Accelerating and Characterizing Seam Carving Using a Heterogeneous CPU-GPU System

Ronald Duarte and Resit Sendag
Department of Electrical, Computer, and Biomedical Engineering,
University of Rhode Island, Kingston, RI, USA

Abstract - *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 becomes computationally unfeasible to do the resizing in real-time using small-scale CPU systems. In this paper, we exploit the highly parallel computational capabilities of CUDA-enabled Graphics Processing Units (GPUs) 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 100x and 14x speed-ups on the computationally-intensive part of the algorithm compared to the faster single-threaded and the faster multithreaded CPU implementations, respectively, on the systems tested. The overall resizing operation is over 6x and 2x faster than the best single-threaded and multithreaded CPU implementations, respectively, which demonstrates the potential to resize videos and large images in real-time.*

Keywords: Seam carving, GPU, CUDA, parallelization, heterogeneous system

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 media and automatically removes or inserts seams to resize the media. This content-aware resizing method has been shown to effectively resize images and videos with little to no perceptible distortion. Seam carving is a computationally-intensive operation. For high-resolution images and videos, it could be difficult to perform the resizing in real-time by using the CPUs in a desktop-scale computer.

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 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 real-time resizing of images and videos.

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 for vertical or horizontal resizing, respectively. The seams are added to or removed from an image[1] in order to increase or reduce 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 and seam map computations (see section 5), in this paper, we focus on accelerating these two computationally intensive parts.

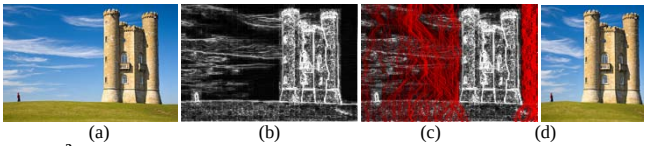


Figure 1[2]: Seam Carving Steps. (a) The original image. (b) The energy of the image using gradient magnitude. (c) The low energy seams with the energy function. (d) Resized image after the seams are removed.

2.1 Energy function

Seam Carving can utilize a variety of energy functions to generate seams [6]. The magnitude of the gradient approach uses equation (1) to compute the energy of each pixel relative

[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.

to its surrounding pixels by quantifying the amount of change in color from one pixel to the next.

$$e(I) = \left\|\overrightarrow{\nabla I}\right\|_1 = \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right| \quad (1)$$

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 Seam map

After finding the energy of each pixel, the result is used to locate the lowest-energy paths or *seams*. We focus on the implementation of the seams required for horizontal resizing, i.e. the vertical seams. 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 use a dynamic programming approach (2) to compute the seam map value at every pixel.

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

In equation (2), S is the seam map table, E is the energy function table, and i and j are the rows and columns indices of the tables. This dynamic programming approach produces the optimal seam/s [6]. The values in the final row of the seam map table correspond to the cumulative energy of the seams. The most important point to note about equation (2) is that the computation for each element is entirely dependent on the result of the three elements directly above it as shown in Figure 2. Therefore, unlike the energy function, the data in the seam map computation is not 100% separable. This makes parallelizing the seam 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. The Mac pro has an NVIDIA 8800GT GPU (G80 architecture) with 112 cores and 512MB of GDDR3 memory.

The second system is a newer machine running the Ubuntu Linux 10.04 operating system, powered by a single 3.4 GHz quad-core Intel Core i7 2600k CPU. The GPU on the Linux machine is the NVIDIA GTX580 Featuring the Fermi architecture with 16 SMs (32 cores per SM) for a total of 512 SPs. The GTX580 has 1.5GB of GDDR5 memory, 768KB L2 cache, and 64KB configurable L1 cache/shared memory per SM. The Intel Core i7 supports simultaneous multithreading (SMT) while the Intel Xeon does not support SMT [8, 9].

