Engineering with Computers
DOI 10.1007/s00366-014-0393-7

ORIGINAL ARTICLE

Efficient parallel optimization of volume meshes on heterogeneous computing systems

Zuofu Cheng · Eric Shaffer · Raine Yeh · George Zagaris · Luke Olson

Received: 3 February 2014 / Accepted: 22 December 2014

Abstract We describe a parallel algorithmic framework for optimizing the shape of elements in a simplicial volume mesh. Using fine-grained parallelism and asymmetric multiprocessing on multi-core CPU and modern graphics processing unit hardware simultaneously, we achieve speedups of more than tenfold over current state-of-the-art serial methods. In addition, improved mesh quality is obtained by optimizing both the surface and the interior vertex positions in a single pass, using feature preservation to maintain fidelity to the original mesh geometry. The framework is flexible in terms of the core numerical optimization method employed, and we provide performance results for both gradient-based and derivative-free optimization methods.

Keywords Mesh optimization · Parallel algorithms · GPU applications

1 Introduction

Computational simulation plays a key role in many modern engineering and scientific endeavors. Simulation offers ability to test designs and theories when physical prototyping is infeasible or even impossible. Modeling of physical geometry for simulation typically involves the creation of a mesh of elements, with simplicial meshes of triangles and tetrahedral being very popular. The shape of mesh elements significantly impacts the efficiency and accuracy of simulation codes. The problem of improving element quality in unstructured tetrahedral meshes and triangular surface meshes is classically framed as an optimization problem. In this framework, the positions of the mesh vertices are adjusted to optimize element quality. In this paper, we explore how modern computer hardware, meaning heterogeneous many-core systems, accelerates these optimization computations. The reward for optimizing meshes faster is significant in practical terms, with fewer simulations failing due to poor mesh quality and a faster engineering design cycle.

The wide availability of massively multi-threaded graphics processing units (GPUs) and multi-core CPUs offers a new direction for the acceleration of mesh optimization algorithms. Our work shows how optimization is effectively accomplished on a per-vertex basis on heterogeneous systems, exposing fine-grained parallelism in the same manner as Freitag et al. [4] did for more traditional parallel architectures. Our algorithm specifically seeks to reduce the maximum and average inverse mean ratio, which detects irregular and inverted simplex elements [5, 6, 15, 18]. The framework of the algorithm is very flexible and accommodates essentially any underlying quality metric and core numerical optimization method.

The main contribution of this paper is an examination of the effectiveness of a hybrid parallel optimization scheme as opposed to GPU-only parallelism for mesh optimization. In addition, we compare the performance of three different core numerical optimization techniques on a Kepler-class GPU. We describe in detail the use of the derivative-free Nelder–Mead simplex method, which in a previous work was seen to converge faster than several peer optimization methods for this application [22]. In our new results, Nelder–Mead continues to demonstrate superiority

Z. Cheng (✉) · E. Shaffer · L. Olson
University of Illinois, Urbana, IL 61801, USA
e-mail: zcheng1@illinois.edu

R. Yeh
Purdue University, West Lafayette, IL 47907, USA

G. Zagaris
Kitware Inc., University of Illinois, Urbana, IL 61801, USA

Published online: 10 January 2015

over gradient-based methods. This is of some interest as even though the global quality function we employ is non-smooth, it is differentiable almost everywhere and one could imagine the use of gradients providing a faster convergence rate. This result has general applicability, as measuring mesh quality in terms of the worst element naturally leads to global quality functions with similar characteristics. Finally, we examine the significant performance gain achieved by running on Kepler-class hardware as opposed to Fermi-class hardware.

In addition to possessing generality, the algorithmic framework we describe is simple to implement and fast. Our framework also incorporates the ability to preserve surface and boundary features in an optimal manner, maintaining close geometric fidelity to the original mesh while striving to obtain the highest possible element quality. In our experiments, the parallel framework was shown to converge to a high-quality solution up to 21 times faster than the serial version of the algorithm based on local optimization. These results demonstrate the scalability and effectiveness of heterogeneous systems as a platform for volume mesh optimization.

2 Previous work

Mesh optimization is a long-standing research area. Typically, mesh improvement is accomplished by targeting the worst element, as determined by a quality metric which is related in some way to the impact the element has on the convergence of a numerical simulation that employs the mesh as a domain. Excellent references on the surprisingly large array of quality metrics employed in practice include [12] and [24]. Some of the worst-element improvement techniques most relevant to the work presented in this paper include [7] which was one of the first to use numerical optimization techniques. While gradient-based optimization methods are often preferred in general practice due to superior convergence rates, the mesh optimization problem is often non-smooth, implying that the gradients are not immediately available. Park and Shontz focus in particular on derivative-free optimization methods in a serial setting [20]. Derivative-free optimization is similarly used in [27], as a pseudo-random search method is used. Another approach is detailed in [21], which reformulates the mesh optimization problem as a contained smooth problem and employs an interior point log-barrier method to optimize the mesh. Both of the latter works are global methods and so not readily applicable to the problem we focus on, namely using local optimization to leverage the fine-grained parallelism of the GPU.

