# Efficient GPU-based Optimization of Volume Meshes

Eric Shaffer [a,1], Zuofu Cheng [a], Raine Yeh [a], George Zagaris [b], and Luke Olson [a]
[a] *Department of Computer Science, University of Illinois at Urbana-Champaign*
[b] *Kitware Inc.*

**Abstract.** We describe a GPU-based algorithmic framework for optimizing the shape of elements in a simplicial volume. Optimization is done on a per-vertex basis using only local neighborhood information in order to exploit the massive fine-grained parallelism on modern GPU hardware. We propose and apply three optimization methods which have potential to be suited for local optimization of element shape and present a framework which may be generalized to other methods. Experiments which compare our method to state-of-the-art algorithms show a more than ten-fold performance increase for a similar final quality in both test and practical real-world meshes.

**Keywords.** Mesh optimization, parallel algorithms, GPU applications

## Introduction

Mesh quality is a key concern in engineering and scientific computing applications. The shape of mesh elements can significantly impact the efficiency and accuracy of simulation codes. In this paper, we consider the problem of improving element quality in unstructured tetrahedral meshes by adjusting the positions of the mesh vertices as an optimization problem. Specifically, we seek to reduce the maximum and average inverse mean ratio, which detects irregular and inverted simplex elements [1,2,3]. The results of our novel algorithm can be compared to existing software for volume mesh optimization such as Mesquite [4]

The wide availability of massively multi-threaded graphics processing units (GPUs) offers a new direction for the acceleration of mesh smoothing algorithms. We show how optimization can effectively be accomplished on a per-vertex basis on the GPU, exposing fine-grained parallelism in the same manner as Freitag et al. [5] did for more traditional parallel architectures. The framework of the algorithm can accommodate essentially any underlying numerical optimization method to compute the new vertex positions. In our experiments, we implemented three different optimization algorithms on the GPU: the gradient descent method, the BFGS (Broyden-Fletcher-Goldfarb-Shanno) method, and the derivative-free Nelder-Mead simplex method. We will present our performance comparison and analysis of the three methods, with the perhaps surprising result that Nelder-

[1]Eric Shaffer: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; E-mail: shaffer1@illinois.edu.

