Parallel L-BFGS-B Algorithm on GPU

Abstract

Due to the rapid advance of general-purpose graphics processing unit (GPU), it is an active research topic to study performance improvement of non-linear optimization with parallel implementation on GPU, as attested by the much research on parallel implementation of relatively simple optimization methods, such as the conjugate gradient method. We study in this context the L-BFGS-B method, or the *limited memory Broyden-Fletcher-Goldfarb-Shanno with boundaries*, which is a sophisticated yet efficient optimization method widely used in computer graphics as well as general scientific computation. By analyzing and resolving the inherent dependencies of some of its search steps, we propose an efficient GPU-based parallel implementation of L-BFGS-B on the GPU. We justify our design decisions and demonstrate significant speed-up by our parallel implementation in solving the centroidal Voronoi tessellation (CVT) problem as well as some typical computing problems.

Keywords:
Nonlinear optimization, L-BFGS-B, GPU, CVT

1. Introduction

Nonlinear energy minimization is at the core of many algorithms in graphics, engineering and scientific computing. Due to their features of rapid convergence and moderate memory requirement for large-scale problems [1], the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm and its variant, the L-BFGS-B algorithm [2, 3, 4], are efficient alternatives to other frequently-used energy minimization algorithms such as the conjugate gradient (CG) [5] and Levenberg-Marquardt (LM) [6] algorithm. Furthermore, L-BFGS-B is favored as the core of many state-of-the-art algorithms in graphics, such as the computation of centroidal Voronoi tessellation (CVT) [7], the mean-shift image segmentation [8], the medical image registration [9], the face tracking for animation [10], and the composition of vector textures [11]. Among these applications, the computation of CVT is the basis of numerous applications in graphics including flow visualization [12], image compression or segmentation [13, 14, 15], surface remeshing [16, 17, 18], object distribution [19], and stylized rendering [20, 21, 22]. Hence, an L-BFGS-B solver of high performance is desired by the graphics community for its wide applications.

L-BFGS-B is an iterative algorithm. After initialized with a starting point and boundary constraints, it iterates through five phases: (1) gradient projection; (2) generalized Cauchy point calculation; (3) subspace minimization; (4) line searching; and (5) limited-memory Hessian approximation. Recently, there has been a trend towards the usage of parallel hardware such as the GPU for acceleration of energy minimization algorithms. Successful examples including the GPU-based CG [23, 24] and GPU-based LM [25] have demonstrated the clear advantages of parallelization. However, such parallelization for L-BFGS-B is challenging since there is strong dependency in some key steps, such as (2) generalized Cauchy point calculation, (3) subspace minimization, and (4) line searching. In this paper, we tackle this problem and make the following contributions:

- We approximate the generalized Cauchy point with much less calculation while maintaining a similar rate of convergence. By doing so, we remove the dependency in the computation to make the algorithm suitable for parallel implementation on the GPU.
- We propose several new GPU-friendly expressions to compute the maximal possible step-length for backtracking and line searching, making it possible to be calculated with parallel reduction.
- We demonstrate the speedup of L-BFGS-B enabled by our parallel implementation with extensive testings and present example applications to solve some typical nonlinear optimization problems in both graphics and scientific computing.

In the remainder of this paper, we first briefly review the BFGS family and optimization algorithms on the GPU in Section 2. Next, we review the L-BFGS-B algorithm in Section 3, and introduce our adaptation on the GPU in Section 4. Experimental results are given in Section 5, comparing our implementation with the latest L-BFGS-B implementation on the CPU [26] using two examples from different fields: the centroidal Voronoi tessellation (CVT) problem [7, 27] in graphics, as well as the Elastic-Plastic Torsion problem in the classical MINPACK-2 test problem set [28] in scientific computing for generality. Finally, Section 6 discusses the limitation of our GPU implementation and Section 7 concludes the paper with possible future work. Our prototype is open source and can be free downloaded from Google Code (`http://code.google.com/p/lbfgsb-on-gpu/`).

2. Related Work

We briefly review the previous work on Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm and its extensions, as well as previous work on GPU-based nonlinear optimization.

2.1. BFGS Optimization

The BFGS algorithm [29] approximates the Newton method for solving several nonlinear optimization problems. Since the memory requirement quadratically increases with the problem size, the BFGS algorithm is not suitable for large scale problems. The seminal work by Liu and Nocedal [2] approximates the Hessian matrix with reduced memory requirement which is linear in the size of the input variables. Their method is called the *L-BFGS* algorithm, where "L" stands *limited memory*. In addition, a bound constrained version of the L-BFGS algorithm, namely the *L-BFGS-B algorithm*, is proposed by Byrd et al. [3], and its implementation in Fortran is given by Zhu et al. [4].

Furthermore, there are some variants [30, 31, 32, 33] that propose improvements by combining the L-BFGS algorithm with other optimization methods. Recently, Morales et al. [34, 26] improve (currently in version 3.0) the step of subspace minimization through a numerical study. We build our prototype based on their code, using the techniques detailed in the next section to parallelize it on the GPU.

2.2. Nonlinear Optimization on GPU

The work proposed by Bolz et al. [24] for the first time mapped two computational kernels of nonlinear optimization on the GPU, specifically, a sparse matrix conjugate gradient solver and a regular grid multi-grid solver. Since then, the topic on how to map the conjugate gradient solver efficiently on the GPU has been extensively studied. Hillesland et al. [35] described a framework with conjugate gradient method for solving many large nonlinear optimizations concurrently on the graphics hardware, which was applied to image-based modeling. Goodnight et al. [36] introduced a multi-grid solver for boundary value problems on the GPU. Krüger and Westermann [37] also proposed some basic operations of linear algebra on the GPU and used them to construct a congruent gradient solver and a Gauss-Seidel solver. Later, Feng and Li [38] implemented a multi-grid solver on the GPU for power grid analysis and Buatois et al. [39] presented a sparse linear solver on the GPU. Recently, in the community of parallel computing the conjugate gradient method has been proposed on multi-GPU [23, 40, 41] or even multi-GPU clusters [42]. A complete survey on this topic is beyond the scope of this paper. Please refer to Verschoor's paper [43] for more details.

Besides the conjugate gradient method, Li et al. [25] described a GPU accelerated Levenberg-Marquardt optimization. There are also some previous attempts on parallelizing L-BFGS method and its variants (e.g. L-BFGS-B method) on the GPU. Yatawatta et al. [44] implemented GPU-accelerated Levenberg-Marquardt and L-BFGS optimization routines. They used a hybrid approach where only the evaluation of the target function and its gradients are implemented on the GPU, and the rest of the optimization work is still on the CPU. They used a hybrid approach where only the evaluation of the target function and its gradients are implemented on the GPU, and the rest of the optimization work is still on the CPU. There are also some other works [27, 46] followed a similar style. None of them implemented the core parts of the optimization (line searching, subspace minimization, etc.) on the GPU. Wetzl et al. [45] introduced a straightforward implementation of the L-BFGS algorithm on the GPU where the boundaries are ignored, which made their implementation unavailable for problems with constraints. As far as we know, our method presented in this paper will be the first method running all the core parts of the L-BFGS-B optimization on the GPU, except for some high-level branching logic control.

