Massively Parallel Non-Convex Optimization on the GPU Through the Graphics Pipeline

By Peter Cottle

Department of Mechanical Engineering, 6195 Etcheverry Hall, University of California, Berkeley, CA 94720-1740, USA

Abstract: This project presents a way to optimize non-convex functions in N-Dimensional space through a series of operations that flow through the graphics pipeline, allowing the algorithm to be implemented in WebGL and other common graphics frameworks.

Keywords: Non-convex Optimization GPU Parallel Computing WebGL

Introduction:

In many areas of engineering, applied mathematics, machine learning, and the sciences, the global minimum of a particular function is desired. This minimum usually represents the "best" solution to some kind of simulation or analysis. This minimum may be the perfect combination of materials for a given engineering application, the perfect price point and selling volume for a market analysis, or the solution to a least-squares error function.

Whatever the application, this minimum represents the perfect combination of the search variables to obtain the given objective in the best way possible. "Optimization" is the process of finding the values of the search variables to combine to obtain this global minimum. The key difficulty with the optimization considered in this project is that the shape of the function is non-convex.

In order to understand this difficulty, we must first define what "convex" means. In layman's terms, a convex function is a function that is very smooth and does not have local minima (only one global minimum). Imagine a smooth bowl-shaped function as depicted in Figure 1.

This function has low curvature, is very smooth, and has one local minima (the global minimum). For these types of surfaces, we can use gradient methods to solve for the minimum. A gradient method uses information about the slope at a given point to estimate which direction to search in.

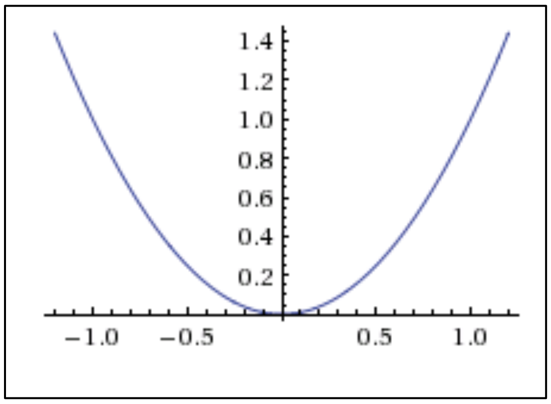

Figure 1: Smooth function y=x*x

For example in Figure 1 function, if we started our search at X = 4 we would calculate the slope at that point to be 8. This slope value means that the function increases with increasing X, so our next search point should have a lower value of X. We might search at X = -1 next, see that the slope is decreasing at this point, and then move to a value of x with a higher value. Eventually after an iterative process we would arrive at X = 0. At X = 0, we find that the slope is 0 (and the second derivative is positive), so we know we have arrived at the global minimum.

This is a simple gradient-based method. Examples of gradient methods include Newton's method, Gradient descent, Conjugate Gradient, and many others. All gradient methods share the same weakness though; they use the slope at a given position to determine the direction to move in. Furthermore, they also use the slope to determine when to terminate the algorithm.

These gradient methods break down when applied to non-convex functions. Imagine the following non-convex function in Figure 2:

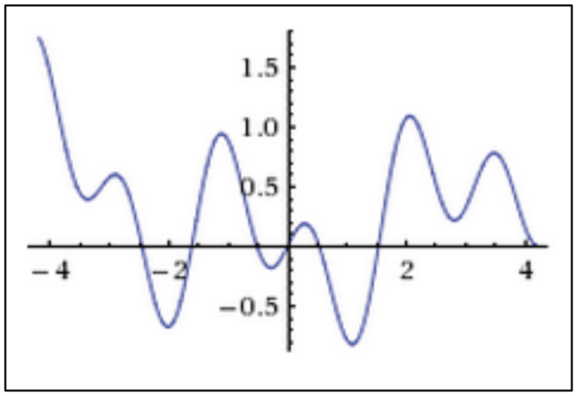

Figure 2: A non-convex function with many local optima

Gradient-based methods have trouble optimizing a function like this for two reasons. First, non-convex functions usually have very high curvature (or even discontinuities) and thus the slope at any one point is rarely indicative of the direction to move in for the global minimum. In the above function, the global minimum is located at X=1 but there are many regions with negative slope above X=1 (and vice versa for below). This means that gradient-based methods will proceed in the wrong direction depending on the initial starting point. Another difficulty is that there are many local minima, so gradient methods often falsely terminate in local minimums rather than "breaking out" of local minimums to search in other areas. In the above function, gradient-based methods may falsely terminate around X=3, -0.5, -2, and -3.

Thus to optimize non-convex functions, we must look towards other types of algorithms. There are many algorithms that perform this type of non-convex optimization – intelligent water drops, genetic algorithms, and ant-routing are just a few examples of these "non-convex" methods. All of these algorithms essentially break down into a few key parts: random search, intelligent selection of new search areas, and the "breeding" of solutions.

These algorithms perform fairly well in practice, but with tough functions they can sometimes take a long time to converge. Furthermore, they do not take advantage of the parallel-computing power available in modern machines. In this project we present a new method of non-convex optimization using the Graphics Processing Unit (GPU) and real-time networking across the internet. This algorithm is fully implemented, runs in a modern web browser with no special hardware, and performs exceedingly quick optimization on a variety of functions.

