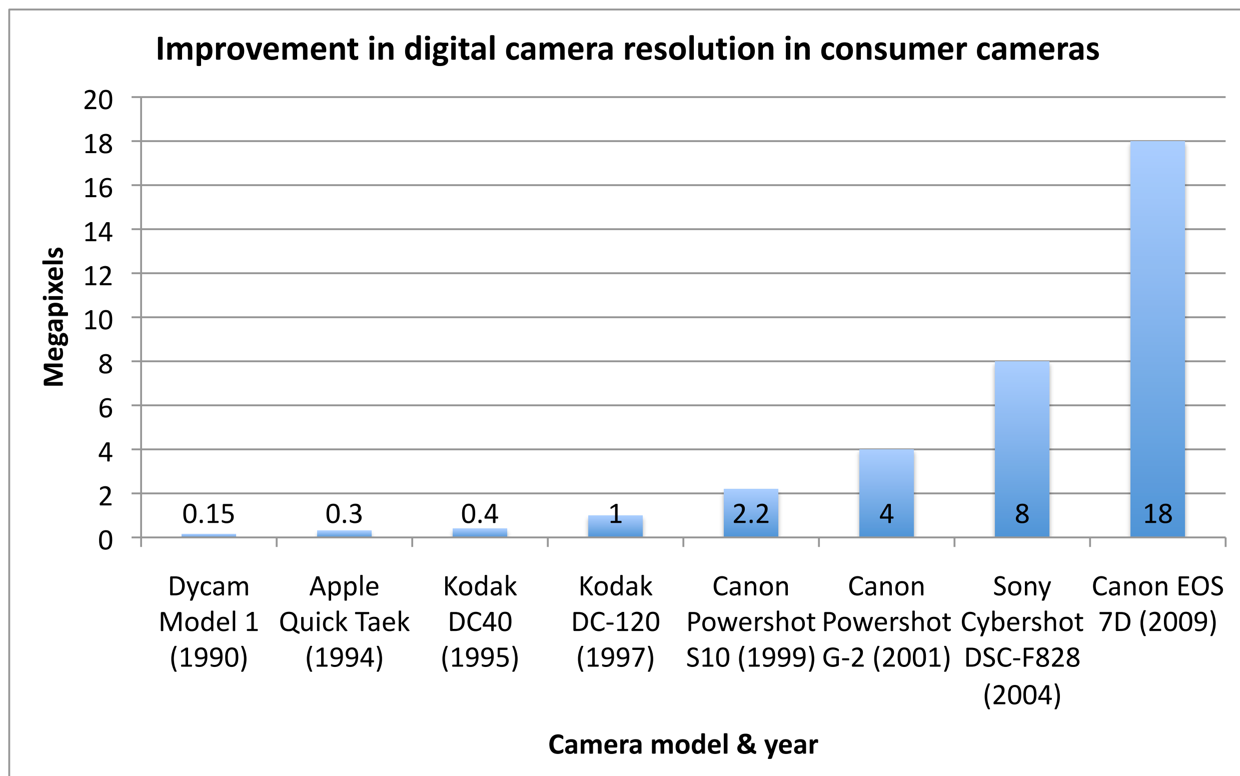


Figure 2.1: Improvements in digital camera resolution in consumer cameras from 1990 to 2009. Each of the cameras shown was (for its time) the consumer grade camera with the most "Megapixels".

2.1.2 Dataset sizes

Computer Vision uses larger and larger data sets and it is already at the point where it is impossible to run credible experiments on real data without access to a small/medium cluster. For example, the sizes of face data sets over the years is shown in Figure 2.2. The figure clearly shows the accelerating trend.

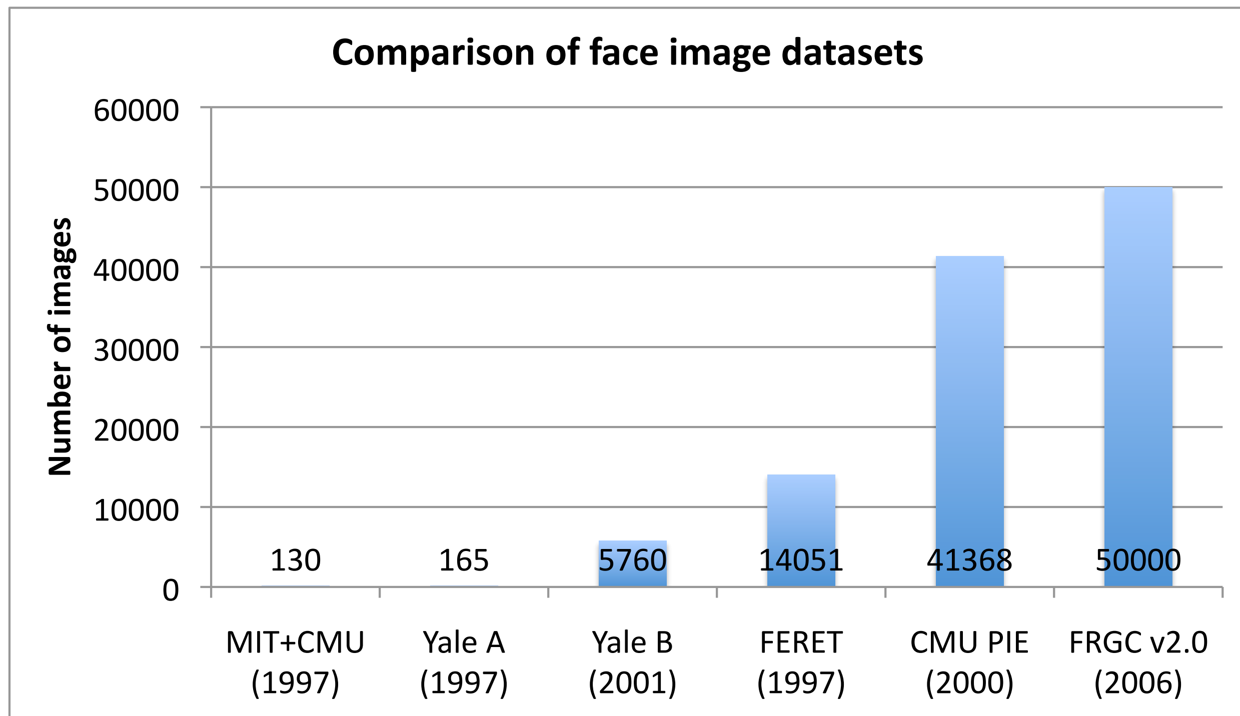


Figure 2.2: Increase in size of popular face recognition data sets from 1997 to 2006.

Let us assume that for this problem, training time scales linearly i.e. as $O(n)$. Since the dataset has been growing at more than $100\times$ a decade, training times also scale at least as much. If training time grows super linearly (e.g. as $O(nlogn)$), then the compute requirement grows an order of magnitude faster than the dataset size. Other problems like segmentation, object recognition, 3D reconstruction etc. also have similar growth in dataset sizes.

2.1.3 Algorithmic complexity

It is not just that image resolution and dataset sizes are growing; the complexity of the algorithms being performed on this data is also growing significantly. We define complexity in this context as the amount of floating point operations performed per pixel (thus normalizing for resolution and dataset sizes) by the algorithm. We look at two important categories of computer vision algorithms - image contour detection and optical flow. Figure 2.3 and 2.4 show the growth of complexity of the algorithms over the years.

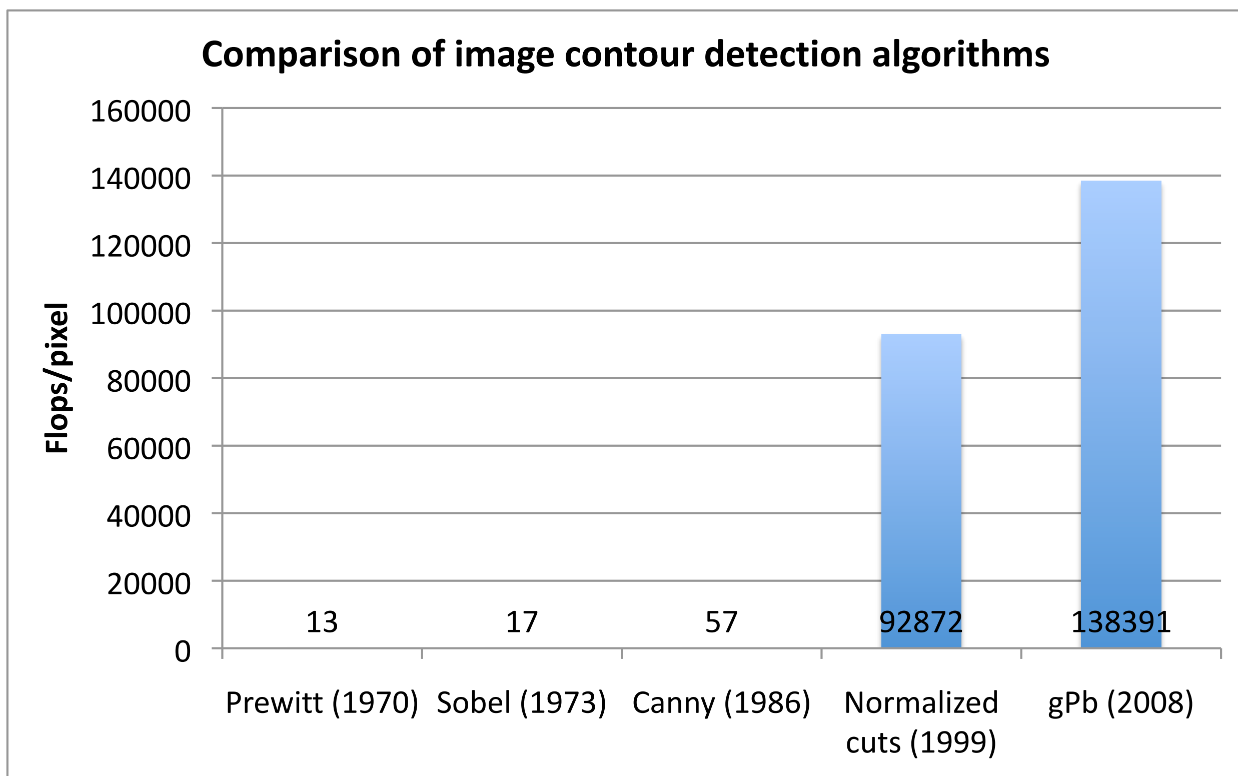


Figure 2.3: Increase in algorithmic complexity of image contour detection algorithms. Numbers shown are estimates based on the amount of computation performed.

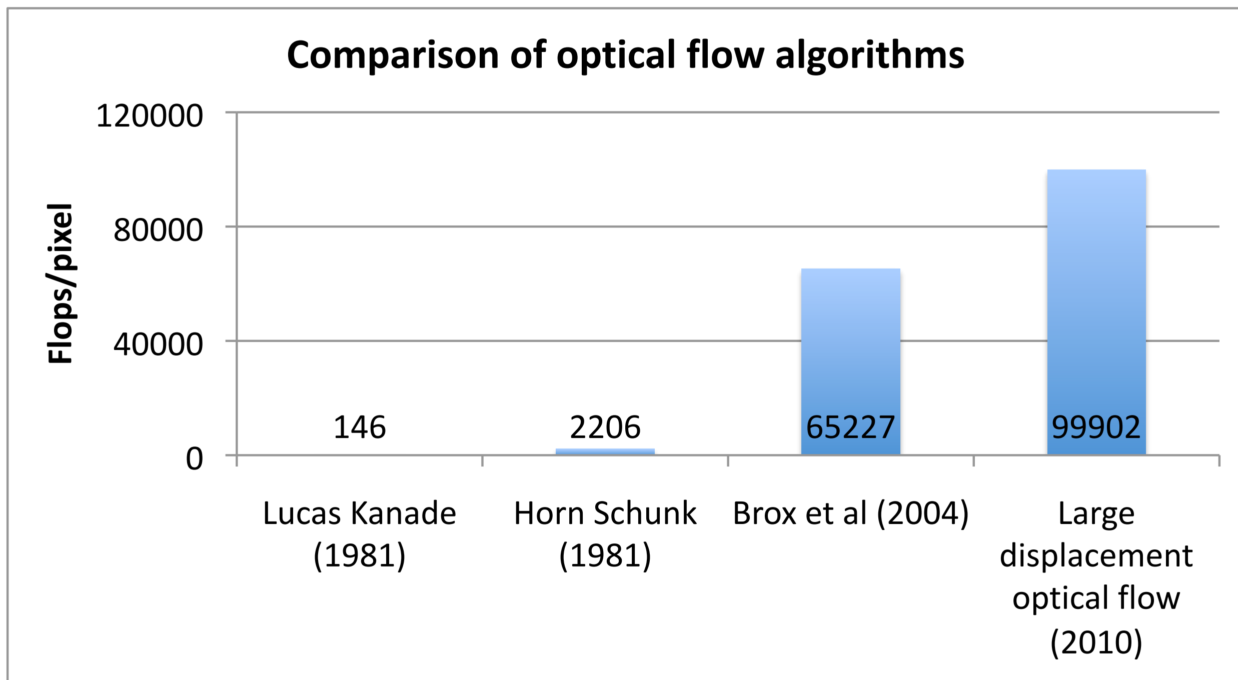


Figure 2.4: Increase in the algorithmic complexity of optical flow algorithms. Numbers shown are estimates based on the amount of computation performed.

From the figures, it can be seen that the complexity is increasing roughly at the rate of about 10× per decade i.e. every 10 years, we do an order of magnitude more floating point operations per pixel.

It is also to be noted that data scaling and computational scaling are multiplicative i.e. data gets larger and larger and we do more processing per unit of data. This means for future hardware to scale with the application requirements, the rate of growth of raw hardware performance must keep pace or be higher than the growth of application requirements. From the previous sections, this growth in computational requirement is of the order of $10,000\times$ per decade.

2.2 Hardware parallelism

Hardware scaling has long been dictated by Moore's law, which says that transistor density doubles every 18 months. This growth has been possible due to improvements in semiconductor devices, materials, software for designing and modeling chips, manufacturing improvements etc. This growth has led to a tremendous increase in the number of transistors in microprocessors. For example, the Intel 8086 had 29,000 transistors in 1978, and the Intel Sandybridge Core i7 Extreme has 2,270,000,000 transistors in 2011.

Until about 2004, the increased transistor count had led to improved performance through a combination of voltage and frequency scaling at the device level and Out-of-

Order execution/VLIW/SIMD superscalar designs at the microprocessor architecture level. Around 2004, newer chip designs had reached their limits on voltage-frequency scaling so that chip frequencies could no longer increase without overheating the chip. Parallelism turned to be the way out of this morass. Replicating more processors on a chip turned out to be a cheap and efficient way to leverage increasing transistor counts without increasing the power density.

Also important to note is the fact that parallelism is happening at all hardware form factors. We have mobile clients (phones, tablets, laptops), non-mobile clients (desktops, workstations) and servers (data centers, cloud computing), all of which are getting more parallel.

GPUs started as special purpose dedicated functional units devoted to 3D graphics processing in the 1990's. However, because of the architectural features they provided (massive parallelism, high bandwidth to DRAM etc.), they began to be used for general purpose computing. GPUs have been getting more and more parallel and hence take advantage of Moore's law quite easily through increasing the number of cores/FPUs. Since their performance comes from parallelism, they perform well on throughput-oriented workloads (as opposed to latency-oriented workloads). Figure 2.5 shows how much the performance of GPU and CPU hardware has increased in the last 7 years.

The figure shows that peak throughput capabilities of GPUs are increasing at the rate of 1.8$\times$ every year or more than 100$\times$ every decade. Bandwidth, however, is increasing linearly about 24 GB/s every year.

2.3 Summary

The hardware scaling (even with parallelism) is improving about 100$\times$ a decade, whereas computer vision algorithms scale much faster (about $10,000\times$ a decade). Without parallelism, hardware improvements have more or less stalled. This means that computer vision researchers and application developers can no longer assume that their software will run faster just by moving to a newer generation processor unless it takes advantage of parallelism. Computer vision researchers need to focus not just on parallelization, but also numerical optimizations in their algorithms.

The move to parallelize and port a lot of computer vision applications to GPUs benefitted from the high memory bandwidth in GPU hardware (compared to CPUs). Even with parallelism exploited, there is almost a 100$\times$ gap between the application requirements and hardware scaling. We believe that a large part of this gap will be closed through better numerical optimizations. If done, we can get closer to exploiting every last flop of performance from the hardware in order to scale efficiently.

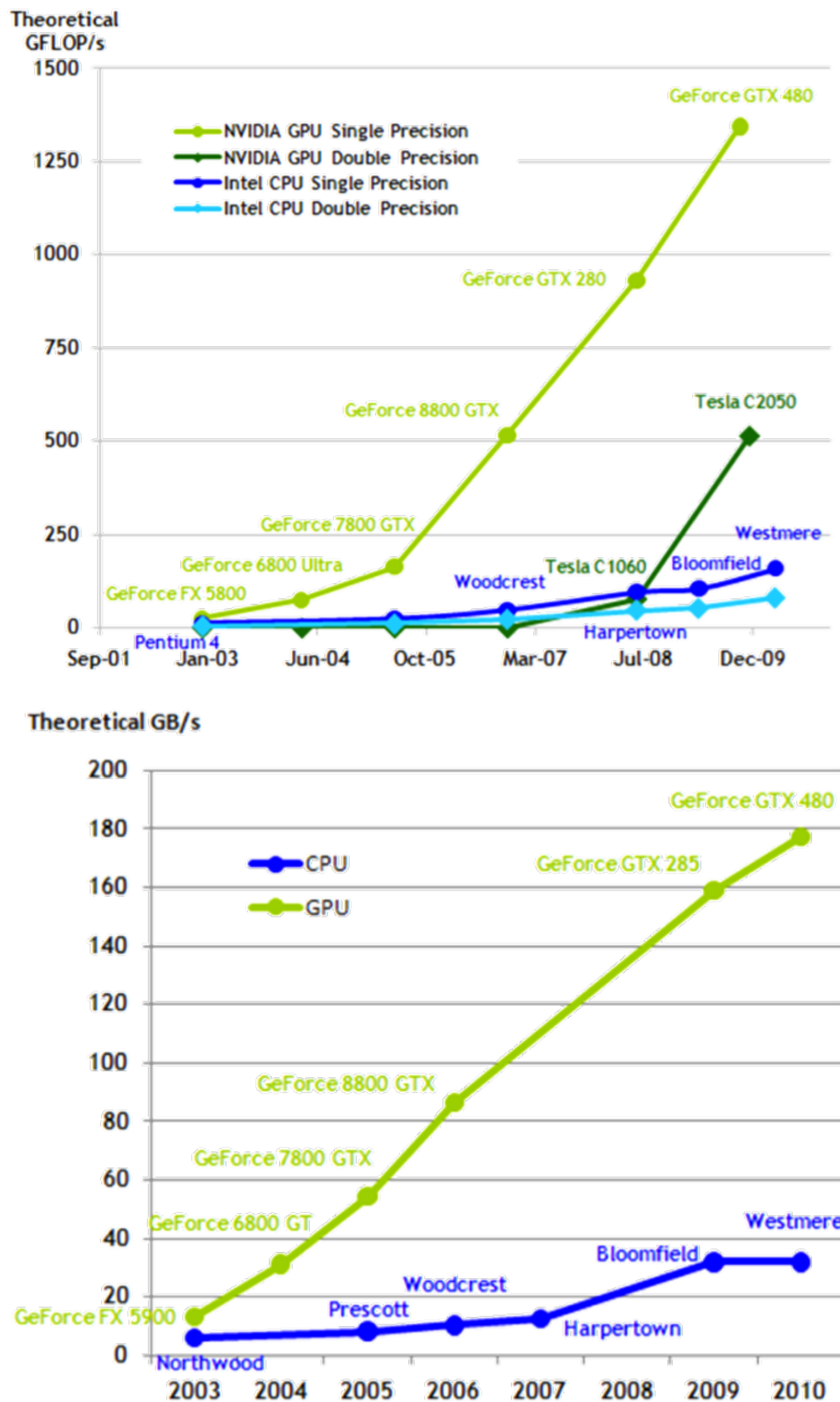


Figure 2.5: Hardware scaling - (a) Improvement in peak theoretical floating point operations per second for CPUs and GPUs from 2003 to 2010. (b) Improvement in memory bandwidth for CPUs and GPUs from 2003 to 2010.

Chapter 3

Understanding Computer Vision Workloads

This chapter discusses the major computational trends in computer vision in terms of algorithms, data structures, approaches to parallelization etc. There has been a lot of research about making processors more amenable to computer vision applications which is discussed in Chapter 4. However, there are not enough rigorous measurements and studies on what kernels constitute computer vision applications. Questions such as what are the main data structures and algorithms used, how they are used and what are the computational requirements etc. have not been satisfactorily answered. We have tried to quantify as much as possible the kinds of computations that characterize computer vision and the computational patterns that they encompass.

Computer vision algorithms are increasingly becoming more and more important workloads. With the advent of smartphones and tablets with powerful cameras, it has become easier to capture high resolution images & videos of the world around us. With increasing data set sizes and computational requirements, future processors have to be good at processing computer vision applications. However, workload analyses of computer vision applications have remained inadequate.

Chen et al [51] and [52] present some workload analyses on computer vision applications. However, they have been constrained in their choice of applications, which are restricted to body tracking and video surveillance respectively. While they argue that parallelization on multicore CPUs is important, it is not clear how many applications are parallelizable or have already been parallelized, what patterns of parallelism are exploited, and what workloads are common in computer vision applications in general. The need to recognize which workloads are essential for computer vision is important if future hardware is required to meet the computational demands of computer vision algorithms. We need to move from the analysis of a single application to looking at a collection of different applications to improve our understanding of what constitutes computer vision computations.

3.1 Patterns and OPL

Patterns are a well understood way to analyze and categorize computational kernels. The Berkeley View paper [10] showed how categorizing computations is key to making them more efficient and to help design better hardware. Kurt Keutzer and Tim Mattson [104] describe a pattern language at multiple levels of abstraction from the application software structure and computations through algorithmic strategy, implementation strategy and hardware. We intend to use this pattern language (Our Pattern Language or OPL) to categorize and analyze computer vision applications. We believe this can help us not only identify key computations used in computer vision algorithms, but also get quantitative information on how prevalent they are in the domain. Figure 3.1 shows the structure of the pattern language.

The primary purposes of OPL are software design and pedagogy. Having a consistent vocabulary for describing computations and strategies for parallelization is crucial to understand and analyze parallel implementations. OPL is a layered, hierarchical pattern language. Each level in the hierarchy addresses a portion of the design problem. Structural patterns describe the overall organization of the application and the way different elements of the application interact with each other. Computational patterns describe the classes of computations that make up the applications. Concurrent algorithmic strategy patterns define high-level strategies used to exploit concurrency in a computation for execution on a parallel computer. Implementation strategy patterns are structures that are realized in source code to support the program and data structure organization. Parallel execution patterns are approaches used to support execution of a parallel algorithm in hardware.

The structural and computational patterns are not restricted to parallel applications, hence are useful in the analysis of workloads in general. Computational patterns are the main layer at which we will be analyzing our applications. Computer vision problems are usually written as constrained optimization problems (continuous or discrete). Out of all the computational patterns, we expect computer vision to consist of dense and sparse linear algebra, graphical models and graph algorithms, Monte Carlo and spectral methods. Of these, we focus on linear algebra and graph algorithms as they seem to be the most relevant and widely used.

Readers are advised to refer to [104] for more details about the pattern language.

3.2 Workload collection & Analysis

For our purposes of analyzing computer vision applications, we need to pick a suitable source of computer vision applications. For that purpose, we pick research papers from Computer Vision and Pattern Recognition (CVPR), International Conference on Computer Vision (ICCV) and European Conference on Computer Vision (ECCV) since 2007. We look at a subset of papers available on-line at *cvpapers.com*. These are not biased along any particular dimension, and hence we think they provide a fair comparison. The decision to use research papers was made for the following reasons:

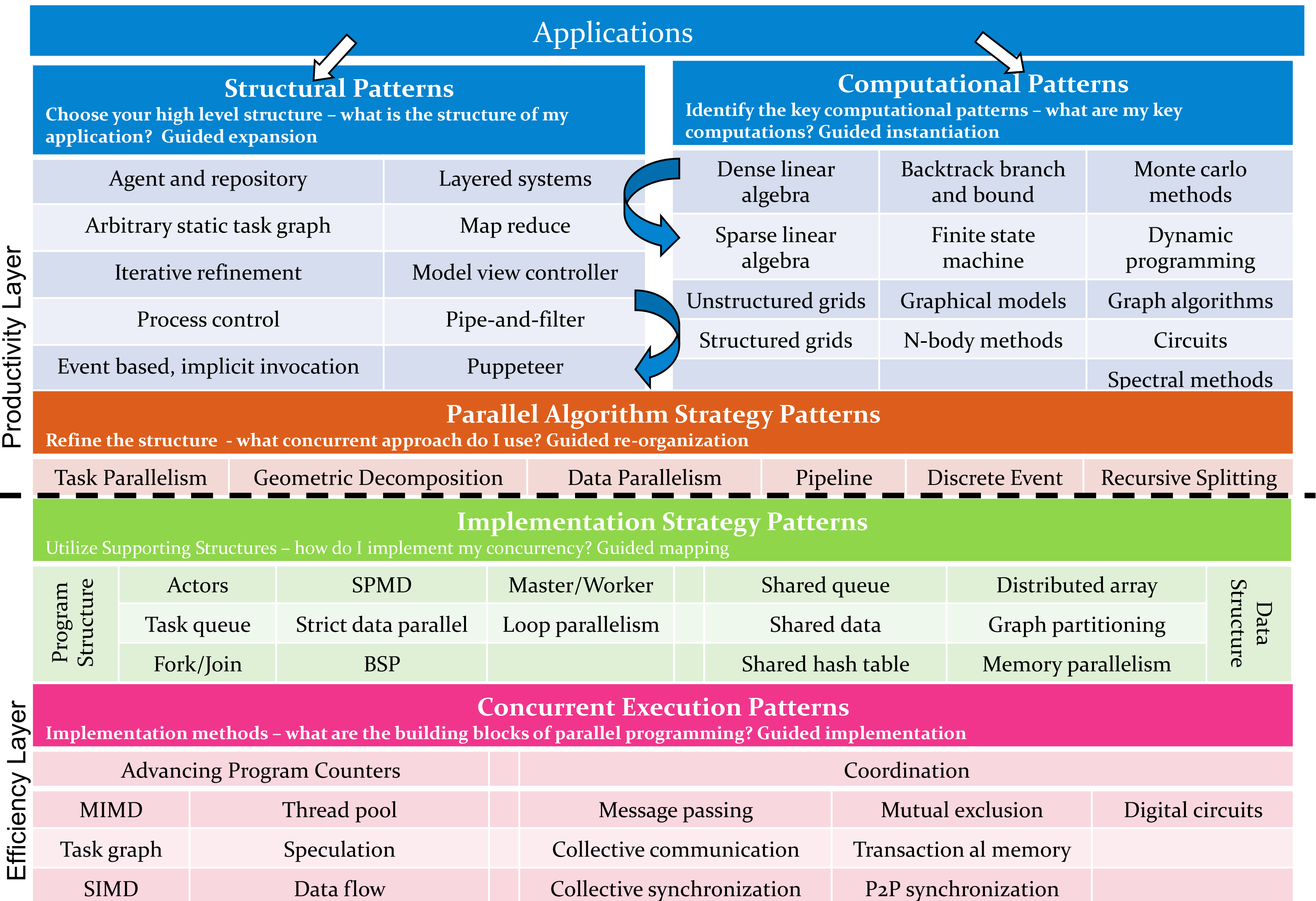


Figure 3.1: Organization of OPL

Conference	Number of papers	% of papers
CVPR 2007	4	4.2%
CVPR 2008	4	3.2%
CVPR 2009	4	2.5%
ICCV 2009	14	6.2%
CVPR 2010	13	7.9%
ECCV 2010	22	10.0%
CVPR 2011	21	6.6%
ICCV 2011	20	9.0%

Table 3.1: Papers referring to parallelism.

1. The algorithms are public unlike proprietary software products made by different companies, so they can be analyzed.
2. They indicate the cutting edge of research in omputer vision, so are likely to indicate the algorithms that will become commonplace in a few years.
3. It is possible to ascertain any emergent trends in algorithms and computations.

3.2.1 Parallelism

First, we look at trends in parallelizing applications in computer vision literature. The simplest way to look at the prevalence of any concept in a body of literature is to see how many times the said concept is referred to in the literature. Hence, we perform our tests on how many references there are to the word "paralleli" (referring to parallelize, parallelism, parallelization etc.) in papers from CVPR 2007 till ICCV 2011. Results are summarized in Table 3.2.1.

In addition to this simple check, we did an analysis on the papers that appeared in ICCV 2011 that included the word "parallel" in order to understand how the algorithms were parallelized and the patterns that were present in the implementation. Out of the 20 papers that talked about parallelized implementations, 9 had used parallel implementations for their experiments while the others only describe their algorithm's potential for parallelization (without a parallel implementation). Interestingly, all the parallelized implementations were done using GPUs except one which was parallelized on clusters. GPUs have become an important computational accelerator for computer vision applications. While the authors of all the 20 papers seemed to understand the importance of parallelism, less than half of them have taken the effort to actually parallelize their code. This proportion must increase if all computer vision computations are to be parallelized so that they can take advantage of Moore's law.

Among the 8 papers that used parallel implementations on GPUs, the predominant algorithmic pattern of choice is data parallelism. Some of the problems were embarrassingly parallel (training independent problems, pixel-parallel computations etc.), but some were

Category	Number of papers
Quadratic Programming/Support Vector Machine	10
BLAS2 (Matrix-vector) operations	9
Linear solver	8
Other continuous optimization	7
Conditional Random Fields	4
BLAS3 (Matrix-matrix) operations	3
Eigenvalue Decomposition	2
Singular Value Decomposition	2
Fast Fourier Transform	2

Table 3.2: Papers referring to linear algebra (out of 50 randomly chosen papers in ICCV 2011).

more complicated (sparse linear algebra). All of the applications are bandwidth bound, which also explains why the performance improvement from GPUs has been significant as GPUs have much higher bandwidth to memory compared to CPUs.

Let us now look at all the applications (not just parallel implementations) to see what computations are most common among them.

3.2.2 Linear Algebra

It is no surprise that numerical linear algebra is important for computer vision applications. Most problems in computer vision are modeled as continuous/discrete optimization problems and require at least some linear algebra in their solutions. We look at a random set of 50 papers from ICCV 2011 and categorize them according to the specific linear algebra problems they solve. A breakdown of linear algebra categorization in the papers is provided in Table 3.2.2.

Most of the linear algebra problems are sparse. Only a few singular value decomposition (SVD), support vector machines (SVM), BLAS2 and BLAS3 problems involve dense linear algebra. It is clear that SVMs and its variants are quite popular machine learning tools in use in computer vision. We look at the computations more closely.

Linear solver

Linear solvers are commonly used in many different computer vision applications. In particular, they are commonly used in solving non-linear optimization problems which are solved iteratively. Each iteration usually involves a linear solve (e.g. Newton's method). They are also useful for implicitly solving differential equations after discretization. They appear in object recognition, compression, pose recognition, 3D reconstruction and machine learning problems [20, 187, 54].

Eigensolvers

Eigensolvers are predominantly found in segmentation problems. The 2 papers that use eigensolvers are related to (a) segmentation and (b) object recognition that is performed after segmentation [120, 8].

SVD

Singular Value Decomposition is used in object recognition, computational geometry and 3D geometry problems [82].

Quadratic Programming/SVM

Support vector machines and its variants are very popular in solving many learning problems in computer vision. Linear SVMs are particularly useful for learning from large datasets. They are useful in object & image recognition, attribute learning and identification of parts of objects [65, 8].

Conditional Random Fields

Conditional random fields are useful for structured prediction and are used in object recognition, image segmentation and learning problems in computer vision [53, 142]. CRFs are undirected probabilistic graphical models. [137] shows how CRFs can be used for object recognition. Parameter learning in CRFs is usually solved through convex optimization approaches such as the Quasi-Newton L-BFGS [42] method. Inference and parameter learning computations utilize BLAS operations for tractable distributions such as the ones belonging to the exponential family.

BLAS2/BLAS3 Computations

BLAS2 computations are similar to dense matrix-vector multiplication and are used in solving optimization problems with gradient descent-like approaches in object recognition, 3D reconstruction, computational photography etc. BLAS3 computations (dense matrix-matrix multiplication) are also part of many algorithms. Distance calculations between sets of vectors, iterative optimization problems etc. invoke BLAS3 routines [107, 171].

Fast Fourier Transform

FFTs are used in computational photography, compression and image matching applications [186].

3.2.3 Graph algorithms

Graphs are another important data structures in computer vision algorithms. They are used to describe relationships between pixels, objects, images etc. Large graph data structures can be structured or unstructured. Some of the graph manipulations like partitioning

Category	Number of papers
K nearest neighbors	4
Shortest path	1
K shortest paths	1
Tree/graph traversal	1
Minimum spanning tree	1

Table 3.3: Papers referring to graphs/trees (out of 50 randomly chosen papers in ICCV 2011).

can be done using linear algebra techniques like eigensolvers. However, there are purely graph-specific techniques like minimum spanning trees, max flow min cut, breadth first search etc. that are specific to these data structures. We list the most common operations on graphs as seen among ICCV 2011 papers [187, 29] in Table 3.2.3.

While we do not consider graph algorithms specifically in the rest of this thesis, this data is presented here in order to provide a view of an important subsection of computer vision algorithms and as a reference to future workload analysts.

3.3 Summary

Linear algebra is at the core of most computer vision applications. Parallel and GPU implementations are becoming more and more important and there is a need to parallelize all computer vision algorithms.

In the rest of the dissertation, we will look at quadratic programming, linear solvers, eigensolvers which together occur in about 50% of computer vision algorithms. We will look at these important kernels that cover a significant part of the computational spectrum. Our approach to algorithmic exploration and productivity improvements will help computer vision application writers by showing that choosing the right algorithm is just as important as performing low level optimization for mapping applications to GPUs.

The next chapter will look at existing approaches to parallelizing computer vision applications and how they are insufficient for solving the productivity and efficiency problems for computer vision domain experts.

Chapter 4

Parallelizing Computer Vision

From Chapter 3, it is clear that many computer vision workloads involve large linear algebra problems with matrix sizes proportional to the number of pixels in an image (matrix sizes of several hundred thousand rows are common), and that parallelization is increasingly becoming common. In this chapter, we will describe how researchers have tried to optimize computer vision algorithms in order to improve their performance. We will examine specific problem domains in computer vision and look at how computational bottlenecks have been overcome by researchers.

There are two aspects to improving the performance of computer vision applications on parallel machines. One is to rethink the numerical algorithms involved in the applications, thereby improving performance on all architectures. This may be done through reformulation of the problem, using different numerical techniques, and changing the problem model. Numerical optimization is, in general, dependent on the particular algorithm that is being optimized and hence is application-specific. We will discuss numerical optimizations in two application domains - image contour detection and optical flow and show how our optimizations were superior to prior work. We will also quantify the performance improvements obtained through such algorithmic exploration. Details of the algorithms and optimizations will be presented in Chapters 5, 6 and 7.

The second way to improve the performance of computer vision algorithms is to tailor the implementation to a particular hardware platform such as GPUs or multicore CPUs, thereby improving running time on that platform. This approach is applicable to a variety of different algorithms and applications. However, these optimizations are usually tied to a particular hardware platform. We will discuss in this chapter different tools and techniques that have been used to parallelize computer vision algorithms and how our optimizations for memory management in CPU-GPU systems and efficient mapping of algorithms to hardware are advantageous. Details of these techniques will be explained in Chapter 8.

We will show how both of these approaches have helped accelerate the performance of computer vision applications like image contour detection and optical flow. We present the numerical optimizations in image contour detection and optical flow followed by the discussion on tools for parallelization.

4.1 Numerical optimizations for image contour detection

Image contour detection involves finding the most important object boundaries (contours) in an image. The difficulties in identifying accurate boundaries in images or videos are due to the fact that there is no precise mathematical definition of "object", objects are not always distinct from the background, identification depends on scale e.g. small objects are often perceived as texture or background rather than as separate entities. Chapter 2 showed the computational evolution of image contour detection algorithms. Starting with simple edge detection techniques, there have been several algorithms for accurate image segmentation. The aim of these techniques have been to try and match human performance on this task. We describe normalized cuts [151] and its modifications, which are among the state of the art today for image contour detection and segmentation.

4.1.1 Background

An image can be modeled as an undirected graph whose vertex set is the set of all pixels. Edges exist between every pixel and its neighbors within a radius of size r. The weight of an edge $w(i, j)$ is a function of the similarity of pixels i and j. In this model, image segmentation is equivalent to graph partitioning. A partition (or cut) of a graph G is a division of the vertex set V into two disjoint sets A and B.

Normalized cut is a way of measuring the amount of dissimilarity between the vertex sets A and B of a partition. For an undirected graph G with vertex set V, the normalized cut cost for a division into two sets of vertices A and B is given by

$$\text{NormalizedCut}(A, B) = \frac{Cut(A, B)}{Assoc(A, V)} + \frac{Cut(A, B)}{Assoc(B, V)} \tag{4.1}$$

where $Cut(A, B)$ denotes the total weight of edges between A & B, i.e. $Cut(A, B) = \sum_{u \in A, v \in B} w(u, v)$ and $Assoc(A, V) = \sum_{u \in A, t \in V} w(u, t)$ denotes the total weight of connection between A and all the nodes in the graph.

Shi and Malik showed that minimizing the normalized cut of a graph is related to an eigenvalue problem [151]. If we describe the graph G as an adjacency matrix W, then the graph cut with the smallest normalized cut is obtained from

$$y^* = \min_{y, y \in \{1, -b\}, y^T D 1 = 0} \frac{y^T (D - W) y}{y^T D y}. \tag{4.2}$$

The normalized cut x^* is given by $x^* = \frac{\mathbf{y-1+b}}{1+b}$ where D is a diagonal matrix whose elements correspond to the sum of all affinities in a row, i.e. $D = W \cdot \mathbf{1}$, $\mathbf{1}$ is a vector of ones and $b = \frac{\sum_{x_i > 0} D_{ii}}{\sum_{x_i \leq 0} D_{ii}}$. W is also called the affinity matrix in the context of image segmentation because it represents the affinities between pixels. For images, affinity between two pixels i and j can be modeled simply as being proportional to the $e^{-\alpha ||I_i - I_j||^2}$ where I_i and I_j are

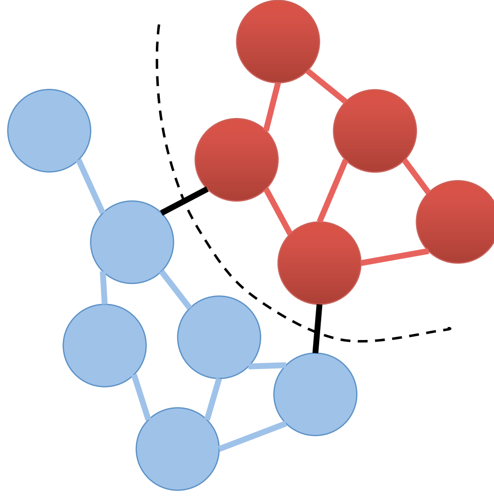


Figure 4.1: Illustration of normalized cut. For the graph cut denoted by the black dotted line, the normalized cut is given by $\frac{\text{Sum of black edges}}{\text{Sum of red and black edges}} + \frac{\text{Sum of black edges}}{\text{Sum of blue and black edges}}$

the respective intensity values [151]. This affinity metric has been improved later by Maire et al [119] and can be calculated as the maximum value of local image gradients along a straight line drawn between the two pixels.

