A Parallel Image and Video Processing Application

H. Parker Shelton, Adam Feinstein

Abstract— **We built a go-fast image and video editor.**

I. Introduction

MULTIMEDIA

II. GPUs and CUDA

Dedicated graphical processing units (GPUs) have become more prevalent in modern computers. This specialized hardware differs from traditional CPUs in that it is designed to preform parallel arithmetic computations. While the GPU is almost always used for 3D intensive applications, such as gaming, it is rarely tapped into for general computing tasks. In order to use this computational power in a general computing environment, NVIDIA has developed Compute Unified Device Architecture (CUDA). CUDA-enabled code can be executed on all of NVIDIA's modern GPUs (those developed in the past three years). These include the consumer GeForce series, the professional Quadro and Tesla cards, and even the ION series of GPUs, found in netbooks and small form factor media centers.

CUDA allows the programmer to allocate the GPU's multiprocessors in order to run parallel computations. A GPU's hardware is divided into blocks and threads. Each block contains a fixed number of threads, and each thread is capable of running one calculation. Each active thread must run the same calculation, but each thread may preform the calculation on a different value in memory. A diagram of the task allocation can be seen in Figure 1.

Memory is not shared by the GPU and the CPU. Memory must be copied from the host (CPU) to the device (GPU) when a calculation must be preformed. This memory transfer runs in $O(n)$ time. Because memory must be copied, certain operations such as matrix subtraction are not well suited to be parallelized. A matrix subtraction on the GPU would take $O(n)$ time for the memory transfer and $O(n/threads)$ time for the operation, whereas whereas the same operation on the CPU will take $O(n)$ time.

Memory on the GPU is divided into global memory, which is accessible from each thread in each block, shared memory, which is accessible from each thread in a given block, but not threads in other blocks, and register memory, which is only accessible from a single thread. The memory diagram for a GPU is shown in Figure 2. Registers are accessed faster than shared memory, which is in turn accessed faster than global memory. When data is moved from the CPU to the GPU, it is placed in global memory.

The authors are with the Departments of Computer Science and Electrical and Computer Engineering, The Johns Hopkins University, Baltimore, MD 21218. Email: {parker.shelton, afeinst1}@jhu.edu.

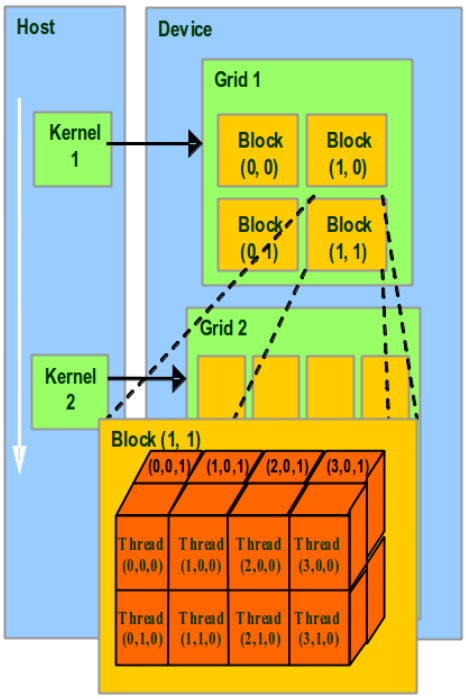

Fig. 1. Task allocation between grids, blocks, and threads in a GPU

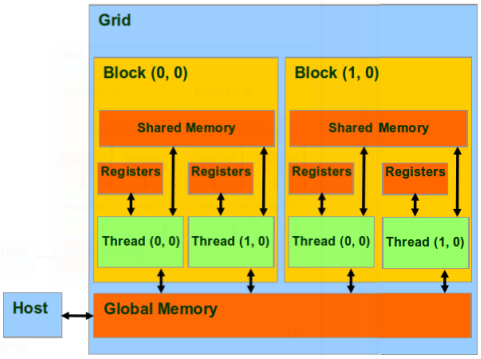

Fig. 2. GPU memory heirarchy

III. User Interface

A significant portion of the development of the editor was spent designing and implementing a graphical user interface (GUI) to display the result of the editing operations performed. The user interface was written in C++ using a cross-platform framework called Qt, available from Nokia under a LGPL open-source license. The use of Qt allowed for rapid GUI development and created a beneficial abstraction of GUI commands into image editing function calls.