Objectives:

The main objective in this project is to obtain a working, implemented algorithm that performs non-convex optimization on the GPU with an added layer of network parallelism. This main objective is broken down into smaller objectives:

A) Obtain a framework of Javascript, HTML, and GLSL shader templates that can sample any N-dimensional mathematical function over pre-defined bounds and project these samples onto a two dimensional surface.
B) Obtain a shader program that transforms this 2D sample surface in a 3D surface where the z coordinate corresponds to the fitness of the sample point. The shader program will then project this 3D surface.
C) Implement an algorithm to scan these projected samples and extract out the position of the minimum.
D) Obtain an abstraction layer of real-time network communication to divide up the search space and dynamically change the input function as network commands come in.
E) Wrap the entire application into a user-friendly graphical interface.
F) Time the performance of this application in comparison to similar algorithms in Matlab.

Background:

Overview of GPU Architecture

In order to understand the overall implementation of this project, a background on computing on the GPU (in the WebGL framework) must be given first.

WebGL is a technology for executing compiled code on the GPU from a web browser. This is the first time that web developers have had direct, low-level access to powerful hardware directly from the browser; consequently, this presents a number of interesting new opportunities for web development. WebGL was created with the intention of being used primarily for computer graphics, so the only access a web developer has to the GPU is through the “graphics pipeline” – a series of graphics operations and concepts.

The overall architecture of the graphics pipeline is comprised of the ideas of vertices, vertex shaders, fragments, and fragment shaders. Vertices are essentially “groups” of data values, each data value being an individual entry from a larger array:

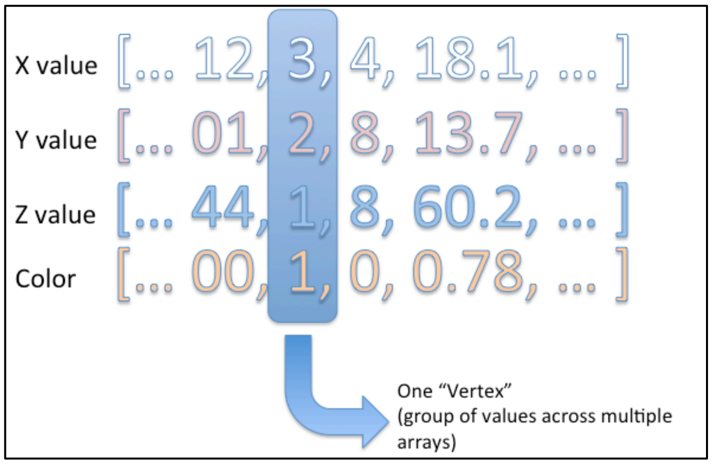
X value [... 12, 3, 4, 18.1, ...]
Y value [... 01, 2, 8, 13.7, ...]
Z value [... 44, 1, 8, 60.2, ...]
Color [... 00, 1, 0, 0.78, ...]
One “Vertex” (group of values across multiple arrays)

Figure 3: Vertex definition

Typically these groups of data values describe the geometry, shading, color, or orientation of the object to be displayed. A typical vertex may include the X,Y,Z location of a point, a color to shade with, and a surface normal for realistic lighting. It is important to note though that these attributes are not pre-defined; a vertex may contain any number of attributes representing any type of data value. We will use this flexibility later in the project.

These vertices (aka “groups” of values) are then passed through a vertex shader. A vertex shader is a small program (sometimes called a kernel) that takes in an individual vertex and outputs the position and color of that vertex on the screen. These vertex shaders also have access to “uniform” attributes on the GPU; a uniform attribute is a variable is the same across all vertex shaders.

A typical vertex shader might take in an individual vertex, look up the global projection matrix for the scene (a uniform attribute), perform the matrix projection, and output the position of the vertex on the screen. This process is depicted in the following figure:

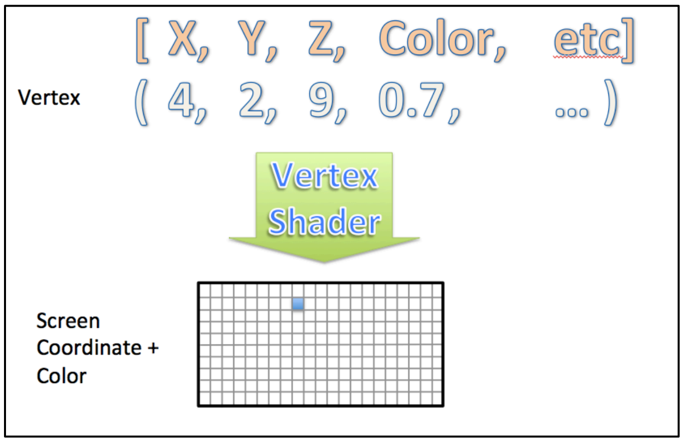

Figure 4: Vertex shader operation

This entire operation is only a few lines of code in GLSL because of GLSL's great native linear algebra operation support.

These vertices are almost always processed in "batches" that represent some geometric primitive. The most common geometric primitive for graphics is a simple triangle – three vertices are processed together, and each vertex outputs a position on the screen. The GPU now knows that it should fill, or "shade," all the pixels in-between these three pixel coordinates to fully draw the triangle. This process is depicted in the following figure:

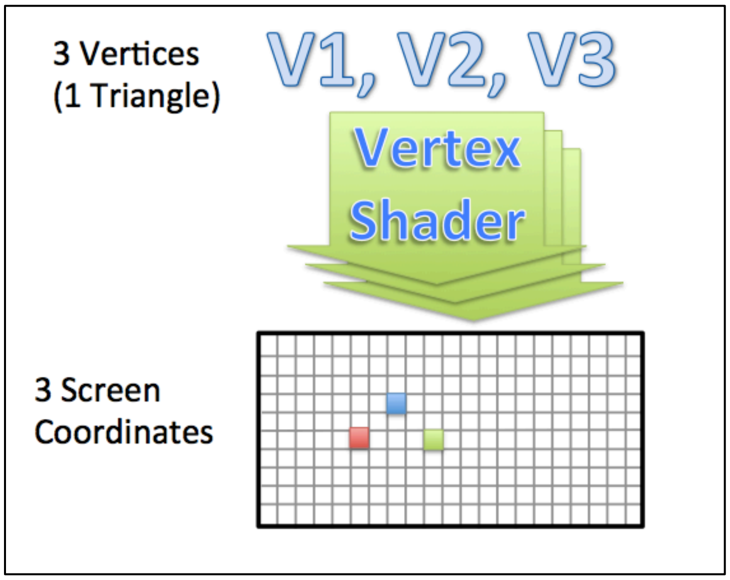

Figure 5: Batch vertex processing for primitives

The problem is that the GPU does not know necessarily how to shade these intermediate pixels. The three vertex shaders may have outputted completely different colors, so it may not be obvious how to shade intermediate pixels. This is the reason why "fragment" (pixel fragment) shaders exist. The GPU runs fragment shaders in order to determine how to shade the intermediate pixels between vertices.

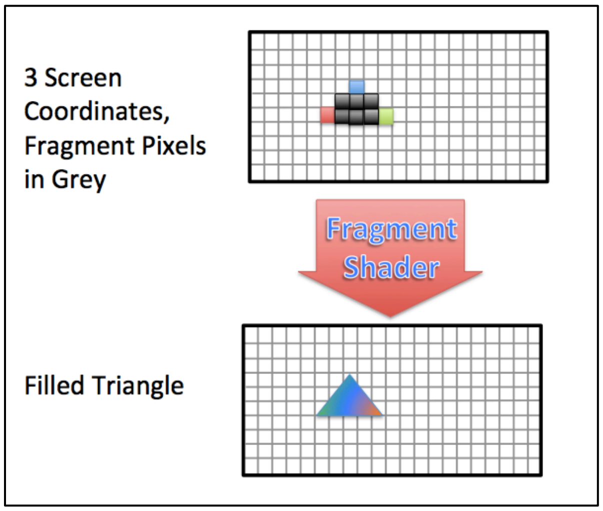

Figure 6: Fragment Shading on Fragment Pixels

The details of this process will not be described here, but fragment shaders usually interpolate between colors and output a final RGB.

DIFFICULTIES WITH GPGPU ON THE WEBGL FRAMEWORK

Now that the architecture of WebGL has been described, the difficulties of general purpose GPU computation (GPGPU) with WebGL can be discussed. The WebGL graphics pipeline is heavily optimized to output colors on a 2D screen; in fact, every operation in WebGL essentially terminates with outputting a series of red, green, and blue values onto a framebuffer. This rigid structure contrasts greatly with flexible frameworks like CUDA or Matlab's GPU toolkit; in these frameworks, developers can write kernel programs that output data in almost any structure.

Because of this WebGL restriction, we must essentially re-implement non-convex optimization in a way that flows through the graphics pipeline. This somewhat clever method is presented in the follow section.

Procedure / Implementation:

Our strategy, from a "10,000 ft view," will to be essentially "encode" the results of the calculations we want into the colors of the outputted image. We can then scan that image in a certain way, find a desired pixel, read out the colors from that pixel, and then convert those colors into the results of our calculations for the CPU to digest. This division between GPU work and CPU work is shown in the following figure:

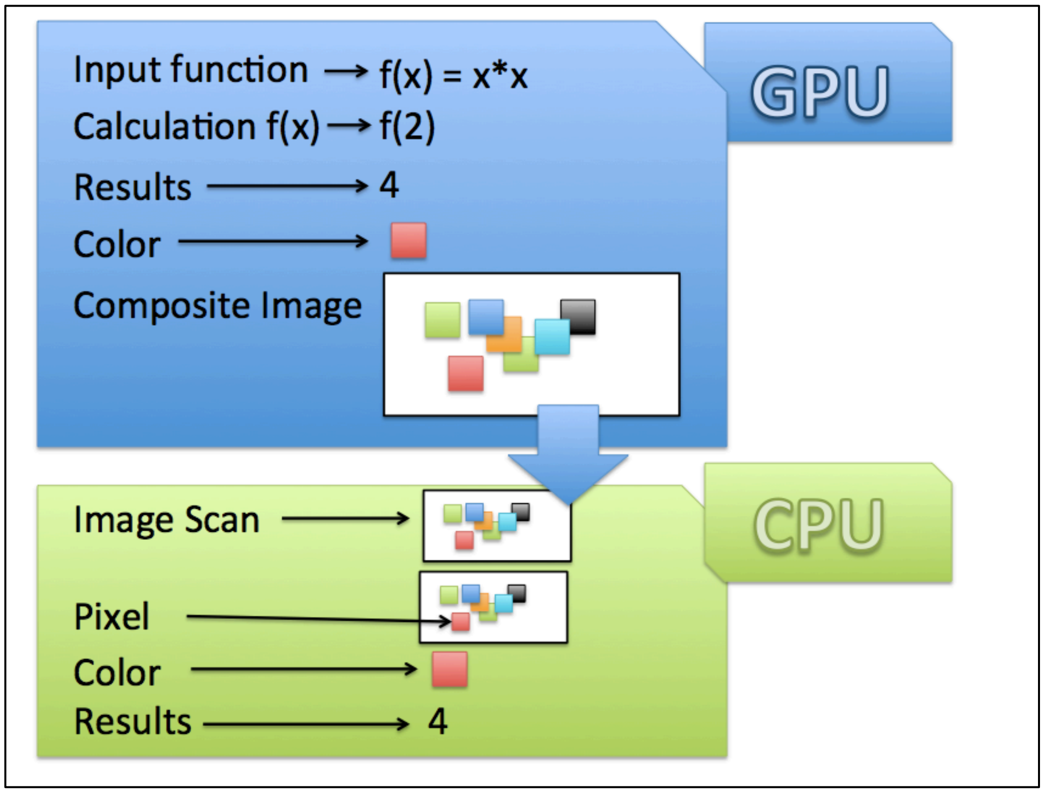

Figure 7: Overall Architecture of GPU-CPU interactions and computational flow

This entire workflow terminates with converting the color of a pixel into a final mathematical result. There are only three color-channels of a pixel: red, green, and blue. Each one of these channels corresponds to one output from our calculation; consequently, three outputs can be achieved each time this workflow is run.

If more than three outputs from our optimization are desired though, say for a 5-dimensional input function, we will have to run this entire process multiple times to obtain all of our outputs. The way this is implemented is to compile multiple versions of the same shader. Each shader is computationally identical and obtains the same results every time; the only difference from shader to shader is the selection of variables to output.

Say our input function is over variables E, D, F, G, and H. The first shader may output variables E, D, and F into color-channels red, green, and blue. The second shader will perform the exact same computation and simply output variables G and H into color-channels red and green. These shader programs are deterministic and obtain the same results every time before the data output step. With a collection of nearly identical shader programs, any number of outputs can be achieved from this graphics pipeline, despite each shader only being able to output three values.

Performing the same computation multiple times introduces redundancy in calculation, but this redundancy is trivial compared to the great leaps in performance achieved by the parallelism of our approach. This implementation detail is irrelevant to all of the proceeding sections but must be presented as a way to initially overcome the limited output abilities of the graphics pipeline.

SAMPLING AND 3D SURFACE CONSTRUCTION

Optimizing a function essentially reduces down to sampling over the "search space" and finding the global minimum. This "search space" consists of the all the possible values for all the sample variables. For a function of one variable, this search space is just a number line extending from the lower bound to the upper bound. If another variable is added, this line becomes a 2 dimensional surface between the variable bounds. A third variable transforms the search space into a cube, and a fourth variable and above transforms the search space into a hypercube (or N-dimensional cube). The search space for an input function of variables x and y is presented below:

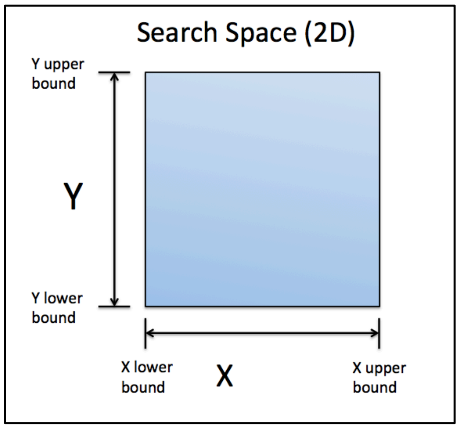

Figure 8: Search space for a 2D input function

This search space must be sampled in some fashion to order to evaluate individual points in the search space for fitness. In this project we choose two distinct yet closely related search methods: uniform sampling and random sampling.

Before we discuss sampling methods, we first must discuss what these sample points will be used for. The goal of this sampling is to eventually construct a 3D surface where the height of a given vertex corresponds to its "fitness" as defined by the input function. This process is shown below for an individual sample point:

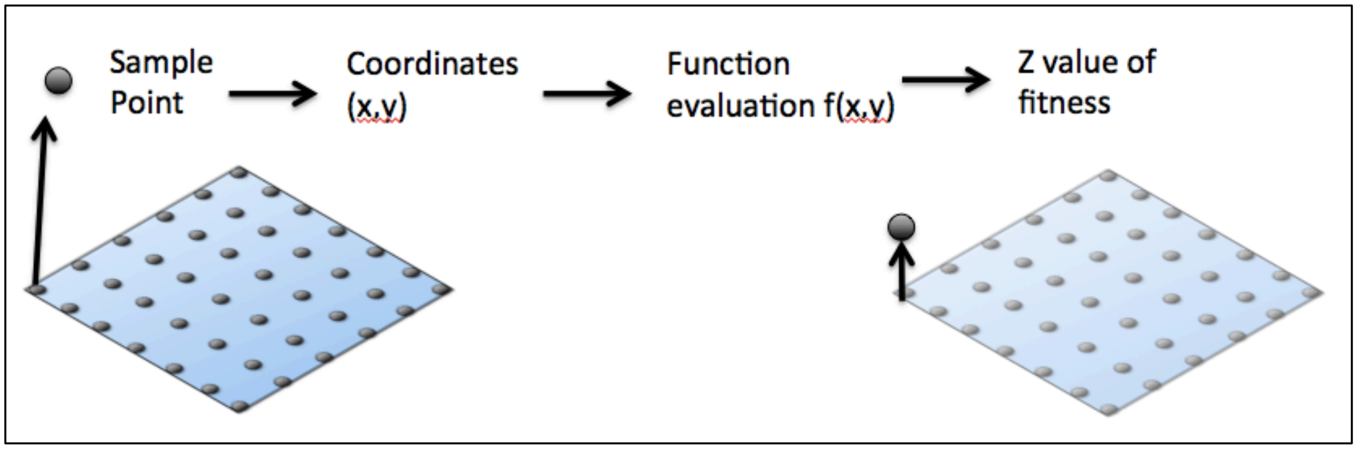

Figure 9: Sample point evaluation for Z coordinate

When performed in aggregate across samples that are evenly distributed, the actual shape of the function can be reconstructed:

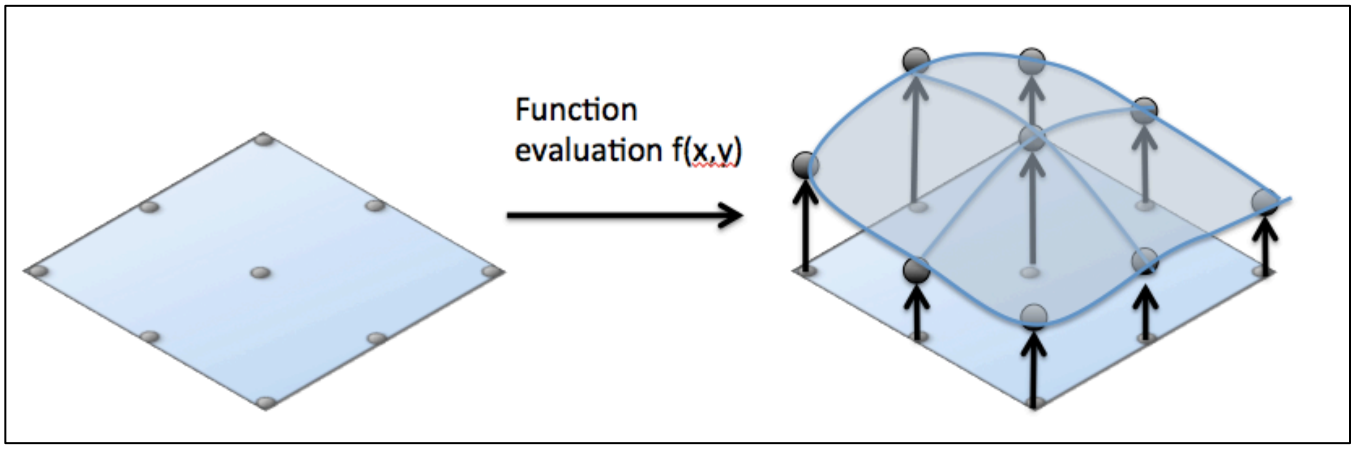

Figure 10: Aggregate function evaluate to construct 3D surface

This 3D surface will then be projected in such a way where the minimum z value can be easily extracted. The important aspect of this process is that GPUs are optimized to project 3D primitives onto a 2D surface for display (the render buffer). Thus, no matter what the dimension of our input function, we will *always* obtain a 3D surface that will then be projected for minimum extraction. The function evaluation essentially transforms a 2D surface into a 3D surface (or manifold). Thus, even if our search space is a 5 dimensional hypercube, we must obtain a 2D projection of the search space before doing the function evaluation step.

With this in mind we can now discuss the sampling methods used. A uniform sampling has vertices evenly distributed across the search space:

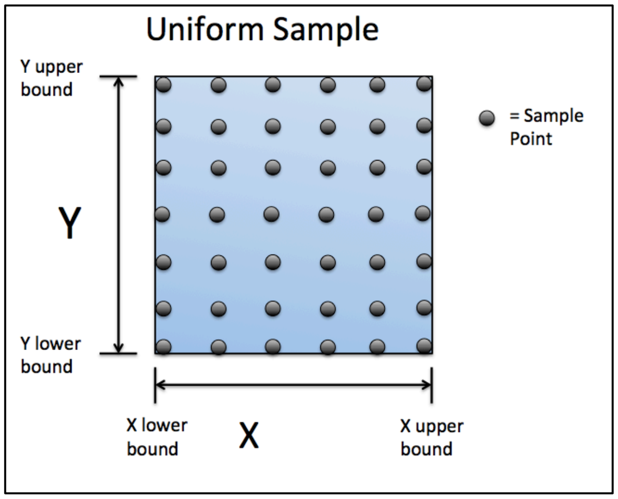

Figure 11: Uniform Sampling in 2D

This uniform distribution of sample points is actually how the GPU internally stores the vertices that are processed through the vertex shader. In our implementation, a 70x70 grid of sample points is buffered onto the GPU, with the bounds extending from -1 to 1 in each dimension. When this grid of sample points is processed through a vertex shader, the grid is dynamically re-scaled and shifted to match the appropriate bounds for the input variables. This dynamic scaling and shifting of the sample grid allows us to quickly change the search bounds or "search window" on the GPU without recompiling the program. This will become very important later during the solving process.

When this 2D uniform sampling is evaluated for a 2D input function, it constructs a very nice looking representation of the input function in 3D space. As an extra optional step, one can choose to map the z value to a hue in order to construct a graphical plot of the input function:

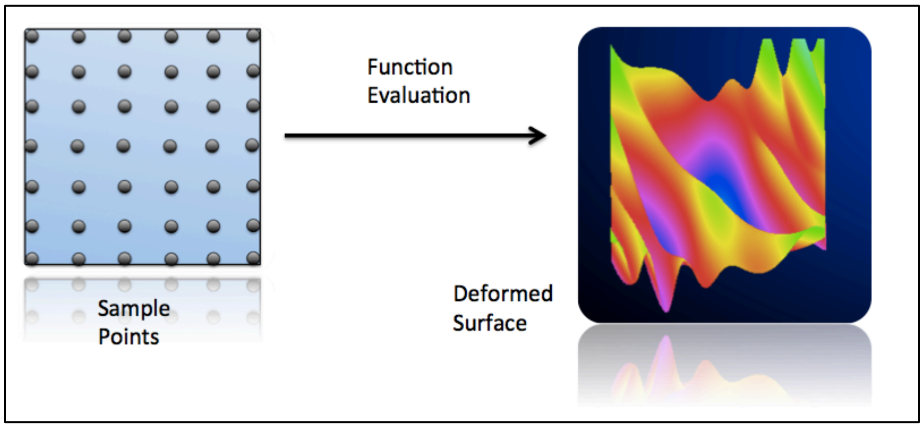

Figure 12: 2D input function to 3D surface

This 2D to 3D transformation due to function evaluation is very intuitive and performed very efficiently on the GPU.