Notable local methods, employed for serial surface mesh optimization, include that of Garimella et al. [8] and Montenegro et al. [17]. In both cases, optimization is performed in a local parametric space. The GETMe algorithm of Vartziotis et al. [25] is a local heuristic for improving volume (specifically tetrahedral) mesh quality. While well suited for parallelization, the algorithm must take extra care to handle elements that have been invalidated, causing extra program flow branches which make it a poor fit for the highly parallel nature of GPU computation. We have preferred to focus on numerical optimization methods which obviate the need for such checks by using constraints.

Parallel mesh optimization is also, as one would expect, a well-studied area. Our work builds upon that of Freitag et al. [4], which uses local mesh optimization in parallel in a traditional high-performance computing MPI setting. Jiao and Alexander [11] developed the idea of using the medial quadric for feature detection and preservation in surface meshes within an MPI setting. We have employed the same construct effectively in a GPU setting, adding the step of performing one-dimensional optimization across ridge features. A current work that is most similar to ours is the CPU–GPU mesh optimization system proposed by D'Amato and Vénere [2] which uses OpenMP and GPU computing, with the latter performing mesh smoothing as our work does. D'Amato and Vénere employ a modified pattern search, a derivative-free method, on the GPU and confine their experiments to meshes of 2 million elements or less. While their reported speedups are less than ours, such a comparison is complicated by the fact that their algorithm likely achieves higher qualities due to a willingness to perform vertex insertions and alter connectivity during their CPU phase. Our results, which provide data on the performance of Nelder–Mead as well as gradient-based optimization on the GPU and across larger mesh sizes, add to the research of D'Amato and Vénere, providing a more complete picture of the performance of various optimization methods on the current GPU hardware.

3 Volume mesh optimization

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

3.1 The optimization problem

We choose to measure the quality of a tetrahedron by the inverse mean ratio metric [19]. 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 Fig. 1. It is clear that the inverse mean ratio

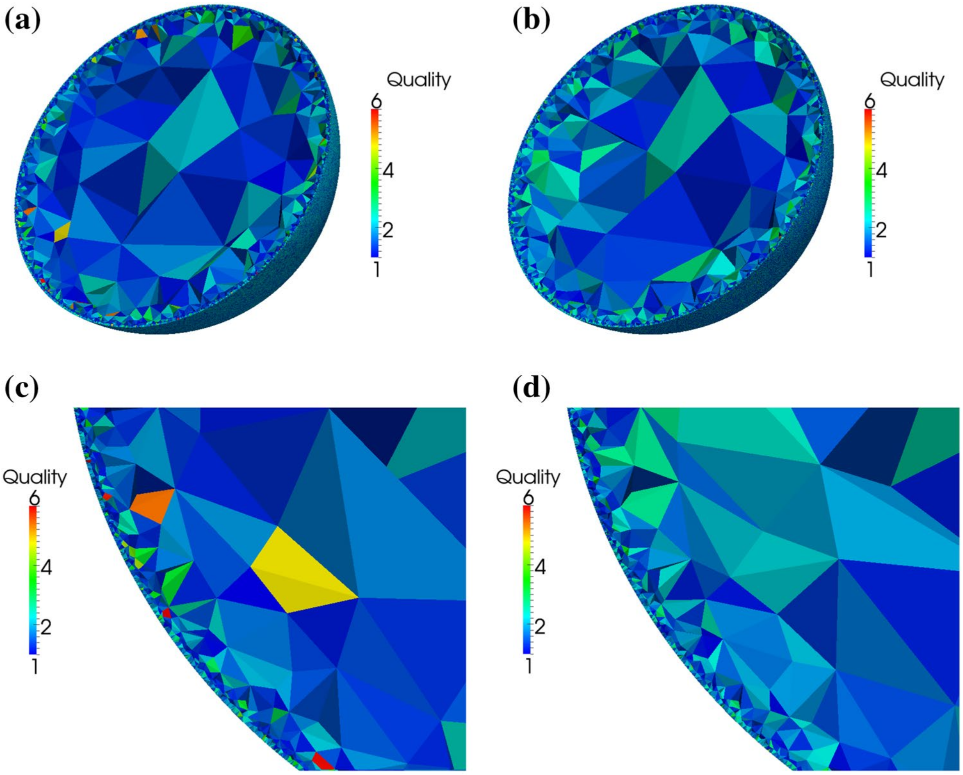

Fig. 1 Optimization of the max inverse mean ratio metric on a sphere mesh. **a** Unopitmized, **b** optimized, **c** unopitmized and **d** optimized

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 emanating from vertex a and W as a square matrix representing the ideal element.

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

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 ∞ 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. Mathematically, we consider a function $f_v : \mathscr{R}^k \rightarrow \mathscr{R}$, where the input, $\mathbf{Q} = [q_1, q_2, \ldots, q_k]$, is a vector of k element qualities around the vertex v. Here, we define $f_v = \max(q_1, q_2, \ldots, q_k)$, which measures the lowest quality tetrahedron incident to v as measured by the inverse mean ratio. As a result, we seek optimal vertex locations $\mathbf{x}_v$ based on this objective function $f(\mathbf{x}_v) = f_v$.

