On the Performance and Energy-efficiency of Multi-core SIMD CPUs and CUDA-enabled GPUs

Ronald Duarte, Resit Sendag, Frederick J. Vetter
Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island
Kingston, RI, USA
rduarte26@my.uri.edu, {sendag, vetter}@ele.uri.edu

***Abstract*—This paper explores the performance and energy efficiency of CUDA-enabled GPUs and multi-core SIMD CPUs using a set of kernels and full applications. 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 for our CPU-only and CPU-GPU platforms. To fairly study performance and energy-efficiency, we also used two applications which utilize several kernels. Finally, we present an evaluation of the implementation effort required to efficiently utilize multi-core SIMD CPUs and CUDA-enabled GPUs for the benchmarks studied. Our results show that kernel-only performance and energy-efficiency could be misleading when evaluating parallel hardware; therefore, true results must be obtained using full applications. We show that, after all respective optimizations have been made, the best performing and energy-efficient platform varies for different benchmarks. Finally, our results show that PPEH (Performance gain Per Effort Hours), our newly introduced metric, can affectively be used to quantify efficiency of implementation effort across different benchmarks and platforms.**

Keywords-SIMD;CPU; GPU; Programming effort;

I. 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].

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 paper, 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 [29, 30] 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] in which performance was compared by applying optimizations appropriate for both GPU and CPU. However, energy efficiency was not evaluated. In this paper, we carefully fine-tuned our kernels and applications 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. 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 can help other researchers make informed decisions. In this paper, we attempt to fairly quantify the implementation effort and share

Table 1: Kernel characteristics. We studied three kernels from the Parboil [23] benchmarks. Seam Carving [3] and CAPPS [4] are the two full applications that we studied each utilizing three and two kernels, respectively, as in the table.

	Kernel	Applications	Characteristics
Parboil [23] benchmarks	mri-q	medical imaging	Compute bound
	stencil	scientific computation, image processing	Compute bound /Bandwidth bound
	histogram	image analysis, statistics	Reduction/synchronization bound
Seam carving [3]	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 [4]	DEsolver	dense linear algebra, scientific computation	Compute bound
	Laplacian	image processing, physics simulation	Compute bound/bandwidth bound

our experiences. Overall, this paper makes the following contributions:

- 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.
- We demonstrate that GPUs facilitate low-cost and energy-efficient computing for computationally intensive numerical applications, such as numerical simulations of cardiac action potential propagation (CAPPS); but also show that applications, such as seam carving (SC), achieve best performance and energy efficiency by efficiently utilizing the true heterogeneity of a CPU-GPU system.
- We show that evaluating the performance and the energy-efficiency of computing platforms by using *only* kernels may lead to incorrect conclusions.
- We show that reducing data width has a profound effect on the performance of SIMD implementations.
- We quantify the implementation effort in writing the SIMD, multithreaded, and CUDA versions of the applications and define a new metric to compare them.

II. THE WORKLOADS

In this work, we studied eight kernels as listed in Table 1. We used the data-parallel kernels, `mri-q`, `stencil` and `histogram` from the *Parboil* [23] benchmarks with diverse data access and communication patterns, and two full applications, `Seam Carving` [3] and `CAPPS` [4] 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.

A. Parboil Benchmarks

`mri-q` computes a matrix Q, representing the scanner configuration for calibration, used in a 3-D magnetic resonance image reconstruction algorithm in non-Cartesian space. `stencil` is an iterative Jacobi stencil operation on a regular 3-D grid. Finally, `histogram` computes a moderately large, 2-D saturating histogram with a maximum bin count of 255. Input datasets represent a silicon wafer validation application in which the input points are distributed in a roughly 2-D Gaussian pattern. For a more detailed descriptions for Parboil benchmarks, refer to [23].

B. Seam Carving

Seam Carving (SC) is a popular content-aware image and video resizing method [3] which has been shown to effectively resize images and videos with little to no perceptible distortion. Other traditional image resizing techniques, such as cropping and scaling [5-7], are oblivious to the content of the image when changing its width or height and thus result in informational loss or image distortion. SC has been widely adapted by popular graphics editing applications, which include Adobe Photoshop, where it is called Content Aware Scaling (CAS) [8], GIMP [9], digiKam [10], and ImageMagick [11]. The importance of content-aware image resizing has made SC a popular application for research [14, 18, 19].

SC 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). The three kernels in the SC 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.

`gradient`: SC is able to utilize several energy functions [3]. In this paper, we use the magnitude of the gradient [13] for the computation of the energy function because the gradient is a highly used kernel [20-22]. Therefore, the characterization and any improvements of the gradient operation will benefit a large range of applications.

`dynamic programming`: In the second phase of SC, we use dynamic programming to compute the cumulative energy sum at every pixel. The last row of the seam matrix contains the total energy of the seams. This dynamic programming approach produces the optimal seams [3]. However, the computation for each element is entirely dependent on the result of the three above connected

elements, which makes the parallel operation challenging.