The parallelization tool for the CPU implementation is POSIX threads (pthread). Both of the CPU implementations were optimized and compiled with *gcc* -O2 optimization level. Finally, the heterogeneous implementations were compiled with the NVIDIA *nvcc* compiler, which uses *gcc* and the -O2 flag 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 utilizes a set of nested *for-loops* to compute each of the partial derivatives (Algorithm 1, lines 1-7), and utilizes the result to compute the magnitude of the gradient (Algorithm 1, lines 8-9). The one-half is factor out of the derivatives and applied after the sum in order to save one multiplication operation per pixel. Instead of directly storing a 2D image in a 2D array, we store the 2D image in a 1D array and map the 1D to a 2D array. This minimizes the allocation time, and the data is stored in a more suitable manner to take advantage of spatial locality. Note that the derivative of each pixel is computed once, the purpose of the nester loops is to ease the computation, but each derivative has O(n) time complexity.

Given that there are no data dependencies in the computation of the energy function, we are able to divide the computation into as many threads as the operating system supports. However, the performance is dictated by the hardware and the CPUs' ability to execute threads simultaneously. We divide the input image into tiles consisting of consecutive rows. The height of each tile is computed based on the number of threads and the height of the image. Figure 3 illustrates the decomposition of the input image for an execution configuration of eight threads. The Algorithm for the multithreaded version is very similar to Algorithm1 with the exception that each thread only loops through its corresponding portion of the image.

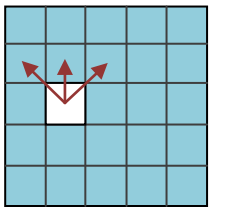


Figure 2. Seam map example.

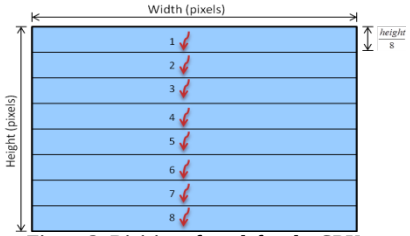


Figure 3: Division of work for the CPU.

Alg. 1 Single threaded Energy function

```
1: Set all elements of Ix and Iy to 0
2:     for i ← 0 to height – 1
3:         for j ← 1 to width – 2
4:             Ix(i, j) ← I(i, j+1) – I(i, j-1)
5:     for i ← 1 to height – 2
6:         for j ← 0 to width – 1
7:             Iy(i, j) ← I(i+1, j) – I(i-1, j)
8:     for all pixel in the image
9:         energy = 0.5 * (|Ix| + |Iy|)
```

Alg. 2 multithreaded Seam map

```
1: for i ← 1 to height – 1
2:     for j ← t_start to t_end
3:         if (j-1 < t_start)
4:             wait for S(i, j-1) to be unlock
5:         if (j+1 > t_end)
6:             wait for S(i, j+1) to be unlock
7:         S(i, j) ← Min(S(i-1, j-1),
               S(i-1, j), S(i-1, j+1))+energy(i, j)
8:         Unlock S(i, j)  //initially locked
```

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). Therefore, we have to perform a row-by-row computation of the seam map, which serializes the execution of rows. We achieve parallelism by dividing each row into fixed-width tiles and computing these tiles in parallel. We synchronize all threads after the execution of each row. 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 than performing the computations.

In an attempt to optimize the seam map implementation, we used locks to create local barriers in place of global barriers. Instead of stalling threads until every thread finishes its part, each thread is only concerned with the execution of its neighboring threads. We therefore use an array of locks to allow each thread to manage the availability of the seam map results of its boundary element on every row. This approach is illustrated in Algorithm 2 where *t_start and t_end* are computed by dividing the width of the image by the number of threads and assigning each thread their respective areas.