3.2 Nelder–Mead method

As a result of previous investigations [22], we have found that derivative-free methods yield the most consistent mesh quality improvement, due to the non-smooth nature of the optimization space. For optimization of both the interior and the surface of the volume mesh, we implement serial, CPU-based parallel and GPU-based parallel versions of the optimization algorithm. The reason for this is twofold: first, we would like to compare the relative speedups of a massively parallel problem with respect to both multiple CPU threads and GPU acceleration; second, optimization on the CPU for one part of the mesh (for example, the surface) takes approximately the same amount of time as optimization of the balance on the GPU, allowing us to leverage both computation devices to speed up the total combined mesh optimization problem.

Nelder–Mead is used as our core optimization algorithm for this experiment. For our purposes, we must implement both the two-dimensional and three-dimensional variants of the Nelder–Mead algorithm. The three-dimensional version is used for the interior element optimization, where the vertices are largely unconstrained. One minor constraint here is that we do not allow the new position of the vertex to move past a point at which the mesh element would invert. Fortunately, this is handled elegantly within the objective function, which automatically penalizes vertex positions which result in inverted elements. This process is more complicated for the surface vertices, where one or more of the constituent tetrahedral elements have a face which resides on the surface. In this case, we cannot simply move the vertex anywhere, as there exists the undesirable possibility of distorting the shape of the mesh. Here, we must move the vertex only in ways which are compatible with the topological features of the mesh [23] as summarized below.

We maintain this constraint by generating a hull out of the volume mesh by selecting the surface faces. We then compute the normal vector for each of the vertices to define an optimization plane, which has the projected coordinate $\langle u, v\rangle$. Note that the Nelder–Mead algorithm searches within $\langle u, v\rangle$ space, but the objective function is still based on the three-dimensional inverse mean ratio; the position of surface vertices may affect the quality of non-surface interior elements. If the feature detection option described in the next section is enabled, vertices which are classified as being on a ridge simply use a one-dimensional line search, while vertices which are in a corner are skipped entirely. The remaining surface vertices are optimized using the two-dimensional Nelder–Mead as described below with a triangular simplex; the algorithm for the three-dimensional variant uses a tetrahedral simplex. Note that the simplex here is a construct used in optimization, different from the simplex elements of the mesh. The simplex for optimization represents the current search region—ideally, the optimization simplex will converge to a single point as a search converges to an optimal vertex position.

In each step, the vertices of the current optimization simplex are ordered by quality, with $Q(p_1)$ being the best and $Q(p_{n+1})$ being the worst (where n is the dimension of the search). The worst point, p_{n+1}, is then reflected across the center of mass of the other vertices in the simplex. If the reflected vertex p_r has a quality better than the worst vertex, the simplex is expanded in the direction of the reflected vertex, increasing the volume contained within the simplex, and the new vertex replaces the worst vertex if it shows improvement. If instead p_r has worse quality than the previous worst vertex such that $Qp_\mathrm{r} > Q(p_{n+1})$, we contract the simplex along the direction of p_r and similarly replace the worst vertex with the new contracted vertex if it is better. Finally, if neither the contraction nor expansion creates a better vertex than p_{n+1}, we shrink the entire simplex towards the best vertex, p_1. This process is repeated until either the specified number of iterations is completed, or until the step size falls below a given threshold. The initial simplex given is determined by first computing the edge lengths toward all the neighboring vertices and then multiplying it by a factor $\alpha < 0.5$. This is so that if the first iteration inverts the search simplex, it does not immediately result in an inverted mesh element.

```
//initialize:
//x as initial point
//s as initial simplex
//e = expand(r, m) computes e = 2r - m
//r = reflect(x, m) computes r = 2m - x
//c = contract(r, m, x) assigns c to the lowest
//objective function evaluation of:
//c1 = .5 (r + m);
//c2 = .5 (x + m);
//find_order returns index of input from low to high
//such that fs = find_order(f)
//returns fs=[0, 3, 2, 1] if f=[1, 4, 3, 2]
for (i = 0; i < num_iterations; i++){
    //evaluate objective at each simplex point
    Vector f = evalf(s);
    Vector fs = find_order(f);
    Vector m = midpoint(s[fs[0]], s[fs[1]], s[fs[2]]);
    //r is highest reflected over m
    Vector r = reflect(s[fs[3]], m);
    if (f[fs[0]] > evalf(e)){
        //expansion
        Vector e = expand(r, m);
        if (evalf(r) > evalf(m)){
            s[fs[2]] = e;
        }
        else {
            s[fs[2]] = r;
        }
    else if (f[fs[2]] > evalf(r) && evalf(r) >= f[fs[0]]) {
        //reflection
        s[fs[2]] = r;
    }
    else {
        //evalf(r) >= f[fs[2]] is implied
        Vector c = contract(r, m, s[fs[3]]);
        float b = (evalf(r) < f[fs[3]]) ? evalf(r) : f[fs[3]];
        if (evalf(c) >= b){
            //reduction
            s[fs[1]] = .5 * (s[fs[0]] + s[fs[1]]);
            s[fs[2]] = .5 * (s[fs[0]] + s[fs[2]]);
            s[fs[3]] = .5 * (s[fs[0]] + s[fs[3]]);
        }
        else {
            //contraction
            S[fs[3]] = c;
        }
    }
}
```

A pseudo-code description of the Nelder–Mead method is summarized as follows:

