University of Rhode Island
DigitalCommons@URI

Open Access Master's Theses

2013

Improving Performance of Data-Parallel Applications on CPU-GPU Heterogeneous Systems

Ronald Duarte
University of Rhode Island, rduarte26@gmail.com

Follow this and additional works at: https://digitalcommons.uri.edu/theses

Terms of Use
All rights reserved under copyright.

Recommended Citation
Duarte, Ronald, "Improving Performance of Data-Parallel Applications on CPU-GPU Heterogeneous Systems" (2013). *Open Access Master's Theses.* Paper 48.
https://digitalcommons.uri.edu/theses/48

This Thesis is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu. For permission to reuse copyrighted content, contact the author directly.

IMPROVING PERFORMANCE OF DATA-PARALLEL

APPLICATIONS ON CPU-GPU HETEROGENEOUS SYSTEMS

BY

RONALD DUARTE

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2013

MASTER OF SCIENCE THESIS

OF

RONALD DUARTE

APPROVED:

Thesis Committee:

Major Professor Resit Sendag

Frederick J. Vetter

Gerard M. Baudet

Nasser H. Zawia
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2013

ABSTRACT

Using two full applications with different characteristics, this thesis explores the performance and energy efficiency of CUDA-enabled GPUs and multi-core SIMD CPUs. Our implementations efficiently exploit both SIMD and thread-level parallelism on multi-core CPUs and the computational capabilities of CUDA-enabled GPUs. We discuss general optimization techniques and cost comparison for our CPU-only and CPU-GPU platforms. Finally, we present an evaluation of the implementation effort required to efficiently utilize multi-core SIMD CPUs and CUDA-enabled GPUs. One of the applications, *seam carving*, has been widely used for content-aware resizing of images and videos with little to no perceptible distortion. The *gradient* kernel was improved and achieves over 102*x* speedup on the GPU; this fraction (gradient kernel) of the seam carving operation has largest execution time. The overall resizing operation achieves 32*x* speedup on multi-core SIMD CPU. The time to resize one minute of a 1920x1080 video with seam carving was reduced from 6 hours to 17 minutes on a heterogeneous CPU-GPU system. The second application, *numerical simulations of cardiac action potential propagation* (CAPPS), is a valuable tool for understanding the mechanisms that promote arrhythmias that may degenerate into spiral wave propagation. Our implementation of CAPPS reduces the simulation time from 10 days (single-core implementation) to approximately 4 hours and 8 minutes. This is 54% faster than the execution time of CAPPS on a 60-core CPU-only cluster using MPI. Moreover, our implementation is 18.4*x* more energy-efficient than the 60-core cluster implementation.

ACKNOWLEDGMENTS

This research project would not have been possible without the support of many people. First and foremost, I would like to thank my Lord and Savior Jesus Christ for His everlasting love, enormous help, and perfect guidance. I wish to express my gratitude to my supervisor, Professor Resit Sendag who was abundantly helpful and offered invaluable assistance, support, and guidance. My deepest gratitude are also due to the members of the supervisory committee, Professor Frederick Vetter and Professor Gerard Baudet for their assistance in the success of this thesis work. I would also like to express my gratitude to Professor Jean-yves Hervé for his assistance and support.

I would like to express my gratitude to Deborah Carroll for her assistance and for helping me believe in myself. I would like to convey thanks to Professor Harry Knickle and the University of Rhode Island for providing financial assistance. Finally, I wish to express my love and gratitude to my beloved families; for their understanding & endless love, through the duration of my studies.

DEDICATION

I would like to lovingly dedicate my thesis work to my family and friends. I specially dedicate this thesis and express a special feeling of gratitude to my loving mother Beatriz Sosa, who worked endlessly through multiple jobs, to ensure her children, Thomas, Harold, Fabery, and I were provided for. It is her example that continuously inspires me to strive for greatness. I hope to be a good example for my brothers, and I pray that they may find Jesus and pursuit degrees of their own.

I dedicate this thesis to my wonderful and brilliant wife Karina Luna for her patience throughout the entire master program process, thank you for being one of my strongest supporters.

I would like to express my deepest gratitude to my wonderful grandmother, Mirta Adon Gonzalez, for her love and effort to ensure that my family and I had a better future. I also would like to thank my aunts and uncle, Clara, Michel, Amy, and Robert. Thank you for planting a seed, loving me as one of your children, and for all your support.

I dedicate this thesis to the memory of my father, Juan Pablo Duarte. Although he is no longer with us, I thank him for always believing in me.

I would like to thank my friends and church family who offered their guidance and encouragements from the beginning. I will always appreciate your love and prayers.

TABLE OF CONTENTS

ABSTRACT .... **ii**

ACKNOWLEDGMENTS .... **iii**

DEDICATION .... **iv**

TABLE OF CONTENTS .... **v**

LIST OF TABLES .... **vii**

LIST OF FIGURES .... **viii**

CHAPTER 1 .... **1**

INTRODUCTION 1

CHAPTER 2 .... **4**

THE WORKLOADS 4

2.1 Parboil Benchmarks 5

2.2 Seam Carving 5

2.3 Numerical Simulations of Cardiac Action Potential Propagation 9

CHAPTER 3 .... **12**

HARDWARE RESOURCES 12

3.1 High-performance Desktop Computer (HPDC) 12

3.2 CPU-only Cluster 12

3.3 Energy Measurements 13

CHAPTER 4 .... **14**

IMPLEMENTATION 14

4.1 Optimized Single-Threaded Implementations 14

4.2 Multi-threaded Implementations 17

4.3 SIMD Implementations ... 18

4.4 Using Short Operands: A Case for SIMD Performance ... 20

4.5 GPU Implementations ... 23

4.6 Seam Carving Specific Optimization: The Energy Update ... 25

4.7 Memory Optimization Techniques ... 26

4.8 SPD, SVS, and SSI Optimization Techniques ... 28

CHAPTER 5 ... 33

PERFORMANCE AND ENERGY EVALUATION ... 33

5.1 Performance and Energy Efficiency Evaluation of the Parboil Benchmarks ... 33

5.2 Performance and Energy Efficiency Evaluation of Seam Carving ... 37

5.3 Performance and Energy Efficiency Evaluation of CAPPS ... 42

5.4 Energy Efficiency and Dynamic Voltage-Frequency Scaling ... 45

5.5 Suggested Modification to improve GPU Architectures ... 45

CHAPTER 6 ... 47

AN ANALYSIS OF PROGRAMMING EFFORT ... 47

6.1 Learning Curve for Intel SSE/AVX and CUDA ... 47

6.2 Performance per Effort Hours (PGPEH) Metric ... 48

6.3 Evaluation of Programming Effort ... 49

CHAPTER 7 ... 51

CONCLUSION AND RELATED WORK ... 51

REFERENCES ... 54

BIBLIOGRAPHY ... 59

LIST OF TABLES

TABLE PAGE

Table 1. Characteristic and applications of the studied kernels. 4

Table 2. Description of optimization techniques 15

Table 3. Optimizations applied for CPU and GPU implementations 15

Table 4. Energy and REDP for the overall execution of the Parboil kernels.............. 35

Table 5. Quantification of the programming effort.. 50

LIST OF FIGURES

FIGURE PAGE

Figure 1. The steps of horizontally resizing an image with seam carving. 7

Figure 2. Dependability among pixels in the seam computation. 9

Figure 3. Division of work for Multi-core CPU 17

Figure 4. A histogram of the CES for a 1200x900 image.. 21

Figure 5. A histogram of the CES for a 3648x2736 image.. 22

Figure 6. Proposed Update Algorithm for recomputing the energy............................ 25

Figure 7. Efficient data casting using SIMD shift and insert. 31

Figure 8. The computation-only speedup of the Parboil kernels 34

Figure 9. The overall speedup of the Parboil kernels 34

Figure 10. Energy consumption of parboil kernels for the computation only 34

Figure 11. Performance evaluation of Seam Carving kernels.................................... 38

Figure 12. Performance of full Seam Carving application 39

Figure 13. Performance of best platforms full SC resizing operation........................ 40

Figure 14. Performance of Seam Carving to resize a 1 minute of video. 41

Figure 15. Energy evaluation of Seam Carving to resize 1 minute of video 42

Figure 16. Performance evaluation of DEsolver kernels. 43

Figure 17. Performance evaluation Laplacian kernels. 43

Figure 18. Performance evaluation of CAPPS.. 44

Figure 19. Energy evaluation of CAPPS, includes relative EDP............................... 45

CHAPTER 1

INTRODUCTION

Modern CUDA-enabled GPUs consist of devices with several streaming multiprocessors (SMs) each containing multiple cores (streaming processors). With their high memory bandwidth (compared to low latency as in CPUs), GPUs are ideal for parallel applications with high-levels of fine-grain data parallelism. To hide memory latency, the CUDA architecture supports hundreds of thread contexts to be active simultaneously [1]. CPUs, on the other hand, contain powerful cores that can outperform the GPU's lightweight cores for many applications with poor data parallelism. Today's CPUs not only exploit instruction-level parallelism (ILP) within each core, but also data-level parallelism (DLP) via single instruction multiple data (SIMD) units and thread-level parallelism (TLP) via multiple processors or multi-cores, and simultaneous multithreading (SMT) [2, 3].

The evolution in parallel hardware has let many researchers to explore TLP on multi-core CPUs and DLP on the GPU. Several researchers have also explored DLP on CPUs by utilizing the SIMD units, although much less research has been done. Another approach is to use a combination of GPUs and multi-core SIMD CPUs to explore the true potential of CPU-GPU heterogeneous systems. In this thesis, we evaluate the performance of multi-core SIMD CPUs and CUDA-enabled GPUs using a set of kernels with various characteristics and two full applications that utilize several of these kernels.

Most prior general-purpose GPU (GPGPU) work focus on mapping kernels onto GPUs to evaluate their performance. Although robust mapping of kernels onto tested platforms gives valuable insights into the capabilities and the limitations of the platforms, kernels are often only part of full applications. For fair evaluation, full applications must also be considered to uncover the true potential of the platforms under test. Evaluating the individual kernel performances may be misleading when comparing computing platforms because of the way these kernels may interact with the rest of the full application.

GPGPU research has predominantly focused on accelerating applications. There has been little research in evaluating the energy consumption and energy efficiency of CPU-GPU systems for general-purpose processing. Some recent work [4, 5] evaluate energy efficiency, however, they fall short in terms of a fair comparison between systems because they either only use data-parallel kernels or they do not utilize all the hardware features; in particular, CPU SIMD lanes are often neglected in GPGPU studies. This was addressed in the *debunking the 100x GPU vs. CPU myth* paper [1]. Although performance was compared by applying optimizations appropriate for both GPU and CPU, in [1], energy efficiency was not evaluated. In this work, the kernels and applications are carefully fine-tuned to explore the best utilization of each platform in terms of performance and energy-efficiency.

Finally, what has never been discussed or evaluated in prior work is the implementation effort. It takes significant effort and time to implement fairly good-performing SIMD, multi-threaded and GPU versions of an application. One might ask the following question, is it worth the implementation effort to map a

sequential/parallel algorithm or application onto GPUs or utilize multi-threading and SIMD? Many of us depend on our prior experiences to answer this question. It is, however, important to share experiences, which collectively help other researchers make informed decisions. In this thesis, we attempt to fairly quantify the implementation effort and share our experiences.

Overall, this thesis makes the following contributions: 1) We evaluate and characterize eight kernels and two full applications on CUDA-enabled GPUs and multi-core SIMD CPUs and discuss platform-specific software optimizations and limitations. 2) We demonstrate that GPUs facilitate low-cost and energy-efficient computing for computationally intensive applications, such as *numerical simulations of cardiac action potential propagation* (CAPPS); but also show that applications, such as *seam carving*, achieve best performance and energy efficiency by efficiently utilizing the true heterogeneity of a CPU-GPU system. 3) We show that evaluating the performance and the energy-efficiency of computing platforms by using *only* kernel programs may lead to incorrect conclusions. 4) We demonstrate that reducing the data width has a profound effect on the performance of SIMD implementations. 5) Finally, we quantify the implementation effort in writing the SIMD, multithreaded, and CUDA versions of the applications and define a new metric to compare them.

CHAPTER 2

THE WORKLOADS

In this thesis, we studied eight kernels as listed in Table 1. We used the data-parallel kernels, *mri-q*, *stencil* and *histogram* from the *Parboil* [6] benchmarks, and two full applications, *seam carving* [7] and *CAPPS* (numerical simulations of cardiac action potential propagation) [8] utilizing three and two kernels, respectively, as shown in Table 1. Overall, these benchmarks cover the application domains of image processing, scientific computing and physics simulation, and demonstrate the benefits of SIMD vectorization, multithreading, and general purpose processing on GPUs, as well as their limitations. As we discuss in the following sections, while some of these kernels are relatively easy to parallelize for the underlying platforms, others are either challenging requiring algorithmic changes and careful data layout reorganizations, or not parallelizable due to hardware limitations.

Table 1: Shows the characteristic and applications of the studied kernels: Three kernels from the Parboil [6] benchmarks. Three kernels from seam carving [7] and two kernels from CAPPS [8].

	Kernel	Applications	Characteristics
Parboil [6] benchmarks	mri-q	medical imaging	Compute bound
	dtencil	scientific computation, image processing	Compute bound/ Bandwidth bound
	histogram	image analysis, statistics	Reduction/ synchronization bound
Seam carving [7]	gradient	image analysis, physics simulation	Compute bound/ bandwidth bound
	dynamic programming	many from image processing to bioinformatics	Synchronization bound
	matrix resizing	signal processing	Bandwidth bound
CAPPS [8]	DEsolver	dense linear algebra, scientific computation	Compute bound
	Laplacian	image processing, physics simulation	Compute bound/ bandwidth bound

2.1 Parboil Benchmarks

The Parboil benchmarks are a set of throughput computing kernels useful for throughput computing architecture and compilers research. The benchmarks incorporate diverse memory access and communication patterns. In this work, we characterize and evaluate the performance of the *mri-q*, *stencil*, and *histogram* kernels. The mri-q Kernel [9] computes a matrix Q, representing the scanner configuration for calibration, used in a 3-D magnetic resonance image reconstruction algorithm in non-Cartesian space. The stencil kernel is an iterative Jacobi stencil operation on a regular 3-D grid. Finally, the histogram kernel computes a moderately large, 2-D saturating histogram with a maximum bin count of 255. Input datasets represent a silicon wafer validation in which the input points are distributed in a roughly 2-D Gaussian pattern. For a more detailed description about the Parboil benchmarks, refer to [6].

