CUDA Parallel Particle Swarm Optimization for EvoLisa
Drew Robb & Joy Ding
CS264 Final Project 2009

Introduction

Particle Swarm Optimization

Particle Swarm Optimization (PSO) (http://www.swarmintelligence.org/tutorials.php) is a stochastic optimization technique inspired by swarm behavior such as the flocking behavior of birds. In PSO, each solution is a particle, or bird, with its own fitness value and "velocity". The velocity of a particle determines the direction and speed at which a particle is flying, but in general particles will follow the direction of the optimal particle. The population is initialized as a population of random solutions. In every iteration, each particle in the population calculates the "best" value it has achieved so far (local best), as well as the best value achieved by the entire population (global best). The particles accelerated as if there were attached to springs connecting them to the local and neighborhood optimum.

The velocity then is updated with

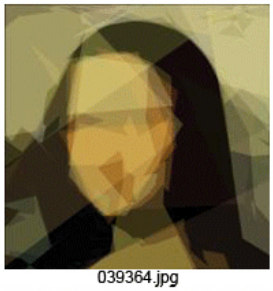
(a) v[] = v[] + c1 * rand() * (pbest[] - present[]) + c2 * rand() * (gbest[] - present[])

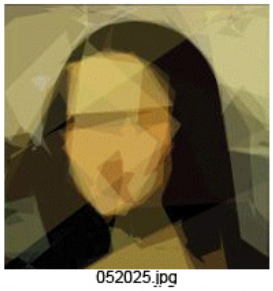
(b) present[] = present[] + v[]

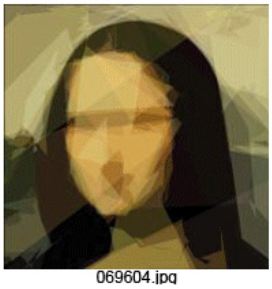
where v[] is the velocity, rand() is a random number between 0 and 1, pbest is the local best, gbest is the global best, present is the current particle, and c1&c2 are "learning factors" in our case 0.7. Each dimension was expected to lie in [0,1]. We can no boundary conditions however, so this allowed for triangles that filled the entire image. We limited the velocity in each interval below a random threshold, as this is an advisable technique.

Evolisa

After we decided to implement PSO, we wanted to choose a meaningful problem to apply the algorithm to. Evolisa is an genetic algorithm produced by Roger Alsing (http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/). The algorithm uses a DNA representation of a set number of polygons, and through random mutation, evolves the DNA to match a given goal image. The original Alsing genetic algorithm (GA) essentially compares the Parent and the Child image fitness (a pixel to pixel comparison with the goal image), allowing the most fit to survive and mutate. This way, with only 100 polygons, impressive results can be achieved (filename numbers represent number of iterations).


039364.jpg


052025.jpg


069604.jpg


099531.jpg


161713.jpg


343336.jpg


649291.jpg


904314.jpg

We implemented an algorithmically different version of EvoLisa on CUDA. Our problem uses the following approach: Initialize a set of S transparent triangles. The triangle coordinates and color channels can vary, thus there are 9 dimensional per triangle. Our problem uses the following approach:

Loop:

Pick one triangle at random from the set. Run Particle swarm optimization (PSO) to optimize a single triangle.

Unless the fitness is within a small the best fitness ever seen (or better), revert to previous state.

We choose this approach because other people had been working on the GA version before, and because we had reason to believe that this would be an algorithmic improvement. PSO is an extremely parallel algorithm, and we thought that the parallelism of PSO would perform better than the potential of a parallel GA (ie, with a large population size). To simplify the problem so that we could focus on the PSO parallelization rather than on polygonal complexity, we chose to use simple triangles rather than n-sided polygons.

Application

Our application is a command line application implemented in PyCuda. It takes inputs as jpgs (other inputs are possible also) and outputs jpg files. The jpg files are the representations of the input file as translucent, overlapped triangles. The program depends on the Jinja2(http://jinja.pocoo.org/2/) and Python image (http://www.pythonware.com/products/pil/) libraries.