3.3 BFGS method

As we have previously stated, previous investigations have found that derivative-free methods performed better for the mesh optimization problem. Nevertheless, we have provided a baseline comparison by including interior mesh optimization results as compared to a popular derivative-based method, the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm. A detailed analysis of the BFGS algorithm in dealing with non-smooth optimization problems such as ours can be found in [14]. The general idea of the BFGS is similar to gradient descent; however, the BFGS algorithm also takes into account the second derivatives by approximating the Hessian matrix iteratively. A pseudo-code description of our implementation of BFGS is described as follows. Note that the core BFGS algorithm only determines a search direction within the optimization space and we must additionally use a line search to evaluate for a minimum along that direction.

```
//initialize:
//x as initial point
//b as 3x3 identity matrix
//old_g and s as zero vectors
for (i = 0; i < num_iterations; i++){
    //n iterations of BFGS
    //compute gradient at x
    Vector g = gradf(x);
    Vector y = g - old_g;
    Vector p = b.inverse * g;
    float y_dot_s = dot(y, s);
    float s_b_s = ((s.transpose * b), s);
    if (i > 0){
        //check stop conditions, not on first iteration
        if (abs(y_dot_s) < A_SMALL_NUMBER ||
            abs(s_b_s) < A_SMALL_NUMBER)
            break //converged or got stuck
    }
    Matrix b_update = Vector.outer(s, s);
    b_update = (b * b_update) * b;
    b_update *= (1.0/s_b_s);
    //update approximate Hessian
    b = b + Vector.outer(y, y)*(1.0/y_dot_s) - b_update;
    //line search in direction of p for step a
    float a = linesearchf(x, p);
    s = a*p;
    x += s;
}
```

3.4 Feature preservation

We detect mesh features by making use of the technique called the medial quadric [12], which analyzes the eigenvalue decomposition of the normal tensor of a vertex. Given a vertex v, the normal tensor M of v is calculated by

$$\mathsf{M} = \sum_i w_i \mathbf{m_i} \mathbf{m_i}^{\mathrm{T}}$$

where $\mathbf{m_i}$ is the face normal of the ith incident face of vertex v, and w_i is the face area associated with the ith face. The eigenvalue decomposition of M is

$$\mathsf{M} = \sum_{i=1}^{3} \lambda_i \mathbf{e_i} \mathbf{e_i}^{\mathrm{T}}$$

where λ_i are the eigenvalues with $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0$, and $\mathbf{e_i}$ are the corresponding eigenvectors. The eigenvalues are used to classify the vertex v to be smooth, on a ridge, or a corner as follows. When $\lambda_2 \ll \lambda_3$ and $\lambda_3 \ll \lambda_1$, the

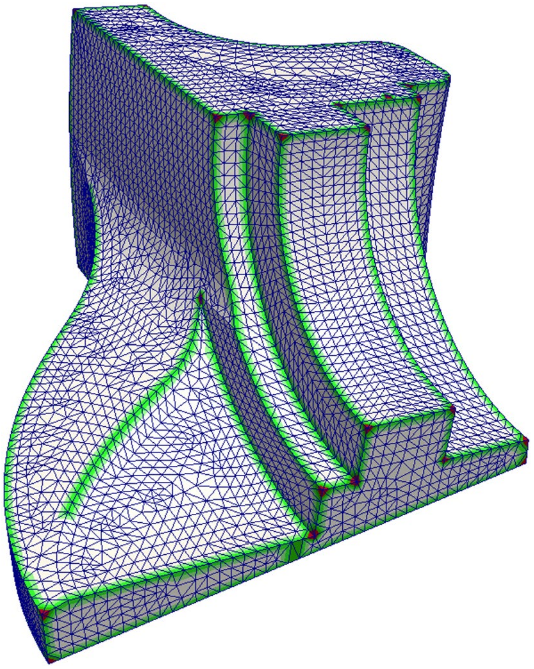

Fig. 2 Example mesh with detected features (ridge or corner) highlighted in *green* (color figure online)

faces surrounding the vertex v are smooth. The face normals are nearly identical and therefore the vertex is classified as being on a flat or smoothly varying portion of the surface. At ridges, there are two distinct normal directions and $\lambda_3 \ll \lambda_2$. At a corner, the eigenvalues will follow $\lambda_1 \approx \lambda_2 \approx \lambda_3$. This method is fairly reliable for non-acute ridges. To handle acute ridges where the vertex may be mistakenly classified as smooth, we add a safeguard by classifying vertices as on a ridge if the dihedral angle of a pair of adjacent faces is less than 90°. An example of a mesh with detected features is shown in Fig. 2.

Our optimization algorithm handles vertices differently depending on their classification. The vertices on smooth surfaces are optimized freely within the surface tangent plane while vertices classified as corners are not moved. To optimize vertices along a ridge, we use the golden section technique for optimizing one-dimensional functions. Note that one-dimensional optimization is all that is required because vertex movement is constrained to occur along the ridge direction estimated by the eigenvector. At a ridge vertex, the null space of M is aligned with the ridge direction, so we use $\mathbf{e_3}$ to estimate the ridge direction. Then we move the vertex along the ridge direction to find a point on the ridge that will maximize mesh quality.