Random Sampling in N-Dimensions

Since the sample points are stored internally on the GPU as a set of vertices distributed across 2D space, it is quite easy to uniformly sample a 2D search space. However when the search space becomes three dimensional or beyond, a 2D surface is no longer adequate at sampling. It is possible to buffer a 3D cube of points onto the GPU (or a N-D hypercube), but each added dimension of the search space would result in an exponential increase of the amount of data on the GPU. Ideally we would want a way for a 2D surface of vertices to sample from a search space of any dimension.

In order to do this, we switch to random sampling in N dimensions. Each vertex in the grid disregards its actual coordinates in 2D space and instead randomly assigns a value for each coordinate based on pseudo-random number generation.

To do this, a pseudo-random number generator had to be implemented on the GPU. When each vertex calls to the pseudo-random number generator, it passes in a "seed" for the number generation. This seed consists of the current time, the vertex coordinates, and the search variable index. By combining all of these characteristics for the seed, we essentially obtain a unique seed for each vertex for each sample variable in each moment of time. This allows us to evenly "distribute" our samples across N-D space.

An example of 2D random search is shown below:

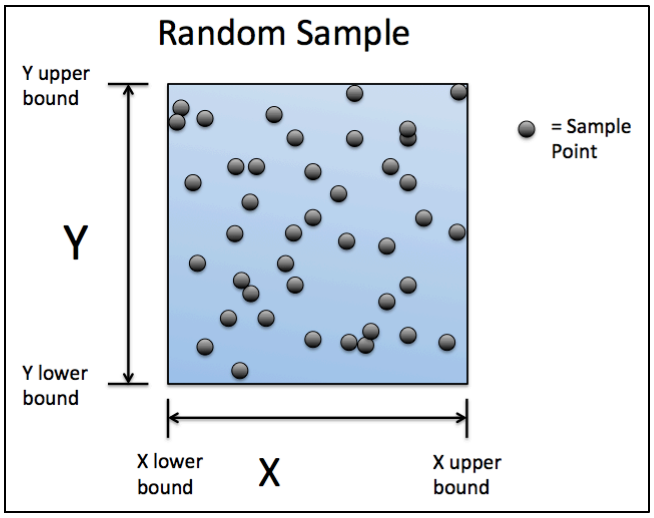

Figure 13: 2D random sampling

As one can see, each vertex is now randomly positioned in the 2D search space. The benefits of this process are not clear until we look at random sampling in a 3D search space:

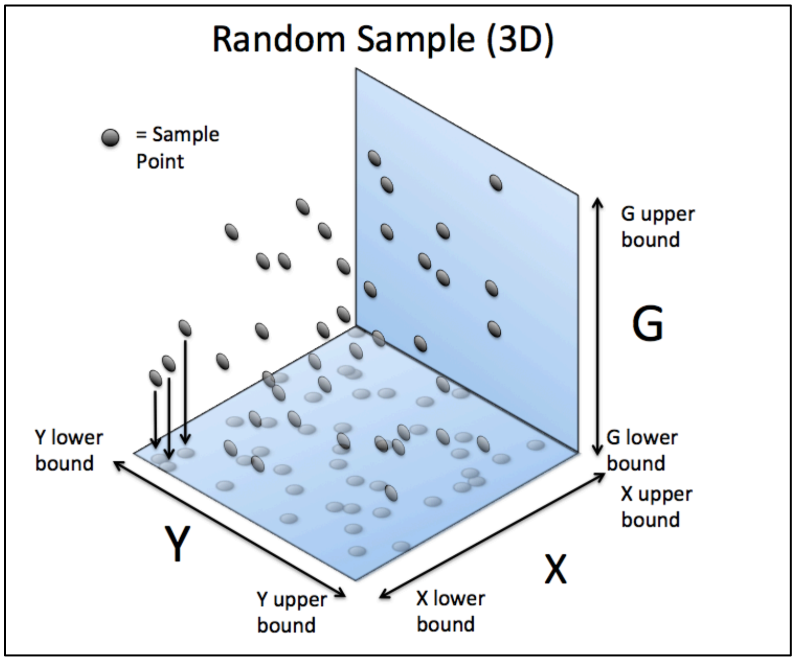

Figure 14: Random Sampling in 3D

Here we can see how a 2D distribution of points can now randomly sample from a 3D search space. This process can extend to any number of dimensions and allows our algorithm to expand to many more functions.