3. Algorithm

The L-BFGS-B algorithm is introduced by Byrd et al. [3]. We follow the notation in their paper to briefly introduce the algorithm in this section.

The L-BFGS-B algorithm is an iterative algorithm that minimizes an objective function $\mathbf{x}$ in R^n subject to some boundary constraints $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$, where $\mathbf{l}, \mathbf{x}, \mathbf{u} \in R^n$. In the k-th iteration, the objective function is approximated by a quadratic model at a point $\mathbf{x}_k$:

$$m_k(\mathbf{x}) = f(\mathbf{x}_k) + \mathbf{g}_k^T(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T \mathbf{B}_k(\mathbf{x} - \mathbf{x}_k), \qquad (1)$$

where $\mathbf{g}_k$ is the gradient at point $\mathbf{x}_k$ and $\mathbf{B}_k$ is the limited memory BFGS matrix which approximates the Hessian matrix at point $\mathbf{x}_k$. In each iteration, the most crucial phases are: (1) the computation for the generalized Cauchy point; and (2) the subspace minimization.

3.1. Generalized Cauchy Point

To simplify notation, following [3], we shall drop the index k of the outer iteration in the rest of this section. Thus, $\mathbf{B}$, $\mathbf{g}$, $\mathbf{x}$, and $\hat{m}$ correspond to $\mathbf{B}_k$, $\mathbf{g}_k$, $\mathbf{x}_k$, and $\hat{m}_k$ used above. Subscripts will be used to denote the components of a vector, and superscripts to denote iteration during the search for the generalized Cauchy point. To minimize $m_k(\mathbf{x})$ in Eqn. 1, the generalized Cauchy point $\mathbf{x}^c = \mathbf{x}(t^*)$ is computed as the first local minimizer t^* along a piece-wise linear path $P(t) = (\mathbf{x}^0 - t\mathbf{g}; \mathbf{l}, \mathbf{u})$ that is to be described below.

Each coordinate $x_i(t)$ of the piecewise linear path $\mathbf{x}(t)$ is defined as

$$x_i^0 - tg_i, \quad t \in [0, t_i] \qquad (2)$$

where the breakpoint t_i in each dimension, which is the bound induced by the rectangular bounding region $(\mathbf{l}, \mathbf{u})$, is given by

$$t_i = \begin{cases} (x_i^0 - u_i)/g_i & \text{if } g_i < 0 \\ (x_i^0 - l_i)/g_i & \text{if } g_i > 0 \\ \infty & \text{otherwise} \end{cases} . \qquad (3)$$

The breakpoints $\{t_i : i = 1, \ldots, n\}$ are sorted into an ordered set $\{t^j : t^{j-1} < t^j, j = 2, \ldots, n\}$. To find the minimizer

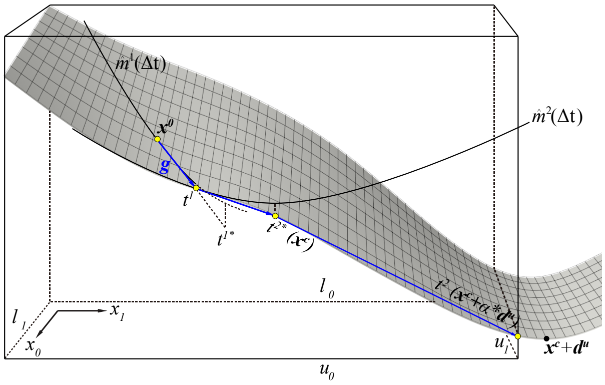


Figure 1: Example of the L-BFGS-B Optimization on 2D domain. The surface represents an energy function constrained by box boundaries.

t^*, the intervals $[t^{j-1}, t^j]$ are sequentially examined until a local minimizer t^* of the objective function $\mathbf{x}$ within the interval is found (i.e. t^* is the first of the ordered set of local minimizers $\{t^{j^*} : t^{j^*} \in [t^{j-1}, t^j]\}$). In each interval, the curve for the quadratic model $m(\mathbf{x}(t))$ can be written in $\Delta t = t - t^{j-1}$ as:

$$\hat{m}^j(\Delta t) = f^j + f^{j\prime}\Delta t + \frac{1}{2} f^{j\prime\prime}\Delta t^2, \tag{4}$$

where $f^j = f(\mathbf{x}^0) + \mathbf{g}^T(\mathbf{x}^j - \mathbf{x}^0) + \frac{1}{2}(\mathbf{x}^j - \mathbf{x}^0)^T B(\mathbf{x}^j - \mathbf{x}^0)$, and $f^{j\prime} = \mathbf{g}^T\mathbf{d}^j + \mathbf{d}^{j^T}\mathbf{B}(\mathbf{x}^j - \mathbf{x}^0)$ ($d_i^j = -g_i$ if $t^j < t_i$ or $d_i^j = 0$ otherwise) and $f^{j\prime\prime} = \mathbf{d}^{j^T}\mathbf{B}(\mathbf{x}^j - \mathbf{x}^0)$ are the first and second order directional derivatives of the one dimensional quadratic at point $\mathbf{x}(t^j)$. Then the minimizer is computed as $t^{j^*} = t^{j-1} - f^{j\prime}/f^{j\prime\prime}$.

To be concrete, we propose Figure 1 for an illustration in a 2D domain. In this example, the generalized Cauchy point $\mathbf{x}^c$ is acquired after searching two intervals, where the minimizer t^{1^*} is discarded due to it is not within $[0, t^1]$, and the minimizer t^{2^*} is accepted since it is in the field $[t^1, t^2]$ marked by the boundary constraints. ($\hat{m}^j(\Delta t)$ is the curve for the quadratic model $m(\mathbf{x}(t))$ in interval $[t^{j-1}, t^j]$. Dotted line means out of the feasible region.)

3.2. Subspace Minimization

After the generalized Cauchy point is obtained, the quadratic function $m_k(\mathbf{x})$ is minimized for the free variables in $\mathbf{x}^c$, i.e. variables whose values are not at lower bound or upper bound. To solve this minimizing problem, a direct primal method based on the Sherman-Morrison-Woodbury formula is used to find a solution vector $\hat{\mathbf{d}}^u$ in the subspace, which gives the minimizer $\bar{\mathbf{x}}_{k+1}$. To backtrack the solution into the feasible region defined by the boundary constraints, a positive scalar α^* is found by a line search as the maximal possible distance of movement along the search direction $\mathbf{d}_k = \bar{\mathbf{x}}_{k+1} - \mathbf{x}_k$:

$$\alpha^* = \max(\alpha : \alpha \leq 1, l_i \leq x_i^c + \alpha \hat{d}_i^u \leq u_i, i \in \mathcal{F}) \tag{5}$$

where $\mathcal{F}$ is a set composed of indices corresponding to the free variables in $\mathbf{x}^c$. The backtracked solution $\hat{\mathbf{d}}^* = \alpha^* \hat{\mathbf{d}}^u$ gives the new point $\mathbf{x}_{k+1}$ for next iteration. This procedure is repeated until certain convergence condition is satisfied. Figure 1 shows a 2D example of this procedure: after $\mathbf{x}^c$ is obtained, variable x_0 is fixed due to x_0^c is at the upper boundary u_0, and the only free variable is x_1. So the maximal possible step-length α^* makes $\mathbf{x}^c + \alpha^*\mathbf{d}^u$ exactly at the upper boundary u_1.