2.2 Seam Carving

One of the most popular uses of diverse mobile devices today is for browsing images and playing videos. However, different devices have different resolution capabilities, so it is necessary to resize images and videos efficiently and effectively to fit them into diverse displays (such as cell phones, tablets, desktop displays, etc), preferably without distortion. Traditional image resizing techniques are oblivious to the content of the image when changing its width or height. Cropping [10-14] has been one of the most popular approaches to resize images. However, cropping may lose an unacceptable amount of visual information when important structures lie at all edges of an image. In addition, it can only remove information, but it cannot add information

to expand the image. Scaling methods, with or without interpolation, tend to produce distorted images, especially when an image is scaled in one dimension.

Avidan and Shamir [7] developed a new approach to image and video resizing, called *seam carving*. Seam carving functions by establishing a number of seams (paths of least importance) in a digital media and automatically removes or inserts seams to resize the media. This popular content-aware resizing method has been shown to effectively resize images and videos with little to no perceptible distortion [7]. Seam carving has been widely adapted by popular graphics editing applications, which include Adobe Photoshop, where it is called Content Aware Scaling (CAS) [15], or Liquid Scaling in GIMP [16], digiKam [17], and ImageMagick [18]. The importance of content-aware image resizing has made seam carving a popular application for research [19-22].

Seam carving has three phases: The energy function, the seam computation, and the removal or duplication of low-energy seams. First, the energy of each pixel is computed using the magnitude of the gradient (*gradient* kernel). Then, low energy paths, called seams, are marked using dynamic programming (*dynamic programming* kernel). Finally, low-energy seams are duplicated or removed from the image/video to perform the resizing (*matrix resizing* kernel). Figure 1 shows the steps of horizontally resizing an example image.

The three kernels in the seam carving algorithm make it an excellent application for evaluating the performance and energy-efficiency of CUDA-enabled GPUs and multi-core SIMD CPUs because of their very different characteristics. A brief description of these kernels follows.

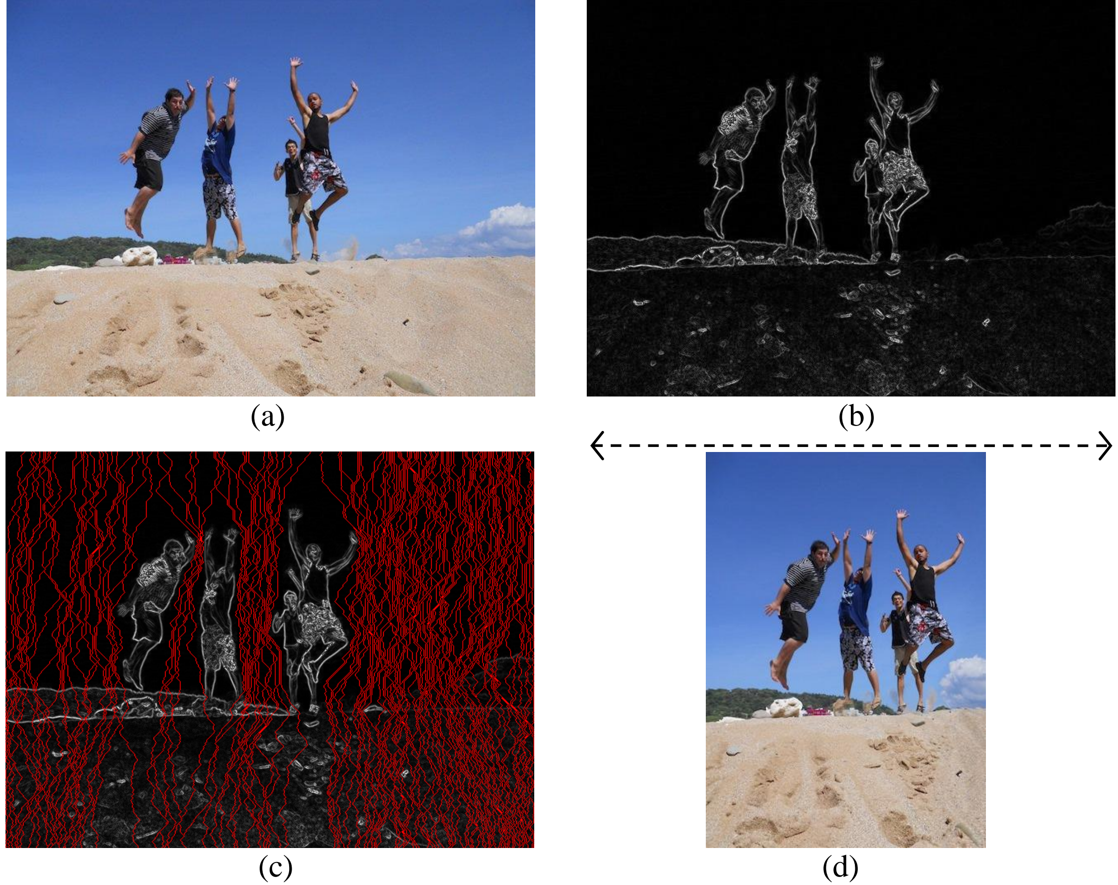

Figure 1: The steps of horizontally resizing an image. (a) The original image. (b) The gradient (energy function) of the image. (c) The low-energy removable seams and the gradient image. (d) The output image, horizontally resized by one half of the original width using seam carving.

Image Gradient

Seam carving is able to utilize several energy functions [7]. In this thesis, we use the magnitude of the gradient [23] for the computation of the energy function because the gradient is a highly used kernel; therefore, the characterization and any improvements of the gradient operation will benefit a large range of applications in image processing [24-26]. The gradient is the directional change in the color or intensity in an image. The magnitude of the gradient can be computed using Equation 2. The components of the gradient vector (Equations 3 and 4) themselves are linear operators, but the magnitude of the gradient is not because of the squaring and square root operations. The implementation of Equation 2 is very computationally intensive.

Therefore, a common practice is to approximate the magnitude of the gradient by using absolute values instead of squares and square roots [23], as in Equation 1. The gradient is preserved only for multiples of 90° when approximated. These results are independent of whether Equation 1 or 2 is used, so nothing of significance is lost in using the simpler of the two equations [23].

$$E = \|\vec{\nabla I}\|_1 = \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right| \tag{1}$$

$$\|\vec{\nabla I}\| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} \tag{2}$$

To reduce the gradient computation of RGB images, the pixels are averaged before computing the gradient. Computing the gradient of each RGB channel separately, and then averaging the results, requires two additional gradient operations. The first norm of the gradient (1) is quite effective and has vast data parallelism, which allows the computation to be perfectly separable. We use Equations 3 and 4 to compute the gradient vectors, which are the x and y derivatives.

$$\frac{\partial I(x,y)}{\partial x} = I(x+1,y) - I(x-1,y) \tag{3}$$

$$\frac{\partial I(x,y)}{\partial y} = I(x,y+1) - I(x,y-1) \tag{4}$$

Dynamic Programming

In the second phase of seam carving, we use dynamic programming to compute the cumulative energy sum of every pixel. The last row of the seam matrix contains the total energy of the seams. The seam matrix denotes the result of seam computation

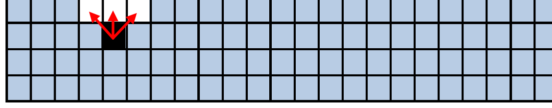

Figure 2: Seam map example. Illustrates the dependability among pixels.

and contains the computed seams that are generated using Equation 5. The first row of the seam matrix is directly obtained from the first row of the *gradient*. This dynamic programming approach produces the optimal seam [7]. However, the computation for each element is entirely dependent on the result of the three above 8-connected elements, as shown in Figure 2. This introduces a higher degree of difficulty for the parallelization of the dynamic programming kernel.

$$S_{i,j} = \begin{cases} E_{i,j}, & i = 0 \\ E_{i,j} + \min\left(S_{i-1,j-1}, S_{i-1,j}, S_{i-1,j+1}\right), & i > 0 \end{cases} \quad (5)$$

Matrix Resizing

The last phase of seam carving is the removal or duplication of low-energy seams and thereby resizing the image. Matrix resizing is widely used in signal processing. *Matlab* [27] has resizing functions based on removing columns and rows of a matrix. Accelerating and characterizing matrix resizing will benefit many applications. For a detailed description about seam carving, refer to [7].

2.3 Numerical Simulations of Cardiac Action Potential Propagation (CAPPS)

Numerical simulations of electrical activity in the heart (specifically propagation of cardiac action potential) are valuable tools for understanding the mechanisms that promote arrhythmias that may degenerate into spiral wave propagation. In [8], the author characterized the convergence properties and numerical stability of a recent

model of the rat ventricular action potential. A model of rat cardiac myocyte action potential [28] with changes from [29] was used in Equation 6. In Equation 6, V_m is the transmembrane voltage, C_m is the membrane capacitance, D is the conductivity of myocardium, I_{ion} is the transmembrane current, and I_{stim} is the stimulus current applied to the cell. The action potential model is used to solve for I_{ion}, and V_m by numerically integrating Equation 6.

$$C_m \frac{\partial V_m}{\partial t} = D \frac{\partial^2 V_m}{\partial x^2} - I_{ion} + I_{stim} \tag{6}$$

In [8], the authors analyzed the numerical convergence of a 1D model on a 1500 μm fiber over a range of uniform spatial steps. The model was then extended to two dimensions for simulating reentrant spiral waves on a plane consisting of 300x300 nodes (9x9 mm^2 surface). The equations were numerically integrated using the explicit Euler technique. The time step was adaptively changed from 100 to 1 ns to insure stability of the integration. The main steps are to compute the transmembrane currents and voltages for all 90,000 nodes. A detailed description about the transmembrane current [28] is beyond the scope of this thesis. It is important, however, to mention that *CAPPS* utilizes the *DEsolver* kernel to solve the action potential model for the transmembrane current, which includes massive amount of computations for solving 25 differential equations (47 *exp*, 320 *mul/div*, 253 *add/sub*, 7 *power*, 2 *log* = 629 floating-point operations per node). The *Laplacian* is implemented using Equation 7 and the results are used to compute the transmembrane voltage, as in Equation 8.

$$Laplacian = \frac{V_{(x,y+1)} + V_{(x,y-1)} + V_{(x+1,y)} + V_{(x-1,y)} + 4 \cdot V_{(x,y)}}{\partial x \cdot \partial y} \tag{7}$$

$$V_{x,y} = \frac{D * Laplacian - Iion_{x,y} + Istim_{x,y}}{C} * \partial t + V_{x,y} \tag{8}$$

CHAPTER 3

HARDWARE RESOURCES

3.1 High-performance Desktop Computer (HPDC)

The HPDC is a heterogeneous CPU-GPU computer composed of a single Intel Core i7-2600k CPU [3] and an NVIDIA GTX580 GPU [30]. The GTX580 has 512 cores organized into 16 SMs with dual warp schedulers. A warp consists of 32 parallel threads executing in lockstep [31]. Ubuntu Linux 10.04 is the operating system installed. The system has 4GB of DDR3 memory and the GTX580 has 1.5GB of GDDR5 memory. The CPU threading model is POSIX threads (*pthread*). The Intel SSE4.2 and AVX (floating-point only) intrinsic instructions are used to write the CPU SIMD code. All implementations on this system were compiled with full optimization using the gcc 4.7 and/or nvcc included in the CUDA SDK 4.2.

3.2 CPU-only Cluster

A 60-core cluster computer was utilized for the numerical simulation. The cluster contains a total of 176 GB of memory. The cluster consists of 18 Intel Xeon 5160 dual-core, 2 Intel Xeon X5355 quad-core, and 4 Intel Xeon X5460 quad-core CPUs. The cluster is organized into 12 individual shared memory systems (9 with 4 cores and 3 with 8 cores). A network connects the 12 systems to form a larger distributed-memory system. The Ubuntu Linux 11.04 operating system is installed on all 12 machines. The interprocess communication was managed by the message-passing

interface (MPI). The numerical simulation for this system was compiled with full optimization using the gcc 4.5.

3.3 Energy Measurements

Energy and power measurements are taken by a digital power meter, which is connected to the wall outlet, and feeds the computing platform being tested. Power data is recorded periodically as kernels are running and then used to compute the energy consumption. We have not subtracted energy consumption of the idle system, so the energy values include the idle system energy. We compare the energy efficiency of tested platforms using energy-delay product (EDP) as a metric.

CHAPTER 4

IMPLEMENTATION

The platform-specific software optimizations presented in this section are critical to fully utilize the compute/bandwidth resources on CPUs and GPUs. Multithreading, reorganization of memory access patterns, and SIMD optimizations are the key for best performance in the CPU. For GPUs, global inter-thread synchronization is very costly and must be minimized. For best performance, user-managed and texture caches must be used efficiently and uncoalesce memory accesses must be minimized. Table 2 and 3 list platform-specific software optimization techniques and the kernels and applications using them, respectively. All optimizations are applied to the baseline single-threaded implementations. The performance numbers of the baseline implementations are on par or better than best reported numbers for each particular kernel or application.

4.1 Optimized Single-Threaded Implementations (ST)

The gradient kernel computes the image energy function using the magnitude of the gradient. The baseline implementation is similar to the implementation described in [19]. We improve the baseline by applying several hand optimizations as listed in Table 3, including *Smart Pointer Dereferencing*, *Arithmetic Optimizations*, *Loop Fusion*, *Smart Value Scaling*, and *Branch Elimination*.

Table 2: Description of different optimization techniques used in the implementations of kernels. The optimizations marked by (*) have a extended description (most likely towards the end of the chapter).