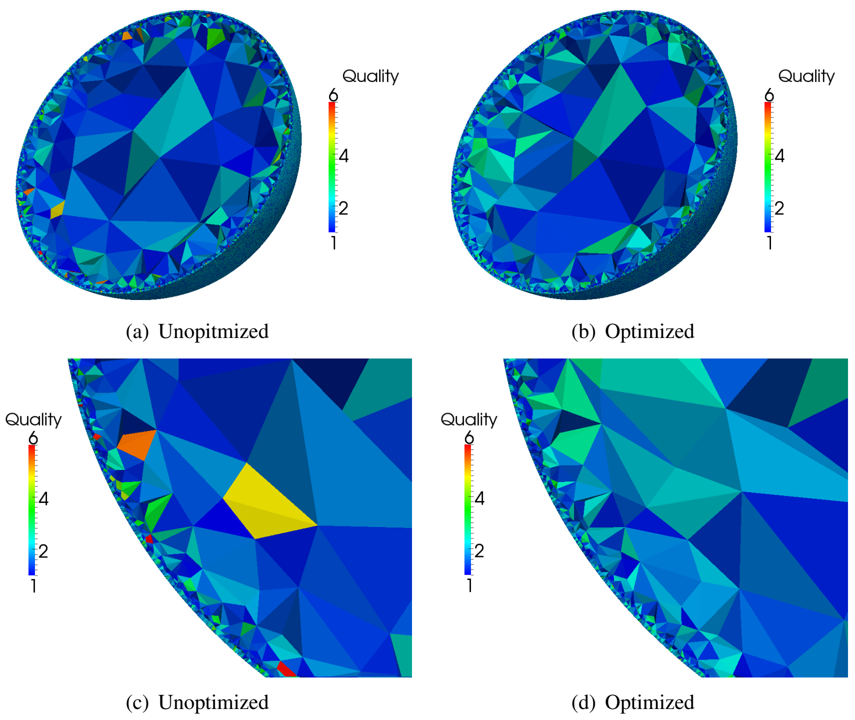

**Figure 1.** Optimization of the Max Inv. Mean Ratio Metric on the Small Sphere

Mead converged fastest. In addition to discussing raw performance numbers, we also examine the register usage and memory access patterns of the three methods and discuss a performance optimization strategy that, counter-intuitively, improves performance while reducing GPU occupancy.

In addition to possessing generality, the algorithmic framework we describe is simple to implement and fast. In our experiments, the GPU-based algorithm was shown to converge to a high-quality solution up to 59 times faster than the serial version of the algorithm based on local optimization. When compared to a state-of-the-art serial method for global mesh optimization, our GPU-based algorithm exhibited up to a 12-fold speedup while producing a solution of comparable quality. These results demonstrate the scalability and effectiveness of the GPU as a platform for volume mesh optimization.

## 1. Volume Mesh Optimization

Consider an unstructured mesh of $N$ elements and $M$ vertices. Let $e_n$ be the $n^{th}$ element, $v_m$ the $m^{th}$ vertex, where $n = 1,2,\cdots,N$ and $m = 1,2,\cdots,M$. For a tetrahedral volume mesh, the dimension of a mesh vertex is $d = 3$ and the number of vertices referenced by an element is $|e_n| = 4$.

## *1.1. The Optimization Problem*

We choose to measure the quality of a tetrahedron by the inverse mean ratio metric [6]. This metric computes the deviation of the element from an ideal element, which we choose to be an equilateral tetrahedron. A graphical example generated using the output produced by our system is shown in Figure 1. It is clear that the inverse mean ratio detects poorly formed elements. The metric is formulated as follows: given that the element has vertices $(a,b,c,d)$, we form matrix $A$ as a square matrix of the edges eminating from vertex $a$ and $W$ as square matrix representing the ideal element.

$$A = [b-a \quad c-a \quad d-a] \qquad\qquad W = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{\sqrt{3}}{6} \\ 0 & 0 & \frac{\sqrt{2}}{3} \end{bmatrix}$$

The inverse mean ratio is then given by:

$$\frac{||AW^{-1}||_F^2}{3|\det(AW^{-1})|^{\frac{2}{3}}} .$$

The values generated by the inverse mean ratio metric range from 1 to $\infty$ with 1 being the optimal value which is achieved when the element is an equilateral tetrahedron. In addition, the inverse mean ratio is invariant to rotation, reflection, and uniform scaling of the element; the choice of $a$ determines the sign of the determinant.

We combine the quality metric evaluations for all the elements including a vertex into the *objective function* for that vertex, which is the basis for optimization. We define this as the maximum of the inverse mean ratios of all neighboring elements, which is a non-smooth function since discontinuities can occur at points where the maximum inverse mean ratio value shifts from one element to another.

## *1.2. The Optimization Algorithms*

As a result, the second derivative is not guaranteed to exist, which complicates the use of gradient-type optimization methods, such as quasi-Newton methods. Because of these difficulties, much of the recent research has focused on avoiding direct differentiation of the objective function, either by using derivative-free optimization [7] or by various methods to increase the smoothness of the quality function [8]. The approach taken here resembles the latter, in that we approximate the gradient, which is equivalent to taking the gradient of locally smooth interpolants of the non-smooth data (see below). With this, we have implemented parallel versions of two derivative-based algorithms as well as a derivative-free algorithm to test our GPU-based parallel optimization system.

For the derivative-based methods, we use a gradient descent and the BFGS (Broyden-Fletcher-Goldfarb-Shanno) methods. The first because of simplicity of implementation and commonality with more complex derivative-based line search methods, and the second because it is faster in situations where computing the Hessian directly is expensive and because current research suggests robustness in non-smooth situations [9], which is consistent with the performance of Hessian-based methods in this domain [10]. For the derivative-free method, we have chosen to employ the Nelder-Mead Simplex method [11] which has been used for mesh improvement in a serial setting[7].