4. Our modifications

In the following, we explain our modifications for finding the generalized Cauchy point and subspace minimization, which make the L-BFGS-B algorithm suitable for current GPU architecture.

4.1. Approximate Generalized Cauchy Point

The inherently sequential searching for the first local minimizer in the original method is quite hostile to the GPU, especially for problems with high dimensions. We observe that in many practical applications, the first local minimizer is either maintained at the value obtained for the first interval $t^{1^*} = -f_1'/f_1''$, or very close to the upper bound t^1 of the first interval $[0, t^1]$. So we simplify the choice of the first minimizer by approximate the generalized Cauchy point $\mathbf{x}^c = \mathbf{x} + t^c\mathbf{g}$ by $t^c = \max(0, \min(t^1, t^{1^*}))$. That is, if the first local minimizer is located in the first interval $[0, t^1]$, we find the exact generalized Cauchy point; otherwise, the generalized Cauchy point is approximated by t_1 in the subsequent computation.

We examine the differences made by the above approximation we introduced with the eight minimization problems in the MINPACK-2 problem set [28], list the results in Table 1. Here we compare the difference between our approximated minimizer t^c and the minimizer computed in the original L-BFGS-B program t^*. The differences in all iterations are categorized and listed as percentages. As we can see, t^c and t^* are the same in more than 85% iterations for problems except the Journal Bearing, and $|t^c - t^*|/t^*$ is less than 5% for more than 90% iterations for all problems. In addition, even if there are some differences for the generalized Cauchy point, there is no significant difference in the final energy values. This demonstrates the efficacy of our approximation scheme.

Since $t^1 = \min\limits_i(t_i)$, it can be computed by a minimal parallel reduction [47]. The directional derivatives f' and f'' are computed in a similar way to the implementation on the CPU, where the dot products and matrix-vector multiplications are calculated using parallel reductions (see Section 4.3 for details).

4.2. Backtracking and Line Search

The original L-BFGS-B implementation uses a sequential searching to find the positive scalar α^* for backtracking, which cannot easily be adapted to the GPU. Here we observe that expression 5 can be evaluated first by computing the maximal possible value individually in each dimension (denoted as α_i below) and then using the minimal value among them as the intersect of the boundary constraints. That is, we first compute α_i as follows,

$$\alpha_i = \begin{cases} (l_i - x_i^c)/\hat{d}_i^u & \hat{d}_i^u < 0, \\ (x_i^c - u_i)/\hat{d}_i^u & \hat{d}_i^u > 0. \end{cases} \tag{6}$$

$\frac{\lvert t^c - t^*\rvert}{t*} \times 100\%$	Elastic-Plastic Torsion	Journal Bearing	Minimal Surfaces	Optimal Design	1-D Ginzburg-Landau	Lennard-Jones Clusters	Stead-State Combustion	2-D Ginzburg-Landau
0	85.23%	55.14%	100.00%	86.41%	100.00%	100.00%	100.00%	98.70%
0~5%	12.19%	35.26%	0.00%	8.99%	0.00%	0.00%	0.00%	1.30%
5%~10%	1.41%	3.40%	0.00%	1.10%	0.00%	0.00%	0.00%	0.00%
10%~15%	0.35%	1.00%	0.00%	0.80%	0.00%	0.00%	0.00%	0.00%
15%~20%	0.00%	0.70%	0.00%	0.30%	0.00%	0.00%	0.00%	0.00%
20%~25%	0.35%	0.60%	0.00%	0.40%	0.00%	0.00%	0.00%	0.00%
25%~30%	0.12%	0.30%	0.00%	0.20%	0.00%	0.00%	0.00%	0.00%
30%~35%	0.00%	0.50%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
≥40%	0.35%	3.10%	0.00%	1.80%	0.00%	0.00%	0.00%	0.00%
Energy Diff	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Table 1: Comparison of our approximated t^c and real t^* in L-BFGS-B algorithm for all the eight minimization problems in MINPACK-2. The size of all problems is 40,000, and the boundary is $[-1, 1]$ for all dimensions. "Energy Diff" means the difference of final energy between using the original generalized Cauchy point and our approximated one.

Then a minimal parallel reduction [47] is performed,

$$\alpha^* = \min(1, \min_i(\alpha_i)) \tag{7}$$

The last step in each iteration of the L-BFGS-B algorithm utilizes a line search to find a new point for next iteration, which can similarly be computed with a minimal parallel reduction.

4.3. *Implementation Details*

We implement our GPU-based L-BFGS-B algorithm using NVIDIA CUDA language [48]. The first problem to be handled is how to operate between the matrices and vectors, which is pervasive in the algorithm, especially when calculating the infinity normal of the projected gradient, the inverse L-BFGS matrix and its pre-conditioners.

For large-scale problems, the number of columns of inverse L-BFGS matrix and its pre-conditioners can be much larger than their number of rows (normally m is between 3 and 8), where m is the maximum dimension of the Hessian approximation). This kind of matrices are often called "panels" or "multi-vectors" in the literatures [49, 50], and it is common to solve the "panel-panel" multiplication [49, 50] using dot-products, which is implemented using the latest parallel reduction technique [47] in our implementation. Take the matrices in Fig. 2 for an example. To calculate this value, in each thread we sample the values from both the left and the right matrix, multiply them (and other calculations if necessary, such as scaling, adding or dividing by an extra value, etc.) and store the result in the shared memory. Then a parallel reduction is performed across the shared memory and the value from the thread whose index is zero is stored in the resulting matrix.

We have also tested NVIDIA CUBLAS Library [51] for this "panel-panel" multiplication, and it proved less efficient than our method (Fig. 3), since it has not been optimized for the matrices of such special dimensions. The dimension of the two matrices in this test are $8 \times M$ and $M \times 8$, where M varies from 10 to over $1,000,000$. The initial values of elements in these two matrices are random numbers in $[0, 1]$. All the computations are in the double precision and the difference between the

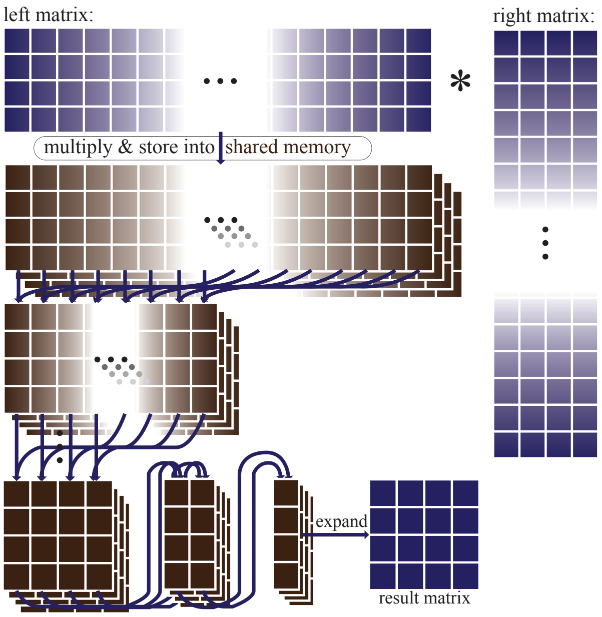


Figure 2: Matrix-matrix multiplication using parallel reduction.

results of our method and of CUBLAS is less than $1E-9$. Although cublasDgemm is faster than our implementation when $M = 10$ (when the matrices are almost square), our implementation outperforms it for several times as soon as $M \geq 20$.

