Abstract

Using a genetic algorithm, one can find a set of polygons that approximately represent an arbitrary image. We tested three implementations of this genetic algorithm—solely on a CPU, solely on a GPU and with a combination of the two—in order to compare performance. We used CUDA and the combination of OpenCL/OpenGL for this study. The CUDA speedup was at least 17x for a 256 × 256 pixel image. The best speedup seen for the OpenCL code was roughly 3.3x for a 512 × 512 pixel image using pixel buffer objects (PBO) with manual read from PBO and copy to an OpenCL memory object.

1 General Description of the Problem and Solution Methods

Genetic programming provides an approximate solution for various optimization and functional minimization problems. It gets its name from the fact that the operation of the algorithms mirrors the evolution of organisms. Let us consider the functional minimization case. We start with a population of candidate solutions. Based off of some criteria for fitness we filter out some of the candidate solutions from our population. For traditional functional minimization, if we have a population of \( N \) candidate solutions we could filter out \( M \) solutions with the highest \( \chi^2 \). We then take the remaining solutions, clone them and modify them. This stage mirrors the reproduction and mutation of organisms. We then repeat the process of filtering, cloning, and changing our solutions until we have converged to an acceptable solution. With the right mutations and filtering algorithm we can generate solutions to problems that more traditional functional minimization algorithms would fail at. Because with many implementations the mutations are random, converging to a good enough solution takes many generations and thus necessitates efficient implementation and fast hardware.

One such problem is finding a finite list of polygons that best represent an image when drawn. If the list of polygons is small enough, and accurately represent the picture, this is a compression algorithm. This was inspired by Roger Alsing [1].

We first need to quantify what we mean by ‘best represent an image.’ For each pixel in the image we find the distance squared in RGB color space between our original image and the rendered polygons. We define our quality of fit (QOF) as the sum of this value over all of the pixels in the image.

\[
QOF = \sum_{\text{pixels}} \left( (R_{\text{image}} - R_{\text{polygons}})^2 + (G_{\text{image}} - G_{\text{polygons}})^2 + (B_{\text{image}} - B_{\text{polygons}})^2 \right) \tag{1}
\]

Where R, G, and B represent the values of red, green and blue at each pixel respectively. A smaller QOF means that the list of polygons better represents the image.

Each member of our population of candidate solutions is represented by a list of polygons. At the selection stage we filter the members by selecting those with the lowest QOF. The mutation of a member of
our population entails modifying the set of polygons in some way. Possible modifications include adding a random polygon, removing a polygon, flipping the order of polygons in the list (for the non-commutative alpha blending this matters), modifying the position of the points of a polygon, and changing the color of the polygon. Having small modifications to the polygons in the list as part of the mutation step greatly reduces the number of steps it takes to converge to a reasonable solution. Having this eventually converge relies on having a reasonable pseudo random number generator.

We tested three separate implementations of this genetic algorithm. One is implemented solely on the CPU not using a graphics library, another is only on an NVIDIA GPU programmed in CUDA, and the final is implemented using OpenCL in conjunction with OpenGL.

1.1 Serial Algorithm Not Using OpenGL

For our serial solution we only considered a population size of 2. At each step we remove the member of the population with the higher $QOF$, and then copy and mutate the winner to restore our population. So at each step we compare the winner of the previous generation with a mutant created from it. This forces the $QOF$ to a monotonically decreasing function of time. Because this code was meant to be ported to a GPU all of the operations are custom made, the rendering of polygons included.

The drawing of polygons was done in a very straightforward manner. For each $y$ value in the image, $y$ spans from 1 to the height of the image, the points where the polygon intersects the horizontal line with that $y$ value are found and then the appropriate pixels between these intersections are colored with an alpha blending rule. For each pixel we are modifying let $c_{prev}$ be the RGB color values before we modify it and $c_{poly}$ be the color of the polygon and $\alpha$ be the alpha value of the polygon, the color of the new pixel is $\alpha \cdot c_{poly} + (1 - \alpha) \cdot c_{prev}$ if we normalize $\alpha$ to 1. Because the rendering code is hand made it most likely is not as optimized as the OpenGL code. This is a primary weakness of this implementation.

Because reduced programming time and efficiency of operation were valued we used the linear congruential generator featured in Numerical Recipes [2].