### *1.2.1. Nelder-Mead Method*

The Nelder-Mead method is widely used and is widely available in numerical software libraries. Nelder-Mead searches a domain for a point $p$ minimizing the objective function $Q(p)$. The algorithm starts by sampling function values on the corners of a non-degenerate simplex $S_p$ in $\mathscr{R}^N$, which are not the simplices of the mesh. The simplices $S_p$ are constructed on-the-fly, enabling the algorithm to explore the function domain in a structured way. Based on the relative values of the function at the sampled points, the simplex is transformed using a combination of expansion, reflection, and contraction operations. These operations move one or more corners of the simplex based on a step-size parameter. This process iterates until the step-size falls below a specified threshold or a given number of iterations have occurred. Ideally, the simplex flows to areas yielding lower function values and contracts around a minimum.

The reasons for the popularity of the Nelder-Mead algorithm are mostly practical. The method benefits from ease-of-implementation and the generality of the algorithm as the method applicable to the most general objective functions. Additionally, although there are no theoretical convergence guarantees in the general case, the approach is often convergent in practice. However, one significant disadvantage of the algorithm is that it requires a large number of objective function evaluations. In addition, optimizing a large number of variables exacerbates this shortcoming.

### *1.2.2. Gradient Descent and BFGS*

The gradient descent and BFGS methods are similar and rely on evaluating the derivative of a function to find a search direction followed by a line search. In the gradient descent, the direction is simply given by the negative of the gradient. In our scenario, in order to circumvent the lack of a well-defined derivative, we use a central difference approximation to numerically estimate the gradient. For the BFGS method, we must also take into account the second derivatives (specifically, a $3 \times 3$ approximate Hessian matrix in our case, which we denote $B_k$). The first step then is to solve for the corrected direction $\vec{p}_k$ given the gradient and the approximate Hessian using:

$$B_k \vec{p}_k = -\nabla f(\vec{x}_k). \tag{1}$$

A line search (also sampling based) is performed in the direction of $\vec{p}_k$ and the approximate Hessian is updated for the next iteration by evaluating:

$$B_{k+1} = B_k + \frac{\vec{y}_k \vec{y}_k^T}{\vec{y}_k^T \vec{s}_k} - \frac{B_k \vec{s}_k \vec{s}_k^T B_k}{\vec{s}_k^T B_k \vec{s}_k} \tag{2}$$

where $\vec{s}_k$ is a vector corresponding to the result of the line search in the direction of $\vec{p}_k$, and $\vec{y}_k$ is the change in the gradient of the new point relative to the old point. More precisely:

$$\vec{s}_k = \vec{x}_k - \vec{x}_{k-1} \tag{3}$$

and

$$\vec{y}_k = \nabla f(\vec{x}_{k+1}) - \nabla f(\vec{x}_k) \tag{4}$$

where $\vec{x}_k$ is the current best point of the $k^{th}$ iteration.

**Algorithm 1** GPU Mesh Optimization

```
for each vertex $v_i$ do
  construct the set $\mathbf{T}_i$ of all neighboring tetrahedra
  $c_i \Leftarrow$ FirstFit($v_i$) // assign $v_i$ a color
end for
for each color $k$ do
  for each vertex $v_i$ do
    if $c_i = k$ then
      add $v_i$ and $\mathbf{T_i}$ to set $\mathbf{S_k}$
      // $\mathbf{S_k}$ is an independent set of vertices and their neighboring elements
    end if
  end for
  Transfer $\mathbf{S_k}$ to the GPU
  for each $v_i \in S_k$ do
    $v_i \Leftarrow$ Optimize($v_i, \mathbf{T_i}$)
    // Optimize($v_i, \mathbf{T_i}$) performs Nelder-Mead, gradient descent, or BFGS
  end for
  Transfer all $v_i \in \mathbf{S_k}$ back to the CPU
end for
```