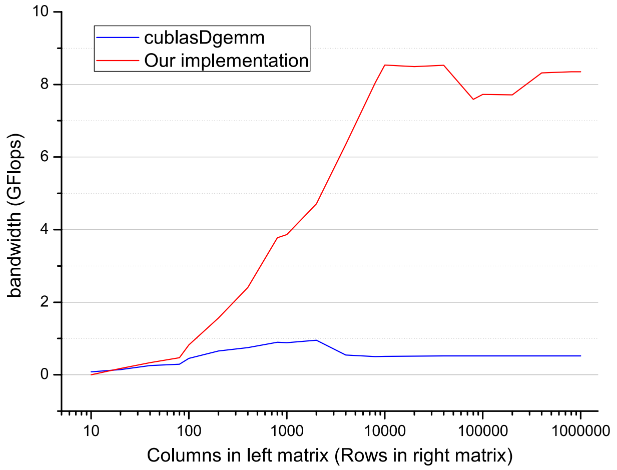


Figure 3: Comparison of throughput for matrix multiplication using CUBLAS Library (the *cublasDgemm* function call) and our method.

According to the literature from Volkov and Demmel [52], in the CUBLAS Library, a matrix is generally treated, and divided into blocks, whose result is accumulated by cycling 16× between the rows internally, resulting in a lower parallelism; also for more synchronization they needs more updates across multiple blocks. This fact may contribute to the result that our implementation can be 16× faster when the number of variables becomes larger than 5,000, which is shown in Fig. 3. We have also experimented the CUSPARSE Library and found that it performed similar as CUBLAS. The reason is that all the matrices in L-BFGS-B are dense. Hence a sparse matrix solver can hardly demonstrate its power.

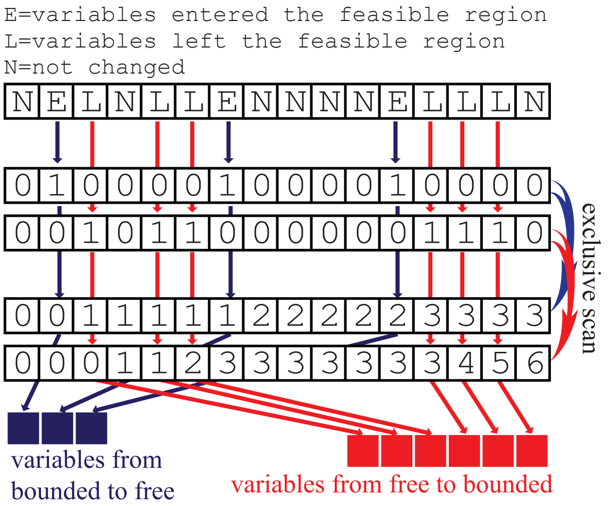

E=variables entered the feasible region
L=variables left the feasible region
N=not changed
N E L N L L E N N N N E L L L N
0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
0 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0
0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3
0 0 0 1 1 2 3 3 3 3 3 3 3 4 5 6
exclusive scan
variables from bounded to free
variables from free to bounded

Figure 4: Managing the status of variables using parallel scan.

In the L-BFGS-B algorithm, the steepest descent direction is projected onto a feasible region defined by the boundary constraints. Variables whose value at the lower or upper boundaries of this region are held fixed. The set of these variables is called “active set” [3] indicating the corresponding boundary constraints are active. Before the Hessian approximation, the status of variables have to be traced to see whether they entered or left the active set. Instead of searching sequentially for all the variables, we first mark variables that entered or left the active set into two arrays, and then perform a parallel scan [53] across each array to transform the boolean marks into sequential indices. Finally, we select the variable as it is marked and put them in positions determined by their indices (this operation is often called “compact” [53]). We use the implementation from the Thrust Library [54] for the parallel scan. This process is illustrated in Fig. 4.

The L-BFGS-B algorithm needs Cholesky factorization to compute $\hat{\mathbf{d}}^u$ for subspace minimization [26]. We use Henry’s code [55] to compute Cholesky factorization. Other linear algebra operations such as solving triangular system are performed using the CUBLAS Library [51].

5. Applications

We compare the efficacy and robustness of our GPU-based L-BFGS-B algorithm and the original CPU-based L-BFGS-B algorithm using two applications described below. All experiments were performed with an Intel Xeon W5590 @ 3.33GHz and an NVIDIA GTX 580 in double precision. The CUBLAS Library and the Thrust Library used are included in CUDA Toolkit version 4.2.

5.1. GPU-based Centroidal Voronoi Tessellation

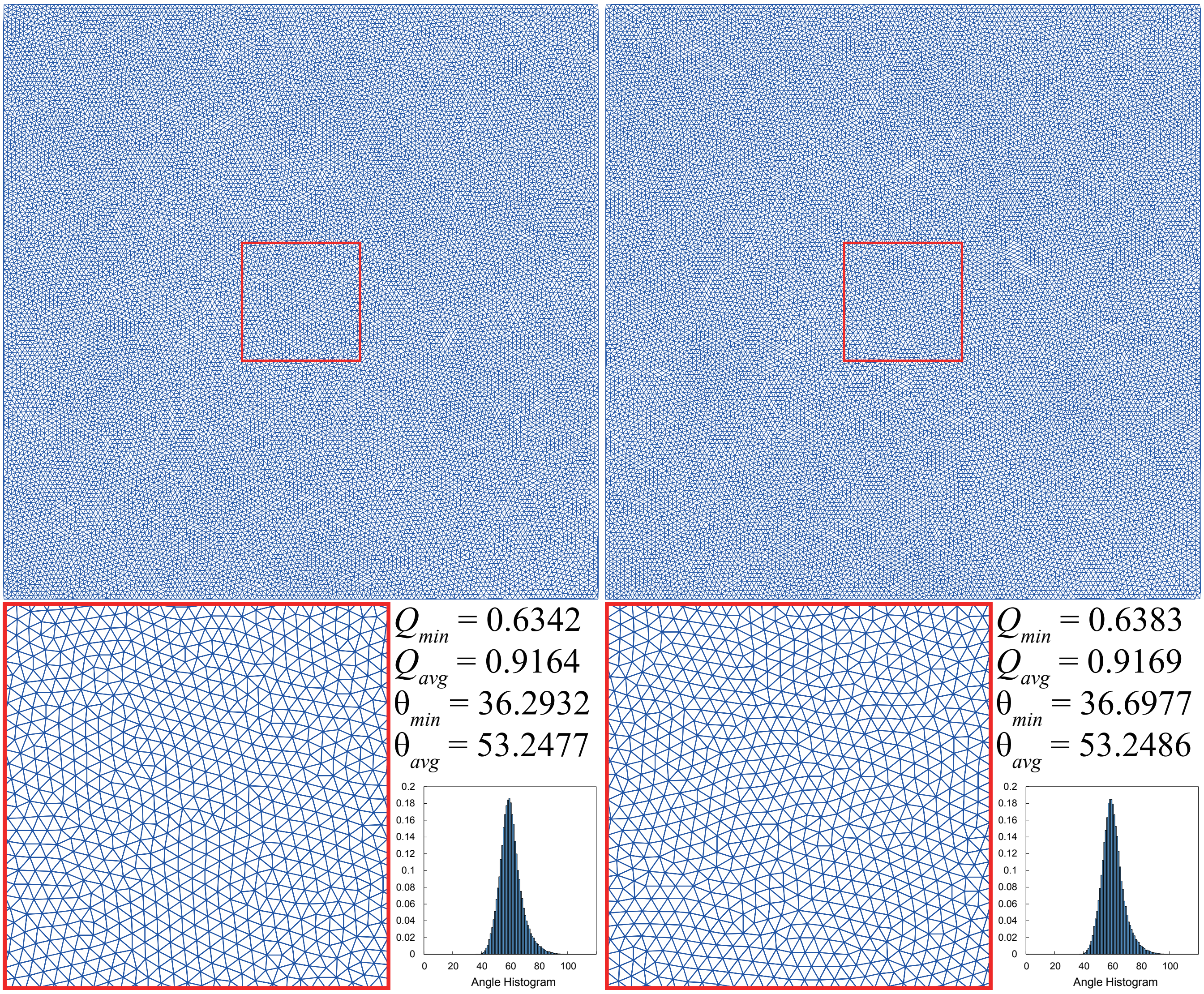
To explore the power of our GPU implementation in graphics, we experimented our L-BFGS-B method on the centroidal Voronoi tessellation (CVT) problem. For the CVT, it is already shown in [27] how to evaluate CVT energy function and compute its gradient on the GPU. However, the L-BFGS-B iterations are still performed on the CPU in [27]. In the following, we will show how to perform L-BFGS-B iterations on the GPU as well, and observe resulting the performance gain.

