Hybrid MPI/GPU Interpolation for Grid DEM Construction

Andrew Danner
Swarthmore College
Swarthmore, PA 19081
adanner@cs.swarthmore.edu

Jake Baskin
Lindblom Math and Science Academy
Chicago, IL 60636
jake.baskin@gmail.com

Alexander Breslow
University of California, San Diego
La Jolla, CA 92093
abreslow@cs.ucsd.edu

David Wilikofsky
Swarthmore College
Swarthmore, PA 19081
dwiliko1@swarthmore.edu

ABSTRACT

The proliferation of lidar technology in remote sensing has resulted in extremely large, high resolution point clouds covering a wide variety of terrain. Constructing a grid digital elevation model (DEM) from these large data sets requires extensive computational resources and ample disk space. We propose a framework for leveraging modern computing resources including multi-core distributed systems and general purpose GPU computing to reduce computational bottlenecks and accelerate DEM construction. We employ an I/O-efficient strategy using quad trees to automatically partition the lidar point clouds into a set of independent *work bundles*. We then distribute these work bundles to multiple GPU-equipped hosts which independently interpolate a portion of the DEM and return partial results. Finally, we gather the partial results and assemble the final DEM I/O-efficiently. Our approach balances I/O, computation, and network communication to reduce bottlenecks. Experimental results show that our approach scales linearly with the number of compute hosts, and achieves speed-ups of 25$\times$ or greater using GPU computing. These results make it practical to use more complex interpolation methods such as regularized splines with tension, which provide geomorphological advantages over simpler interpolation methods such as linear interpolation, nearest neighbor interpolation, or natural neighbor interpolation.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: Distributed programming, Parallel programming

General Terms: Algorithms, Design, Performance

Keywords: Interpolation, MPI, CUDA, Terrain Modeling

1 Introduction

Modern remote sensing methods such as lidar continue to acquire spatial data at unprecedented rates. Airborne lidar sensors are capable of measuring the elevation of Earth's surface at horizontal resolutions of 30cm or better. Many applications in transportation, hydrology, ecology, and urban planning benefit from this very high resolution data. These applications, however, do not work directly on the raw waveforms or scattered point clouds generated by the lidar sensors. Instead, users compute a digital elevation model (DEM) from the lidar points and run analysis on the digital elevation model. The extremely large size of the input lidar data sets makes the computation of a large, high resolution DEM a computationally intensive and algorithmically challenging task.

For simplicity and efficiency, one of the widely used digital elevation models used in GIS is the grid model in which a single functional value is stored in each cell of a two-dimensional uniform grid, $\mathcal{G}$. Lidar points are not sampled on a uniform grid, but instead are often represented as a set $\mathcal{S}$ of scattered points in $\mathbb{R}^2$ associated with a height function $h : \mathcal{S} \to \mathbb{R}$. An *interpolation* or *approximation* method transforms the scattered lidar representation $\mathcal{S}$ to a real valued height function $f : \mathbb{R}^2 \to \mathbb{R}$ which can then be evaluated at regularly spaced grid cells to compute the final grid DEM $\mathcal{G}$.

Interpolation and approximation of point data is a well studied problem in GIS. A variety of methods including linear interpolation, inverse distance weighting (IDW), kriging, spline interpolation and natural neighbor interpolation have been developed; refer to [15] for a thorough overview. Because of the complexity and quantity of lidar data, simpler methods such as triangulating the point set and linearly interpolating triangles are often preferred over more complex methods. Given the high density of lidar data, this may produce adequate elevation models, but the underlying model is not smooth, and any attempts to extract relevant geomorphological parameters such as slope and curvatures result in numerous discontinuities. The regularized spline with tension (RST) [16] approximates the surface using higher order, differentiable functions, to produce a smooth surface and provide improved means to evaluate slope and curvature. This better quality surface approximation for geomorphology applications comes at a significantly increased computational cost.