This problem of minimizing normalized cuts is NP-Hard. If the discrete solution is relaxed to take real values, then the solution is given by the eigenvectors of an eigensystem described below:

$$D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}z = \lambda z \tag{4.3}$$

$$z = D^{\frac{1}{2}}y. \tag{4.4}$$

Restricting the generalized eigenvector y to two values only (through the selection of a good splitting point) gives us a two-way segmentation.

Eigenvectors corresponding to the smallest eigenvalues of this matrix represent segmentations. Note that the smallest eigenvalue is zero and the corresponding eigenvector is ignored. Figure 4.2 shows the first 2 eigenvectors corresponding to the affinity matrix for the image shown. Affinity values have been calculated between each pixel and its neighbors in a 5-pixel radius. It is easy to see that the eigenvectors correspond to segmentations of the image.

4.1.2 Problem description

The dominant portion of the runtime of the gPb detector [119] is the eigensolver. For images in the Berkeley Segmentation Dataset (BSDS), the resolution is typically 481×321 (0.15 megapixels). Performing image segmentation on one such image takes more than 4 minutes on a serial machine [47]. This is clearly unacceptably slow for running on a large

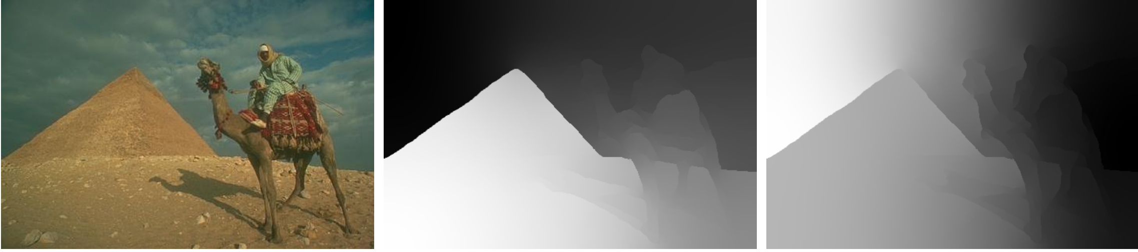


Figure 4.2: Example image from the Berkeley Segmentation Dataset and its first 2 eigenvectors for the affinity matrix derived as in [119].

dataset (1000's of images). Also, no interactive applications can run with such long latencies. Hence effective ways to improve the runtime of the algorithm need to be found.

Given the large amount of parallelism in linear algebra problems, there have been significant efforts to optimize and parallelize normalized cuts and its variations. Most numerical optimizations have tried to increase the convergence of the eigensolver through better initialization from coarse image resolutions or reformulation into non-eigenvalue problems.

4.1.3 Related work

A few of the best known techniques to improve performance and efficiency of normalized cuts/gPb are as follows:

Algorithmic reformulation Since the eigensolver is a substantial computational load for solving image segmentation, reformulation of the normalized cuts problem into non-eigensolver based problems could, in theory, lead to better performance. Dhillon et al [66] reformulated normalized cuts as a form of a weighted kernel k-means algorithm that can be solved efficiently. In particular, it was shown that minimizing the normalized cut is equivalent to a trace maximization problem which in turn can be solved as a weighted kernel k-means algorithm. However, improvements to normalized cuts algorithm like gPb [119, 9] use the eigenvectors themselves for segmentation. Kernel k-means only provides us the graph cut information, which is insufficient. Since gPb has been shown to perform better than normalized cuts for image segmentation, these reformulations have not remained popular for image segmentation problems where the eigenvectors themselves are needed. However, these reformulations are useful in other contexts like clustering data or partitioning general graphs where only the cuts (and not the eigenvectors) are needed.

Numerical optimizations Existing numerical optimizations try to accelerate the convergence of the eigensolver through improved initialization or speed up the sparse matrix vector multiplication through approximations. Tolliver et al [165] use a multigrid method on an image pyramid. The eigenproblem is solved fully at coarse levels and the eigenvectors from the coarse levels are used to initialize solutions at finer levels.

These are then refined using inverse iterations. Other techniques like [59] approximate the affinity matrix using low-rank approximations. This provides better speedups for larger graphs with a large connectivity radius, but is not necessarily more accurate in terms of the final segmentation produced.

Other models Image segmentation algorithms not based on normalized cuts have also been recently proposed by Felzenszwalb et al [74]. Their algorithm uses minimum spanning tree on graphs and agglomerative clustering to produce image segmentations. However, [74] faces two problems - the algorithm is sequential and the accuracy trails behind normalized cuts based approaches like gPb [119]. The algorithms may be parallelizable with some relaxations, however this is yet to be explored.

4.1.4 Our approach

In contrast to the approaches mentioned here, our numerical optimizations are based on the properties of the affinity matrix and the problem context. Given that the eigenvectors correspond to segmentations of the image and the eigenvalues correspond to the relative weights of these segmentations, we assume that the segmentations are uniquely weighted for natural images. Mathematically, we can assume that the eigenvalues of the matrix are distinct. Numerical implementations of Krylov subspace methods like Lanczos for eigenvalue problems accumulate errors in their eigenvectors as iterations proceed. These errors are due to the limitations of floating point arithmetic, which if not corrected can lead to the eigenvectors becoming non-orthogonal. Correcting these errors is very expensive computationally and not very parallelizable. However, if the eigenvalues of the matrix are unique, then these errors can be allowed to accumulate and can be corrected later through the Cullum-Willoughby test [60]. We show how this has been exploited for better performance in Chapter 7.

4.1.5 Results

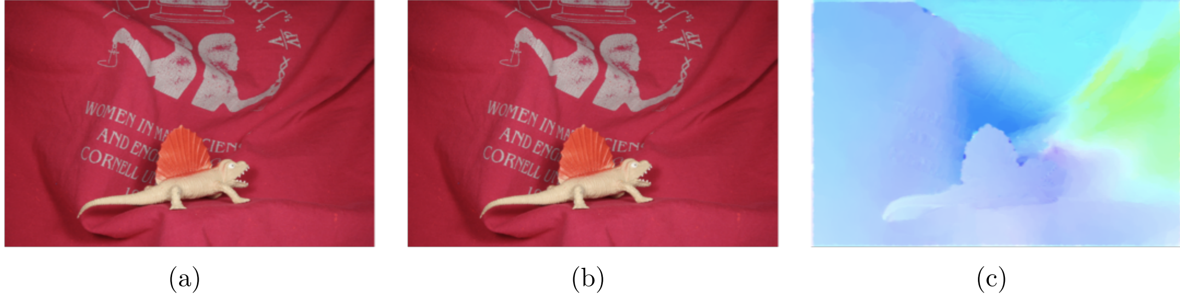
Using the Cullum-Willoughby test and avoiding reorthogonalization of eigenvectors led to a $20\times$ speedup in the runtime of the eigensolver on the Nvidia GTX 280 GPU. In conjunction with the parallelization of the gPb image contour detector on the GPU, the runtime of our implementation was less than 2 seconds compared to greater than 4 minutes earlier. More details on the algorithmic exploration and the results of parallelization on multiple platforms are discussed in Chapter 7.

We now look at numerical optimizations in another application domain - optical flow and see how our approach outperforms earlier ones.

4.2 Numerical optimizations in Optical flow

Optical Flow involves calculating the displacement map or flow field between two consecutive frames of a video sequence with no constraints on camera or object motion. Figure

4.3 shows an example of an optical flow field given two frames from a video. Optical flow is a required intermediate step for many computer vision algorithms on video sequences.



(a) (b) (c)

Figure 4.3: Two consecutive frames (a) and (b) from the Middlebury dataset [14] and the calculated optical flow using [159] (c).

4.2.1 Background

In order to find the displacements of pixels from one frame to another, it is necessary to assume that the properties of points do not change between the two frames i.e. corresponding points in the two frames are likely to have similar brightness, colors, gradients etc. The key principle in optical flow algorithms is this idea of constancy. For example, assume that the brightness values on the points on an object do not change as the object and/or camera moves. Let the video be represented as a 3 dimensional brightness signal $I(x, y, t)$. Mathematically, brightness constancy is expressed as $I(x, y, t) = I(x+u, y+v, t+1)$ where u and v are the displacement vectors for the point (x, y, t). Assuming linear variations of I in time and space, this gives us one constraint that we can use for estimating optical flow:

$$I_x u + I_y v + I_t = 0 \tag{4.5}$$

where I_x denotes the partial derivative of I with respect to x, I_y denotes the partial derivative of I with respect to y, and I_t denotes the partial derivative of I with respect to t. Since we have two variables and only one constraint, we use other constraints such as regularization to make the problem wellposed.

In local methods such as Lucas-Kanade [118], one assumes that neighboring points have very similar optical flow vectors, and hence the optical flow at any point can be recovered by solving a system of linear equations. The system of equations is overconstrained, hence one solves it to minimize the sum of squared error. In global methods like Horn-Schunck [94], one performs an energy minimization of the form

$$E(\vec{u}) = \int_\Omega M(I, \vec{u})dx + \alpha \int_\Omega R(\nabla\vec{u})dx \tag{4.6}$$

where $\Omega \subset \Re^2$ is the image domain, $\vec{u} = [u, v]^T$, M is a data term signifying the brightness

constancy assumption and R is a regularization term signifying smoothness of the flow field. Horn and Schunck, for instance, use $M = ||I(x+u, y+v, t+1) - I(x,y,t)||^2$ and $R = ||\nabla \vec{u}||^2$. Most modern variational techniques are derived from Horn and Schunck's formulation. Horn and Schunck's technique is solved through linearization and discretization of this continuous formulation. The remaining linear system is solved through Jacobi or Gauss-Seidel iterations. The matrix being solved has as twice as many rows as there are pixels in the frame, as one variable indicates the x-displacement and one indicates the y-displacement per pixel.

More sophisticated variational formulations have since been shown to be better at solving optical flow accurately. Shown below is Thomas Brox's formulation of variational optical flow [34].

$$\begin{aligned} E(\vec{u}) &= \int_\Omega \Psi(||I(\vec{x}+\vec{u}, t+1) - I(\vec{x}, t)||^2)dx + \\ &\quad \alpha \int_\Omega \Psi(||\nabla I(\vec{x}+\vec{u}, t+1) - \nabla I(\vec{x}, t)||^2)dx + \\ &\quad \gamma \int_\Omega \Psi(||\nabla \vec{u}||^2)dx \end{aligned} \tag{4.7}$$

In addition to the brightness constancy assumption, this formulation also uses brightness gradient constancy. Instead of using the L_2 norm, the formulation uses a smooth L_1-like norm through the penalty function $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$. This formulation has been shown to be quite accurate in computing optical flow compared to earlier techniques [34].

We use the large displacement optical flow technique which performs both feature matching using histogram of oriented gradients (HOG) and variational optical flow in a single mathematical setting [36]. The formulation is similar to the one shown earlier with added terms of large displacement tracking using HOG features.

Unfortunately for almost all the variational techniques, the minimization problem is non-linear and non-convex, making the search for the global optimum difficult. This problem is overcome using a coarse-to-fine refinement strategy in order to avoid getting stuck in local minima. At each scale, the non-linear problem is solved through fixed-point iterations in order to remove the non-linearities. The innermost (linear) problem is solved using a linear solver.

4.2.2 Problem description

The runtime of the large displacement optical flow algorithm before parallelization is unacceptably high. It takes roughly 68 seconds per pair of 640×480 sized frames. Since most video is captured at a rate of 30 frames/second, this processing represents a slowdown of $2040\times$ compared to data acquisition. In addition, optical flow by itself is not directly useful; it is useful for other video processing algorithms that need motion information.

The challenges in parallelization are two fold - Since the most important part of the problem is the numerical non-linear, non-convex solver, significant performance improvement can be obtained through numerical optimizations. The numerical linear solver, in

particular is a major computational bottleneck and needs to be accelerated for optical flow to be practical. Interestingly, any improvements to the linear solver not just benefits large displacement optical flow, but can be used to improve the efficiency of a large class of optical flow algorithms.

4.2.3 Related work

There is plenty of pixel-level parallelism in the problem, and there are a variety of methods to optimize the problem and improve performance and/or accuracy. A few of these techniques are given below:

Algorithmic reformulations Different optical flow formulations other than the ones shown earlier have been used to varying degrees of success. For instance, [109] uses a convex formulation that is guaranteed to find the global minimum irrespective of initialization. However, the accuracy of these techniques in practice have been inferior to non-convex formulations. Other formulations use L_1 norm as a regularizer instead of L_2 and a total variation based approach [177]. It has been proven that these methods can be solved efficiently through their dual formulations [50] and have similar accuracy to our techniques [14].

Numerical optimizations Since the linear solver is a predominant portion of the computation, many techniques have been tried to improve and optimize this aspect of the problem. For instance, Bruhn et al [40] use different discretizations of the Euler-Lagrange equations that derive from the variational formulation and use a multigrid solver in order to run small optical flow problems in real time. [88] uses a red-black solver instead of a serial Gauss-Seidel solver in order to run on parallel hardware such as the IBM Cell processor.

Parallelization on different hardware There have also been efforts to improve the performance of optical flow algorithms through the use of parallel hardware such as FPGA [176], GPU [85], Cell processor [88] etc. These techniques are useful, but are limited to particular hardware and are usually not portable.

4.2.4 Our approach

In contrast to the above mentioned approaches, we perform an algorithmic exploration of different linear solvers for solving optical flow. The linear solver at the core of the numerical optimization is the most compute intensive portion of the optical flow calculation. Most of the previous work relaxed the serial Gauss-Seidel solver into doing red-black iterations. However, further improvements are possible through better algorithmic exploration. Better solvers are possible if we can exploit the specific properties of the system being solved. In particular, solvers like conjugate gradient require the linear system to be positive definite. In Chapter 6, we prove this and use this fact to construct an efficient preconditioner for the conjugate gradient solver. This improves performance on all hardware including multicore

CPUs and GPUs. Compared to existing serial Gauss-Seidel solvers, parallel preconditioned conjugate gradient solvers are about 4× faster on the CPU. Compared to parallel red-black solvers, parallel preconditioned conjugate gradient is about 40% faster on the GPU.

4.2.5 Results

As a result of our algorithmic exploration and parallelization, we were able to reduce the runtime of the large displacement optical flow algorithm by 37× from over 68 seconds to less than 2 seconds. Of the 37× speedup, about 10× is attributable to the difference in memory bandwidth & hardware from GPUs and the rest to algorithmic improvements. The linear solver alone has had its performance improved by a factor of 53× overall. At this level of performance, we can run LDOF on long video sequences and use it for applications like video segmentation. Details are discussed in Chapter 6.

We now focus on the implementation details and how we can parallelize computer vision applications for running on specific hardware platforms.

4.3 Implementation issues

The performance of the final parallelization depends on how the algorithm is implemented using the parallel programming model of choice for the particular hardware. We will discuss prior work on parallelizing computer vision in general through hardware and software improvements. We will focus in particular on the parallelization of computer vision algorithms on GPUs and multicore CPUs.

4.3.1 Background

Before going into the challenges associated with mapping computer vision algorithms on parallel processors, we briefly delve into the details of current parallel programming models on desktop machines. We look at both GPU and CPU style parallelism and how concurrency in algorithms must be expressed in order to exploit the parallelism.

Nvidia provides a programming environment for its GPUs called the Compute Unified Device Architecture (CUDA) [130]. The user codes in annotated C++, accelerating compute intensive portions of the application by executing them on the GPU.

Figure 4.4 illustrates how the GPU appears to the programmer. The programmer organizes the computation into grids, which are organized as a set of thread blocks. The grids run sequentially on the GPU, meaning that all computation in the grid must finish before another grid is invoked. Grids may be run concurrently if the programmer guarantees that their computations are independent (CUDA streams). As mentioned, grids contain thread blocks, which are batches of threads that execute together, sharing local memories and synchronizing at programmer specified barriers. A maximum of 1024 threads can comprise a thread block, which puts a limit on the scope of synchronization and communication in the computation [130]. However, an enormous number of blocks can be launched in parallel in

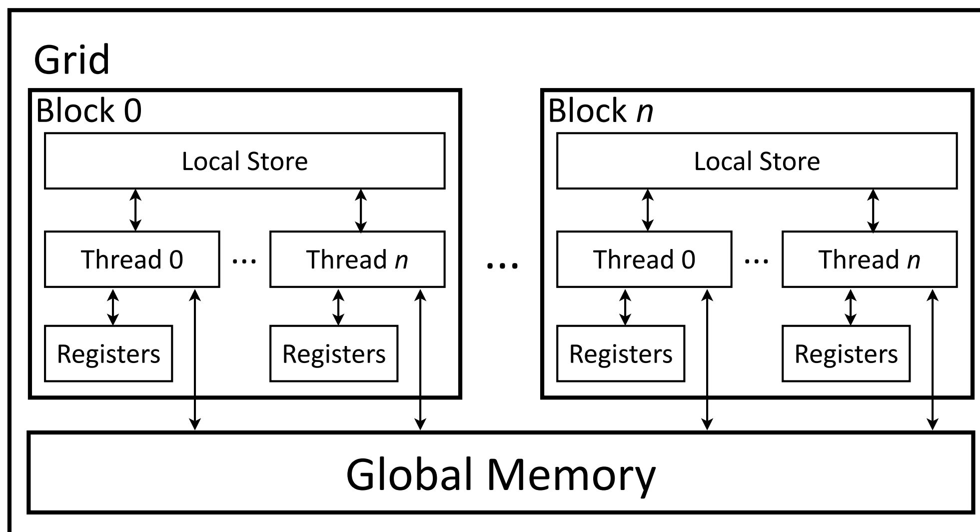


Figure 4.4: Logical organization of Nvidia GPUs

the grid. In practice, we need a large number of thread blocks to ensure that the compute power of the GPU is efficiently utilized.

Multicore CPUs like Intel and AMD's microprocessors have a different programming model. One could think of them logically as a group of cache-coherent processors with vector lanes for performing floating point arithmetic. They are also programmed by instantiating a number of threads typically equal to the number of processors, each of which is capable of running vectorized code. Also, CPUs have typically much larger cache memory compared to GPUs. Exploiting caches is important while mapping to CPUs as memory bandwidth to cache is much higher than bandwidth to DRAM. Figure 4.5 shows how multicore CPU architectures are logically arranged.

Even though CPUs and GPUs have very different models, there are some strong similarities. Both architectures organize hardware parallelism in a hierarchical manner. Both of them have parallelism at the fine-grained level (SIMD) and coarse-grained level (multiple processors). There are however, differences in how threads get scheduled, multi-tasking and preemption, memory latencies and bandwidth etc., and those differences will be highlighted where necessary.

The important question is how computer vision applications map to this programming model. Computer vision applications usually have a lot of parallelism, especially at the pixel level. Given sufficient regularity and locality, this maps well to multicore architectures with each SIMD lane performing computations for a single pixel. There are however, challenges to be overcome which are explained in the next section.

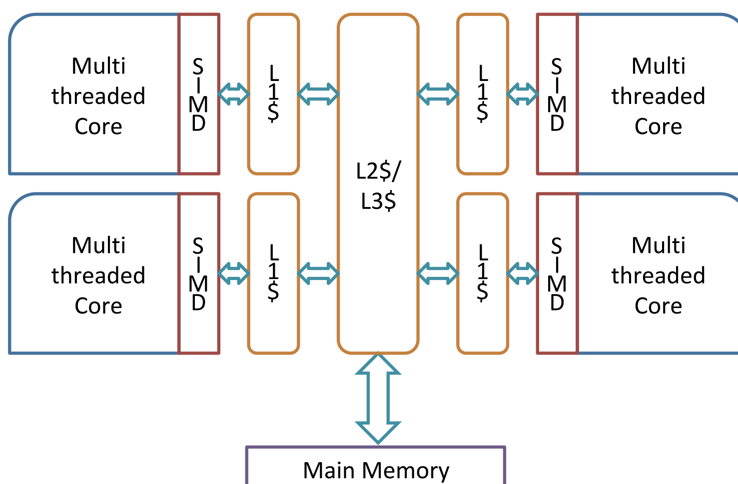


Figure 4.5: Logical organization of multicore CPUs

4.3.2 Problem Description

The challenge in parallelizing computer vision algorithms is not in finding concurrency in the application but in mapping the concurrency efficiently onto the parallel hardware. Most of the tools available today have not been very successful in this regard. As mentioned in Section 4.3.1, multicore CPUs and GPUs have a hierarchy of parallel hardware from SIMD lanes to multiple independent cores. Improving the performance of computer vision algorithms on these parallel platforms remains a challenge. For example, Vision and Cognition were recognized as "Grand Challenge" problems that needed high performance computing involvement in 1992 [2]. The report estimated that by 2000, large computer vision applications would require a floating point performance of over one Teraflops (10^{12} floating point operations) per second. Today we can have several teraflops of performance on a desktop by exploiting GPUs and CPUs. Parallelism is a way to get sufficient performance to run such applications.

One of the major problems in performing an efficient GPU implementation of computer vision algorithms is the disjoint memory space between CPUs and GPUs. For historic reasons, we have been forced into a corner where GPUs have only been able to communicate with the CPUs through a low bandwidth, high latency PCI-e connection. Memory management is a critical aspect of the implementation. For applications that move large amounts of data, the cost of moving data between the CPU and GPU can be very significant. For example, for image convolutions with a 3×3 kernel, CPU-GPU memory transfers take up to 70% of the total runtime. This is a problem that is only bound to get more important in scope as data sizes have been increasing exponentially and DRAM size grows much slower. This problem is not just restricted to GPUs, but to any accelerator model where the accelerator communicates with the CPU using a high latency, low bandwidth connection. Also, every call to the GPU has a latency of a few microseconds. This problem also manifests in

other applications. The eigensolver in image contour detection requires a large amount of memory in order to store the temporary data structures in the computation. Since GPUs only have limited memory, this becomes a problem for smaller GPUs.

4.3.3 Related work

Researchers have been parallelizing a variety of computer vision algorithms on a range of parallel computer hardware. They have tried to achieve performance through parallelism by either creating new parallel hardware architecture well-suited for computer vision or by trying to exploit existing parallel hardware.

Most of the previous efforts at parallelizing computer vision can be classified into one of the following categories -

Hardware synthesis The more difficult (and arguably less successful) option to exploit parallelism for computer vision is to create new hardware capable of handling vision workloads well. The earliest attempt to make a special purpose processor for accelerating computer vision applications is the Image Understanding Architecture by Weems et al [175]. The Image Understanding Architecture differed from other general purpose parallel computers through the use of heterogeneous processors specifically tailored towards different levels of image processing (low, mid and high level). Their claim was that since computer vision computations are very regular at low level and get progressively irregular at higher levels, their architecture was meaningful from an application perspective. Weems et al were able to build prototypes of the proposed architecture; however, the evolution of computer vision algorithms and the improvement in performance of general purpose processors have left such efforts largely unsuccessful in the long run.

Another option to exploit custom hardware for accelerating specific vision tasks is through the use of programmable hardware like Field Programmable Gate Arrays (FPGA). For example, Ratha et al [140] parallelized image convolutions using a custom hardware architecture - the Splash-2, which was based on Xilinx 4010 FPGAs. FPGAs have been used for accelerating compter vision applications because of the inherent parallelism and scalability of these applications. Another reason for the use of FPGAs is their ability to be very efficient at fixed tasks. Some examples of the use of FPGAs in computer vision include [140, 176]. However, difficulty in programming FPGAs and emergence of better programmability in GPUs have made FPGAs less popular for accelerating vision applications [23].

Multimedia processing has long been moved to hardware because of the need for real-time performance. H.264 decoders and encoders (e.g. Intel Quick Sync Video in 2011), MP3 decoders, JPEG encoders for cameras etc. have been around for about a decade now. Specific instructions now exist in almost all architectures for accelerating multimedia functions. Sum of Absolute Difference (SAD) instruction has been present in almost all architectures [155]. Intel introduced it in SSE2 (PSADBW) in 2001, ATI Radeon HD 5870 has had it since 2009, ARM has had it since 2002 (ARM11).

Specific hardware for computer vision have not appeared widely because almost all hardware that supports data parallelism also helps vision applications in general. With SIMD units present in all general purpose hardware (desktop/mobile CPUs, GPUs etc.), vision applications get a sizeable boost in performance.

In general, creating custom hardware for computer vision has not been successful because of low programmability, low productivity and changing algorithms.

Manual mapping In contrast to developing new hardware, efforts to exploit the concurrency in computer vision applications using existing parallel computers have been more successful. Since parallelism is abundant in computer vision algorithms in general, there have been no dearth of efforts to parallelize vision algorithms in order to improve their run times. We cite a few of the relevant literature. Many computer vision algorithms are data-parallel, and hence map well to a variety of different parallel hardware architectures. Prasanna et al [135] parallelized low level computer vision algorithms like edge detection and morphological operations on a parallel machine - the Connection Machine (CM5). Wang et al [172] parallelized several low and mid level vision algorithms including image segmentation, perceptual grouping and object recognition using message passing libraries like MPI [126] on another parallel machine - the IBM SP-2. Much of the work in the 1990's on parallel computer vision focussed on large distributed computing systems and were programmed using MPI.

With the availability of more powerful processors in the desktop and single node multiprocessors (multicore CPUs and GPUs), the focus of parallelization has also shifted to these architectures. Recently, there has been an explosion in the amount of parallelized computer vision implementations for different platforms like GPUs, multicore CPUs and clusters. Many computer vision algorithms have been parallelized on desktop and laptop multicore processors using tools and libraries like OpenMP [131], TBB [164] etc. Computer vision workloads have become part of computer architecture oriented benchmarks like Parsec [17] (body tracking benchmark).

Starting with the use of graphics libraries like OpenGL for offloading compute intensive data parallel kernels to the GPU, vision researchers have explored using GPUs for speeding up their applications. With the development of more general purpose frameworks like BrookGPU [41] and Stream [15], porting vision applications to GPUs became easier. Even then, the GPU was restricted to only data parallel kernels and some reductions (like sum). General purpose computing remained out of the bounds of most vision practitioners until after the advent of Nvidia's CUDA in 2006 [129, 130]. The applicability of GPUs to accelerate computer vision algorithms improved tremendously because it was now possible to map several classes of irregular computations to the GPU. Classes of vision algorithms parallelized on GPUs include feature detection and description like SIFT [153, 178] and KLT [154], optical flow [159, 177, 183], 3D reconstruction [76] and libraries like OpenCV [77, 6]. More recently, there have been efforts to rent time on commodity clusters (i.e. cloud computing) like Amazon EC2 in order to scale to tens of thousands of cores [3].

Compilers Since there is abundance of concurrency in computer vision applications, vector/SIMD parallelizing compilers have always been considered an important tool for enabling parallelization. The main problem in parallelizing computer vision algorithms is not the difficulty in finding concurrency but the difficulty in mapping the concurrency to parallel hardware. For example, with SSE-enabled single core processors, fine-grained concurrency is easy to exploit (SIMD units in hardware). However on modern multicore machines, concurrency must be mapped to both fine grained and coarse-grained parallelism in hardware. GPUs have different memory structures and code must be tuned to the particular memory hierarchy for achieving the best performance. Optimizing compilers for one platform usually do not transfer well to different hardware architectures.

Optimizing and vectorizing compilers work well for parallelizing computer vision code for microprocessors with a single level of hardware parallelism(e.g. single core with SIMD units like SSE). Vectorizing compiler improvements [5, 103] have led to improved usage of SIMD processors in modern CPUs through efficient parallelization of data parallel loops. Improvements like the polyhedral model has enabled compilers to parallelize even non-data-parallel loops efficiently [25]. Data parallel languages like APL [99] and NESL [92] with their corresponding compilers have also been used for parallelizing computer vision algorithms on machines with vector processors or SIMD units. However, as mentioned before, all these methods work much less efficiently when there are multiple levels of parallelism in hardware (multiple sockets, each socket with multiple cores, each core with SIMD units etc.).

In general, compilers alone have been unable to bridge the performance gap between computer vision applications and parallel hardware with no additional help from programmers in the form of pragmas [131], intrinsics or special instructions [130, 129].

Most computer vision researchers use very high level languages like MATLAB. It is very hard to directly compile such code efficiently as they are hard to analyze statically (dynamic typing, interpreted). Hence, even though the productivity benefits provided by high level languages are high, their performance is usually much inferior compared to languages like C/C++ e.g., Catalytic (Matlab to C converter) [4].

We believe that parallelizing compilers, while useful, are not capable of parallelizing and optimizing computer vision applications on modern hardware architectures without any programmer effort. They are also incapable of performing any of the numerical optimizations that are very crucial for improving performance.

Libraries and Frameworks Providing optimized, parallelized libraries is another way to shield computer vision researchers from the details of parallelization while still providing the benefits. However, the main shortcoming is its inflexibility. That said, there have been many useful libraries and packages for the vision community. We describe a few here.

OpenCV [28] is a library of programming functions for real time computer vision. However, it is not parallelized by default. Projects like OpenVidia [77] and GPUCV

[6] have tried to exploit the existing GPUs in user's workstation without having to manually parallelize existing code based on OpenCV.

A significant portion of Computer vision algorithms utilizes linear algebra which consumes a significant portion of their runtime. The use of optimized Basic Linear Algebra Subroutines (BLAS) and Linear Algebra PACKage (LAPACK) [7] libraries for many different hardware architectures solves a significant problem especially if the problem is dense. Since BLAS and LAPACK are parallelized and optimized (for shared memory architectures), their use provides much better runtimes than rolling your own code for functions like dot products, matrix multiplication, matrix factorization, dense linear solver etc. This approach does not work well for sparse matrix problems, for which specialized solutions are still significantly faster [16, 113]. Similar libraries exist for distributed memory architectures (ScaLAPACK [21]), multicore+GPU systems (MAGMA [95]) etc.

All these libraries simplify the work of optimizing computer vision algorithms, but are not always comprehensive. The efficiency on particular hardware platforms is not portable, i.e. code optimized for one platform does not necessarily perform well on others.

A framework is a software environment in which user customization may only be in harmony with the underlying software architecture. This is a guarantee that as long as application developers code under certain restrictions, the framework/toolchain will be able to optimize it so that the output is scalable, optimized, parallel code. They have had some successes (e.g. MATLAB is fast as long as code is "vectorized"). Failure to follow the restrictions may lead to code that cannot be compiled or code that runs much slower. If the algorithm fits the software architecture that the framework supports, then it can provide a large productivity benefit.

MATLAB provides high productivity but low performance for user written code in general. For some cases like performing matrix-matrix multiplication, solving a dense linear system etc., MATLAB just calls the underlying BLAS/LAPACK libraries and can be efficient. However, in order to get good performance on general user code, it is necessary to write the code that contains data parallelism using arrays. MATLAB is able to auto-vectorize code in these situations and is able to give acceptable performance without sacrificing productivity. The difference in performance between vectorized and non-vectorized MATLAB code can be an order of magnitude or even higher. Specialized frameworks within MATLAB like Jacket or Parallel computing toolkit allow users to run parallel programs on the GPU.

Keutzer et al have produced a pattern language categorizing software patterns present in applications with a view to enable frameworks for parallel processing [104]. Since computer vision algorithms heavily use specific patterns like data parallelism, frameworks targeted towards writing efficient data parallel code are useful for parallelizing many vision applications. Copperhead [45] is one such example, that utilizes an embedded Just In Time (JIT) compiler in Python in order to compile a data parallel subset of Python efficiently to GPUs. It has been successfully used to parallelize

Technique	Productivity	Numerical Optimizations	Memory management in CPU-accelerator systems
Hardware	-	-	++
Compilers	+	-	-
Libraries & frameworks	+	-	-
Our techniques	+	++	++

Table 4.1: Different parallelization techniques and their relative performance on various parameters.

linear solvers in optical flow problems and training machine learning algorithms like Support Vector Machines.

In general, data parallel frameworks are helpful, but not sufficient for parallelizing computer vision since it is impossible to find large computer vision workloads that do not have any non-data-parallel components.

There is no single solution to efficiently parallelize computer vision algorithms. Many aspects still need to be done manually e.g. finding good mapping from algorithm to hardware, managing the disjoint CPU-GPU memory space etc.

Table 4.1 shows each of the techniques discussed before in the context of productivity, numerical optimizations and memory management.

4.3.4 Our approach

We use a strategy of performing manual mapping from algorithm to hardware, along with a framework for memory management. Given that none of the previous attempts have solved the problem of minimizing CPU-GPU memory transfers, we use an automated strategy for mapping CPU-GPU problems that involve data parallel computations. Using pseudo-boolean optimization, we solve the data transfer scheduling problem, leading to significant performance gains [162]. This lets us solve large problems where the GPU memory is not large enough to store all the data required for the application. The transfer of data between the CPU and GPU is handled automatically and this lets the application scale from small, mobile GPUs to large desktop GPUs. This is explained in detail in Chapter 8.

We use a slightly modified algorithm for the eigensolver in image contour detection. The modified version does not allocate all of the temporary data structures in memory at the start of the computation. It uses the fact that in any iteration, the algorithm only needs a small working set even if the total working set is quite large. Through manual memory management, we have been able to run the eigensolver even in cases where there is not enough GPU memory. More details of the approach will be presented in Chapter 7.

In addition to efficient scheduling of memory, we also efficiently hand-coded the mapping of nested concurrency to hierarchical parallelism in applications. Some of the cases where we applied these include image downsampling in optical flow, eigensolver in video segmentation,

scratchpad memory optimizations for support vector machines. All these components are explained in the chapters 6, 7, and 5 respectively.

4.3.5 Results

Better scheduling of data transfers between CPU and GPU reduced the amount of data moved by $1.7 - 7.8\times$, leading to improvements in runtime by a factor of 2. This can be included as part of future compilers or frameworks and encompasses a solution for many different hardware configurations. The eigensolver in the image contour detection algorithm is fully portable across GPUs with different memory sizes. There is no performance penalty when there is sufficient GPU memory. The eigensolver has to do 2 passes on the data if it runs out of memory, which effectively doubles the runtime. However, we believe the ability to run the application on a variety of platforms more than offsets the runtime disadvantage. These aspects are explained in Chapters 7 and 8.

Manual mapping and parallelization alone have led to speedups of $10\times$ for the range of computer vision applications considered. More details on the speedups are explained in the respective chapters.

4.4 Summary

In almost all computer vision applications, we have found that numerical optimizations had huge positive impacts on performance. Our numerical optimizations take advantage of the specific problem context and properties to accelerate applications. This has led to speedups of over $20\times$ in the case of image contour detection and $37\times$ in the case of large displacement optical flow. In order to be taken advantage of the hierarchical parallelism that both CPUs and GPUs expose, we need a good mapping from the algorithmic layer to hardware. In addition, our techniques for manual mapping and managing memory in CPU-GPU systems have led to portability and performance improvements in a range of applications.

Chapter 5

Parallelizing Support Vector Machine

In this chapter, we will discuss the problem, parallelization approaches and optimizations for a machine learning algorithm - Support Vector Machine (SVM). We discuss optimizations for both SVM training and classification on GPU and CPU architectures. We compare our solution against LIBSVM, a standard package for solving SVM problems and show that our approach is much faster and more scalable. We will discuss the algorithmic and code optimizations needed to achieve this performance.

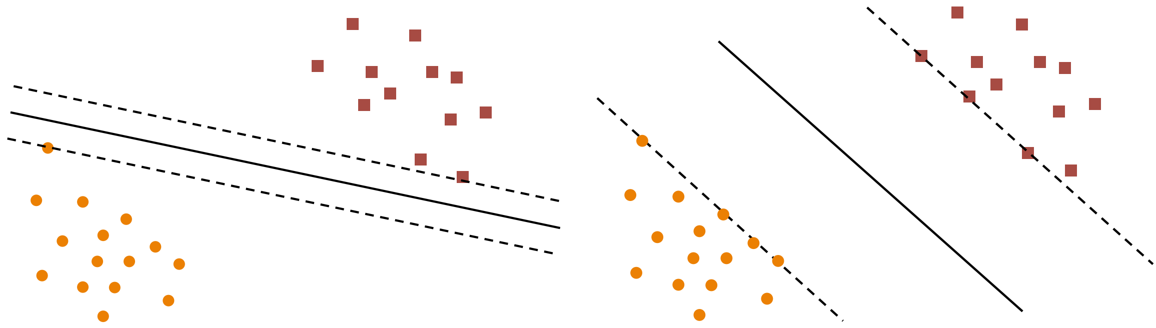
5.1 SVM background

Support Vector Machine (SVM) is a classification and regression algorithm in machine learning that has become widely popular in the last few decades [57]. Thanks to their robust classification performance, SVMs have found use in diverse tasks, such as image recognition, bioinformatics, and text processing. SVMs belong to a class of classification algorithms called "maximum margin classifiers" because they not only try to find a classification boundary between the positive and negative examples, but try to do so with maximum separation between the classes. Figure 5.1 shows an illustration of a separating hyperplane in 2 dimensions and one with the maximum margin. It is clear that a maximum margin separating hyperplane is a better classifier.

We consider the standard two-class soft-margin SVM classification problem (C-SVM), which classifies a given data point $x \in \mathbb{R}^n$ by assigning a label $y \in \{-1, 1\}$. We discuss the problem formulation for both SVM training and classification for the case of C-SVM.