OPTIMIZATION	DESCRIPTION
Smart Pointer Dereferencing (SPD)*	Reduces the number of memory accesses by dereferencing pointers outside of loops. This technique is most useful when kernels access data elements that are encapsulated in a SoA or stored in a multi-level arrays.
Arithmetic Optimizations (AO)	Simplifies math to eliminate unnecessary arithmetic. Reduces the number of index transformation when using linear array to store 2-D/3-D dataset.
Loop Fusion (LF)	Improves locality and cache performance by fusing loops to perform computation with a single loop pass.
Smart Value Scaling (SVS)*	Scales values by a smart fraction to replace division operation by logical shift (only used when the kernel tolerates the errors).
Loop Interchange (LI)	Exchanges the order of nested loops to improving locality of access and take advantage of cache.
Data Structure Transformation (DST)	Transforms data structure to improve memory performance. For example, transforming an Array of Structure (AoS) to a Structure of Array (SoA).
Branch Elimination (BE)	Eliminates unnecessary branches by executing the boundaries conditions outside of the loops (improves ILP).
Reduced-width Operands (RWO)*	Reduces the data width to improve cache performance and SIMD parallelization.
Fast Math (FM)	Optimized math functions [31, 32] to improve arithmetic performance (reduces accuracy, not noticeable in some kernels).
Ping-Pong Buffering (PPB)	Improves performance by eliminating redundant memory copy and branches.
Array Padding (AP)	Improves performance by guaranteeing that every matrix row starts on an aligned memory location or new cache line.
Shared Memory Caching (SMC)	Improves performance by reducing the number of same-data memory access.
Texture Cache (TC)	Hardware managed, optimized for 2D spatial locality (benefits kernels with irregular access patters or low locality).
Lookup Table (LT)	Eliminates multiple computations of functions with the same input. Pre-computes the outputs for all inputs and stores the output in memory.
SIMD Shift and Insert (SSI)*	When data is loaded into SIMD register, eliminates extract loads of nearby data by using register-shift and data-insertion.

Table 3: Optimizations (which are listed in Table 2) applied to the single- and multi-threaded CPU, and GPU implementations. COT states for all non-SIMD CPU optimization techniques.

Kernel / Application	**Optimization**																				
	CPU										**SIMD**				**GPU**						
	SPD	**AO**	**LF**	**BE**	**RWO**	**SVS**	**LI**	**FM**	**LT**		**COT**	**DST**	**SSI**		**DST**	**PPB**	**AP**	**SMC**	**TC**	**FM**	**LT**
mri-q		x						x			x						x				
stencil							x				x						x	x			
histogram	x	x									x							x			
gradient	x	x	x	x							x	x	x				x	x	x	x	
dynamic programming	x	x		x	x						x		x				x	x			
matrix resizing	x										x		x		x	x					
Laplacian	x												x		x	x	x	x			
DEsolver	x	x							x						x						x
SC	x	x	x	x	x	x					x	x	x		x	x	x	x	x	x	
CAPPS	x	x							x				x		x	x	x				x

The dynamic programming kernel computes the *cumulative minimum energy* for the seam carving application. In the baseline implementation, we use the C++ *min* function to find the minimum value of the above 8-connected pixels, and add the result to the pixel's energy to obtain the new cumulative minimum energy value. We locate the lowest-energy seam by searching the last row of the seam matrix. The baseline implementation inherits many optimizations techniques used for the *gradient*. The *Branch Elimination* optimization, in particular, shows a considerable improvement over the baseline.

The *matrix resizing* baseline implementation loops through the rows and columns of the image and move each pixel to the left in their respective rows; starting one pixel after the removable pixel. We use another technique that utilizes the C/C++ *memmove* and *memcopy* functions to resize each row, which performs slightly better than the baseline implementation. Although employing linked-list data structures would have allowed data resizing to be very efficient, because this kernel is part of the seam carving application where non-resizing operations account for a much larger fraction of the execution time, as in many applications, the overall seam carving performance would have been negatively affected. Thus, we are forced to implement the resizing using array data structures.

As we discussed in Chapter 2, *CAPPS* utilizes two kernels: the *DEsolver* and the *Laplacian*. We implemented these kernels using the equations given in [8]. We used the optimization techniques discussed for gradient for achieving optimal performance. Finally, instead of computing exponential operations, a lookup table (*LT*) is employed for better performance without significant reduction in accuracy.

The baseline implementations for the three kernels, mri-q, stencil and histogram, are taken from the Parboil benchmark suite [6]. We improved the baseline implementation for the mri-q kernel (by about 86%) by applying *Fast Math* optimizations. We apply the *Loop Interchange* optimization to the stencil kernel, which greatly improved locality and resulted in 7*x* performance gain over the baseline.

4.2 Multi-threaded Implementations (MT)

For the multi-threaded implementation of the gradient kernel, we partition the input image into tiles consisting of consecutive rows (see Figure 3); a column-based division reduces locality. The number of rows in a tile depends on the number of threads and the height of the image. Unlike the gradient, the cumulative minimum energy computation uses a dynamic programming approach that is not parallelization friendly. This approach serializes the execution of rows. We therefore perform a row-by-row computation of the seam matrix (dynamic programming kernel) by dividing each row into fixed-width tiles and compute these tiles in parallel. We synchronize all threads after the execution of each row.

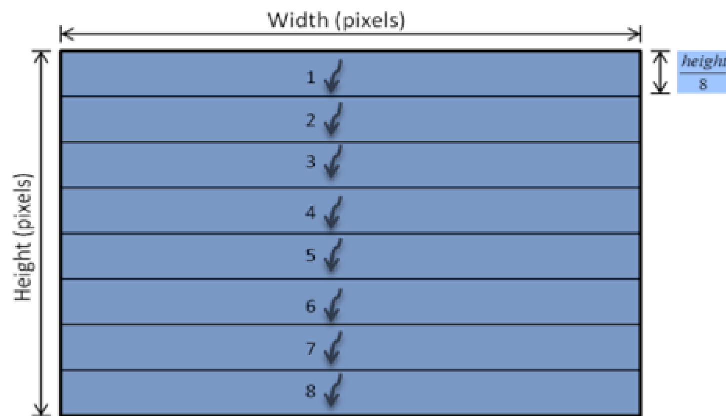

Figure 3: Division of work for multi-core CPU**.**

For the matrix resizing, mri-q, stencil, histogram, DEsolver and Laplacian kernels, the multi-threaded implementation is similar to that of the gradient kernel (see Figure 3). Given that there are no data dependencies in the computation of these kernels, we are able to divide the computation among threads as described above for the gradient. The multi-threaded implementation of the histogram kernel uses the reduction technique, where each thread first updates a thread-private (local) histogram and then "reduces" (adds) its local histogram to the global histogram only once at the end of the computation. Updates to these local histograms can execute in parallel but additions to the global histogram still require atomic operations. Finally, we also used message-passing interface (MPI) to parallelize CAPPS in order to take advantage of our in-lab cluster as described in Chapter 3.

4.3 SIMD Implementations (SIMD)

The *gcc 4.7* compiler is capable of auto-vectorizing programs to explore the potential of the SIMD units on the CPU. However, for successful compiler auto-vectorization, often, the programmer needs to write the code in a certain way. Besides the Laplacian kernel, no other baseline implementations for the kernels in this thesis were successfully auto-vectorized. Optimizations such as, *Data Structure Transformation*, *Loop Interchange*, *Smart Value Scaling*, and *Branch Elimination* have helped generating auto-vectorized code with some success for four of the eight optimized kernels: These are the gradient, dynamic programming, stencil, and Laplacian kernels. However, the SIMD units achieved much higher performance when the code was carefully vectorized by hand.

For the gradient kernel, after using the *Smart Value Scaling* optimizations, *gcc 4.7* was able to auto-vectorize the code (with 3.67x performance gain). By further using the *Data Structure Transformation* and *SIMD Shift and Insert* (*SSI* in Table 2) optimizations, we achieve a 33x performance gain over the baseline with a hand-tuned SIMD implementation. The hand-tuned SIMD implementation of the gradient uses Equations 9 and 10 [23] as an alternative method to compute the derivatives. By incorporating these changes, we eliminated three loads (one for each RGB channel), three register-insert operations, and eight logical and arithmetic operations per pixel.

$$\frac{\partial I(x,y)}{\partial x} = I(x+1,y) - I(x,y) \quad (9)$$

$$\frac{\partial I(x,y)}{\partial y} = I(x,y+1) - I(x,y) \quad (10)$$

By moving the cumulative minimum energy computation of the first and the last column outside of the loop, we reduce the boundary check instructions (*Branch Elimination*) for the dynamic programming kernel. This optimization helps the compiler vectorize dynamic programming (4.89x performance gain over baseline). By implementing the *SIMD Shift and Insert* mechanisms, our hand-tuned SIMD implementation achieves 2.25x over the compiler auto-vectorized code. Auto-vectorization did not work for the matrix resizing kernel. However, with some hand-tuning, we were able to use the SIMD lanes to move 16 bytes simultaneously by loading each RGB channel into three separate registers and relocating 5.33 simultaneous pixels on average.

The stencil kernel was auto-vectorized after we have modified the baseline (from Parboil suite [6]) with the *Loop Interchange* transformation. The hand-tuned and

compiler auto-vectorized versions provided the same speedup of 3.34*x* over optimized single-threaded implementation. *gcc* 4.7 was not able to auto-vectorize the mri-q kernel because it uses *sine* and *cosine* functions that are not part of the SSE/AVX instruction extension. We were able to hand-vectorize this kernel by implementing *sine* and *cosine* with AVX instructions with very good accuracy using [35]. The hand-tuned SIMD was 3.27*x* better than the optimized single-threaded version.

The full CAPPS application and the DEsolver kernel were unable to utilize the SIMD units on the CPU due to the current SEE/AVX limitations; no support for special functions such as exponential exists. Unlike, the mri-q kernel, CAPPS requires very high floating-point precision. Therefore, we were not able to use [35] to vectorize the DESolver kernel. As a result, we do not have a SIMD CAPPS implementation. We could have used SIMD after incorporating the *Lookup Table* optimization, but SIMD CPUs do not have gather/scatter SIMD operations yet, which also affected the vectorization of the histogram kernel. We were able to utilize SIMD for the reduction phase of the multi-threaded histogram kernel. Each thread-private histograms is copied to separate global histogram (multithreaded reduction), which are then added into one global histogram using the SIMD lanes.

4.4 Using Short Operands: A Case for SIMD Performance

The width of an operand has important implications on SIMD performance: the shorter the operand is the more parallelism there is. Therefore, it is important to carefully decide on operand widths. It is wasteful to use 32-bit operands when 8- or 16-bit operands suffice.

Occasionally, although not directly applicable, scaling down values, when it does not hurt accuracy of the overall results, allows the use of shorter operands and thereby improves SIMD performance significantly. We observed an example to this in the dynamic programming kernel, which is used to compute the cumulative minimum energy (seam matrix) in the seam carving algorithm. In theory, the values of the seam matrix could grow beyond 64K (unsigned short). Therefore, the baseline implementation uses 32-bits (unsigned integer) to store the values of the seam matrix. However, Figure 4 shows a histogram of the seam matrix values, which reveal that the values do not exceed 7,500. We analyzed many images with different sizes and characteristics, and found that even in very large images (with high energy) the largest value was below 19,500 (Figure 5). This allows us to use 16-bit instead of 32-bit operands. This simple optimization doubled the performance of the SIMD implementation for the dynamic programming kernel. Short operands also improved the non-SIMD implementation by 24%, due to cache performance.

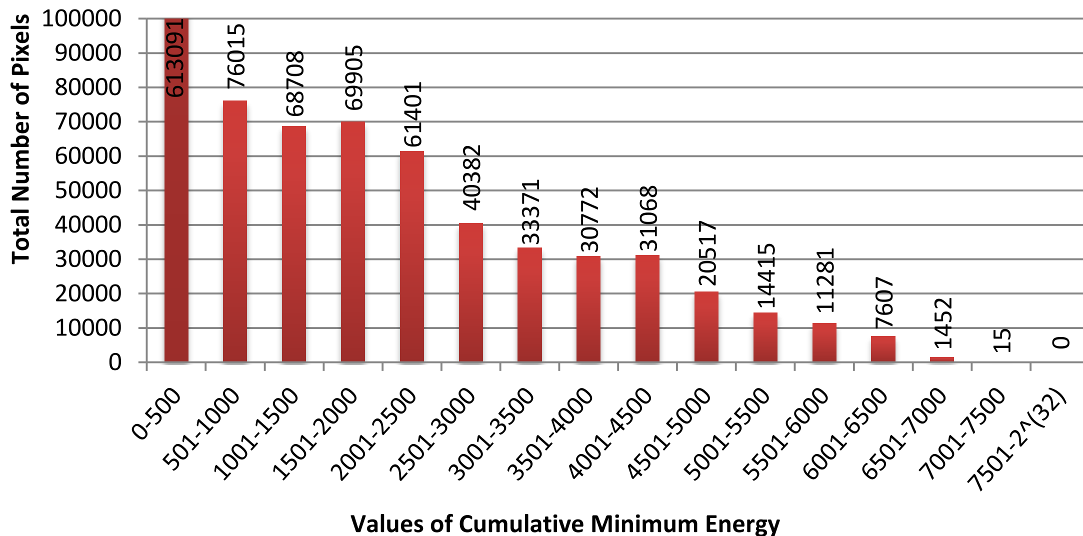

Figure 4: A histogram of the CES for a 1200x900 image.