4.2 Energy function on the GPU

4.2.1 Naive implementation

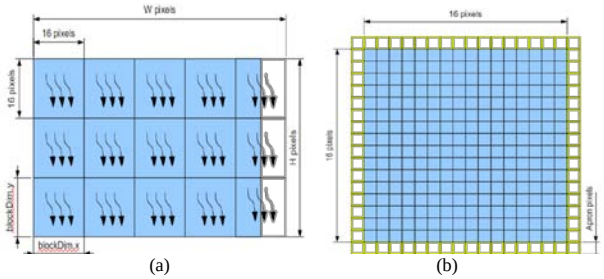
The first GPU method presented in this paper is the *naive-non-aligned* implementation. In this implementation, the image was partitioned into 16x16 tiles containing 256 pixels as illustrated in Figure 4a. In addition to loading the corresponding data into shared memory, the kernel also needs to load the pixels immediately adjacent to the tile. These outer pixels are known as the tile apron, shown in white in Figure 4b. Each tile utilizes one 2D block of 324 threads (18x18), thus assigning one thread per pixel load.

All CUDA threads in the Naive kernel execute Algorithm 3. First, each thread calculates the necessary pixel indices to copy a pixel from the global to the shared memory (lines 1-4). Line 6 caches the pixels in shared memory and line 8 ensure that all data transfer completes before performing the computation. Finally, we use 256 out of 324 threads to compute the gradient and store result (lines 10-12).

At first, we used a three-byte data structure to store the RGB components of each pixel. A three-byte data structure causes unaligned memory accesses, which reduces performance. We solved this problem by aligning pixels to word length (4 bytes). This implementation waste 25% of the total memory. However, if the memory size is sufficient, this is an excellent tradeoff. In addition, there are times when we are interested in preserving the alpha component of all pixels; this implementation guarantees that all pixels retain the original alpha value. Another optimization technique is to allocate memory on the device using the *cudaMallocPitch* function [7]. Using the two-dimensional allocation and copy functions (*cudaMemcpy2D*), we guarantee that each row of the image starts on a 64 or 128-bytes boundary in global memory depending on the device architecture.

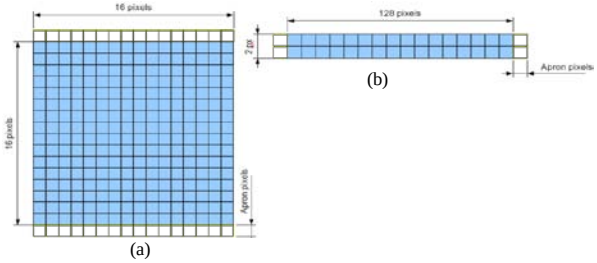
Alg. 3 Naive implementation of the energy function in CUDA

```
1:   col ← block_x * tile_width + x – 1     // x : thread_x
2:   row ← block_y * tile_height + y –1     // y : thread_y
3:   k ← row * image_width + col
4:   i ← y * bw + x                         // bw : block_width
5:   Ix ← 0,  Iy ← 0
6:   if (k is an index within the image)  SMEM(i) ← image(k)
7:   else  SMEM(i) ← 0
8:   Synchronize_threads
9:   if (x > 0 and x ≤ tile_width and y > 0 and y ≤ tile_height)
10:      Ix ← SMEM(i+1) – SMEM(i–1)
11:      Iy ← SMEM(i+bw) – SMEM(i–bw)
12:      ENERGY(k) ← 0.5 * (|Ix| + |Iy|)
```



(a) (b)

Figure 4: (a) Division of work between threads in the naive GPU implementations; the white area represents idle threads. (b) Partition of block; the tile apron is shown in white and the writable pixels in blue.



(a) (b)

Figure 5: Partition of the image. (a) Vertical tiles. (b) Horizontal tiles. White represents the apron pixels and blue/dark the workable pixels

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 reorganize the thread grid to suit the kind of memory accesses expected for each direction of the derivatives. Both the apron pixels and workable pixels in the vertical direction are always aligned to 16 pixels as shown in Figure 5a, allowing coalesced accesses of 64 bytes.