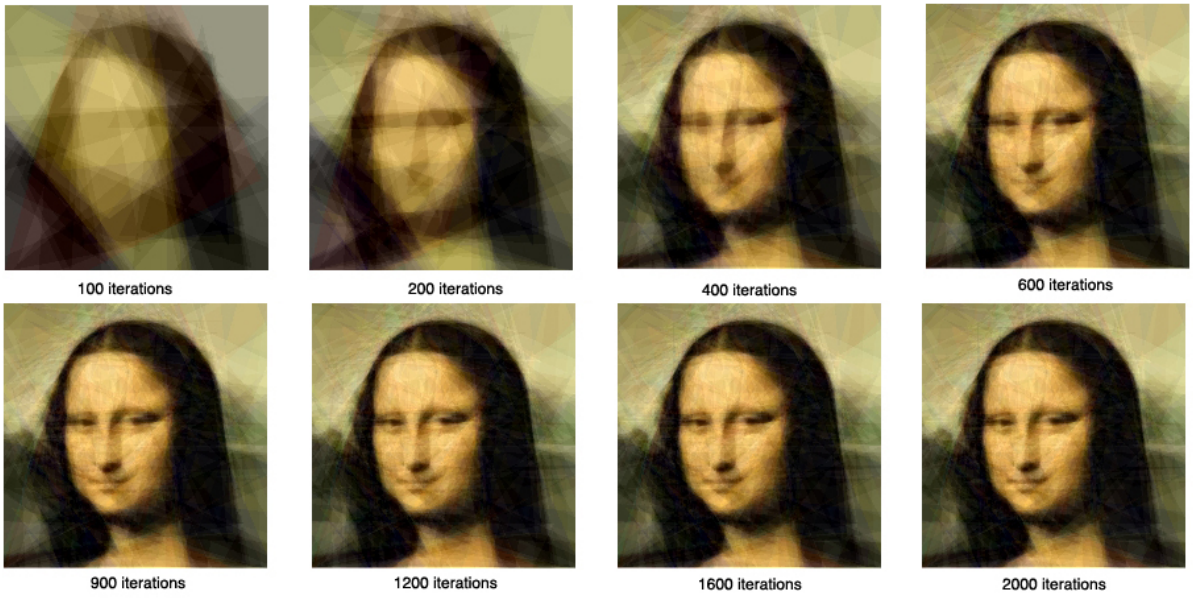
Performance

This project was really fun to optimize, as there were so many parameters and opportunities for parallelization. The various parameters we focused on for testing were population dimension, neighborhood size, number of triangles in the image and alpha values for those triangles. We found that visually, a neighborhood size of 20, a population size of 60, 1000 polygons, and an alpha value of 1/(number of triangles/50, i.e. 0.05 for an image with 1000 triangles) showed fairly efficient results after a few thousand iterations. We also tried many schemes for parallelization. We began by attempting to implement PSO as a state-space search in which each particle was a complete solution of all the polygons. However, we are working with about 1000 polygons in each image (which we have found to provide optimal results), the complexity of shading the entire solution many times at every PSO stage was slow. Additionally, solving a 9000 dimensional optimization problem in this method is extremely challenging. Instead, we choose to optimize a single triangle at each stage, in a mini 9-dimensional PSO problem. This allowed for evaluation of the fitness function to shade just a single triangle, and more importantly, to evaluate the fitness only looking at the pixels covered by the triangle, rather than the entire image. We found PSO to be extremely successful at this smaller optimization problem, and had a great speedup. However, in some cases, the PSO found a position for the triangle that worsened the global fitness. Handling these positions in the total solution constituted a second optimization problem. We implemented a simulated annealing like method, allowing solutions to worsen within a certain percentage of the best observed fitness. We found that the threshold for allowable fitness regression was an extremely important parameter. It needed to be a fixed percentage based on the inverse number of triangles. Having a threshold allows the program to shuffle more triangles around, reducing the number of failed triangle optimizations. However, if the regression is too large, then the program will take a long time to return/surpass the best fitness. Figuring out parameters like this algorithmically was one of our greatest challenges with this project, especially since because we are working with such large problem spaces, it was very difficult to have meaningful results until the program had run for a fairly significant amount of time.

On the CUDA side of things, we actually found optimization less important, because algorithmic optimization yielded so many results. Our basic model was one block per particle on PSO. Each block contained 16x16 threads. The bulk of the thread workload was shading pixels, and this was done on a 16x16 chunk of the triangle at a time. We took advantage of our image format and utilized texture memory to store and access our image files, and used shared memory to optimize access times. The typical PSO run required about 1000 iterations. Rather than launching a kernel 1000 times from PyCuda, we moved this loop into the kernel. The problem created here was that the global updates necessary after each iteration couldn't be done simultaneously, because blocks are asynchronous. We also had a kernel that shading all the triangles fresh at

every step, However, we found that asynchronous updates didn't result in big reduction in convergence, but saved a huge overhead in kernel launching. For the 100x100 test image, this resulted in a 20x speedup! However, on large images, this speedup was much smaller, since the kernel launching overhead was no longer a bottleneck.