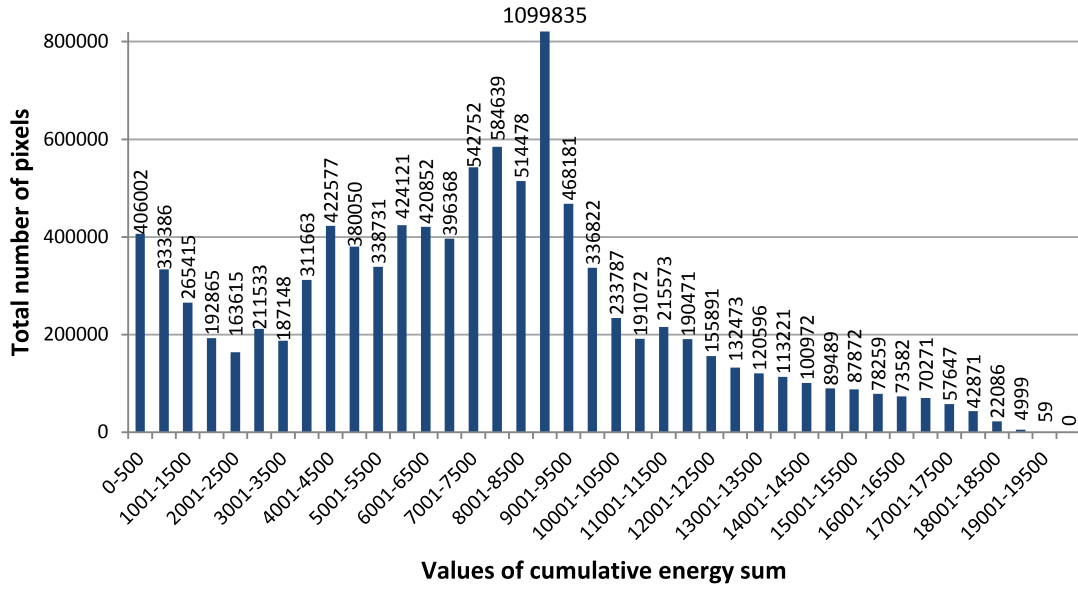

Figure 5: A histogram of the CES for a 3648x2736 image.

Pitfalls of Short-width Operands on SIMD Units

When performing SIMD arithmetic, especially on short-width operands, it is the programmer's responsibility to ensure that no overflow occurs. This is because unlike non-SIMD execution units, where the results of short-width operands (e.g. 8-bit) are store in a 32- and 64-bit register, the Intel SSE and AVX extensions partitions the SIMD registers into the various supported length, as specified by the programmer. In the case of 8-bit arithmetic, the resulting values are placed in an 8-bit location of an SSE 128-bit register, which will cause overflow if the results are beyond 255 (unsigned) or 127 (signed). Therefore, it is the programmer's responsibility to be cautions when taking advantage of the performance gain of short-width operands on the SIMD units. We recommend a well understanding of the operation before exploring short-width operands on SIMD CPUs. A proper methodology is to conduct similar analysis as we have done in the previous subsection.

4.5 GPU Implementations (GPU)

The need for accessing neighboring pixels to compute the gradient strongly influences the way we access memory on the GPU. In [19], the authors present an incremental approach towards improving the performance of the energy function computation, which we incorporate into our GPU implementation of the gradient kernel. We also use Equations 9 and 10 to benefit from similar improvement as in the SIMD implementation. With a 32x9-block configuration (warp 9 only assist in caching), each thread loads a single pixel; achieving full coalesce accesses for the computational pixels and the bottom-neighboring pixel. Our caching method works well, but the best performance is achieved by careful optimizations including the use of *Texture Cache* and *Fast Math* optimizations.

The GPU implementation of dynamic programming partitions the rows into horizontal tiles. Since there is no synchronization among different thread blocks, the kernel is invoked once per row and we synchronize in between calls.

For the matrix resizing kernel on the GPU, we launch one thread per data element (pixel in the case of seam carving) in order to achieve one data-element relocation per thread. None of our previous resizing methods achieved such high parallelization; we are able to move hundreds of pixels simultaneously. To prevent neighboring threads from overwriting the pixels before they can be read, we used the *Ping-Pong Buffering* optimization technique, as described in Table 2.

The GPU Implementation of `CAPPS` exploits the fact that the computation of the transmembrane current is 100% separable; we can compute the transmembrane current, which includes 629 floating-point operations, for every node in parallel

without any data dependencies. The memory access patterns and computation for the *Laplacian*, used for the computation of the transmembrane voltage, are very similar to that of the gradient. This implies that we can benefit from the GPU optimizations employed in the gradient kernel. For CAPPS, we implemented two GPU kernels: a kernel to compute the transmembrane current (*DEsolver* kernel), which includes the computation of the differential equations, and a kernel to compute the transmembrane voltage, which includes the Laplacian and one differential equation. We placed all constants in the constant memory using the guidelines in [36]. Furthermore, we place most of the data on the GPU to minimize the host-to-device and device-to-host memory transfers. The only device-to-host memory transfer occurs when the CPU needs to write the transmembrane voltage to the output file, which occurs every 10,000 iterations. One iteration simulates one time step, and it involves the computation of the transmembrane voltages and currents for all 90,000 nodes. The optimized GPU version of CAPPS required us to apply many optimizations, including, *Data Structure Transformation*, *Ping-Pong Buffering*, *Arithmetic Optimizations*, and *Lookup Table*, as described in Table 2. We also used *Page-locked Memory* to avoid the host-to-device memory copy of the *stimulus current*, which is updated by the CPU for every node on every iteration. Results show that accessing the CPU memory directly from the GPU incurs less overhead than that of the *stimulus current* host-to-device memory copy, for the implementation of CAPPS. Finally, we have evaluated the GPU implementations of the mri-q, stencil and histogram kernels from the Parboil benchmarks [6].

4.6 Seam Carving Specific Optimization: The Energy Update (EU)

In seam carving, when removing seams, the frequency at which the energy function (*gradient* kernel) is recomputed has a significant impact on the quality of the resized image. The best quality is obtained when the energy is recomputed after the removal of a single seam [7]. To reduce the computation, it is possible to recompute the energy function after a predetermined number of seams have been removed [37].

In this thesis, we implemented a new method to improve the performance of recomputing the energy function and preserve the best resizing quality. When a single seam is removed and the energy function is recomputed, the majority of the energy values remain unchanged. The only pixels affected by the removed seam (the dark gray pixels in Figure 6a) are the left and right neighboring pixels illustrated in white in Figures 6a and 6b. Thus, we only recompute the energy of the pixels that undergo an energy change, to reduce the computation. This method produces the same results as recomputing the entire energy function using much less computation, which improves the performance significantly (see Chapter 5).

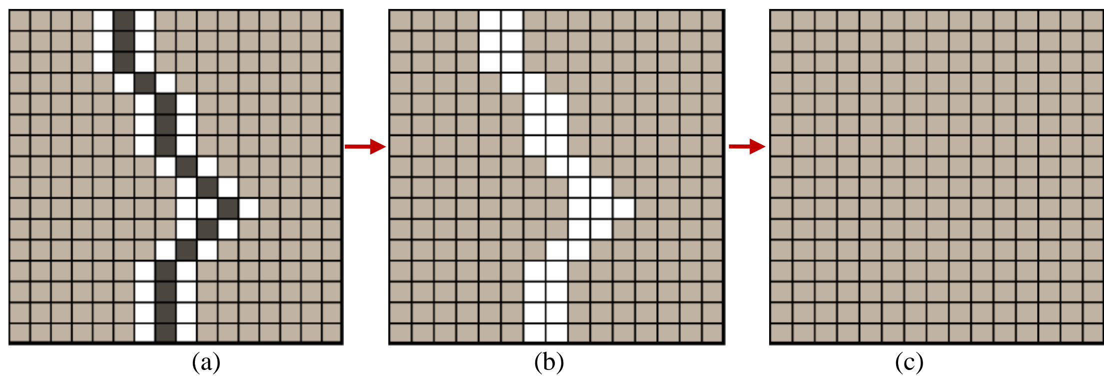

Figure 6: Proposed Update Algorithm for recomputing the energy. (a) The dark (removable) pixels only affect the white pixels. (b) Only recompute the energy for the affected pixels. (c) After update.

4.7 Memory Optimization Techniques

GPUs have a wide memory bus for simultaneously loading large amounts of data in order to supply the high demand imposed by the many executing threads. Unlike the CPUs that hide the memory latency by utilizing large caches and complex logic such as pre-fetching, the GPU memory exhibits high bandwidth and high latency. The GPU high memory latency could be hidden by accessing memory in the most favorable pattern that takes advantage of the GPU memory organization.

Implicit and Explicit Caching

Modern CPUs contain different levels of caches, which are managed implicitly by the hardware to store the most frequently used data. The programmer, however, is able to use techniques such as *Loop Fusion* and *Loop Interchange* to take advantage of the cache and improve the memory performance. GPUs implement user-managed caches (*Shared Memory*) and give explicit control to the programmer. When necessary, it is very important to utilize the GPUs' shared memory to take advantage of locality or the *Texture Cache* to benefit from 2-D spatial locality. When utilizing *Shared Memory* on the GPU, special attention must be given to bank conflicts to prevent serialization of threads within a warp.

Array of Structure (AoS) vs. Structure of Arrays (SoA)

Using the CPU SIMD unit places restrictions on the layout of the data. For example, operands must be loaded and the results of SIMD operations are stored into 128-bit (SSE) or 256-bits (AVX) registers. To achieve the best performance, data should be placed into an address-aligned data structure. For example, for 8-wide single

precision floating-point SIMD, the best performance will be when the data is 32-byte aligned. For vector addition, the vectors' data must be loaded into two 256-bit registers. To achieve the best performance, eight vector components from each vector must be loaded simultaneously. This can only be achieved if data is stored sequentially in an address-aligned location.

Let us look at a different example. Suppose that we need to find the minimum value between the RGB channels for every pixel in an image. *Array of Structure* and linear arrays (with alternating channels e.g. {R, G, B, R, G, B…}) are two common data structures used to store the image data. However, depending on the application and the memory access patters, an AoS or a linear array might not be the best solution. These data structure place the individual RGB channels at least three bytes apart. Such data structure makes it impossible to perform a single load of sixteen simultaneous red elements with SSE; currently, there is no AVX support for integer arithmetic. Instead, we are forced to insert each element one-by-one, which diminishes the performance by a significant amount. A better solution is to store the data in a *Structure of Arrays*, which allows registers to be loaded with 16 bytes on a single load. Implementing the correct data structure, an *Array of Structure* or a *Structure of Arrays* in particular, could reduce the number of loads and stores by up to a factor of 16. In the case of the gradient kernel, the *Data Structure Transformation* permitted us hand to vectorize the code, and achieve a speedup of 13.78*x* over the Optimized Single-Threaded Implementations (ST).

It is important to mention that *Structure of Arrays* with large number of structure members could incur memory access penalties due to the limited number of pages that

can be maintained by the system [33], but GPUs are not affected by this phenomenon. We recommend analyzing the applications' memory access patterns and the memory organization of the system to select the appropriate data structure, see [33].

Final Comments Memory Optimization Techniques

Memory Coalescing on the GPU utilizes 100% of the available memory bandwidth. Non-aligned memory and inappropriate data structure causes uncoalesced accesses and wastes significant device memory bandwidth. Much research has been conducted to find methods of automatically transforming data structure, changing access patterns, and identifying suitable memory spaces [34]. Techniques such as 1-D to 2-D array mapping, array padding and data caching, see [19], are valuable in boosting performance. Finally, we advise caching data in registers when the same thread only reuses the data. This method was used in the DEsolver kernel and in CAPPS.

4.8 SPD, SVS, and SSI Optimization Techniques

Smart Pointer Dereferencing

The *Smart Pointer Dereferencing* (SPD) optimization technique uses a smart method to access the data fields of *C-Struct* and to access multi-dimensional arrays. For instance, for the seam carving operation, we encapsulate all of the data in a *C-struct*. This is a common practice for the organization and reusability of the code. When computing the gradient for the seam carving operation, instead of dereferencing the *image width* and *height*, and the *image data* and *gradient arrays* inside the loops, we dereference these fields and store them in a local (automatic) variable before

entering the loops, which most likely will be place in a register by the compiler. For the computation of the gradient of a 1200x900 image, this optimization reduces the amount of pointer dereferences from over 17 million (16 pointers dereference per pixel) to four dereferences for the entire computation of the gradient. In addition, the *Smart Pointer Dereferencing* optimization technique is completely independent of the size of the image, which is not the case when the dereferencing occurs inside the loop.

For multi-dimensional arrays that are access inside of nested-loops, a similar optimization could be applied. Let us assume that we need to access a 2-D array (`A`) inside a 2-level nested-loop. We could simply write `A[i][j]` inside the second loop body. However, unless optimized by the compiler, such implementation will incur unnecessary memory loads for the memory reference indicated by `A[i]`. For instance, suppose that we need to access all elements in a 1000x1000 array, this will cause a significant amount of unnecessary memory loads (999,000). Since "*i*" is constant for the access of an entire row, as an alternative, we can assign `A[i]` to a local pointer, call it `Arow`, outside of the second loop body. `Arow` is then use to access all of the elements for the current row as follows: `Arow[j]`. This will reduce the number of loads per row of `A[i]` from 1000 to 1. In this case, 999 unnecessary loads are removed per row. Thus, we are able to reduce the number of loads of `A[i]`from 1,000,000 to 1,000. This is a significant improvement when 2-D arrays are required.

Unnecessary pointer dereferencing is constantly use by software developers. In this work, we show the performance benefit, and suggest adding this technique to the compiler optimization phase to improve performance.

Smart Value Scaling

Some applications may be tolerant to the scaling of value. Scaling values by a carefully selected fraction could improve performance because it allows us to replace divisions with logical shifts. In the case of the gradient, we normally convert RGB images to grayscale before computing the gradient as previously stated in this chapter. A simple technique is to average the RGB channels: ($\frac{R+G+B}{3}$). In this case, scaling down by $\frac{3}{4}$ permits us to simplify the mathematics to ($\frac{R+G+B}{4}$), (the 3s cancel out), and convert the division by 3 to a 2-bit right logical shift. This technique should only be used when the relative value suffices and the exact value is not needed. We omitted this optimization technique from our best implementation of the gradient kernel. Since this is an application specific optimization, we decided to show the performance of the gradient kernel that produces the exact values. However, this technique was necessary for the auto-vectorization of the gradient kernels, since currently, SSE does not support integer division. This technique does not affect the seam carving operation; the resizing quality is the same when the values of the gradient are scaled down by three over four.

SIMD Shift and Insert

Many times, applications have a variety of operands with different data-width. For example, in seam carving, the energy values require 8-bits while the seam matrix values are either 32 or 16 bits. In either case, the 8-bit value must be cast to either 32 or 16 bits before we could perform any arithmetic on the two. Figure 7 illustrates an efficient method of performing the cast from 8 to 32 bits. Instead of looping through the array, using strides of 4 and performing unaligned loads, we loop through the array

using strides of 16. On every stride, we load 16 8-bit values into a 128-bit register. We cast the values in the register, which takes the lower four bytes and places them in another register (Figure 7a). Instead of reloading the next four bytes, which will result in an unaligned load, we perform a logical right shift of 4 bytes. We repeat the byte-to-integer conversion (Figure 7b) three more times to cast the 16 8-bit values to integers. Be aware that the values in the second register in Figure 7a must be utilized or store elsewhere before the operation in Figure 7b takes place. Otherwise, the previous data (A_3-A_0) will be replace by (A_7-A_4).