Centroidal Voronoi tessellation requires minimizing the following CVT function [7]:

$$F(\mathbf{X}) = \sum_{i=1}^{n} \int_{\Omega_i} \rho(\mathbf{x}) \|\mathbf{x} - \mathbf{x}_i\|^2 \, d\sigma, \qquad (8)$$

where $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ is an ordered set of n point sites, Ω_i is the Voronoi cell of site $\mathbf{x}_i$, and $\rho(\mathbf{x})$ is a density function at $\mathbf{x}$.

We use the code from [27] for Voronoi tessellation on the GPU (VTGPU for short in the following) with both CPU L-BFGS-B and our GPU L-BFGS-B implementations. We also transplant all the new “tweaks” to L-BFGS-B we made on the GPU to its implementation on the CPU, such as using an approximate generalized Cauchy point, maximizing the dimension of Hessian approximation, and maximizing the step length in each iteration. This is fair treatment because it has been observed that these “tweaks” make L-BFGS-B faster and simpler



(a) CPU L-BFGS-B

(b) GPU L-BFGS-B

Figure 5: Delaunay triangulations of 20, 000 vertices. In the bottom figure only a part of them are demonstrated. The results are generated from CVT using VTGPU with either CPU and GPU L-BFGS-B iterations. θ is the smallest angle in a triangle; "min"=the minimal value of all triangles in the mesh; "avg"=the average value of all triangles in the mesh.

Size	Resolution	Evaluation	CPU L-BFGS-B		GPU L-BFGS-B		spd/itr	Energy Difference
			itr	L-BFGS-B (I/O)	itr	L-BFGS-B		
60,000	2,048	44.97	28	112.07 (12.81)	28	4.06	27.60×	0
40,000		43.96	34	74.18 (9.19)	34	3.18	23.33×	0
20,000		42.43	45	30.09 (4.05)	55	2.19	13.74×	-2.31E-08
10,000		41.92	69	15.48 (2.27)	69	1.71	9.05×	0
8,000		41.87	76	12.61 (1.92)	72	1.60	7.88×	-3.38E-08
6,000		41.84	89	11.2 (1.53)	71	1.53	7.32×	3.79E-07
4,000		41.38	106	6.51 (1.18)	106	1.40	4.65×	0
4,000	1,024	8.85	48	6.21 (1.12)	48	1.33	4.67×	0
2,000		8.76	61	3.37 (0.74)	61	1.24	2.72×	0
1,000		8.70	74	1.81 (0.51)	74	1.16	1.56×	0
500	512	2.38	56	1.02 (0.38)	56	1.13	0.90×	0
200		2.39	53	0.65 (0.38)	59	1.09	0.60×	-4.79E-05

Table 2: Statistics of VTGPU with the original CPU L-BFGS-B implementation and our GPU L-BFGS-B implementation. All the timings are for each iteration and in milliseconds. "itr" is the number of iterations; "Evaluation" is the time spent on function and gradient evaluation using the VTGPU algorithm; "L-BFGS-B" is the time spent on each L-BFGS-B iteration, including the time spent on data exchanging (denoted by "I/O"); "spd/itr" is the speed-up of L-BFGS-B in each iteration.

without compromising its convergence rate. In both implementations, we stop the iteration when the decrement of energy is less than $1E-64$. Our experimental results are summarized in Table 2. We record the final CVT energy, iterations required, and timing data for different parts of both methods, to be detailed in the following. A comparison of the total costs per iteration between [27] and ours is shown in Fig. 8.

5.1.1. Jump Flooding Algorithm

The VTGPU uses the jump flooding algorithm (JFA) [56, 57, 58] to compute the Voronoi diagram on the GPU. The timing of this stage is labeled as "Evaluation" in Table 2. Because both competitors use the same code for this stage, their iteration-wise timings are generally the same.

5.1.2. Data Exchange

Using VTGPU with the CPU L-BFGS-B also requires data exchanging between the device and host memory because the computation of CVT energy and its gradient is on the GPU. We mark the time spent on this communication as "I/O" in Table 2. It increases linearly with both the number of sites and the number of iterations. The data exchange costs may occupy more than 20% of the L-BFGS-B time on average, which is totally saved in the GPU implementation.

5.1.3. L-BFGS-B Optimization

The time spent on the optimization is the main part that made the difference in the comparison. We break down this part into the different columns. The number of iterations is recorded in the "itr" column of Table 2. The GPU version generally requires similar iterations to the CPU version. Some exceptions are due to the different definitions of double precision between the CPU and the GPU.

The time per iteration of the L-BFGS-B algorithm is recorded in the "L-BFGS-B" column of Table 2, with the speed-up recorded in the "L-BFGS-B" of the "Speed-up" column. Time of the two implementations increases linearly with the number of sites, however, the CPU version grows much faster — about 20× more than the GPU implementation. Fig. 6 compares the two implementations, where the slope of CPU L-BFGS-B is $1.61E-3$ and the slope of GPU L-BFGS-B is $8.03E-5$.