The large, multi-gigabyte size of modern lidar data sets, makes processing the data more computationally demanding. However, technical advances in computing power have enabled the development of new algorithms to efficiently use modern hardware resources, particularly parallel processing capabilities in both multi-core CPU systems and graphics processing units (GPUs). Traditionally GPUs have been used to render 3D geometric models on a two dimensional display of pixels. Typically, the calculations involve executing the same operation, e.g., a matrix multiplication for lighting, on a large stream of geometric objects represented by vectors of vertices. GPUs have been optimized for massively parallel execution of vector operations and large memory bandwidth. By exposing a model of computation for programming these GPUs, as done by NVIDIA's CUDA library [19] or OpenCL [13], new applications can extend the computation of GPUs beyond computer graphics and into general purpose GPU computing (GPGPU). In the area of geospatial processing, GPGPU computing has seen increased use [12, 10, 14, 5], including for the problem of DEM construction [4].

Our results. In this paper we develop a framework for efficiently constructing large, high-resolution grid DEM from lidar point clouds. Our system is designed to reduce computation bottlenecks and leverage modern hardware resources including multi-core CPUs and GPGPUs. We experimentally validate our results using a 96 node GPU cluster Forge [22] available from NCSA and XSEDE [9]. Our approach has the following key components:

1. A scalable, I/O-efficient, quad tree segmentation algorithm to automatically partition a large lidar point cloud into smaller independent *work bundles* representing a subset of the final output grid DEM.
2. A distributed multi-core component using a message passing interface (MPI) implementation to assign work bundles to multiple processes.
3. A GPU component which accelerates interpolation, particularly for complex interpolation methods like regularized spline with tension (RST).

Our experimental results show that our approach scales linearly from two to over 100 hosts. Using GPU RST interpolation further accelerates our computation by a factor of 25× or greater, depending on the tunable size of the work bundles. In one typical case, we computed a 1000x1000 DEM from 1.2 million lidar points in 50 seconds using 48 GPU enabled hosts. The same experiment using a CPU only algorithm on a single host required 3.3 hours, resulting in a 239× speed-up for our approach.

Our paper is organized as follows. In Section 2 we review the quad tree based segmentation algorithm we use to decompose large lidar point sets into smaller regions. In Section 3 we describe our approach for distributing computation tasks across multiple hosts, collecting the results, and assembling the final output DEM. Section 4 gives a brief overview of GPU programming and the regularized spline with tension interpolation method. It also describes how we implement RST interpolation on the GPU. We conclude with a summary of experiments and results which demonstrate the scalability of our framework.

2 Segmenting large points sets

The computational complexity of most interpolation methods makes it impractical to directly interpolate very large point sets commonly collected via lidar. For this reason, scalable algorithms employ a segmentation routine to partition the region of the final output grid into a disjoint set of sub-regions, where each sub-region contains a smaller subset of the original lidar point set. Once the data are partitioned, each sub-region can be interpolated independently. A possible side effect of the partitioning is that boundary effects may be visible where two sub-regions share a common edge. A solution to this side effect is to use both points within a sub-region R and points from neighboring sub-regions adjacent to R to interpolate the surface within R.

A frequently used data structure for partitioning large point sets is the quad tree [6]. For very large data sets, quad trees can be built I/O-efficiently using the methods of Agarwal et al. [2] or Hjaltason and Samet [11]. A variant of the first approach was further used by Agarwal et al. [1] to construct a DEM from very large lidar point sets. While their implementation runs only as a single threaded application, the quad tree construction method used forms the basis of our approach. For completeness, we include a summary of this quad tree construction algorithm, and highlight the features that make this method particularly suitable for our approach.

2.1 Quad tree construction

The result of Agarwal et al. [1] constructs a grid DEM in three phases; a *segmentation* phase which constructs a quad tree on an input point set; a *neighbor finding* phase which computes for each sub-region R, the points in all regions that share a border with R; and an *interpolation* phase, which processes each region and constructs a final output grid DEM. Our paper proposes a replacement interpolation phase which is distributed across multiple processes and can be executed on a GPU. However, our approach relies on the segmentation phase and neighbor finding phase we summarize below.