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, by increasing the tile width to 128 pixels (Figure 5b), some of the wasted bandwidth due to uncoalesced memory accesses is hidden. This improves the memory efficiency allowing only two uncoalesced loads per every eight coalesced loads, and thus increasing bandwidth usage. Algorithm 4 and 5 show the implementation of the *split-aligned* method to compute the energy function.

Alg. 4 Vertical implementation of the gradient in CUDA

```
1:  col ← block_x * bw + x   // x/y : thread_x/y,  bw : block_width
2:  row ← block_y * tile_height + y –1
3:  k ← row * image_pitch + col
4:  i ← y * bw + x,  Ix ← 0,  Iy ← 0
5:  if (k is an index within the image )  SMEM(i) ← image(k)
6:  else  SMEM(i) ← 0
7:  Synchronize_threads
8:  if ( y < tile_height )
9:      row ← row + 1,  i ← i + bw
10:     k ← row * energy_pitch + col
11:     if (row < image_height)
12:         Iy ← SMEM(i + bw) – SMEM(i – bw)
13:         ENERGY(k) ← |Iy|
```

Alg. 5 Split-aligned implementation of energy in CUDA

```
1:  col ← block_x * block_width + thread_x
2:  row ← block_y * block_height + thread_y
3:  k ← row * image_pitch + col
4:  x ← thread_x + 1,  y ← thread_y,  Ix ← 0
5:  i ← y * HORIZ_WIDTH + x
6:  if (k is an index within the image)  SMEM(i) ← image(k)
7:  if (thread_x == 0 and row < image_ height)
8:      pbase ← col * image_pitch
9:      SMEM(y * HORIZ_WIDTH) ← image(pbase + block_width * block_x – 1 )
10:     SMEM(y * 2 * HORIZ_WIDTH – 1) ← image(pbase + (block_width+1) * block_x)
11: Synchronize_threads
12: if (col < image_width)
13:     k ← row * energy_pitch + col
14:     Ix ← SMEM(y * HORIZ_WIDTH + x + 1) – SMEM(y * HORIZ_WIDTH + x – 1)
15:     ENERGY(k)← 0.5 * (|Ix| + ENERGY(k))
```

4.2.3 Locality-aware 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 porting the implementation to the 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 accesses of up to 128-bytes on a single coalesce load.

Apart from the two outermost pixels that surround the entire image, all pixels are utilized four times in the energy function computation; twice for each of 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* method breaks the image into 2D blocks of 512 threads. Each tile contains 64 columns and 8 rows. With a 64x8-block configuration, two warps are assigned per row. 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. Loading the top and bottom aprons also results in fully coalesce loads. The uncoalesce loads are introduced by the left and right aprons of the tile. Increasing the number of rows in a tile improves locality, but increases the number of uncoalesce loads. Increasing the block width minimizes the number of uncoalesce loads, but reduces locality. After careful analysis and performance tests, we found that 8 rows and 64 columns generate the best performance. Algorithm 6 gives an in-depth description of the *locality-aware* implementation of the energy function.

Alg. 6 Locality-aware implementation of the energy in CUDA

```
1:  col ← block_x * block_width + thread_x      // We ensure col
2:  row ← block_y * block_height + thread_y     // and row are within
3:  k ← row * image_pitch + col                 // the image width
4:  j ← thread_x + 1,  i ← thread_y + 1         // and height
5:  SMEM(i, j) ← image(k)
6:  if (thread_x == 0 and col ≠ 0)  SMEM(i, 0) ← image(k – 1)
7:  if (thread_x == block_width–1 and col ≠ image_width–1)
8:      SMEM(i, block_width + 1) ← image(k +1)
9:  if thread_y == 0 and row ≠ 0
10:     SMEM(0, j) ← image(k – image_pitch)
11: if thread_y == block_height–1 and row ≠ image_height–1
12:     SMEM(block_height + 1, j) ← image(k + image_pitch)
13: Synchronize_threads
14: Ix ← 0,  Iy ← 0,  k ← row * energy_pitch + col
15: if pixel are not on the edge of the image
16:     Ix ← SMEM(i, j + 1) – SMEM(i, j –1)
17:     Iy ← SMEM(i + 1, j) – SMEM(i – 1, j)
18: else if pixel is in the first or last rows
19:     Ix ← SMEM(i, j + 1) – SMEM(i, j – 1)
20: else if pixel is on the first or last columns
21:     Iy ← SMEM(i + 1, j) – SMEM(i – 1, j)
22: ENERGY(k)← 0.5 * (|Ix| + |Iy|)
```