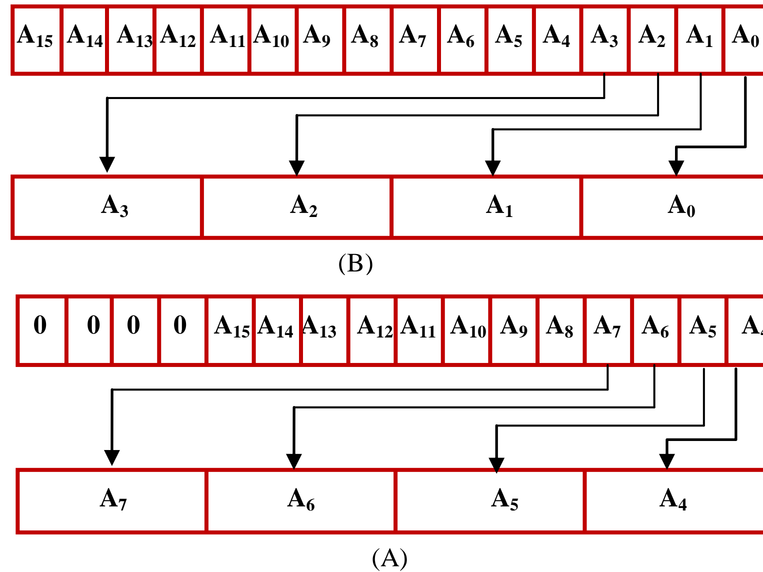

Figure 7: Efficient data casting using SIMD shift and insert.

Another advantage of the *SIMD Shift and Insert* optimization technique is to minimize the number of unaligned loads caused by the left and right neighboring elements in an array. This technique was applied to the gradient, Laplacian, and stencil kernels to eliminate unaligned loads. These kernels need to access their left and right neighboring elements, which are part of their convolution operation. The rest of the elements are aligned, which forces the left and right elements to be unaligned. To

compensate and improve performance, we load 16 elements into an SSE register (gradient) or 8 elements into an AVX register (Laplacian and stencil). We then shift the register one element to the left or right, and insert the missing element. The three register will contain the same data as if performing one aligned and two unaligned loads. However, we accomplish this with one aligned load, and two *logical-shift* and *register-inset* operations. Using this technique, we were able to improve the SIMD performance of the gradient, Laplacian, and stencil kernels.

CHAPTER 5

PERFORMANCE AND ENERGY EVALUATION

In this chapter, we present the performance and energy-efficiency evaluation for the mri-q, stencil, and histogram kernels (Parboil benchmarks). We then evaluate seam carving, starting with the gradient, dynamic programming, and matrix resizing kernels, and then the full application. Following seam carving, the Laplacian and DEsolver kernels, and the full CAPPS application are evaluated. For the kernels that are part of seam carving and CAPPS, we do not evaluate energy-efficiency separately. The reason is that we get better insights by evaluating the energy-efficiency of the entire seam carving and CAPPS applications.

5.1 Performance and Energy Efficiency Evaluation of the Parboil Benchmarks

Figure 8 shows the performance results for the Parboil kernels on the CPU and GPU platforms. The performance improvement is measured for the kernel-only computation and the overall execution time. The total execution time includes the overhead, such as data transfer between the CPU and GPU. Figure 8 shows the kernel-only performance gain, in which the GPU performs best, with substantial speedups over the baseline; 214*x*, 773*x* and 39*x* for the mri-q, stencil and histogram, kernels respectively. However, the impact of the overhead may offset the benefits from the GPU. For example, for histogram, the GPU overall execution time is actually 9*x* worse

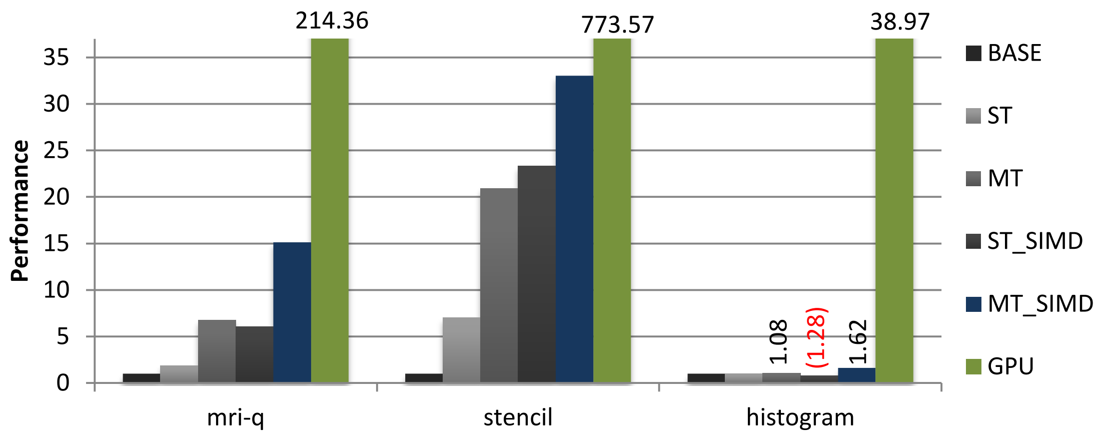

Figure 8: The computation-only speedup of the Parboil kernels. For histogram, MT_SIMD is the same as MT, but uses SEE for reduction. ST_SIMD is worse than ST; thus, we do not have an MT version.

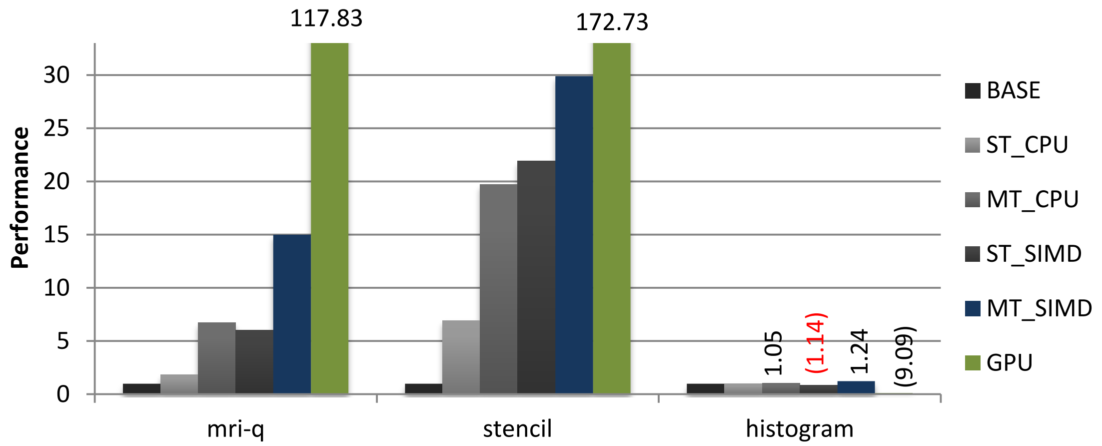

Figure 9: The overall speedup of the Parboil kernels. For histogram, MT_SIMD is the same as MT, but uses SSE for reduction. ST_SIMD is worse than ST; thus, we do not have an MT version. The values in red or () means that the performance worsen by the indicated amount.

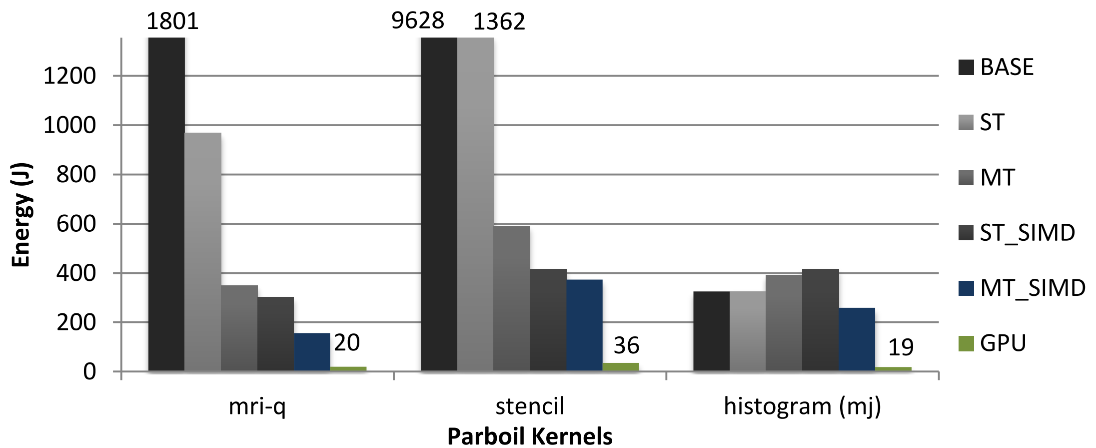

Figure 10: Energy consumption of parboil kernels for the computation only. The energy values for the histogram are in millijoules for displaying purpose.

Table 4: Energy consumption and Relative Energy-Delay Product (REDP) for the overall kernels.

		Kernels		
Implementations	**Energy/RDEP**	**mri-q**	**Stencil**	**histogram**
Base	Energy	1807	9756	0.65
	RDEP	1	1	1
ST	Energy	969	1396	0.65
	RDEP	0.044	0.021	1
MT	Energy	352	635	0.804
	RDEP	0.029	0.003	1.28
ST_SIMD	Energy	305	448	0.74
	RDEP	0.0278	0.0021	1.29
MT_SIMD	Energy	158	419	0.681
	RDEP	0.006	0.001	0.92
GPU	Energy	37	165	12.8
	RDEP	0.00017	0.0001	173.31

than the baseline, as illustrated by Figure 9. We also observed a significant performance drop compared to the kernel-only times for mri-q and stencil. The stencil kernel undergoes a 4.7*x* reduction in performance, in comparison to the kernel-only speedup, due to the overhead. The CPU implementations do not incur such overhead, and the overall execution time remains similar to their kernel-only computational time. This favors the CPU for applications that use kernels like the histogram, where the computation to memory operation ratio is not high enough to fully utilize the capabilities of the GPU.

Overall, the histogram does not scale on data parallel hardware. Figure 9 shows that the multi-threaded implementation (MT) provides a 5% speedup over the baseline. Most of the performance gain is lost during reduction phase of the histogram where the thread-private histograms are reduced to a global histogram. Without the reduction, the multi-threaded implementation achieves a 2.77x gain over the baseline (not shown in figure). We further utilize SIMD lanes for the multi-threaded reduction, which results in 24% speedup (MT_SIMD). Both the mri-q and the stencil are well suited for the GPU, achieving beyond 117x and 172x over the baseline

implementation, respectively. It is important, however, to do a fair comparison between the CPU and GPU. By applying the *Loop Interchange* optimization, we improve the performance of the base implementation of stencil (taken from the Parboil benchmarks) by 7x. By utilizing the SIMD unit on the CPU, a 21.9x performance boost is achieved using a single CPU core (ST_SIMD). Although the SIMD implementation of stencil does not scale linearly on multi-core CPU, we are able to improve the overall performance by 29.9x, with all four cores on a quad-core CPU. By properly utilizing the CPU, the performance achievement of the GPU over the CPU is 5.7x – a much smaller number than 773x! This shows the danger in comparing the platforms unfairly.

For completeness, Figure 10 shows the computation-only energy evaluation for the Parboil kernels. Table 4 present the energy consumption and energy-efficiency of the overall execution of the Parboil kernels. We measure the energy-efficiency with the EDP metric. For the data-parallel mri-q and stencil kernels, the GPU is the clear winner in both energy-consumption and energy-efficiency. The multi-threaded SIMD implementation provides the second best energy-consumption and efficiency. For the histogram kernel, the GPU has the worst energy-consumption and dramatically worse EDP. However, if kernel only energy consumption and energy-efficiency were evaluated (figure not shown, relative EDP=0.0015), GPU would have been the best by far for the histogram, which is not true and might be misleading.

5.2 Performance and Energy Efficiency Evaluation of Seam Carving

Evaluation of Seam Carving Kernels

We now discuss the performance evaluation of the seam carving kernels and the performance and energy-efficiency of the full seam carving application. We have not evaluated the energy-efficiency for individual kernels separately for reasons described in Chapter 2. We conduct the kernel evaluations using a 1200x900 RGB image. Figure 11 shows the performance gain after applying the single-threaded optimizations (ST) to the gradient baseline, a 2.43*x* speedup. ST scales well on multi-core CPUs (MT), and achieves a 3.07 scalability, which translates to a 7.47*x* speedup. By employing the *Smart Value Scaling* optimization technique, auto-vectorization becomes possible. The auto-vectorized code achieves a 10.7*x* speedup over the baseline (figure not shown). Our hand-vectorized implementation (ST_SIMD) gains a 33.5*x* performance boost over the baseline. This implementation does not scale on multi-core CPUs (MT_SIMD).

The best implementation of the gradient on the GPU uses the *Texture Cache* instead of *Shared Memory*. The reason is that the overhead introduced by caching the apron pixels (see [19]) was much greater than the performance gain from limited locality (each pixel is only accessed by three different threads). Overall, by using the GPU, we improved the gradient 102.6*x* over the baseline, which translates to a 3.06*x* speedup over best CPU implementation. This is a fair comparison that could have been misleading if the CPU version was not fully optimized and did not use the SIMD units.