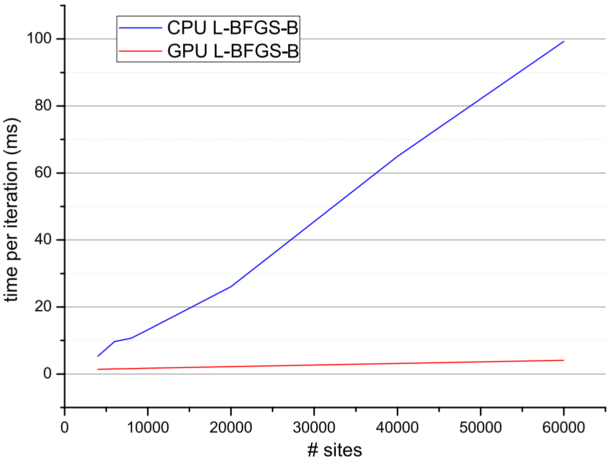


Figure 6: Performance comparison for L-BFGS-B iterations in CVT problem.

We also compare our generalized Cauchy point approximation with the CPU implementation without the approximation. The results are listed in Table 3. It is clear that our approximation is satisfying for the CVT problem.

5.1.4. Convergence

We compute the difference between final energies from both methods, and record the $F_{GPU} - F_{CPU}$ in the "Energy Difference" column of Table 2. The result of the GPU version is mostly the same as the result of the CPU version. Occasionally the two implementations generate different final energies, which indicates that different local minimum points are reached. To visually demonstrate this statement, we compare in Fig. 5 the two Delaunay triangulations, which are dual to the CVTs, with $20,000$ sites, as well as their quality measurements. Here the quality of a triangle is measured by $Q = 2\sqrt{3}S/ph$ [59],

$\frac{\lvert t^c - t^* \rvert}{t*} \times 100\%$	resolution 512		resolution 1,024			resolution 2,048						
	200	500	1000	2000	4000	4000	6000	8000	10000	20000	40000	60000
0	100.00%	98.39%	100.00%	100.00%	98.41%	99.07%	99.14%	98.72%	98.61%	96.43%	97.37%	97.06%
0~5%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
5%~10%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.28%	0.00%	1.79%	2.63%	0.00%
10%~15%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
15%~20%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.39%	0.00%	0.00%	0.00%
20%~25%	0.00%	1.61%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
25%~30%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
30%~35%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.86%	0.00%	0.00%	0.00%	0.00%	0.00%
≥40%	0.00%	0.00%	0.00%	0.00%	1.59%	0.93%	0.00%	0.00%	0.00%	1.79%	0.00%	2.94%

Table 3: Comparison of our approximated t^c and real t^* in L-BFGS-B algorithm for VTGPU with different sites and different resolutions. The first row in the head shows resolutions and the second row shows site numbers.

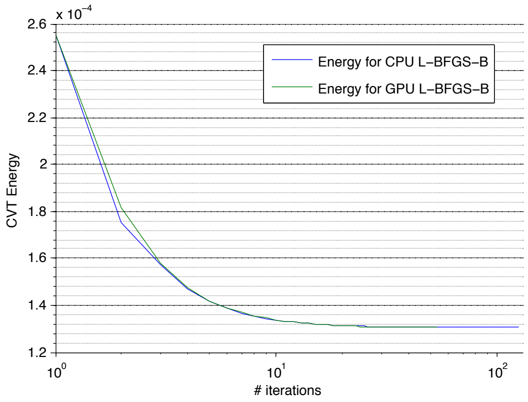


Figure 7: During the optimization, the CVT energy changes likewise by using CPU and GPU. The figure is generated in the same setting as Fig. 5.

where S is the area, p is the half-perimeter, and h is the length of the longest edge. It is clear that the two implementations generate final results of similar quality. Our GPU implementation can also guarantee a stable convergence, as shown in Fig. 7, where the energies from both methods will finally decrease to the same level, and only slightly differ during the optimization process.

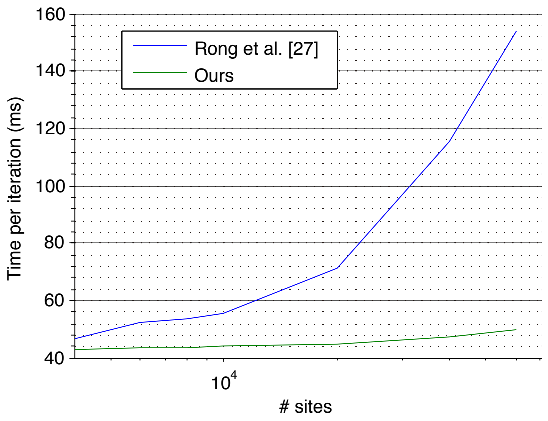


Figure 8: Performance comparison (total time in each iteration) between [27] and ours.

5.2. Elastic-Plastic Torsion in MINPACK-2

For generality, we also evaluate our implementation by solving the Elastic-Plastic Torsion problem in the classical MINPACK-2 problem set [28], which is a representative optimization problem in scientific computing. In this problem the elastic-plastic stress potential in an infinitely long cylinder is determined when torsion is applied, which is equivalent to that of minimizing a complementary energy q on a square feasible region D [60]:

$$q(v) = \int_D (\frac{1}{2} \|\nabla v(x)\|^2 - cv(x))dx. \tag{9}$$

We use the same Fortran code on the CPU for evaluating q and gradient $\nabla v(x)$, but use two versions, specifically, our GPU implementation and the original CPU implementation [26] for the L-BFGS-B iterations. Comparisons are presented in Table 4. We solve the 2D problem where the two dimensions are set equivalent and their multiplication is the "Size" of the problem. We evaluate from four hundred to four million variables, using $c = 5.0$ (recommended by [60]) for the constant c in (9).

The number of iterations required by our implementation on the GPU ("itr" columns) is similar to that of the CPU implementation. Besides, the effectiveness of our implementation has outperformed the CPU implementation in each iteration ("L-BFGS-B" columns), beginning at size=6,400. While yielding nearly the same final energy value, our implementation requires much less time. Fig. 9 compares the time per iteration of different problem sizes for GPU and CPU implementations. We can see that the curve for CPU implementation increases roughly 29× faster than the GPU implementation with the increase of the problem size; here we treat the curvature as linear dependence, the slope of the CPU implementation is $8.75E - 4$ while the slope of the GPU implementation is $3.03E - 5$. The speedup per iteration is recorded in the "spd/itr" column.