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

The CUDA runtime environment has functionalities to allocate and use *page-locked* memory in place of regular *pageable* host memory [7]. We included this feature in our heterogeneous implementation to optimize the memory transfer. In the performance evaluation, we demonstrate that *page-locked* memory does not affect the performance or results of individual kernels, but it improves the execution time of the memory transfer between the host and device.

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. The energy function takes the largest fraction of the total execution time, followed by the seam computation. Hence, in this paper, 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 6 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). The execution of the energy function single-threaded implementation takes 31.5 ms to complete on the Intel Xeon CPU. This is the base system in Figure 6. Results show that the newer Intel Core i7 CPU outperforms the Intel Xeon processor for all of the thread configurations. The best CPU performance for the energy function computation of a 1200x900 image is with the Intel Core i7 and 16 threads. Overall, the energy function computation scales well on multi-core CPUs. With eight cores, the Intel Xeon achieves 7x performance improvement. With four cores and eight hardware threads, the Intel Core i7 achieves 7x speedup over its single-threaded execution. In addition, the Intel Core i7 achieves a 10x performance improvement over the Intel Xeon single-threaded execution. Finally, as the number of threads launched increase beyond the number of hardware threads in the system, the performance gain becomes smaller due to the thread switching overheads. The only exception is the 16-thread execution of the Intel Core i7, which needs further research and it is left as future work.

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 exposed another problem that significantly affects the parallelization of the seam map computation. Figure 7 illustrates the performance results of the seam map. The Figure shows that barriers impose a substantial overhead, resulting in a zero gain in performance. In the multithreaded implementation, 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 locks inflict less overhead. However, we are only able to achieve 26% and 60% improvement with 2 threads on the Intel Xeon and Intel Core i7, respectively. This speedup is minor in comparison to the speedups achieved for the energy function. Beyond two threads, we see a large drop in performance even though both systems have eight hardware threads.

5.2 GPU performance evaluation

5.2.1 Naive-non-aligned energy function

On the 8800GT, the *naive-non-aligned* method achieves 5.7x and 8x performance improvement over the single-threaded CPU implementation executing on the Intel Xeon and Core i7, respectively, as shown in Figure 8. This performance improvement is similar to that of the multithreaded CPU implementations. However, this implementation does not take advantage of the GPU's wide memory bus. Its memory access patterns are not coalesced due to the data not being aligned. Since the *naive-non-aligned* method only utilizes three bytes per pixel, a warp will only load 96 bytes and a half of a warp will load 48 bytes. Such memory access pattern is not aligned. The *naive-non-aligned* method was initially designed with the G80 architecture in mind. However, with minimum modifications, this implementation yields 89.7x and 62x performance improvement on the Fermi GTX580, over the single-threaded implementation running on Intel Xeon and Intel Core i7, respectively (see Figure 9).