5.1.1 SVM Training

Given a labeled training set consisting of a set of data points $x_i, i \in \{1, ..., l\}$ with their accompanying labels $y_i, i \in \{1, ..., l\}$, the problem of SVM training can be written as the



(a) Separating hyperplane with non-maximum margin (b) Separating hyperplane with maximum margin

Figure 5.1: Examples of separating hyperplanes in 2 dimensions.

following optimization problem:

$$\begin{aligned} \min_{w,\xi,b} \; G(w,\xi) &= \frac{1}{2}w^T w + C\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(w^T\phi(x_i) - b) \geq 1 - \xi_i, \; \forall i \in 1\dots l \\ & \xi_i \geq 0 \end{aligned} \tag{5.1}$$

where $x_i \in \mathbb{R}^n$ is training data point i, $y_i \in \{-1, 1\}$ is the label attached to point x_i. $\phi(x)$ is a linear/non-linear mapping of x from $\mathbb{R}^n$ to $\mathbb{R}^m$ (where m can be infinity) and w is the hyperplane with maximum margin given by $w^T\phi(x) = b$. ξ is an indicator variable denoting separation from hyperplane with $\xi_i = 0$ for correctly classified points outside the margins and $\xi_i > 0$ for all other points. C is a parameter which trades classifier generality for accuracy on the training set. This is the primal form of the optimization problem.

The dual formulation of this optimization problem is a quadratic program (QP):

$$\begin{aligned} \max_{\alpha} \; F(\alpha) &= \sum_{i=1}^{l}\alpha_i - \frac{1}{2}\alpha^T Q\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \; \forall i \in 1\dots l \\ & y^T\alpha = 0 \end{aligned} \tag{5.2}$$

where α_i is a set of weights, one for each training point, which are being optimized to determine the SVM classifier. Q is the kernel matrix i.e. $Q_{ij} = y_i y_j \Phi(x_i, x_j)$, where $\Phi(x_i, x_j)$ is a kernel function which implicitly calculates $\phi(x_i)^T\phi(x_j)$. The optimization problem is usually solved in this dual space. We consider the standard kernel functions shown in table 5.1.

The advantage of using kernel functions is that it gives us the ability to construct very complicated non-linear classification surfaces. These surfaces are hyperplanes in a higher

Table 5.1: Common Kernel Functions for Support Vector Machines

LINEAR	$\Phi(x_i, x_j) = x_i \cdot x_j$
POLYNOMIAL	$\Phi(x_i, x_j; a, r, d) = (ax_i \cdot x_j + r)^d$
GAUSSIAN	$\Phi(x_i, x_j; \gamma) = \exp\{-\gamma \|\|x_i - x_j\|\|^2\}$
SIGMOID	$\Phi(x_i, x_j; a, r) = \tanh(ax_i \cdot x_j + r)$

dimensional space, but their projections on the original space are non-linear.

5.1.2 SVM Classification

The SVM classification problem is as follows: for each data point z which should be classified, compute

$$\hat{z} = \operatorname{sgn}\left\{ b + \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i, z) \right\} \tag{5.3}$$

where $z \in \mathbb{R}^n$ is a point which needs to be classified, $\operatorname{sgn}(x)$ is the sign function, and all other variables remain as previously defined.

From the classification problem definition, it follows immediately that the decision surface is defined by referencing a subset of the training data, or more specifically, those training data points for which the corresponding $\alpha_i > 0$. Such points are called **support vectors**.

Generally, we classify not just one point, but a set of points. We exploit this for better performance, as explained in the later sections.

5.2 Problem

Training Support Vector Machines and using them for classification remains very computationally intensive. Much algorithmic research has been done to accelerate training time, such as Osuna's decomposition approach [133], Platt's Sequential Minimal Optimization (SMO) algorithm [134], Joachims' SVM^{light} [100], which introduced shrinking and kernel caching, and the working set selection heuristics used by LIBSVM [71]. Despite this research, SVM training time is still significant for large datasets. In particular for non-linear kernel functions, the training time for SVMs scales almost quadratically with the number of training points [134]. Linear SVMs can solved much faster, as they scale linearly with the problem size through techniques like [101]. However, training non-linear SVMs for large datasets remains a challenge.

For linear SVMs, SVM classification reduces to a simple matrix-vector multiplication problem. However, for non-linear SVMs, this is not possible in the general case. Classification time is critical because it directly impacts the applicability of the approach. One trains the classifier once, but usually uses it to classify a large number of points. Slow classification time for kernels like the Gaussian kernel is one of the main obstacles to the use of SVM for classification, even though such non-linear kernels can be very accurate.

5.3 Related work

5.3.1 Training

Improving the performance of SVM training and classification has been an active area of research. As mentioned earlier, Osuna et al's decomposition approach [133] has been the most important for solving the optimization problem for SVM training. Osuna et al proved that the special structure of the SVM training problem in the dual form (equation 5.2) means that one could solve a QP on a small, fixed subset of the variables and make progress towards the optimal solution. The algorithm proceeds iteratively; it starts with a subset of variables, optimizes the smaller problem, updates the Karush-Kuhn-Tucker (KKT) conditions for the rest of the variables, chooses a "working set" which is a set of variables with at least one variable whose KKT condition is not satisfied, and repeats the process until convergence. This enables us to solve large SVM training problems using only a small scale QP solver. Sequential Minimal Optimization (SMO) [134] uses the decomposition methodology at the finest level, optimizing on a subset of just 2 variables in a given iteration. This makes the small QP analytically solvable, thereby easing implementation. In addition, Platt showed that it is also as fast as other SVM training implementations. However, none of these approaches took explicit advantage of the parallelism in the problem.

There have been previous attempts to parallelize the SVM training problem. The one most similar to ours is [44], which parallelizes the SMO algorithm on a cluster of computers using MPI. Both our approach and their approach exploit the concurrency inherent in the KKT condition updates as the major source of parallelism. In general, parallelizing highly iterative problems like SVM training using distributed computing is hard because the synchronization costs in a distributed cluster are much higher than on single chip multiprocessors. Even with a 32-processor cluster, the parallel efficiency (= speedup/number of processors) becomes less than 70% due to Amdahl's law effects. Hence, SVM training parallelization is best performed on a single node using parallel hardware like multicore CPUs or GPUs.

Many other approaches for parallelizing SVM training have been presented earlier. The cascade SVM [83] is another proposed method for parallelizing SVM training on clusters. It uses a method of divide and conquer to solve large SVM problems. Cascade SVM was implemented with distributed computing in mind, hence the communication was kept to a minimum. The amount of parallelism is small (10's of nodes) and the amount of concurrency decreases as the problem converges. Zanni et al [185] parallelize the underlying QP solver using parallel gradient projection technique. Work has been done on using a parallel interior point method for solving the SVM training problem [179]. Collobert et al [56] proposes a method where the several smaller SVMs are trained in a parallel fashion and their outputs weighted using an artificial neural network. Ferreira et al [75] implement a gradient based solution for SVM training, which relies on data parallelism in computing the gradient of the objective function for an unconstrained QP optimization at its core. Some of these techniques, for example the training set decomposition approaches like the Cascade SVM are orthogonal to the work we describe, and could be applied to our solver. Bottou et al

[26] give an extensive overview of parallel SVM implementations.

Most of the prior work has only looked at distributed computing (clusters) as the source of parallelism, and not single chip multiprocessors. In terms of implementation, GPUs present a completely different model than clusters, and hence the amount of parallelism exploited, such as the number of threads, granularity of computation per thread, memory access patterns, and data partitioning are very different. We also implement more sophisticated working set selection heuristics.

5.3.2 Classification

SVM classification is also a compute-intensive problem. SVM classification has always been considered embarrassingly parallel i.e. each point that needs to be classified can be processed independent of other points, thus providing as much concurrency as there are points to be classified. However, exploiting only this concurrency can be suboptimal as there is also concurrency to be exploited in calculating the output for a single point. Exploiting only the former concurrency leads to a BLAS2 (matrix-vector product) computation which is limited by the memory bandwidth, whereas exploiting all the concurrency leads to a much more efficient BLAS3 (matrix-matrix product) computation which is limited by the floating point throughput of the hardware. We will show later how exploiting all levels of concurrency improves performance significantly.

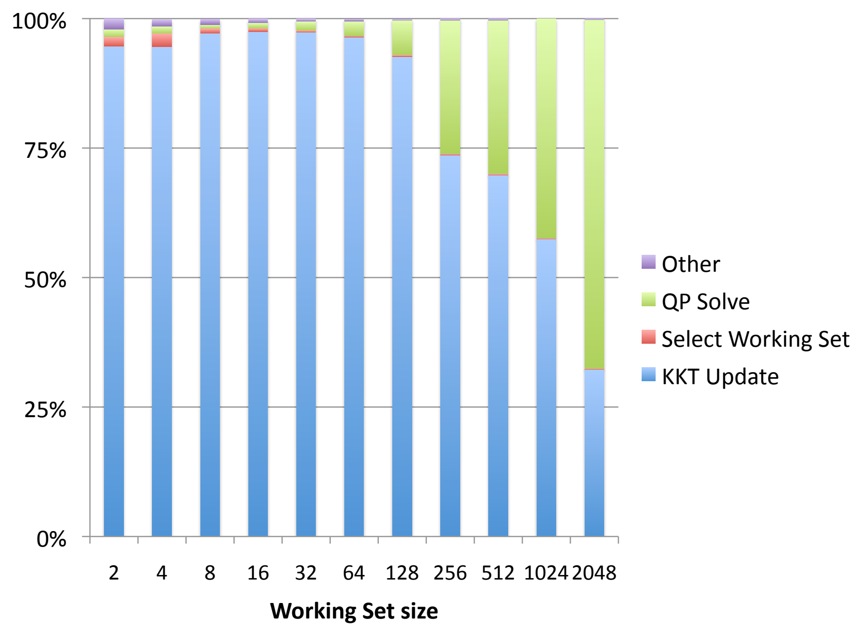
5.4 Our approach

5.4.1 Training

We chose Sequential Minimal Optimization as our SVM training algorithm because of its relative simplicity, yet high performance and robust convergence characteristics. We compare SMO against decomposition based algorithms that differ in the size of their working set i.e. the number of variables in the QP optimized in a single iteration. Figure 5.2 shows the results from the analysis. We used SVM^{light} [100] in order to vary the working set sizes. The implementation is based on Osuna's decomposition algorithm [133]. On an Intel Core2 Duo 2.66 GHz processor, the runtime variations with varying working set sizes were recorded for the face dataset [143].

From the figure 5.2(a), it is clear that solving the SVM training optimization using a large working set is slower than using smaller working sets on the CPU. Interestingly, the runtime does not improve between a working set of size 2 and 64. In terms of the components of the computation, most of the runtime is spent on KKT updates (Figure 5.2(b)). Given this profile, a working set of size 2 (SMO) provides the best tradeoff between simplicity and efficiency of implementation. For a working set of size 2, the solution to the optimization problem can be calculated analytically as we are only minimizing a convex quadratic function in 2 dimensions subject to a simple set of constraints. A larger working set size would require us to implement a more general quadratic optimization routine that, while being much more complicated to implement, would not necessarily improve runtime

(a) Runtime vs working set size



(b) Task runtime distribution vs working set size

Figure 5.2: Effect of working set size on SVM training.

significantly. Most of the runtime is spent in the KKT updates and these updates will be parallelized in either case.

The SMO algorithm is a specialized optimization approach for the SVM quadratic program. It takes advantage of the sparse nature of the support vector problem and the simple nature of the constraints in the SVM QP to reduce each optimization step to its minimum form: updating two α_i weights. The bulk of the computation is then to update the Karush-Kuhn-Tucker optimality conditions for the remaining set of weights and then find the next two weights to update in the next iteration. This is repeated until convergence. We state this algorithm briefly, for reference purposes.

Algorithm 1 Sequential Minimal Optimization

Input: training data x_i, labels y_i, $\forall i \in \{1..l\}$
Initialize: $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1..l\}$,
Initialize: b_{high}, b_{low}, i_{high}, i_{low}
Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
repeat
  Update f_i, $\forall i \in \{1..l\}$
  Compute: b_{high}, i_{high}, b_{low}, i_{low}
  Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
until $b_{low} \leq b_{high} + 2\tau$

For the first iteration, we initialize $b_{high} = -1$, $i_{high} = \min\{i : y_i = 1\}$, $b_{low} = 1$, and $i_{low} = \min\{i : y_i = -1\}$.

During each iteration, once we have chosen i_{high} and i_{low}, we take the optimization step:

$$\alpha'_{i_{low}} = \alpha_{i_{low}} + y_{i_{low}}(b_{high} - b_{low})/\eta \quad (5.4)$$

$$\alpha'_{i_{high}} = \alpha_{i_{high}} + y_{i_{low}} y_{i_{high}}(\alpha_{i_{low}} - \alpha'_{i_{low}}) \quad (5.5)$$

where $\eta = \Phi(x_{i_{high}}, x_{i_{high}}) + \Phi(x_{i_{low}}, x_{i_{low}}) - 2\Phi(x_{i_{high}}, x_{i_{low}})$. To ensure that this update is feasible, $\alpha'_{i_{low}}$ and $\alpha'_{i_{high}}$ must be clipped to the valid range $0 \leq \alpha_i \leq C$. This step is referred to as "QP" in Figure 5.2(b).

The optimality conditions can be tracked through the vector $f_i = \sum_{j=1}^{l} \alpha_j y_j \Phi(x_i, x_j) - y_i$, which is constructed iteratively as the algorithm progresses. After each α update, f is updated for all points. This is one of the major computational steps of the algorithm, and is done as follows:

$$\begin{aligned} f'_i = f_i &+ (\alpha'_{i_{high}} - \alpha_{i_{high}}) y_{i_{high}} \Phi(x_{i_{high}}, x_i) \\ &+ (\alpha'_{i_{low}} - \alpha_{i_{low}}) y_{i_{low}} \Phi(x_{i_{low}}, x_i) \end{aligned} \quad (5.6)$$

This step is referred to as "KKT update" in Figure 5.2(b).

In order to evaluate the optimality conditions, we define index sets:

$$I_{high} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \\ \cup \{i : y_i < 0, \alpha_i = C\} \tag{5.7}$$

$$I_{low} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \\ \cup \{i : y_i < 0, \alpha_i = 0\} \tag{5.8}$$

Because of the approximate nature of the solution process, these index sets are computed to within a tolerance ϵ, e.g. $\{i : 0 < \alpha < C\}$ becomes $\{i : \epsilon < \alpha_i < (C - \epsilon)\}$.

We can then measure the optimality of our current solution by checking the optimality gap, which is the difference between $b_{high} = \min\{f_i : i \in I_{high}\}$, and $b_{low} = \max\{f_i : i \in I_{low}\}$. When $b_{low} \leq b_{high} + 2\tau$, we terminate the algorithm.

Working Set Selection

During each iteration, we need to choose i_{high} and i_{low}, which index the α weights which will be changed in the following optimization step. The first order heuristic from [102] chooses them as follows:

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \tag{5.9}$$

$$i_{low} = \arg\max\{f_i : i \in I_{low}\} \tag{5.10}$$

The second order heuristic from [71] chooses i_{high} and i_{low} to optimize the unconstrained SVM functional. An optimal approach to this problem would require examining $\binom{l}{2}$ candidate pairs, which would be computationally intractable. To simplify the problem, i_{high} is instead chosen as in the first order heuristic, and then i_{low} is chosen to maximally improve the objective function while still guaranteeing progress towards the constrained optimum from problem (5.2). More explicitly:

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \tag{5.11}$$

$$i_{low} = \arg\max\{\Delta F_i(\alpha) : i \in I_{low}, f_{i_{high}} < f_i\} \tag{5.12}$$

After choosing i_{high}, we compute for all $i \in \{1..l\}$

$$\beta_i = f_{i_{high}} - f_i \tag{5.13}$$

$$\eta_i = \Phi(x_{i_{high}}, x_{i_{high}}) + \Phi(x_i, x_i) - 2\Phi(x_{i_{high}}, x_i) \tag{5.14}$$

$$\Delta F_i(\alpha) = \beta_i^2 / \eta_i \tag{5.15}$$

We then find the maximum ΔF_i over all valid points ($i \in I_{low}$) for which we are guaranteed to progress towards the constrained optimum ($f_{i_{high}} < f_i$).

The second order heuristic utilizes more information from the SVM training problem, and so it generally reduces the number of iterations necessary during the solution process. However, it is more costly to compute. In our GPU implementation, the geometric mean

of iteration time over our benchmark set using the second order heuristic increased by 1.9× compared to the first order heuristic. On some benchmarks, the total number of iterations decreased sufficiently to provide a significant speedup overall, but on others, the second order heuristic is counterproductive for our GPU implementation. [48] describes an adaptive heuristic that can choose between the two selection heuristics dynamically in order to achieve speedups on all benchmark problems.

Implementation

Since GPUs need a large number of threads to efficiently exploit parallelism, we create one thread for every data point in the training set. For the first phase of the computation, each thread computes f' from equation 5.6. We then apply a working set selection heuristic to select the next points which will be optimized.

The SMO algorithm is a good fit for the GPU architecture for the following reasons:

1. From figure 5.2(b), it is clear that for small working set sizes, most of the time is spent in updating the KKT conditions. This fits the GPU well because the individual updates are independent and the memory accesses involved are regular.

2. The amount of shared state between the threads is small - only two points that are involved in the QP have to be shared. This is important as the amount of shared memory available is limited on the GPU.

3. By storing the data in both row major and column major formats, all memory accesses can be coalesced, leading to good performance.

If we only stored the training data matrix (size is number of points × dimensions) in column major format, then the memory loads for the 2 shared data points (for loading them into shared memory) become uncoalesced. Uncoalesced access occurs with column major format during the QP update step also where the dot product of two vectors is calculated. To alleviate this, we store both the row major and column major versions of the data and read from the appropriate version.

The Map Reduce pattern has been shown to be useful for many machine learning applications [55], and is a natural fit for our SVM training algorithm. Exploiting parallelism in a map reduce pattern is relatively straightforward. The map stage is composed of independent computations and can be directly parallelized. The reduce stage, by its very nature, requires communication and synchronization. This makes implementation of the reduce stage complicated and can lead to performance bottlenecks, especially on very parallel architectures which have limited synchronization and communication abilities between threads, such as the GPU. It can be shown that choosing a single reduction method a priori, that is optimal for a particular data set size can yield performance that is up to 60× worse than the optimal method for a different data set size [49]. We created a simple Map Reduce framework that can help improve both productivity and efficiency while implementing the SVM training application. Using autotuning [18], parameters such as the amount of local memory used,

number of threads in a thread block, algorithm used for reduction etc. were optimized for different data sizes.

For the first order heuristic, the computation of f_i' for all points is the map function, and the search for b_{low}, b_{high}, i_{low} and i_{high} is the reduction operation. For the second order heuristic, there are two Map Reduce stages: one to compute f_i', b_{high} and i_{high}, and another where the map stage computes ΔF_i for all points, while the reduce stage computes b_{low} and i_{low}.

Because the CUDA programming model has strict limitations on synchronization and communication between thread blocks, we organize the reductions in two phases. The first phase does the map computation, as well as a local reduce within a thread block. The second phase finishes the global reduction. Each phase of this process is implemented as a separate call to the GPU.

In addition to parallelization, other optimizations for SVM training such as caching and using local stores are performed. Caching rows of the kernel matrix $K_{i,j} = \Phi(x_i, x_j)$ reduces the amount of computation needed if the row is already present in a precomputed form. This is an optional optimization and does not change the correctness of the algorithm, for example when the training data is large and there is no memory available for cache allocation. We also use the local stores in the GPU to load the vectors corresponding to the two points whose α's were changed (if the corresponding rows do not exist in the cache). When the vectors are too large to fit the GPU's shared memory, this optimization gets disabled and the algorithm proceeds without using the local store. Each thread is in charge of computing two rows of the kernel matrix corresponding to the two points.

As mentioned in Chapter 4, GPUs have limited memory. In order to enable SVM training to completely execute on the GPU, the only assumption we make is that the training data is small enough to fit in GPU memory. If the data is too large to fit in GPU memory, then techniques like Cascade SVM [83] can be used to solve smaller SVMs whose training set fits in the memory and then combine the results.

5.4.2 Classification

As mentioned earlier, we look at the SVM classification as a BLAS3 computation in order to exploit all the concurrency in the problem. We make use of vendor supplied Basic Linear Algebra Subroutines - specifically, the matrix-matrix multiplication routine (SGEMM), which calculates $C = \alpha AB + \beta C$, for matrices A, B, and C and scalars α and β. For the Linear, Polynomial, and Sigmoid kernels, calculating the classification value involves finding the dot product between all test points and the support vectors, which is done through SGEMM. For the Gaussian kernel, we use the simple identity $||x - y||^2 = x \cdot x + y \cdot y - 2x \cdot y$ to recast the computation into a matrix-matrix multiplication where $x \cdot y$ can be calculated using SGEMM. The Gaussian kernel requires computing $D_{ij} = -\gamma||z_i - x_j||^2 = 2\gamma(z_i \cdot x_j) - \gamma(z_i \cdot z_i + x_j \cdot x_j)$, for a set of unknown points z and the set of support vectors x. We then apply a map reduce computation to combine the computed D values to get the final result of the SVM classifier.

Continuing the Gaussian example, we then exponentiate D_{ij} element wise and multiply

each column of the resulting matrix by the appropriate $y_j\alpha_j$. We then sum the rows of the matrix and add b to obtain the final classification for each data point as given by equation (5.3). Other kernels require similar Map Reduce calculations to finish the classification.

We also discuss the optimizations that were performed on the CPU-based SVM classifier. LIBSVM classifies data points serially. This effectively precludes data locality optimizations and produces significant slowdown. It also represents data in a sparse format, which can cause overhead as well.

To optimize the CPU classifier, we performed the following:

1. We changed the data structure used for storing the support vectors and test vectors from a sparse indexed set to a dense matrix.

2. To maximize performance, we used BLAS routines from the Intel Math Kernel Library to perform operations similar to those mentioned earlier.

3. Wherever possible, loops were parallelized (4-way for the quad-core machine) using OpenMP.

The disadvantage of using SGEMM instead of performing SVM classification one-at-a-time is increased memory footprint. If classification is performed one point at a time, then we only need to store the kernels $\Phi(z, x_i)$ with respect to all the support vectors x_i. However, SGEMM requires storing a matrix of size $p \times l$ where p is the number of test vectors and l is the number of support vectors. If the number of test vectors is large, this matrix can exceed the amount of GPU memory present in the system. The solution is to partition the test vectors so that we process only as many as can fit the GPU memory.

5.5 Results

The SMO implementation on the GPU is compared with LIBSVM, as LIBSVM uses Sequential Minimal Optimization for SVM training. We used the Gaussian kernel in all of our experiments, since it is widely employed. The CPU used for measurement is an Intel Nehalem Core i7 920 (quad core). The GPU used is an Nvidia Tesla C2050.

5.5.1 Training

We tested the performance of our GPU implementation versus LIBSVM on the datasets detailed in tables 5.2 and 5.3.

The sizes of the datasets are given in table 5.3. References for the datasets used and the (C, γ) values used for SVM training are provided in table 5.2.

We ran LIBSVM on an Intel Core i7 920 Nehalem processor, and gave LIBSVM a cache size of 1 GB, which is larger than our GPU implementation was allowed. CPU-GPU communication overhead was included in the solver runtime, but file I/O time was excluded for both our solver and LIBSVM. Table 5.4 shows results from our solver. File I/O varies from 1.2 seconds for USPS to about 12 seconds for Forest dataset. The CPU - GPU data transfer

Table 5.2: SVM Datasets - References and training parameters

Dataset	C	γ
Adult [12]	100	0.5
Web [134]	64	7.8125
MNIST [110]	10	0.125
USPS [97]	10	2^{-8}
Forest [12]	10	0.125
Face [143]	10	0.125

Table 5.3: Dataset Sizes

Dataset	# Points	# Dimensions
Adult	32,561	123
Web	49,749	300
MNIST	60,000	784
USPS	7,291	256
Forest	581,012	54
Face	6,977	381

overhead was also very low. The time taken to transfer the training data to the GPU and copy the results back was less than 0.6 seconds, even for our largest dataset (Forest).

Since any two solvers give slightly different answers on the same optimization problem, due to the inexact nature of the optimization process, we show the number of support vectors returned by the two solvers as well as how close the final values of b were for the GPU solver and LIBSVM, which were both run with the same tolerance value $\tau = 0.001$. As shown in the table, the deviation in number of support vectors between the two solvers is less than 2%, and the deviation in the offset b is always less than 0.1%. Our solver provides equivalent accuracy to the LIBSVM solver, which will be shown again in the classification results section.

Table 5.4: SVM Training Convergence Comparison

Dataset	Number of SVs		Difference in b (%)
	GPU Adaptive	LIBSVM	
Adult	18,674	19,058	-0.004
Web	35,220	35,232	-0.01
MNIST	43,730	43,756	-0.04
USPS	684	684	0.07
Forest	270,351	270,311	0.07
Face	3,313	3,322	0.01

Table 5.5: SVM Training Results

Dataset	GPU 1st Order		GPU 2nd Order		LIBSVM		Speedup (×)	
	Iter.	Time (s)	Iter.	Time (s)	Iter.	Time (s)	1st Order	2nd Order
Adult	116211	10.42	39994	8.06	43735	448.29	43.0	55.6
Web	82232	62.96	81498	125.91	85299	4557.96	72.4	36.2
MNIST	67932	172.28	67812	310.28	76385	13146.58	76.3	42.4
USPS	6356	0.464	3715	0.495	4614	2.175	4.7	4.4
Forest	2055409	1618.41	235046	652.85	277353	22122.74	13.7	33.9
Faces	6009	0.906	4844	1.16	5342	11.59	12.8	10.0

Table 5.5 contains performance results for the two solvers. We see speedups in all cases from 4.4× to 76.3×. For reference, we have shown results for the solvers using both heuristics statically.

5.5.2 Classification

Results for our classifier are presented in table 5.7. We achieve $4 - 372\times$ speedup over LIBSVM on the datasets shown. As with the solver, file I/O times were excluded from overall runtime. File I/O times vary from 0.3 seconds for USPS dataset to about 17 seconds for MNIST dataset.

For some insight into the CPU-based classifier results, we note that the optimized CPU classifier performs best on problems with a large number of input dimensions, which helps make the SVM classification process compute bound. For problems with a small number of input dimensions, the SVM classification process is memory bound, meaning it is limited by memory bandwidth. Since the GPU has much higher memory bandwidth, as noted in Chapter 4, it is even more attractive for such problems.

These optimizations improved the classification speed on the CPU by a factor of $9-47\times$. The speedup numbers for the different datasets are shown in table 5.7. It should be noted that the GPU version is better than the optimized CPU versions by a factor of $0.4 - 11\times$. For very small problems like USPS, GPUSVM shows a slight slowdown compared to the optimized CPU version. This is because of overheads involved in moving data over to the GPU compared to the amount of computation needed for this problem. On all other problems, GPUSVM has a significant performance improvement compared to the CPU version.

We tested the combined SVM training and classification process for accuracy by using the SVM classifier produced by the GPU solver with the GPU classification routine, and used the SVM classifier provided by LIBSVM's solver to perform classification with LIBSVM. Thus, the accuracy of the classification results presented in table 5.6 reflect the overall accuracy of the GPU solver and GPU classifier system. The results are identical, which shows that our GPU based SVM system is as accurate as traditional CPU based methods.

Table 5.6: Accuracy of GPU SVM classification vs. LIBSVM

Dataset	GPU Accuracy	LIBSVM Accuracy
Adult	6619/8000	6619/8000
Web	3920/4000	3920/4000
MNIST	2365/2500	2365/2500
USPS	1948/2007	1948/2007
Face	23665/24045	23665/24045

Table 5.7: Performance of GPU SVM classifier compared to LIBSVM and Optimized CPU classifier

Dataset	LIBSVM	CPU Optimized classifier		GPU Classifier		
	Time (s)	Time (s)	Speedup (×) compared to LIBSVM	Time (s)	Speedup (×) compared to LIBSVM	Speedup (×) compared to CPU optimized code
Adult	57.45	2.06	27.3	0.3435	163.5	6
Web	105.55	4.94	27.3	0.4370	241.5	11.3
MNIST	211.63	4.43	21.3	0.5676	372.9	7.8
USPS	0.77	0.080	9.7	0.1884	4.1	0.4
Face	74.13	2.21	33.47	0.3470	213.6	6.4

5.6 Summary

We discussed how support vector machine training and classification algorithms can be parallelized and optimized on GPUs and multicore CPUs. The runtime improvement of Support Vector Machines has been possible because of numerical optimizations, efficient implementations and good memory management. Our contributions are as follows:

1. Showing that Sequential Minimal Optimization is the right SVM training algorithm for GPUs because of its inherent concurrency and ease of implementation.

2. Performing reformulations to change SVM classification from a BLAS2 computation to a BLAS3 computation, thereby increasing the compute efficiency on all parallel platforms.

3. Performing memory management optimizations to fit data structures used for SVM training and classification in local stores and in GPU memory respectively without affecting the accuracy.

The transformation of the SVM classification from a BLAS2 to a BLAS3 problem improved the speed by a factor of $9-47\times$ on CPUs and $4-372\times$ on GPUs compared to LIBSVM. Improving the SVM training implementation using multiple copies of data in different formats, kernel caching using local memories on the GPU, proper address alignment,

using dense instead of sparse data structures etc. lead to a highly competitive and scalable implementation with a $4 - 76\times$ speedup. Optimizations to make large SVM classification problems fit in GPU memory have made the implementation more portable by adapting to different GPUs and problem sizes.

The overall contribution is showing that computationally heavy algorithms in machine learning can be run on GPUs without compromising accuracy even with restrictions in programming model and memory capacity (DRAM and local store). Mapping these computations to hardware using frameworks based on patterns (such as Map Reduce) is the one of the main ways to improve productivity and efficiency while moving to parallel platforms. Data parallel frameworks built on these concepts such as Copperhead [45] have been shown to improve productivity for the SVM training problem.

Chapter 6 describes the parallelization and algorithmic exploration for mapping optical flow & point tracking to GPUs.

Chapter 6

Optical Flow and Point Tracking

In this chapter we discuss optical flow and tracking, techniques that are needed for motion analysis in video sequences. When analyzing video data, motion is probably the most important cue, and the most common techniques to exploit this information are difference images, optical flow, and point tracking. Difference images are useful when the camera is static, most of the scene is stationary and there is limited motion. In that case, taking the difference of two images directly gives us the location and magnitude of motion. Optical flow calculates a displacement map or flow field for every pixel in every frame of the video. There are no restrictions on camera or scene motion and is usually calculated between successive frames of a video sequence. Hence, it is extremely localized in time i.e. optical flow is dense spatially, but sparse temporally. Point tracking tries to track points over several frames, thus is temporally denser compared to optical flow. It is usually sparse spatially i.e. only a few pixels are tracked in each frame. We will focus on only optical flow and point tracking as they allow us to extract rich information that is not restricted to static cameras. The goal here is to enable accurate motion tracking for a large set of points in the video in as close to real time as possible. The quality of the estimated flow field or a set of point trajectories is very important as small differences in the quality of the input features can make a high level approach succeed or fail. To ensure accuracy, many methods only track a sparse set of points; however, dense motion tracking enables us to extract information at a much finer granularity compared to sparse feature correspondences. Hence, one wants to use the most recent motion estimation technique providing the most reliable motion features for a specific task. For dense and accurate tracking there are usually computational restrictions. Video data processing requires far more resources than the analysis of static images, as the amount of raw input data is significantly larger. For example, video captured at 1080p resolution at 30 frames/sec produces about 11 GB of raw data per minute.This often hinders the use of high-quality motion estimation methods, which are usually quite slow [147] and require expensive computer clusters to run experiments efficiently. For this reason, ways to significantly speedup such methods on commodity hardware are an important contribution as they enable more efficient research in fields that build upon motion features. Fast implementations of the KLT tracker and optical flow [184, 183] are examples that have certainly pushed research.

In this chapter, we present a fast parallel implementation of large displacement optical flow (LDOF) [36], a recent variational optical flow method that can deal with faster motion than previous optical flow techniques. The numerical schemes used in [36] and most variational methods are based on a coarse-to-fine warping scheme, where each level provides an update by solving a nonlinear system given by the Euler-Lagrange equations followed by fixed point iterations and a linear solver, as described in [34]. However, the relaxation techniques used in the linear solver that work best for serial processors are not efficient on parallel processors. We investigate alternative solvers that run well on parallel hardware, in particular red-black relaxations and the conjugate gradient method. We show that the conjugate gradient method is faster than red-black relaxations, especially on larger images. We also prove that the fixed point matrix is positive definite, thus guaranteeing the convergence of the conjugate gradient algorithm. We obtain a speedup of about 37×, which allows us to compute high quality LDOF for 640×480 images in 1.8 seconds compared to a sequential C++ implementation. Extrapolating the current progress in technology, the same code will even run in real-time in only a few years. While additional speedup can often be obtained at the cost of lower quality, we ensured in our implementation that the quality of the original method is preserved.

We also propose a method for dense point tracking by building upon the fast implementation of large displacement optical flow. Point trajectories are needed whenever an approach builds on long term motion analysis. The dominant method used for this task is the KLT tracker [150], which is a sparse technique that only tracks a very small number of designated feature points. While for many tasks like camera calibration such sparse point trajectories are fully sufficient, other tasks like motion segmentation or structure-from-motion would potentially benefit from higher densities. In [147] and [146], a method for point tracking based on dense variational optical flow has been suggested. The method proposed in [147] is computationally very expensive and impractical to use on large datasets without acceleration. The point tracking we propose uses a similar technique, as points are propagated by means of the optical flow field; however, we do not build upon another energy minimization procedure that detects occluded points mainly by appearance, but do the occlusion reasoning by a forward-backward consistency check of the optical flow. In a quantitative comparison on some sequences from [117], where close to ground truth optical flow has been established by manually annotating the objects in the scene, we show that we can establish much denser point trajectories with better quality than the KLT tracker. At the same time, our method is more accurate and runs an order of magnitude faster than the technique in [146].

6.1 Background

As mentioned in Chapter 4, optical flow computations rely on a brightness constancy assumption along with a constraint on flow smoothness. Most of the commonly used optical flow techniques lack the ability to handle large displacements of small objects like limbs or balls in sports videos. This is due to the fact that they rely on coarse-to-fine refinement in order to solve the optimization problem and avoid getting stuck in a local minimum. This

is based on the assumption that the motion is small at all scales. However, this is not valid for fast moving small objects. Under this refinement the objects are too small to be seen at the scale in which their motion is small, whereas at the scale at which the objects are visible, the small motion assumption is not valid.

Large displacement optical flow (LDOF) is a variational technique that integrates discrete point matches, namely the midpoints of regions, into the continuous energy formulation and optimizes this energy by a coarse-to-fine scheme to estimate large displacements also for small scale structures [32]. As pointed out in [36], region matching can be replaced with matching other features like densely sampled histograms of oriented gradients (HOG) [61]. These simpler features allow us to implement both the variational solver and the discrete matching efficiently on the GPU.

The considered energy functional that is minimized reads:

$$\begin{aligned} E(\mathbf{w}) \;=\; & \int_\Omega \Psi_1\big(|I_2(\mathbf{x}+\mathbf{w}(\mathbf{x})) - I_1(\mathbf{x})|^2\big)d\mathbf{x} \\ & +\gamma \int_\Omega \Psi_2\big(|\nabla I_2(\mathbf{x}+\mathbf{w}(\mathbf{x})) - \nabla I_1(\mathbf{x})|^2\big)d\mathbf{x} + \alpha \int_\Omega \Psi_S\big(|\nabla u(\mathbf{x})|^2 + |\nabla v(\mathbf{x})|^2\big)d\mathbf{x} \\ & +\beta \int_\Omega \delta(\mathbf{x})\ \rho(\mathbf{x})\ \Psi_3(|\mathbf{w}(\mathbf{x}) - \mathbf{w_1}(\mathbf{x})|^2)d\mathbf{x} + \int_\Omega \delta(\mathbf{x})\ |\mathbf{f_2}(\mathbf{x}+\mathbf{w_1}(\mathbf{x})) - \mathbf{f_1}(\mathbf{x})|^2 d\mathbf{x} \end{aligned} \tag{6.1}$$

where $\mathbf{w} = (u\ \ v)^T$ and $\Psi_*(s^2)$ is a general penalizer function with its derivative $\Psi'_*(s^2) > 0$. A popular choice in the literature is $\Psi_*(s^2) = \sqrt{s^2 + \epsilon^2}$ [36]. The first term in the energy functional is the brightness constancy term. The second is brightness gradient constancy which is used to avoid problems with global illumination changes. The third term is the regularization for keeping the flow smoothly varying. The last two terms correspond to large displacement matching. $\rho(\mathbf{x})$ corresponds to the confidence of a feature match and $\mathbf{f_1}(\mathbf{x})$ & $\mathbf{f_2}(\mathbf{x})$ are the feature vectors at $\mathbf{x}$ in the two frames. $\mathbf{w_1}(\mathbf{x})$ is the matching position in frame 2 for the point $\mathbf{x}$ in frame 1.

We minimize (6.1) by writing the Euler-Lagrange equations and solving them through a coarse-to-fine scheme with fixed point iterations. This results in a sequence of linear systems to be solved, where each pixel corresponds to two coupled equations in the linear system:

$$\begin{aligned} & (\Psi'_1 {I_x^k}^2 + \gamma\Psi'_2({I_{xx}^k}^2 + {I_{xy}^k}^2) + \beta\rho\Psi'_3)du^{k,l+1} \\ & + (\Psi'_2 I_x^k I_y^k + \gamma\Psi'_2(I_{xx}^k I_{xy}^k + I_{xy}^k I_{yy}^k))dv^{k,l+1} \\ & - \alpha\ \mathrm{div}(\Psi'_S \nabla(u^k + du^{k,l+1})) \\ & = -\Psi'_1(I_x^k I_z^k) + \gamma\Psi'_2(I_{xx}^k I_{xz}^k + I_{xy}^k I_{yz}^k) - \beta\rho\Psi'_3(u^k - u_1) \end{aligned} \tag{6.2}$$

$$
\begin{aligned}
&(\Psi_1' I_y^{k2} + \gamma\Psi_2'(I_{yy}^{k\ 2} + I_{xy}^{k\ 2}) + \beta\rho\Psi_3')dv^{k,l+1} \\
&+ (\Psi_2' I_x^k I_y^k + \gamma\Psi_2'(I_{xx}^k I_{xy}^k + I_{xy}^k I_{yy}^k))du^{k,l+1} \\
&- \alpha\ \mathrm{div}(\Psi_S' \nabla(v^k + dv^{k,l+1})) \\
&= -\Psi_1'(I_y^k I_z^k) + \gamma\Psi_2'(I_{yy}^k I_{yz}^k + I_{xy}^k I_{xz}^k) - \beta\rho\Psi_3'(v^k - v_1)
\end{aligned}
$$

For details on the derivation of these equation we refer to [36]. The overall structure of the solver is shown in Figure 6.1.