From the statistical data, we observe again the efficacy of our approximation to the generalized Cauchy point. The average step length for computing the generalized Cauchy point is listed in the "t^*" and "t^c" columns. Clearly, the values for both methods are quite similar but our approximation brings about considerable performance gain. Furthermore, our new strategy may even reduce the number of iterations (c.f. "itr" column) with an equivalent final energy in some cases. Due to the different definitions of double precision between CPU and GPU, there may be some energy differences. Nevertheless, they are

Size	Evaluation	CPU			GPU			spd/itr	Energy Difference
		t^*	itr	L-BFGS-B	t^c	itr	L-BFGS-B (I/O)		
4,000,000	141.09	0.09	4197	3351.68	0.10	3971	137.16 (17.92)	24.44×	5.88E-11
2,250,000	79.62	0.09	2917	2200.41	0.10	3213	78.58 (9.97)	28.00×	-4.52E-12
1,440,000	51.29	0.07	2915	1473.86	0.09	2636	51.50 (6.45)	28.62×	8.78E-12
1,000,000	36.07	0.08	2310	782.62	0.08	1907	36.61 (4.30)	21.38×	7.86E-12
640,000	23.12	0.07	1685	427.75	0.08	1793	24.23 (2.70)	17.65×	3.98E-12
360,000	13.11	0.08	1272	220.40	0.09	1315	15.27 (1.57)	14.43×	2.44E-12
160,000	5.91	0.17	921	117.74	0.18	1035	8.17 (0.63)	14.41×	-1.85E-12
40,000	1.53	0.24	549	15.09	0.24	464	3.27 (0.20)	4.61×	-1.91E-13
10,000	0.39	0.26	241	3.59	0.26	237	1.92 (0.19)	1.87×	3.00E-14
6,400	0.25	0.26	193	2.29	0.26	199	1.60 (0.08)	1.43×	-6.90E-14
3,600	0.14	0.26	144	1.34	0.27	155	1.50 (0.08)	0.89×	1.20E-14
1,600	0.06	0.26	133	0.93	0.26	105	1.41 (0.05)	0.66×	-9.99E-16
400	0.02	0.26	52	0.20	0.25	49	1.17 (0.04)	0.17×	-9.99E-15

Table 4: Statistics for the Elastic-Plastic Torsion problem. All timing data are in milliseconds. “Evaluation” is the time spent on function and gradient evaluation; t^* and t^c are average step lengths for computing the generalized Cauchy point; “itr” is the number of iterations; “L-BFGS-B” is the time spent on each L-BFGS-B iteration, including the time spent on data exchanging (denoted by “I/O”); “spd/itr” is the speed-up of L-BFGS-B in each iteration.

no more than $5.88E-11$, and sometimes the energy computed by our implementation is lower than that produced by the original implementation (c.f. negative terms in the “Energy Difference” column)

In both experiments, we use a stop criterion less than $1E-64$, that is, we run the experiment until the amount of energy decreased is less than $1E-64$. Note that in this experiment, since the computation of the energy function value and its gradients are still on the CPU, we need to transfer data between the CPU and the GPU in each iteration. Even with this overhead, the GPU implementation is still faster than the CPU one.

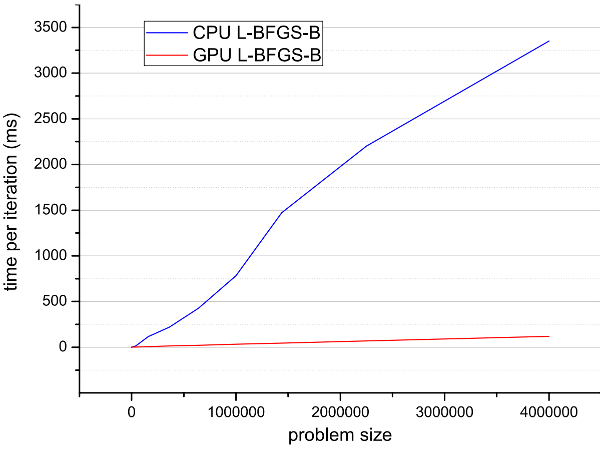


Figure 9: Performance comparison for L-BFGS-B iterations in Elastic-Plastic Torsion problem.

6. Limitations

Currently, the performance of our method is limited by the memory bandwidth between the global video memory and the on-chip memory (shared memory, registers, etc.). We have also tested our implementation on a Tesla C2050. Although the Tesla C2050 has a much higher peak performance on the calculation in double precision (515GFlops) than the GTX580 (193GFlops), its performance on running our GPU L-BFGS-B algorithm is lower. More specifically, the ratio of the performance of the two cards is exactly the ratio of their memory bandwidth (144GB/sec. vs. 192.4GB/sec.), indicating the memory bandwidth is the bottleneck. Besides, our method still needs to read a few scalars from the GPU to the CPU in some stages, to control the high-level branching logic in the L-BFGS-B algorithm. A solution to these problems is to divide the variables into segments, calculate for each, and then combine. With this strategy, instead of cycling between stages, one can pack all the iterations into a single kernel where the global memory is accessed only at the beginning and at the end of the algorithm. However, this solution requires the evaluation of the function, as well as the calculation of the gradient vector, is divisible, which is obviously not available to many optimization problems.

7. Conclusion and Future Work

In this paper, we presented the first parallel implementation of the L-BFGS-B algorithm on the GPU. Our experiments show that our approach makes the L-BFGS-B algorithm GPU-friendly and easily parallelized, so the time spent on solving large-scale optimizations is radically reduced. Future work includes breaking the bottleneck of memory bandwidth and exploring the parallelism of L-BFGS-B on multiple GPUs or even clusters for problems of larger scales.

References

[1] ALGLIB Project . Unconstrained optimization: L-BFGS and CG. 2013. `http://www.alglib.net/optimization/lbfgsandcg.php#header3`.

[2] Liu DC, Nocedal J. On the limited memory BFGS method for large scale optimization. Math Program 1989;45(1):503–28.

[3] Byrd RH, Lu P, Nocedal J, Zhu C. A limited memory algorithm for bound constrained optimization. SIAM J Sci Comput 1995;16(5):1190–208.

[4] Zhu C, Byrd RH, Lu P, Nocedal J. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. ACM Trans Math Softw 1997;23(4):550–60.

[5] Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. 1952.

[6] Marquardt DW. An algorithm for least-squares estimation of nonlinear parameters. SIAM J Soc Ind Appl Math 1963;11(2):431–41.

[7] Liu Y, Wang W, Lévy B, Sun F, Yan D, Lu L, et al. On centroidal Voronoi tessellation – energy smoothness and fast computation. ACM Trans Graph 2009;28(4):101.

[8] Yang C, Duraiswami R, DeMenthon D, Davis L. Mean-shift analysis using quasi-Newton methods. In: Proceedings of ICIP '03; vol. 2. IEEE; 2003, p. II–447.

[9] Chen YW, Xu R, Tang SY, Morikawa S, Kurumi Y. Non-rigid MR-CT image registration for MR-guided liver cancer surgery. In: Proceedings of ICME '07. IEEE; 2007, p. 1756–60.

[10] Hyneman W, Itokazu H, Williams L, Zhao X. Human face project. In: ACM SIGGRAPH '05 Courses. ACM; 2005, p. 5.

[11] Wang L, Zhou K, Yu Y, Guo B. Vector solid textures. ACM Trans Graph 2010;29(4):86.

[12] Du Q, Wang X. Centroidal Voronoi tessellation based algorithms for vector fields visualization and segmentation. In: Proceedings of Vis '04. IEEE; 2004, p. 43–50.

[13] Du Q, Faber V, Gunzburger M. Centroidal Voronoi tessellations: Applications and algorithms. SIAM Rev 1999;41(4):637–76.

[14] Du Q, Gunzburger M, Ju L, Wang X. Centroidal Voronoi tessellation algorithms for image compression, segmentation, and multichannel restoration. J Math Imaging Vis 2006;24(2):177–94.

[15] Wang J, Ju L, Wang X. An edge-weighted centroidal Voronoi tessellation model for image segmentation. IEEE Trans Image Process 2009;18(8):1844–58.