Recent research into one-dimensional optimization along ridges and curves has been done by McLaurin and Shontz [16]. Their method is designed for more general situation and focuses on minimizing arc-length deficit. We opt for a simpler approach that requires only local information, as opposed to knowledge of the entire ridge, and is better suited for parallelization. The golden section search is a technique that finds an extremum of a function by successively narrowing the range of value where the extremum is known to exist. Like Nelder–Mead, golden section makes few assumptions and can be applied to non-smooth functions. In fact, all that is necessary to assure convergence to the true function extremum is that the function be unimodal, meaning that local extrema do not exist.

In this case, we find the maximum of the minimum angle function,

$$f(x) = \min_{j \in S_i} \theta_j(x)$$

where S_i is the set of angles affected by the movement of the vertex being optimized, and $\theta_j(x)$ is the angle function of the jth angle. Note that with a smooth movement of the vertex on a line, $\theta_j(x)$ will be a continuous function. Whichever direction the vertex moves, as long as moving the vertex will not result in an inverted triangle, $\theta_j(x)$ will either be monotonically increasing, monotonically decreasing, constant, or a concave function. The function $f(x)$, in taking the minimum of this group of functions, will be a weakly unimodal function. There may be many maxima along a single contiguous plateau, but there will not be more than one peak. For the function we employ, golden section search finds a global maximum.

3.5 Finding independent sets

The core of our algorithm involves assigning each vertex of the mesh to a separate thread, CPU if the vertex is located on the surface or GPU otherwise. This allows us to leverage both fine-grained parallelism on the vertex level and asymmetric parallelism by using both the CPU and GPU simultaneously. The high-level steps of the optimization algorithm can be summarized in Algorithm 1. 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