### *1.3. GPU Implementation*

The high-level steps in the GPU optimization algorithm can be summarized in Algorithm 1. In our implementation, each vertex of the mesh is assigned to a thread, which is mapped at runtime to execute in parallel on an available streaming processor (SP) on the GPU. One issue that manifests immediately is that the calculation of the quality metric for each vertex is not completely independent, as the objective function for a given vertex depends on the positions of its neighbors being constant. To work around this issue, we start by labeling all the vertices in such a way that no two neighbors have the same label using a First Fit algorithm [12] on the CPU where each label denotes an *independent set* of vertices. This allows us to optimize the set of vertices which have a given label simultaneously with the guarantee that no neighbors of the current working set are also being optimized. This also means that the overall optimization is not exactly the same between the serial and parallel versions, as the order of the optimization is not the same. This causes slight differences in final mesh quality between the CPU and GPU versions of each algorithm, as we will see later.

A key metric often used to guide GPU optimization is the number of threads in use divided by the total thread capability of the GPU, which is termed *occupancy*. This is important because the GPU has the ability to dynamically swap groups of threads during execution in order to avoid stalls due to high latency operations, such as I/O or main memory accesses. Because there are finite amounts of GPU resources such as registers, cache, and shared memory, the occupancy is typically a function of the program's requirements in each of these areas. Typically, having higher occupancy means the scheduler has more options to avoid a stall to maximize throughput and results in higher performance in applications where main memory access is the bottleneck.

For the gradient based methods, the occupancy at 42 percent is fairly good in comparison to the number of global memory accesses (which only consist of reading the initial vertex position, the neighborhood vertex positions, and writing the modified position). Unfortunately this is not true with the Nelder Mead method, which uses many more registers (63, with a spill store of 104 bytes), and hence has both lower occupancy (33 percent in our initial attempt), and causes some of the register accesses to convert to high latency global memory operations. In order to improve the performance of the Nelder Mead implementation, we use GPU shared memory as a manually managed cache space to store the neighborhood of the vertex being optimized. In addition, to allow for more shared memory to be used, we changed the cache configuration of the Fermi GPU to allow for 48KB of shared memory (at the cost of a smaller L1 cache) in the cases when this optimization is being used. Because this adds another constraint (shared memory), this in fact decreases the occupancy down to a minuscule 4 percent. However, because global memory access due to register spillage is eliminated, this counter-intuitively improves performance by up to 25 percent compared to our initial implementation, in line with performance gains seen by a similar optimization by Volkov et al in [13].

Unlike the Nelder-Mead GPU implementation, the derivative-based GPU algorithms use fewer registers, and therefore did not benefit from the shared memory optimization. Specifically, the gradient descent method uses 50 registers compared to the Nelder-Mead's 63, which is the maximum number of allowed registers per thread in CUDA and does not account for register spilling into slow local memory. This increases the occupancy to 42 percent for the gradient descent kernel. Interestingly, the BFGS GPU implementation also uses 63 registers like Nelder-Mead, but has a smaller stack frame, 88 bytes as compared to 104 bytes for Nelder-Mead. This indicates that while both the Nelder-Mead and the BFGS use the same number of registers, there is less register pressure in the BFGS implementation. This reduces the spills to local memory, and will reduce the impact of low occupancy even if the occupancy itself is not improved.

An observed performance bottleneck with the GPU version is the inefficient use of terminated threads because the GPU scheduler cannot reuse a terminated thread unless every thread within the block (in our case, 64 threads) has also terminated. This means that in the case of a mesh that has widely varying initial quality, many of the threads will be idle as they have already terminated. One possible solution is to reuse threads by allowing each thread to work on multiple vertices. However, this requires that the vertex assignment be dynamic, and therefore requires synchronization and communication between each thread, perhaps using shared memory to tally completed vertices. Effectively dealing with early thread termination is direction for future work.

