Run-time Image and Video Resizing Using CUDA-enabled GPUs

Ronald Duarte and Resit Sendag
Department of Electrical and Computer and Biomedical Engineering
University of Rhode Island, Kingston, RI
(rduarte, sendag)@ele.uri.edu

ABSTRACT

A recently proposed approach, called *seam carving*, has been widely used for content-aware resizing of images and videos with little to no perceptible distortion. Unfortunately, for high-resolution videos and large images it is not computationally feasible to do the resizing in real-time using small-scale CPU systems. In this paper, we exploit highly parallel computational capabilities of CUDA-enabled Graphics Processing Units (GPUs) in a heterogeneous computer system for accelerating the content-aware resizing of videos and images. The performance results show that our implementation of the seam carving algorithm achieves up to 235x and 30x speed-ups on the computationally-intensive part of the algorithm compared to the single-threaded and the multithreaded CPU implementations, respectively, on the systems tested. The overall resizing operation is up to 7x and 4x faster than the single-threaded and multithreaded CPU implementations, respectively, which demonstrates the potential to resize videos and large images in real-time.

1. INTRODUCTION

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, PDAs, desktop displays, etc), preferably without distortion.

Cropping [1-5] 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 direction.

Avidan and Shamir [6] recently provided 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 medium and automatically removes seams to reduce its size or inserts seams to extend it. This content-aware resizing method has been shown to effectively resize images and videos with little to no perceptible distortion. Seam carving has a two-part process: the energy function and the seam map, both of which involve computationally-intensive operations. For high-resolution images and videos, it may become impossible to perform this resizing in real-time by using the CPUs in a desktop-scale computer.

Parallelizing seam carving algorithm and running on servers with many cores may facilitate real-time resizing. However, when many images and videos may need to be resized, this may not be a viable solution. For example, some web servers may need a content-aware image resizing application to adjust the size of images embedded in web pages when the pages need to be displayed at different resolutions and/or aspect ratios. If the servers are providing personalized page content that is dynamically generated, then suitable resizing of images is even more important. Since the visitor traffic of web pages can be huge, a least expensive and more energy-efficient way of performing resizing is necessary.

The advent of commodity massively parallel architectures, such as modern GPUs, is a compelling option for inexpensively removing the computationally-intensive operations from the CPU. In this paper, we exploit the data-parallel execution model of GPUs for the implementation of content-aware image and video resizing. This paper makes the following contributions:

1) We evaluate and characterize GPU-based seam carving algorithm on two CUDA-enabled NVIDIA GPUs.
2) We compare single- and multi-threaded CPU versions of the algorithm with the GPU versions.
3) We demonstrate that, GPUs facilitate low-cost and energy-efficient real-time resizing of images and videos.

2. BACKGROUNDGraphics Processors

With convenient programming libraries, modern GPUs have evolved into programmable and highly parallel computational devices, containing hundreds of processing cores that can be used for general purpose computing. The architecture of a modern CUDA-enabled NVIDIA GPU consists of devices with several *streaming multiprocessors* (SMs) each containing multiple execution cores (streaming processors or SPs) operating on SIMD (Single Instruction Multiple Data) programs. With their high memory bandwidth (compared to favoring low latency as in CPUs), GPUs are ideal for parallel applications with high-levels of fine-grain data parallelism.

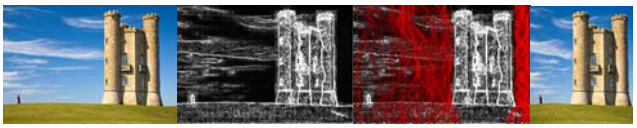
In the execution model of a GPU, a host program running on the CPU instructs the GPU to launch the kernel (a C/C++ function that executes on the GPU as many threads) after input data is transferred from the host memory to GPU memory by the DMA controller. The GPU, then, executes the threads in parallel. Finally, the DMA controller transfers the results back to host memory from device memory.

One of the major differences between the CPU and the GPU is its memory system [8]. 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. The CUDA memory hierarchy consists of six different user-managed memories and a hardware-managed L1 and L2 cache for the newer architecture. A fast and heavily-banked *shared* memory (local to the SMs) is managed explicitly by the programmer among thread blocks and is often used for caching frequently used data from *global* memory. *global* memory is similar to the main memory used by CPUs, but has a large bandwidth, in the order of several hundreds of GB/s. *Registers* are the faster memory on the GPU with access time of one cycle. Each SM contains 8K-32K 32-bit registers depending on the architecture, which are shared between every thread block executing on an SM. *local, constant* and *texture* memories are implemented with DRAM similar to the *global* memory. *local* memory is used for register spill and arrays that are local to the each thread. *constant* memory is cached for fast access. *texture* memory is optimized for 2D spatial locality.