`matrix resizing`: The last phase of SC is the removal or duplication of low-energy seams and thereby resizing the image. Matrix resizing is widely used in signal processing. *Matlab* [24] has resizing functions based on removing columns/rows of a matrix. Accelerating and characterizing `matrix resizing` will benefit many applications. For a detailed description about SC, refer to [3].

C. Numerical Simulations of Cardiac Action Potential Propagation (CAPPS)

CAPPS [4] implements a model for numerical simulations of electrical activity (specifically, propagation of cardiac action potential) in the heart. Used in arrhythmia research, this model is very computationally intensive and takes days to run on a high-end workstation.

In [4], the authors characterized the convergence properties and numerical stability of a recent model of the rat ventricular action. 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.

CAPPS utilizes two kernels: the `DEsolver` 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); and the `Laplacian` to compute the transmembrane voltage.

III. Hardware Resources

A. High-performance Desktop Computer (HPDC)

The HPDC is a heterogeneous CPU-GPU computer composed of a single Intel Core i7-2600K CPU and an NVIDIA GTX580 GPU. The GTX580 has 512 cores organized into 16 SMs with dual warp schedulers. A warp consists of 32 parallel threads executing in lockstep [12]. Ubuntu Linux 10.04 is the operating system installed. The system has 8GB 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*.

B. CPU-only Cluster

A 60-core cluster computer was utilized for CAPPS. 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 system supports 128-bit SSE instructions. However CAPPS, the only application that utilizes this system, does not use SIMD for reasons mentioned in Section 4. CAPPS was compiled with full optimization using *gcc 4.5* on this system.

C. 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. Results include the energy consumption of idle CPUs, if any, when the kernels are executing. We compare the energy efficiency of tested platforms using energy-delay product (EDP) as a metric.

IV. 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.

A. Optimized Single-Threaded Implementations (ST)

The baseline implementation of the `gradient` is similar to the implementation described in [14]. We improve the baseline by applying several hand optimizations as listed in Table 3, including *AO*, *BE*, *SPD*, and *LF*.

The `dynamic programming` kernel computes the minimum cumulative energy sum (`CES`) for the SC application. In the baseline implementation, we use the C++ *min* function to find the minimum value of the above-connected pixels, and add the result to the pixel's energy to obtain the `CES` value. We locate the lowest-energy seams by searching the last row of the seam matrix. The baseline implementation inherits many optimizations techniques used for the `gradient`. *BE* 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 SC application where non-resizing operations account for a much larger fraction of the execution time, the overall SC performance would have been negatively affected. Thus, we are forced to implement the resizing using array data structures.

Table 2: Description of different optimization techniques used in the implementations of kernels

Optimization	Description
SPD	**Smart Pointer Dereferencing:** Reduces the number of memory accesses by dereferencing pointers at the beginning of kernels, instead of inside loop iterations (most useful when kernels access data elements that are encapsulated in a *C-Struct*, or multi-level arrays).
AO	**Arithmetic Optimizations**: Simplifies arithmetic to reduce the number of operations, and 2D to 1D index transformations.
LF	**Loop Fusion**: Improves locality and cache performance by fusing loops to perform computation with a single loop pass.
SVS	**Smart Value Scaling**: Scales values by a smart fraction to replace division operation by logical shift (when the kernel tolerates errors).
LI	**Loop Interchange:** Exchanges the order of nested loops to improving locality of access and take advantage of cache.
DST	**Data Structure Transformation:** Transforms data structure to improve memory performance, for example transforming AoS to SoA.
BE	**Branch Elimination:** Eliminates unnecessary branches by executing the boundaries conditions outside of the loops (improves ILP).
RWO	**Reduced-width Operands:** Reduces the data width to improve cache performance and SIMD parallelization.
FM	**Fast Math:** Optimized math functions [12, 26] to improve arithmetic performance (reduces accuracy, not noticeable in some kernels).
PPB	**Ping-Pong Buffering:** Improves performance by eliminating redundant memory copy and branches.
AP	**Array Padding:** Improves performance by guaranteeing that every matrix row starts on an aligned memory location or new cache line.
SMC	**Shared Memory Caching**: Improves performance by reducing the number of same-data memory access.
TC	**Texture Cache:** Hardware managed, optimized for 2D spatial locality (benefits kernels with irregular access patters or low locality).
LT	**Lookup Table:** Eliminates multiple computations of functions (same input) by pre-computing outputs for all inputs (store in memory).
SSI	**SIMD Shift and Insert:** When data is loaded into SIMD register, eliminates extract loads of nearby data (by data shift and insert).
COT	**CPU Optimization Techniques:** SIMD version incorporates all of the non-SIMD CPU Optimization previously used by the kernel.

Table 3: Optimizations (which are listed in Table 2) applied for CPU implementations (single-threaded and multi-threaded) and the GPU implementations. SC and CAPPS inherit all optimizations of their kernels. SC tolerates the scaling error and uses SVS to improve performance.