As Qt also contains a multimedia framework, its use sim-

plified the extension of the program to standard image formats; the QImage class provided support for reading .bmp, .gif, .jpg, .png, .pgm, and .tif files from disk, as well as writing .bmp, .jpg, .png, .tif with an optional quality setting. In addition to the standard image formats provided by QImage, custom image (.ppc) and video (.pvc) codecs were created and encoders and decoders implemented. These will be discussed further in a later section.

The GUI is arranged to display the original image on the left and the modified image on the right. The scale of the images can be changed using zoom functions, and the modified image can be reset to the original image without requiring reopening, and thus decoding, the file from disk. This functionality was provided in lieu of undo/redo functionality, which would require either a large amount of memory to be allocated or the creation of inverse functions for all implemented functionality. Videos can be played using a control bar at the bottom that also allows for frame by frame inspection.

Compression is implemented so as to display a new dialog containing both the modified media and the compressed version. This allows for side-by-side comparison of the compression quality achieved as well as comparison among multiple compression levels. The peak signal-to-noise ratio (PSNR) is also reported as a measure of the compression quality chosen.

IV. Image Editing

Several common image editing operations were explored and then implemented in the editor. These were then further extended to video editing by applying them to each frame sequentially. Each operation implemented is discussed in detail below.

A. Grayscale

Because the image was stored internally in ARGB format, the conversion to grayscale requires application of the luminance formula to each pixel:

$$luminance = 0.3 * red + 0.59 * green + 0.11 * blue$$

This is in contrast to an image in YCrCb format, where the color space would simply be thrown away. The luminance result was then clamped to fit into an 8-bit unsigned char ($0 \leq$ value ≤ 255) and assigned to all of the red, green, and blue color values. All values resulting from color calculations in all future editing operations can be assumed to be clamped to the same range before being assigned to the resulting image.

B. Brightness

Increasing the brightness of an image can be achieved by scaling each of the RGB values by a specified factor. This adjusts the luminance of the entire image.

C. Contrast

Contrast is a measure of the distance of each pixel from the average luminosity. In order to increase contrast, a linear combination of the average luminosity and the RGB values are taken:

$$red = (1 - factor) * average\_lum + (factor) * red$$

D. Saturation

In contrast to contrast, increasing saturation requires scaling the RGB values of a pixel away from its luminosity, rather than the global average luminosity. The formula appears the same as for contrast, with $average\_lum$ replaced with lum.

E. Crop

Cropping was implemented to remove unwanted portions or highlight desired portions of an image. The function takes four parameters defining the top-left and bottom-right corners of the desired sub-image.

F. Rotation

Rotation was implemented using both Gaussian sampling and nearest-neighbor sampling of the original image in order to determine pixel color values in the rotated image. If the image rotation is a multiple of 90°, nearest-neighbor sampling is used to generate a one-to-one matching, while Gaussian rotation is used otherwise. The Gaussian used was defined with variance 0.6 and radius 4 pixels.

G. Scale

Scaling was also implemented using Gaussian sampling, also with variance 0.6, but with a radius that linearly depends on the scaling factor to compensate for the enlarged (or shrunken) size of the image.