It is important to note that since each vertex will be randomly distributed in the search space, the constructed 3D surface will no longer be a nice visual representation of the input function. This essentially means that **the surface geometry in 3D space is now meaningless** except for the Z coordinate (which corresponds to the fitness). An example of this surface can be seen in the following figure:

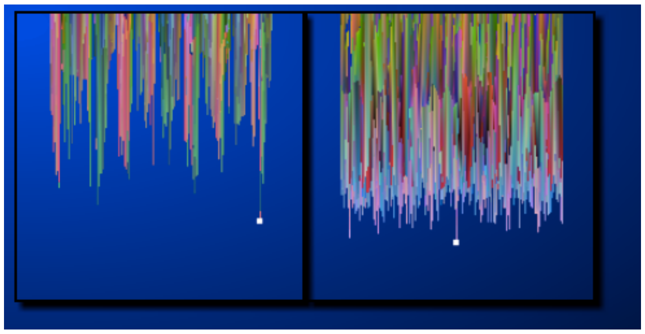

Figure 15: Random Sampling Surface Projection for two different search windows

The way we make use of this random noise is to encode the position of each vertex in the color of the outputted pixels. In the above figure, each pixel has a red, green, and blue value that corresponds to its position in X, Y, and G space. *Thus we can essentially sample from an N-dimensional space, project these samples onto a 2D surface, evaluate each sample for its fitness, encode that fitness in the z coordinate displacement, project the surface, find the minimum pixel, and then reconstruct the coordinates of the minimum sample by looking at the colors of the pixel. This is the heart of the algorithm presented in this project.*

Projection of 3D surface

We have now successfully sampled the input search space and constructed a 3D surface where the z coordinate corresponds to the fitness of each sample point. We would now like to find the sample point that has the lowest fitness value.

In order to do this, we will use an orthographic projection matrix to project the sample surface onto the render buffer in 2D space. We choose to orient the surface in a particular manner such that it is viewed from the side with the z-axis vertical in the image. Since the z-axis corresponds to "fitness" of the calculation, and since we choose to look for the pixel with the lowest fitness, we can simply scan upwards from the bottom of the image until we hit an individual pixel. This will be, by definition, the pixel with the minimum z value. The colors of this pixel will correspond to the location of the sample point in the search space:

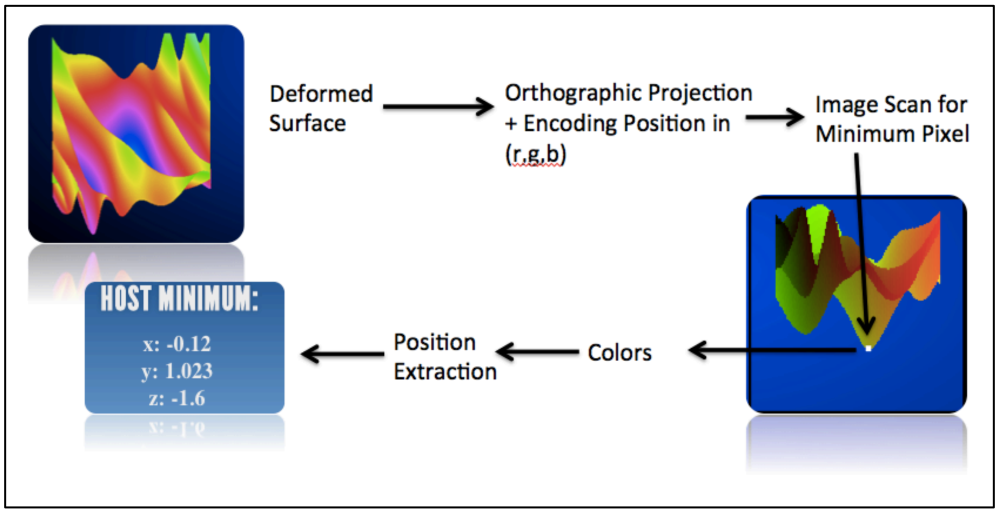
Deformed Surface
Orthographic Projection + Encoding Position in (r,g,b)
Image Scan for Minimum Pixel
Colors
Position Extraction
HOST MINIMUM:
x: -0.12
y: 1.023
z: -1.6

Figure 16: Overall computation flow for minimum extraction

The surface projection step is performed on the GPU and stored into a temporary render buffer. This render buffer is then scanned by the CPU in an upwards fashion until it hits a pixel with a non-zero alpha value. This scanning process has a worst-case runtime of O(n), where n is the number of pixels in the image. For added efficiency, one can choose to construct the render buffers such that they have a width of one pixel. This essentially maps all horizontal points (with the same fitness value) to the same pixel, allowing for a more efficient scan of the output render buffer. This extra step was coded in our implementation, but it was found that it made a trivial performance difference. Furthermore, it was far less intuitive to new users what the render buffer was doing; thus, this feature was not implemented in the final version.

It is also important to note that the GPU has a built-in z-buffer that can easily sort thousands of points and present the one with the minimum z value. For added efficiency, one could instead project all the sample points to the same pixel in space and let the GPU do the extraction of the global minimum (rather than the CPU). It is hypothesized that this would not make a large performance difference and would prevent intuitive understanding of the frame buffer.

Breeding or Mutating of Solutions

Now that we have a global minimum from our sample points, we can then perform a step of "breeding" on this solution to obtain more sample points. These other sample points can then be evaluated for a more accurate and possibly better global minimum.

The way we "breed" these solutions is by constructing a scaled version of the search space, centered at the location of the current "best" solution found by the initial solving pass. In our implementation we scale the search space to 5% of its original bounds. This smaller search space can then be sampled once again to look in the nearby "neighborhood" search space of the current solution. This sampling can either be uniform 2D or random N-D sampling. This additional step of breeding usually immensely improves the accuracy and value of the final solution.

Division of Search Space

Finally, this entire process can be performed in parallel across multiple machines for a multi-parallelism approach. Our implementation chose to use "room" IDs to represent a group of workers for a particular problem. When a new user visits the application home page (without a room ID in the URL), a new "room" is created and the room ID is appended to the URL. The user can then share this URL with other users to have them join their particular room. When another user visits a link with an embedded room ID in the URL, the application loads the room information and redistributes the search space among the existing workers.