Algorithm 1 CPU/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
    if OnSurface($v_i$) then
        $f_i \Leftarrow$ Classify($v_i$) // classify and assign $v_i$ a feature
        $\mathbf{U_i} \Leftarrow$ Project($v_i$) // compute projection frame of $v_i$
    end if
end for
for each color $k$ do
    for each vertex $v_i$ do
        if $c_i = k$ then
            if OnSurface($v_k$) then
                add $v_i$ and $\mathbf{T_i}$ to set $\mathbf{S_k}$
                // $\mathbf{S_k}$ is an independent set of interior vertices and their neighboring elements
            else
                add $v_i$ , $\mathbf{T_i}$, and $f_i$ to set $\mathbf{P_k}$
                // $\mathbf{P_k}$ is an separate independent set of surface vertices and their neighboring elements
            end if
        end if
    end for
    Transfer $\mathbf{S_k}$ to the GPU
    for  each $v_i \in S_k$ do
        $v_i \Leftarrow$ Optimize3D($v_i, \mathbf{T_i}$)
        // Optimize3D($v_i, \mathbf{T_i}$) performs 3D Nelder-Mead
    end for
    for  each $v_j \in P_k$ do
        // simultanously on the CPU
        if $f_j$ = CORNER then
            // do nothing
        else if $f_j$ = RIDGE then
            $v_i \Leftarrow$ LineSearch($v_j, \mathbf{T_j}$)
        else
            $u_i \Leftarrow$ Project ($v_j, \mathbf{U_j}$)
            $u_i \Leftarrow$ Optimize2D($u_j, v_i, \mathbf{T_j}$)
            $v_i \Leftarrow$ UnProject ($u_j, \mathbf{U_j}$)
        end if
    end for
    Transfer all $v_i \in \mathbf{S_k}$ back to the CPU
end for
```

that no two neighbors have the same label using a First Fit algorithm [10] on the CPU where each label denotes an independent set of vertices. Note that the First Fit method does not produce an ideal labeling in terms of generating the fewest number of independent sets (which is known to be an NP-hard problem). While many algorithms for heuristic-based vertex labeling have been shown to perform better in terms of generating fewer sets [3, 13], the effect of having at most one more label than is needed (the worst-case bound for greedy labeling) does not result in a significant performance penalty in the run time of the algorithm, as in general the overhead associated with a CUDA kernel launch without additional host to device memory transfers is less than 1 ms. In addition, surface vertices and interior vertices must have independent sets, as the CPU similarly cannot optimize a vertex while the GPU may be optimizing the neighbors. Each pass of the algorithm works within the sets and the total set of vertices is only synchronized at the end of the optimization process.

Graph coloring is a key component of both the CPU–GPU mesh optimization system of D'Amato and Vénere [2] and the MPI–OpenMP system of Gorman et al. [9]. In both cases, the authors employ First-Fit heuristics as well. One key consideration in terms of scalability is the memory access pattern of the heuristic. For meshes large enough to strain system memory, thrashing and severely degraded performance are distinct possibilities we have encountered in practice. Boman et al. use a distributed coloring algorithm which obviates that problem to a significant degree [1]. That is not an option in our case, as our framework is not designed for a cluster environment and so multiple pools of memory are not available. Implementing a memory coherent layout of the mesh data would address the problem in a single system environment and is an area of future work within our framework. Also of interest would be a study of how independent set construction affects optimization quality. It is clear that different orderings of vertex optimizations can produce different final mesh qualities. However, it is not clear how one could adjust the coloring heuristic to drive toward higher quality or what the maximum impact of a different ordering can be. A greater understanding of that phenomenon could have implications for system design choices.

3.6 Implementation

Our method performs surface optimizations on the CPU and interior optimizations on potentially both the CPU and GPU. This presents a load balancing issue, as not all meshes have a large number of interior vertices as compared to surface vertices. While we suspect that this is the case for most practical meshes, one of our test meshes (the "big sphere") has a large number of surface vertices as compared to interior vertices to test performance in this unusual situation. It turns out that due to poor load balancing between the CPU and GPU, this mesh results in the least speedup using our combined optimization algorithm, although it manages a respectable speedup of a factor of 10 over the serial method. Future work will split CPU and GPU loads based on general work units rather than based on surface/interior position, but this has the difficulty in that performance estimation of the CPU or GPU time required to optimize a particular vertex is difficult to quantify due to the differing constraints on surface vertices because of the feature classification. Because of the additional constraint on the independence of optimization threads "in-flight", load balancing while keeping into account these additional constraints remains a challenge.

Our combined algorithm uses two separate GPU paths for interior vertex optimization depending on GPU generation. The first is intended for Fermi and first-generation Kepler GPUs (compute capability 2.0–3.0). In this optimization path, we noticed that register bottlenecks inside the three-dimensional Nelder–Mead process caused poor performance, even with fairly high utilization [22]. This is because the Nelder–Mead algorithm requires the use of many GPU registers to store intermediate variables, such as the vertices of the optimization simplex, the quality at each vertex, etc. When the register usage exceeds the number of registers available on the hardware, register operations which normally take a single GPU cycle convert into high-latency global memory accesses and is a condition we would like to avoid. Therefore, an optional optimization here uses the 48 KB of shared memory as a manually managed cache space for the local neighborhood of the currently optimized vertices. Using this optimization requires a much smaller block size, typically 64 threads, (due to the limited size of the shared memory), but prevents register spill situation on GPUs where the maximum number of registers per thread is 63. This approach essentially uses the GPU as a streaming processor, bypassing the inherent load balancing and scheduling logic by manual management of the memory accesses, similar to the optimization done by Volkov et al. in [26]. On some architectures, this optimization results in a performance improvement of up to 25 %.

On second-generation Kepler GPUs (compute capability 3.5), this optimization is counterproductive due to the higher number of registers allowed per thread. Here, the block size is 256 threads to take into account the larger number of streaming-processors in the Kepler SMX. On this platform, our Nelder–Mead implementation uses 84 registers per thread. The straightforward approach here results in the performance of the algorithm on a GTX Titan to be roughly 2.5 times faster than a Tesla C2050, in line with the roughly three times increase in single precision Gflops.