Providing a performance analysis is particularly challenging for our project because, to our knowledge, no one has attempted a CPU version of this variation on the algorithm before. We found that our algorithm could produce outstanding results in a few number of iterations (2-5 x the number of triangles), however, to achieve exceptional results, our program required costly parameter setting that slowed down the initial iterations. Our algorithm basically uses parallelism to produce exceptionally optimized genetic mutations, such that each mutation is almost a guaranteed improvement. However, the parallelism of the GPU was easily exhausted in this costly step. It wasn't clear that the algorithmic change was holistically better. Roger Asling claims that his latest software can produce a similarly rendered image of the 200x200 Mona Lisa in ~90seconds using about 100 9 sided polygons. Using 500 triangles, we can render the same image in ~90 seconds also, and we subjectively judged our images to be better visually.



The above image shows the process of 2000 iterations with 500 triangles on the same Mona Lisa image Alsing uses. We've included other test images we ran in our tar folder

While we would have liked to have a clear performance advantage, we were happy with our results because: a) We implemented an entirely new technique. b) We achieve similar performance. c) Others have tried to implement the GA Evolisa in CUDA, and could not outperform the Alsing's multithreaded implementation.

Challenges and Future

Essentially, our greatest challenge was racing the clock when testing our program. Especially because we are writing EvoLisa and PSO from scratch, it was literally impossible to write a non-parallelized version because we had to sit and wait for hours to see if it was even working properly. Many times, because we are working with an image related problem, even when the numbers showed that it was probably working (given a decreasing best), the output images were inconclusive or qualitatively bad. Even with CUDA optimizations and significant speedups in performance, it still took a significant amount of time to simply run tests of our program.

Because of time constraints, there are many expansions of this program/algorithm that we didn't have a change to explore. From an optimization standpoint, we found better performance of the algorithm by limiting the maximum threshold of alpha value, i.e., so that it would take at the minimum 10 triangles to produce a pure white. This encouraged triangle overlap, which help the optimization. We ought to have made this a dynamic parameter, as at the later stages optimization this constraint grew unnecessary.

We also think that our triangle shader, an essential part of the program and the inner loop, could have been optimized more. However, we found that our time was better spent on higher level algorithmic optimization, where we achieved great progress, rather than low level hacks.
There are also many improvements that can be made on a features level. One simple improvement would allow the user to specify the complexity of the polygons used to render the image and to test the accuracy of the image at different levels. Similarly, the program could use attempt to use ovals or blobs and see the effect of non-polygonal shapes/curved lines to increase accuracy. Another improvement would be to create a GUI interface for the project, so that the user might be able to specify and highlight areas of importance (such as faces) with weights so that the algorithm would focus more on those areas of foreground importance rather than be distracted by noisy background areas. We could also add another level of parallelization and attempt the PSO/Evolisa algorithm on video footage, where we could parallelize the frames, using neighboring frames as starting points. Because the technique of creating realistic unrasterized and scalable representations of images is such a cool and relevant problem, the possibilities for expansion of this algorithm are exciting and numerous.

README

1) Upload the entire robb_joyding_evolisa.tgz onto SEAS and untar. The tgz contains the python modules PIL (python image library) and jinja2 (templating). They have been compiled for resonance, and the extracted tgz should work. If something goes from, you may need to actually install the two modules yourself on whatever system you are using, and delete the PIL and Jinja2 folders.
2) Log in to resonance
3) Load PyCuda modules: module load packages/pycuda/0.93; module load packages/cuda/2.2
4) Run with command: python evo.py imagename.jpg. No parameters are available by command line, but easily editable in the python script. The resultant images (one ever 100 iterations) will be outputted to ./temp_images. The stdout gives the fitness of the solution after every iteration.
We have provided a test image small and large test image:
python evo.py slisa.jpg for small image
python evo.py lisa.jpg for large image
The small image will render to decent quality in a minute or so, the large image takes about 10-20 minutes.