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 8x to 18x and 5.7x to 12.4x on their respective systems as shown in Figure 8. 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 aligned 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

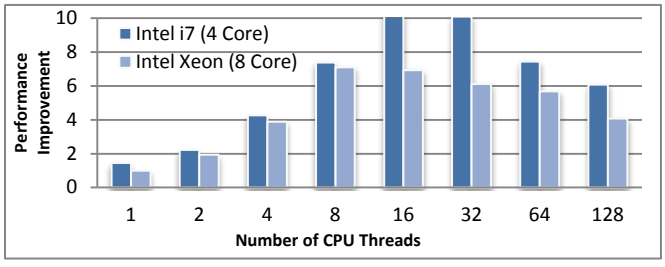


Figure 6: Improvement of the energy function over the single-threaded executing of the Intel Xeon for a 1200x900 image.

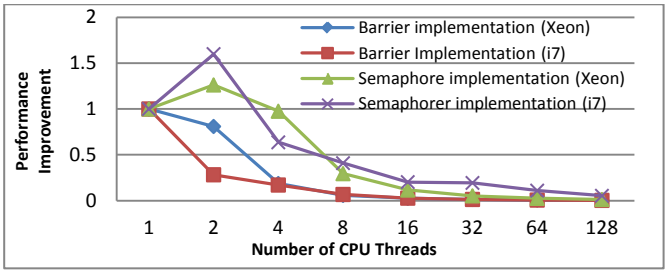


Figure 7: Performance of multi-threaded implementations of Seam map

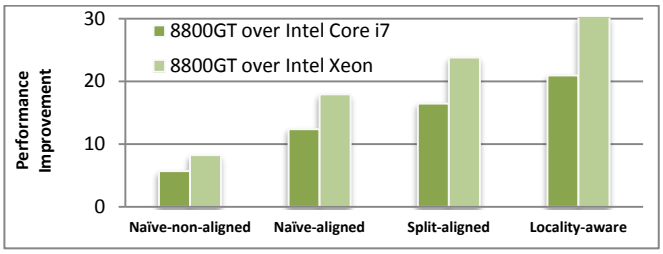


Figure 8: speedup of the energy function over the single threaded CPU

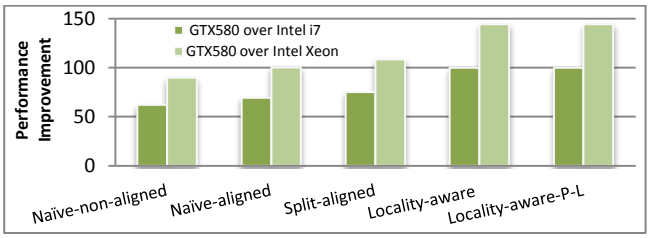


Figure 9: Improvement of the energy function on the GTX580 (Fermi) over the single threaded CPU implementations

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 100x and 69x over the respective Intel Xeon and Core i7 single-thread CPU implementations (see Figure 9).

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 24x and a 16.5x improvement over the single-threaded CPU version on the Intel Xeon and Core i7, respectively, as shown in Figure 8. As expected, the CUDA profiler reveals that for a 1200x900 image, approximately, only 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, the *split-aligned* achieves 108.6x improvement over the Intel Xeon CPU and 75x over the Intel Core i7 as shown in Figure 9.

5.2.4 Merging the split-aligned method

The l*ocality-aware* method described in section 4.2.3 achieves the highest performance improvement on both GPUs for the energy function computation. By merging the computation of the two derivatives in a way that the number of coalesce loads remains close, and by further taking advantage of locality of accesses, we manage to improve the performance of the energy function by 144.5x and 100x over the single-threaded CPU version on the Intel Xeon and Core i7, respectively, as shown in Figure 9. In addition, when executed on the 8800GT, this method shows a performance improvement of 30x and 21x over the Xeon and Core i7 single-threaded version (Figure 8).

5.2.5 Seam map

The seam map GPU implementation exhibits approximately 4x performance improvement over the single-threaded CPU implementation on the Intel Xeon and no improvement over the Intel Core i7 single-threaded implementation (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 the lacking of optimal methods for synchronizing threads among different blocks. Launching the kernel 899 times for a 1200x900 image imposes a significant overhead. Approximately 57% of the seam map execution time is due to kernel launch overhead. Minimizing the launch overhead could potentially improve the performance by a factor of two.