The distribution of search space is a simple one-dimensional partitioning algorithm:

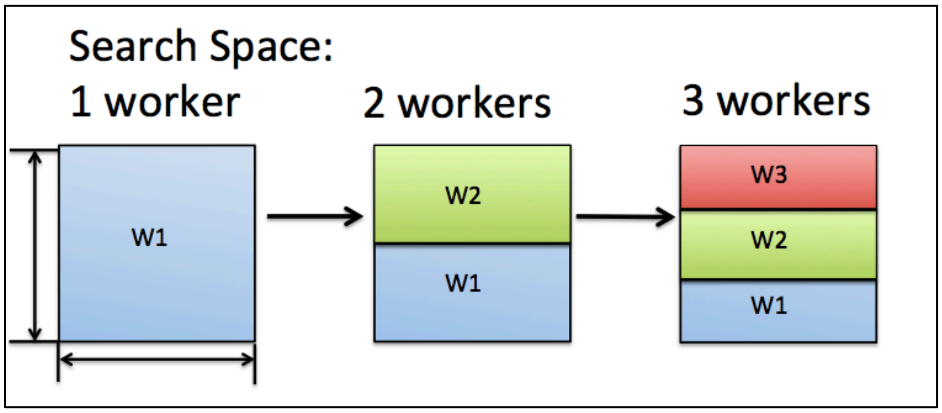

Figure 17: Division of Search Space

And thus only enhances the accuracy of the solutions found or speeds up the convergence of random search. One important note is that if enough workers are distributed along one dimension, the range of that dimension for each worker will approach a very small value. At a certain point, an entire dimension of the search space can be eliminated:

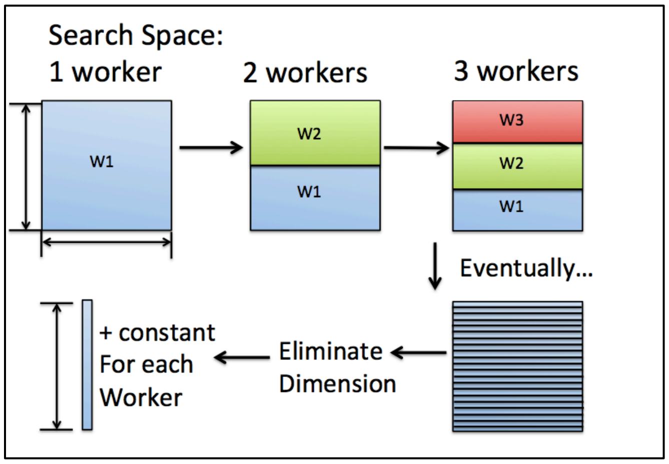

Figure 18: Search Space Dimension Elimination from added workers

In practice, the parallelism approach of the GPU overshadowed the linear gains in parallelism by adding a few individual workers. It is hypothesized that adding an immense number of workers could dramatically improve the convergence of the algorithm, but adding just a few shows no considerable difference.

Conclusions:

In order to compare the speed of this algorithm against similar methods, a genetic algorithm Matlab script was constructed and timed in its optimization of the "Vanilla (static)" function from the web application. The timings for 10 trials are below along with timings from the web application:

Timings for Maltab Genetic Algorithm (Mean of 0.1944 Std Dev of 0.0278)									
0.185866	0.177614	0.208840	0.192452	0.210299	0.200843	0.229317	0.232592	0.150189	0.156252

Timings for Genetic GPU Algorithm (Mean of 0.0035 Std Dev of 7.8571e-04)									
0.0033	0.0049	0.0043	0.0026	0.0042	0.0037	0.0037	0.0026	0.0027	0.0032

It is important to note that the web application also presented a 3D plot of the surface, responded to UI events, updated the HTML DOM representation of the page, and displayed the render buffers on the HTML page with sub-optimal canvas tags. If these functions were stripped and the computation was purely mathematical (like the Matlab script), there might be even more performance improvement.

That being said, the web application is about ~55x faster than the Matlab script – a considerable improvement.

Future Work:

In this application we are performing the optimization of functions that can be expressed as one mathematical expression. The functions that are optimized in industry and research, however, are usually much more complicated than a single mathematical expression. The cost function for common genetic algorithms could be anything from a course FEM simulation to an entire fluid dynamics simulation. These much more complicated fitness functions would need to be implemented on the GPU for the existing application to work. This represents a considerable programming challenge to new users unfamiliar with GLSL; furthermore, GLSL is somewhat limited in terms of the data storage available to each kernel. If the fitness function was too complex and required too much data, the current application may not be able to optimize those types of input functions.

It would be interesting to pair this web application with a GLSL converter that took in common programming languages (C, Matlab, Python, etc) and converted cost functions to GLSL. This would most likely serve a much wider market than the current application does.

Lastly, the search methods used in this implementation are quite simple. They essentially consist of a search across the entire space and then a smaller search in the neighborhood of the best solution. There may be considerable performance improvements if other search strategies were constructed. One idea is to take the convex hull of the top N solutions and use that as the new search space. Another idea is to randomly “jitter” the uniform grid in different directions in order to cover previously unseen areas of the function. If these more intelligent search algorithms are implemented, considerable performance gains may be obtained.

Appendix:

Application URL:

http://petercottle.com/GeneticGPU/index.html

Application Source Code:

https://github.com/pcottle/GeneticGPU