Kernels	Optimization																		
	CPU									SIMD				GPU					
	SPD	AO	LF	BE	RWO	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			
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			
DEsolver	x	x						x						x						x

As we discussed in Section 2, CAPPS utilizes two kernels: the `DEsolver` and the `Laplacian`. We implemented these kernels using the equations given in [4]. We used the optimization techniques discussed for `gradient` for achieving optimal performance. Finally, instead of computing exponential, 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 benchmarks suite [23]. We improved the baseline implementation for `mri-q` (by about 86%) by applying *FM* optimizations. We applied *LI* optimizations for `stencil`, which greatly improved locality and resulted in a 7*x* performance gain over the baseline.

B. Multi-threaded Implementations (MT)

For multi-threaded `gradient`, we partition the input image into tiles consisting of consecutive rows; a column-based division reduces locality. The number of rows in a tile depends on the number of threads and the image height.

Unlike the `gradient`, the `CES` 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 by dividing each row into fixed-width tiles and compute these tiles in parallel. We synchronize all threads after the execution of each row.

For `matrix resizing`, `mri-q`, `stencil`, `histogram`, `DEsolver`, and `Laplacian`, the multi-threaded implementation is similar to the `gradient`. `histogram` uses the reduction technique, where each thread first updates a thread-private local histogram and then "reduces" (adds) it 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 Section 3.

C. SIMD implementations (SIMD)

The *gcc 4.7* compiler is capable of auto-vectorizing to explore the potential of the SIMD units on the CPU. But, for successful compiler auto-vectorization, often the programmer needs to write the code appropriately. Besides the `Laplacian`, no other baseline implementations for the kernels in this paper were successfully auto-vectorized.

Optimizations such as *DST*, *LI*, *SVS* and *BE* have helped generating auto-vectorized code with some success for four of the eight optimized kernels: `gradient`, `dynamic programming`, `stencil`, and `Laplacian`. However, the SIMD units achieve much higher performance if the code is carefully vectorized by hand.

For `gradient`, after using *SVS* optimizations, *gcc 4.7* was able to auto-vectorize the code (with 3.67*x* performance gain). By further using *DST* and *SSI* optimizations, we achieve a 33*x* performance gain over the baseline with a hand-tuned SIMD implementation.

By moving the `CES` computation of the first and the last column outside of the loop, we reduce the boundary check instructions (*BE*) in the `dynamic programming` kernel. This optimization helps the compiler vectorize the `CES` computation (4.89*x* performance gain over the baseline). By implementing the SIMD Shift and Insert (*SSI* in Table 2) mechanisms, our hand-tuned SIMD implementation achieves 2.25*x* over the compiler auto-vectorized code.

Auto-vectorization did not work for `matrix resizing`. 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 relocate 5.33 simultaneous pixels on average.

`stencil` was auto-vectorized after we have modified the baseline (from Parboil suite [23]) with *LI* 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 `mri-q` because sine and cosine functions are not part of native SSE/AVX. We were able to hand-vectorize this kernel by implementing AVX sine and cosine with very good accuracy using [25]. The hand-tuned SIMD implementation of `mri-q` was 3.27*x* better than the optimized single-threaded implementation.

CAPPS and `DEsolver` were unable to utilize the SIMD units on the CPU due to the current SSE/AVX limitations; no support for special functions such as exponential exists. Therefore, we do not have a SIMD CAPPS implementation. We could have used SIMD with *LT* optimization, but SIMD CPUs do not have gather/scatter SIMD operations yet, which also affected the vectorization of `histogram`. We were able to utilize SIMD for the reduction phase of the multi-threaded `histogram`. Each thread-private histograms is copied to a separate global histogram (multithreaded reduction), which are then added into one global histogram using the SIMD lanes.

D. 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.

Sometimes, although not directly applicable, scaling down values when it does not hurt accuracy, allows the use of shorter operands, which has the potential to improving SIMD performance significantly. We observed an example

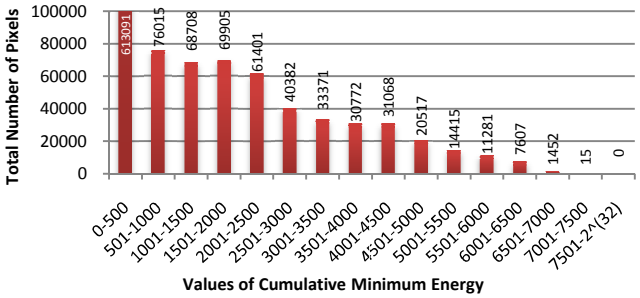

Figure 1: A histogram of the CES for a 1200x900 image.

to this in the `dynamic programming` kernel, which is used to compute the `CES` in the SC algorithm. In theory, the values of `CES` could grow beyond 64K (unsigned short). Therefore, the baseline implementation uses 32-bits (unsigned integer) to store the `CES` values. However, Figure 1 shows a histogram of the `CES` values, which reveal that the values do not exceed 7500 (without scaling). We analyzed many images with different sizes and characteristics, and found that even in very large images (and with high energy) the largest value was below 19500. This allows us to use 16-bit instead of 32-bit operands. This simple optimization doubled the performance of the SIMD implementation for `dynamic programming`. Short operands also improved the non-SIMD implementation by 24%, due to cache performance.