Analyzing the speed of the algorithm is rather straightforward. Let us set the dimensions of our image as $M \times N$ and the number of polygons in a list as $P$. Let’s set the amortized cost of reading a polygon from memory as $m_g$ and the cost of modifying a pixel in our drawn image as $m_p$. If we make the assumption that the number of points in each polygon is less than the number of polygons and the number of pixels in the image we should expect the mutation time to be $O(P \times m_g)$ where all of the other operations are of order constant. Drawing the polygons in the worst case takes $O(P \times m_g + P \times M \times N \times m_p)$. Calculating the $QOF$ for each takes $O(M \times N \times m_p)$. Clearly the drawing is the limiting step in this operation.
Figure 1: An example of the polygonal fit. The first block is the image we are fitting to, the others are successive approximate solution. These approximate solutions are separated by an equal number of generations. Notice that the rate of improvement declines quickly.

(a) Starting Image  
(b) Polygonal Fit after 4 Million Generations

Figure 2: Comparison of starting image and computed image after 4 million generations
2 Parallel Implementation on a NVIDIA GPU

To divide the task into independent blocks meant to be run in parallel on a GPU, implementing the genetic algorithm is a little less straightforward. Because we need some level of synchronization between blocks each iteration refers to a separate running of the kernel. The thread blocks perform two very different tasks. All of the blocks except for the 0th block are assigned a different rectangular subsection of the image. They take all of the polygons for the current mutation and draw them on that section, calculate the $QOF$ over that section and then store the local $QOF$ in a global array. The 0th block sums the local $QOF$’s calculate during the previous iteration to generate a $QOF$ for the mutation the other blocks had used on the previous iteration. It then uses that $QOF$ to determine whether the mutation the other blocks will see on the next iteration will come from the mutant of the previous generation or the master from the previous generation. Master refers to the set of polygons that the mutant was generated from. Because this is a bit complicated it helps to see a picture, see Figure 3.

2.1 Details of the Implementation

The polygon sets, $QOF$ arrays, and the image were stored in global memory. Because the 0th thread block is summing the $QOF$’s and modifying a set of polygons while the other thread blocks are calculating those local $QOF$’s from a different set of polygons, there were two separate global $QOF$ arrays and mutant polygon lists. So while the 0th block is using one of the set the other blocks are using the other set. This was to ensure that there would be no race conditions with writes and reads. Which set is used by each is determined by whether the iteration number was even or odd.

The mutation operation performed by the 0th block is identical to the serial case. The 0th block also handled keeping track of the sets of polygons. Four sets of polygons were stored in memory. Along with the mutant the not 0th thread blocks are finding the local $QOF$’s for and the one that the 0th block is generating for the next iteration, the previous generation’s mutant and master are stored. The details of keeping track of the sets of polygons and $QOF$ arrays is a little tedious and not very illustrative. What it comes down to is that the 0th block needs to sum one $QOF$ array, copy two polygon sets and modify one.

The section of the image that each thread block (other than the 0th) is calculating the $QOF$ over is limited by the shared memory. In order to keep the time to it takes to modify a pixel in the subsection low the goal was to keep the section entirely enclosed in shared memory. Each of the threads in the block are assigned a horizontal line in the section. Each thread draws all of the polygons in that line, and generate the $QOF$ for that line. The 0th thread in the block then sums the $QOF$’s for each line and then writes the value to the global array. Care was taken to avoid bank conflict.

We will use the notation of the serial runtime analysis the dimensions of our image as $M \times N$ and the number of polygons in a list as $P$. The amortized cost of reading a polygon from memory is $m_g$, the cost of modifying a pixel in our drawn image as $m_p$ and the cost of reading or writing to a $QOF$ array is $m_a$. The number of thread blocks is $T + 1$. Each subsection of the image that the individual thread blocks will cover is $m \times n$. $m$ is the number of threads in the block. The expected time it takes for the 0th thread block to run is $O(2P \times m_g + m_a \times T)$. The expected time it takes for the other thread blocks is $O(P \times m_g \times m + P \times n \times m_p + n \times m_p + m \times m_a)$. This represents a significant speedup over the serial case.
Figure 3: Selection and reproduction are handled by the 0th thread block. Fitness is handled by all of the other thread blocks. The squares on the top row represent the set of polygons that the $N - 1$ blocks are finding the local QOF for. The squares in the bottom row represent the set of polygons that is being modified for use with the next generation. During the first iteration the 0th block modifies our red image to generate the green hexagon, while the other blocks calculate the QOF of the set of polygons represented by the blue square. Each of the thread blocks calculates the QOF for a subsection of the image and then stores the value in an array. During the second iteration the 0th block sums the QOFs from the blue image stored in the array and then compares the sum with the QOF of the polygons represented by the red block. It finds that the QOF of the red is lower so it modifies that image to form the blue star image. During the next iteration, while the other blocks are calculating the QOF of the blue star image, the 0th block compares the previous generations modified image with the previous winner and finds that the previous candidate solution is better so it modifies it to form the next generation. During the fourth iteration the 0th blocks finds that the blue star is better than the green hexagon so it uses that to generate the next mutation. While this is happening the other blocks are evaluating the local QOFs for a modification of the green hexagon.
3 OpenCL and OpenGL Interaction