Surface vertices are optimized in parallel on the CPU using OpenMP. Here, like the GPU implementation, optimization must work on independent sets of vertices which are also all on the surface. This makes for much smaller sets, and therefore the surface optimization is well suited for the smaller number (16–64) of OpenMP threads, rather than the thousands of GPU threads. One advantage of this approach is that the overhead due to thread termination is low; GPU threads can only terminate when the entire block has terminated, but a CPU thread has no such constraints, allowing for higher utilization.

4 Experimental results

We perform experiments to evaluate the performance of our combined algorithm in different configurations. The hardware system in our primary experiments contained an Intel Xeon X5650 CPU with six cores supporting up to 12 threads clocked at 2.67 GHz with 12 MB of cache memory and 16 GB of main memory. The GPU was a Tesla C2050 (Fermi architecture) with 448 Cores and 2.6 GB of memory with ECC enabled. Experiments are done in Linux using code compiled by gcc 4.4.0 and the Nvidia CUDA toolkit version 4.2. Supplementary tests are performed on a GTX Titan with 2688 cores and 6 GB of memory (with no ECC support) and CUDA toolkit 5.0. Optimization is set to the highest level (for speed) in both gcc and nvcc, no assembly optimizations or SIMD intrinsics are used in the serial and parallel CPU versions (beyond those required to support OpenMP).

4.1 Quality comparisons

The first of our experiments consisted of a single pass of the previously described Nelder–Mead algorithm with 100 iterations. In our previous work, we have found that breaking down the optimization iterations into multiple passes (for example, 4 passes of 50 iterations versus 200 iterations of a single pass) can also improve final mesh quality. We speculated that this behavior is due to the local nature of our optimization process and the dependency of each vertex having a neighborhood which is already optimized. For the purposes of comparing our combined optimization to previous interior element-only optimization methods, we did not re-run the multi-pass tests; however, we suspect that the improvement will still hold. In any case, as Table 1 shows, our optimization improves the final global maximum inverse mean ratio of the test meshes in all tested situations. One interesting point is that only optimizing the internal elements improved the overall maximum inverse mean ratio, even if the vertices on the surface did not move at all. This suggests that either that the surface mesh is of fairly good quality already, or that the one free vertex in a tetrahedral element where one face is on the surface is sufficient to improve the quality of this situation. Even so, optimizing the surface vertices in addition to the interior vertices significantly improved mesh quality over only optimizing the internal elements.

In supplementary tests, to justify our algorithm as a framework, we also performed tests replacing the Nelder–Mead algorithm with the previously defined BFGS and also the common gradient descent algorithms. Note that these are derivative-based methods; therefore, an approximate derivative using the central difference method was used for this experiment. Also, since the BFGS and gradient descent methods only define a direction of search, we used a uniformly sampled line-search to find the best point once a direction was determined by either gradient descent of BFGS. Note that this means that one “iteration” of the gradient methods can potentially result in many more objective function evaluations than Nelder–Mead, depending on the density of the sampled line search. To normalize for this, we allowed the gradient methods to run for the same run time on the GPU as in Table 1 using a line search density of 200 points per direction and compared the resulting mesh quality in Table 2. Tests were performed on a GTX Titan using the combined optimization for this experiment, although in theory any device with the same numeric precision should show similar normalized results.

It is clear that these results are consistent with our previous findings that Nelder–Mead outperformed derivative-based methods. In addition, it is not clear whether BFGS adds any advantage over simple gradient descent. This is likely because of the use of the central difference approximation for derivative calculation, as we cannot expect to capture an accurate representation of the second derivative of the optimization space using only three points. Therefore, it makes sense that the Hessian approximation in our BFGS calculation does not converge to an accurate representation of the underlying optimization space and ultimately the BFGS results are very similar to that achieved with gradient descent.

4.2 Performance comparisons

Table 3a–c compares the performance of our combined CPU/GPU algorithm on both the Tesla C2050 and the GTX Titan compared to serial and OpenMP only performance. We see that, in general, the parallel simultaneous optimization of the mesh using both the CPU and GPU results in the best mesh quality in the shortest amount of time. The last column of Table 3 shows speedups of over ten times when performing a surface and interior parallel optimize (using the GTX Titan) over the serial code, while also yielding the

Table 1 Quality of optimized meshes compared to the original

Mesh	Number of elements	Maximum IMR after no optimization	Quality interior only optimization	Quality full optimization
Small rocket	468,623	2.229	1.992	1.84
Big sphere	4,720,255	8.645	5.813	3.705
Big rocket	14,992,367	14.971	5.0897	3.566

Table 2 Quality of combined optimization with different methods on GTX Titan (normalized to GPU time of 100 iterations of Nelder–Mead)

Mesh	Number of elements	Quality Nelder–Mead	Quality BFGS	Quality gradient descent
Small rocket	468,623	1.84	1.99	1.99
Big sphere	4,720,255	3.705	4.34	4.44
Big rocket	14,992,367	3.566	3.93	3.84

Table 3 Performance comparison of parallel methods with serial methods

Mesh	Interior vertices	Serial (s)	OpenMP (speedup)	C2050 (speedup)	GTX Titan (speedup)
(a) Speedup over serial of interior vertex optimization					
Small rocket	58,981	60.7	8.4	8.0	14.0
Big sphere	290,739	571.1	7.1	9.4	21.5
Big rocket	2,202,793	1967.2	7.7	5.9	16.4

Mesh	Surface vertices	Serial (s)	OpenMP (speedup)
(b) Speedup over serial of surface vertex optimization			
Small rocket	38,673	16.0	3.1
Big sphere	1,048,578	341.1	2.5
Big rocket	576,688	381.5	4.1

Elements	Total vertices	Serial (s)	Parallel CPU + GPU (speedup)
(c) Speedup over serial of combined optimization (GTX Titan)			
468,623	97,654	76.8	14.6
4,720,255	1,339,317	912.2	10.9
14,992,367	2,779.481	2,348.7	15.7

best mesh quality, as noted previously. Note that the total optimization time for the CPU/GPU combined optimize is the maximum of the GPU time and the CPU time, while the total optimize time for the serial code-path is the sum.