H. Blur

Blurring was performed using a 3x3 convolution filter that weights the original pixel heavily, but allows for contributions from neighboring pixels:

$$mask[3][3] = \left\{ \left\{ \frac{1}{16}, \frac{2}{16}, \frac{1}{16} \right\}, \left\{ \frac{2}{16}, \frac{4}{16}, \frac{2}{16} \right\}, \left\{ \frac{1}{16}, \frac{2}{16}, \frac{1}{16} \right\} \right\}$$

I. Edge Detection

Edge detection was similarly implemented as a 3x3 matrix convolution, but with a mask that highlights pixels with colors sufficiently different from its neighbors:

$$mask[3][3] = \{\{-1, -1, -1\}, \{-1, 8, -1\}, \{-1, -1, -1\}\}$$

V. Compression

The two compression algorithms that were implemented both sequentially and in parallel were the biorthogonal 9/7 discrete wavelet transform (DWT) with a dead-zone quantizer for images and a closed-loop DPCM quantizer with motion estimation for videos.

A. Image Compression

The particular implementation of the DWT that was used was the lifting implementation. This implementation is the same one used in the JPEG2000 standard. The forward transform (FWT) works on an array of real values. The values are filtered into low and high frequency subbands, which are each downsampled by a factor of two. This is done with two preditction filters and two update filters. A prediction filter sums each odd index with a scalar multiple (λ) of the sum of its neighbors, while an update filter does the same, but for even indices. The coefficients used were $\lambda_{predict0}$ = -1.586134342, $\lambda_{update0}$ = -0.05298011854, $\lambda_{predict1}$ = 0.8829110762, and $\lambda_{update1}$ = 0.4435068522. The result is an array of the same length, with the high pass sub-band retained in the odd indices of the array and the low pass sub-band is retained in the even indices of the array. The inverse transform (IWT) is preformed using the same computational structure, but with the filters in reverse order and using the negatives of their coefficients.

A sketch of the parallel algorithm for image transformation is as follows. First, the image is transferred into GPU memory. The different color components must be transformed separately, so the 32 bit 0xffRRGGBB image data is rearranged such that all the red, blue, and green values are in continuous memory. This is done in one parallel step, where each thread moves one color value, and is not done in-place. Each row of each continuous color region is then transformed in parallel. Due to the iterative nature of the FWT, the algorithm cannot be preformed in one step. Once each row is transformed, each continuous color region is transposed separately. The FWT is then applied to each row (previously column) of each continuous color region in parallel. This completes the one level 2D FWT. Another transpose, FWT, transpose, and FWT is applied to perform the two level 2D FWT.

The parallel IWT is simply the same process with filter order reversed and negated coefficients. The IWT must also rearrange the continuous color regions into 32 bit 0xffRRGGBB format.

Once the image was transformed, a deadzone quantizer was applied. This quantizer zeroes out all coefficients smaller than a given threshold, and rounds all of the coefficients outside the deadzone to integers.

B. Video Compression

A closed-loop DPCM uniform quantizer with motion estimation was implemented for video compression. A sketch of the algorithm is as follows. For the first frame, the previous frame is set to all zeros. The encoded previous frame is subtracted from the current frame, $d[n]$, which is then quantized and inverse quantized, $\hat{d}[n]$. This value is summed with the encoded previous frame, $\hat{x}[n-1]$ and set as the new encoded previous frame for the next iteration, $\hat{x}[n]$. Motion vectors are then found for each 8x8 block which minimize $\sum|x[n]$ - $\hat{x}[n-1]|$, where $\hat{x}[n-1]$ is shifted by the motion vector, and the quantized result along with the motion vectors are sent to the communication channel for encoding.

Decoding is simpler than encoding and involves simply summing the previous decoded frame, $\hat{x}[n-1]$, shifted by the motion vectors, with the inverse quantized incoming frame, $\hat{d}[n]$.

The parallelization comes into play with the search for optimal motion vectors. As previously stated, simple operations such as matrix addition or subtraction do not offer significant speedup when preformed on the GPU. An exhaustive search for motion vectors lends itself well to parallelization, however. Each block on the GPU is assigned one 8x8 block of pixels and is given the current and previous frames. Each thread within that block is assigned one motion vector, and calculates the sum of the absolute values of the differences between the pixels within the current and previous block, shifted by the motion vector. From this pool of all potential motion vectors, a parallel reduction is preformed, reducing the time to find the minimum value in an array from $O(n)$ in a serial implementation to $O(log(n))$ in parallel. The first iteration of the reduction compares n/2 sets of 2 elements, and returns the smallest of each. The next iteration then compares n/4 sets of 2 elements, and returns the smallest of each. This continues until the smallest element is found. In this case, the motion vector with the smallest mean absolute difference is found and returned.