2.2 Seam Carving

Seam carving [6] transforms the size of images and videos by carving-out pixels that form a path of low-energy. These low-energy connected paths, called *seams*, go from top to bottom or left to right depending on the resizing operation. For vertical resizing, the algorithm computes the seams that start on the left side of the

image and end on the right. In the same manner, horizontal resizing require the generation of seams to travel from top to bottom. The seams are added to or removed from an image[1] for increasing or reducing its size with minimal observable distortion. Figure 1 shows the steps of horizontally resizing an example image. Since the majority of execution time is spent on the energy function computations and the seam map generation, in this paper, we focus on accelerating these two computationally intensive parts.



(a) (b) (c) (d)

Figure 1[2]: Seam Carving Steps. (a) The original image. (b) The energy of the image using gradient magnitude method. (c) The low energy seams with the energy function. (d) Resized image after the seams are removed.

2.2.1 Energy function

Seam Carving can utilize a variety of energy functions to generate the seam map [6]. The gradient magnitude method uses equation (1) to compute the energy of each pixel relative to its surrounding pixels by quantifying the amount of change in color from one pixel to the next.

$$e(I) = \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right| \qquad (1)$$

The energy function computation exhibits vast data parallelism. However, attention must be given to memory access patterns which may have a huge implication on performance. The need for accessing neighboring pixels to compute the energy function strongly influences the way we access memory on the GPU.

2.2.2 Seam Map

After finding the energy of each pixel, lowest-energy paths, the *seams*, must be found. In order to generate vertical or horizontal seams, we compute their respective seam map. We focus on the implementation of the seams required for horizontal resizing, i.e. the vertical seams. The seam map is computed using the result of the energy function. The first row of the seam map is directly obtained from the first row of the energy function. Starting from the second row of the image, we compute the minimum value of the seam map for the three pixels directly above the current pixel of interest, and add its energy value to the result. This operation is performed for every pixel in the image. Finally, the result of the above operation becomes the seam map value for each respective pixel. The recurrence relation below illustrates the full operation:

$$S_{i,j} = \begin{cases} E_{0,j} & \text{if } i = 0 \\ E_{i,j} + \min\left(S_{i-1,j-1}, S_{i-1,j}, S_{i-1,j+1}\right) & \text{otherwise} \end{cases} \qquad (2)$$

In equation (2), S is the seam map table, E is the energy function table, and i and j are the row and column indices of the tables. This dynamic programming approach produces the optimal seam/s [6]. The seam map values in the final row of the image correspond to the cumulative energy of the W most optimal seams, where W is the width of the image. The most important point to note about equation (2) is that the computation for each element is entirely dependent on the result of three elements directly above it. Therefore, unlike the energy function, the seam map computation is not 100% separable. This makes parallelizing the seam map computation much more difficult than the energy function.

3. HARDWARE RESOURCES

All of the implementations presented in this paper were executed on two different heterogeneous computer systems. The First system is a Mac Pro running the Mac OS X 10.6 operating system powered by two 2.8GHz quad-core Intel Xeon E5462s CPUs, with 32KB of L1 and 12MB of L2 cache. The second system is a newer machine running the Ubuntu Linux 10.04 operating system, powered by a 3.4 GHz quad-core Intel Core i7 2600k CPU with 32KB, 256KB, and 8MB of L1, L2, and L3 cache respectively. The Intel Core i7 supports simultaneous multithreading (SMT) while the Intel Xeon does not support SMT [9, 10].

The Mac pro has an NVIDIA 8800GT GPU featuring the G80 architecture with 112 SPs or CUDA cores. This GPU has 512MB of GDDR3 memory with 256-bits wide memory bus. The GPU on the Linux machine is the NVIDIA GTX580 Featuring the Fermi (GF100) architecture. The GTX580 has 16 SMs, 32 CUDA cores per SM, for a total of 512 SPs. Features of the GTX580 memory are: 64KB configurable L1 cache/shared memory (16KB/48KB), 768KB L2 cache, and 1.5GB of GDDR5 memory with 384-bits wide bus. The device compute capability of the 8800GT and the GTX580 are 1.0 and 2.0 respectively.