E. GPU implementations (GPU)

The need for accessing neighboring pixels to compute the `gradient` strongly influences the way we access memory on the GPU. In [14], 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`. With 32x9-blocks configuration, each thread loads a single pixel; achieving full coalesce accesses for the computational pixels (32x8) and the bottom-neighboring pixel. Our caching method works well, but the best performance is achieved by careful optimizations including *TC* and *FM*.

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 `matrix resizing` on the GPU, we launch one thread per data element (pixel in the case of SC) in order to achieve one data-element relocation per thread. None of our previous resizing methods or [14] achieved such high parallelization; we are able to move 100s of pixels simultaneously. To prevent neighboring threads from overwriting the pixels before they can be read, we used the *PPB* 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. The memory access patterns for voltage computation are very similar to that of the `gradient`, which points that the computation of the voltage would also benefit from the GPU. We implemented two GPU kernels: a

kernel to compute the transmembrane current (`DEsolver`) and a kernel to compute the transmembrane voltage (`Laplacian`). We placed all constants in the constant memory using the guidelines in [17]. 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 an output file (which occurs every 10,000 iterations). The optimized GPU version of CAPPS required us to apply many optimizations, including, *DST*, *PPB*, *AP*, and *LT*, as described in Table 2.

Finally, we have used the GPU implementations of `mri-q`, `stencil` and `histogram` from [23] although we have also implemented our own versions that performed similarly.

F. SC Specific Optimization: The Energy Update (EU)

In SC, when removing seams, the frequency at which the energy function (`gradient`) 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 [3]. To reduce the computation, it is possible to recompute the energy function after a predetermined amount of seams has been removed.

In this paper, we implement 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 (dark pixels, Figure 2a) are the left and right neighboring pixels. Thus, we can recompute the energy of the pixels that undergo an energy change (white pixels, Figure 2b). This approach produces the same results as recomputing the entire energy function, but the computation is greatly reduced, improving the performance significantly, as shown in Section 5.

V. Performance and Energy Evaluation

A. Parboil Benchmarks

Figure 3 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 3 shows the kernel-only performance gain, in which the GPU performs best, with dramatic speedups over the baseline: 214*x*, 773*x* and 39*x* for `mri-q`, `stencil`, and `histogram`, 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 than the baseline. We also observed a significant performance drop compared to the kernel-only times for `mri-q` and `stencil`. `stencil` undergoes a 4.7*x* reduction in performance gain due to the overhead. The CPU implementations do not incur such overhead. 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. MT implementation provides a 5% speedup over the baseline. Most of the performance gain is lost during the reduction phase of the histogram where thread-private histograms are reduced to a global histogram. Without the reduction, MT achieves a 2.77x gain over the baseline (not shown in Figure). We further utilize the SIMD lanes for the multi-threaded reduction that result in 24% speedup (mt_simd). Both the `mri-q` and the `stencil` are well suited for the GPU, achieving beyond 117*x* and 172*x* over the baseline implementation, respectively. It is important, however, to do a fair comparison between the CPU and GPU. By applying *LI* optimization, we improve the performance of base implementation of `stencil` (taken from the Parboil benchmarks) by 7*x*. By further using the SIMD unit on the CPU, a 21.9*x* performance boost is achieved using a single CPU core. Although the SIMD `stencil` does not scale linearly on multi-core CPU, we are able to improve the overall performance by 29.9*x*, with all four cores on a quad-core CPU. By properly utilizing the CPU, the performance achievement of the GPU over the CPU is 5.7*x* – a much smaller number than 773*x*! This shows the danger in comparing the platforms unfairly.

Table 4 present the energy consumption and energy-efficiency of the Parboil kernels. We measure the energy-efficiency with EDP metric. For the data-parallel `mri-q` and `stencil` kernels, GPU is the clear winner in both energy-consumption and energy-efficiency. Multi-threaded SIMD implementation provides second best energy-consumption and efficiency. For `histogram`, 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 might be misleading.

B. Seam Carving (SC)

We now discuss the performance evaluation of the SC kernels and the performance and energy-efficiency of the full SC application. We have not evaluated energy-efficiency for individual kernels separately. We conduct the kernel evaluations using a 1200x900 RGB image. Figure 4 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. With *SVS* optimization, auto-vectorization is possible. The auto-vectorized code achieves 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. Our SIMD 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 [14]) was much greater than the performance gain from

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

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

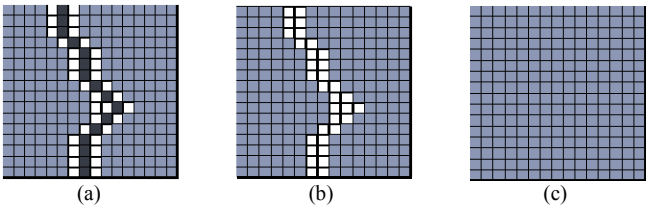

Figure 2: Our proposed Update Algorithm for recomputing the energy. **(a)** The dark (removable) pixels only affect the white pixels**. (b)** Only recompute the energy for the affected pixels.**(c)** Updated energy function.