Given a set $\mathcal{S}$ of N points, a desired output grid bounding box $[x_1, x_2] \times [y_1, y_2]$, and a desired maximum number of points per region, k, Agarwal et al. [1], construct a quad tree [6] $\mathcal{T}$, on $\mathcal{S}$, such that each quad tree leaf has no more than k points. The basic premise is to incrementally build $\mathcal{T}$ top-down by inserting the points of $\mathcal{S}$ into an initially empty quad tree. For each point p, we traverse a path from the root of $\mathcal{T}$ to the one leaf q whose corresponding sub-region contains p. If q contains fewer than k points, we simply add p to q. Otherwise, we split q into four new leaves, each representing a quadrant of q and distribute each point within q to the appropriate new leaf.

To make this approach scalable to data sets too large to fit in main memory, not all levels of the quad tree are computed in memory in one pass. Instead, a *layer* of l levels of the quad tree are constructed in a single pass over the data. The parameter l is chosen such that a sub-tree of $\mathcal{T}$ containing l levels can fit in memory along with a buffer for each leaf in the sub-tree. The basic construction algorithm is the same, but when a point reaches a leaf q of depth l in a layer of a sub-tree of $\mathcal{T}$ containing more than k points, the leaf is not sub-divided further, but instead the points are written to an external buffer L_q for later processing. Once all points have

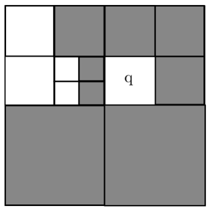

Figure 1: A quad tree leaf q and its eight shaded neighboring regions.

been inserted into a particular layer, the quad tree layer is written to disk and any external buffers are handled recursively by constructing new layers for each external buffer L_i. See [1] for further details.

2.2 Finding neighbors

Once we have constructed the quad tree $\mathcal{T}$ on the set of points $\mathcal{S}$, our next step is to identify for each leaf q of $\mathcal{T}$, the set of points S_q in all *neighboring* leaves of q. Two leaves are neighbors of each other if the regions associated with each leaf share a common boundary; either an edge or a corner. Figure 1 shows an example of all the neighbors of a sample quad tree leaf.

As was the case for the construction of the quad tree, Agarwal et al. [1] use a layered approach to identify neighboring points for each quad tree leaf. We begin with a list $\mathcal{L}$ of all quad leaves in $\mathcal{T}$ and the regions associated with each leaf in $\mathcal{L}$. For each leaf $q \in \mathcal{L}$, we filter the leaf through the layers of $\mathcal{T}$, starting with the layer associated with the root of $\mathcal{T}$. Starting with the root v of each layer, we compare q to each region associated with the children of v. If q shares a boundary with a child u of v, or if the region associated with q is contained by the region associated with u, then q is a neighbor of at least one leaf in the tree rooted in u. We then recursively visit each child node that neighbors or contains q. Using the layered approach, each query either terminates at a leaf w of $\mathcal{T}$ or an internal node z at the bottom of one layer of the quad-tree. In the first case, we add a pair (q, S_w) to a list Λ on disk where S_w is the set of points contained in w, a leaf adjacent to q. For the queries neighboring or contained within an internal node z at the bottom of a layer, we add q to a external list $\mathcal{L}_z$. After processing all leaves in $\mathcal{L}$ through the first layer of $\mathcal{T}$, we recursively identify neighboring points for each leaf in each list $\mathcal{L}_u$ for each layer rooted at u.

After processing all layers of $\mathcal{T}$, the list Λ contains a set of pairs (q, S_u) indicating that the points in S_u are in a leaf which is a neighbor of the leaf q. At this point, it is easy to construct the set S_q containing all points in all neighbors of a leaf q; we simply sort Λ by the first element in the pair (q, S_u) and scan the resulting list. All neighboring points of q will be contiguous in the final sorted list.