5.3 Evaluation of total execution time of the resizing operation on the GTX580

As previously stated, the energy function and seam map computations account for the largest fraction of the execution time of seam carving. 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 copy 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 to 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 10 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 11 shows the performance improvement for the entire operation.

Figures 10 and 11 illustrate that the GPU methods perform better than the CPU methods, especially when the size of the image increases. Overall, Figure 11 shows that the total execution time of the best resizing implementation on the GTX580 is about 6x faster than the best single-threaded CPU implementation and over 2x faster than the best multithreaded CPU implementation. The best execution time on entire operation is achieved with the *locality-aware* method using *page-locked* memory. The reason is that the CUDA run-time environment can optimize the 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*, to take advantage of *page-locked* memory, which yields the best overall performance as shown in Figures 10 and 11.

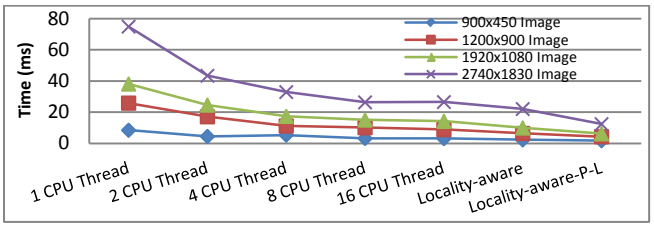


Figure 10: Total time to copy to and from the device, compute the energy function and seam map on the GTX580 and Core i7

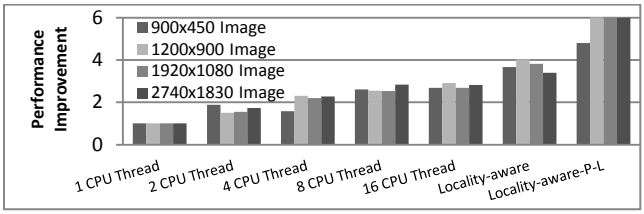


Figure 11: Total time to copy to and from the device, compute the energy function and seam map on the GTX580 and Core i7

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, which makes it difficult to perform on large images or videos in real-time.

To the best of our knowledge, this paper is the first to implement a real-time content-aware resizing method on GPUs. Our implementation works very well on computing the energy function (over 100x and 144x 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 4x speedup over the Intel Xeon and no improvement over the i7). A recent work [10] 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 and future work

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

In this paper, we exploit the 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* method using *page-locked* memory, which achieved a performance improvement of 100x over the best single-threaded execution time and 14x over the best CPU multithreaded version of the energy function executing 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.

In the future, we are planning to vectorize the CPU implementation to take advantage of the SIMD instructions on the Intel CPUs. Another important part of our future work is to find a better approach to parallelize the seam map computation.

8 Acknowledgement

We would like to thank the anonymous reviewers for their review. We also would like to thank Harry Bock for his contributions. This work was supported in part by US National Science Foundation grant CCF-1117467.

9 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, Vol. 20, No. 11, pp. 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, pp. 95–104

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

[4] Ciocca G, Cusano C, Gasparini F, et al. "Self-adaptive image cropping for small displays," IEEE Trans Consumer Electr, 2007, Vol. 53 No. 4 pp.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, pp. 771–780

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

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

[8] Intel, "2nd Generation Intel Core Processor Family Desktop," (2011). [online]. Available: http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html

[9] Intel, "Quad-Core Intel Xeon Processor 5400 Series, Datasheet" (2008). [Online]. Available: http://www.intel.com/assets/PDF/datasheet/318589.pdf

[10] H. Hua, F. TianNan, R. Paul L, Q. Chun, "Real-time Content-aware Image Resizing," Science in Chine Series F: Information Sciences, 2009, Sci. in China Press.