## 2. Experimental Results

We performed preliminary experiments to evaluate the performance of our GPU implementation and compared it to both single-threaded serial versions of the same algorithm and current state-of-the-art global optimization methods. The hardware system used in our experiments contained an Intel Xeon X5650 CPU with 6 Cores supporting up to 12 threads clocked at 2.67GHz with 12MB of cache memory and 16GB of main memory. The GPU was a Tesla C2050 (Fermi architecture) with 448 Cores and 2.6GB of memory with ECC enabled. Experiments were done in Linux using code compiled by gcc 4.4.0 and the Nvidia CUDA toolkit version 4.2. Optimization was set to the highest level (for speed) in both gcc and nvcc, no assembly optimizations or SIMD intrinsics were used in the serial and parallel CPU versions (beyond those required to support OpenMP).

**Table 1.** Speedup of GPU Mesh Optimization over Serial Mesh Optimization

.

| Mesh | Vertices | Method | Quality GPU | Quality CPU | Time(s) GPU | Speedup |
|---|---|---|---|---|---|---|
| Small | | Nelder-Mead | 3.71 | 3.26 | 15 | 12 |
| Sphere | 332,995 | BFGS | 3.65 | 3.52 | 16 | 33 |
| | | Gradient | 3.61 | 3.60 | 18 | 43 |
| Big | | Nelder-Mead | 4.23 | 3.79 | 56 | 12 |
| Sphere | 1,339,317 | BFGS | 4.27 | 3.60 | 59 | 37 |
| | | Gradient | 4.27 | 3.72 | 67 | 43 |
| | | Nelder-Mead | 2.28 | 2.22 | 446 | 17 |
| Wing | 4,881,071 | BFGS | 2.42 | 2.22 | 470 | 44 |
| | | Gradient | 2.48 | 2.22 | 540 | 59 |

**Table 2.** Performance of GPU Nelder-Mead and CPU Feasible Newton

| Mesh | Number of Vertices | Number of Tets | Method | Converge Time(s) | Speedup | Inv. Mean Ratio | |
|---|---|---|---|---|---|---|---|
| | | | | | | max | avg |
| Big Sphere | 1,339,317 | 4,720,255 | unopt. | — | | 8.6 | 1.6 |
| | | | CPU-FN | 527 | | 5.5 | 1.47 |
| | | | GPU-NM | 67 | 7.9 | 4.72 | 1.45 |
| Big Rocket | 2,202,793 | 14,992,367 | unopt. | — | | 15 | 1.35 |
| | | | CPU-FN | 2336 | | 10.5 | 1.20 |
| | | | GPU-NM | 191 | 12 | 3.37 | 1.20 |
| Wing | 4,484,039 | 27,725,125 | unopt. | — | | 6.1 | 1.1 |
| | | | CPU-FN | 3808 | | 3.33 | 1.10 |
| | | | GPU-NM | 442 | 8.6 | 2.55 | 1.10 |

**Table 3.** Speedup of OpenMP based CPU Nelder-Mead

.

| Mesh | Vertices | Quality Serial | Quality OpenMP | Time(s) OpenMP | Speedup |
|---|---|---|---|---|---|
| Big Sphere | 1,339,317 | 3.789 | 3.788 | 190 | 3.811 |
| Big Rocket | 2,779,481 | 2.833 | 2.833 | 931 | 3.306 |
| Wing | 4,881,071 | 2.259 | 2.261 | 2067 | 3.835 |

### *2.1. Comparison to local serial optimization*

As shown in Table 1, computing on the GPU offers a significant speedup over the serial implementation of the same algorithm for all three core numerical algorithms. The performance improved by factor of up to 17 for Nelder-Mead, and up to a factor of 59 for