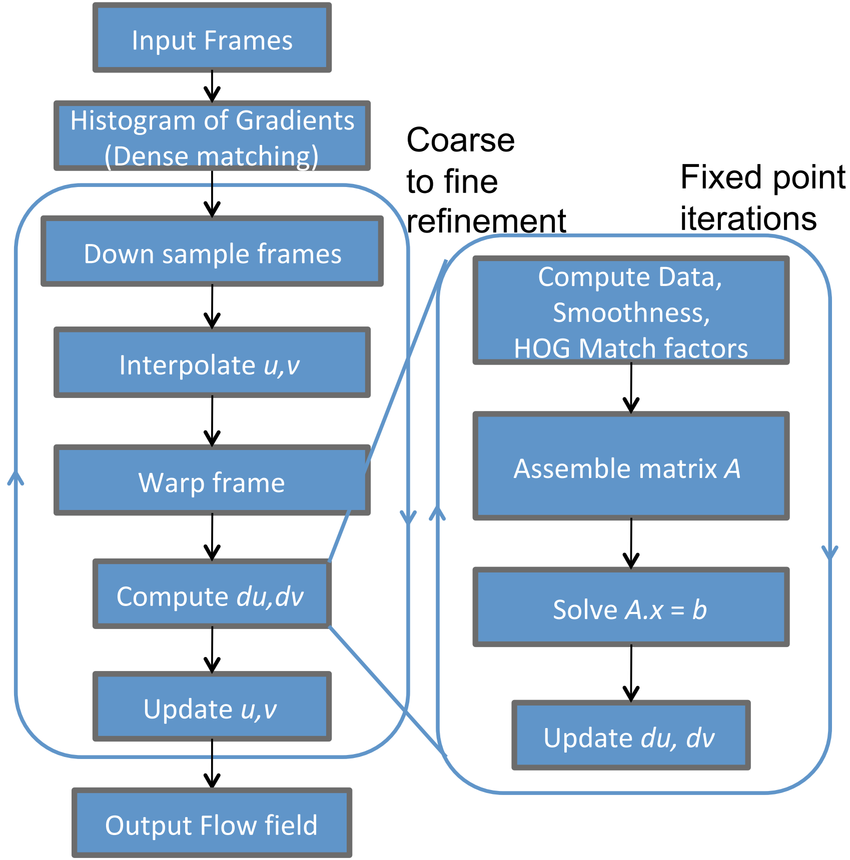


Figure 6.1: High level structure of the large displacement optical flow solver.

6.2 Problem

Optical flow is a much needed computation for performing a variety of video content analysis applications. However, its running time on even 640 × 480 sized frames is unacceptably high. Given the amount of video data that is being generated, faster than real time optical flow is critical for performing large scale video analysis. For example, youtube has more than 83 million videos (average 3 minutes long) and adds more than 24 hours

of video data every minute [182]. Performing optical flow on this database on a cluster of 36000 nodes with current serial optical flow implementations will take about 6 years, which is clearly not practical. In addition, optical flow is only a feature that is used by other algorithms downstream in the processing pipeline. In this scenario, techniques that improve the runtime of optical flow algorithms are an important contribution.

Dense point tracking is important for long term video motion analysis. Point tracks have been used for camera calibration, video object segmentation, creating video annotations [80] etc. The biggest bottleneck in many of these algorithms is computing these point tracks. Techniques that try to extract high quality point tracks like [147] take an inordinate amount of time in processing (more than 100 seconds per frame, excluding optical flow). Speeding up this computation is critical for wide applicability of point tracking in video applications. At the algorithmic level, point tracking requires integrating optical flow over a sequence of frames. Our hypothesis is that high quality point tracking just needs better optical flow, and not better integration techniques. Thus a large amount of computation that goes into clever integration techniques such as [147] can be eliminated and point tracking performed efficiently.

6.3 Related work

Finding efficient solutions to variational optical flow problems has been an active area of research. On serial hardware, multi-grid solvers based on Gauss-Seidel have been proposed in [39]. A GPU implementation of the formulation in [39] has been proposed using Jacobi solvers [85]. Compared to [85], our implementation handles large displacements through dense descriptor matching. The descriptor matching in large displacement optical flow induces high additional costs apart from the linear solver. Such extensions enable us to handle fast motion well [32], [36]. A multi-grid red-black relaxation has been suggested in a parallel implementation of the linear CLG (Combined Local-Global) method [88]. Very efficient parallel GPU implementations of other variational optical flow models have been proposed in [183, 174, 177]. These approaches follow a different numerical scheme by directly solving the nonlinear system based on dual formulations of the original problem rather than running fixed point iterations with a linear solver.

The conjugate gradient algorithm is a popular solver for convex problems and has been used for optical flow problems with convex quadratic optimization [109]. In order to theoretically justify the use of conjugate gradients, we prove that the system matrix of general variational optical flow methods is positive semi-definite (and non-singular) and thus the conjugate gradient solver is guaranteed to converge. It was previously proven that the Horn-Schunck matrix is positive definite [125]. Our proof is more general and applicable to most variational formulations such as [39], [34], [40] and [32].

The most popular point tracker is the Kanade-Lucas-Tomasi (KLT) tracker [150], which constructs an image pyramid, chooses points that have sufficient structure and tracks them across frames. New features are periodically detected to make up for the loss of features because of occlusions and tracking errors. This is generally considered to be fast and accurate, but it tracks only a few points. Efficient GPU implementations of the KLT tracker

have been released in [154] and [184]. While the KLT algorithm itself is quite old, the implementation in [184] compensates for changes in camera exposure to make it more robust. Non-local point trackers that use global information have also been proposed [19].

The more advanced point tracker in [147] and [146] tracks points by building on top of a variational technique. This comes with high computational costs. It takes more than 100 seconds to track points between a pair of 720×480 frames. Moreover, this technique cannot deal with large displacements of small structures like limbs, and it has never been shown whether tracking based on variational flow actually performs better than the classic KLT tracker.

6.4 Our approach

The biggest challenges in parallelizing optical flow involve optimizing the numerical schemes used in the energy minimization problem. In addition to efficient parallelization of all the computations in the optical flow solver, better numerical techniques are essential for good performance. Section 6.5 shows the results of using better numerical techniques on both CPUs and GPUs.

A parallel implementation of the descriptor matching is relatively straightforward since several points are being searched for in parallel without any dependencies between them. It is important, however, to take advantage of coalesced memory accesses (vector loads/stores) in order to maximize the performance of the GPU. In the rest of the section, we will focus on the parallel implementation of the variational solver that considers these point correspondences.

As shown in Section 6.1, we have to solve a linear system of equations in the inner loop of the optical flow solver. From symmetry considerations, the problem discretization usually produces a symmetric block pentadiagonal matrix with 2×2 blocks (for a 5-point Laplacian stencil). From equation (6.2), it is clear that only the diagonal blocks are dense, while the off-diagonal blocks are diagonal matrices. In fact, for the isotropic functionals we consider here, they are scaled identity matrices.

6.4.1 Positive semi-definiteness of the fixed point matrix.

We prove that the fixed point matrix is symmetric positive semi-definite because (a) the diagonal blocks are positive definite and (b) the matrix is block diagonally dominant [73]. An interesting takeaway from the proof is that it is not restricted to convex penalty functions Ψ_*. The only restriction on Ψ_* is that it should be increasing. Moreover, the proof technique generalizes to most variational optical flow methods, e.g. [34], [39],[32] and [40]. Details of the proof are provided below.

Proof of positive definiteness

The sparsity structure of the matrix derived from the system of equations (6.2) is shown in Figure 6.2. The connectivity looks similar to a 2D Laplacian stencil matrix.

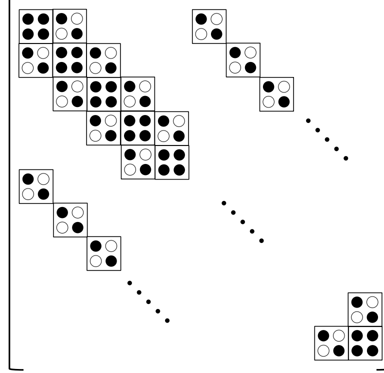


Figure 6.2: Sparse matrix showing the decomposition into dense 2 × 2 blocks. Filled dots represent non-zeroes and hollow dots & unspecified elements represent zeroes.

We present a brief introduction to the necessary mathematical background. We refer the readers to [73] for more details. Throughout the following discussion, we consider only real matrices.

$||A||$ denotes the norm of the matrix and is defined as follows:

$$||A|| = \sup_{x \neq 0} \frac{||Ax||}{||x||} \tag{6.3}$$

where $||x||$ refers to the L_2 norm of the vector. If A is a symmetric square matrix, then $||A|| = |\lambda_{max}(A)|$.

A is a block diagonally dominant matrix with blocks $\{A_{i,j} : 1 \leq i, j \leq N\}$ if

$$||A_{i,i}^{-1}||^{-1} \geq \sum_{j=1, j \neq i}^{N} ||A_{i,j}|| \quad 1 \leq i \leq N \tag{6.4}$$

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,N} \\ A_{2,1} & A_{2,2} & \dots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N,1} & A_{N,2} & \dots & A_{N,N} \end{pmatrix}$$

If all $\{A_{i,j}\}$ are symmetric square matrices, then the condition for block diagonal dominance becomes,

$$\lambda_{min}(A_{i,i}) \geq \sum_{j=1, j \neq i}^{N} |\lambda_{max}(A_{i,j})| \tag{6.5}$$

Accordingly, the matrix is positive semi-definite if the diagonal blocks are positive semi-

$$\begin{pmatrix} \Psi_1' I_x^2 + \gamma\Psi_2'(I_{xx}^2 + I_{xy}^2) + \beta\rho\Psi_3' + \alpha(\sum_{j\in N(i)} \Psi_S'(j)) & \Psi_1' I_x I_y + \gamma\Psi_2'(I_{xx}I_{xy} + I_{xy}I_{yy}) \\ \Psi_1' I_x I_y + \gamma\Psi_2'(I_{xx}I_{xy} + I_{xy}I_{yy}) & \Psi_1' I_y^2 + \gamma\Psi_2'(I_{yy}^2 + I_{xy}^2) + \beta\rho\Psi_3' + \alpha(\sum_{j\in N(i)} \Psi_S'(j)) \end{pmatrix}$$

Figure 6.3: Diagonal blocks for pixel i. Ψ_1', Ψ_2' and Ψ_S' are defined as follows: $\Psi_1' := \Psi'((I_x^k du^k + I_y^k dv^k + I_z^k)^2), \Psi_2' := \Psi'((I_{xx}^k du^k + I_{xy}^k dv^k + I_{xz}^k)^2 + (I_{xy}^k du^k + I_{yy}^k dv^k + I_{yz}^k)^2))$, $\Psi_3' = \Psi'((u^k + du^k - u_1)^2 + (v^k + dv^k - v_1)^2)$ $\Psi_S' := \Psi'(|\nabla(u^k + du^k)|^2 + |\nabla(v^k + dv^k)|^2)$. $N(i)$ denotes the neighborhood of pixel i.

definite and the matrix is block diagonally dominant [73].

Lemma 6.4.1 *The diagonal blocks of the matrix in the system of equations (6.2) are positive definite.*

Proof Consider a 2×2 matrix

$$\begin{pmatrix} x & y \\ y & z \end{pmatrix}$$

This matrix is positive semi definite iff

$$x + z \geq 0 \text{ and } y^2 \leq xz. \tag{6.6}$$

Construction of the 2×2 diagonal block matrix in the fixed point matrix [36] is shown in Figure 6.3. For convenience, we drop the pixel reference i from now on. All assignments are assumed to be made for a pixel i.

Let

$$s = \alpha(\sum_{j\in N(i)} \Psi_S'(j)) \tag{6.7}$$
$$p = \Psi_1' I_x^2 + \gamma\Psi_2'(I_{xx}^2 + I_{xy}^2) + \beta\rho\Psi_3' \tag{6.8}$$
$$q = \Psi_1' I_y^2 + \gamma\Psi_2'(I_{yy}^2 + I_{xy}^2) + \beta\rho\Psi_3' \tag{6.9}$$
$$r = \Psi_1' I_x I_y + \gamma\Psi_2'(I_{xx}I_{xy} + I_{xy}I_{yy}) \tag{6.10}$$

It is obvious that $p, q \geq 0$ and $s > 0$. If the gradients are not zero, we have $p, q > 0$.

The 2×2 matrix becomes

$$\begin{pmatrix} p+s & r \\ r & q+s \end{pmatrix}$$

From relation (6.6), the necessary and sufficient conditions for positive definiteness become

$$r^2 < (p+s)(q+s) \tag{6.11}$$
$$(\text{as } p + q + 2s > 0)$$
$$\Rightarrow r^2 < pq + s(p+q) + s^2 \tag{6.12}$$

It is sufficient to prove that $r^2 \leq pq$ as the rest of the terms on the right hand side are non-negative.

$$\begin{aligned}
pq - r^2 &= [\Psi_1' I_x^2 + \gamma\Psi_2'(I_{xx}^2 + I_{xy}^2) + \beta\rho\Psi_3'] \\
&\quad \times[\Psi_1' I_y^2 + \gamma\Psi_2'(I_{yy}^2 + I_{xy}^2) + \beta\rho\Psi_3'] \\
&\quad -[\Psi_1' I_x I_y + \gamma\Psi_2' I_{xy}(I_{xx} + I_{yy})]^2 \qquad (6.13) \\
&= \Psi_1'\Psi_2'\gamma[I_x^2(I_{yy}^2 + I_{xy}^2) + I_y^2(I_{xx}^2 + I_{xy}^2)] \\
&\quad +\gamma^2\Psi_2'^2(I_{xx}^2 I_{yy}^2 + I_{xy}^4) - 2\gamma^2\Psi_2'^2 I_{xx}I_{yy}I_{xy}^2 \\
&\quad -2\gamma\Psi_1'\Psi_2' I_x I_y I_{xx} I_{xy} - 2\gamma\Psi_1'\Psi_2' I_x I_y I_{xy} I_{yy} \\
&\quad +\beta\rho\Psi_3'[\Psi_1'(I_x^2 + I_y^2) + \gamma\Psi_2'(I_{xx}^2 + I_{yy}^2 + 2I_{xy}^2)] \\
&= \Psi_1'\Psi_2'\gamma[(I_x I_{yy} - I_y I_{xy})^2 \\
&\quad +(I_y I_{xx} - I_x I_{xy})^2] \\
&\quad +\gamma^2\Psi_2'^2(I_{xx}I_{yy} - I_{xy}^2)^2 \\
&\quad +\beta\rho\Psi_3'\Psi_1'(I_x^2 + I_y^2) \\
&\quad +\beta\gamma\rho\Psi_3'\Psi_2'(I_{xx}^2 + I_{yy}^2 + 2I_{xy}^2) \qquad (6.14) \\
&\geq 0 \qquad (6.15) \\
\Rightarrow r^2 &\leq pq \qquad (6.16) \\
\Rightarrow r^2 &< pq + s(p+q) + s^2 \quad \text{(as } s > 0\text{)} \qquad (6.17)
\end{aligned}$$

Lemma 6.4.2 *The matrix from the system of equations (6.2) is block diagonally dominant*

Proof Let $a = max(p, q)$ and $b = min(p, q)$. The eigenvalues of this matrix are $a + s + \delta$ and $b + s - \delta$ where

$$\delta = \frac{1}{2}[\sqrt{(a-b)^2 + 4r^2} - (a-b)] \qquad (6.18)$$

Note that $\delta \geq 0$. The smallest eigenvalue (λ_{min}) is $b + s - \delta$.

The off-diagonal matrices are diagonal. In fact, due to the equivalence between the x and y dimensions, they are identity matrices scaled by a constant ($\alpha\Psi_S'$). Therefore the sum of the maximum eigenvalues of the off-diagonal matrices becomes,

$$\begin{aligned}
\sum_{j=1, j\neq i}^{N} |\lambda_{max}(A_{i,j})| &= \alpha(\sum_{j\in N(i)} \Psi_S'(j)) \qquad (6.19) \\
&= s \qquad (6.20) \\
&\quad \text{(from equation (6.7))}
\end{aligned}$$

For block diagonal dominance, we require

$$b + s - \delta \geq s \tag{6.21}$$
$$b \geq \delta \tag{6.22}$$
$$\Rightarrow 2b \geq \sqrt{(a-b)^2 + 4r^2} - (a-b) \tag{6.23}$$
$$(a+b)^2 \geq (a-b)^2 + 4r^2 \tag{6.24}$$
$$4ab \geq 4r^2 \tag{6.25}$$
$$pq \geq r^2 \tag{6.26}$$

which is true from equation (6.16).

Hence, the matrix is block diagonally dominant.

Theorem 6.4.3 *The matrix from the system of equations (6.2) is symmetric positive semi-definite.*

Proof From Lemmas 6.4.1 and 6.4.2, the inner point matrix has positive definite diagonal blocks and is block diagonally dominant. It is also irreducible (as the connectivity extends to another element in the same row and column - usually true for any discrete Laplacian stencil). Due to symmetry in the Laplacian stencil, the matrix is also symmetric.

Therefore, the matrix is symmetric positive semi-definite. If the gradient is not the same everywhere, i.e., the matrix is not singular, it is even positive definite [73]. We will assume that the matrix is non-singular for our application.

Linear solvers involving positive (semi-)definite matrices can be much more efficient than those involving indefinite matrices. We will discuss how this property can be exploited for improving the performance of the linear solver and consequently the optical flow solver.

6.4.2 Linear solvers.

On the CPU, the linear system is usually solved using *Gauss-Seidel* relaxations, which have been empirically shown to be very efficient in this setting [37]. The Gauss-Seidel method is guaranteed to converge if the matrix is symmetric positive definite. Unfortunately, the Gauss-Seidel technique is inherently sequential as it updates the points in a serial fashion. It is hard to parallelize it efficiently on multi-core machines and even harder on GPUs.

It is possible to choose relaxation methods that have slightly worse convergence characteristics, but are easy to parallelize, such as *red-black relaxation* [157]. A single red-black relaxation consists of two half iterations - each half iteration updates every alternate point (called red and black points). The updates to all the red points are inherently parallel as all the dependencies for updating a red point are the neighboring black pixels and vice versa. Usually, this method is used with successive overrelaxation. Since we have a set of coupled equations, each relaxation will update (u_i, v_i) using a 2×2 matrix solve. Red-black relaxations have been used in a previous parallel optical flow solver [88].

Besides red-black relaxation, we consider the *conjugate gradient* method. This requires symmetric positive definiteness as a necessary and sufficient condition for convergence. The

convergence of the conjugate gradient technique depends heavily on the condition number of the matrix $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$. The condition numbers of the matrices obtained in the optical flow problems are very large and hence, convergence is usually slow.

A standard technique for improving convergence for ill-conditioned matrices is preconditioning to reduce the condition number of the system matrix. The pre-conditioner must be symmetric and positive definite. The special structure of the matrix allows for several regular pre-conditioners that work well in practice. In particular, we know that the diagonal blocks of the matrix are positive definite. Hence, a block diagonal matrix with only the diagonal blocks of the matrix is symmetric and positive definite and forms a good pre-conditioner. This pre-conditioner is usually referred to as a *block Jacobi preconditioner*. From now on, unless specified, we use the term conjugate gradient solver to refer to the preconditioned conjugate gradient solver with a block Jacobi preconditioner.

Performing this algorithmic exploration is important as choosing the right algorithm for the right platform is essential for getting the best speed-accuracy tradeoff. This fast LDOF implementation can now be used to track points in video.

6.4.3 Point tracking with large displacement optical flow

We demonstrate the utility of our LDOF implementation by suggesting a point tracker. In contrast to traditional local point trackers like KLT [150], variational optical flow takes global smoothness constraints into account. This allows the tracking of far more points as the flow field is dense and tracking is not restricted to a few feature points. Moreover, large displacement optical flow enables tracking limbs or other fast objects more reliably than conventional trackers.

Our tracking algorithm works as follows: a set of points is initialized in the first frame of a video. In principle, we can initialize with every pixel, as the flow field is dense. However, areas without any structure are problematic for tracking with variational optical flow as well. For this reason, we remove points that do not show any structure in their vicinity as measured by the second eigenvalue λ_2 of the structure tensor

$$J_\rho = K_\rho * \sum_{k=1}^{3} \nabla I_k \nabla I_k^\top, \tag{6.27}$$

where K_ρ is a Gaussian kernel with standard deviation $\rho = 1$. We ignore all points where λ_2 is smaller than a certain portion of the average λ_2 in the image.

Depending on the application, one may actually be interested in fewer tracks that uniformly cover the image domain. This can be achieved by spatially subsampling the initial points. Fig. 6.4 shows a subsampling by factor 8. The coverage of the image is still much denser than with usual keypoint trackers.

Each of the points can be tracked to the next frame by using the optical flow field $\mathbf{w} := (u, v)^\top$:

$$(x_{t+1}, y_{t+1})^\top = (x_t, y_t)^\top + (u_t(x_t, y_t), v_t(x_t, y_t))^\top. \tag{6.28}$$

As the optical flow is subpixel accurate, x and y will usually end up between grid points.

Figure 6.4: **Left: (a)** Initial points in the first frame using a fixed subsampling grid. **Middle: (b)** Frame number 15 **Right: (c)** Frame number 30 of the cameramotion sequence. Figure best viewed in color.

We use bilinear interpolation to infer the flow at these points.

The tracking has to be stopped as soon as a point gets occluded. This is extremely important, otherwise the point will share the motion of two differently moving objects. Usually occlusion is detected by comparing the appearance of points over time. In contrast, we detect occlusions by checking the consistency of the forward and the backward flow, which we found to be much more reliable. In a non-occlusion case, the backward flow vector points in the inverse direction as the forward flow vector: $u_t(x_t, y_t) = -\hat{u}_t(x_t + u_t, y_t + v_t)$ and $v_t(x_t, y_t) = -\hat{v}_t(x_t + u_t, y_t + v_t)$, where $\hat{\mathbf{w}}_t := (\hat{u}_t, \hat{v}_t)$ denotes the flow from frame $t+1$ back to frame t. If this consistency requirement is not satisfied, the point is either getting occluded at $t+1$ or the flow was not correctly estimated. Both are good reasons to stop tracking this point at t. Since there are always some small estimation errors in the optical flow, we grant a tolerance interval that allows estimation errors to increase linearly with the motion magnitude:

$$|\mathbf{w} + \hat{\mathbf{w}}|^2 < 0.01 \left(|\mathbf{w}|^2 + |\hat{\mathbf{w}}|^2\right) + 0.5. \tag{6.29}$$

We also stop tracking points on motion boundaries. The exact location of the motion boundary, as estimated by the optical flow, fluctuates a little. This can lead to the same effect as with occlusions: a tracked point drifts to the other side of the boundary and partially shares the motion of two different objects. To avoid this effect we stop tracking a point if

$$|\nabla u|^2 + |\nabla v|^2 > 0.01\, |\mathbf{w}|^2 + 0.002. \tag{6.30}$$

In order to fill the empty areas caused by disocclusion or scaling, in each new frame we initialize new tracks in unoccupied areas using the same strategy as for the first frame.

6.5 Results

The implementation platform consists of an Intel Core i7 920 processor running at 2.67 GHz in conjunction with a Nvidia GTX 480 GPU. For the LDOF implementations, almost all of the computation is done on the GPU and only minimal amount of data is transferred between the CPU and the GPU. We use Nvidia CUDA tools (v3.0) for programming the

GPU. The CPU code was vectorized using the Intel compiler with all the optimizations enabled.

For the tracking experiments, the KLT tracker used also runs on GPUs. A description of the exact algorithm is provided in [184]. The implementation in [184] also compensates for changes in camera exposure and provides real-time performance on the GPU considered. Default parameters were used unless otherwise specified.

6.5.1 Large displacement optical flow

Runtime for large displacement optical flow has come down from 68 seconds for the previous serial implementation on CPU to 1.84 seconds for the parallel implementation on GPU, a speedup of **37**$\times$ for an image size of 640×480. This implementation searches for Histogram of Gradients (HOG [61]) feature matches in a neighborhood of ± 80 pixels, uses $\eta = 0.95$, 5 fixed point iterations and 10 Conjugate gradient iterations to achieve the same overall Average Angular Error (AAE) as the CPU version on the Middlebury dataset. It is also possible to run the optical flow algorithm at a slightly reduced accuracy (AAE increase of about 11%) at 1.1 seconds per frame. We look closely at the choice of the linear solver that enabled this speedup.

Performance of linear solvers

Figure 6.5 shows the convergence of different solvers for the optical flow problem. We measure convergence through the squared norm of the residual $||b-Ax^m||^2$. The rates of convergence are derived from 8 different matrices from images in the Middlebury dataset [14]. Red-black and Gauss-Seidel solvers use successive overrelaxation with $\omega = 1.85$. The matrices considered were of the largest scale (smaller scales show very similar results). The initial vector in all the methods was an all-zero vector. Using a better initialization procedure (the result of a previous fixed point iteration, for instance) also shows similar results.

From Fig. 6.5, we can see why the Gauss-Seidel solver is the preferred choice for serial platforms. It converges well and is relatively simple to implement. In the numerical scheme at hand, however, we do not desire absolute convergence, as solving any one linear system completely is not important to the solution of the nonlinear system. It is more important to have a quick way of refining the solution and removing all the large errors. For a few iterations (30 or less), it is clear that the preconditioned conjugate gradient solver converges fastest. Non-preconditioned conjugate gradient is not as efficient because of the high condition number of the matrix.

Although it is clear from Fig. 6.5 that conjugate gradient converges quickly in terms of the number of iterations required, a single iteration of conjugate gradient requires more computation than a Gauss-Seidel or a red-black iteration. Table 6.1 shows the runtimes of the solvers. Even though red-black relaxations are also parallel, we can see from Fig. 6.5 that we require roughly 3$\times$ as many red-black iterations as conjugate gradient iterations to achieve the same accuracy. Red-black iterations are 1.4$\times$ slower than CG overall. Gauss-Seidel iterations, running on the CPU, are 53$\times$ slower compared to conjugate gradient on the

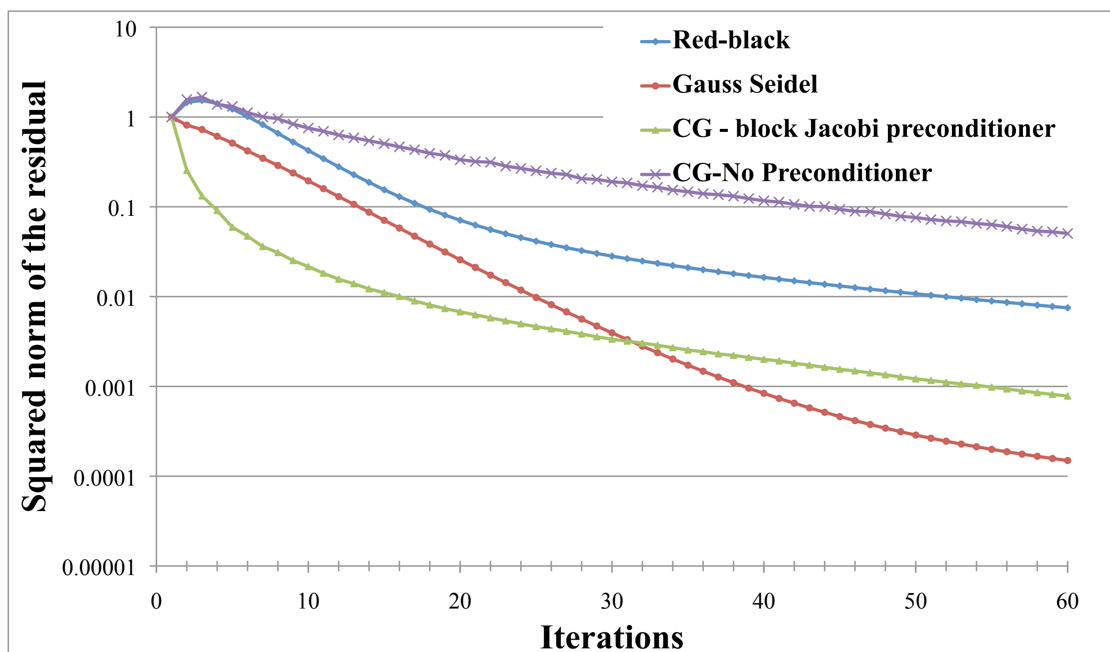


Figure 6.5: Rates of convergence for different techniques considered. Y-axis shows the value of the residual normalized to the initial residual value averaged over 8 different matrices. Figure best viewed in color

GPU. Note that the preconditioned conjugate gradient method is beneficial on multicore CPUs as well with a speedup of 4.6× with 4 threads. This shows that the numerical optimizations are independent of hardware and need to be optimized irrespective of the target architecture.

Linear Solver	Time taken (in milliseconds)
Gauss-Seidel (Serial CPU)	445.93
Red-black (Parallel GPU)	11.97
Conjugate Gradient (Parallel GPU)	8.39
Conjugate Gradient (Parallel CPU)	96.93

Table 6.1: Average time taken by the linear solvers for achieving residual norm $< 10^{-2}$ of initial norm

The major bottleneck in the conjugate gradient solver is the sparse matrix-vector multiply (SpMV). Care has been taken to lay out the matrix in memory properly to enable the data access patterns needed for the implementation. Since the matrix is block pentadiagonal, we use the diagonal format (DIA)[16] for storage. This leads to very efficient loads and stores from GPU memory. For parallelization, we perform a decomposition of the image into 16×16 segments, each of which is processed by a single thread block. There is limited reuse in the vector in SpMV and this can be exploited using GPU's local scratchpad memory (shared memory). All these optimizations enable the SpMV to run at 53 GFlops on the GPU. This is significant considering that the matrix is quite sparse (≤ 6 non-zeros per row). Under such conditions, most of the time in the kernel is spent fetching data to and from GPU main memory. Similar behavior is seen with the red-black relaxations, where 25% of the time is spent in floating point operations, while 75% of the time is spent in memory loads and stores. Red-black relaxations also have less computation to communication ratio (all the points are read, but only half the points are updated), which reduces their performance. The block Jacobi preconditioner is calculated on the fly every iteration. For applying the preconditioner, each pixel requires a 2×2 matrix-vector multiplication that can be performed independently.

Image Downsampling

Downsampling takes about 6% of the total runtime of large displacement optical flow. Since we perform a coarse-to-fine refinement, we have to downsample the original images to several different scales. For 640×480 sized frames at $\eta = 0.95$, there are 67 refinement levels. For each of the levels, we downsample from the original full-size images to the required resolution.

It is also to be noted that implementing such direct downsampling on a GPU is complicated. In particular, depending on the size of the downsampled image, different ways of parallelizing the computation can be more efficient. This is because of the fact that

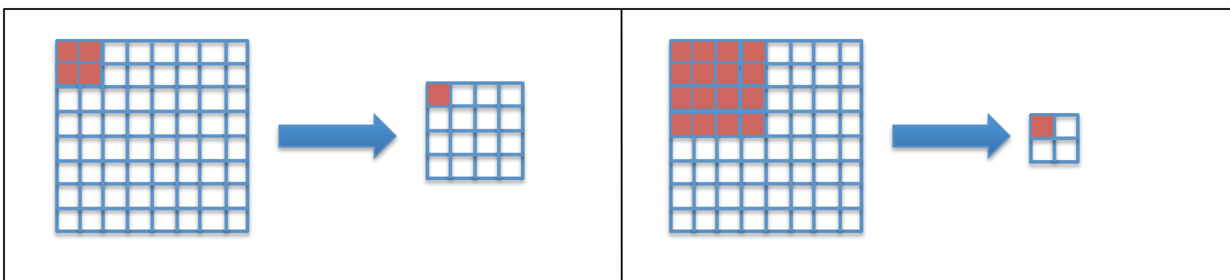
the amount of computation at different levels of the parallelism hierarchy vary smoothly as we go from coarse to fine resolutions. Figure 6.6 shows two different strategies that can be adopted for this computation to fit GPU's 2-level hierarchy (multiprocessor vs SIMD). Such a hierarchy exists in almost all parallel hardware today. In particular, one can map downsampling to GPU in 2 ways:

Downsampling by thread: Each GPU thread (SIMD lane) calculates the value of a single output pixel by performing the averaging serially.

Downsamping by block: Each GPU block calculates the value of a single output pixel by parallelizing the averaging over its threads.

Algorithm: Downsampling
Input: In (Input image)
s (downsampling scale < 1)
Output: Out (Output image)
1 **for each** pixel (x,y) in Out
2 $Out(x,y) \leftarrow$ **Average** of pixels in $In\{\lfloor \frac{y}{s} \rfloor \; to \; \lfloor \frac{y+1}{s} \rfloor, \lfloor \frac{x}{s} \rfloor \; to \; \lfloor \frac{x+1}{s} \rfloor\}$
3 **end for**

(a) Downsampling Algorithm. Both lines 1 and 2 contain concurrency. The amount of concurrency in each stage depends on the scale of downsampling.



(b) Graphical representation of downsampling strategies. **Left** Downsampling at 50% of original resolution. There is more concurrency in line 1 of the algorithm above. **Right** Downsampling at 25% of original resolution. There is more concurrency in line 2 of the algorithm above. In both figures, the red pixels on the left indicate the pixels that need to be averaged over to get the value of the pixel on the right.

Figure 6.6: Concurrency in downsampling

Figure 6.7 shows the runtime variation for downsampling the input images as we downsample the 640×480 images from 20×15 (0.1% pixels) to 608×466 (90% pixels). We see that the crossover point is at a scale of 4.5% (0.2% pixels). For downsamping scales below 4.5%, downsampling by block is better, whereas for larger scales downsampling by thread is better. This combination is the strategy we use for the rest of our computation.

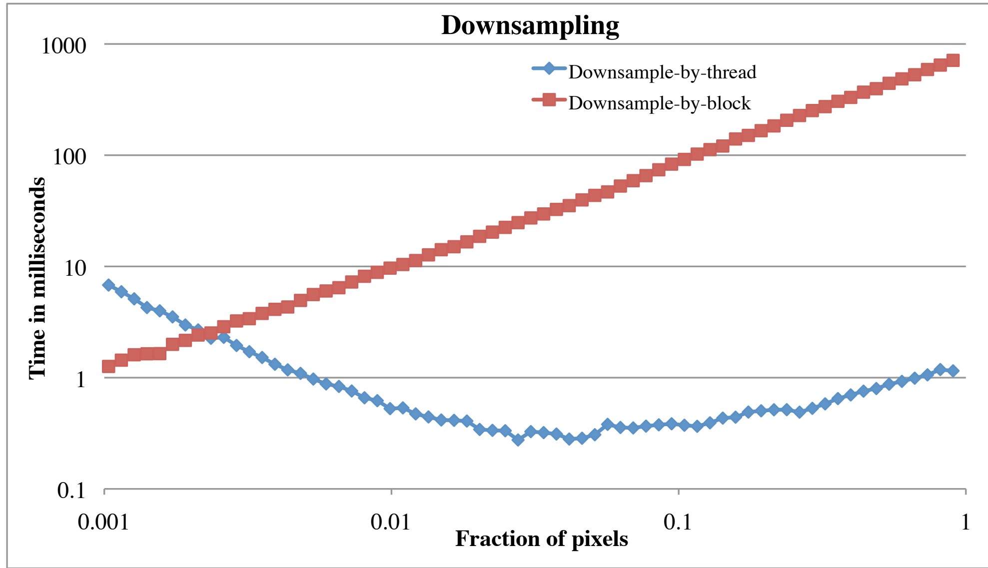


Figure 6.7: Comparison of different downsampling strategies. The crossover point is at a scale of 4.5% (0.2% of pixels).

Runtime breakdown

Figure 6.8 shows the breakdown of the serial optical flow solver that uses Gauss-Seidel and the parallel solver that uses conjugate gradient. The solvers were run with $\eta = 0.95$, 5 fixed point iterations and 25 Gauss-Seidel iterations/10 Conjugate gradient iterations to achieve similar AAE on the Middlebury dataset. From both Figure 6.8(a) and 6.8(b), it is clear that the HOG matching and the linear solver are the most computation intensive components in the solvers. In both cases, they take more than 75% of the total runtime.

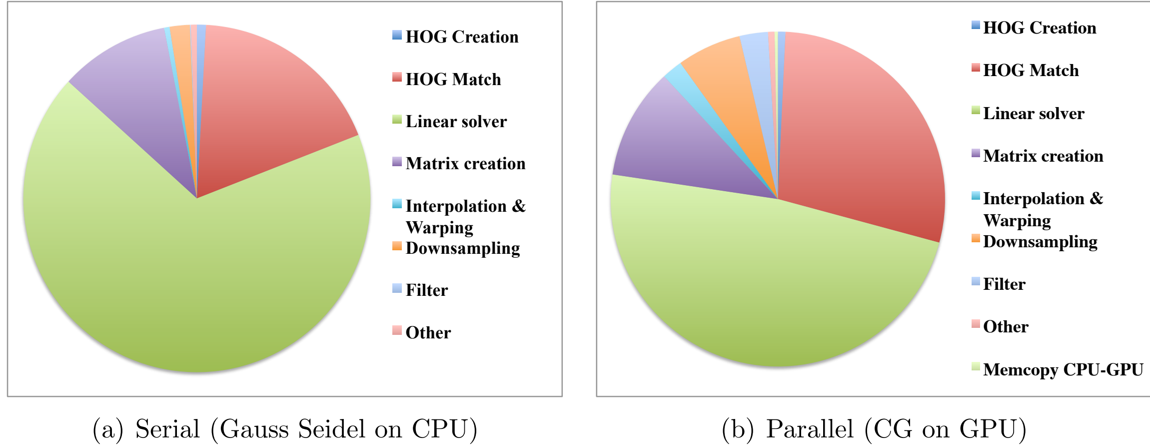
Accuracy

Table 6.2 shows the average angular error measured using our technique on the Middlebury dataset. These results have been achieved with the setting (γ = 4, β=30, α = 9, η = 0.95, fixed point iterations = 5, Gauss-Seidel iterations = 25/CG iterations = 10). The data shows that the method provides similar accuracy to the CPU version while running fast on the GPU.

Data	Dimetrodon	Grove2	Grove3	Hydrangea	RubberWhale	Urban2	Urban3	Venus	Average
AAE(CPU)	1.84	2.67	6.35	2.44	3.96	2.55	4.79	6.46	3.88
AAE(GPU)	1.84	2.51	5.94	2.37	3.91	2.47	5.43	6.38	3.86

Table 6.2: Average Angular Error (in degrees) for images in the Middlebury dataset.

For faster computations, we use the parameter set ($\eta = 0.75$, 5 fixed point iterations, 10 linear solve iterations) to reduce the runtime by 38% with a degradation in AAE of 11%.



(a) Serial (Gauss Seidel on CPU) (b) Parallel (CG on GPU)

Figure 6.8: Breakdown of execution times for serial and parallel variational optical flow solvers. Both solvers are run at a scale factor of 0.95, with 5 fixed point iterations and 25 Gauss-Seidel iterations/10 CG iterations to achieve similar AAE. Figure best viewed in color.