The threading model used is POSIX threads (pthread). Since the Mac OS X does not support the pthread barrier API, we implemented a custom barrier method using condition variables. All CPU-only implementations were compiled with the *gcc* 4.2 on the Mac Pro and the *gcc* 4.4 on the Linux machine. Both compilers were configured with -O2 optimization level. Finally, the heterogeneous implementations were compiled with the NVIDIA *nvcc* compiler, which uses *gcc* of the respective systems to compile the CPU code.

4. IMPLEMENTATION

4.1 CPU Implementation

4.1.1 Energy Function

The single threaded CPU implementation of the energy function is straightforward; we divide the computation into two partial derivative computation, and combine the result later by summing the magnitude of the two results. The partial derivatives in both the *x* and the *y* direction require two nested loops for images that are stored in a 2D array of pixels and a single loop for images stored in a 1D array.

The energy function can easily be parallelized because pixel energies can be computed independent of each other. Therefore, we are able to divide the computation into as many threads as the operating system could support. Although the operating system might be able to support a large amount of threads, the performance is dictated by the hardware and the CPUs ability to execute threads simultaneously. In the multithreaded version, we divide the input image into tiles consisting of consecutive columns. The width of each tile is computed based on the number of threads and the width of the image. Figure 2 illustrates the decomposition of the input image.

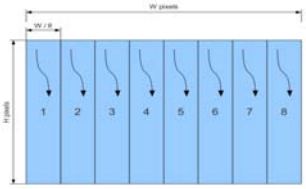


Figure 2: Division of work between threads in the CPU implementation.

[1] For most of the paper, we only discuss images. However, a video is a set of images or frames displayed at the video rate of 30 frames per seconds. Seam carving is equally applicable to both images and videos.

[2] Image taken from Wikimedia Commons.

4.1.2 Seam Map

Unlike the energy function, the seam map computation uses a dynamic programming approach that is not parallelization friendly (see section 2.2.2). Using this method, we are only able to parallelize each row of the image one at the time. We perform the row-by-row computation of the seam map by dividing each row into fixed-width tiles and computing these tiles in parallel. Following each row, each thread waits on a barrier for the rest of the row to complete. This method does not yield any significant benefit over the single-threaded version; in fact, with more than two threads, the program spends more time synchronizing the threads than actually performing the computations.

In an attempt to improve threaded performance, we used semaphores to create local barriers in place of global barriers across the rows. Instead of stalling threads until every thread finishes its part, each thread is only concerned with the execution of its neighboring threads. In addition, semaphores are more efficient than pthread barriers and the communication among conditional variables used on the Mac Pro. For this implementation, we use an array of semaphores to allow each thread to manage the availability of the seam map results of its first and last element on every row. The size of the array is obtained by doubling the product of the number of threads by the height of the image.

4.2 Energy Function on the GPU

Many computational steps that were described in Section 4.1 also apply to the GPU. Hence, we only discuss the most significant parts of the GPU implementations.

4.2.1 Naive Implementations

The first GPU implementation that we present in this paper is the *naive non-aligned* implementation. In this implementation of the energy function, the image was partitioned into 16x16 tiles containing 256 pixels, as illustrated in Figure 3. In addition to loading the corresponding data into shared memory, the kernel also needs to load the pixels immediately adjacent to the tile. These outer pixels are known as the tile apron, shown in yellow in Figure 4. Each tile utilizes one 2D block of 324 threads (18x18), thus assigning one thread per pixel load. It is important to note that the gradient value at the four corner pixels is set to zero. The rationale is that there is not enough information to compute the partial derivative with respect to *x* or *y*. From the previous statement, it is obvious that those tiles at the edge of the image require fewer loads.

The series of steps taken by each thread are as follows: First, a thread calculates the thread pixel index that it needs in order to load a pixel from the global memory. Second, the thread calculates the pixel index that it requires to write to shared memory. Third, the thread loads the relevant global pixel value (or zero) into shared memory. Fourth, the thread waits at a barrier for the remaining threads in the block to load their pixel. Finally, if the thread index is within the workable part of the tile, *i.e.* not an apron pixel, use the adjacent pixel values in shared memory to compute the gradient and store result.

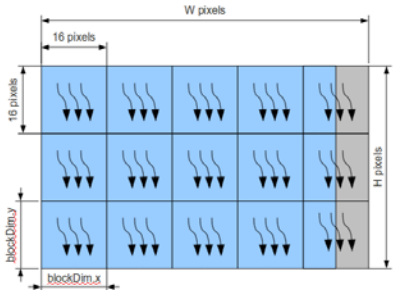


Figure 3: Division of work between threads in the naive GPU implementation. The gray area represents idle threads within the block.