3.1 Overview

OpenGL is a popular graphics programming language that efficiently renders triangles using the GPU. This API abstracts the underlying hardware to allow for easy and efficient programming of image processing tasks. This project used extensions from OpenGL 2.0 and as such, requires a graphics card with driver support of OpenGL 2.0 or higher.

OpenCL is a recently created programming language that allows for scientific computing on heterogeneous hardware. The language supports run-time compilation targets of CPUs or GPUs. For graphics card, OpenCL support is provided by the driver. This project used OpenCL 1.0 for the computationally intensive routines.

3.2 Rendering Platform

We used an Intel Kentsfield series Q6600 quad core processor with 2 GB of RAM, and an ATI HD 4830 graphics card with 512MB of on-board RAM (RV770LE GPU). The motherboard supported PCI Express 2.0. We used the latest ATI Stream SDK (v2.1 at the time of this report) loaded on a Windows 64-bit Vista machine running the latest ATI Catalyst display driver (10.4). Code development was done using Visual Studio 2008 for compiling and debugging. Most of the code development was done on the previous Stream revision v2.01, which did not explicitly support OpenCL/OpenGL interoperability features described in [3]. The project was compiled in 32-bit mode.

3.3 OpenCL Limitations

The latest ATI Stream SDK v2.1 was released on May 3, 2010, which enabled support for OpenGL/OpenCL interoperability. We were unable to use the image features of OpenCL (including OpenGL frame buffer objects to OpenCL memory objects) because this is only supported in HD5xxx cards and higher and we were using a HD4xxx (RV7xx GPU) series card. To work around this, we used OpenGL pixel buffer objects (PBO) for passing data to OpenCL.

3.4 Implementation Details

The code was developed on top of the ATI Stream SDK class structure located at [4]. This made for a clean interface to develop the OpenCL portion of the code, which managed the computation of the fitness function on the GPU. The OpenGL portion of the code was encapsulated in its own class and managed the vertex positions and associated vertex color, rendering OpenGL objects on the screen, and mutating the vertex positions and colors.

The code used two OpenGL buffer objects: a vertex buffer object (VBO) for storing the vertex positions and color data/vertex and a pixel buffer object (PBO) for storing the rendered pixels. Figure 5 details the steps required for setting up the OpenGL/OpenCL computational environment. OpenCL has functions for compiling kernels into targeted executable code, which we used in our implementation. The main use of this feature is to do just-in-time compilation to take advantage of driver patches that improve computational performance. The SDK included support for platform independent reading and writing of bitmaps; we used these features to read in the image to approximate and to save intermediate images.

The genetic algorithm heavily relied on uniform random numbers for computing new triangles. We used the Mersenne Twister algorithm [5, 6] to compute random numbers used for mutation and reseeded the algorithm with the wall time every 100,000 generations.
The fitness portion of the code was deemed to be the bottleneck in the computation so that piece was parallelized using OpenCL. The ATI Stream SDK has an OpenCL implementation of the SIMD version of the Mersenne Twister algorithm so it is possible to perform the mutation portion of the genetic algorithm in OpenCL by directly reading and writing an OpenGL VBO.

The computational structure of OpenCL is shown in Figure 4. The fitness computation reduction was done in two stages: compute the distance on the workgroup shared local memory; then output the results back to the host to finish the computation in host memory. The reduction was done with a tree structure to allow for parallelization using more of the available streaming cores. Barriers are used for synchronization on each pass of the reduction and after the computation of the distance. The final sum is approximately $O(1)$ because it is limited to the number of work group blocks times 4. The OpenCL code is shown in Listing 1 and is a modification of the ATI Stream Sample “Reduction” kernel.