6.5.2 Tracking

We measure the accuracy of the tracking algorithms with the MIT sequences [117]. This dataset provides the ground truth optical flow for whole sequences and the sequences are much longer. This allows us to evaluate the accuracy of tracking algorithms. After obtaining the point trajectories from both KLT and LDOF, we track points using the given ground truth to predict their final destination. Tracking error is measured as the mean Euclidean distance between the final tracked position and the predicted position on the final frame according to the ground truth for all the tracked points. LDOF is run with $\eta = 0.95$, 5 fixed point iterations and 10 iterations for the linear solver in all the following experiments. Since the default parameters for the KLT tracker failed in tracking points in long sequences, we increased the threshold for positive identification of a feature from 1000 to 10000 (SSD_threshold parameter).

Accuracy

For our first experiment, we compare the accuracy of the trackers for a short fixed time (first 10 frames of all the sequences). Since tracking algorithms should ideally track points over long times without losing points, we only consider those points that are tracked through all the 10 frames. Table 6.3 shows the accuracy and density of the tracking algorithms.

Table 6.3 also shows the accuracy of all the tracked points and also of only the common points. Even without this normalization of number of tracks using only common points, LDOF outperforms KLT by 28%. When considering only those points that are tracked by both techniques, LDOF outperforms KLT by 40%. This means that even if one is interested in only sparse points, it makes sense to use the more accurate LDOF tracker.

Table 6.3: Tracking accuracy of LDOF and KLT over 10 frames of the MIT sequences

Sequence name	All tracked points				Common points only			
	LDOF		KLT		LDOF		KLT	
	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked
table	1.57	135803	3.19	403	1.18	363	1.69	363
camera	0.46	149654	1.69	576	0.43	353	0.98	353
fish	0.98	234993	3.82	507	0.82	388	2.98	388
hand	5.84	176673	3.91	549	2.59	505	3.34	505
toy	1.43	414183	1.61	906	1.01	800	1.17	800

We also compare the accuracy of the trackers for the entire length of the sequences. The real advantage of our approach comes in while tracking long sequences. Trackers like KLT keep losing features and need to be constantly detecting new features every few frames to track well. From Table 6.4, it is clear that LDOF tracks almost three orders of magnitude more points than KLT with 46% improvement in overall accuracy. For tracking only the common points, the LDOF tracker is 32% better than KLT. These numbers exclude the fish sequence since it has transparent motion caused by dust particles moving in the water. Although we were able to track this sequence well, performance on this sequence is sensitive to parameter changes.

Sequence name	Number of frames	All tracked points				Common points only			
		LDOF		KLT		LDOF		KLT	
		Mean error in pixels	Points tracked	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked
table	13	1.48	114651	3.78	363	1.04	293	1.39	293
camera	37	1.41	101662	3.78	278	1.01	185	2.64	185
fish	75	3.39	75907	35.62	106	3.12	53	5.9	53
hand	48	2.14	151018	3.11	480	1.87	429	2.39	429
toy	18	2.24	376701	2.88	866	1.70	712	1.89	712

Table 6.4: Tracking accuracy of LDOF and KLT over all the frames of the MIT sequences

Compared to the Particle Video point tracker in [146], our tracker is 66% more accurate for the common tracks. Since ground truth data does not exist for the sequences used in [146], it is not possible to have objective comparisons on metrics other than the average round trip error (The videos are mirrored temporally, so all unoccluded pixels should return to their starting positions). For comparison, we use only the full-length particle trajectories provided by the authors of [146] at http://rvsn.csail.mit.edu/pv/data/pv. Details of the comparison are provided in Table 6.5.2.

Occlusion handling

We use the region annotation data from the MIT dataset to measure the occlusion handling capabilities of the algorithms. The LDOF tracker has an occlusion error of 3%

Sequence name	Number of frames	All tracked points				Common points only			
		LDOF		Particle Video		LDOF		Particle Video	
		Mean error in pixels	Points tracked	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked	Mean error in pixels	Points tracked
Mouth	70	0.51	114643	7.62	6364	0.79	1232	2.88	1232
Hand	70	0.59	140726	3.15	7029	0.74	3998	3.00	3998
Cars	50	0.30	142052	1.09	10812	0.38	5612	0.84	5612
Branches	50	0.60	125086	5.49	2883	0.71	946	4.49	946
Hall	50	0.73	108641	2.01	7649	0.78	4205	1.57	4205
Person	50	0.47	114335	8.15	6342	0.48	4048	8.18	4048
Shelf	50	0.60	119881	5.98	9016	0.55	4723	5.90	4723
Treetrunk	50	0.40	204777	3.65	9002	0.49	7132	3.71	7132
Plant	70	0.50	170361	2.07	8078	0.84	3347	1.17	3347
Tree	70	0.27	168905	1.51	5529	0.29	2566	1.17	2566
RectFast	80	0.08	112992	0.78	10516	0.06	4558	0.10	4558
RectLight	80	0.11	112478	0.66	9277	0.08	3909	0.14	3909
RectSlow	80	0.12	119457	0.61	11046	0.09	4812	0.14	4812
CylFast	50	0.15	109939	1.90	10688	0.11	4069	0.27	4069
CylLight	50	0.19	120985	1.17	11789	0.14	4214	0.32	4214
CylSlow	50	0.15	126924	1.08	12208	0.11	4816	0.19	4816
ZoomIn	40	2.07	15908	3.17	8880	1.93	416	3.64	416
ZoomOut	40	2.45	14838	8.50	9334	2.43	253	8.59	253
RotateOrtho	90	3.34	28984	3.07	11934	3.03	800	2.66	800
RotatePersp	90	2.90	19660	2.11	10965	2.84	432	1.14	432
Average	61.5	0.83	109579	3.20	8967	0.84	3304	2.51	3304

Table 6.5: Tracking accuracy of LDOF and Sand-Teller tracker over the sequences used in [146]

(tracks that drift between regions/objects) while the KLT tracker has an occlusion error of 8%. Given that KLT tracker is already very sparse, this amounts to a significant number of tracks that are not reliable (they do not reside on the same object for the entire time). After excluding all the tracks that were known to have occlusion errors, LDOF outperforms KLT by 44%. Since all the ground truth occlusions are known, we measure the tracking density (% of unoccluded points that the tracker was successful in tracking through the entire sequence without any occlusion errors). The LDOF tracker has an average tracking density of 48%, i.e., it tracks roughly half of the points that are not occluded for the entire length of the sequence, while KLT has a density of about 0.1%. Table 6.6 presents the data on occlusion handling.

Sequence	KLT		LDOF		
	Number of occluded tracks	Mean error with no occlusion	Number of occluded tracks	Mean error with no occlusion	Tracking Density (%)
table	11	2.73	853	1.41	39.6
camera	8	3.68	558	1.37	39.9
fish	30	31.79	8321	2.7	53.0
hand	10	2.90	2127	1.81	46.8
toy	31	2.58	5482	2.11	61.4

Table 6.6: Occlusion handling by KLT and LDOF trackers based on region annotation from the MIT data set. Occluded tracks indicate tracks that are occluded according to the ground truth data, but not identified as such by the trackers.

Large displacements

The MIT sequences still mostly contain small displacements and hence KLT is able to track them well (if it does not lose the features). However, there are motion sequences with large displacements that are difficult for a tracker like KLT to capture. In the tennis sequence [32], there are frames where the tennis player moves very fast, producing motion that is hard to capture through simple optical flow techniques. Since ground truth data does not exist for this sequence, we manually labeled the correspondences for 39 points on the player between frames 490, 495 and 500. These points were feature points identified by KLT in frame 490. The results for the points tracked on the player are shown in Table 6.7 and Figure 6.9. It is clear that the LDOF tracker tracks more points with better accuracy, while capturing the large displacement of the leg.

Runtime

The cameramotion sequence with 37 frames of size 640×480, requires 135 seconds. Out of this, 125 seconds were spent on LDOF (both forward and backward flow). Such runtimes allow for convenient processing of large video sequences on a single machine equipped with cheap GPU hardware.

Frames	LDOF		KLT	
	Mean error in pixels	Points tracked on player	Mean error in pixels	Points tracked on player
490-495	2.55 (22)	8157	3.21 (19)	21
490-500	2.62 (8)	3690	4.12 (4)	4

Table 6.7: Tracking accuracy of LDOF and KLT for large displacements in the tennis sequence with manually marked correspondences. Numbers in parentheses indicate the number of annotated points that were tracked.

Figure 6.9: **(Top)** Frame 490 of the tennis sequence with **(left)** actual image, **(middle)** KLT points and **(right)** LDOF points. **(Bottom)** Frame 495 of the sequence with **(left)** actual image, **(middle)** KLT points and **(right)** LDOF points. Only points on the player are marked. KLT tracker points are marked larger for easy visual detection. Figure best viewed in color.

6.6 Summary

We described how large displacement optical flow can be efficiently parallelized and implemented on GPUs. We saw that the most compute intensive component of the optical flow solver is a linear solver. We proved that the linear system is positive definite, thus justifying the use of the conjugate gradient solver. We described the tradeoffs between runtime per iteration and number of iterations among different serial and parallel linear solvers and showed why we decided on the preconditioned conjugate gradient solver as our solver of choice. We discussed how our fast optical flow can be used as a base for a point tracking system.

The main contributions are as follows -

1. Showing that the linear system of equations used in optical flow systems in positive semi-definite even when the penalty function is non-convex, thus theoretically guaranteeing the convergence of all the solvers used (assuming non-singular matrix).

2. Showing that the preconditioned conjugate gradient solver is more efficient than red-black relaxations and is more scalable on multicore processors and GPUs.

3. Showing that accurate optical flow is more important for point tracking than complicated integration techniques and developing a point tracker that is denser and more accurate than existing state-of-the-art point trackers while still tracking large displacements.

We showed how specializing numerical algorithms to specific problems is essential when parallelizing computer vision applications. Linear solvers that worked well in the serial case are easily overtaken by other solvers on parallel hardware. Our experiments quantitatively show for the first time that tracking with dense motion estimation techniques provides better accuracy than KLT feature point tracking by 46% on long sequences and better occlusion handling. We also achieve 66% better accuracy than the Particle Video tracker. Our point tracker based on LDOF improves the density by up to three orders of magnitude compared to KLT and handles large displacements well, thus making it practical for use in motion analysis applications. The overall contribution, in addition to efficient implementations of optical flow and point tracking, is showing how important algorithmic exploration is in the context of computer vision algorithms, especially with the use of iterative algorithms where the solution is only required to converge to a fixed numerical accuracy. Many linear algebra problems in computer vision are iterative algorithms like linear solvers, eigensolvers, singular value decomposition etc. In the next chapter, we will discuss image and video segmentation algorithms that involve large eigensolvers.

Chapter 7

Image and Video Segmentation

In this chapter, we discuss how image and video segmentation applications can be accelerated. We show how image segmentation problems can be speeded up significantly through improvements to local gradient and eigensolver computations. By ensuring portability of the eigensolver across different GPUs, in particular those with less memory, we have been able to extend our approach to solve video segmentation problems. We also show that our video segmentation approach is superior to other existing video segmentation techniques.

7.1 Background and Motivation

Segmentation of images and videos is required before analyzing either media for object detection and recognition etc. However, the process of segmentation is not simple because it mirrors a subjective human process. Textures, illumination variations, color differences, depth discontinuities etc. create artificial boundaries within the same object while different objects can look similar. Segmentation algorithms need to be aware of this problem and take it into account. The original image segmentation algorithms were edge detection algorithms based on spatial gradients in the image. Examples of such edge detection techniques include Prewitt, Sobel and Canny [43] edge detectors etc. These detectors were mostly based on calculating local gradients on brightness images. More accurate techniques for image contour detection are based on spectral segmentation and normalized cuts as introduced by Shi and Malik [151].

As discussed in Chapter 4, the normalized cuts approach requires solving an eigenvalue problem on an image affinity graph. For our work, we perform image segmentation using global Probability of boundary (gPb) [119], a technique that combines multiscale localized boundary detectors and normalized cuts. The highest quality image contour detection currently known, as measured on the Berkeley Segmentation Dataset [123], is the gPb detector. Details of this detector are given in Section 7.2.

The normalized cuts approach can also be used for video segmentation. However, the major constraining factors are the size of the affinity matrix and the computational requirements. There is also a need to extend the calculation of pixel affinities to pixels belonging to different frames. This can be done only when we know the correspondence between pixels

in both the frames as the camera or scene objects could have moved. This correspondence problem is exactly the one solved by optical flow. Hence, optical flow is essential for extending the normalized cuts approach to video segmentation. More details on how we extend the segmentation to video are given in Section 7.5.3.

We will look at the details of the gPb image contour detector in the next section.

7.2 The gPb Detector

The gPb detector consists of many modules, which can be grouped into two main components: mPb, a detector based on local image analysis at multiple scales, and sPb, a detector based on the Normalized Cuts criterion. An overview of the gPb detector is shown in figure 7.1.

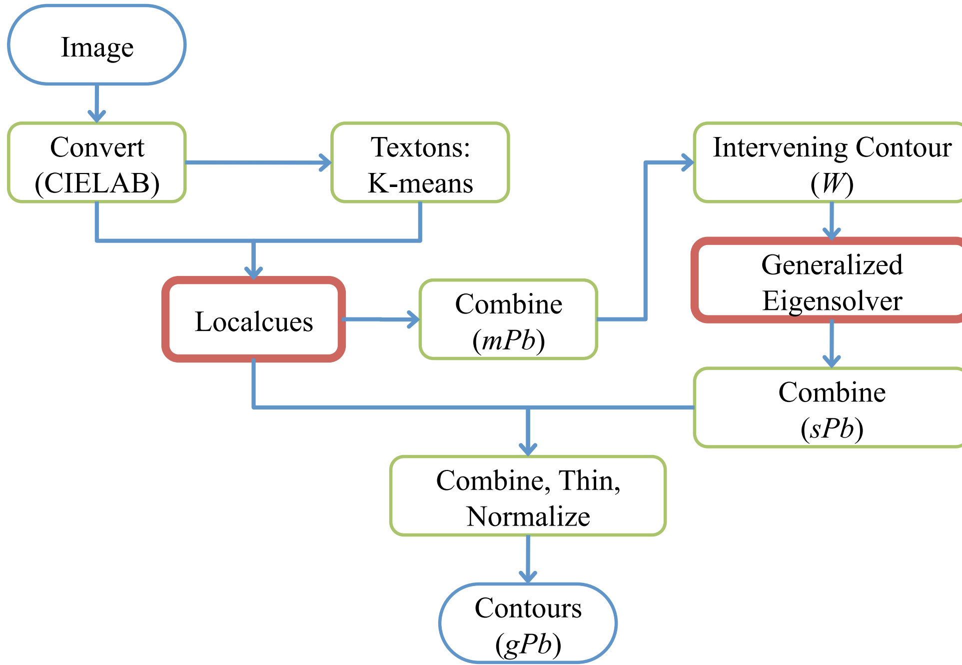


Figure 7.1: The gPb detector

The mPb detector is constructed from brightness, color and texture channels from the image. The brightness and color channels are obtained from a RGB to CIELAB transformation. The L channel measures brightness and the A,B channels measure the color. Textons are calculated through k-means clustering from a multiscale filter bank [121]. For each channel, we estimate the probability of boundary using the multiscale detector from [122]. This probability of boundary $Pb_{C,\sigma}(x, y, \theta)$ is calculated at each pixel (x, y) and 8 orientations for each channel C at multiple scales. At each pixel, $Pb_{C,\sigma}(x, y, \theta)$ is calculated as the χ^2 distance between the histograms computed on channel C over the two halves of a

disk centered at (x, y) with radius σ and dividing the disk at an angle θ. The mPb detector is then constructed as a linear combination of the local cues, where the weights α_{ij} are learned by training on an image database:

$$mPb(x, y, \theta) = \sum_{i=1}^{4} \sum_{j=1}^{3} \alpha_{ij} Pb_{C_i, \sigma_j}(x, y, \theta) \tag{7.1}$$

The mPb detector is then reduced to a pixel affinity matrix W, whose elements W_{ij} estimate the similarity between pixel i and pixel j by measuring the intervening contour [112] between pixels i and j. Due to computational concerns, W_{ij} is not computed between all pixels i and j, but only for some pixels which are near each other. In this case, we use Euclidean distance as the constraint, meaning that we only compute W_{ij} $\forall i, j$ $s.t.$ $||(x_i, y_i) - (x_j, y_j)|| \leq r$, otherwise we set $W_{ij} = 0$. We set $r = 5$ for all our experiments.

This constraint, along with the symmetry of the intervening contour computation, ensures that W is a symmetric, sparse matrix, which guarantees that its eigenvalues are real, significantly influencing the algorithms used to compute sPb. Once W has been constructed, sPb follows the Normalized Cuts approach [151], which approximates the NP-hard normalized cuts graph partitioning problem by solving a generalized eigensystem. To be more specific, we must solve the generalized eigenproblem:

$$(D - W)v = \lambda D v, \tag{7.2}$$

where D is a diagonal matrix constructed from W: $D_{ii} = \sum_j W_{ij}$. Only the $k + 1$ eigenvectors v_j with smallest eigenvalues are useful in image segmentation and need to be extracted. In this case, we use $k = 16$. The smallest eigenvalue of this system is known to be 0, and its eigenvector is not used in image segmentation, which is why we extract $k + 1$ eigenvectors. After computing the eigenvectors, we extract their contours using Gaussian directional derivatives at multiple orientations θ, to create an oriented contour signal $sPb_{v_j}(x, y, \theta)$. We combine the oriented contour signals together based on their corresponding eigenvalues:

$$sPb(x, y, \theta) = \sum_{j=2}^{k+1} \frac{1}{\sqrt{\lambda_j}} sPb_{v_j}(x, y, \theta) \tag{7.3}$$

The final gPb detector is then constructed by linear combination of the local cue information and the sPb cue:

$$gPb(x, y, \theta) = \gamma \cdot sPb(x, y, \theta) + \sum_{i=1}^{4} \sum_{j=1}^{3} \beta_{ij} Pb_{C_i, \sigma_j}(x, y, \theta) \tag{7.4}$$

where the weights γ and β_{ij} are also learned via training. To derive the final $gPb(x, y)$ signal, we maximize over θ, threshold to remove pixels with very low probability of being a contour pixel, skeletonize, and then renormalize.

We will discuss how we make this computation efficient and how we can extend this approach to videos.

7.3 Problem

gPb computation requires more than $138,000$ floating point operations per pixel as previously mentioned in Chapter 2. Given the rise in the computational requirements for image contour detection algorithms over the years, it is necessary to find ways to execute these algorithms efficiently through numerical optimizations and efficient implementations that take advantage of the hardware.

Dense eigenproblems usually scale as $O(N^3)$, where N is the matrix dimension. However, since we need only few extreme (smallest in our case) eigenvalues and eigenvectors of a real, symmetric positive semidefinite matrix, we use efficient techniques like Lanczos algorithm. Unfortunately, not all steps of the Lanczos algorithm are efficiently parallelizable; in particular, the reorthogonalization of Lanczos vectors is computationally very expensive. Performing eigendecomposition on video affinity matrices takes much longer because of the increase in the size of the matrix by a factor of $O(m)$, where m is the number of frames in the video.

The affinity matrices required for performing eigenvalue decomposition are large. For example, for a 640×480 image, the matrix is of size 100 MB. Each iteration of the eigensolver needs 1.2MB of memory. With a GPU with memory of 1 GB, the maximum number of iterations that can be fit is only 750. This may not be sufficient as the the eigensolver usually requires 500 - 1000 iterations for convergence. Hence, we need ways to efficiently solve the eigenvalue problem within a finite amount of memory. In addition to this problem, many of the GPUs in use are small ones, used primarily for graphics. These GPUs, commonly used in laptops and low-end machines have less than 512 MB of graphics memory. In order to enable portability to these machines, it is necessary to pay particular attention to this memory problem.

This challenge is even more pronounced for video segmentation. For a video sequence with a resolution of 640×480 and having 100 frames, the affinity matrix reaches a size of about 30 GB. Adding the memory required to store the temporary values during the computation, the total memory required for the computation reaches 150 GB. This is clearly infeasible even with a small cluster of GPUs unless the algorithms are modified to use less memory.

Even if we manage to calculate the eigenvectors for video affinity matrices, there are further obstacles. It is well known that using spectral segmentation over long time scales leads to smooth transitions and blending. Ideally, pixels belonging to the same cluster have similar values in the eigenvector space no matter how far they are (spatially or temporally). In practice, smooth transitions occur over long time periods and the identity of pixels belonging to a single object changes over time. Therefore in addition to making the computations efficient, we also need to find a way to produce robust and accurate segmentations from the eigenvector data.

7.4 Related Work

7.4.1 Image segmentation

Image segmentation remains an active field of study for many reasons and in fact segmentation and recognition are the main challenges in computer vision. Normalized cuts [151] has become the most popular method for performing image contour detection even though it is computationally very intensive. Prior to that, image contour detection was mainly done through edge detection techniques such as Sobel, Prewitt and Canny [43]. In the case of Sobel and Prewitt edge detectors, we apply 3×3 filters (which are similar to differential operators) on the image. Canny edge detector detects edges at multiple orientations and uses non-maximum suppression to pick an edge orientation for every pixel. This edge magnitude gets thresholded and an edge map results. Apart from normalized cuts, the other popular technique in use today is the one by Felzenszwalb et al [74]. This technique uses minimum spanning tree as the basis for agglomerative clustering to produce image segmentations. However, these other techniques have been shown to be inferior in accuracy to techniques based on normalized cuts like gPb [119].

We also briefly summarize some of the prior work that have tried to improve the runtime and accuracy of the normalized cuts technique. Techniques like [124] showed that using multiple cues like brightness, texture and color improves segmentation accuracy. [141] showed that using multiscale techniques is essential for good segmentation. Improvements like gPb [119] have combined both local and global segmentations in order to produce better image contours. gPb has been shown to have the highest accuracy on the Berkeley Segmentation Data Set [123]. However, it is still impractical to use on large datasets and video sequences because of the high computational requirements. It takes more than 4 minutes to segment a 481×321 image using gPb [47]. Techniques to parallelize and improve on the runtime of gPb and in particular, the eigensolver used, are an important contribution to the field.

Improvements to the eigensolver have been studied by a few researchers. Dhillon et al [66] reformulated normalized cuts as a form of weighted kernel k-means clustering algorithm that can be solved very efficiently. However, gPb requires the eigenvectors of the affinity matrix, not just the graph partition corresponding to the smallest normalized cut. Techniques like [165] use a multigrid strategy in order to improve runtime. The eigenproblem is solved at a coarser scale and is used to initialize the problem at finer scales and refined using inverse iterations. [59] approximate the affinity matrix using low-rank approximations that theoretically improves accuracy by increasing the connectivity. All the above mentioned techniques have been shown to increase accuracy or speed at the cost of the other i.e. the techniques have not been able to increase speed without sacrificing accuracy. On the other hand, our parallelization of gPb on GPUs and optimizations to the eigensolver have resulted in speedups with no loss of accuracy. The improvements to the eigensolver have been portable across many different applications such as video segmentation [161], point trajectory classification [35] etc., which are all based on spectral segmentation.

As mentioned earlier, we use the Lanczos algorithm for solving the eigenproblem. Since reorthogonalization of vectors turned to be a major bottleneck in this method, approaches

like [60] have been used to remove the need for reorthogonalization. However, a straightforward application of this technique on GPUs will not be successful due to the limited amount of memory that GPUs have. We need to be able to run large eigensolvers in limited GPU memory, which will improve the portability of the technique while simultaneously taking advantage of the improved memory bandwidth that GPUs provide.

7.4.2 Video segmentation

Dense video segmentation is being actively studied by the computer vision community. We will discuss some of the more recent approaches to this problem. Some of the recent techniques for doing video segmentation include Multiple Hypothesis Video Segmentation (MHVS) [167], hypergraph segmentation [96], hierarchical graph based segmentation [86], Circular Dynamic Time Warping (CDTW) [30] and [63]. MHVS [167] works by performing 2D superpixel segmentation of the frames at multiple scales and tries to find the best hypothesis that connects the superpixels from one frame to another. In order to do this, MHVS uses higher order potentials on a Conditional Random Field (CRF) and performs inference on the resulting graphical model. Hierarchical graph based segmentation [86] (which builds on [74]) proposes the creation of a hierarchical 3D super pixel (super voxel) representation for videos. They achieve this by creating a region graph and combining adjacent nodes if the "internal variation" of both the nodes is larger than the edge weight between the nodes. Hypergraph segmentation [96] works by extending the classic normalized cuts approach [151] to hypergraphs created from 2D superpixels produced from the oversegmentation of individual frames. DeMenthon [63] uses a Hough transform-like approach with 3D volume to identify objects in video. Brendel and Todorovic [30] use CDTW to temporally coalesce 2D superpixels obtained from image segmentation into 3D volumes. All of these approaches are either based on 2D superpixels (in which case temporal coherency has been difficult to maintain) or are not very robust (segmentation is noisy).

On the other hand, multi-body factorization and related methods are being used for motion based classification in videos. These methods usually rely on point trajectories extracted from the video. Some of the recent techniques that have been proposed for classifying point trajectories based on motion include [35, 139, 169, 181]. The technique in [35] performs spectral clustering on point trajectories. It also introduces a data set of labeled images sampled densely in space and time. We use this data set (also referred to as the motion segmentation data set) for obtaining quantitative results for video segmentation. Other methods for performing clustering of point trajectories are factorization based approaches like Generalized Principal Component Analysis (GPCA) [169], Local Subspace Analysis(LSA) [181] and Agglomerative Subspace Clustering (ALC) [139]. A major drawback of all these approaches is the difficulty in getting dense labels for videos. At best, these techniques label only about $3 - 4\%$ of the pixels in the video. This might be sufficient for identifying large moving blobs like people or vehicles, but is not adequate for doing tasks like editing videos by cutting/pasting objects or segmenting objects that are distinct in appearance but are not moving independently.

Using Spectral clustering for segmentation was popularized by normalized cuts [151]. Shi

and Malik [152] used a normalized cuts approach to do video segmentation. However, the technique was constrained by the lack of computational power, limiting video segmentation to only ∼10 frames. Also, image contour detection has since improved through the addition of multiscale local information and an improved integration of spectral and multiscale local information (gPb) [119]. These additions along with the improved reliability in extracting optical flow from videos and tremendous improvements in computational power from GPUs have brought back the viability of organizing video segmentation as a generalization of image segmentation.

7.5 Our approach

In order to improve the performance of gPb and extend it to video, we made the following contributions.

1. Showed that using integral images for calculating local cues does not reduce accuracy while speeding up the computation
2. Engineered an eigensolver that is able to run even when there is only about 256MB of memory, thus making it portable across a wide range of GPU hardware from laptops to clusters
3. Extended the technique to long video sequences and showing that the accuracy of this technique is comparable to or better than other existing techniques.

7.5.1 Calculating Local cues

First, we explain how we perform local cues computation in gPb. As mentioned in Section 7.2, in globalPb [119], multiscale gradients are calculated for brightness, color and texture channels. Brightness, texture and color gradients are calculated as χ^2-distance between the histograms of half-disks at every pixel at different orientations and scales on the brightness, texture and color channels respectively. This computation requires a lot of redundant computation. For example, local sums calculated at scale σ can be reused while calculating histograms at scales $> \sigma$. This can be done using integral images [115]. Integral image $I(x, y)$ for an image $f(x, y)$ is defined as follows:

$$I(x, y) = \sum_{i=1}^{x} \sum_{j=1}^{y} f(i, j) \tag{7.5}$$

Integral image has proved to be a very successful technique for reducing the runtime of computing Haar-like features [115], histograms in rectangular regions [170] etc. With integral images, the sum of $f(x, y)$ over any rectangular region can be calculated in constant time

by equation 7.6.

$$\sum_{i=x_1}^{x_2}\sum_{j=y_1}^{y_2} f(i,j) = I(x_1-1, y_1-1) + I(x_2, y_2) - I(x_1-1, y_2) - I(x_2, y_1-1) \tag{7.6}$$

By having an integral image for every bin of computed histogram, we can accelerate the local cues computation. However, the key challenge here is that the sums need to be calculated at multiple orientations not just rectangles aligned with the x and y axes.

In order to use integral images, we approximate the half-disks over which histograms are computed as rectangles. See [122] for more details. The rest of this section deals with our technique for computing the rotated integral images and showing that their error performance is better than that of nearest neighbour interpolation for rotation, while also improving performance.

Computing rotated integral images

In order to produce rotated integral images, we first rotate the image using a method of choice and then compute the integral image as is done for unrotated images. The problem associated with computing rotated images is to maintain the same number of pixels with a one-to-one correspondence between the original and rotated images. The usual solution to this problem is to use nearest neighbor interpolation to fill in pixels that do not exactly correspond to a pixel in the input image. Doing nearest neighbor interpolation produces artifacts due to missing pixels (some pixels in the original image do not appear on the rotated image) and multiple counting (some pixels in the original image appear many times in the rotated image). Getting rid of this problem is essential because for some categories (like texture channels using textons), interpolation does not make sense. For example, a pixel that needs to be interpolated from pixels with textons 1 and 3 cannot be interpolated to the mean value 2 as the categories are unordered. We propose using Bresenham lines to do the rotation because it overcomes this problem and is more accurate. Bresenham lines are widely known for producing lines of a given length and orientation on a discrete grid [31].

Bresenham rotation introduces some computational inefficiencies due to the fact that they increase the size of the image on which integral image should be computed. Using Bresenham lines to maintain image integrity i.e. have a unique pixel for every pixel in the original image, produces rotated images that are larger than the original images. This is because a line of n pixels at an angle θ produced by Bresenham rotation is of actual length $\frac{n}{cos\theta}$. This can be more easily seen in figure 7.2. Bresenham rotation produces images that are atmost twice as large as the original image, the largest occurring at $\theta = \frac{\pi}{4}$. Bresenham rotation is more accurate than nearest neighbor interpolation, since pixels are not missed or multiply counted during the image integration, as occurs using nearest neighbor interpolation. Therefore, we use it in our local cue detector.

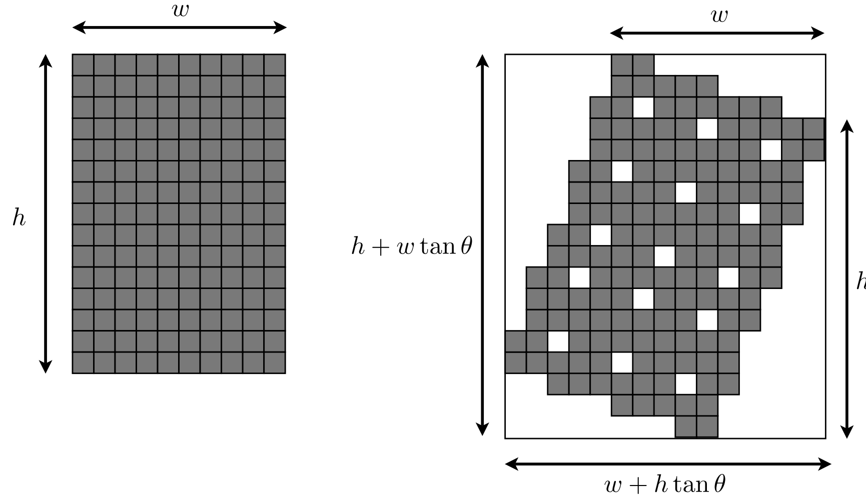


Figure 7.2: Bresenham rotation: Rotated image with $\theta = 18°$ clockwise, showing "empty pixels" in the rotated image.

7.5.2 Memory management in eigensolver

As previously mentioned, the eigensolver in gPb is implemented using the Lanczos algorithm. We use Lanczos because this algorithm is optimized for finding the extremal eigenvalues and eigenvectors of symmetric positive semidefinite matrices. In our case, we find the n smallest eigenvalues and the corresponding eigenvectors. Figure 7.3 shows the pseudocode of the Lanczos algorithm.

Since the Lanczos algorithm is heavily bandwidth bound, running the algorithm on GPUs can improve performance as GPUs have high memory bandwidth. However, the disadvantage of using GPUs is their limited memory capacity. Lanczos algorithm is iterative, and requires us to store a large amount of intermediate data (V_j in figure 7.3). We describe how we overcome this shortcoming and manage to run the eigensolver on GPUs with insufficient memory.

The Lanczos algorithm requires reorthogonalization of the Lanczos vectors because of the loss of orthogonality caused by floating point round off errors as calculations are done with finite precision. This process of reorthogonalizing the vectors is compute intensive. However, we note an important property of the problem; since the eigenvectors of the matrix correspond to segmentations, we expect the eigenvalues of the matrix to be unique. In other words, different eigenvectors have distinct eigenvalues. This assumption lets us use the Cullum-Willoughby method [60] and remove the need for reorthogonalization. Catanzaro et al [47] showed how this optimization can produce speedups of over 20× compared to doing full reorthogonalization at every iteration.

However, the problem with doing no reorthogonalization is that it requires many more iterations than when performing reorthogonalization. In such a scenario, it is very easy to run out of GPU memory to store the Lanczos vectors. We note that not all the Lanczos

Algorithm: Lanczos
Input: A (Symmetric Matrix)
v (Initial Vector)
Output: Θ (Ritz Values)
X (Ritz Vectors)

```
1  Start with r ← v
2  β_0 ← ||r||_2
3  for j ← 1, 2, ..., until convergence
4      (r, v_j, α_j) = LanczosIteration(r, β_{j-1}, A, v_{j-1})
5      Reorthogonalize if necessary
6      β_j ← ||r||_2
7      Compute Ritz values T_j = SΘS^T
8      Test bounds for convergence
9  end for
10 Compute Ritz vectors X ← V_j S

function LanczosIteration(r_in, β_{j-1}, A, v_{j-1})
1  v_j ← r_in/β_{j-1}
2  r ← Av_j
3  r ← r − v_{j-1}β_{j-1}
4  α_j ← v_j^* r
5  r ← r − v_j α_j
6  return (r, v_j, α_j)
```

Figure 7.3: The Lanczos algorithm.

vectors V are required to be stored as they are only needed for calculating $X \leftarrow V_j S$. If S is known, then this matrix multiplication can be split into several smaller matrix multiplications e.g., $X \leftarrow V_j^1 S^1$, followed by $X \leftarrow X + V_j^2 S^2$ where $V_j = [V_j^1 V_j^2]$ and $S^T = [S^{1T} S^{2T}]$. This would let us avoid storing V_j and execute the computation without overflowing GPU memory. However, we need to run the Lanczos algorithm to calculate S. Since we do not need to store more than two vectors when calculating T_j, we can fit the entire calculation on the GPU and run the Lanczos algorithm until line 9 in the algorithm (Figure 7.3). Once S has been calculated, we rerun the algorithm again, but this time we update X within the loop. We could technically update X using an outer product update every iteration. However, we can improve the computational efficiency by doing an update every k iterations, where k is calculated according to the amount of GPU memory available. Figure 7.4 shows the pseudocode of the modified Lanczos algorithm.

This optimization, while allowing the eigensolver to run on small GPUs, has no effect

Algorithm: Lanczos with no reorthogonalization modified to use less memory
Input: A ($N \times N$ Symmetric Matrix)
v (Initial Vector)
n (Number of eigenvalues/eigenvectors needed)
Output: Θ (Ritz Values)
X (Ritz Vectors)

1 Start with $r \leftarrow v$
2 $k \leftarrow$ (Amount of GPU memory available - $N \cdot n$) / $(N + n)$
3 $\beta_0 \leftarrow \|r\|_2$
4 **for** $j \leftarrow 1, 2, \ldots,$ **until** k
5 $\quad (r, v_j, \alpha_j) = \text{LanczosIteration}(r, \beta_{j-1}, A, v_{j-1})$
10 $\quad \beta_j \leftarrow \|r\|_2$
11 $\quad$ Compute Ritz values $T_j = S\Theta S^T$
12 $\quad$ Test bounds for convergence using Cullum-Willoughby method
13 $\quad$ **if** (converged)
14 $\quad\quad$ Compute Ritz vectors $X \leftarrow V_j S$
15 $\quad\quad$ return Θ, X
16 $\quad$ **end if**
17 **end for**
18 **for** $j \leftarrow k+1, k+2, \ldots$ **until** convergence
19 $\quad (r, v_j, \alpha_j) = \text{LanczosIteration}(r, \beta_{j-1}, A, v_{j-1})$
24 $\quad \beta_j \leftarrow \|r\|_2$
25 $\quad$ Compute Ritz values $T_j = S\Theta S^T$
26 $\quad$ Test bounds for convergence using Cullum-Willoughby method
27 $\quad p \leftarrow j$
28 **end for**
29 **for** $j \leftarrow 1, 2, \ldots$ **until** p
30 $\quad (r, v_j, \alpha_j) = \text{LanczosIteration}(r, \beta_{j-1}, A, v_{j-1})$
35 $\quad \beta_j \leftarrow \|r\|_2$
36 $\quad$ **if** (j mod k == 0)
37 $\quad\quad X \leftarrow X + V_j^k S^k$
38 $\quad$ **end if**
39 **end for**
40 return Θ, X

Figure 7.4: The Lanczos algorithm modified to use less memory.

on the performance on larger GPUs. When the GPU has sufficient memory to fit the whole problem, the execution is exactly similar to the original Lanczos algorithm. However, when the GPU memory is insufficient, we revert to the 2-pass algorithm in order to run the eigensolver. In essence, we are able to achieve both efficiency and portability for this computation.

Moving to video segmentation, this is all the more important because the number of iterations required for convergence increases with the size of the matrix as $O(ln\ N)$ [106]. For video segmentation, N is of the order of 10^7. In such cases, even with multiple GPUs, there is no possibility of being able to fit all the vectors in memory. As we show later, the eigensolver for the video case takes around 2000 iterations, while there is space in the GPUs to store only about 160 vectors. Being able to accommodate the limited memory capacity of GPUs has been critical for extending the technique to video segmentation.

7.5.3 Extension to video segmentation

Given the performance improvements achieved though our work on image segmentation [47], we decided to apply similar techniques to solve the problem of video segmentation. We demonstrate that video object segmentation can utilize techniques similar to image segmentation techniques that are known to work like gPb [119]. In particular, this obviates the need for extensive probabilistic reasoning and/or hyper graph creation that other video segmentation algorithms utilize when dealing with 2D super pixels. In contrast to such techniques, we perform spectral segmentation *at the pixel level*.

Our technique works as follows : We compute boundaries at the frame level independently using intensity, color, texture and motion cues. We use these cues to create an affinity matrix for the entire video. The generalized eigenvectors of the normalized affinity matrix corresponding to the smallest eigenvalues are calculated. A clustering algorithm is run on the eigenvectors and 3D superpixels are obtained. This initial boundary is refined using ultrametric contour maps and a stable segmentation is produced.