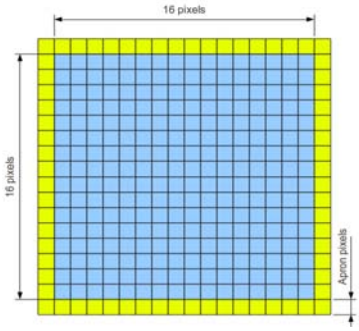


Figure 4: Position of apron pixels (yellow) and writable pixels (blue).

At first, we used a three-byte data structure to store the RGB components of each pixel since the alpha in the RGB color space is not utilized. The problem with a three-byte data structure is that memory access are not aligned, resulting in a reduction in performance. We solved this problem by aligning pixels to word length (4 bytes). By aligning pixels to a word, we waste 25% of the memory (one extra byte per pixel). However, if the memory size is sufficient, this is an excellent tradeoff. Moreover, there are times when we are interested in maintaining the alpha component of the pixel, this implementation guarantees that the pixel remains in its original form. In addition, we allocate memory on device and copy the image data to the GPU using the **cudaMallocPitch** and **cudaMemcpy2D** [7]. Using the two-dimensional allocation and copy functions, we guarantee that each row of the image starts on a 64 and 128-bytes boundary in global memory for devices with compute capability1.x and 2.x, respectively.

4.2.2 Split-aligned Implementation

To achieve nearly full coalesced memory access, we decided to separate the energy function calculation into two separate kernels, a horizontal and a vertical gradient kernel, and combine the results of the two. This allows us to rearrange the way we organize the thread grid to suit the kind of memory accesses expected for each direction of the derivatives. The purpose of the split method is to ensure that when the pixel data is loaded from global memory, it incurs minimal uncoalesced accesses. Note that both the apron pixels and workable pixels in the vertical direction are always aligned to 16 pixels as shown in Figure 5, allowing coalesced accesses of 64 bytes at a time.

For the horizontal calculation, it is not possible with this approach to avoid uncoalesced memory accesses because the apron pixels lie outside the alignment boundary. However, this implementation attempt to hide the wasted bandwidth of the uncoalesced memory accesses per row by increasing the tile width to 128 pixels, as shown in Figure 6. This improves the memory efficiency allowing only two uncoalesced loads per every eight coalesced loads, and thus increasing bandwidth usage.

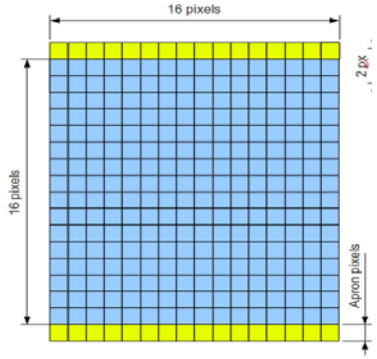


Figure 5: Position of apron pixels (yellow) and writable pixels (blue) in shared memory blocks for the vertical derivative

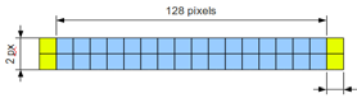


Figure 6: Position of apron pixels (yellow) and writable pixels (blue) in shared memory blocks for the horizontal derivative

4.2.3 Locality-aware-split Implementation

The *split-aligned* method has the potential to reduce the number of uncoalesced memory accesses by a significant amount, but it does not make the best use of locality. Therefore, when migrating the implementation to a newer heterogeneous system, we decided to revise the *split-aligned* method in order to take advantage of locality and the new capabilities provided by the Fermi architecture. The GTX580 offers approximately 4.5X more SPs than the 8800GT. It also supports memory access of up to 128-bytes on a single coalesce load.

Apart from the two outermost pixels that surround the entire image, all other pixels are utilized four times in the energy function computation; twice for each of the partial derivatives. Even when these pixels are cached in shared memory, which saves one global load per derivative, the *split-aligned* method requires that each pixel be loaded twice. Therefore, we decided to go back to implementing the energy computation using a single kernel to compute both partial derivatives.