This particular construction and post-processing step enables our intuitive parallel interpolation phase. What the neighbor finding phase produces is a sequential stream where we can access information about the boundary of each quad tree leaf q and the points contained both within q and the neighbors of q in a simple sequential scan. Agarwal et al. [1], proceeded by serially processing and interpolating the region associated with q using the points S_q. We describe our improvements to this serial approach in the next two sections.

3 Distributed Computation

The quad tree construction decomposes the entire output grid DEM into a set of smaller sub-regions so we can interpolate each sub-region independently using a small number of points instead of trying to interpolate the entire region using the entire input point set $\mathcal{S}$. Furthermore, the neighbor finding phase of the previous section pre-computes for each region represented by a quad tree leaf q, the points in neighboring leaves of q and arranges all these neighboring points sequentially on disk in a large list Λ. This naturally leads to the following sequential algorithm for constructing the entire DEM: For each quad tree leaf q, we sequentially construct an interpolated surface $f(x, y)$ using the points S_q inside q and its neighboring leaves. For each cell (col, row), we compute the corresponding (x, y) values for the center of the cell and evaluate the interpolated surface $z = f(x, y)$. For scalability, rather than writing this cell value directly to the final location in the grid DEM using random access I/O, we write the tuple (col, row, z) to a file G and sort G by grid order once all leaves have been interpolated. This approach is summarized in Algorithm 1. We refer to a single pair (q, S_q) as a *work bundle.* A work bundle represents the finest level of granularity of the entire interpolation task.

Algorithm 1 Sequential Interpolation

```
$G \leftarrow \emptyset$
**for all** $(q, S_q) \in \Lambda$ **do**
    construct interpolated surface $f(x, y)$ using $S_q$
    **for all** cells $(col, row) \in q$ **do**
        $z \leftarrow f(x(col), y(row))$
        write $(col, row, z)$ to $G$
    **end for**
**end for**
Sort $G$ by grid order
Scan sorted stream $G$ and write to final output DEM
```

Since the region corresponding to each leaf q is interpolated independently, the outer loop of Algorithm 1 can easily be parallelized and multiple processing units could simultaneously processes individual regions. We develop our parallel algorithm so that it is easy to implement in a popular parallel application framework, MPI [7]. MPI defines a message passing interface that allows multiple processing units to communicate via the exchange of *messages.* Each message has a source ID, a destination ID, a message tag ID, and a buffer of typed data containing the contents of the message. The MPI framework provides `send` and `receive` functions to implement communication between processes. Each process participating in a parallel computation has unique ID known to all other processes. Our parallel solution for interpolation uses $P \geq 3$ processes to distribute computation. A particular process serves one of three roles: *scatterer*, *gatherer*, or *worker.* In our approach, there is one dedicated scatterer, one dedicated gatherer and $P - 2$ worker processes. These roles are illustrated in Figure 2 and described below.

The scatterer is the only process that reads from the file Λ containing the pairs (q, S_q). While there are still unread pairs, the scatter process reads the next pair from Λ, waits for a free worker process i and distributes a work bundle (q, S_q) to i by exchanging a few messages containing information about q, the grid sub-region associated with q,