Sample code which came with the CUDA SDK was modified and used for both the reduction and the transpose. A second set of non-parallelized algorithms are also supplied to compare with GPU accelerated algorithms, and for users who do not have a compatible GPU.

VI. Theorem

VII. Performance Metrics

Since speed was the ultimate goal of this project, time elapsed for each operation was measured. In addition, image quality per compression and filesize after encoding were measured. Each number in this section is the average of twenty runs.

A. Image Compression

On the GPU, lena.jpg, a 512x512 image, was transformed and inverse transformed in 136.95ms. The same pair of transforms on the CPU took 374.95ms. The resulting speedup was 2.74x. On the GPU, monarch.jpg, a 768x512 image, was transformed and inverse transformed in 198.1ms. The same transforms on the CPU took 557.35ms. The resulting speedup was 2.81x.

As seen in the graphs PUT FIGURE NUMBERS HERE, the program was able to maintain a relatively high PSNR of about 30dB while zeroing out up to 95 percent of the coefficients. The transform and inverse transform are not lossless, as even if the deadzone of the quantizer does not exist, it still rounds coefficients to integer values.

B. Video Compression

Glasgow100.qcif and bus.cif were used to test video encoding speed. Glasgow100.qcif contains 100 frames of 176x144 video. Bus.cif contains 150 frames of 352x288 video.

On the GPU, glasgow100.qcif was encoded in 3465.44ms. On the CPU, glasgow100.qcif was encoded in 34274.95ms. The resulting speedup was 9.89x. The On the GPU, bus.cif was encoded in 21303.86ms. On the CPU, bus.cif was encoded in 206443.33ms. The resulting speedup was 9.69x.

As seen in the graphs PUT FIGURE NUMBERS HERE, the program was able to maintain a PSNR of over 30dB when it had more than approximately 100 quantization levels. Interestingly, there are spikes of very high quality video for quantization levels which are powers of two. The PSNR when there were 512 quantization levels was infinite, meaning that compression was lossless.

C. Encoding

D. Image and Video Editing

On the GPU, world.jpg, a 1024x1024 image, was edge-detected in 153.9ms. The same operation on the CPU took 372.9ms. The speedup was 2.42x. These results can also be representitave of the bluring operation, as both operations preform a 3x3 pixel convolution, just with a different mask.

On the GPU, world.jpg was turned to grey in 53.45ms. The same operation on the CPU took 102.85ms. The resulting speedup was 1.92x. This result is also representative of the contrast, saturate, and brighten opearations, as they are all per-pixel operations.

Rotate, crop, and scale had no analogous GPU functions. Video editing was preformed framewise, so the speedups would be the same as for the still image case.

VIII. Conclusion

For glasgow100.qcif, on the GPU, each frame was calculated in 34.6544ms. This time equates to encoding 28.85 frames per second. This means that the application can preform the exhaustive search for the optimal mean absolute difference motion vectors in real time. Each frame in bus.cif was encoded in 142.02ms on the GPU, which equates to 7.04 frames per second. For a video of this size, real time motion estimation was not achieved.

Due to slight arithmetic differences, the GPU and the CPU did not always find the same vectors. In motion-ambiguous regions, like the sky in the beginning of glasgow100.qcif, there was some variation with the motion vectors.

Speedups were noticed for image operations and compression, but not as signifigant as the video speedups. As stated earlier, per-pixel operations are not expected to receive large speedups from parallelization. The time it takes for image compression could be further reduced by breaking the image into tiles and using shared memory on the GPU, but this would lead to tiling artifacts, one of the things the DWT is supposed to get rid of.