The *locality-aware-split* method breaks the image into 2D blocks of 256 threads. Each tile contains 64 columns and 8 rows. This approach is similar to the *naive-aligned* method, the major difference is that, here, we choose the block size to be a multiple of the warp's size which benefits the SIMT programming model. With a 64x8-block configuration, two warps are assigned per rows. All 32 threads in a warp are able to cache their corresponding pixel on a single coalesce load of 128-bytes for 16 fully coalesce loads. The top and bottom apron are also loaded using a fully coalesce access per warp. The uncoalesce loads are introduced by the left and right apron of the tile. By increasing the number of rows, we improve the locality while adding more uncoalesce loads. In addition, by increasing the block width by a multiple of the warp size, we minimize the number of uncoalesce loads. This method exhibits properties from both the *naive-aligned* and *split-aligned* methods. Therefore, it is safe to say that the *locality-aware-split* method merges the important features of the two; more coalesce loads and better use of locality. After careful analysis and performance tests, we found that eight rows produce the best performance.

4.3 Seam map on the GPU

The GPU implementation of the seam map computation is very similar to the multithreaded CPU implementation, which is described in Section 4.1.2. Each row of the image is broken into horizontal tiles, whose width is carefully selected in order to maximize the occupancy of the GPU. Given that blocks are not scheduled deterministically and that there is no synchronization among threads on different blocks, we must resort to calling the kernel once per row and synchronize in between calls. For this implementation, wider images should perform much better than narrow images. Similar to the multithreaded CPU implementation, a significant amount of the data is not separable. This limits the amount of parallel execution per kernel launch.

4.4 Page-Locked Memory Optimization

Since a video is a series of images, all of the above methods are applicable to videos. The only restriction is that all frames are processed in 33 milliseconds or less to accomplish the full video rate of 30 frames per second. We introduce a new approach that exposes the true heterogeneity of these systems. The CUDA runtime environment provides functionalities to allocate and use **page-locked** memory in place of regular pageable host memory [7]. This method only differs in the way memory is used; any of the kernels that are described above, may be used to compute the energy function or the seam map.

5. PERFORMANCE EVALUATION

The overall time that it takes to remove a single seam of an image depends highly on the size of the image. For an image of size 1200x900, it takes approximately 85.6 milliseconds on the Mac Pro with the Intel Xeon CPUs. The energy function takes approximately 60% of the total execution time to remove one seam. The seam map computation takes the second largest fraction of time, approximately 24%. This implies that finding the removable seam and resizing the image takes 16% of the total execution time. This 16% also includes any basic initial steps or computations that are neither part of the initial program setup nor part of energy function and the seam map computations. Therefore, in our performance evaluation, we focus on improving the energy function and the seam map computations. However, we also compare and discuss the total execution times.

5.1 CPU Evaluation and Results

5.1.1 Energy Function

Figure 7 illustrates the performance gained by multithreading the energy function computations and executing the implementation on the Intel Core i7 (4-cores each with SMT) and Xeon CPUs (8-cores, no SMT), respectively. The execution of the energy function single threaded implementation takes 51.2 ms to complete on the Intel Xeon CPU. The base system, in Figure 7, is the Intel Xeon single-threaded execution time. Results show that the newer Intel Core i7 CPU outperforms the Intel Xeon processor for two or fewer threads. With eight threads, however, the Intel Xeon exhibits better scalability and produces the best CPU performance for the energy function computation. Overall, energy function computation scales well on multi-core CPUs. With eight cores, more than 7x performance improvement is possible. As the number of threads launched increase beyond the number of hardware threads in the system the performance gain gets smaller due to the thread switching overheads.

5.1.2 Seam Map

In section 4.1.2, we discussed the implementation of the seam map on the CPU and the dependability among rows of pixels. We emphasized how dependability due to the dynamic programming approach serialized the execution of rows. However, the results expose another problem that significantly affects the parallelization of the seam map computation. Figure 8 depicts the performance results of the seam map. The Figure illustrates that barriers impose a considerable overhead and the best we can achieve is 8% performance improvement. Beyond two threads, the performance is worse than that of the single-threaded implementation.

As mentioned in section 4.1.2, a more efficient approach is to synchronize locally instead of at the global level. This implementation performs better because semaphores inflict less overhead. We can see from Figure 8 that Intel core i7 shows more than 3x performance improvement over the single-threaded implementation for 8 threads. The implementation scales well for the Intel core i7; it is not until we reach the maximum number of hardware threads that the performance diminishes. Intel core i7 outperforms the Intel Xeon system significantly. The performance improvement on the Intel Xeon system reaches to 1.5x times with 4 threads. However, with 8-threads, we see a large drop in performance even if the system has 8 hardware threads. This behavior needs further research and left as future work.