**Table 4.** Effect of Multiple GPU Passes on the Quality of the Small Sphere Mesh

| Method | Iterations per Pass | Passes | Time(s) | Max Inv. Mean Ratio |
|---|---|---|---|---|
| Local Iteration Only | 200 | 1 | 30 | 4.1 |
| Multiple Passes | 50 | 4 | 34 | 3.5 |

the derivative-based methods. There are multiple reasons for this discrepancy in GPU versus CPU performance between Nelder-Mead and the derivative-based methods. First, the derivative-based methods have significantly less branch divergence due to the lack of control flow code. In Nelder-Mead, the manipulation of the optimization simplex based on the results at the vertices makes branching unavoidable. Also, differences in register usage contribute to differences in occupancy as well as stalls due to register spilling.

In terms of raw speed on the GPU, and the CPU as well, Nelder-Mead was the fastest of the three core optimization methods. While all three methods produced similar quality meshes, Nelder-Mead converged faster. This can be partially attributed to the non-smooth nature of the optimization problem. Computing the gradients numerically does not provide information as accurate as can be obtained from the analytical gradients available for smooth functions. Whatever information the gradients provide does not make up for the increased computational cost of the gradient-based methods. The line search inherent in both gradient descent and BFGS requires more function evaluations than Nelder-Mead, allowing the latter to win the performance competition.

In some cases, the performance improvement seems to exceed the raw performance factor (in GFLOP/s) between the CPU and GPU. This is expected because our CPU results were done with a single-threaded program, while the raw CPU performance is quoted across all the constituent cores. In order to confirm that this is the case, we started primary investigations into a multi-threaded CPU implementation. Using the same labeling method as the GPU implementation, our preliminary 32-thread OpenMP implementation resulted in a speedup factor of between 3.3 to 3.8 over the single-threaded implementation, which was largely independent of the size and initial quality of the mesh. This is also consistent with our expectations, since unlike the GPU, the CPU has no concept of a block and so the thread termination issue described previously does not exist. These results are summarized in Table 3.

Mesh quality for the GPU and multi-threaded CPU based algorithms is slightly degraded from the serial version in essentially all cases. We attribute this to the different order in which elements are optimized on the GPU as opposed to the CPU. The serial algorithm has the advantage that more vertices are optimized with at least some portion of their neighbors already having been optimized, giving the optimization algorithm better information with which to work. This asynchronous convergence can be ameliorated to some extent by making more passes over the entire mesh. Table 4 shows that noticeable quality improvement can be achieved by making multiple passes over the mesh as opposed to relying only on locally iterating the optimization method. The overhead for making 3 extra passes over the mesh, in this case the Small Sphere, is only 4 seconds which is surprisingly low. The results in Table 1 used 3 passes over the mesh, and it is likely the additional passes would have yielded increased quality at the expense of more compute time. However, the quality difference between the CPU and GPU results was small enough that further passes did not seem justified.

### 2.2. *Comparison to global serial optimization*

In current practice, a production environment would most likely use the Mesquite toolkit to perform mesh optimization. We performed a series of experiments, summarized in Table 2, to compare the performance of the Nelder-Mead GPU code to that of Mesquite. We attempted to configure Mesquite as we anticipated most users would. Freitag *et. al* [14] performed a series of experiments showing that one of their custom algorithms, Feasible Newton, exhibits super-linear convergence for most problems. We chose to use Feasible Newton to minimize the average of the inverse mean ratio values of all the elements. Using the average of the quality metrics was necessitated by Feasible Newton requiring a smooth function. In an effort to determine the trade-offs between global and local optimization, Feasible Newton was applied as a global algorithm, potentially moving every vertex in the mesh each iteration. This is in contrast to the local optimization the GPU algorithm, which operates on each vertex independently.