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.

Figure 4 also illustrates the performance for the various implementations of the `dynamic programming`. By utilizing similar optimization techniques as in the `gradient`, and with the addition of *BE* and *RWO* 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 incurred by dynamic programming, both non-SIMD and SIMD implementations exhibit very poor scalability on multi-core. The ST_SIMD CPU implementation of `dynamic programming` yields the best performance, an 11*x* performance boost over the baseline and 6.84*x* over the GPU. *RWO* optimization only helps slightly with cache performance for non-SIMD implementations. For 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.

In Section 4, we presented different CPU methods for `matrix resizing`**.** 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.

Performance and Energy Evaluation of SC: 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 5 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

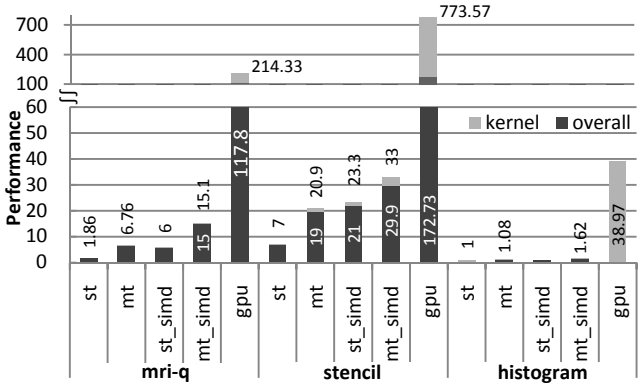

Figure 3: The computation-only and overall speedup of the Parboil kernels. For histogram, mt_simd is the same as MT, but uses SSE for the reduction. st_simd has bad performance, therefore we do not have an MT.

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. However, the GPU only gains a 61% improvement in the `dynamic programming` kernel, while the SIMD CPU achieved 11*x*. `dynamic programming` takes the second larger fraction of the execution time for the SC 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.

Instead of targeting the CPU or GPU individually, we explored the computational capability of a combined CPU-GPU heterogeneous system. Figure 6 shows that as the image size increases, a CPU-GPU heterogeneous implementation (HET/HET_EU) performs much better than a CPU- or GPU-only implementation. It does not achieve the best performance on the small and midsize images, but it is almost 2*x* faster than the best SIMD CPU implementation for the high-resolution images. Therefore, for large data set, a better approach is to utilize a true heterogeneous implementation of SC 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`. Therefore, our HET method uses the SIMD lanes for `dynamic programming` and the GPU for `gradient` and `matrix resizing`.

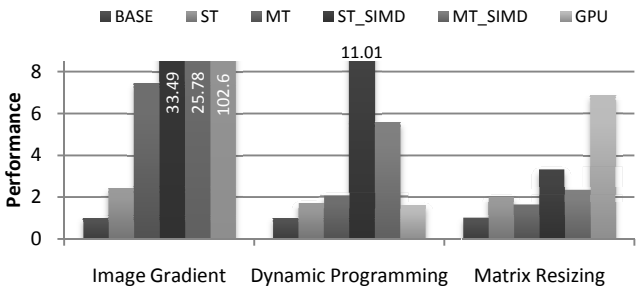

Figure 4: Performance evaluation of Seam Carving kernels.

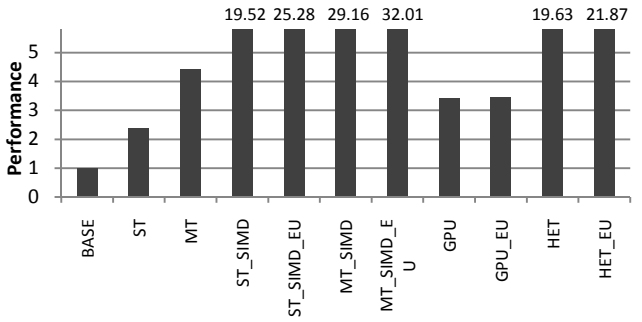

Figure 5: Performance of full Seam Carving application.

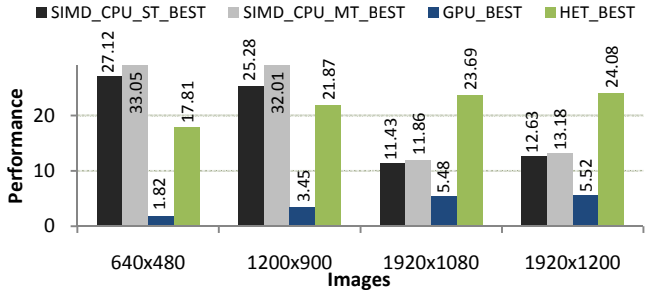

Figure 6: Performance of best platforms full SC resizing operation.