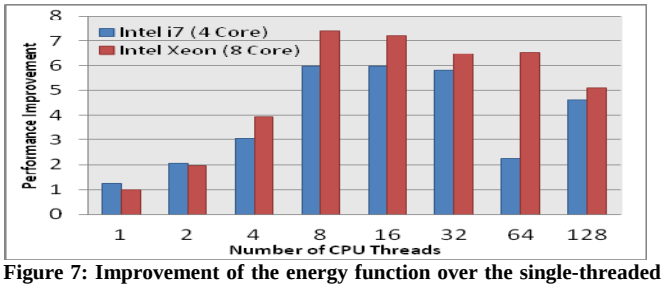


Figure 7: Improvement of the energy function over the single-threaded executing of the the Intel Xeon

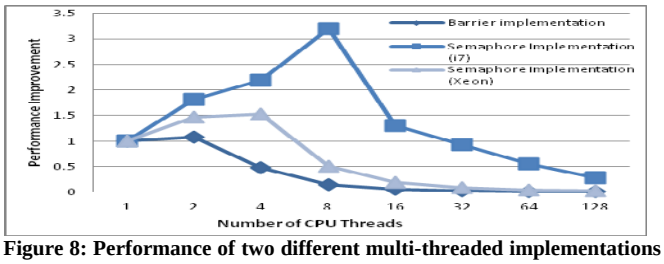


Figure 8: Performance of two different multi-threaded implementations of the Seam Map.

5.2 GPU Performance Evaluation

5.2.1 Naive-non-aligned Energy Function

On the 8800GT, the *naive-non-aligned* GPU implementation of the energy function yields a 13x and an 11x performance improvement over the single-threaded CPU implementation executing on the Intel Xeon and core i7, respectively, as shown in Figure 9. This significant improvement is hard to achieve with either of the CPUs and their respective amount of cores. Nevertheless, the *naive-non-aligned* implementation does not take advantage of the GPU's wide memory bus. Its memory access patterns are not coalesced because the loads are not aligned. Since the alpha in the RGB color space is not used in the energy function computation, the naive implementation only stores three bytes for the RGB. As a result, a warp will need to load 96 bytes while a half of a warp will need to load 48 bytes. Another Problem with this implementation is the block size, which is not a multiple of the warp size. The *naive-non-aligned* method was initially designed with the G80 architecture in mind. However, with a minimum modification, this implementation yields 146x and 116x performance improvement on the Fermi GTX580, over the single-threaded implementation running on Intel Xeon and Intel Core i7, respectively (see Figure 10).

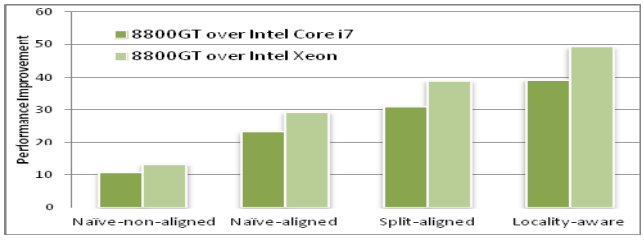


Figure 9: Improvement of the energy function over the single threaded CPUon the 8800GT

5.2.2 Naive-aligned Energy Function

The changes to transform the naive method from a non-aligned to an aligned implementation (see Section 4.2.1) improve the performance relative to the single-threaded version from 13x to 29x and 11x to 23x on their respective systems as shown in Figure 9. Utilizing the CUDA profiler, we were able to determine the remaining source of our performance problems, uncoalesced accesses. The first naive version incurred over 500,000 uncoalesced loads and 300,000 uncoalesced stores for a 1200x900 image (≈ 1 megapixel); the improved alignment version incurred only 100,000 uncoalesced loads and 50,000 uncoalesced stores. This is still significantly more than one would expect, as an image with this amount of pixels should only need 16,875 loads assuming the GPU can bring in 64 bytes per coalesced loads. The *naive-aligned* method was also designed for the 8800GT. When executed on the GTX580, This implementation shows a performance improvement of 163x and 130x over the respective Intel Xeon and Core i7 single-thread CPU implementation (see Figure 10).

5.2.3 Split-aligned Energy Function

The *split-aligned* method described in Section 4.2.2 achieves an average of 850 megapixels per second throughput, a 39x and a 31x improvement over the single-threaded CPU version on the Intel Xeon and Core i7, respectively, as shown in Figure 9. As expected, the CUDA profiler reveals that for a 1200x900 image, only approximately 31,000 loads and 15,000 stores were needed (each pixel must be loaded from global memory twice, once for each directional kernel), reducing the total memory access latency by an order of magnitude. On the GTX580, *split-aligned* achieves 176x improvement over the Intel Xeon CPU and 140x over the Intel Core i7 as shown in Figure 10.