Figure 11 also illustrates the performance for the various implementations of the dynamic programming kernel. By utilizing similar optimization techniques as in the gradient, and with the addition of *Branch Elimination* and *Reduced-width Operands* optimizations, we managed to improve the single-thread performance by 71%. The GPU implementation undergoes a significant kernel launch overhead, and only achieves a 61% speedup over the baseline. Because of the synchronization problem incurred by dynamic programming, both non-SIMD and SIMD implementations exhibit very poor scalability on multi-core. The single-threaded SIMD CPU (ST_SIMD) implementation of dynamic programming yields the best performance, an 11*x* and 6.84*x* performance boost over the baseline and GPU, respectively. The *Reduced-width Operands* optimization only helps slightly with cache performance for the non-SIMD implementations. For the SIMD implementations, however, it is a critical optimization step as it doubles the amount of data-elements that can simultaneously execute on the SIMD units. Hence, by reducing the data width from 32- to 16-bits, we were able to double the performance of ST_SIMD (5.*16x* with 32-bit).

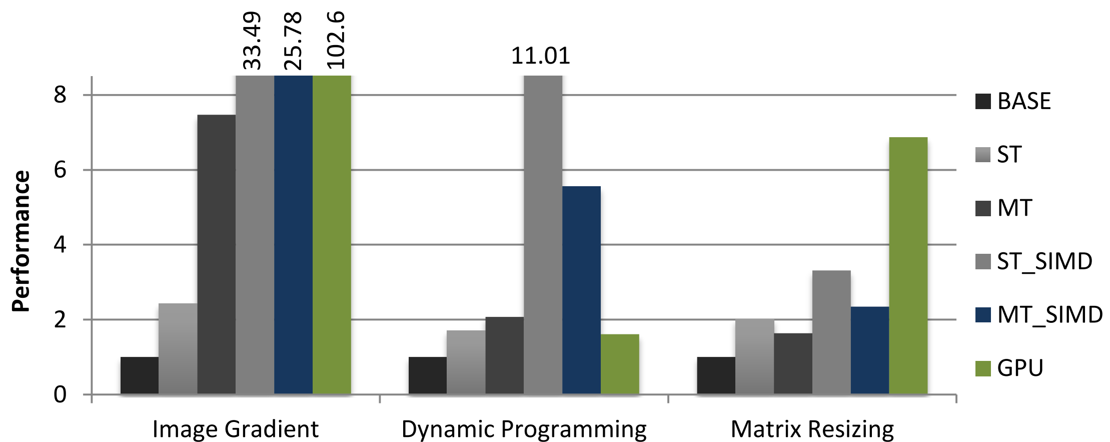

Figure 11: Performance evaluation of Seam Carving kernels.

In Chapter 4, we presented different CPU methods for the matrix resizing kernel. This kernel does not scale on multi-core. The SIMD CPU implementation, which is capable of moving an average of 5.33 pixels per operation, accounts for the best CPU performance with a 3.31*x* speedup. Even so, our best method for resizing is on the GPU by assigning one thread per pixel relocation. This implementation achieves a performance boost of 6.87*x* over the baseline and 2*x* over the best CPU implementation, which uses the SIMD units (Figure 11).

Evaluation of Seam Carving Application for the Resizing of Images

To evaluate the performance of the full implementation of the seam carving operation, we use the same 1200x900 RGB image and reduce the width of the image by one-third of its original width. Figure 12 shows that the multi-core SIMD CPU, with/without energy update (MT_SIMD, MT_SIMD_EU), performs the best for the SC resizing operation, 29.16*x* and 32*x* overall speedup, respectively. Given that the GPU achieved the best overall performance for the gradient and matrix resizing kernels, it would be expected that the GPU would also achieve very good performance on the resizing operation.

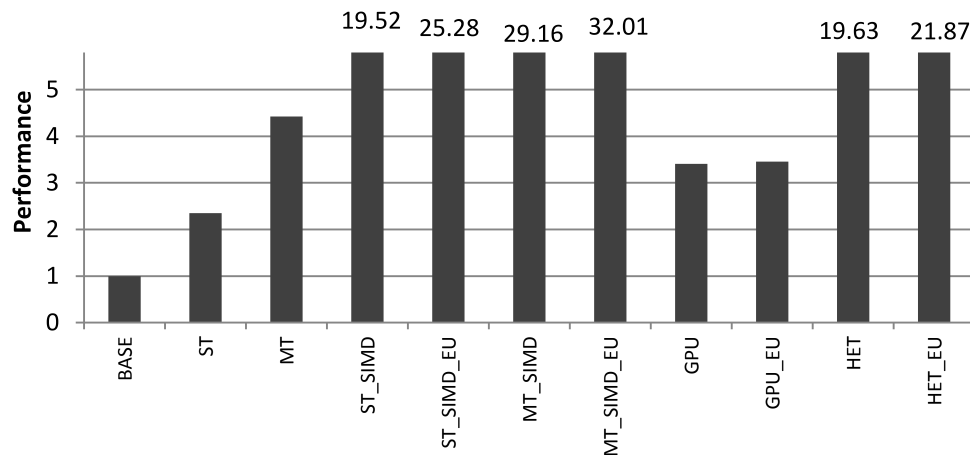

Figure 12: Performance of full Seam Carving application.

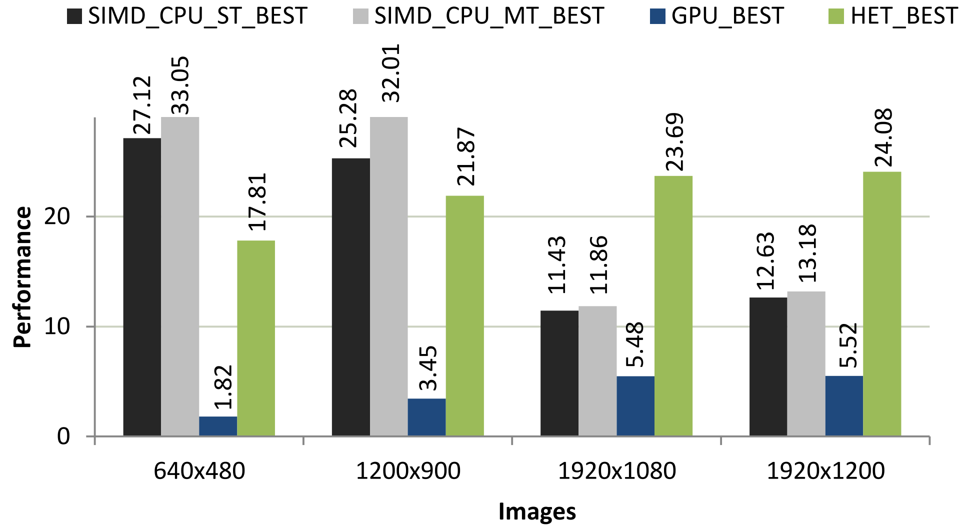

Figure 13: Performance of best platforms full SC resizing operation.

However, the GPU only gains a 61% improvement in the dynamic programming kernel, while the SIMD CPU achieved 11*x*. The dynamic programming kernel takes the second larger fraction of the execution time for the seam carving removal operation (behind the gradient). Other sequential components, such as backtracking to construct the minimum energy seam, can also be limiting factors, and reduce the overall performance. This shows the importance in considering full applications for performance evaluation. It is important to fully evaluate the performance and characteristics of multi- and many-core architectures. Kernels, however, are not able to expose all of the hardware constrains as good as full applications.

Figure 13 shows that as the image size increases, the CPU-GPU heterogeneous implementation (HET/HET_EU) performs much better than the other implementations. It does not achieve the best performance on the small and midsize images, but it is almost 2*x* faster than the best CPU implementation for the high-resolution images. Therefore, for large data set, a better approach is to utilize a true heterogeneous implementation to explore the best of both platforms. SIMD units offer

implicit synchronization, which is ideal for dynamic programming. GPUs offer high-bandwidth and 1000s of active threads, which makes it ideal for the gradient and matrix resizing.

Evaluation of Seam Carving Application for the Resizing of Videos

Figure 14 shows the execution time and performance improvement for the resizing of a HD video (1920x1080). This verifies that our CPU-GPU heterogeneous implementations (HET and HET_EU) are the best approach for resizing large images and video. Seam carving is a computationally-intensive operation, which makes video resizing very time consuming. It takes over six hours to resize a one minute of video (by one-third). By using the hardware efficiently, we are able to decrease the resizing time from 6 hours to 17 minutes. Figure 15 shows the energy consumption and relative EDP of the video resizing operation. We see that the HET_EU is not only the fastest implementation, but also the most energy efficient. The SIMD implementations provide the second best performance and energy-efficiency followed by the GPU.

Figure 14: Performance of Seam Carving to resize a 1 minute of video.

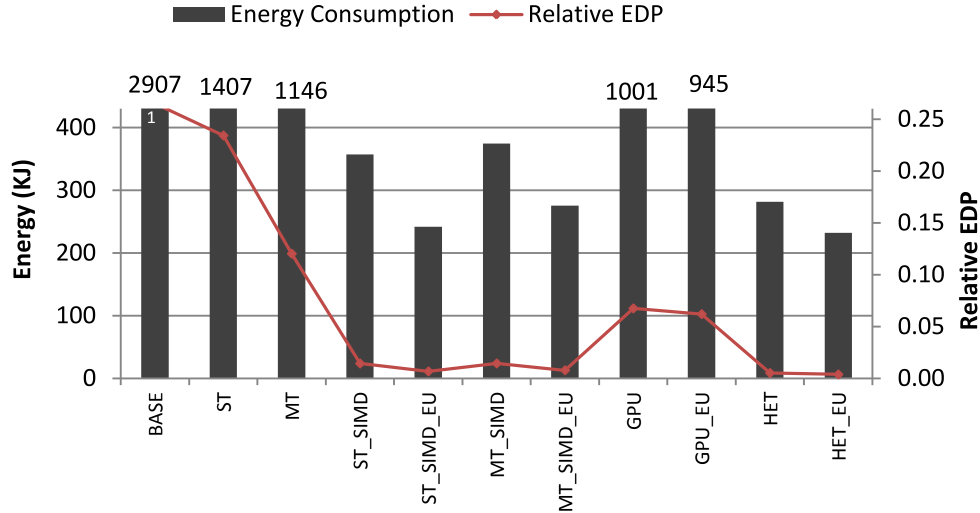

Figure 15: Energy evaluation of Seam Carving to resize 1 minute of video.

5.3 Performance and Energy Efficiency Evaluation of CAPPS

The single-core implementation of CAPPS takes approximately 10 days to carry out a single simulation. Driven by the need to reduce the execution time, we first implemented a parallel version of CAPPS using MPI and ran it on a cluster with 60 cores. Figure 16 and 17 show the performance of the DEsolver and Laplacian kernels, respectively. The DEsolver kernel has good scalability and achieves a 36.2*x* over the baseline running on a 60-core cluster. The Laplacian kernel does not scale well on multi-core and we do not show results beyond four threads. The GPU implementation achieves an impressive 61.4*x* speedup.

Figure 18 shows the results for CAPPS. When executing the simulation on multiple cores, the large dataset is partitioned into smaller subsets, which benefits the cache performance. This is one explanation for achieving super-linear speedup with 2, 4, and 8 cores; a 2.07*x*, 4.25*x*, and 8.23*x* speedup, respectively. For 16 cores, two shared-memory systems are used to form a distributed-memory system. The network

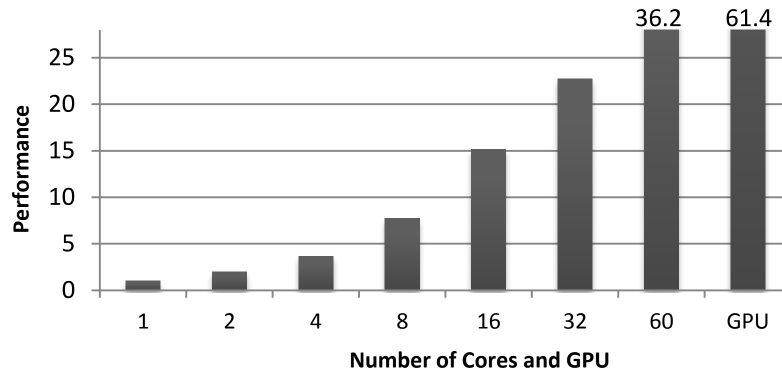

Figure 16: Performance evaluation of the DEsolver kernel.

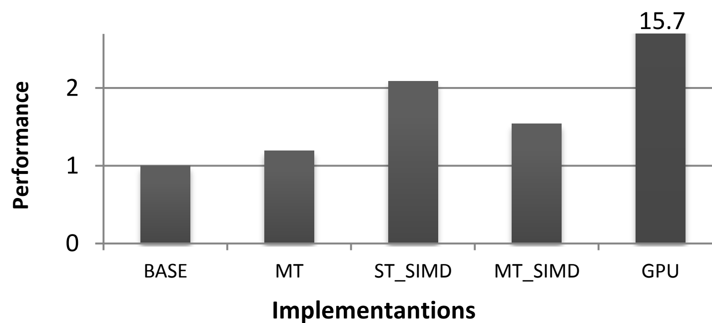

Figure 17: Performance evaluation of the Laplacian kernel.

overhead is small with two systems, and the performance improvement of the cache helps hide the network latency. This implementation achieves 16*x* speedup resulting in 14 hours and 57 minutes to complete a single simulation. Beyond 16 cores, the speedups are no longer linear due to the network overhead. The best performance on the CPU cluster is achieved with 60 cores. This configuration does not exhibit the best scalability, but it performs the simulation in 6 hours and 22 minutes, a 37.8*x* speedup.

In Figure 19, we show the energy consumption of running one CAPPS simulation on the CPU cluster. Although the 60-core cluster performs the simulation in 6 hours and 22 minutes, it is very energy-inefficient; it consumes 92.55 MJ for a single CAPPS simulation. The configuration that consumes the least amount of energy (37.94 MJ) on the CPU is the 8-core implementation. With 16 cores, the energy

consumption is slightly larger (<1MJ) and the performance is approximately 2*x* faster than the 8-core implementation. An extra mega joule could be a reasonable tradeoff in order to double the performance, see Figures 18 and 19. However, an extra 53.66 MJ is required to reduce the execution time from approximately 15 hours to 6 hours and 22 minutes. In summary, Figures 18 and 19 illustrates that by adding more machines to the cluster, we are able to reduce the execution time. However, the increase in performance comes at a cost. The energy consumption increases rapidly as we increase the number of network-interconnected machines.