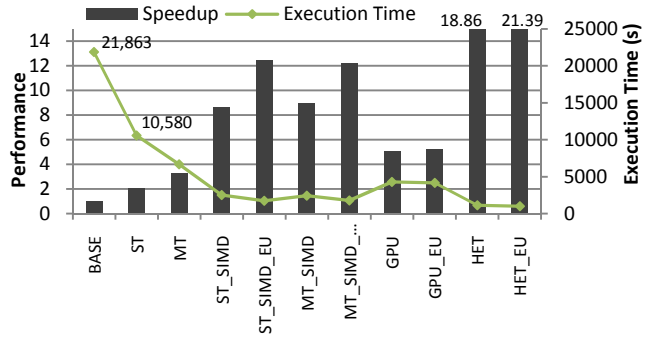

Figure 7: Performance of Seam Carving to resize one minute of video.

Figure 7 shows the execution time and performance improvement for the resizing of a HD video (1920x1080). This confirms that 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 one minute of video (by one-third of its width). By using the hardware efficiently, we are able to decrease the resizing time from 6 hours to 17 minutes. Figure 8 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. SIMD implementations provide the second best performance and energy-efficiency followed by the GPU.

C. *Performance and Energy 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 parallel version of CAPPS using MPI and ran on a cluster with 60 cores. Figure 9 shows the performance of the `DEsolver` and `Laplacian` kernels. `DEsolver` has good scalability and achieves 36.2*x* over the baseline running on a 60-core cluster. `Laplacian` 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 10 shows the results for the full 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 (2.07*x*, 4.25*x*, 8.23*x*). For 16 cores, two shared-memory systems are used to form a distributed-memory system. The network 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 11, we show the energy consumption of running one CAPPS simulation on the CPU cluster. Although the 60-core 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 10 and 11. 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, Figures10 and 11 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 10 and 11 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 Section 3. Using a CPU-GPU heterogeneous system, we are able to perform a

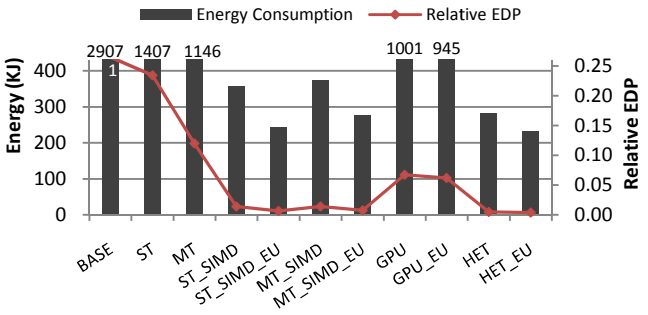

Figure 8: Energy evaluation of Seam Carving to resize 1 minute of video.

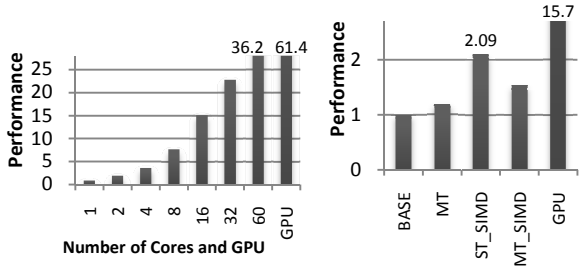

Figure 9: Performance of DEsolver (left) and Laplacian (right).

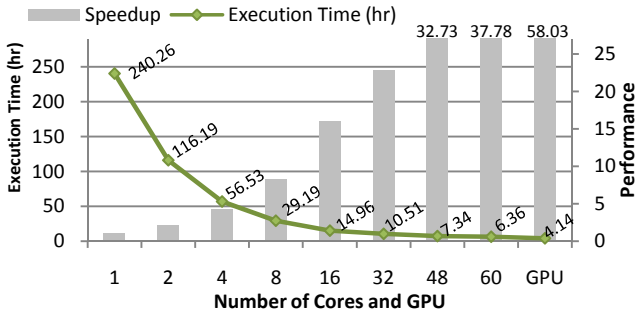

Figure 10: Performance of CAPPS.

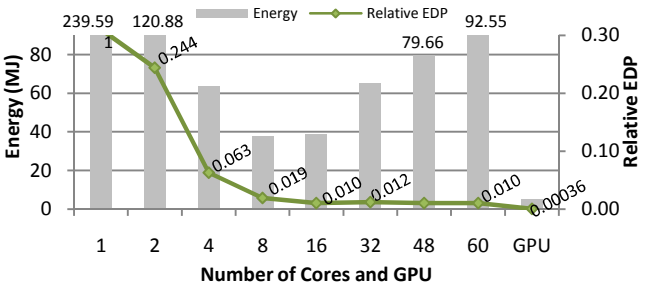

Figure 11: Energy evaluation of CAPPS, includes relative EDP.