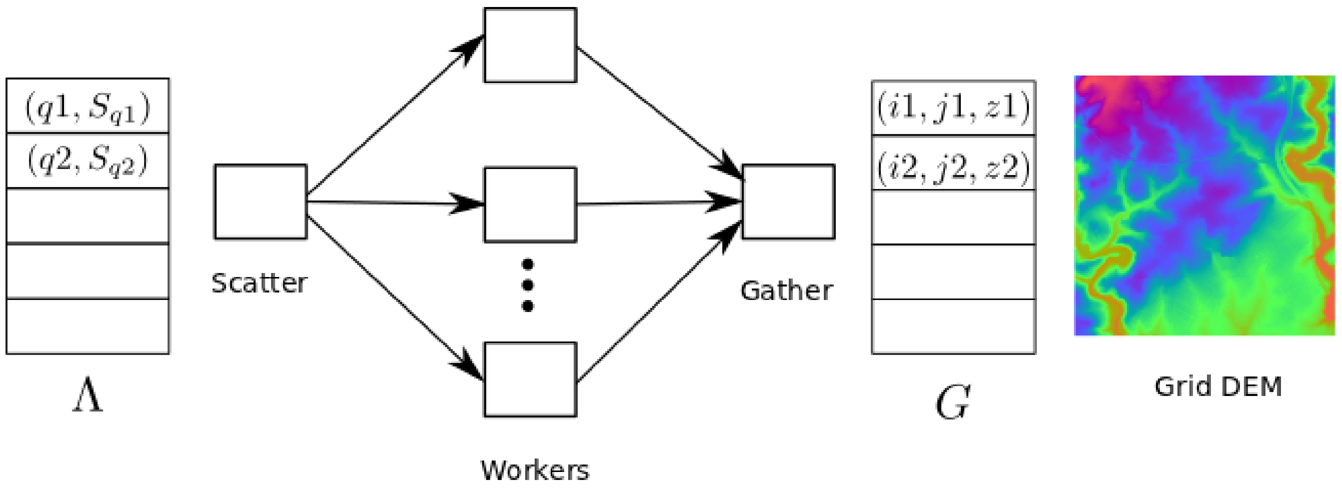

Figure 2: Accelerating interpolation by distributing work bundles across multiple worker processes using MPI. A scatter process handles reading work bundles from disk and distributing them to workers. The gather process collects cells from workers and writes them unsorted to a file G. A final sort of G produces the final DEM.

and one or messages containing the points in S_q. See Algorithm 2.

Algorithm 2 Scatter

```
for all $(q, S_q) \in \Lambda$ do
    find free worker, $i$
    send $(q, S_q)$ to process $i$ via MPI
end for
```

A worker process repeatedly receives messages containing a work bundle, constructs the interpolated surface as done in the serial algorithm, and evaluates the cell values as seen in Algorithm 3. Instead of each worker writing the (i, j, z) cell values directly to the output file G, each worker sends the cell values to the gatherer process.

Algorithm 3 Worker

```
while $(q, S_q)$ received via MPI from scatterer do
    construct interpolated surface $f(x, y)$ using $S_q$
    for all cells $(col, row) \in q$ do
        $z \leftarrow f(x(col), y(row))$
        send (col,row,z) to gatherer via MPI
    end for
    Notify scatterer via MPI that this process is ready.
end while
```

The gatherer process described in Algorithm 4 collects (i, j, z) values from worker processes and writes the values to G. Once the gatherer has received all the cells, it sorts G by grid order and writes the final output DEM.

Decomposed in this way, only two processes need to perform file I/O. The scatterer is the only process reading files and the gatherer is the only process writing files. The exchange of messages is implemented over the network interconnect, so that no disk I/O is necessary for the worker processes. Aside from the sorting by the gatherer process, all computation is done by workers. If the I/O bandwidth is sufficient, a single process could server multiple roles. For instance, the role of scatterer and gatherer could be assigned to a single process.

Algorithm 4 Gather

```
$G \leftarrow \emptyset$
Find worker $i$ ready to send computed cells
while cells available from some worker $i$ do
    receive $(col, row, z)$ from worker $i$ via MPI
    write $(col, row, z)$ to $G$
    Find another worker $i$ ready to send computed cells
end while
Sort $G$ by grid order
Scan sorted stream $G$ and write to final output DEM
```

Besides limiting I/O contention, our distributed interpolation has a number of advantages. Our approach is scalable as there is limited communication between nodes. To reduce message overhead, points sent to workers and cells sent to the gatherer are not sent one at a time, but are instead buffered into blocks of multiple points/cells. Because the scatter and gather processes do not perform much computation, they can devote hardware resources to file I/O and network communication. Conversely, for a particular work bundle, a worker performs little network communication and no I/O and can thus devote most of its hardware resources to interpolation.