Figure 18 and 19 also show the results of our GPU implementation of CAPPS. This implementation achieves an impressive performance of 58.1*x*, 54% better than the 60-core cluster ($60,000 value) on a desktop system equipped with a GPU (a $1,250 value) as described in Chapter 3. Using a CPU-GPU heterogeneous system, we are able to perform a CAPPS simulation in 4 hours and 8 minutes. Most importantly, as Figure 19 shows, our GPU implementation is 18.4*x* more energy-efficient than the MPI on a 60-cores cluster. Our results show that GPU is the clear winner in terms of performance, energy-efficiency and hardware cost for an application like CAPPS.

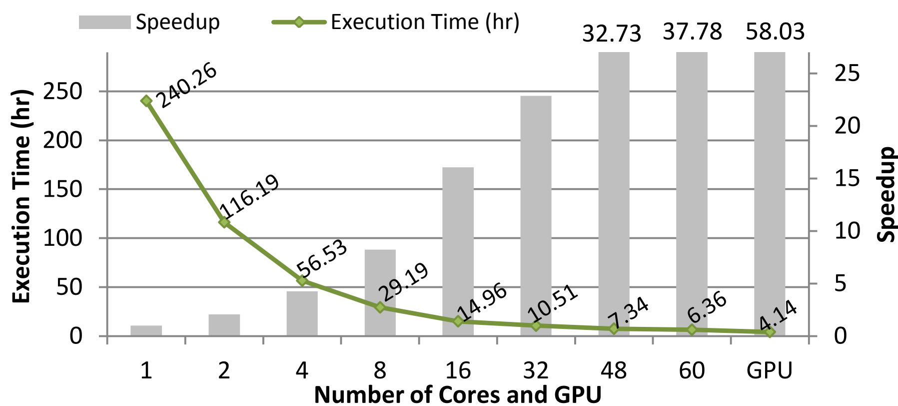

Figure 18: Performance evaluation of CAPPS.

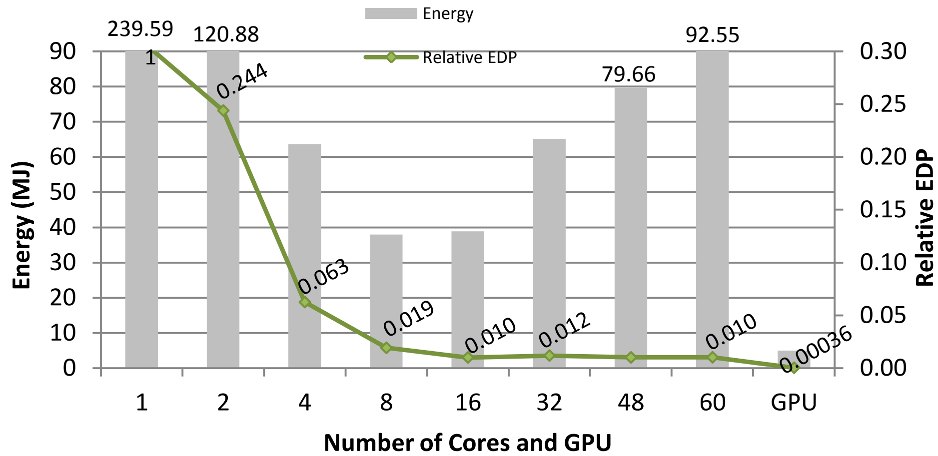

Figure 19: Energy evaluation of CAPPS, includes relative EDP.

5.4 Energy Efficiency and Dynamic Voltage-Frequency Scaling (DVFS)

In this thesis, we have not evaluated the impact of *dynamic voltage-frequency scaling* on energy-efficiency. We observe, however, that for our compute-intensive benchmarks, energy cannot be saved by lowering the core clock, because when the clock is down-scaled, then the execution time is highly increased, which results in an increase on cumulative energy consumption. To the best of our knowledge, the system software does not employ DVFS for GPUs. GPUs may not be energy-efficient when FLOPs/J drops under a threshold. DVFS algorithms are worth pursuing for GPU systems. Memory clock scaling may be effective for compute-intensive workloads because scaling down the memory clock would not significantly affect their execution time.

5.5 Suggested Modification to improve GPU Architectures

This thesis suggests algorithmic changes and careful choice of data structures based on the architecture. Our experiments with the seam carving kernels and full

application also suggest that some hardware changes can significantly improve the performance. For most image/video processing applications and other applications that perform many operations on short and byte data types, GPUs could offer better performance if their execution units were also vectirized to increase the parallelism of short-width operands, see Chapter 4.

CHAPTER 6

AN ANALYSIS OF PROGRAMMING EFFORT

While many papers evaluate GPU/SIMD implementations of varying applications, we are not aware of any that discuss the undertaken programming effort. In this chapter, we attempt to quantify our implementation effort for various versions of the kernels and applications that we studied in this thesis; this includes the optimized single-threaded, SIMD, multi-core, and the GPU versions. We believe that sharing such experiences, give valuable insights to researchers and engineers for deciding whether the SIMD or GPU implementation effort is worth the anticipated performance gain. Table 5 summarizes our approximated programming efforts in terms of one graduate student hour. We do not quantify the effort for the baseline implementations because it depends on the algorithms and does not provide any useful insights for this study.

6.1 Learning Curve for Intel SSE/AVX and CUDA

The programming effort in Table 5 is based on a programmer with SIMD and CUDA programming experience. It is however also important to comment on the learning curve. For CUDA, the learning curve is similar to threaded C programming; however, large performance gains require mapping the programs to specific underlying architecture, which worsens the learning curve. This learning curve has, in many cases, alienated many potential CUDA programmers. To help increase CUDA

usage, NVIDIA provides webinars and online lectures through university partnerships and offers the necessary tools and most of the CUDA libraries for free. The available resources have softened the learning curve.

SSE and AVX also have a steep learning curve; but unlike CUDA, good documentation on the subject is scarce. Most of the documentation consist of reference manuals [2, 3] listing available instructions and short tutorials. Intel's optimization reference manual [38] provides a better discussion on SSE/AVX and general SIMD design concerns. However, it is very low level in nature. The Intel Intrinsics Guide [39] provides a list of high-level intrinsics functions with short descriptions, which is very helpful for programming, but do not provide detailed information about SSE/AVX. Finally, good tools and libraries such as the Intel C++ compiler, which supports the vector math library (VML), are not available for free.

Perhaps the easiest way to exploit SSE/AVX is through compiler auto-vectorization. This, however, must not be taken for granted because it requires careful choice of algorithms and data structures as discussed in Chapter 4.

6.2 Performance per Effort Hours (PGPEH) Metric

In order to compare efficiency of the implementation effort across different implementations and different benchmarks, we define a new metric called *Performance Gain Per Effort Hours or PGPEH* that quantifies the efficiency in effort. The PGPEH metric provides a good insight into the performance gained for every hour spent on the various implementations of the kernels and applications. PGPEH is not a constant that we can use to estimate the overall performance that could be

achieved if we continue to work on improving the kernels. Instead, PGPEH illustrates the efficiency of the effort and provides us with a way to make comparisons between different platform implementations. A higher PGPEH does not imply an overall higher performance improvement; it tells us that we achieved a higher speedup per effort hour invested in a particular implementation.

6.3 Evaluation of Programming Effort

Table 5 shows that multithreading the kernels with pthreads requires approximately one effort hour, plus the effort to optimize and/or vectorize the kernels (shown as ST/SIMD+MT). The MPI and full application implementations are more complex and require 5 to 10 hours. The process to produce a fine-tuned single-threaded implementation is well illustrated by our PGPEH metric, which shows that a 47% speedup was achieved for every hour spent optimizing the gradient. The stencil kernel gains a 7*x* speedup for one effort hour for the single-threaded implementation (ST). The best efficiency is obtained with the GPU implementation of the DEsolver and stencil kernels, with a PGPEH of 20.46 (6x better PGPEH than best CPU's) and 30.65 (1.4x better PGPEH than base SIMD), respectively. The stencil kernel's PGPEH drops to 17.27 (because of the 10 extra hours) for optimized GPU implementation suggesting that it is not worth spending the extra hours if the 61.3*x* speedup suffice.

For seam carving, the energy-update (EU) algorithm adds an extra 3 hours (table not shown). We combined the energy-update with the SIMD and GPU versions for the resizing of a 1920x1080 video. The resulting effort for the SIMD_EU and GPU_EU is 29 and 43 hours, with PGPEH of 43% and 12%, respectively (table not shown). Thus,

using SIMD CPUs to resizing an HD video with SC is not only faster (12.49x over base) than the GPU (5.25x over base), but requires less effort and the gained for every effort hour is much higher. Better results are found in our CPU-GPU heterogeneous version, which requires 40 hours that result in 21.39x speedup – a PGPEH of 53%.

Table 5: A quantification of the programming effort. *: includes the analysis effort to evaluate the kernel for use of shorter operands. C: compiler auto-vectorized, only accounts for ST Opt. effort. **: Multi-core effort only. NPI: No performance improvement; we do not calculate the PGPEH since there was no improvement. The ST_AV column shows if the single-threaded optimization is required for auto-vectorization. N/A: no room for optimization; or could not implement in the platform; or we used the optimized kernels for full applications. SAV: ST Needed for Auto Vectorization.

Kernel / Application	Programming Effort								
	ST Opt.		SAV	Base SIMD		Opt. SIMD		Multi-core	
	hr	PGPEH		hr	PGPEH	hr	PGPEH	hr	PGPEH
mri-q	2	0.91	N/A	N/A	N/A	5	1.22	5+1	2.52
stencil	1	7.02	*yes*	1 C	23.33	4	5.87	1+1	16.66
histogram	N/A	N/A	N/A	2	NPI	N/A	N/A	3**	NPI
gradient	5	0.47	*yes*	5 C	0.73	14	2.39	5+1	1.11
dynamic programming	2*	0.855	*yes*	2 C	2.45	8	1.38	2**	NPI
matrix resizing	1	2.02	N/A	3	1.1	N/A	N/A	1**	NPI
seam carving	8	0.29	*yes:*	N/A	N/A	26	0.75	26+5	0.94
Laplacian	N/A	N/A	N/A	2	1.04	N/A	N/A	1**	NPI
DEsolver	5	0.36	N/A	N/A	N/A	N/A	N/A	5+5	3.61
CAPPS	5	0.36	N/A	N/A	N/A	N/A	N/A	5+10	2.52

Kernel / Application	Programming Effort					
	Best CPU		Base GPU		Opt. GPU	
	hr	PGPEH	hr	PGPEH	hr	PGPEH
mri-q	6	2.52	N/A	N/A	12	9.81
stencil	2	16.66	2	30.65	10	17.27
histogram	N/A	N/A	N/A	N/A	16	NPI
gradient	14	2.39	7	5.84	24	4.275
dynamic programming	8	1.38	6	0.27	N/A	N/A
matrix resizing	3	1.1	3	2.29	N/A	N/A
seam carving	31	0.94	N/A	N/A	40	0.085
Laplacian	2	1.04	5	3.14	N/A	N/A
DEsolver	10	3.61	3	20.46	N/A	N/A
CAPPS	15	2.52	N/A	N/A	13	4.46

CHAPTER 7

CONCLUSION AND RELATED WORK

General-purpose computation on GPUs (GPGPUs) has been an active research topic. Extensive work has been published on GPGPU computation; this is well summarized in [5]. A number of studies [1, 6, 37, 40, 41] discuss similar kernels and applications as in this thesis. Many of them focus on mapping the kernel/application onto GPU efficiently. Their GPU-optimized implementations are often compared only with single-threaded CPU baseline. Sometimes multi-threading is also evaluated, however, SIMD is often neglected. An exception is [1], where, as in this work, authors present a fair performance evaluation by utilizing all available hardware resources. However, in [1], energy-efficiency has not been studied. A few recent papers [4, 5] evaluate energy-efficiency. Different from previous work, we have shown that kernel-only evaluation is not sufficient to draw conclusions for performance and energy-efficiency. Furthermore, to the best of our knowledge, this work is the first to combine a detailed characterization and performance evaluation of kernels and full applications with a quantification of the programming effort for various platform-specific implementations.

In this thesis, aside from kernels, we studied two full applications that utilize several of these kernels. We evaluate kernel-only and full-application performances and energy-efficiencies separately, which we have not seen done in previous work. Several papers [19, 37, 40, 41] explore GPU implementation of the seam carving

application. In [40], a different algorithm is proposed to help parallelization. In [37], a heuristics is used to eliminate the dynamic programming in the seam matrix computation of seam carving. Changing the algorithm entirely may help parallelization but it may also reduce the quality of the resized image/video. [41] focused on optimizing and parallelizing the original seam carving algorithm [7]. However, they evaluate the removal of one seam, which is only part of the resizing operation. They also have not evaluated kernels and the full-application separately. [40] and [41] compare their seam carving implementation against the single-threaded CPU baseline only. In addition, none of the prior seam carving work evaluates energy-efficiency. In this thesis, we show that true heterogeneous implementation utilizes the best hardware resources for the seam carving operation to provide best performance and energy efficiency.

CAPPS is implemented using MPI in [8], which explores the performance of a CPU-only cluster with 16 cores. In this thesis, we utilize a 60-core cluster and a CPU-GPU heterogeneous system, and evaluate the performance and energy consumption of the systems.

In this thesis, we exploit the highly-parallel computational capabilities of CUDA-capable GPUs and multi-core SIMD CPUs to evaluate the performance and energy-efficiency of eight kernels and two full applications. For all of these applications, we fairly utilize the hardware capabilities of both CPUs and GPUs. The compute-intensive parts of the applications have been parallelized using a combination of SIMD, *pthreads*, MPI, and CUDA.

We have evaluated 15 optimization techniques to utilize hardware resources in both CPUs and GPUs. Our results show that only when all appropriate optimizations have been applied, a fair comparison between CPUs and GPUs can be made. We have also found that kernel-only performance and energy-efficiency evaluation may be misleading because of the way a kernel might be used in an application and therefore true results must be obtained using full applications. The best-performing platform for each of our kernels and applications vary. The GPU is best for data-parallel scientific application and kernels such as, CAPPS, mri-q, stencil, gradient and matrix resizing. The CPU is best for the histogram and dynamic programming kernels. Finally, a heterogeneous CPU-GPU implementation is best for applications with diverse kernels such as seam carving.