CAPPS simulation in 4 hours and 8 minutes. Most importantly, as Figure 11 shows, our GPU implementation is18.4*x* more energy-efficient than the MPI on 60-cores. Our results show that GPU is the clear winner in terms of performance, energy-efficiency and hardware cost for an application like CAPPS.

D. A word on Energy Efficiency and Dynamic Voltage-Frequency Scaling (DVFS)

In this paper, we have not evaluated the impact of DVFS 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 as for `dynamic programming` in this study. 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.

VI. An Analysis of Programming Effeort

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

A. 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. However, it is 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] listing available instructions and short tutorials. Intel's optimization reference manual [27] provides a better discussion on SSE/AVX and general SIMD design concerns. However, it is very low level in nature. The Intel Intrinsics Guide [28] 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

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 single-threaded optimization effort. **: Multi-core effort only. NPI: No performance improvement; we do not calculate the PPEH since there was no improvement. ST_AV: 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.

Kernel / Application	Programming Effort														
	STOpt.		ST_AV	Base SIMD		Opt. SIMD		Multi-core		Best CPU		Base GPU		Opt. GPU	
	hr	PPEH		hr	PPEH	hr	PPEH	hr	PPEH	hr	PPEH	hr	PPEH	hr	PPEH
Mri-q	2	0.91	N/A	N/A	N/A	5	1.22	5+1	2.52	6	2.52	N/A	N/A	12	9.81
Stencil	1	7.02	*yes*	1 C	23.33	4	5.87	1+1	16.66	2	16.66	2	30.65	10	17.27
Histogram	N/A	N/A	N/A	2	NPI	N/A	N/A	3**	NPI	N/A	N/A	N/A	N/A	16	NPI
Gradient	5	0.47	*yes*	5 C	0.73	14	2.39	5+1	1.11	14	2.39	7	5.84	24	4.275
Dynamic programming	2*	0.855	*yes*	2 C	2.45	8	1.38	2**	NPI	8	1.38	6	0.27	N/A	N/A
Matrix Resizing	1	2.02	N/A	3	1.1	N/A	N/A	1**	NPI	3	1.1	3	2.29	N/A	N/A
Seam Carving	8	0.29	*yes:*	N/A	N/A	26	0.75	26+5	0.94	31	0.94	N/A	N/A	40	0.085
Laplacian	N/A	N/A	N/A	2	1.04	N/A	N/A	1**	NPI	2	1.04	5	3.14	N/A	N/A
DEsolver	5	0.36	N/A	N/A	N/A	N/A	N/A	5+5	3.61	10	3.61	3	20.46	N/A	N/A
CAPPS	5	0.36	N/A	N/A	N/A	N/A	N/A	5+10	2.52	15	2.52	N/A	N/A	13	4.46

C++ compiler, which supports the vector math library (VML), are not available for free.

Perhaps 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 Section 4.

B. Performance per Effort Hours (PPEH) 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 *PPEH* that quantifies the efficiency in effort. The PPEH metric provides a good insight into the performance gained for every hour spent on various implementations of the kernels and applications. PPEH 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, PPEH illustrates the efficiency of the effort and provides us with a way to make comparisons between different platform implementations. A higher PPEH does not imply an overall higher performance improvement; it tells us that we achieved a higher speedup per effort hour.

C. Evaluation of Programming Effort

Table 5 shows that multithreading the kernels with *pthreads* requires approximately one effort hour, plus the effort to optimize and 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 PPEH metric, which shows that a 47% speedup was achieved for every hour spent optimizing the `gradient`. `stencil` gains a 7*x* speedup for one effort hour for the ST implementation. The best efficiency is obtained with the GPU implementation of `DEsolver` and `stencil`, with a PPEH of 20.46 (6*x* better PPEH than best CPU's) and 30.65 (1.4*x* better PPEH than base SIMD), respectively. `stencil`'s PPEH 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 SC, the *energy-update* (EU) algorithm adds an extra 3 hours (not shown). We combined the SC EU for the resizing of a 1920x1080 video with the SIMD and GPU versions. The resulting effort for the SIMD_EU and GPU_EU is 29 and 43 hours, with a PPEH of 43% and 12%, respectively (Table not shown). Thus, using SIMD CPUs to resizing an HD video with SC is not only faster (12.49*x* over base) than the GPU (5.25*x* over base), but requires less effort and the performance gained for every effort hour is much higher. Better results are found in our CPU-GPU heterogeneous version, which requires 40 effort hours that result in 21.39*x* speedup – a PPEH of 53%.

VII. 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 [30, 31]. A number of studies discuss similar kernels and applications as in this paper. 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 our paper, the authors present a fair performance evaluation by utilizing all available hardware resources. However, in [1], energy-efficiency and programming effort have not been studied. A few recent papers [29, 30] evaluate energy-efficiency. Different than 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 paper is the first that attempts to quantify the programming effort for various platform-specific implementations.