Optical Flow

Optical flow is required for two reasons in video segmentation.

1. In order to differentiate pixels within a frame based on their motion.

2. In order to link corresponding pixels in different frames according to the motion.

We use Large displacement optical flow (LDOF) [36] for computing the motion vectors between adjacent frames. LDOF has been shown to be accurate in tracking fast moving objects compared to other trackers. We use the parallel optical flow technique described in Chapter 6 for running on GPUs. LDOF works well in natural videos where fast motion is common (e.g. fast moving hands, legs etc. of people). Fast and accurate optical flow computation is a crucial component for the success of video segmentation.

In addition to computing the optical flow, we also calculate the confidence that the computed flow is correct. If the forward (from frame i to $i+1$) and backward (from frame

$i+1$ to i) flow vectors do not match at any point, then it implies an error in the optical flow calculation or occlusion, both of which are valid reasons to reduce our confidence. We use the following function for forward-backward reliability check and scale the affinities accordingly. The confidence factor is given by

$$C(w,\hat{w}) = 1 - \tanh\left(\frac{|\mathbf{w}+\hat{\mathbf{w}}|^2}{\alpha\left(|\mathbf{w}|^2+|\hat{\mathbf{w}}|^2\right)+\beta}\right) \tag{7.7}$$

where where $\mathbf{w} := (u,v)$ denotes the flow from frame i to $i+1$ and $\hat{\mathbf{w}} := (\hat{u},\hat{v})$ denotes the flow from frame $i+1$ back to frame i. We use $\alpha = 0.01$ and $\beta = 0.5$.

Eigensolver for video segmentation

At the core of our video segmentation algorithm is spectral clustering. The video is represented as a graph with pixels as nodes and interpixel affinities as edge weights. Affinities are computed over a small radius around every pixel and a pixel affinity matrix is computed. In our case, the affinities are calculated between pixels in the same frame and between frames. This is essential for successful clustering of moving regions in different frames. In the absence of inter-frame pixel affinities, the problem just decomposes into computing the eigenvectors of different frames independently and no information is shared. The affinity values computed between pixels in different frames provides the necessary linkage.

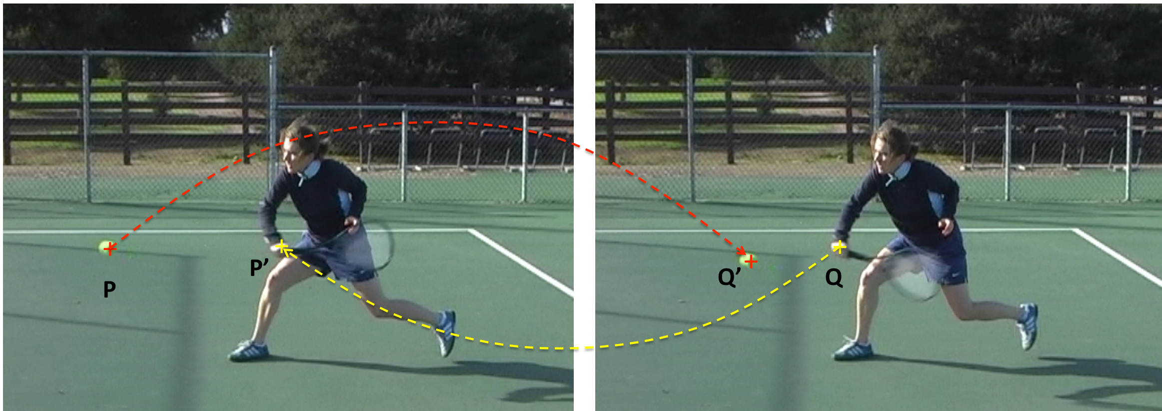


Figure 7.5: Calculating inter frame pixel affinities. Frames 45 and 46 of the Tennis sequence [35] are shown. In order to calculate the affinity between pixels P and Q, we calculate the forward projection of P (Q') and the backward projection of Q (P'). Affinity between P & Q is calculated using equation (7.8).

Figure 7.5 shows how we calculate inter-frame pixel affinities. Simply trying to extend the concept of local gradients and intervening contours to 3 dimensions is problematic for the following reason: While defining affinities based on the intervening contour is possible in images in spite of the (relatively) large neighborhood sizes, it becomes impractical to do

in 3D (2D space + time) because of the explosion in matrix size to impractical limits. We limit this by only calculating affinities between pixels in adjacent frames.

After limiting the pixel affinities to just 2 frames at a time, we define the neighborhood around which affinities are calculated based on optical flow. Figure 7.5 shows how this is done. For a pixel P in frame i, we calculate affinities between P and pixels centered around the point Q' in frame $i+1$. We calculate the affinity between pixel P in frame i and point Q in frame $i+1$ as follows:

$$f(P,Q) = \min(\ f_i(P,P'),\ f_{i+1}(Q',Q)\ ) \tag{7.8}$$

where Q' is the projection of point P in frame $i+1$ using forward flow, P' is the back projection of point Q in frame i using backward flow and $f_i(a,b)$ refers to the single frame affinity between pixels a and b in frame i. $f_i(a,b)$ is calculated using the intervening contour technique that gPb uses. Intervening contour assigns affinities between pixels based on the presence of edges along the line joining the two pixels (strong edge implies low affinity). This affinity is scaled according to the confidence we have on the optical flow vectors (equation 7.7).

The overall affinity matrix W is a symmetric matrix of size = number of pixels in the video. It has the following structure -

$$W = \begin{pmatrix} W_{1,1} & W_{1,2} & 0 & 0 & \cdots & 0 \\ W_{1,2}^T & W_{2,2} & W_{2,3} & 0 & \cdots & 0 \\ 0 & W_{2,3}^T & W_{3,3} & W_{3,4} & \cdots & 0 \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & W_{n,n} \end{pmatrix}$$

where $W_{i,i}$ are intra frame pixel affinity matrices and $W_{i,j}$ are inter frame affinity matrices.

We compute the normalized graph Laplacian of W i.e. $A = D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}$ is calculated from the pixel affinity matrix W. D is a diagonal matrix that equals $diag(W \cdot \mathbf{1})$ [151].

This matrix is then run through the eigensolver to get the eigenvectors corresponding to the n smallest eigenvalues. By construction, the smallest eigenvalue is 0 and the eigenvector corresponding to it is ignored. As noted by [47], the pixel affinity matrices derived from natural images have properties that can be exploited to produce very efficient eigensolvers. In particular, it was noticed that it is very unlikely for these matrices to have multiple eigenvalues converging to the same number and that the eigenvalues are well separated. These properties can be exploited for creating an efficient eigensolver as described in Section 7.5.2.

Parallelization & Implementation details

The video segmentation algorithm was implemented using CUDA for GPU programming and Message Passing Interface (MPI) for cluster programming. As mentioned earlier, the affinity matrix for the video is very large - about 20 GB for 100 frames of 640×480. Almost

all of the optical flow, local gradient and eigensolver computations happen in the GPUs. It is to be noted that the individual GPUs have limited memory (only 3 GB in our case). Hence, we divide the video into smaller sets of frames and assign it to different GPUs. The affinity matrix is partitioned row-wise among the different GPUs. Each GPU stores only a small portion of the matrix and the eigenvectors (corresponding to a few frames each). Every iteration of the Lanczos algorithm involves two forms of communication between the GPUs - (1) all-to-all communication for computing vector dot products and (2) adjacent-node communication for computing Sparse Matrix Vector Multiply ($y = Ax$).

We use a modified version of Sparse Matrix Vector Multiply (SpMV) routine developed by [47] as part of our routine. Because of the way the inter frame pixel affinities are calculated, $W_{i,i+1}$ is not a symmetric matrix. Hence, maximum computational efficiency can be obtained only if we represent $W_{i,i+1}$ and $W^T_{i,i+1}$ in different data strucutres. However, in order to save storage space at the expense of computational efficiency, we use store only $W_{i,i+1}$ and use a slower SpMV for $W^T_{i,i+1}$ computations.

Since we have only one copy of the matrix $W_{i,i+1}$, we resort to using atomic operations on the GPU while performing $W^T_{i,i+1}x$. The downsides of using atomic operations are (1) increase in runtime and (2) introduction of non-determinacy. Non-determinacy is also introduced because of the use of reduction across multiple GPUs in a cluster. Non-determinacy is not an issue if the GPU memory was large enough to hold all the data for the eigensolver iterations i.e. complete the Lanczos algorithm in a single pass. Since that is not the case, non-determinacy can introduce divergence between the two runs of the Lanczos iterations because of different roundoffs in floating point computations. This can be fatal if, for example, at the end of the second run, the solver has not converged even though we have performed as many iterations as the first run. Also, the use of the S matrix from the first run is erroneous as it may not correspond to the new set of iterations. Since it is impractical to remove all sources of non-determinism, we have adopted ways to reduce its effects, in particular, the divergence of different runs. We found that this divergence can be eliminated for all practical purposes by reusing the values of α and β from the first run without recomputing them a second time. This ensured that the eigensolver converged in all cases.

We run the application on a GPU cluster hosted at the National Energy Research Scientific Computing Center (NERSC). Each node has a Nvidia Tesla C2050 having 3 GB of onboard memory and 14 processors (448 Floating point units). Each processing node also has a dual socket quad core Intel Nehalem processor with 24 GB of RAM. We allocated 2-6 frames per GPU depending on the size of the frame and used as many GPUs as needed according to the length of the video processed. It is not necessary to run the program on the cluster and the code could also be run on a single GPU taking proportionally more time while storing the matrix in DRAM. Runtime results are discussed in Section 7.6.

7.5.4 Postprocessing

For globalPb, postprocessing after eigenvector calculation consists of calculating spectral Pb and combining it linearly with local gradients, as shown in Section 7.2. The resulting

oriented edge signal is then maximized over orientation, thresholded and skeletonized. The gPb signal may also be used for computing image segments using oriented watershed and UCM algorithms [9].

Video segmentation requires different postprocessing than image segmentation. We start with the eigenvectors obtained from the eigensolver. There are some problems with using the raw eigenvectors for video segmentation. The number of eigenvectors needed for effective segmentation may increase with the length of the video (Objects entering/exiting, occlusions etc.). In practice, we found that calculating 21 eigenvectors (including the information-less eigenvector with eigenvalue = 0) was sufficient for all the sequences shown in this chapter. Increasing or decreasing this number slightly does not alter the results significantly. Another option is to threshold the number of eigenvectors calculated based on the eigenvalue e.g. calculating all eigenvectors with eigenvalues $\leq 10^{-3}$.

There is also the known problem of leakage i.e. eigenvectors can have smooth transitions leading to the break up of a single object into multiple objects at different time scales. This is clearly illustrated in Figure 7.6. The figure shows how the identity of the person shifts through the sequence as seen from an eigen-perspective. The values of the eigenvector on the person varies between 0.07 and 0.9 (the eigenvector is normalized to be in $[0, 1]$), effectively taking the full range of possible values. In order to avoid this problem, we use ultrametric contour maps for segmentation as a post processing step. This step provides the benefit of combining segment boundaries that are created artificially due to the smooth transitions.

Figure 7.6: Illustration of "leakage" in eigenvectors for long range video sequences. This sequence is one of the sequences in the Motion Segmentation data set [35] and is a scene from the movie "Miss Marple : Murder at the vicarage". **Top** Frames 1, 65, 120, 200 from the sequence. **Bottom** Corresponding portions of the second eigenvector. The vectors are normalized independently for better visualization. However, note the scale difference in the eigenvectors for each of the frames [0.899, 1.00], [0.430, 0.941] , [0.156, 0.293], [0.004, 0.066].

The eigenvectors are clustered into 500 clusters using k-means, giving us an over-segmentation of the video in the form of 3D superpixels. It is essential to perform an oversegmentation at

this stage so that we do not miss any significant boundaries. In order to avoid the k-means algorithm getting stuck in a local optimum, we run k-means clustering 3 times and pick the best clustering as one that has the maximum edge weight along the segment boundaries (computed using the gPb signals of the individual frames). Ultrametric Contour Maps [9] is applied to the resulting supervoxels to cluster them into a small number of regions. This step removes unwanted boundaries created by the k-means clustering due to the smooth transitions in the eigenvectors.

It is important to note that our segmentation algorithm does not rely only upon motion information, but rather motion is just one of the many features used (along with intensity, color and texture). In such a scenario, it is not expected that we will segment only those regions that have different motions. Objects with distinct appearances are segmented even if they do not have motion that is independent of the background or other objects. Depending on the weights assigned to the multiple cues, different features may be emphasized. [119] had performed training using the Berkeley Segmentation Data Set (BSDS) in order to get optimal weights for finding boundaries in images. We use the same weights as [119], and assign the motion features the same weights as the color features. We have seen that this assignment produces good results while still being robust to errors in optical flow (especially along motion boundaries).

It is possible to run a second level cluster processing algorithm along the lines of [35] or [91] to identify affine motion parameters for the different clusters and combine them based on the information. This will reduce the number of final segments obtained while emphasizing differences in motion and de-emphasizing other differences.

7.6 Results

7.6.1 Image segmentation

We perform our accuracy experiments on the Berkeley Segmentation Dataset. The original gPb algorithm achieves an F-score of 0.70 (maximum harmonic mean of precision and recall) on this dataset.

First, we show that using Bresenham lines to perform rotation improves the accuracy of integral image technique compared to using other methods of rotation like nearest neighbor. Figure 7.7 shows that using Bresenham lines reduces the error consistently for computing the χ^2-distances over rectangles at varying angles and for varying areas. We computed the error produced by the rotation by computing the χ^2-distance between the two halves of a square centered at a point and rotated clockwise by an angle of θ where $0 \leq \theta \leq \frac{\pi}{4}$ (Other angles will produce similar results as they can be written as the sum of an angle between 0 and $\frac{\pi}{4}$ and a multiple of $\frac{\pi}{2}$; the error for a rotation by $\frac{n\pi}{2}$ is zero). The patches were obtained from 10 randomly selected points from each of the images in the BSDS[123]. Figure 7.7(a) shows the average relative error in the χ^2-distances using both the rotation techniques when performed over patches with varying sizes, by keeping the angle of rotation constant. Figure 7.7(b) shows the average relative error in the χ^2-distances, and is performed over patches with varying angles keeping the area of the patch constant. For both the figures,

the standard against which the values are compared against is produced by finding the χ^2-distance between the two halves of the rotated square centered around the point defined in the original image.

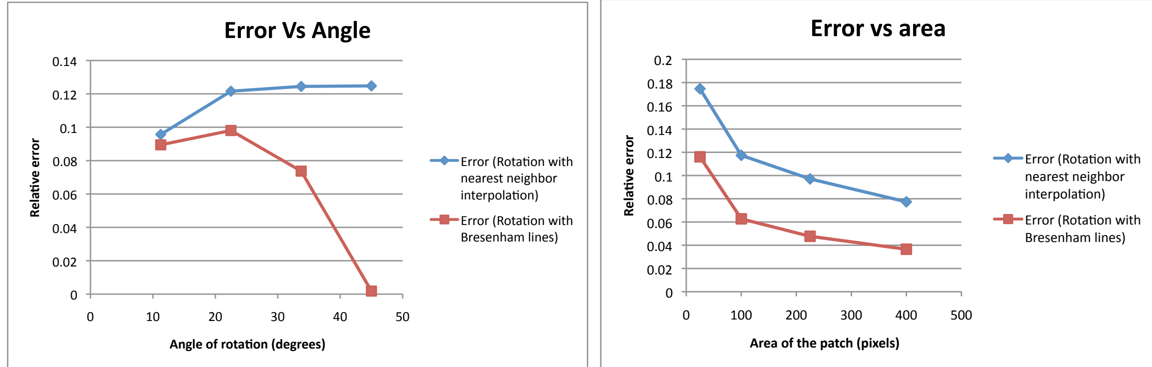


Figure 7.7: Relative errors for computing χ^2-distances over rotated rectangular patches produced by Bresenham rotation vs nearest neighbor rotation (a) Relative error as a function of the angle of rotation (b) Relative error as a function of the area of the patch.

The relative error is not zero even with Bresenham rotation because bresenham rotation is also approximate i.e. its accuracy depends on the angle of rotation & the number of pixels used to define the line.

- Bresenham rotation does not double-count or miss pixels and hence is necessarily better than neartest neighbor interpolation. For other angles, the error is due to the fact that a small segment of the bresenham rotated line does not necessarily have the same angle with zero error. Hence, some error is expected, which should decrease as the length of the side increases.

- For $\theta = \pi/4$, the error due to Bresenham rotation goes to zero. This is because the error caused doe to length truncation effect goes to zero for that angle i.e. a line segment of any length models the angle accurately.

- With increasing window sizes, error in both techniques reduces. This is due to an "area-vs-perimeter" effect - errors that occur in incorrectly counting boundary pixels reduces relative to the number of pixels counted. In the case of using Bresenham rotation, increased accuracy in modeling the angle with longer lines helps in addition to the area-vs-perimeter effect.

Overall, we find that using Bresenham lines for rotation produces a reduction of 44% in the relative error with respect to the nearest neighbor rotation.

Using integral images and Bresenham rotation also improves the runtime of the parallel gPb algorithm. From table 7.1, it is clear that integral images give a 7× performance improvement over the original implementation. In addition, the accuracy of the overall segmentation remains completely unchanged [47].

Local cues	Explicit Method	Integral Images
Runtime (s)	4.0	0.569

Table 7.1: Local Cues Runtimes on GTX 280

Scalability

We ran our detector on a variety of commodity, single-socket graphics processors from Nvidia, with widely varying degrees of parallelism [47]. These experiments were performed to demonstrate that our approach scales to a wide variety of processors. The exact specifications of the processors we used can be found in Table 7.2.

Processor model	Cores (Multi processors)	FPUs per core	Memory Bandwidth GB/s	Clock Frequency GHz	Available Memory MB
9400M	2	8	8.3	1.10	256
8600M GT	4	8	12.8	0.92	256
9800 GX2	16	8	64	1.51	512
GTX 280	30	8	141.7	1.30	1024
C1060	30	8	102	1.30	4096
C2050	14	32	144	1.15	3072

Table 7.2: GPU Processor Specifications

Figure 7.8 shows how the runtime of our detector scales with increasingly parallel processors. Each of the 6 processors we evaluated is represented on the plot of performance versus the number of FPUs. We have two processors with the same number of cores, but different amounts of memory bandwidth, which explain the different results at 240 FPUs. Of these, the 9400M and 8600M are mobile GPUs with very limited memory. The eigensolver requires 500-1000 iterations which require 300-600 MB of space in GPU memory for the eigensolver. Clearly, it would have been infeasible to run gPb on these machines if not for our memory optimizations.

Our work efficiently capitalizes on parallel processors and we see benefits with increasing scaling. We can, however, see that we are hitting the limits of Amdahl's law with the Tesla C2050. Also note that the bandwidth has not scaled significantly between the GTX280 and C2050 and that the C2050 runs at a lower clock rate, which explains the slight performance degradation of C2050 compared to the GTX280.

As shown in [47], our optimizations have not reduced the accuracy of segmentation. We still achieve an F-measure of 0.70 on the Berkeley Segmentation Dataset.

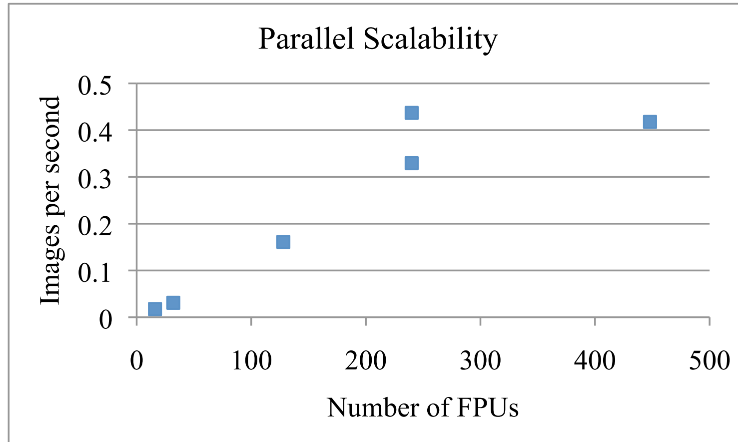


Figure 7.8: Performance scaling with parallelism (0.15 MP images)

7.6.2 Video Segmentation

To the best of our knowledge, there is currently no available literature that does a quantitative comparison between segmentation algorithms based on point trajectories and dense video segmentation algorithms on the same data set. The motion segmentation dataset [35] provides ground truth data for studying both sparse and dense labeling in videos. In particular, the data set has a good number of frames labeled densely in space and time. Although point trajectory segmentation ostensibly does not use visual information but only motion information, it should be noted that optical flow and tracking algorithms themselves utilize visual information. With increasing density of coverage and ability to use incomplete tracks, motion segmentation is getting closer to full video segmentation. We provide quantitative comparison of sparse and dense video segmentation on a single data set. We also compare against other recent video segmentation works like [96, 86].

Comparison to point trajectory based segmentation

We compare our technique against the motion segmentation results from [35] and [139] on the data set from [35]. Comparisons to other sparse motion trajectory classifiers on this data set can be found in [35]. Table 7.3 shows that our technique compares favorably to the sparse trajectory classifier in [35]. The overall pixel error is the number of bad labels over the total number of labels on a per-pixel basis. The overall accuracy is the total number of correctly labeled pixels over the total number of pixels in the video. The average error is computed similar to the overall error, but over regions (not pixels) after computing the error for each region separately. Since the evaluation tool automatically combines segments to avoid high penalties for oversegmentation, the average number of clusters combined is also reported. The total number of objects extracted with $\leq 10\%$ error is also reported.

We achieve comparable errors while labeling 100% of the pixels in the video as compared to only about 3% by sparse segmentation algorithms such as [35, 139]. In absolute terms,

Name	Label density density	Overall (per pixel) error	Overall (per pixel) accuracy	Average (per region) error	Over segmentation	Extracted objects
10 frames (26 sequences)						
Our technique	100%	6.97%	93.03%	19.72%	7.15	20
Brox and Malik	3.34%	7.75%	3.08%	25.01%	0.54	24
ALC incomplete	3.34%	11.20%	2.97%	26.73%	0.54	19
50 frames (15 sequences)						
Our technique	100%	8.42%	91.58%	28.76%	13.0	5
Brox and Malik	3.27%	7.13%	3.04%	34.76%	0.53	9
ALC incomplete	3.27%	16.42%	2.73%	49.05%	6.07	2
200 frames (7 sequences)						
Our technique	100%	15.46%	84.54%	45.05%	5.86	3
Brox and Malik	3.43%	7.64%	3.17%	31.14%	3.14	7
ALC incomplete	3.43%	19.33%	2.77%	50.98%	54.57	0
Hierarchical Graph Segmentation	100%	20.77%	79.23%	40.37%	10.42	0

Table 7.3: Comparison of our video segmentation with other approaches.

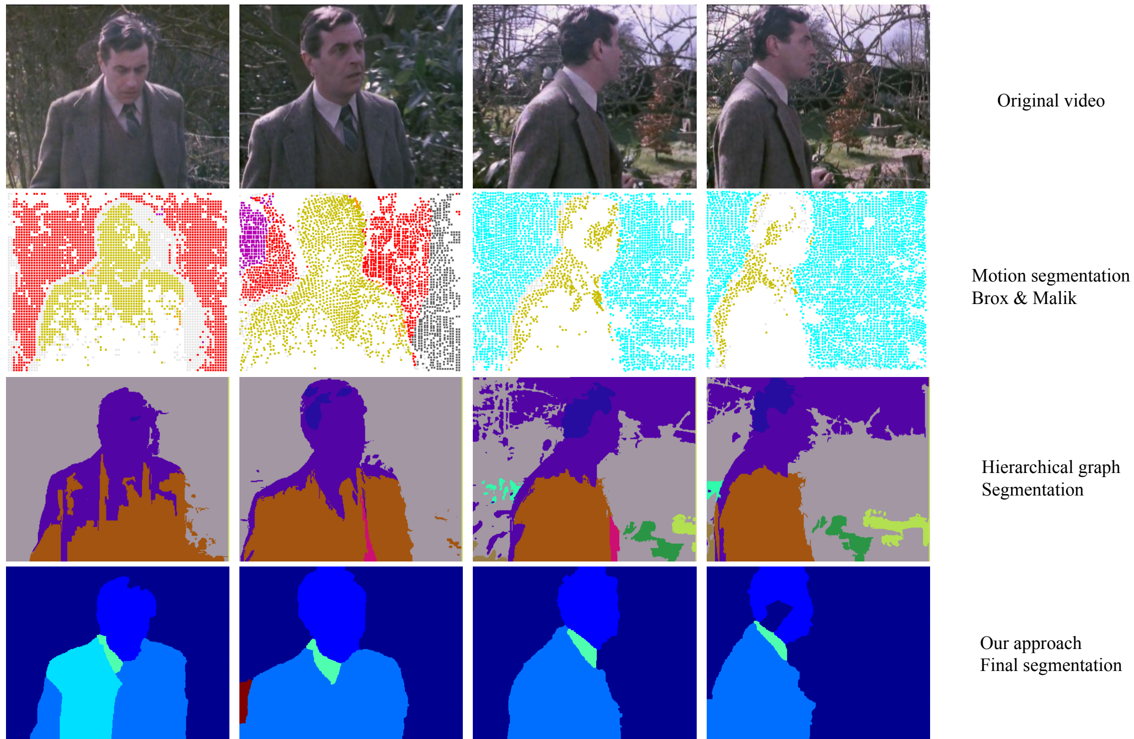


Figure 7.9: Comparison of our technique vs motion segmentation by [35] and [86] on one of the sequences from [35]. **From top to bottom** Video frames(1, 50, 140, 200), Segmentation results from [35], Hierarchical graph segmentation [86], Our final segmentation result. Each tracked point in [35] is shown as 9 pixels large for illustration purposes. Notice that that boundary shifts identity in [35]. Hierarchical graph segmentation produces oversegmentation and non-smooth boundaries. Also note the confusion between the person and the background in the later frames. Figure best viewed in color.

we label far more pixels in the video accurately compared to the sparse techniques. Figure 7.9 shows the results of our approach on one of the sequences in the data set. The motion segmentation results from [35] shifts the identity of the background during the sequence. It is to be noted that the ground truth in this data set is limited to labeling based on motion only. We believe that video segmentation is not limited to just segmentation based on motion, but rather segmentation based on both motion and appearance. Hence, the segmentation we produce almost always produces a certain degree of oversegmentation intentionally.

Comparison to video segmentation algorithms

Table 7.3 also shows that our technique compares favorably to hierarchical graph segmentation [86] on sequences of 200 frames. Both techniques can reduce the per pixel error through oversegmentation; however, from the table it is clear that our technique produces far less oversegmentation (about a factor of 2) with slightly better accuracy. Oversegmentation can be quite problematic for many algorithms that use video segmentation and post processing cannot fix it in many cases. Hence, a technique that produces the same accuracy with less oversegmentation is preferable.

From Figure 7.9, we can also see that hierarchical graph segmentation produces segmentation edges that are not smooth. There is also a tendency to produce segments that are unconnected spatially (even though they are connected in 3D). These artifacts make the segmentation results hard to use for algorithms that require them for editing video or unsupervised learning. Our results conform to the natural image contours and provides mostly spatially connected components. The final segmentations shown are thresholded at 65% of peak for our technique and 90% for hierarchical graph segmentation. Our technique is also able to retrieve 3 objects with less than 10% error compared to 0 objects for hierarchical object segmentation.

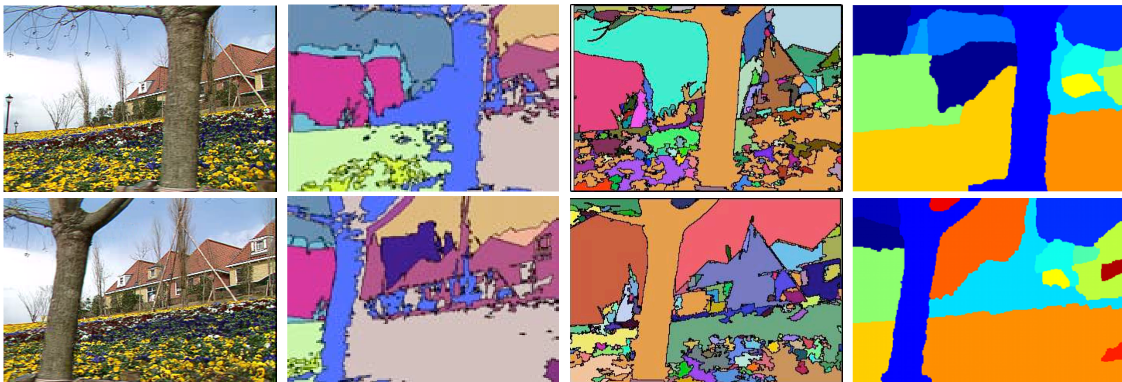


Figure 7.10: Flowergarden sequence (30 frames). **Far Left** Frames 1 and 30 of the sequence. **Left Center** Segmentation results from Hierarchical Graph segmentation [86]. Note the noise in the tree edges and the flower bed. **Right Center** Results from [30]. Notice the identity shift of the house and background. **Far Right** Our results. The segmentation edges are smooth and tracking is consistent.

We compare our technique to video segmentation techniques in [86, 30] on the flower garden sequence in Figure 7.10. Note that we avoid the oversegmentation in the flower bed that [86] and [30] produce due to its highly textured nature. Due to smooth illumination changes on the tree, the segmentation edges from [86] are not smooth. Also notice the presence of multiple "holes" in the segmentation. The result from [30] uses mean shift clustering, and is unstable in noisy and textured regions. The identity shifts of the house and background are other shortcomings of this method. Our result demonstrates the effect of combining image contour detection with optical flow. The segmentation edges are smooth and the noise in the flower bed is removed because of the use of gPb texture gradient processing. Also note that we are able to identify each house as a separate segment.

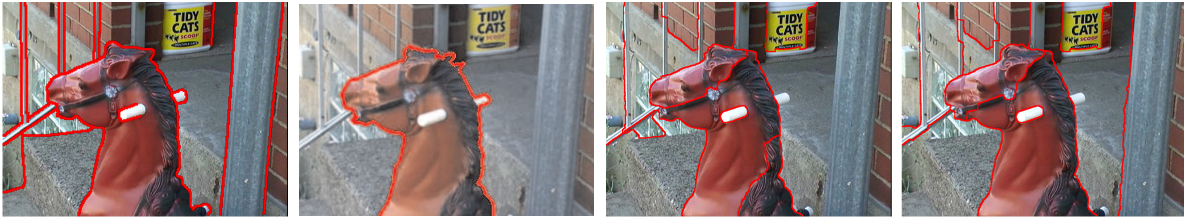


Figure 7.11: Rocking horse sequence (10 frames) [156] **Far Left** Center frame of the sequence with ground truth superimposed. **Left Center** Results from [96]. This requires knowledge of number of segments. **Right Center** Our segmentation results. Note that the ground truth has open edges whereas our method only produces closed edges. **Far Right** Our results with increased weights to motion features. Notice the absence of the edge along the mane and the presence of the occlusion edge along the wall. Figure best viewed in color.

Figure 7.11 compares our segmentation with hypergraph segmentation [96] on the rocking horse sequence from the occlusion data set [156]. This data set contains video sequences of objects with little to no motion. Camera motion is predominant and the objective is to identify occlusion boundaries. Hypergraph segmentation requires knowledge of the number of segments in the video. Our technique does not have this problem. Our method is more robust as we automatically choose our desired segmentation granularity after the processing is complete. Another advantage of our technique is the ability to tailor our result to the choice of segmentation. Since this dataset considers only occlusion boundaries, we doubled the weights of the motion features for the local gradient computation in our method. This emphasizes motion and de-emphasizes other features. The results from the new weights is also shown in the figure. We see that some of the appearance-based edges are removed and occlusion edges are identified.

Runtime analysis

The following runtime analysis is performed on 50 frames of the marple1 sequence on 10 nodes. The total runtime for this sequence on a cluster of 10 machines is 232 seconds. About 50% of the runtime is spent on the eigensolver. The eigensolver takes 108 seconds

for 2000 iterations. A runtime breakdown for the video segmentation problem is shown in Figure 7.12.

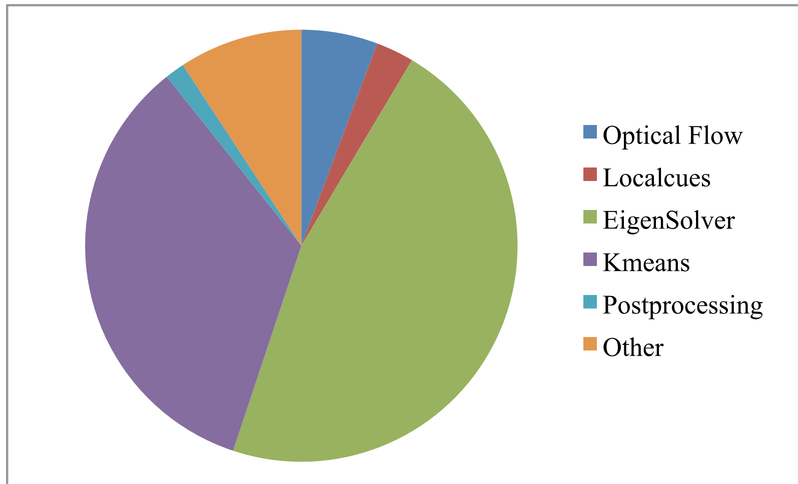


Figure 7.12: Breakdown of the video segmentation runtime.

Among the components of the eigensolver, the SpMV routine is the most compute intensive, taking more than 95% of the eigensolver runtime. As mentioned earlier in Section 7.5.3, the SpMV is composed of both a regular Ax calculation and an irregular $A^T x$ calculation. The $A^T x$ calculation takes about 2.2× as much time as the Ax calculations. More than 86% of the SpMV runtime is spent on the sparse matrix vector multiplication with $W_{i,i}$ and $W_{i,i+1}^T$ in the individual GPUs. The rest of the time in spent on nearest neighbor MPI communication in the cluster since a part of the vector has to be transferred to the neighboring nodes. Figure 7.13 shows the breakdown of the sparse matrix vector multiply computation. Potential future work includes the use of communication avoiding techniques like [93] in order to improve the runtime of the eigensolver.

The total runtime does not change significantly when there are more frames as we scale the number of GPUs along with the length of the video sequence (weak scaling).

7.7 Summary

We discussed how we can parallelize image and video segmentation algorithms. We described how we are able to improve the performance of gPb through algorithmic exploration. We saw how eigensolver improvements have made it portable and thus enabled us to apply it to other problems like video segmentation. We discussed how scalable algorithms for image segmentation and optical flow helped move high quality image segmentation algorithms to video sequences. Our contributions are 4 fold -

1. Showing that using integral images for computing local cues for image segmentation leads to no loss of accuracy while improving runtimes

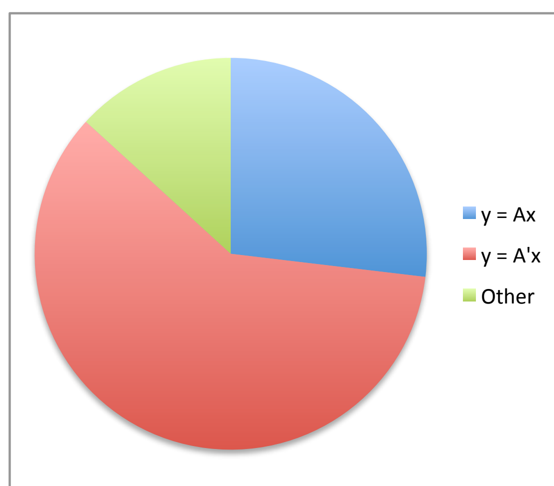


Figure 7.13: Breakdown of the runtime of the Sparse Matrix Vector Multiply routine.

2. Engineering an eigensolver to run with limited memory and making it portable across a wide range of GPU hardware from laptops to clusters
3. A novel technique to combine image segmentation and optical flow in a single framework running efficiently on a cluster of GPUs.
4. Quantitative results on an existing data set comparing the accuracy of our video segmentation technique against existing point trajectory-based classifiers.

We showed how improved runtime capabilities can bring about new applications. For example, we were able to revive video segmentation techniques based on spectral segmentation and show that they can produce better or comparable results to existing techniques. This has been possible only through the improvements to optical flow and image segmentation techniques. Managing large linear algebra computations on a cluster brings about its own set of challenges in terms of data management, non-deterministic behavior etc. Managing these challenges effectively is the key to a successful implementation.

The overall contribution is an exploration of techniques to tradeoff memory with computation and vice versa. For both eigensolver and integral images, proving that we can run the algorithm in two modes (performance or portability) without loss of accuracy was essential. Such techniques are also essential for ensuring scalability. This is one of the biggest challenges of GPU-based programming and overcoming it in a clean way is important for better productivity. For a subset of computations, we can manage GPU memory automatically in an optimal way. This technique is discussed in Chapter 8.

Chapter 8

Memory Management in CPU-GPU systems

In this chapter, we discuss the problem of memory management caused by the limited amount of memory in GPUs. The problem occurs because GPUs have a separate memory hierarchy that is distinct from that of the CPU. This leads to manually managed data transfers between the CPU and GPU, which can be a huge performance bottleneck if not handled properly. We will show how this problem can be solved efficiently for a class of problems that involve data parallel operators using pseudo-boolean optimization and heuristic approaches. We achieve more than 100× reduction in the amount of data transfers and the associated computations are consequently up to 7.8× faster.

8.1 Background and Motivation

In most CPU-GPU systems with discrete GPUs, the CPU and GPU memory systems are distinct and connected through PCI express. The PCI-e bandwidth is typically several times smaller than the internal core-to-memory bandwidth in either systems. For example, on a machine with an Intel Nehalem core i7-920 and an Nvidia GTX 480, the peak memory to core bandwidth for the CPU and GPU are 25.6 and 147.7 GB/s respectively. The peak PCI-e v2.0 bandwidth, on the other hand, is only 8 GB/s. Hence it is preferable to keep large data structures in GPU memory and not move them to CPU DRAM if not required. GPUs include non-user-extensible GDDR5 in order to provide high memory bandwidth to the cores. An unfortunate consequence of this is that the amount of memory is fixed and cannot be increased by the end users even if they want more memory without buying a complete system altogether with more memory. Figure 8.1 gives a schematic of a typical CPU-GPU system.