[16] Alliez P, De Verdire E, Devillers O, Isenburg M. Isotropic surface remeshing. In: Proceedings of SMI '03. 2003, p. 49–58.

[17] Du Q, Wang D. Anisotropic centroidal Voronoi tessellations and their applications. SIAM J Sci Comput 2005;26(3):737–61.

[18] Lévy B, Liu Y. L_p centroidal Voronoi tessellation and its applications. ACM Trans Graph 2010;29(4):119.

[19] Hiller S, Hellwig H, Deussen O. Beyond stippling methods for distributing objects on the plane. Comput Graph Forum 2003;22(3):515–22.

[20] Secord A. Weighted Voronoi stippling. In: Proceedings of NPAR '02. ACM; 2002, p. 37–43.

[21] Battiato S, Di Blasi G, Farinella GM, Gallo G. Digital mosaic frameworks – an overview. In: Comput. Graph. Forum; vol. 26. Wiley Online Library; 2007, p. 794–812.

[22] Deussen O, Isenberg T. Halftoning and stippling. In: Image and Video-Based Artistic Stylisation. Springer; 2013, p. 45–61.

[23] Cevahir A, Nukada A, Matsuoka S. Fast conjugate gradients with multiple GPUs. In: Proceedings of ICCS '09. 2009, p. 893–903.

[24] Bolz J, Farmer I, Grinspun E, Schröoder P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans Graph 2003;22(3):917–24.

[25] Li B, Young AA, Cowan BR. GPU accelerated non-rigid registration for the evaluation of cardiac function. In: Proceedings of MICCAI '08. 2008, p. 880–7.

[26] Morales JL, Nocedal J. Remark on "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization". ACM Trans Math Softw 2011;38(1):1–4.

[27] Rong G, Liu Y, Wang W, Yin X, Gu XD, Guo X. GPU-assisted computation of centroidal Voronoi tessellation. IEEE Trans Vis Comput Graph 2011;17(3):345–56.

[28] Averick BM, Carter RG, Moré JJ, Xue GL. The MINPACK-2 test problem collection. Tech. Rep. MCS-P153-0692; Argonne National Laboratory; 1992.

[29] Broyden C, Dennis Jr J, Moré J. On the local and superlinear convergence of quasi-Newton methods. IMA J Appl Math 1973;12(3):223–45.

[30] Jiang L, Byrd RH, Eskow E, Schnabel RB. A preconditioned L-BFGS algorithm with application to molecular energy minimization. Tech. Rep. CU-CS-982-04; Department of Computer Science, University of Colorado; 2004.

[31] Gao G, Reynolds A. An improved implementation of the LBFGS algorithm for automatic history matching. In: Proceedings of ATCE '04. 2004, p. 1–18.

[32] Schraudolph N, Yu J, Günter S. A stochastic quasi-Newton method for online convex optimization. In: Proceedings of AISTATS '07. 2007, p. 433–40.

[33] Liu Y. HLBFGS. 2010. http://research.microsoft.com/en-us/UM/people/yangliu/software/HLBFGS/.

[34] Morales J. A numerical study of limited memory BFGS methods. Appl Math Lett 2002;15(4):481–7.

[35] Hillesland KE, Molinov S, Grzeszczuk R. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. In: ACM SIGGRAPH '05 Courses. 2005,.

[36] Goodnight N, Woolley C, Lewin G, Luebke D, Humphreys G. A multigrid solver for boundary value problems using programmable graphics hardware. In: Proceedings of HPG '03. 2003, p. 102–11.

[37] Krüger J, Westermann R. Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans Graph 2003;22(3):908–16.

[38] Feng Z, Li P. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. In: Proceedings of ICCAD '08. 2008, p. 647–54.

[39] Buatois L, Caumon G, Levy B. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. Int J Parallel, Emergent Distrib Sys 2009;24(3):205–23.

[40] Ament M, Knittel G, Weiskopf D, Strasser W. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In: Proceedings of PDP '10. 2010, p. 583–92.

[41] Dehnavi M, Fernandez M, Giannacopoulos D. Enhancing the performance of conjugate gradient solvers on graphic processing units. IEEE Trans Magn 2011;47(5):1162–5.

[42] Cevahir A, Nukada A, Matsuoka S. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. Comput Sci Res Dev 2010;25(1):83–91.

[43] Verschoor M, Jalba A. Analysis and performance estimation of the conjugate gradient method on multiple GPUs. Parallel Comput 2012;38(10-11):552–75.

[44] Yatawatta S, Kazemi S, Zaroubi S. GPU accelerated nonlinear optimization in radio interferometric calibration. In: Proceedings of IPC '12. 2012, p. 1–6.

[45] Wetzl J, Taubmann O, Haase S, Köhler T, Kraus M, Hornegger J. GPU Accelerated Time-of-Flight Super-Resolution for Image-Guided Surgery. In: Tolxdorff T, Deserno TM, editors. Bildverarbeitung für die Medizin. 2013, p. 21–6.

[46] Sellitto M. Accelerating an imaging spectroscopy algorithm for submerged marine environments using heterogeneous computing. Master's thesis; Department of Electrical and Computer Engineering, Northeastern University; 2012.

[47] Harris M. Optimizing parallel reduction in CUDA. NVIDIA Corporation; 2007. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[48] CUDA C programming guide. NVIDIA Corporation; 2007. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[49] Gunnels J, Lin C, Morrow G, van de Geijn R. A flexible class of parallel matrix multiplication algorithms. In: Proceedings of IPPS/SPDP '98. 1998, p. 110–6.

[50] Humphrey J, Price D, Spagnoli K, Paolini A, Kelmelis E. CULA: hybrid GPU accelerated linear algebra routines. In: Proceedings of SPIE '10. 2010, p. 770502.

[51] CUBLAS Library. NVIDIA Corporation; 2008. http://docs.nvidia.com/cuda/cublas/index.html.

[52] Volkov V, Demmel JW. Benchmarking GPUs to tune dense linear algebra. In: Proceedings of SC '08. 2008, p. 31:1–31:11.

[53] Sengupta S. Efficient primitives and algorithms for many-core architectures. Ph.D. thesis; University of California, Davis; 2010.

[54] Thrust. NVIDIA Corporation; 2009. http://docs.nvidia.com/cuda/thrust/index.html.

[55] Henry S. Parallelizing Cholesky's decomposition algorithm. Tech. Rep.; INRIA Bordeaux; 2009.

[56] Rong G, Tan TS. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In: Proceedings of I3D '06. 2006, p. 109–16.

[57] Rong G, Tan TS. Variants of jump flooding algorithm for computing discrete Voronoi diagrams. In: Proceedings of ISVD '07. 2007, p. 176–81.

[58] Yuan Z, Rong G, Guo X, Wang W. Generalized Voronoi diagram computation on GPU. In: Proceedings of ISVD '11. 2011, p. 75–82.

[59] Frey P, Borouchaki H. Surface mesh evaluation. In: Proceedings of IMR '97. 1997, p. 363–74.

[60] Dolan E, Moré J, Munson T. Benchmarking optimization software with COPS 3.0. Tech. Rep. ANL/MCS-TM-273; Argonne National Laboratory; 2004.