In this paper, aside from kernels, we studied two full applications that utilize several of these kernels. We evaluate kernel-only and full-application performances separately. Several papers [14-16] explore GPU implementation of the SC application. In [15], a different algorithm is proposed to help parallelization but it also reduces the quality of the resized image/video. [14] and [16] focused on optimizing and parallelizing the original SC algorithm [3]. However,

they evaluate removal of one seam, which is only part of the resizing operation. They also have not evaluated kernels and the full-application separately. [15] and [16] compare their SC implementation against the single-threaded CPU baseline only. Also, none of the prior SC papers evaluate energy-efficiency.

CAPPS is implemented using MPI in [4], which explores the performance of a CPU-only cluster with 16 cores. In this paper, we utilize a 60-core cluster and a CPU-GPU heterogeneous system, and evaluate the performance and energy consumption of the systems.

VIII. CONCLUSION

In this paper, 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 hardware capabilities of both CPUs and GPUs.

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. Best performing platform for each of our kernels and applications vary. GPU is best for `CAPPS`, `mri-q`, `stencil`, `gradient` and `matrix resizing`. CPU is best for `histogram` and `dynamic programming`, heterogeneous CPU-GPU is best for `SC`.

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 our programming effort for various implementations of the studied 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 *PPEH*.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful suggestions. we would like to thank Jean-yves Hervé for his contributions. This work is partly supported by the National Science Foundation under Grant No. 1117467 and by the University of Rhode Island EGRA grant.

REFERENCES

[1] Intel, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," ISCA 2010.

[2] Intel, SSE4 Programming Reference, 2007

[3] S. Avidan and A. Shamir, "Seam Carving for Content-Aware Image Resizing," In SIGGRAPH, 2007.

[4] M. Amani, F.J. Vetter, "Unstable spiral waves in a 2D numerical model of myocardium," 35th Northeast Bioengineering Conference, pp. 66-67. Boston, MA, 2009.

[5] Sue et al. "Automatic thumbnail cropping and its effectiveness," In: Proc. of User Interface Software and Tech., 2003, pp. 95–104

[6] Chen et al. "A visual attention model for adapting images on small displays," Multimedia Syst, 2003, Vol. 9 No. 4, pp. 353–364

[7] G. Ciocca et al. "Self-adaptive image cropping for small displays," IEEE Trans Cons. Electr, 2007.

[8] http://www.photoshopsupport.com/photoshop-cs4/what-is-new-in-photoshop-cs4.html

[9] Available: http://liquidrescale.wikidot.com

[10] http://www.digikam.org/node/439

[11] http://www.imagemagick.org/ Usage/resize

[12] http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[13] R.C. Gonzalez, R.E. Woods. "Image Enhancement in the Spatial Domain," Digital Image Processing, Prentice-Hall, 2002, pp.125-127

[14] R. Duarte and R. Sendag, "Accelerating and Characterizing Seam Carving Using a Heterogeneous CPU-GPU System," PDPTA, 2012.

[15] Chiang et al. "Fast AND-Based Video Carving with GPU Acceleration for Real-Time Video Retargeting," IEEE Trans Circ. and Sys. for Video Tech., 2009, Vol. 19 , No. 11, pp. 1588–1597.

[16] R. Češnovar et al "Optimization of a single seam removal using a GPU" PDPTA 2011.

[17] Jang et al, "Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures", IEEE TPDS, Jan 2011.

[18] K. Thilagam, S. Karthikeyan, "Optimized Image Resizing using Piecewise Seam Carving," Int. J. of Comp Apps, 2012.

[19] R. Achanta and S. Süsstrunk, "Saliency detection for content-aware image resizing," *16th IEEE Int Conf Image Processing*, Nov. 2009.

[20] Xu et al " Image smoothing via L_0 gradient minimization," In *Proceedings of ACM SIGGRAPH Asia*, Dec. 2011, Vol. 30, No. 174.

[21] Sun et al " Image super-resolution using gradient profile prior," *IEEE Conf on Computer Vision and Pattern Recognition*, 2008, pp. 1-8.

[22] D. Scharstein, "Matching images by comparing their gradient fields," International Conference on Pattern Recognition, 1994.

[23] Stratton et al., "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," IMPACT Technical Report, IMPACT-12-01, U. of Illinois at Urbana-Champaign, 2012.

[24] MathWorks, http://www.mathworks.com/products/matlab/

[25] ILCS, "AVX-optimized," software-lisc.fbk.eu/avx_mathfun.

[26] The vdt mathematical library, https://svnweb.cern.ch/trac/vdt.

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

[28] http://software.intel.com/en-us/articles/ intel-intrinsics-guide.

[29] Song Huang, Shucai Xiao, Wu-chun Feng, "On the Energy Efficiency of Graphics Processing Units for Scientific Computing," IPDPS 2009.

[30] Cebrin et al. "Energy efficiency analysis of GPUs," IPDPS 2012.

[31] http://www.gpgpu.org