We have observed that data width has a profound effect on the performance of SIMD implementations and therefore we have drawn attention into choice of operand width and value scaling in applications. Finally, we discuss the programming effort for various implementations of the studied kernels and applications. In order to compare efficiency of effort across different benchmarks and platforms, we have defined a new metric called Performance gain Per Effort Hours or PGPEH.

REFERENCES

[1] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P., "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *Proceedings ACM International Symposium on Computer Architecture*, June 2010, pp. 451-460.

[2] Intel, "SSE4 Programming Reference", Available at: http://software.intel. com/ sites/default/files/m/9/4/2/d/5/17971-intel_20sse4_20programming_20 reference. pdf, April 2007.

[3] Intel, "2nd Generation Intel Core Processor Family Desktop," Available at: http://www.intel.com/ content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html, 2012.

[4] Huang S., Xiao, S., Feng, W., "On the Energy Efficiency of Graphics Processing Units for Scientific Computing," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2009.

[5] Cebrian, J., Guerrero, G., and Garcia, J., "Energy efficiency analysis of GPUs," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2012.

[6] Stratton, J., Rodrigues, R., Sung, I., Obeid, N., Chang, L., Anssari, N., Liu, G., Hwu, W., "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *IMPACT Technical Report*, IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[7] Avidan, S. and Shamir, A., "Seam Carving for Content-Aware Image Resizing," in *Proceedings ACM SIGGRAPH*, 26, July 2007.

[8] Amani, M., Vetter, F. J., "Unstable spiral waves in a 2D numerical model of myocardium," in *Proceedings of the 35th Annual Northeast Bioengineering Conference*, April 2009, pp. 66-67.

[9] Stone, S., Haldar, J., Tsao, S., Hwu, W., Liang, Z., Sutton, B., "Accelerating Advanced MRI Reconstructions on GPUs," in *Proceedings of conference on Computing frontiers*, 2008, pp.261-272.

[10] Itti, L., Koch, C., Niebur, E., "A model of saliency-based visual attention for rapid scene analysis," in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Nov. 1998, pp. 1254–1259.

[11] Suh, B., Ling, H., Bederson, B., Jacobs, D., "Automatic thumbnail cropping and its effectiveness," in *Proceedings of User Interface Software and Technology*, 2003, pp. 95–104.

[12] Chen, L., Xie, X., Fan, X., "A visual attention model for adapting images on small displays," *Multimedia Systems*, 9, 2003, pp. 353–364.

[13] Ciocca, G., Cusano, C., Gasparini, F., Schettini, R., "Self-adaptive image cropping for small displays," in *IEEE International Conference on Consumer Electronics*, 2007.

[14] Santella, A., Agrawala, M., DeCarlo, D., Salesin, D., Cohen, M., "Gaze-based interaction for semi-automatic photo cropping," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006, pp. 771–780.

[15] Adobe, “Photoshop Support”, Available at: http://www.photoshopsupport.com/ photoshop-cs4/what-is- new-in-photoshop-cs4.html, 2012.

[16] Wikidots Liquid Rescale, "Content-aware resizing for the GIMP" Available: http:// liquidrescale.wikidot.com, 2010.

[17] Digikam, “Liquid Rescale tool” Available at: http://www.digikam.org/node/439, 2012.

[18] ImageMagick, “Resize or Scaling (General Techniques)”. Available at: http://www.imagemagick. org/Usage/resize, 2012.

[19] Duarte, R. and Sendag, R., "Accelerating and Characterizing Seam Carving Using a Heterogeneous CPU-GPU System," in *Proceedings of the 18th Annual International Conference on Parallel and Distributed Processing Techniques and Application*, July 2012, pp. 658-663.

[20] Thilagam, K., Karthikeyan, S., "Optimized Image Resizing using Piecewise Seam Carving," in *International Journal of Computer Applications*, 2012, pp. 24-30.

[21] Conge, D., Kumar, M., Miller, R., Luo, J., Radha, H., "Improved seam carving for image resizing," In *IEEE Workshop on Signal Processing Systems*, Oct. 2010, pp. 345-349.

[22] Achanta, R., and Süsstrunk, S., "Saliency detection for content-aware image resizing," *16th IEEE Inernationalt Conference on Image Processing*, Nov. 2009.

[23] Gonzalez, R. C., Woods, R. E., “Image Enhancement in the Spatial Domain,” *Digital Image Processing*, 2002, pp.125-127.

[24] Xu, L., Lu, C., Xu, Y., Jia, J., " Image smoothing via L_0 gradient minimization," In *Proceedings of ACM SIGGRAPH Asia*, Dec. 2011, Vol. 30, No. 174.

[25] Sun, J., Sun, J., Xu, Z., Shum, H., "Image super-resolution using gradient profile prior," *IEEE Conference on Computer Vision and Pattern Recognition*, 2008, pp. 1-8.

[26] Scharstein, D., "Matching images by comparing their gradient fields," in *Proceedings of the 12th IAPR International Conference on Computer Vision & Image Processing*, 1994.

[27] MathWorks, "Matlab" Available at: http://www.mathworks.com/products/matlab/, 2013.

[28] Pandit, S. V., Clark, R. B., Giles, W. R., Demir, S. S., "A mathematical model of action potential heterogeneity in adult rat left ventricular myocytes," *Biophysical Journal*, 81, Dec. 2001, pp. 3029-51.

[29] Fast V. G., "Effects of electrical shocks on Cai2+ and Vm in myocyte cultures," *Circulation Research*, 94, May 2004, pp. 1589-1597.

[30] NVIDIA, "NVIDIA GeForce GTX 580 GPU Datasheet," Available at: http://www.geforce.com/ Active/en_US/en_US/pdf/GTX-580-Web-Datasheet-Final.pdf, 2010.

[31] NVIDIA, "NVIDIA CUDA C Programming Guide 4.2," Available at : http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.

[32] VDT Math, "The VDT Mathematical Library", Available at: https://svnweb.cern.ch/trac/vdt, 2012.

[33] Abel, J., "Applications Tuning for Streaming SIMD Extensions," *Intel Technology Journal*, Q2, 1999.

[34] Mistry, P., Schaa, D., Jang, B., Kaeli, D., Dvornik, A., Meglan, D., "Data structures and transformations for physically based simulation on a GPU," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, 2010, pp. 162-171.

[35] ILCS, "AVX-optimized," Available at: software-lisc.fbk.eu/avx_mathfun, 2012

[36] Jang, B., Schaa, D., Mistry, P., Kaeli, D., "Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures", in *IEEE Transactions on Parallel and Distributed Systems*, Jan. 2011, pp. 105-118.

[37] Hua, H., Fu, T., Rosin, P. L., Qi, C. "Real-time Content-aware Image Resizing," *Science in China Series F: Information Sciences*, 52, Feb. 2009 pp. 172-182.

[38] Intel, "Intel Architectures Optimization Reference Manual," Available at: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, 2012

[39] Intel, "Intel Intrinsics Guide" Available at: http://software.intel.com/en-us/articles/intel-intrinsics-guide, 2013.

[40] Chiang, C., Wang, S., Chen, Y., Lai, S., " Fast AND-Based Video Carving with GPU Acceleration for Real-Time Video Retargeting," in *IEEE Transactions on Circuits and Systems for Video Technology*, 19, 2009, pp. 1588–1597.

[41] Češnovar, R., Bulić, P., Dobravec, T., "Optimization of a single seam removal using a GPU" in *Proceedings of the 17th Annual International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011.

BIBLIOGRAPHY

Abel, J., "Applications Tuning for Streaming SIMD Extensions," *Intel Technology Journal*, Q2, 1999.

Achanta, R., and Süsstrunk, S., "Saliency detection for content-aware image resizing," *16th IEEE Inernationalt Conference on Image Processing*, Nov. 2009.

Adobe, “Photoshop Support”, Available at: http://www.photoshopsupport.com/photoshop-cs4/what-is- new-in-photoshop-cs4.html, 2012

Amani, M., Vetter, F. J., “Unstable spiral waves in a 2D numerical model of myocardium,” in *Proceedings of the 35th Annual Northeast Bioengineering Conference*, April 2009, pp. 66-67.

Avidan, S. and Shamir, A., "Seam Carving for Content-Aware Image Resizing," in *Proceedings ACM SIGGRAPH*, 26, July 2007.

Cebrian, J., Guerrero, G., and Garcia, J., “Energy efficiency analysis of GPUs,” in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2012.

Češnovar, R., Bulić, P., Dobravec, T., "Optimization of a single seam removal using a GPU" in *Proceedings of the 17th Annual International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011.

Chen, L., Xie, X., Fan, X., "A visual attention model for adapting images on small displays," *Multimedia Systems*, 9, 2003, pp. 353–364.

Chiang, C., Wang, S., Chen, Y., Lai, S., " Fast AND-Based Video Carving with GPU Acceleration for Real-Time Video Retargeting," in *IEEE Transactions on Circuits and Systems for Video Technology*, 19, 2009, pp. 1588–1597.

Ciocca, G., Cusano, C., Gasparini, F., Schettini, R., "Self-adaptive image cropping for small displays," in *IEEE International Conference on Consumer Electronics*, 2007.

Conge, D., Kumar, M., Miller, R., Luo, J., Radha, H., "Improved seam carving for image resizing," In *IEEE Workshop on Signal Processing Systems*, Oct. 2010, pp. 345-349.

Digikam, "Liquid Rescale tool" Available at: http://www.digikam.org/node/439, 2012

Duarte, R. and Sendag, R., "Accelerating and Characterizing Seam Carving Using a Heterogeneous CPU-GPU System," in *Proceedings of the 18th Annual International Conference on Parallel and Distributed Processing Techniques and Application*, July 2012, pp. 658-663.

Fast V. G., "Effects of electrical shocks on Cai2+ and Vm in myocyte cultures," *Circulation Research*, 94, May 2004, pp. 1589-1597.

Gonzalez, R. C., Woods, R. E., "Image Enhancement in the Spatial Domain," *Digital Image Processing*, 2002, pp.125-127.

Hua, H., Fu, T., Rosin, P. L., Qi, C. "Real-time Content-aware Image Resizing," *Science in China Series F: Information Sciences*, 52, Feb. 2009 pp. 172-182.

Huang S., Xiao, S., Feng, W., "On the Energy Efficiency of Graphics Processing Units for Scientific Computing," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2009.

ILCS, “AVX-optimized,” Available at: software-lisc.fbk.eu/avx_mathfun, 2012

ImageMagick, “Resize or Scaling (General Techniques)”. Available at: http://www.imagemagick. org/Usage/resize, 2012

Intel, “SSE4 Programming Reference”, Available at: http://software.intel.com/sites/default/files/m/9/4/2/d/5/17971-intel_20sse4_20programming_20reference.pdf, April 2007.

Intel, "2nd Generation Intel Core Processor Family Desktop," Available at: http://www.intel.com/ content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html, 2012.

Intel, “Intel Architectures Optimization Reference Manual,” Available at: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, 2012.

Intel, "Intel Intrinsics Guide" Available at: http://software.intel.com/en-us/ articles/intel-intrinsics-guide, 2013.

Itti, L., Koch, C., Niebur, E., "A model of saliency-based visual attention for rapid scene analysis," in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Nov. 1998, pp. 1254–1259.

Jang, B., Schaa, D., Mistry, P., Kaeli, D., "Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures", in *IEEE Transactions on Parallel and Distributed Systems*, Jan. 2011, pp. 105-118.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P., “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput

Computing on CPU and GPU," in *Proceedings ACM International Symposium on Computer Architecture*, June 2010, pp. 451-460.

MathWorks, "Matlab" Available at: http://www.mathworks.com/products/ matlab/, 2013.

Mistry, P., Schaa, D., Jang, B., Kaeli, D., Dvornik, A., Meglan, D., "Data structures and transformations for physically based simulation on a GPU," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, 2010, pp. 162-171.

NVIDIA, "NVIDIA GeForce GTX 580 GPU Datasheet," Available at: http://www.geforce.com/ Active/en_US/en_US/pdf/GTX-580-Web-Datasheet-Final.pdf, 2010.

NVIDIA, "NVIDIA CUDA C Programming Guide 4.2," Available at : http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.

Pandit, S. V., Clark, R. B., Giles, W. R., Demir, S. S., "A mathematical model of action potential heterogeneity in adult rat left ventricular myocytes," *Biophysical Journal*, 81, Dec. 2001, pp. 3029-51.

Santella, A., Agrawala, M., DeCarlo, D., Salesin, D., Cohen, M., "Gaze-based interaction for semi-automatic photo cropping," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006, pp. 771–780.

Scharstein, D., "Matching images by comparing their gradient fields," in *Proceedings of the 12th IAPR International Conference on Computer Vision & Image Processing*, 1994.

Stone, S., Haldar, J., Tsao, S., Hwu, W., Liang, Z., Sutton, B., "Accelerating Advanced MRI Reconstructions on GPUs," in *Proceedings of conference on Computing frontiers*, 2008, pp.261-272.

Stratton, J., Rodrigues, R., Sung, I., Obeid, N., Chang, L., Anssari, N., Liu, G., Hwu, W., “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” *IMPACT Technical Report*, IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

Suh, B., Ling, H., Bederson, B., Jacobs, D., "Automatic thumbnail cropping and its effectiveness," in *Proceedings of User Interface Software and Technology*, 2003, pp. 95–104.

Sun, J., Sun, J., Xu, Z., Shum, H., "Image super-resolution using gradient profile prior," *IEEE Conference on Computer Vision and Pattern Recognition*, 2008, pp. 1-8

Thilagam, K., Karthikeyan, S., "Optimized Image Resizing using Piecewise Seam Carving," in *International Journal of Computer Applications*, 2012, pp. 24-30.

VDT Math, "The VDT Mathematical Library", Available at: https://svnweb.cern.ch/trac/vdt, 2012.

Wikidots Liquid Rescale, "Content-aware resizing for the GIMP" Available: http://liquidrescale.wikidot.com, 2010.

Xu, L., Lu, C., Xu, Y., Jia, J., " Image smoothing via L_0 gradient minimization," In *Proceedings of ACM SIGGRAPH Asia*, Dec. 2011, Vol. 30, No. 174.