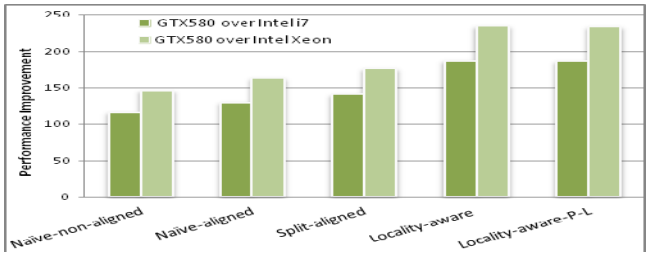


Figure 10: Improvement of the energy function on the GTX580 (Fermi) over the single threaded CPU

5.2.4 Results of Merging the Split-aligned

The l*ocality-aware-split* method described in section 4.2.3 achieves the highest performance improvement on both GPUs for the energy function computation. By merging the two derivative computation in a way that the number of coalesce loads remains the same, and by further taking advantage of locality of accesses, we manage to improve the performance of the energy function by 235x and 187x over the single-threaded CPU version on the Intel Xeon and core i7, respectively, as shown in Figure 10. In addition, when executed on the 8800GT, this method shows a performance improvement of 49x and 39x over the Xeon and Core i7 single - threaded version (see Figure 9).

5.2.5 Seam Map

The seam map GPU implementation exhibits an 8x and a 4.8x performance improvement over the single-threaded CPU implementation on the Intel Xeon and Core i7, respectively (figure not shown). This performance gain is relatively small in comparison to the energy function speedup. The performance is heavily impacted by the profound dependability among rows in the image. This limits the amount of parallel computation by serializing the execution of rows. Another significant performance impact is that there is no optimal method for synchronizing threads among different blocks. Launching the kernel the height of the image minus one times (e.g. 899 for an image of height 900), imposes a significant overhead. Approximately 57% of the seam map execution time is due to kernel launch overhead. Minimizing the kernel launch overhead could potentially improve the seam map performance by a factor of two.

5.3 An Evaluation of Total Execution Time of Resizing Operation on the GTX580

As previously stated, the energy function and seam map computation account for 84% of the execution time that it takes to remove one seam. Therefore, by improving these two parts, one would normally achieve a high overall performance improvement. However, there is a penalty when performing computation on the GPU device. The data must be copied from the host memory to the device memory. Once the computation is performed, we must copy the results back to the host memory if we care to use the results on the CPU side. Both of these operations introduce additional overhead. For extensive GPU computation, the overhead is easily hidden. However, this is not the case for seam carving given that the computations are in the order of micro and milliseconds.

In order to use this GPU implementation of the seam carving in a real word application, we need to utilize the operation described above. Therefore, we need incorporate the total time that it takes to copy the image from the host to the device, compute both the energy function and the seam map, and copy the result back to the host memory. Figure 11 illustrates the total time that the entire operation takes on the Intel Core i7 and on the GTX580, respectively. This heterogeneous system is selected because it performs the best for both the CPU and the GPU. Figure 12 shows the performance improvement for the entire operation.

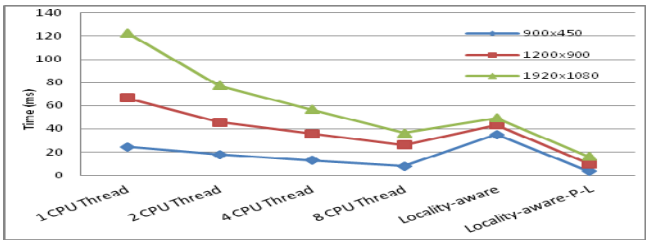


Figure11: Total time to copy to and from the device, compute the energy function and seam map on the GTX580 and Core i7

In the results illustrated by Figure 12, we introduce a forth image of size 2740x1830 pixels. The point is to show how much better than the CPU, the GPUs implementation perform when the size of the image increases. Overall, Figure 12 shows that the total execution time of the best resizing implementation on the GTX580 is about 7x faster than the single-threaded CPU implementation and about 2.5x faster than the multithreaded CPU implementation.