The working set sizes of applications in use are increasing rapidly. As mentioned in Chapter 2, the data set sizes are doubling in roughly 1.5 years. An important aspect of our work is the focus on applications with data sizes that do not fit within GPU memory. Many interesting applications have data sets that do not fit into the GPU memory. Such applications pose a particular challenge to the programmer, who needs to explicitly break

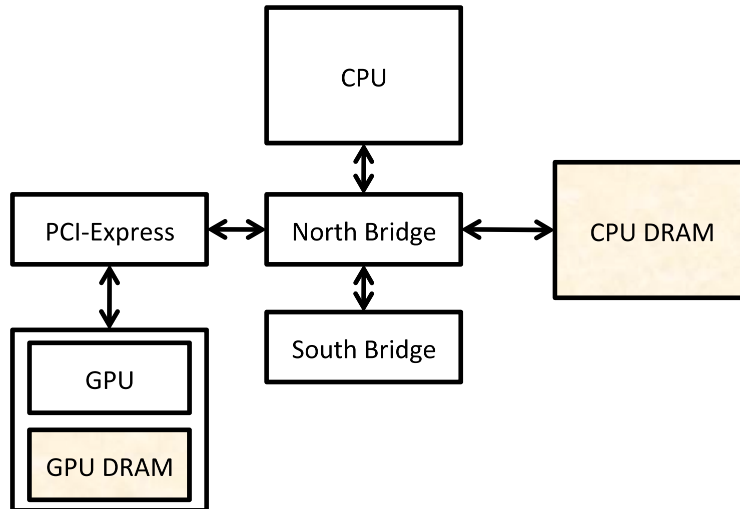


Figure 8.1: Schematic of a typical CPU-GPU system. Both CPU and GPU DRAM are shaded.

down the computations and associated data structures so as to fit within the limited GPU memory, specify the sequence of GPU operations and data transfers, and manage the allocation of GPU memory and the explicit copying of data back-and-forth between the host memory and the GPU memory to achieve correct and efficient execution.

GPU memory capacities vary widely across different platforms (256MB to 6GB). Writing code so that it is portable across all the variants without loss of efficiency is a key challenge that needs to be solved.

8.2 Problem

Many data parallel applications are well suited for acceleration using parallel hardware like GPUs. Even for problems which are suited for GPUs, there is overhead in moving data to and from the CPU. This extra overhead in transferring data can kill any performance improvements from exploiting the parallelism in GPUs if care is not taken to minimize it. In particular, this is apparent when certain portions of a large application are chosen for accelerated performance. In such a scenario, the input and output data for that portion must reside in CPU memory, while the computations happen on the GPU. Any temporary data structures required for this computation need not reside on the CPU. Optimizing this organization and scheduling of data transfers is hard to perform manually. We take Convolutional Neural Networks on large medical images as a leading example for driving the problem. Details of this application are provided in Section 8.5.1.

We describe the memory management challenge in GPUs in the rest of the section.

8.2.1 Challenge 1: Scaling to data sizes larger than the GPU memory

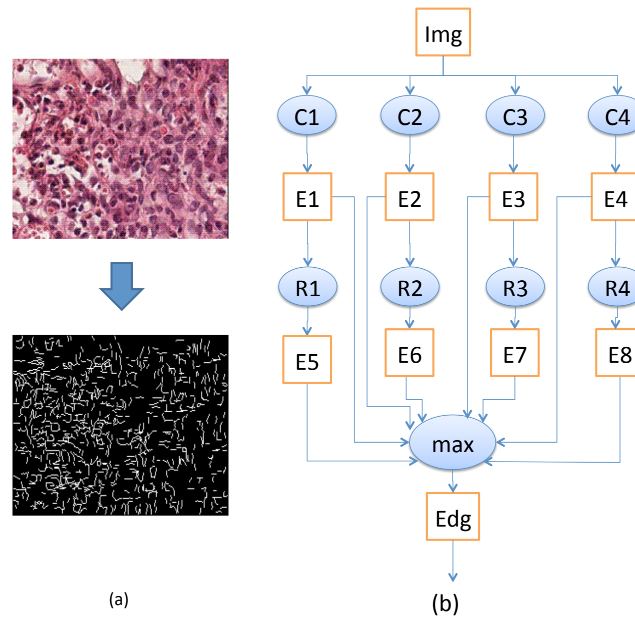
In order to demonstrate the need for scalability to large data sizes, we consider an algorithm for edge detection from images. We apply this edge detection algorithm to extract edges from a high-resolution image that represents a histological micrograph of a tissue sample used for cancer diagnosis [58]. The original image and output edge map are shown in Figure 8.2(a). The algorithm is depicted as a data-flow graph of parallel operators in Figure 8.2(b), where the ellipses represent operators and rectangles represent the input, output, and intermediate data structures used in the algorithm.

Suppose that we would like to execute the edge detection algorithm shown in Figure 8.2(b) on the NVIDIA Tesla C870 GPU computing platform that has 1.5 GB of memory.

Figure 8.2(c) presents the memory requirements for various operators in the the edge detection algorithm as a function of the input image size. The *max* operator has the largest memory footprint (roughly nine times the input image size), while the other operators $C1 - C4, R1 - R4$ have a memory footprint of roughly twice the input image size. The graph is divided up into regions for which different strategies must be used in order to execute the edge detection algorithm within the limited memory of the C870 platform. These regions are separated by the vertical dashed lines, and the corresponding strategies used for executing the algorithm are indicated in the text above the graph.

- For image sizes of less than 150 MB, all the data structures associated with the edge detection algorithm fit in the GPU memory.
- For image sizes between 150 MB and 166.67 MB, the algorithm's memory footprint exceeds the GPU memory. However, the algorithm can be split into two parts - one executes the operators $C1 - C4$ and $R1 - R4$, and the second executes the *max* operator.
- For image sizes between 166.67 MB and 750 MB, the *max* operator itself does not fit in the GPU memory and therefore it needs to be *split*.
- For image sizes between 750 MB and 1500 MB, the operators $C1 - C4$ and $R1 - R4$ also need to be split in addition to the *max* operator.
- For image sizes larger than 1500 MB, the input image itself does not fit in the GPU memory and therefore the entire algorithm needs to be divided to process the input image in chunks.

Clearly, the task of manually writing a *scalable* GPU implementation of the simple edge detection algorithm, *i.e.*, one that can handle input images of various sizes, is quite challenging. The application programmer needs to separately consider all the cases described above, determine the optimal execution strategy for each case, and combine the various scenarios into a single implementation using a framework such as CUDA. Debugging and maintaining such an implementation is likely to be quite a formidable task. In addition to



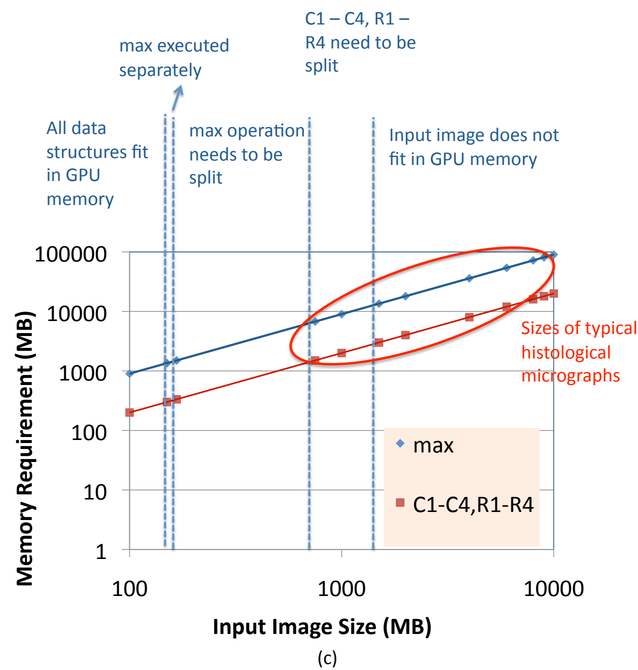


Figure 8.2: Edge detection in histological micrograph images (a) input/output images, (b) Algorithm used for edge detection, and (c) Memory requirements for the edge detection algorithm as a function of the input image size

that, problems arise when the code needs to be executed on another GPU platform that has a different memory capacity. Lower-level frameworks such as CUDA do not address the problem of automatically organizing computations so as to fit within limited GPU memory - this task is left to the application programmer.

Figure 8.2(c) also indicates the typical range of sizes of histological micrograph images that are encountered in the cancer diagnosis application. Clearly, the data sets are much larger than the memory capacities of even high-end GPU computing platforms. Many other applications in Recognition and Mining workloads [68] process large data sets, making this challenge a common and important one to address.

Most of the GPU porting efforts to date typically assume that all the data fits within the GPU memory. While GPU memory capacities are certainly increasing, the rate of increase is much slower than the growth in many of the data sets that need to be processed. In order to achieve high internal memory bandwidth, GPUs use non-expandable GDDR memory that is packaged with the graphics processor in a single unit, and cannot be upgraded by the end user. whereas CPU memory (DDR2 or DDR3) can be easily added to the system by the end user (*e.g.* in the form of DIMM cards).

In summary, a critical need in GPU computing is to address the challenge of executing workloads whose memory footprint exceeds the available GPU memory. A related challenge is to ensure that programs written for today's GPUs and data sets will work efficiently with the data sets and GPU platforms of the future with minimal programmer effort. Finally, it is also desirable that applications can be re-targeted to work on concurrently available GPU platforms that have different memory capacities (*e.g.*, high-end and low-end product variants). This problem has not been addressed in prior work on GPU computing.

An observation that falls out of the above example is that, as data sizes increase, a given computation needs to be divided up into smaller and smaller units in order to fit into the GPU memory, leading to increased data transfers between the CPU and GPU. This leads to the next challenge, namely minimizing the overheads of data transfer between the host and GPU.

8.2.2 Challenge 2: Minimizing data transfers for efficient GPU execution

One of the major performance limiting factors in GPU computing is the limited CPU to GPU communication bandwidth. GPU cards using the PCIe bus achieve a host-to-GPU bandwidth of around 4-5 GB/s, which is much smaller than the internal memory bandwidth of GPU platforms which is over 100 GB/s. This limitation is especially significant for applications that do not fit in the GPU memory and hence need to frequently transfer data structures between the host and GPU memory.

Figure 8.3 presents the breakdown of the time required to perform edge detection on an image of size 8000×8000 with kernel matrices of varying sizes on the NVIDIA Tesla C870 platform. For various kernel matrix sizes ranging from 2×2 to 20×20, the figure presents the breakdown of the total execution time into the time spent in data transfer to and from the GPU memory, and the time spent in computation on the GPU. The data

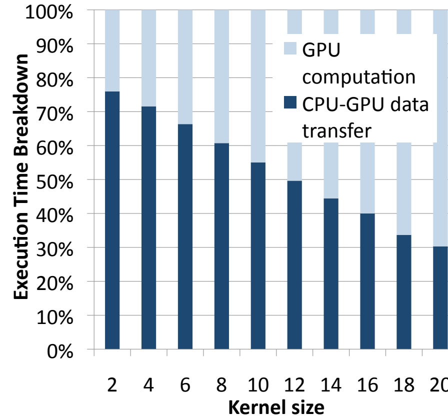


Figure 8.3: Execution time breakdown for executing image convolution operations with varying kernel matrix sizes on a GPU

transfer time varies from 30% of the overall execution time (for large kernel sizes, where more computation is performed per unit data) to 75% of the overall execution time for small kernel sizes. From our experience with recognition applications such as edge detection and convolutional neural networks, we found that operations executed on the GPU generally spend up to 50% of the total runtime in data transfers between the CPU and GPU (where the data starts and ends in CPU memory).

Current GPU programming models offer little help in managing data movement. The programmer needs to explicitly write code to copy data structures between the CPU and GPU, due to the fact that they have separate memory spaces. If all the data structures needed do not fit in the GPU memory, it is essential to manage the allocation of GPU memory over the duration of execution to store the most performance-critical data structures at any given time, to achieve the best possible execution efficiency.

We observe that the data transfer overheads in GPU execution are significantly influenced by the manner in which an application is broken down into computations that are atomically executed on the GPU, and the sequence of execution of these computations. In the context of domain-specific templates that are represented as graphs of parallel operators, operator scheduling has a very significant impact on the total volume of data transferred between the host and GPU.

To illustrate the large impact that scheduling has on data transfer overheads, we consider two alternative schedules for the edge detection algorithm that are shown in Figures 8.4(a) and 8.4(b). For the sake of illustration, we assume that the input image Im is of size 2 units. Note that the convolution, re-mapping, and max operators have been split into two in order to reduce their memory footprints. Therefore, all other data structures $E1', E1'', \ldots, E', E''$ are of size 1 unit each. We assume that the GPU memory capacity is 5 units. Consider the two different operator schedules shown in Figure 8.4(a) and 8.4(b). The schedule shown in Figure 8.4(a) executes the operators on the GPU in the

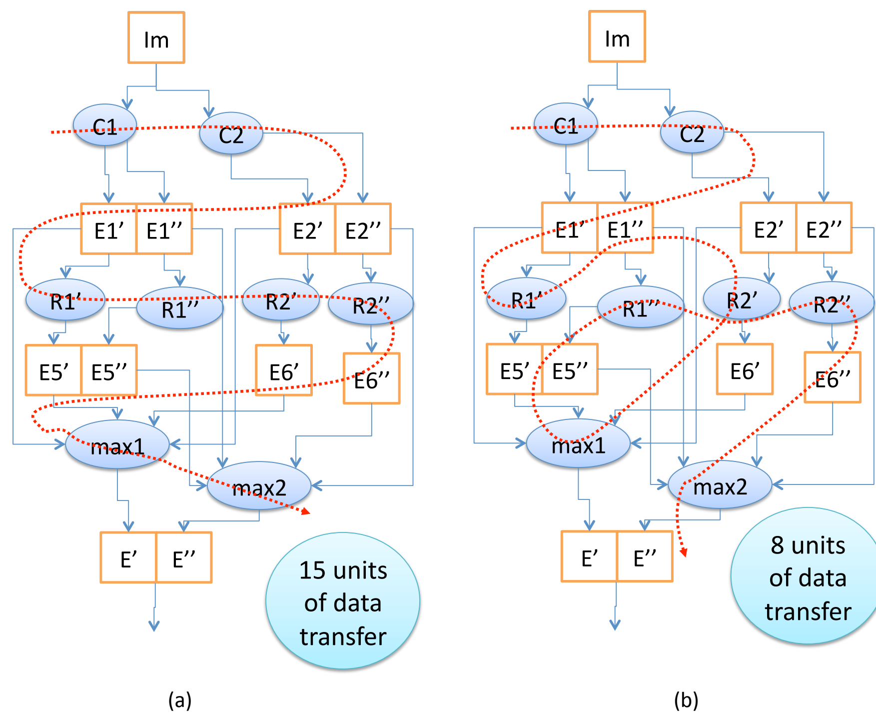


Figure 8.4: Two alternative schedules for edge detection that illustrate the impact of operator scheduling on data transfers

sequence $C_1 \rightarrow C_2 \rightarrow R'_1 \rightarrow R''_1 \rightarrow R'_2 \rightarrow R''_2 \rightarrow max_1 \rightarrow max_2$. It can be shown that this schedule requires 15 units of data transfer between the CPU and the GPU. On the other hand, the schedule shown in Figure 8.4(b) that executes the operators in the order $C_1 \rightarrow C_2 \rightarrow R'_1 \rightarrow R'_2 \rightarrow max_1 \rightarrow R''_1 \rightarrow R''_2 \rightarrow max_2$ requires only 8 units of data transfer. Since data transfers can take over 50% of the execution time, the schedule shown in Figure 8.4(b) will result in a GPU implementation that is much more efficient that the schedule shown in Figure 8.4(a). In general, the optimal schedule depends on the application characteristics (structure of the operator graph and sizes of data structures), and GPU memory capacity. For large applications (we consider operator graphs with thousands of operators), determining an efficient schedule of operators and data transfers is quite challenging for an application programmer.

In summary, GPU execution frameworks should address the challenges of scalability to data sizes that do not fit into the GPU memory and efficient offloading by minimizing the data transfer overheads.

8.3 Related Work

A significant body of work has addressed the development of GPU programming frameworks and tools. The framework that is closest to ours conceptually is Accelerator [163]. Accelerator uses a operation graph representation of programs to map them to GPUs. However, the concerns of Accelerator are very different - it assumes that GPUs are not general-purpose, generates code in shader language, and tries to merge operations aggressively as it assumes that the overhead for a GPU call is unacceptably high. Our framework, on the other hand, does not try to generate low-level GPU code. It instead relies on frameworks such as CUDA. We focus on executing computations that do not fit into the GPU memory, and in managing the CPU-GPU memory transfers efficiently and in a scalable manner.

Recently, work has been done to enable high-level frameworks like Map Reduce on GPUs [116, 49, 90, 149]. There have also been efforts to optimize code for GPUs using source-to-source compilation and auto-tuners [145, 144, 111, 138]. Streaming programming frameworks have been popular for GPUs and several of them have been proposed, notably BrookGPU [41], Peakstream (acquired by Google), and Rapidmind (now, Array Building Blocks [128]). Our framework differs in that it targets applications that do not necessarily fit into the streaming model of computation. Dryad [98] works by creating a generic task graph from the application and managing the scheduling and load balancing on a set of machines. It has been created primarily for the case of managing machines in a cluster. Also, none of the streaming frameworks address the issue of mapping computations to the GPU when the data sizes are too large to fit the GPU memory.

Improving GPU programmability by presenting a unified memory space through hardware techniques has also been proposed. EXOCHI [173] is an attempt to create a unified view by trying to manage both CPU and GPU resources dynamically. CUBA [79] is an attempt to avoid data management between CPU and GPU by letting the GPU access data present in CPU memory directly and allowing it to cache them in GPU memory, thereby providing better programmability through hardware modifications.

Our work can also be viewed as an instance of understanding and exploiting the memory hierarchy present in current systems. In that respect, our work is comparable to Sequoia [105]. Sequoia is a programming language that directly deals with the memory hierarchy problem that we are trying to address. By invoking "tasks" that execute entirely in one level of memory hierarchy, Sequoia tries to optimize programs. However, Sequoia does not provide the mechanisms for reducing the data movement, which is one of our main objectives in our work.

Our framework is complementary to these efforts since we address the problem of organizing computations that do not fit into GPU memory such that data transfer between the host and GPU is minimized.

The problem of minimizing data transfers is related to the problem of register allocation in compilers. Under a fixed instruction schedule, finding the allocation of local variables to hardware registers in order to minimize spillage to memory is similar to our problem of minimizing data transfers under a fixed task schedule. The register allocation problem is known to be NP-complete [72]. Since the data transfer minimization problem is at least as hard as register allocation (register allocation is a special case of data transfer minimization with a fixed operator schedule and similarly sized data structures) and any solution can be verified to take $\leq N$ transfers in polynomial time, our problem is also NP-Complete. Approximate solutions to the register allocation problem have been proposed [81, 72, 114]. The difference between register allocation and our problem is that our data structures have different sizes, whereas registers are of uniform size. In that sense, register allocation is a special case of our problem. Minimizing data transfers to scratchpad memory as studied by [168] is similar to ours. However, that solution works only in the context of a fixed task ordering. For our problem, both the task ordering and data transfer scheduling need to be optimized.

The problem of finding optimal task orderings has been studied in the context of task scheduling for multiprocessors [84, 24, 78, 108, 132]. Most of these approaches minimize the makespan of the task graph for reducing the execution time, and do not minimize data transfers. They primarily optimize for load balancing and do not consider fixed memory-size constraints. Our problem, in contrast, is not to reduce the makespan or load balancing since our task execution is sequential on the GPU. Our aim is to find the optimal task schedule under which the amount of data transfers can be minimized under the fixed memory size constraint on modern GPUs.

8.4 Our approach

8.4.1 Problem specific approaches

For some of the problems with memory management that we encountered before in Chapters 5 and 7, we had to resort to manual solutions that were dependent on the problem being solved. For example, for the SVM classification problem, we manually split the test data so that each portion can be solved independent of the other sequentially, thus minimizing the overhead for data transfers between the CPU and GPU. More details on

this approach are presented in Section 5.4.2.

In the case of the eigensolver in image and video segmentation, the problem was more complicated as the amount of memory required increases with the number of iterations performed. In that case, we resorted to a modified algorithm (Figure 7.4) which performs two passes if the GPU memory is insufficient. In both cases, the solutions were specific to the problem being considered. In the next section, we show that we can solve the memory management problem automatically for a class of problems involving graphs of data parallel operators.

8.4.2 Operator splitting: Achieving scalability with data size

The first challenge is to ensure that the operations can be executed on the GPU regardless of its memory limitation. We assume that the operators are split-able *i.e.*, if the data needed for an operation does not fit in the GPU memory, it can be split (executed on several small portions of its input). Splitting enables us to execute arbitrary sized computations on the GPU, providing scalability. Data parallel operators provide an easy target for splitting. However, our framework can also handle other split-able, but not data parallel operators (*e.g*, convolution and reduction).

The operator splitting algorithm can be summarized as follows:

1. Compute the memory requirements of all operators (sum of sizes of data structures associated with each operator). Note that any operator whose memory requirements are larger than the available GPU memory cannot be executed without any modifications.

2. Split the operators whose memory requirements are greater than the GPU memory. This step ensures feasibility for all operators. When an operator is split, other operators that produce/receive data from the split operator also need to be modified.

3. Perform steps 1 & 2 until it is feasible to execute all operators on the GPU.

We consider convolutions, which are not strictly data parallel operations since they depend on a local neighborhood of points. This results in a need for more intelligent splitting that is dependent on the size of the convolution kernel. For example, a 100×100 matrix convolved with a 5×5 kernel matrix results in an output of size 96×96(ignoring borders). Splitting this operation into two must produce two 100×52 input matrices (not 100×50)and two 96×48 output matrices. This size and offset computation can be done by traversing the split graph from the leaves to the root and inferring the sizes and offsets from the kernel and output sizes.

The ability to split operators can be taken for granted in the case of operators that are data parallel. For other operators, one could provide splitting rules, or hints to the framework to split the input/output data of an operator in specific ways. For example, a large matrix-matrix multiply that does not fit in the GPU memory can be split by breaking up one of the input matrices and the output matrix. In the case of image convolutions, only the image matrix must be split. The convolution kernel matrix (which is also an input) should not be split. Even if an operator is not splitable, our framework is usable as long as this operator fits in the GPU memory.

8.4.3 Operator and data transfer scheduling

Once the operators are split such that every individual operation can be run on the GPU, they will have to be scheduled at a macro level to get the most efficient code. Finding the optimal schedule (in terms of minimal data transfers) given a template whose operators are individually schedulable is the key problem to be solved. We model the problem formally as a pseudo-boolean optimization problem. We also propose a heuristic method to solve the problem that uses a depth-first heuristic for scheduling the operators and a "latest time of use" based data transfer scheduling heuristic. We discuss these methods in detail below.

Formulation as a Pseudo-Boolean Optimization Problem

The problem of optimizing CPU-GPU data transfers can be written as a constraint satisfiability problem. In particular, it is possible to formulate it as a pseudo-boolean (PB) optimization problem. The pseudo-boolean optimization problem is a generalization of the satisfiability (SAT) problem. In a PB optimization problem, the variables are all boolean, the constraints can be specified as linear equalities/inequalities and the objective function to be optimized is a linear function of the variables.
The variables used in our formulation are as follows:
$x_{i,t}$ is 1 if operator i is executed at time step t
$g_{j,t}$ is 1 if data structure j is present in the GPU at time step t
$c_{j,t}$ is 1 if data structure j is present in the CPU at time step t
$Copy_to_GPU_{j,t}$ is 1 if data structure j has to be copied from the CPU to the GPU at time step t
$Copy_to_CPU_{j,t}$ is 1 if data structure j has to be copied from the GPU to the CPU at time step t
$done_{i,t}$ is 1 if operator i has been executed by time step t
$dead_{j,t}$ is 1 if data structure j is not needed after time step t

The following constants are specific to the GPU platform and the template:
D_j is the size of the data structure j
$Total_GPU_Memory$ is the total amount of GPU memory present in the system
$Output$ is the set of all the data structures that are outputs of the template (needed on the CPU)
$IA_{i,j}$ is 1 if data structure j is an input to operator i
$OA_{i,j}$ is 1 if data structure j is an output of operator i

At a given time step t, the following sequence of events are assumed to happen. Any data structure(s) that needs to be copied to the GPU from the CPU memory is copied over. Then, the operator scheduled for execution at time t is executed on the GPU. After the computation is over, any data structure that needs to be copied from GPU to CPU memory is copied over. If any data structure is not needed after time t, it is deleted from GPU memory.

Our PB formulation is given in Figure 8.5. Constraints (1-3) encode the precedence

$$minimize \sum_{j=1}^{J} \sum_{t=1}^{N} (Copy_to_CPU_{j,t} + Copy_to_GPU_{j,t}) \cdot D_j$$

Precedence & Scheduling

(1) $\forall t \sum_{i=1}^{N} x_{i,t} = 1$

(2) $\forall i \sum_{t=1}^{N} x_{i,t} = 1$

(3) $\forall i_1 \to i_2, t_1 > t_2,\ x_{i_1,t_1} + x_{i_2,t_2} \leq 1$

Memory

(4) $\forall t\ \sum_{j=1}^{J} g_{j,t} D_j \leq Total_GPU_Memory$

GPU copy & persistence

(5) $\forall i, j, t\ [IA_{i,j} \vee OA_{i,j}] \wedge x_{i,t} \Rightarrow g_{j,t}$

(6) $\forall i, j, t\ IA_{i,j} \wedge x_{i,t} \wedge \neg g_{j,t-1} \Rightarrow Copy_to_GPU_{j,t}$

(7) $\forall j, t\ Copy_to_GPU_{j,t} \Rightarrow g_{j,t}$

(8) $\forall j, t\ g_{j,t} \Rightarrow g_{j,t-1} \vee Copy_to_GPU_{j,t} \vee [\bigvee_{i=1}^{N} (OA_{i,j} \wedge x_{i,t})]$

CPU copy & persistence

(9) $\forall i, j, t\ OA_{i,j} \wedge x_{i,t} \wedge \neg Copy_to_CPU_{j,t+1} \Rightarrow \neg c_{j,t+1}$

(10) $\forall j, t\ \neg c_{j,t} \wedge \neg Copy_to_CPU_{j,t+1} \Rightarrow \neg c_{j,t+1}$

Initial & Final conditions

(11) $\forall j\ c_{j,0} = 1$

(12) $\forall j\ g_{j,0} = 0$

(13) $\forall j \in Output\ c_{j,N+1} = 1$

Data liveness

(14) $\forall i, t\ done_{i,t} \Leftrightarrow x_{i,t} \vee done_{i,t-1}$

(15) $\forall i\ done_{i,0} = 0$

(16) $\forall j \in Output, t\ dead_{j,t} = 0$

(17) $\forall j \notin Output, t\ dead_{j,t+1} \Leftrightarrow dead_{j,t} \vee [\bigwedge_{i=1}^{N} (\neg IA_{i,j} \vee done_{i,t})]$

(18) $\forall j\ dead_{j,1} = 0$

(19) $\forall j, t\ \neg dead_{j,t} \Rightarrow c_{j,t} \vee g_{j,t}$

Figure 8.5: Pseudo-Boolean Formulation of Offload and Data Transfer Scheduling

and scheduling requirements. There must be only one operation executing on the GPU at any given time. A task which is dependent on other tasks (data dependencies) can only execute after its dependencies are met. Constraint (4) specifies that at any time, the amount of data stored on the GPU cannot exceed the GPU memory. Constraints (5-8) encode the data movement and persistence on the GPU. Specifically, the input and output data required for an operation must exist on the GPU memory before it can be executed. If they do not exist, then they have to be copied to the GPU. For output data, enough space must be reserved before the execution of the operation. Constraints (9-10) specify similar properties for data in the CPU memory. Data resident on the CPU has to be invalidated if it is overwritten by a GPU operation. Data copied to the CPU or GPU remain there until moved, deleted or invalidated. Constraints (11-12) specify the initial conditions (all data resides on CPU, none on GPU). Constraint (13) gives the final condition (outputs must be in CPU memory). Constraints (14-19) specify data liveness. Data that is required in the future needs to be kept live (either in CPU or GPU memory). Otherwise, it can deleted from the system.

We can solve this formulation using PB solvers like MiniSAT+ [69]. However, the number of constraints in this formulation scale as $O(N^2M)$ where N is the number of operators and M is the number of data structures. This method of solving it is feasible only for relatively small problems (up to few tens of operators). For problems containing hundreds or thousands of operators and data structures, solving the pesudo-boolean optimization is practically infeasible. Heuristic approaches are scalable, though may be suboptimal. When the operator schedule is known, the number of constraints in the data transfer scheduling problem scale as $O(NM)$. This problem is sufficiently large that it cannot be solved exactly using pseudo-boolean solvers for large graphs.

Current GPUs have the ability to perform asynchronous data transfer and computation at the same time (as long as they are independent). This can be included in the formulation by changing the objective function to count only those transfers that involve data needed for the current computation. We did not overlap computation and communication in our experiments since it did not improve the performance more than synchronous data transfers.

The pseudo-boolean formulation ignores the fact that that GPU memory can get fragmented. In practice, the $Total_GPU_Memory$ parameter in the formulation is set to a value less than the actual amount of GPU memory present in the system to account for fragmentation.

It has been shown since in [148] that of all the variables shown in the pseudo-boolean optimization problem, $x_{i,t}$ and $Copy_to_GPU_{j,t}$ can be considered as primary optimization variables. The other variables can be expressed as functions of the primary variables. [148] proposes an MILP formulation of the problem that uses $O(E)$ variables and $O(E + N)$ constraints where E is the total number of dependence edges between the operators and data structures (number of edges in the task graph). However, even this formulation could not be directly solved using commercial solvers like CPLEX. These observations motivate taking a heuristic approach.

Heuristic approaches

The data transfer optimization problem can be thought of as composed of two sub-problems - find a good operator schedule, and then, find the optimal data transfer schedule given this operator schedule.

Finding a good operator schedule is a difficult problem. This problem is similar to a data-flow scheduling problem. Since the aim is to maximize data reuse so that we need not transfer things back and forth between the CPU and GPU, we decided to adopt a *depth-first schedule* for the operators. In a depth-first schedule we try to schedule the entire sub-tree belonging to a child of a node before exploring its sibling. If a node cannot be scheduled due to precedence constraints (all its inputs are not ready), we backtrack to its parent and explore its other children. The drawback of the approach is that the operator schedule does not take into account the GPU memory limitations at all. While this heuristic can lead to reasonable results, there is scope for improvement by using information about the available GPU memory.

After a schedule for the operators is obtained, the data transfers can be scheduled. The only constraint to this problem is the amount of GPU memory available. This problem is similar to a cache replacement policy as we have a limited amount of fast memory where we would like to keep as much data as possible. We know that the optimal solution for the cache replacement problem is to replace cache lines that are going to be accessed furthest in the future. Based on this insight, we formulate a data transfer scheduling algorithm as follows:

1. Calculate the "latest time of use" for each data structure (since the operator schedule is known, this can be computed statically).

2. When a data structure needs to be brought into the GPU memory (*i.e.*, it is the input or output of the operator being executed at the current time step), and there is insufficient space, move the data structures that have the furthest "latest time of use" to the CPU until the new data structure can be accommodated.

3. Remove data eagerly from GPU memory *i.e.*, delete them immediately after they become unnecessary.

The solution generated by above algorithm will be optimal for a given operator schedule provided all the data structures are of the same size and are consumed exactly once. If all the data structures are of the same size but are consumed more than once, the problem becomes equivalent to a register allocation problem which is proven to be NP-Complete. Register allocation is the problem of mapping program variables to hardware registers in order to minimize register spilling to memory. In our case, program variables correspond to data structures, register space corresponds to GPU memory and memory corresponds to CPU memory. Register allocation problem directly reduces to our data transfer scheduling problem and hence our problem is NP-Hard. The data scheduling problem is in NP as any solution that claims to take $\leq N$ transfers can be verified in polynomial time and so

the problem is NP-complete. However, in spite of this complexity, we have found that our heuristic works reasonably well in practice.

Better results can be obtained through better decomposition techniques like the one proposed in [148]. It has been shown that once the operator schedule is fixed, solving the data transfer scheduling problem is relatively easy. Commercial solvers like CPLEX can solve the data transfer scheduling problem in less than a second for graphs with thousands of nodes. The operator scheduling problem, on the other hand, is much harder to solve optimally. Hence [148] adopt simulated annealing in order to solve it as opposed to a depth-first heuristic. This can lead to a 44% reduction in the amount of data transfers at the expense of more time spent in scheduling.

8.5 Results

We present the results of using our framework on two different applications that use the templates we described earlier. We performed our experiments on two different systems. One system had a dual socket Quad core (Intel Xeon 2.00 GHz) as our primary CPU with 8 GB of main memory and an NVIDIA Tesla C870 as the GPU. The second system had an Intel Core 2 Duo 2.66 GHz with 8 GB of main memory and an NVIDIA GeForce 8800 GTX as the GPU. The Tesla C870 has 1.5 GB of memory while the GeForce 8800 GTX has 768 MB of memory. Both the GPUs have the same clock frequency (1.35 GHz) and degree of parallelism (128 FPUs) and differ only in the amount of memory. The two systems were running Redhat Linux and Ubuntu Linux, respectively. CUDA 2.0 was used for GPU programming.

For comparison purposes, we propose the following execution pattern as the baseline for GPU execution. For each operator, transfer input data to the GPU, perform the operation and copy the results back to the CPU immediately. There is no persistent storage in GPU memory. It allows any operator to execute on the GPU without any interference from other operators. However, this is suboptimal when all the temporary data structures fit the GPU memory (the optimal solution would be to move only the overall inputs and outputs). Currently, most GPU porting efforts implicitly make the assumption that the GPU memory is large enough to hold all the data. We are interested in large problems where the data does not fit the GPU memory.

8.5.1 Description of Applications

Edge detection

Edge detection is one of the most important image processing operations that is performed as a pre-processing/feature extraction step in many different applications. Computationally, it involves convolving the input image with rotated versions of an edge filter at different orientations and then combining the results by doing a reduction such as addition/max/max absolute value, *etc.* The general template for edge detection is given as

follows :

$$\begin{aligned} edge_map = & find_edges(Image, Kernel, \\ & num_orientations, Combine_op) \end{aligned}$$

This is similar to the template shown in Figure 8.2 where some convolutions are replaced by "remap" (R) operators. We obtained this template from a cancer detection application [58]. Edge detection alone contributes to about 30% of the total runtime of the application.

We performed edge detection using a 16×16 sized edge filter at four different orientations (2 convolutions and 2 remaps) and combined the results using a Max operation as in Figure 8.2. The edge detection template is reasonably small so we can apply the pseudo-Boolean solver to find the optimal solution as described in Section 8.4.3.

We perform our experiments on both small (1000×1000) and large (10000×10000) input images to the template. The results are also shown in Tables 8.1 and 8.2.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are used extensively in many machine learning tasks like handwritten digit recognition, OCR, face detection, *etc.* We obtained our CNN from an application that performs face and pose detection of the driver through a vehicular sensor network. This application is a CNN with 11 layers (4 convolutional layers, 2 sub-sampling layers and 5 tanh layers). CNNs involve a huge amount of computations in the form of data parallel operations. The structure of CNNs is usually very regular and symmetric and allows many different permutations of operations to get to the same result. We restrict ourselves to using simple non-separable 2D convolutions, data parallel additions and tanh operations. In this context, it becomes necessary to order the operations in a convolutional layer (restricting the freedom in scheduling the operations). The CNNs used were constructed based on primitives in the torch5 library [1]. The operations involved in a single convolutional layer (with 3 input planes and 2 output planes) and its transformation are illustrated in Figure 8.6.
The operation that is performed in a single convolutional layer is given below:

$$output[i][j][k] = bias[k] + \sum_{l} \sum_{s=1}^{kW} \sum_{t=1}^{kH} weight[s][t][l][k] * input[dW*(i-1)+s)[dH*(j-1)+t][l] \quad (8.1)$$

where $input[*][*][l]$ is the lth input plane
$output[*][*][k]$ is the kth output plane
$bias[k]$ is the bias value for the kth output
$weight[*][*][l][k]$ is the convolutional kernel used for the lth input and k^{th} output plane
kW is the width of the kernel
kH is the height of the kernel
dW is the subsampling factor along the width
dH is the subsampling factor along the height

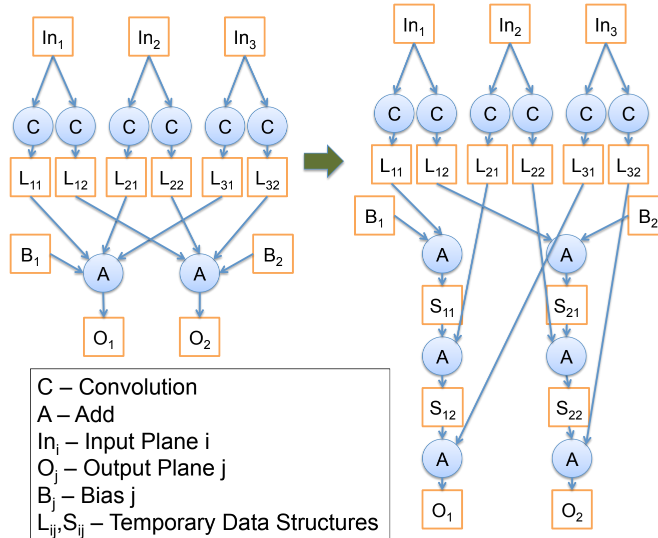


Figure 8.6: Convolutional neural network - layer transformation

We applied our framework to two different CNNs (a small CNN with 11 layers, 1600 operators, and 2434 data structures and a large CNN with 11 layers, 7500 operators, and 11334 data structures). These CNN templates are large enough that the operator scheduling and data transfer optimizations were practically infeasible to solve optimally using the pesudo-Boolean solver. These instances were solved using the heuristics mentioned in Section 8.4.3. They were run on both GPUs with different input image sizes (640×480, 6400×480 and 6400×4800). The baseline GPU execution times are also given in Table 8.2.

8.5.2 Performance improvement and data transfer reduction

Tables 8.1 and 8.2 present the results of optimizing for data transfers between the CPU and GPU for the two different systems. From the results, it is clear that our framework helps in reducing the amount of communication between the host and GPU and that this results in superior performance. Entries in the table with "N/A" indicate infeasible configurations (data exceeds GPU memory) or inconsistent results (due to thrashing).