On the GPU side, we altered the objective function to also be the average of the element qualities in order to make the optimization target mirror that used by Feasible Newton as closely as possible. Of the three core optimization methods, Nelder-Mead exhibited the best performance for this objective function as well, although the differences among the three were slight. For the GPU runs, we performed 3 passes over the mesh with Nelder-Mead using 10 iterations to locally optimize each vertex. The overhead of performing the additional passes simply amounts to the time taken to perform the extra transfers of data to and from the GPU, and this time is included in the overall run time on our results. n measuring performance, we neglect the I/O time spent reading and writing files as being immaterial and measure only CPU time for Mesquite. For the GPU algorithm we measure the wall-clock time to organize the data on the CPU, generating independent sets and local neighborhoods, as well the GPU data transfer and compute times.

As is shown in Table 2, the GPU-based algorithm and Feasible Newton produced meshes of very similar quality. The GPU algorithm generated meshes of slightly higher quality in terms of the maximum inverse mean ratio value, while Feasible Newton slightly better meshes in terms of the average value. The main advantage of the GPU-based optimization method is its speed and scalability. The GPU algorithm exhibited speedups ranging from 8 to 12 as compared to global Feasible Newton for our test meshes.

## 3. Conclusion

Our GPU-based framework offers a very promising platform for mesh optimization. Our results have shown that GPU-based optimization can be significantly faster than state-of-the-art serial global and local optimization while delivering high quality meshes. Moreover, we believe the local mesh optimization framework employed on the GPU will ultimately prove more scalable than global optimization techniques, which is a critical consideration as computational simulation moves towards exascale-level problems.

## References

[1] Lori Freitag, Patrick Knupp, Todd Munson, and Suzanne Shontz. A comparison of inexact Newton and coordinate descent mesh optimization techniques. In *Proceedings of the 13th International Meshing Roundtable*, pages 243–254, Williamsburg, VA, September 2004.

[2] Lori Freitag, Patrick Knupp, Todd Munson, and Suzanne Shontz. A comparison of two optimization methods for mesh quality improvement. *Invited Submission. Engineering with Computers*, 22(2):61–74, May 2006.

[3] Todd Munson. Optimizing the quality of mesh elements. *SIAG/Optimization News and Views*, 16:27–34, 2005.

[4] Lori Freitag, Patrick Knupp, Thomas Leurent, and Darryl Melander. MESQUITE design: Issues in the development of a mesh quality improvement toolkit. In *Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 159–168, 2002.

[5] Lori Freitag, Mark Jones, and Paul Plassmann. A parallel algorithm for mesh smoothing. *SIAM Journal on Scientific Computing*, 20(6):2023–2040, 1999.

[6] Todd Munson. Mesh shape-quality optimization using the inverse mean-ratio metric. *Mathematical Programming*, 110:561–590, May 2007.

[7] Jeonghyung Park and Suzanne M. Shontz. Two derivative-free optimization algorithms for mesh quality improvement. *Astrophysical Journal Supplement Series*, 186:457–484, 2010.

[8] Shankar Prassad Sastry, Suzanne M. Shontz, and Stephen A. Vavasis. A log-barrier method for mesh quality improvement. In *Proceedings of the 20th International Meshing Roundtable*, 2011.

[9] A.S. Lewis and M.L. Overton. Nonsmooth optimization via BFGS. *Submitted to SIAM Journal of Optimization*, 2009.

[10] Shankar Prassad Sastry and Suzanne M. Shontz. A comparison of gradient- and Hessian-based optimization methods for tetrahedral mesh quality improvement. In *Proceedings of the 18th International Meshing Roundtable*, 2009.

[11] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[12] A. Gyrfs and J. Lehel. On-line and first fit colorings of graphs. *Journal of Graph Theory*, 12(2):217–227, 1988.

[13] Vasily Volkov and Brian Kazian. Fitting FFT onto the G80 architecture. *University of California, Berkeley*, 40, 2008.

[14] Lori Freitag, Patrick Knupp, Todd Munson, and Suzanne Shontz. A comparison of two optimization methods for mesh quality improvement. In *Proceedings, 11th International Meshing Roundtable*, pages 29–40, September 2002.