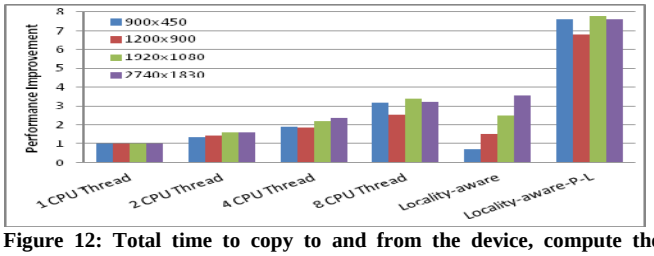


Figure 12: Total time to copy to and from the device, compute the energy function and seam map on the GTX580 and Core i7

Page-Locked Memory Results: The best implementation that performs the entire operation is the *locality-aware-split-page-lock* method. The reason is that the CUDA run-time environment can optimize the memory host to device and device to host memory copy if the CPU memory is allocated as *non-pageable* memory (see [7]). We therefore modified our fastest implementation, *locality-aware-split*, to take advantage of page-locked memory. The resulting implementation yields the best overall performance as shown in Figures 11 and 12. Figure 10 shows that the *locality-aware-split-page-lock* method achieves 186x performance improvement over the Intel Core i7 single-threaded implementation.

6. RELATED WORK

Resizing images and videos have been studied extensively in the literature. One of the most popular approaches is to perform cropping [1-5], which involves finding the best rectangular sub-window in the image. However, cropping may lose an unacceptable amount of visual information when important structures lie at all edges of an image. Scaling methods, with or without interpolation, tend to produce distorted images, especially when an image is scaled in one direction.

Avidan and Shamir [6] recently provided a new approach to image and video resizing, called *seam carving*. Seam carving is an algorithm for content-aware resizing of images and videos with little to no perceptible distortion. Seam carving is a computationally-intensive method that could make it difficult to perform on large images or videos at run-time.

To the best of our knowledge, this paper is the first to implement run-time content-aware resizing method on the GPUs. Our implementation works very well on computing energy function (over 230x is possible), but the other computationally-intensive part, *seam map*, is implemented using dynamic programming which limits the amount of data parallelism that can be exploited (only 4.8x). A recent work [11] implemented a faster way to compute the seam map by finding the optimal matches within a weighted bipartite graph composed of the pixels in adjacent rows or columns. In future work, we will adapt this method, which we believe, will improve our results greatly for the seam map computation.

7. CONCLUSION

Seam carving is a powerful method for resizing of images and videos. This content-aware resizing method has been shown to effectively resize images and videos with little to no perceptible distortion. However, seam carving algorithm is computationally-intensive and for high-resolution images and videos, it may become impossible to perform this resizing in real-time by using the CPUs in a desktop scale computer.

In this paper, we exploit highly parallel computational capabilities of CUDA-capable GPUs in a heterogeneous computer system for accelerating the resizing of videos and images through seam carving. Out of the four different GPU methods that we implemented, our results show that the best is the *locality-aware-split-page-lock* method, which achieved a performance improvement of 186x over the best single-threaded execution time and 31.5x over the best CPU multithreaded version for the energy function on the Intel core i7. Overall, our results show that the GPU-based implementation has a significant impact on the performance of seam carving and has the potential to resize videos and large images in real-time.

8. REFERENCES

[1] Itti L, Koch C, Niebur E. A model of saliency-based visual attention for rapid scene analysis. IEEE Trans Patt Anal Mach Intell, 1998, 20(11): 1254–1259

[2] Sue B, Ling H, Bederson B, et al. Automatic thumbnail cropping and its effectiveness. In: Proc. of User Interface Software and Tech., 2003. 95–104

[3] Chen L, Xie X, Fan X, et al. A visual attention model for adapting images on small displays. Multimedia Syst, 2003, 9(4): 353–364

[4] Ciocca G, Cusano C, Gasparini F, et al. Self-adaptive image cropping for small displays. IEEE Trans Consumer Electr, 2007, 53(4): 1622–1627

[5] Santella A, Agrawala M, DeCarlo D, et al. Gaze-based interaction for semi-automatic photo cropping. Proc. of Human Factors in Comp. Sys, 2006. 771–780

[6] S. Avidan and A. Shamir, Seam Carving for Content-Aware Image Resizing, In *SIGGRAPH '07 ACM SIGGRAPH*, 2007.

[7] NVIDIA, NVIDIA CUDA C Programming Best Practices Guide 4.0, May 2011,

[8] B. Jang, D. Schaa, P. Mistry, D. Kaeli, Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures, *IEEE TPDS*, 2011.

[9] Intel, 2nd Generation Intel® Core™ Processor Family Desktop, Oct 2011.

[10] Intel, Intel Xeon Processor Series Datasheet.

[11] Real-time content-aware image resizing, Science in Chine Series F: Information Sciences, 2009 Sci. in China Press.