The final 2 entries in Table 8.2 gave inconsistent results. The large CNN running on GeForce 8800 GTX gives very erratic timing results. The amount of CPU-GPU memory transferred under the optimization is close to the amount of main memory (8 GB) in this case. It turns out that a significant amount of this data is active on the CPU and this leads to thrashing effects in main memory, thereby making the execution time depend heavily on OS effects (like paging and swapping). This was verified by looking at the times actually spent inside the GPU device driver using the CUDA profiler. Execution using our framework spends 51.33 seconds in the GPU driver (13.40 seconds in memcopy), whereas without our

Table 8.1: Reduction in data transfers between the host and GPU memory

Template	Input data size	Total temporary data needed (floats)	Number of floats transferred between CPU and GPU			
			I/O transfers only (lower bound)	Baseline implementation	Optimized for Tesla C870	Optimized for GeForce 8800 GTX
Edge detection	1000x1000	6,000,512	2,000,512	13,000,512	2,000,512	2,000,512
Edge detection	10000x10000	600,000,512	200,000,512	N/A	400,000,512	400,000,512
Small CNN	640x480	59,308,709	4,870,082	157,022,568	4,870,082	4,870,082
Small CNN	6400x480	606,855,749	49,230,722	1,596,371,688	49,230,722	49,230,722
Small CNN	6400x4800	6,261,866,429	501,282,002	16,326,219,528	501,282,002	2,536,173,770
Large CNN	640x480	163,093,609	6,649,882	313,105,568	6,649,882	6,649,882
Large CNN	6400x480	1,686,960,649	67,282,522	3,212,182,688	67,282,522	67,282,522
Large CNN	6400x4800	17,664,611,329	691,377,802	33,262,586,528	760,262,830	7,877,915,800

Table 8.2: Improvements in execution time from optimized data transfer scheduling

Template	Input data size	Time (seconds)			
		Tesla C870		GeForce 8800 GTX	
		Baseline implementation	Optimized implementation	Baseline implementation	Optimized implementation
Edge detection	1000x1000	0.28	0.036	0.19	0.034
Edge detection	10000x10000	N/A	4.12	N/A	3.92
Small CNN	640x480	1.70	0.62	1.21	0.41
Small CNN	6400x480	6.96	2.06	5.95	1.76
Small CNN	6400x4800	54.00	16.66	47.76	20.95
Large CNN	640x480	4.29	2.57	2.94	1.60
Large CNN	6400x480	15.71	6.62	13.96	5.48
Large CNN	6400x4800	262.45	112.99	N/A	N/A

framework it spends 80.20 seconds (52.92 seconds in memcopy). The rest of the time is spent on the CPU.

8.5.3 Scalability

For scalability analysis, we performed experiments using the edge detection template. Figure 8.7 shows the plot of the execution time versus the size of the input image on the Tesla C870 platform. The edge template uses 16×16 kernels. We define the following configuration as the "best possible" - Assume that the GPU has infinite memory and all the operations can be combined into a single optimized GPU kernel call. For the edge template, this corresponds to a single GPU kernel that takes in the input image and produces the output edge map directly. This is the optimal implementation in terms of data transfers (only input and output need to be transferred) and GPU call overhead (only one GPU kernel call). From the figure, it is clear that our methodology provides scalability and produces results that are within 20% of the best possible. Note that the baseline stops working (due to insufficient GPU memory) before the input dimension reaches 8000.

As we can see, using our methodology gives excellent speedups ($1.7\times$ to $7.8\times$) compared

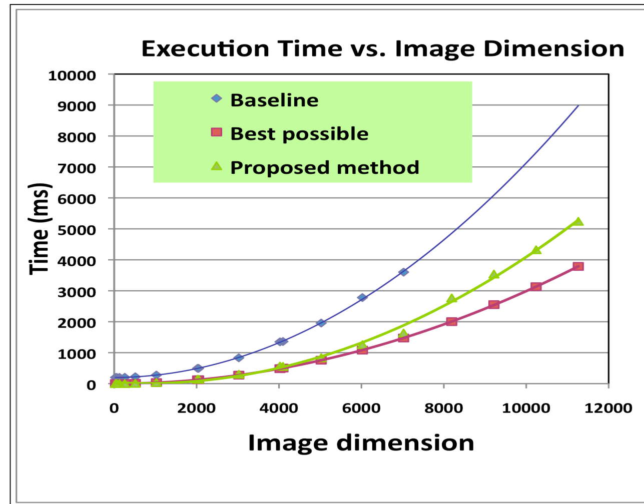


Figure 8.7: Performance of the edge detection template for scaling input data size

to baseline GPU implementations. With GPUs being increasingly used in general purpose computations, the need for software frameworks to hide their programming complexity has increased. Frameworks like ours will be needed in the future to help manage the complexity of heterogeneous computing platforms.

8.6 Summary

We have shown that we can solve the problem of scheduling and optimizing data transfers between the GPU and CPU for graphs of data parallel operators efficiently. If not managed properly, data transfers can have a huge impact on the performance of the application. Given that GPU memory capacities can vary significantly, reducing programmer effort for ensuring portability and efficiency is an important contribution. This is especially useful when adding parallelism incrementally to existing applications using accelerators like GPUs as it is hard to manage the memory manually and consider portability across multiple generations of GPUs with the associated memory capacity limitations. Starting from ad-hoc approaches for specific problems, we formulate and heuristically solve a pseudo-boolean optimization problem that produces up to 7.8× better performance compared to the unoptimized versions. We have shown that solving this problem optimally is possible for small graph sizes and that heuristics can be used to scale to larger problems. Using problem decomposition and simulated annealing, we can get even better results [148].

Looking at computer vision workloads in Chapter 3, it is clear that most applications are highly data parallel. When incrementally parallelizing them, optimizing data transfers will help scale to larger problems easily using existing GPU hardware. Most of the workloads we have seen are heavy on numerical iterative procedures with large memory requirements.

The need to exploit hardware parallelism efficiently without sacrificing productivity and portability is the major reason for the difficulty in using GPUs and other parallel accelerators and our work is a step towards addressing those concerns for helping computer vision domain experts.

Chapter 9

Conclusions

Parallelism in hardware has become inevitable. With increasing computational requirements, it behooves us to create better ways for computer vision experts to reason about parallelism and create tools to make their lives easier. Given the preponderance of numerical computations in computer vision, it is crucial to focus on ways to improve their performance on modern parallel hardware. With CPU-GPU systems currently the major focus of computer vision researchers for parallelism, improving the programmability of such systems is essential.

We saw that parallelism has been the main driver for providing performance improvements since ∼2005 by fitting in more cores with the increased number of transistors on die. The major consequence, as mentioned previously in this dissertation, is that software developers are now forced to be aware of the details of the hardware in order to get good performance for their application. It is no longer the case that improvements in hardware clock speeds would improve the performance of software automatically. Software developers now need to be aware of the presence of hardware parallelism and make their algorithms parallel and scalable, so that they can continue to take advantage of Moore's law.

The main challenges for this to happen are: (1) computer vision researchers' focus on productivity, (2) the fact that efficient numerical algorithms are different for serial and parallel systems, (3) the absence of algorithmic exploration for performance, and (4) GPU's ability to offer more memory bandwidth but with limited memory capacity.

9.1 Contributions

We reiterate and explain our contributions in the following section.

9.1.1 Pattern analysis of computer vision workloads

We have performed a quantitative analysis of patterns used in computer vision applications. Using a randomly chosen set of 50 papers from a recent conference (ICCV 2011), we analyzed the presence of different computational kernels in computer vision workloads. This is important as this has given us a quantitative measure of the relative importance of

different kernels in linear algebra being used in these applications. Among the workloads analyzed, we are able to show that our work on numerical algorithms, optimizations and parallelism is relevant to about 50% of the applications. We have been able to show where the existing gaps are and hence have shown framework developers the set of kernels that need to be developed. These kernels can then help researchers and developers efficiently and productively parallelize and optimize their applications. Unlike previous work, we do not look at just individual applications, but rather at a varied set of different applications. This gives us a "big picture view" of what computations are important for computer vision. This analysis can help computer architects who would like to improve the microarchitecture to better target computer vision, framework developers who can focus on relevant and important numerical computations and computer vision researchers who apply parallelism in their applications.

9.1.2 Improvements to numerical algorithms

We showed numerical optimizations for parallelizing computer vision applications on multi-core and manycore processors in the context of the following applications - support vector machines, optical flow & tracking and image & video segmentation. Each of these applications have unique computational requirements and challenges that needed to be overcome for an efficient implementation on a manycore platform such as a GPU.

In the case of support vector machine training, most of the work was in making sure that the data structures are well-aligned to achieve peak memory bandwidth (as the problem is bandwidth bound). SVM classification, on the other hand, required a reformulation of the distance computations in order to take advantage of high performance BLAS routines that are optimized for the platform at hand. This reformulation was shown to improve performance not just in GPUs, but also in multicore CPUs thus demonstrating that algorithmic changes are essential for scaling and efficiency on all parallel platforms. We showed a speedup of 4–76× for SVM training and 4–372× for SVM classification compared to LIBSVM, a widely used serial implementation.

Optical flow and tracking provided different challenges. We showed how optimizing the linear solver in the inner loop of the large displacement optical flow algorithm can have a significant impact on the overall performance. Solvers used for serial platforms (Gauss-Seidel solver) perform 53× worse compared to optimized solvers for parallel platforms (preconditioned conjugate gradient). Again, we showed that this improvement is not just observed on GPUs, but also multicore CPUs. We also proved that the linear system is positive definite, thus justifying the use of conjugate gradient solvers. We demonstrated a speedup of 37× compared to a serial implementation, which also formed the base of our point tracker. We showed that our point tracker, which relied on accurate optical flow and not complicated integration procedures, performed much better in terms of both accuracy (46-66% better) and density (10-200×) compared to existing trackers. The speed benefits obtained from an optimized parallel implementation of optical flow has meant that the tracker can be used in many different applications.

Image and video segmentation rely on eigensolvers which are their main computational

bottlenecks. While performing algorithmic exploration to improve the performance of image contour detection as shown by [47], it is also essential to ensure that the overall application level accuracy is maintained. This was shown for the optimizations done to local cues computations that used integral images rotated using Bresenham lines and histograms over squares instead of disks. The main bottleneck to the parallelization of the eigensolver on GPUs is their limited memory capacity. Since the amount of intermediate data generated (200-500MB) can far exceed the amount of DRAM on GPUs (as low as 256 MB), we modified the eigensolver to enable it to run with limited memory while not restricting performance on GPUs with more memory. This optimization not only helped us with image segmentation, and enabled us to get a speedup of 130× compared to the original serial implementation, but was also crucial for video segmentation. Given the increased size of the video segmentation affinity matrices (∼20GB), it is essential to be able to run eigensolvers with limited memory usage. Optical flow was used to extend the idea of an affinity matrix to videos. We have shown that it is possible to run eigensolvers on matrices with ∼20 million rows on a cluster of GPUs. Our video segmentation algorithm runs in about 5 minutes for 200 frames with 34 nodes. We demonstrated the feasibility of extending high quality image segmentation techniques to video segmentation using the increased performance provided by parallelism and numerical optimization. The video segmentation algorithm thus obtained has also been shown to be better than other existing video segmentation algorithms.

We have shown that parallelization of existing algorithms brings new capabilities that were not realizable earlier due to the computational burden of the algorithms. In general, improvements to the runtime of algorithms can be utilized for further research in one of two ways: (1) running the same algorithm on a much larger dataset e.g. being able to run large displacement optical flow and tracking on long video sequences and (2) running even more computationally demanding algorithms that build on top of current algorithms e.g. being able to run video segmentation using principles from image segmentation. We have shown that both of these possibilities can be exploited for our applications.

9.1.3 Memory management in CPU-GPU systems

In many of the applications considered, including support vector machines and image & video segmentation, the limited memory capacity of GPUs had been a productivity bottleneck. Even when the details of parallelization are known, it takes a large amount of work to manage the movement of data between CPU and GPU so that all the required computations can happen without exceeding the GPU memory capacity. We have shown that for a limited class of applications that can be expressed as operator graphs, we can automate the process of task and data transfer scheduling. We showed how we can describe this problem as a pseudo-boolean optimization problem and how we can solve it using heuristics. Solving this problem has enabled a speedup of 1.7-7.8× compared to a parallel implementation that does not optimize the movement of data. We believe that this optimization is important for making applications that use large data to be insensitive to changes in the memory capacity of GPUs.

9.2 Future Work and Extensions

There are may open problems in trying to improve the performance and parallelizability of computer vision algorithms. Some of the ways in which our work can be extended and be made more broadly available are as follows -

9.2.1 Integration with frameworks

Memory management algorithms such as the one described in Chapter 8, while very useful, are limited by how they fit in with existing software infrastructure. Here is where data parallel frameworks like Copperhead [45] can help. The Copperhead framework, which is built on a Selective Embedded Just In Time Specialization (SEJITS) principle [46], can be helpful in providing large productivity benefits to software developers while hiding details like memory management from the programmer. It is also much easier to get information about task graphs and data structures when (1) the tasks are generated by the framework and the dependences are known, (2) the sizes of the data structures are known at runtime to the framework. Using heuristics that are fast is essential when the optimization procedure has to be solved during the execution of the application. This would vastly improve the applicability of the memory management algorithms, as we can now use them for applications whose task and data structure sizes are known only at runtime as opposed to compile time.

9.2.2 More coverage

Our optimizations and tools have only managed to cover a subset of the types of computations that computer vision entails. Chapter 3 showed a list of the important kernels that occur commonly in computer vision applications. We have explored algorithmically a few of the computations - linear solvers in optical flow, eigensolvers for image and video segmentation, quadratic programming for support vector machine etc. While these computations do correspond to a significantly large portion of the linear algebra computations that are in use, there are others like solving Conditional Random Fields that are still open. While some work has been done on parallelizing them, we believe there is much more to be done.

9.3 Summary

Since hardware parallelism has become an inescapable route in order to guarantee performance improvments, it has become essential for all software developers to pay attention to the parallelizability of their algorithms. In particular, computer vision software also needs to adapt to a massively parallel world. This dissertation is a step in that direction. We have looked at multiple computer vision applications to see how they can be adapted to multicore and manycore parallelism. In addition to looking at computer vision workloads and characterizing them in terms of patterns, our work demonstrates the utility and importance of adapting numerical algorithms, reformulation for performance and tools for managing mem-

ory in order to take advantage of massively parallel platforms and make computer vision computationally more efficient.

Bibliography

[1] Torch5 library. `http://torch5.sourceforge.net`.

[2] Grand Challenges: High performance computing and communications. The FY 1992 U.S. Research and Development program, 1992.

[3] Sameer Agarwal, Noah Snavely, Ian Simon, Steven M. Seitz, and Richard Szeliski. Building Rome in a day. In *Proc. International Conference on Computer Vision*, 2009.

[4] R. Allen. Compiling high-level languages to DSPs: Automating the implementation path. *IEEE Signal Processing Magazine*, 22:47–56, May 2005.

[5] R. Allen and K. Kennedy. Vector register allocation. *Computers, IEEE Transactions on*, 41(10):1290 –1317, October 1992.

[6] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. GpuCV: an opensource GPU-accelerated framework for image processing and computer vision. In *Proceedings of the 16th ACM international conference on Multimedia*, MM '08, pages 1089–1092, New York, NY, USA, 2008. ACM.

[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[8] Relja Arandjelovic and Andrew Zisserman. Smooth object retrieval using a bag of boundaries. In *Proc. International Conference on Computer Vision*, pages 375–382, 2011.

[9] P. Arbeláez, M. Maire, Charles Fowlkes, and J. Malik. From contours to regions: An empirical evaluation. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2009.

[10] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[11] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[12] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

[13] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, 2000.

[14] S. Baker, D. Scharstein, J.P. Lewis, S. Roth, M.J. Black, and R. Szeliski. A database and evaluation methodology for optical flow. In *Proc. International Conference on Computer Vision*, pages 1–8, Oct. 2007.

[15] Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai-Ahmed. Scientific and engineering computing using ATI stream technology. *Computing in Science Engineering*, 11(6):92 –97, nov.-dec. 2009.

[16] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[18] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[19] Stanley T. Birchfield and Shrinivas J. Pundlik. Joint tracking of features and edges. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2008.

[20] Neil Birkbeck, Dana Cobzas, and Martin Jägersand. Basis constrained 3D scene flow on a dynamic proxy. In *Proc. International Conference on Computer Vision*, pages 1967–1974, 2011.

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[22] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[23] John Bodily, Brent Nelson, Zhaoyi Wei, Dah-Jye Lee, and Jeff Chase. A comparison study on implementing optical flow and digital communications on FPGAs and GPUs. *ACM Trans. Reconfigurable Technol. Syst.*, 3:6:1–6:22, May 2010.

[24] Shahid Hussain Bokhari. On the Mapping Problem. *IEEE Transactions on Computing*, C-30(5):207–214, 1981.

[25] Uday Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI)*, June 2008.

[26] Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston. *Large-Scale Kernel Machines*. The MIT Press, 2007.

[27] Lubomir Bourdev and Jitendra Malik. Poselets: Body part detectors trained using 3D human pose annotations. In *International Conference on Computer Vision (ICCV)*, 2009.

[28] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 1st edition, October 2008.

[29] Steve Branson, Pietro Perona, and Serge Belongie. Strong supervision from weak annotation: Interactive training of deformable part models. In *Proc. International Conference on Computer Vision*, pages 1832–1839, 2011.

[30] W. Brendel and S. Todorovic. Video object segmentation by tracking regions. In *Proc. International Conference on Computer Vision*, 2009.

[31] Jack Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[32] T. Brox, C. Bregler, and J. Malik. Large displacement optical flow. *Proc. International Conference on Computer Vision and Pattern Recognition*, 0:41–48, 2009.

[33] Thomas Brox. Tennis sequence. *http://www.cs.berkeley.edu/ brox/videos/index.html*.

[34] Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High accuracy optical flow estimation based on a theory for warping. In *ECCV (4)*, pages 25–36, 2004.

[35] Thomas Brox and Jitendra Malik. Object segmentation by long term analysis of point trajectories. In *Proc. European Conference on Computer Vision*, 2010.

[36] Thomas Brox and Jitendra Malik. Large displacement optical flow:descriptor matching in variational motion estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, March 2011.

[37] A. Bruhn. *Variational Optic Flow Computation: Accurate Modelling and Efficient Numerics*. PhD thesis, Faculty of Mathematics and Computer Science, Saarland University, Germany, 2006.

[38] A. Bruhn, J. Weickert, C. Feddern, T. Kohlberger, and C. Schnorr. Variational optical flow computation in real time. *IEEE Transactions on Image Processing*, 14(5):608–615, May 2005.

[39] Andres Bruhn and Joachim Weickert. Towards ultimate motion estimation: Combining highest accuracy with real-time performance. In *ICCV '05: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, pages 749–755, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr. Lucas/Kanade meets Horn/Schunck: combining local and global optic flow methods. *Int. J. Comput. Vision*, 61(3):211–231, 2005.

[41] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kavyon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, 2004.

[42] R. H. Byrd, P. Lu, and J. Nocedal. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, 16(5):1190–1208, 1995.

[43] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6), 1986.

[44] L.J. Cao, S.S. Keerthi, Chong-Jin Ong, J.Q. Zhang, U. Periyathamby, Xiu Ju Fu, and H.P. Lee. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Transactions on Neural Networks*, 17:1039–1049, 2006.

[45] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[46] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Katherine Yelick, and Armando Fox. SEJITS: getting productivity and performance with Selective Embedded JIT Specialization. In *First Workshop on Programming models for Emerging Architectures*, September 2009.

[47] Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, and Kurt Keutzer. Efficient high quality image contour detection. In *International Conference on Computer Vision*, September 2009.

[48] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *International Conference on Machine Learning*, July 2008.

[49] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Proc. Third Workshop on Software Tools for Multi Core Systems (STMCS)*, April 2008.

[50] Antonin Chambolle. An algorithm for total variation minimization and applications. *J. Math. Imaging Vis.*, 20:89–97, January 2004.

[51] Trista P. Chen, Dmitry Budnikov, Christopher J. Hughes, and Yen-Kuang Chen. Computer vision on multicore processors: Articulated body tracking. In *International Conference on Multimedia and Expo*, 2007.

[52] Trista P. Chen, Horst Haussecker, Alexander Bovyrin, Roman Belenov, Konstantin Rodyushkin, Alexander Kuranov, and Victor Eruhimov. Computer vision workload analysis: Case study of video surveillance systems. *Intel Technology Journal*, 9(2), May 2005.

[53] Yutian Chen, Andrew Gelfand, Charless C. Fowlkes, and Max Welling. Integrating local classifiers through nonlinear dynamics on label graphs with an application to image segmentation. In *Proc. International Conference on Computer Vision*, pages 2635–2642, 2011.

[54] Sunghyun Cho, Jue Wang, and Seungyong Lee. Handling outliers in non-blind image deconvolution. In *Proc. International Conference on Computer Vision*, pages 495–502, 2011.

[55] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, Cambridge, MA, 2007.

[56] R. Collobert, S. Bengio, and Y. Bengio. A parallel mixture of SVMs for very large scale problems. *Neural Computation*, 14(5):1105–1114, 2002.

[57] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[58] E. Cosatto, M. Miller, H. P. Graf, and J. S. Meyer. Grading nuclear pleomorphism on histological micrographs. In *Proc. Int. Conf. Pattern Recognition*, pages 1–4, 2008.

[59] Timothee Cour, Florence Benezit, and Jianbo Shi. Spectral segmentation with multiscale graph decomposition. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, pages 1124–1131, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Jane K. Cullum and Ralph A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Vol. I: Theory. SIAM, 2002.

[61] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2006.

[62] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[63] D. DeMenthon. Spatio-temporal segmentation of video by hierarchical mean shift analysis. In *Statistical Methods in Video Processing Workshop*, 2002.

[64] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2009.

[65] Zhonghai Deng, Arjan Gijsenij, and Jingyuan Zhang. Source camera identification using auto-white balance approximation. In *Proc. International Conference on Computer Vision*, pages 57–64, 2011.

[66] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 551–556, New York, NY, USA, 2004. ACM.

[67] Jianxiong Dong, Adam Krzyzak, and Ching Y. Suen. A fast parallel optimization for training support vector machine. *Lecture Notes in Computer Science: Machine Learning and Data Mining in Pattern Recognition*, 2734/2003:96–105, 2003.

[68] P. Dubey. A platform 2015 workload model: Recognition, mining and synthesis moves computers to the era of tera. *Intel Technology Journal*, 2007. `ftp://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf`.

[69] N Een and N Sorensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, Jan 2006.

[70] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[71] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005.

[72] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573, 1998.

[73] David G Feingold and Richard S Varga. Block diagonally dominant matrices and generalizations of the Gerschgorin circle theorem. *Pacific J. Math*, 12(4):1241–1250, 1962.

[74] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2), September 2004.

[75] Leonardo V Ferreira, Eugenius Kaskurewicz, and Amit Bhaya. Parallel Implementation of Gradient-based Neural Networks for SVM Training. *International Joint Conference on Neural Networks*, April 2006.

[76] Jan-Michael Frahm, Pierre Georgel, David Gallup, Tim Johnson, Rahul Raguram, Changchang Wu, Yi-Hung Jen, Enrique Dunn, Brian Clipp, Svetlana Lazebnik, and Marc Pollefeys. Building Rome on a cloudless day. In *Proc. European Conference on Computer Vision*, 2010.

[77] James Fung, Steve Mann, and Chris Aimone. OpenVIDIA: Parallel GPU Computer Vision. In *Proceedings of the ACM Multimedia*, pages 849–852, November 2005.

[78] Daniel D. Gajski and Jib-Kwon Peir. Essential Issues in Multiprocessor Systems. *IEEE Computer*, 18:9–27, June 1985.

[79] Isaac Gelado, John Kelm, Shane Ryoo, Steven Lumetta, Nacho Navarro, and Wen mei Hwu. CUBA: An architecture for efficient CPU/co-processor data communication. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, Jun 2008.

[80] Dan B. Goldman, Chris Gonterman, Brian Curless, David Salesin, and Steven M. Seitz. Video object annotation, navigation, and composition. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, pages 3–12, New York, NY, USA, 2008. ACM.

[81] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Journal on Software Practical Experiments*, 26(8):929–965, 1996.

[82] Raghuraman Gopalan, Ruonan Li, and Rama Chellappa. Domain adaptation for object recognition: An unsupervised approach. In *Proc. International Conference on Computer Vision*, pages 999–1006, 2011.

[83] Hans Peter Graf, Eric Cosatto, Léon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade SVM. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 521–528. MIT Press, Cambridge, MA, 2005.

[84] Ronald L. Graham, Eugene L. Lawler, Jan K. Lenstra, and Alexander H.G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. North-Holland, 1979.

[85] Harald Grossauer and Peter Thoman. GPU-based multigrid: Real-time performance in high resolution nonlinear image processing. In *ICVS*, pages 141–150, 2008.

[86] Matthias Grundmann, Vivek Kwatra, Mei Han, and Irfan Essa. Efficient hierarchical graph-based video segmentation. *Proc. International Conference on Computer Vision and Pattern Recognition*, page 8, June 2010.

[87] C. Gu, J. Lim, P. Arbeláez, and J. Malik. Recognition using regions. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2009.

[88] P. Gwosdek, A. Bruhn, and J. Weickert. High performance parallel optical flow algorithms on the Sony Playstation 3. *Vision, Modeling and Visualization*, pages 253–262, October 2008.

[89] P. Gwosdek, A. Bruhn, and J. Weickert. Variational optic flow on the Sony PlayStation 3 - accurate dense flow fields for real-time applications. *Journal of Real-Time Image Processing*, 2009.

[90] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindarajulu, and Tuyong Wang. Mars : A mapreduce framework for graphics processors. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

[91] Xuming He and Alan Yuille. Occlusion boundary detection using pseudo-depth. In *Proc. European Conference on Computer Vision*, ECCV'10, pages 539–552, Berlin, Heidelberg, 2010. Springer-Verlag.

[92] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[93] Mark Frederick Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, Apr 2010.

[94] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185 – 203, 1981.

[95] Mitch Horton, Stanimire Tomov, and Jack Dongarra. A Class of Hybrid LAPACK Algorithms for Multicore and GPU Architectures. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '11, pages 150–158, Washington, DC, USA, 2011. IEEE Computer Society.

[96] Yuchi Huang, Qingshan Liu, and Dimitris Metaxas. Video object segmentation by hypergraph cut. *Proc. International Conference on Computer Vision and Pattern Recognition*, page 8, 2009.

[97] J. J. Hull. A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):550–554, 1994.

[98] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, March 2007.

[99] Kenneth E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[100] Thorsten Joachims. Making large-scale support vector machine learning practical. In *Advances in kernel methods: support vector learning*. MIT Press, Cambridge, MA, USA, 1999.

[101] Thorsten Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 217–226, New York, NY, USA, 2006. ACM.

[102] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Comput.*, 13(3):637–649, 2001.

[103] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[104] Kurt Keutzer and Tim Mattson. A Design Pattern Language for Engineering (Parallel) Software. *Intel Technology Journal, Addressing the Challenges of Tera-scale Computing*, 13(4), 2010.

[105] Timothy J Knight, Ji Young, Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, March 2007.

[106] J. Kuczyński and H. Woźniakowski. Estimating the largest eigenvalues by the power and Lanczos algorithms with a random start. *SIAM J. Matrix Anal. Appl.*, 13(4):1094–1122, October 1992.

[107] Abhijit Kundu, K. Madhava Krishna, and C. V. Jawahar. Realtime multibody visual slam with a smoothly moving monocular camera. In *Proc. International Conference on Computer Vision*, pages 2080–2087, 2011.

[108] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[109] Shang-Hong Lai and Baba C. Vemuri. Reliable and efficient computation of optical flow. *International Journal of Computer Vision*, 29(2), 1998.

[110] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[111] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'09)*. ACM Press, February 2009.

[112] Thomas Leung and Jitendra Malik. Contour continuity in region based image segmentation. In *Proc. European Conference on Computer Vision*, pages 544–559. Springer-Verlag, 1998.

[113] Xiaoye S. Li. Direct solvers for sparse matrices, August 2011. `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf`.

[114] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of Algorithms for Local Register Allocation. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction*, pages 137–152, 1999.

[115] R. Lienhart and J. Maydt. An extended set of Haar-like features for rapid object detection. In *Proc. International Conference on Image Processing*, pages 155–162, New York, USA, 2002.

[116] Michael Linderman, Jamison Collins, Hong Wang, and Teresa Meng. Merge: A programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Mar 2008.

[117] Ce Liu, William T. Freeman, Edward H. Adelson, and Yair Weiss. Human-assisted motion annotation. *Proc. International Conference on Computer Vision and Pattern Recognition*, 0:1–8, 2008.

[118] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Seventh International Joint Conference on Artificial Intelligence*, pages 674–679, Vancouver, Canada, August 1981.

[119] M. Maire, P. Arbeláez, C. Fowlkes, and J. Malik. Using contours to detect and localize junctions in natural images. *Proc. International Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[120] Michael Maire, Stella X. Yu, and Pietro Perona. Object detection and segmentation from joint embedding of parts and pixels. In *Proc. International Conference on Computer Vision*, pages 2142–2149, 2011.

[121] Jitendra Malik, Serge Belongie, Jianbo Shi, and Thomas Leung. Textons, contours and regions: Cue integration in image segmentation. In *Proc. International Conference on Computer Vision*, page 918, Washington, DC, USA, 1999. IEEE Computer Society.

[122] D. Martin, C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using brightness and texture. In *Advances in Neural Information Processing Systems*, volume 14, 2002.

[123] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. International Conference on Computer Vision*, volume 2, pages 416–423, July 2001.

[124] David R. Martin, Charless C. Fowlkes, and Jitendra Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:530–549, 2004.

[125] Amar Mitiche and Abdol-Reza Mansouri. On convergence of the Horn and Schunck optical-flow estimation method. *IEEE Transactions on Image Processing*, 13(6):848–852, June 2004.

[126] MPI. MPI-2.2 Standard. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[127] Qualcomm Developer Network. Fastcv. url=https://developer.qualcomm.com/develop/mobile-technologies/computer-vision-fastcv.

[128] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael D. McCool, Anwar M. Ghuloum, Stefanus Du Toit, Zhi-Gang Wang, Zhaohui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *CGO*, pages 224–235, 2011.

[129] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6:40–53, March 2008.

[130] Nvidia. Compute Unified Device Architecture Programming Guide, 2007.

[131] OpenMP. OpenMP API specification for parallel programs. `http://openmp.org`.

[132] Heikki Orsila, Tero Kangas, Erno Salminen, and Timo D. Hamalainen. Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor SoCs. In *Proc. of the International Symposium on System-On-Chip*, pages 1–4, November 2006.

[133] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pages 276–285, 1997.

[134] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[135] V.K. Prasanna, C.-L. Wang, and A.A. Khokhar. Low level vision processing on connection machine CM-5. In *Computer Architectures for Machine Perception, 1993. Proceedings*, pages 117 –126, Dec 1993.

[136] Electronics.ca Publications. Global machine vision and vision guided robotics market. May 2011. `http://www.electronics.ca/publications/products/Global-Machine-Vision-and-Vision-Guided-Robotics-Market.html`.

[137] Ariadna Quattoni, Michael Collins, and Trevor Darrell. Conditional random fields for object recognition. In *Advances in Neural Information Processing Systems*, pages 1097–1104. MIT Press, 2004.

[138] J. Ramanujam. Toward automatic parallelization and auto-tuning of affine kernels for GPUs. In *Workshop on Automatic Tuning for Petascale Systems*, July 2008.

[139] Shankar Rao, Roberto Tron, Rene Vidal, and Yi Ma. Motion segmentation via robust subspace separation in the presence of outlying, incomplete or corrupted trajectories. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2008.

[140] Nalini K. Ratha, Ani K. Jain, and Diane T. Rover. Convolution on splash 2. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 204–213. CS Press, 1995.

[141] Xiaofeng Ren. Multi-scale improves boundary detection in natural images. In *Proc. European Conference on Computer Vision*, 2008.

[142] Gemma Roig, Xavier Boix, Horesh Ben Shitrit, and Pascal Fua. Conditional random fields for multi-camera object detection. In *Proc. International Conference on Computer Vision*, pages 563–570, 2011.

[143] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, 1998.

[144] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[145] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[146] Peter Sand and Seth Teller. Particle video: Long-range motion estimation using point trajectories. *Proc. International Conference on Computer Vision and Pattern Recognition*, 2006.

[147] Peter Sand and Seth Teller. Particle video: Long-range motion estimation using point trajectories. *International Journal of Computer Vision*, 80(1):72–91, 2008.

[148] Nadathur Satish, Narayanan Sundaram, and Kurt Keutzer. Optimizing the use of GPU Memory in Applications with Large data sets. In *International Conference on High Performance Computing (HiPC)*, December 2009.

[149] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, 2007.

[150] J. Shi and C. Tomasi. Good features to track. In *Proc. International Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[151] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, Aug 2000.

[152] Jianbo Shi and Jitendra Malik. Motion segmentation and tracking using normalized cuts. In *Proc. International Conference on Computer Vision*, page 7, 1998.

[153] Sudipta N. Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. GPU-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[154] Sudipta N. Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, November 2007.

[155] Nathan Slingerland and Alan Jay Smith. Performance analysis of instruction set architecture extensions for multimedia. In *Proc. 3rd Workshop on Media and Streaming Processors*, pages 53–75, December 2001.

[156] Andrew Stein, Derek Hoiem, and Martial Hebert. Learning to find object boundaries using motion cues. *Proc. International Conference on Computer Vision*, page 8, Aug 2007.

[157] Klaus Stüben and Ulrich Trottenberg. *Multigrid methods: Fundamental algorithms, model problem analysis and applications*, volume 960. Springer Berlin / Heidelberg, 1982.

[158] Bor-Yiing Su, Tasneem Brutch, and Kurt Keutzer. Parallel BFS graph traversal on images using structured grid. In *Proc. International Conference on Image Processing*, pages 4489–4492, September 2010.

[159] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense point trajectories by GPU-accelerated large displacement optical flow. In *Proc. European Conference on Computer Vision*, 2010.

[160] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense point trajectories by GPU-accelerated large displacement optical flow. Technical Report UCB/EECS-2010-101, EECS Department, University of California, Berkeley, June 2010.

[161] Narayanan Sundaram and Kurt Keutzer. Long term video segmentation through pixel level spectral clustering on GPUs. In *Third IEEE international workshop on GPUs in Computer Vision (GPUCV)*, November 2011.

[162] Narayanan Sundaram, Anand Raghunathan, and Srimat Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *International Parallel and Distributed Processing Symposium*, May 2009.

[163] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct 2006.

[164] TBB. Thread Building Blocks for open source. `http://threadbuildingblocks.org`.

[165] David Tolliver, Robert T. Collins, and Simon Baker. Multilevel spectral partitioning for efficient image segmentation and tracking. In *Proceedings of the Seventh IEEE Workshops on Application of Computer Vision (WACV/MOTION'05) - Volume 1 - Volume 01*, WACV-MOTION '05, pages 414–420, Washington, DC, USA, 2005. IEEE Computer Society.

[166] R. Tron and Rene Vidal. A benchmark for the comparison of 3-D motion segmentation algorithms. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2007.

[167] Amelio Vazquez-Reina, Shai Avidan, Hanspeter Pfister, and Eric Miller. Multiple hypothesis video segmentation from superpixel flows. *Proc. European Conference on Computer Vision*, page 14, 2010.

[168] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, 2004.

[169] Rene Vidal, Yi Ma, and Sankar Sastry. Generalized principal component analysis. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2003.

[170] Michael Villamizar, Alberto Sanfeliu, and Juan Andrade-Cetto. Computation of rotation local invariant features using the integral image for real time object detection. In *International Conference on Pattern Recognition*, 2006.

[171] Shiv Naga Prasad Vitaladevuni, Pradeep Natarajan, Rohit Prasad, and Prem Natarajan. Efficient orthogonal matching pursuit using sparse random projections for scene and video classification. In *Proc. International Conference on Computer Vision*, pages 2312–2319, 2011.

[172] Cho-Li Wang, P.B. Bhat, and V.K. Prasanna. High-performance computing for vision. *Proceedings of the IEEE*, 84(7):931 –946, jul 1996.

[173] Perry Wang, Jamison Collins, Gautham Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2007.

[174] Andreas Wedel, Thomas Pock, Christopher Zach, Horst Bischof, and Daniel Cremers. An improved algorithm for TV-L1 optical flow. *Statistical and Geometrical Approaches to Visual Motion Analysis: International Dagstuhl Seminar, Dagstuhl Castle, Germany*, pages 23–45, july 2008.

[175] Charles Weems, Steven Levitan, Allen Hanson, Edward Riseman, David Shu, and J. Nash. The image understanding architecture. *International Journal of Computer Vision*, 2:251–282, 1989. 10.1007/BF00158166.

[176] Z. Wei, D. Lee, B. Nelson, and J. Archibald. Real-time accurate optical flow-based motion sensor. In *International Conference on Pattern Recognition*, December 2008.

[177] Manuel Werlberger, Werner Trobin, Thomas Pock, Andreas Wedel, Daniel Cremers, and Horst Bischof. Anisotropic Huber-L1 optical flow. In *Proceedings of the British Machine Vision Conference (BMVC)*, London, UK, September 2009.

[178] Changchang Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). `http://cs.unc.edu/~ccwu/siftgpu`, 2007.

[179] Gang Wu, Edward Chang, Yen Kuang Chen, and Christopher Hughes. Incremental approximate matrix factorization for speeding up support vector machines. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 760–766, New York, NY, USA, 2006. ACM Press.

[180] Kesheng Wu and Horst Simon. Thick-restart lanczos method for large symmetric eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, pages Vol.22, No. 2, pp. 602–616, 2001.

[181] J. Yan and M. Pollefeys. A general framework for motion segmentation: Independent, articulated, rigid, non-rigid, degenerate and non-degenerate. In *Proc. European Conference on Computer Vision*, 2006.

[182] Youtube. Youtube press library, March 2010. `http://www.youtube.com/t/press_timeline`.

[183] C. Zach, T. Pock, and H. Bischof. A duality based approach for realtime TV-L1 optical flow. In *Pattern Recognition - Proc. DAGM*, volume 4713 of *LNCS*, pages 214–223. Springer, 2007.

[184] Christopher Zach, David Gallup, and Jan-Michael Frahm. Fast gain-adaptive KLT tracking on the GPU. *CVPR Workshop on Visual Computer Vision on GPU's (CVGPU)*, 2008.

[185] Luca Zanni, Thomas Serafini, and Gaetano Zanghirati. Parallel software for training large scale support vector machines on multiprocessor systems. *J. Mach. Learn. Res.*, 7:1467–1492, 2006.

[186] Zhengwu Zhang, Eric Klassen, Anuj Srivastava, Pavan K. Turaga, and Rama Chellappa. Blurring-invariant riemannian metrics for comparing signals and images. In *Proc. International Conference on Computer Vision*, pages 1770–1775, 2011.

[187] Ying Zheng, Steve Gu, Herbert Edelsbrunner, Carlo Tomasi, and Philip Benfey. Detailed reconstruction of 3D plant root shape. In *Proc. International Conference on Computer Vision*, pages 2026–2033, 2011.