A more detailed look reveals some of the interesting issues with respect to the relative CPU and GPU performance. Undoubtedly, the GTX Titan performs the best in interior vertex optimization. While we expected the C2050 to also beat the Xeon, this is not the case in all situations. Even when it does not, however, the performance numbers suggest that having the GPU optimize the interior is preferable, as the CPU can simultaneously optimize the surface vertices. Also of interest is the relatively higher speedup for the OpenMP interior optimization vs. the OpenMP surface optimization. This suggests that perhaps the projection and un-projection code is not as friendly to the Xeon's cache as the rest of the optimization process, most likely due to the additional data required for fetching the local projection frame saturating the limited cache memory.

Finally, it is interesting to consider performance differences between the C2050 and GTX Titan GPUs. In general, the GTX Titan performed up to 2.5 times faster, in line with the increase in single-precision between the generations. The C2050 performed the worst on the Big Rocket mesh, even though the GTX Titan performs respectably. We suspect that this is because this particular mesh has a large number of tetrahedral elements as compared to vertices, resulting in a large number of incident elements into each vertex (that is, each vertex is shared by many tetrahedral elements). This makes the shared memory-based manual caching perform poorly, as more data have to be cached per GPU multiprocessor, resulting in even fewer threads per block, leaving much of the GPU underutilized. This constraint is not present in the GTX Titan, so this particular optimization is not useful with the higher number of registers per thread.

5 Conclusion and future work

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

A number of targets exist for future work. In particular, the framework should be expanded to include a schedule that determines which sets of vertices to optimize on the GPU and which to optimize on the CPU. It should be feasible to have the scheduler be system aware and consider factors such as the specific GPU latency and processor speeds of the system in making such decisions. Memory efficiency looms as the most immediate problem in terms of scalability. A memory-efficient graph-coloring method is required to ultimately scale up to process truly massive meshes on a single system. If that bottleneck can be passed, the rest of the system can be implemented using an out-of-core approach. Finally, a production system that would be useful in engineering would require support for mixed-element meshes as well.

References

1. Boman E, Bozda D, Catalyurek U, Gebremedhin A, Manne F (2005) A scalable parallel graph coloring algorithm for distributed memory computers. In: Cunha J, Medeiros P (eds) Euro-Par 2005 parallel processing, vol 3648., Lecture notes in computer scienceSpringer, Berlin, pp 241–251
2. DAmato J, Vnere M (2013) A CPUGPU framework for optimizing the quality of large meshes. J Parallel Distrib Comput 73(8):1127–1134
3. Fleurent C, Ferland JA (1996) Genetic and hybrid algorithms for graph coloring. Ann Oper Res 63(3):437–461
4. Freitag L, Jones M, Plassmann P (1999) A parallel algorithm for mesh smoothing. SIAM J Sci Comput 20(6):2023–2040
5. Freitag L, Knupp P, Munson T, Shontz S (2004) A comparison of inexact Newton and coordinate descent mesh optimization techniques. In: Proceedings of the 13th international meshing roundtable, Williamsburg, pp 243–254
6. Freitag L, Knupp P, Munson T, Shontz S (2006) A comparison of two optimization methods for mesh quality improvement. Invit Submiss Eng Comput 22(2):61–74
7. Freitag LA, Plassmann P (2000) Local optimization-based simplicial mesh untangling and improvement. Int J Numer Methods Eng 49(1–2):109–125
8. Garimella RV, Shashkov MJ, Knupp PM (2004) Triangular and quadrilateral surface mesh quality optimization using local parametrization. Comput Methods Appl Mech Eng 193(911):913–928
9. Gorman G, Southern J, Farrell P, Piggott M, Rokos G, Kelly P (2012) Hybrid OpenMP/MPI anisotropic mesh smoothing. Proc Comput Sci 9(0):1513–1522 (proceedings of the international conference on computational science, ICCS 2012)
10. Gyrfs A, Lehel J (1988) On-line and first fit colorings of graphs. J Graph Theory 12(2):217–227
11. Jiao X, Alexander P (2005) Parallel feature-preserving mesh smoothing. In: Gervasi O, Gavrilova M, Kumar V, Lagan A, Lee H, Mun Y, Taniar D, Tan C (eds) Computational science and its applications ICCSA 2005, vol 3483., Lecture notes in computer scienceSpringer, Berlin, pp 1180–1189
12. Jiao X, Bayyana NR (2008) Identification of c1 and c2 discontinuities for surface meshes in CAD. Comput Aided Des 40:160–175
13. Jones MT, Plassmann PE (1993) A parallel graph coloring heuristic. SIAM J Sci Comput 14(3):654–669
14. Lewis A, Overton M (2009) Nonsmooth optimization via BFGS. SIAM J Optim (submitted)
15. Liu A, Joe B (1994) Relationship between tetrahedron shape measures. BIT Numer Math 34(2):268–287
16. McLaurin D, Shontz S (2014) Automated edge grid generation based on arc-length optimization. In: Sarrate J, Staten M (eds) Proceedings of the 22nd international meshing roundtable, pp 385–403. Springer, New York
17. Montenegro R, Escobar J, Montero G (2005) Quality improvement of surface triangulations. In: Hanks B (ed) Proceedings of the 14th international meshing roundtable. Springer, Berlin, pp 469–480
18. Munson T (2005) Optimizing the quality of mesh elements. SIAG/Optim News Views 16:27–34
19. Munson T (2007) Mesh shape-quality optimization using the inverse mean-ratio metric. Math Program 110:561–590
20. Park J, Shontz SM (2010) Two derivative-free optimization algorithms for mesh quality improvement. Proc Comput Sci 1(1):387–396 (ICCS 2010)
21. Sastry S, Shontz S, Vavasis S (2014) A log-barrier method for mesh quality improvement and untangling. Eng Comput 30(3):315–329
22. Shaffer E, Cheng Z, Yeh R, Zagaris G, Olson L (2014) Efficient GPU-based optimization of volume meshes. In: Bader M (ed) Parallel computing: accelerating computational science and engineering (CSE). Advances in parallel computing, vol 25. IOS Press BV, Amsterdam, pp 285–294
23. Shaffer E, Zagaris G (2011) GPU accelerated derivative-free mesh optimization. In: Hwu W-M (ed) GPU computing gems jade edition. Applications of GPU computing series, chap 13, 1st edn. Morgan Kaufmann, Waltham, pp 145–154
24. Shewchuk J (2002) What is a good linear element? Interpolation, conditioning, and quality measures. In: Proceedings of the 11th international meshing roundtable, pp 115–126
25. Vartziotis D, Wipper J, Schwald B (2009) The geometric element transformation method for tetrahedral mesh smoothing. Comput Methods Appl Mech Eng 199(14):169–182
26. Volkov V, Kazian B (2008) Fitting FFT onto the G80 architecture, vol 40. University of California, Berkeley
27. Walton S, Hassan O, Morgan K (2013) Reduced order mesh optimisation using proper orthogonal decomposition and a modified cuckoo search. Int J Numer Methods Eng 93(5):527–550