We implemented two versions of the code that were conditionally compiled with `#define` directives. Both versions used PBOs to speed up the transfer of pixels from the screen back into GPU memory. The first version took the pixels in the PBO and directly copied them to an OpenCL memory object using `memcpy`. The second version used the new features of ATI Stream SDK v2.1. The PBO was bound to an OpenCL memory object using `clCreateFromGLBuffer` and prior to kernel execution, the PBO was acquired with `clEnqueueAcquireGLObjects`.

![Figure 4: Hierarchy of OpenCL as described in [7]](image)

### 4 Experimental

For the CUDA code, we compared two cases: 1) All code in the inner loop was on the CPU. 2) All code in the inner loop on the GPU.

For the OpenCL/OpenGL code, we wanted to look at the performance of OpenCL for different compilation conditions. We compared the following test cases: 1) Fitness function implemented on CPU using standard $O(n)$ algorithm. 2) Fitness function implemented in OpenCL using reduction algorithm in Listing 1 compiled for CPU (use standard PBO-`memcpy` implementation). 3) Fitness function implemented in OpenCL using reduction algorithm in Listing 1 compiled for GPU (use standard PBO-`memcpy` implementation). 4) Fitness function implemented in OpenCL using reduction algorithm in Listing 1 compiled for GPU (use...
Figure 5: Flowchart for OpenCL/OpenGL interoperation
clCreateFromGLBuffer implementation).

Listing 1: OpenCL Kernel Code

```c
__kernel void GAreduce(__constant uchar4* inputRefImage, __constant uchar4* inputImage, / 
    __global uint4* output, __local uint4* sdata)
{
    // load shared mem
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int localSize = get_local_size(0);

    sdata[tid].x = (inputRefImage[gid].x - inputImage[gid].x) * (inputRefImage[gid].x - inputImage[gid].x);
    sdata[tid].y = (inputRefImage[gid].y - inputImage[gid].y) * (inputRefImage[gid].y - inputImage[gid].y);
    sdata[tid].z = (inputRefImage[gid].z - inputImage[gid].z) * (inputRefImage[gid].z - inputImage[gid].z);
    sdata[tid].w = 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    // do reduction in shared mem
    for(unsigned int s = localSize / 2; s > 0; s >>= 1)
    {
        if(tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if(tid == 0) output[bid] = sdata[0];
}
```

5 Results

We ran test cases for 56x56, 128x128, 256x256, and 512x512 pixel images. We measured the time to compute 10,000 generations and computed the speedup over baseline case (labeled 1 in the previous section).
Figure 6: Comparison of CPU and CUDA Figures of Merit

Figure 7: Comparison of OpenCL and OpenGL Figures of Merit
6 Conclusions

When comparing the serial implementation to the CUDA implementation of the code a massive speed up was seen as was expected.

As can be seen in the speedup graph, the OpenCL implementation on the CPU was substantially slower than the naive case. This might have something to do with the structure of the reduction code since the CPU is not massively parallel like the graphics card so the distribution of work into many small computational jobs is not as efficient as one large SIMD operation. The OpenCL code using PBO/memcpy was the fastest at about a 3.3x speed up for the 512x512 pixel image case. The OpenCL using PBO/interoperability was about half as fast as the baseline code hinting that there might be some performance tuning issues with the new interoperability extensions. Since the ATI Stream SDK used GLUT and Win32 WGL functions, perhaps the use of the SDL library for creating the window might have a latent performance issue.

Convergence of the image to a true image is highly dependent on the structure of the fitness function: using RGB distance as described in Equation (1), the image will converge; however, using RGBA distance instead, the image will not converge.

7 Future Work

Pushing all the inner loop code into OpenCL would provide the best case comparison to the CUDA case in terms of speedup possible and total runtime comparisons. In addition, the OpenGL PBOs are asynchronous in nature, but the current implementation is essentially synchronous since we consider a population of 2. By increasing the size of the population at each generation, the computation can be further pipelined to improve the computational efficiency. In addition, further study of the interaction between OpenGL and OpenCL should be performed to determine where the bottleneck is in performance. The best case would be to use frame buffer objects (FBOs) to remove the need of using pixel buffer objects and OpenGL glReadPixels function. The use of FBOs would allow for off-screen rendering, which removes the constraints on the image height and width that exist for standard on-screen renderbuffers.

For the CUDA implementation, a lot of limitations were placed based off of the amount of shared memory. Large speed ups should be attained with future generations with higher CUDA capability.

References