Our approach also follows naturally from the quad tree decomposition and no additional post processing of the file of work bundles $(q, S_q) \in \Lambda$ is needed to switch from the sequential algorithm to the distributed algorithm. As we later describe in Section 5, the MPI implementation also allows us to leverage multi-core nodes in distributed clusters by assigning multiple workers to a single multi-core host. We show that distributing computation using the methods in this section provide linear speedups across a large range for the number of processes. However, we can achieve even greater performances by parallelizing the interpolation of a single work bundle using GPU computing.

4 GPU Accelerated Interpolation

The quad tree segmentation and distribution of work bundles are independent of the interpolation method used. To construct terrains based on geomorphological principles, and to demonstrate the performance benefits of parallelization, we use the regularized spline with tension method described by Mitasova et al. [18] as our interpolation method. This method models the surface as a thin plate spline under tension, and is an example of one many different spline methods that have been proposed for surface approximation. While seemingly complicated, this method has many advantages over other simpler approximation schemes. In particular, it can accurately compute secondary surface properties such as slope, profile curvature, and tangential curvature, which are important in landform analysis and landscape process modeling. We present the details of the regularized spline with tension method here for completeness.

Given n input points $\{\vec{r_1}, \vec{r_2}, \ldots, \vec{r_n}\}$, where $\vec{r_i} = (x_i, y_i)$, each with a value z_i, the surface is defined by

$$z(\vec{r}) = a_1 + \sum_{j=1}^{n} \lambda_j R(\rho_j), \tag{1}$$

$$R(\rho_j) = -[E_1(\rho_j) + \ln \rho_j + C_E],$$

where $z(\vec{r})$ is the value at an arbitrary point $\vec{r} = (x, y)$, a_1 is a constant trend, λ_j are a set of coefficients, and $R(\rho_j)$ is a radial basis function. In the function $R(\rho_j)$, $C_E = 0.577215\ldots$ is the Euler constant, and $E_1(\rho_j) = \int_{\rho_j}^{\infty} \frac{e^{-u}}{u} du$ is the exponential integral function. $\rho_j = (\varphi|\vec{r} - \vec{r_j}|/2)^2$, where $|\vec{r} - \vec{r_j}|$ is a Euclidean distance function in $\mathbb{R}^2$, and φ is a tunable tension parameter. As $\varphi > 0$ is decreased, the approximation surface is tuned from acting like a rigid metal sheet to a flexible membrane.

The $n + 1$ coefficients a_1 and λ_j are found by solving the following linear system of equations

$$a_1 + \sum_{j=1}^{n} \lambda_j [R(\rho_i) + \delta_{ij} w_0/w_j] = z_i, \quad i = 1, \ldots, n \tag{2}$$

$$\sum_{i=1}^{n} \lambda_j = 0, \tag{3}$$

where w_0/w_j are positive weights representing a smoothing parameter for each point r_j. Setting the smoothing parameter w_0/w_j to 0 results in an interpolation method where the surface must pass through all the input points. Increasing the smoothing for a particular point $\vec{r_i}$ allows the surface to approximate z_i at $\vec{r_i}$. A particular advantage of this method is that in addition to computing an interpolated surface, high order derivatives of the surface can be computed by direct evaluation of the derivative of $z(\vec{r})$.

Typically, a standard LUP decomposition is used to solve the linear system of equations. This approach is used in implementations by Agarwal et al. [1] and Mitasova et al. [17, 18]. For a set of n points, solving the linear system for coefficients a_1 and $\lambda_i, 1 \leq i \leq n$ can be done in $O(n^3)$ time. We use this serial implementation based on Crout's algorithm for in-place matrix LUP decomposition as our comparison to our GPU approach described below.

4.1 CUDA architecture overview

For developing our GPGPU interpolation solution, we use the CUDA architecture and computation model developed for use on modern NVIDIA GPUs. CUDA capable GPUs feature a very large number of computing processors, or *cores*, high memory bandwidth, and floating point computation performance exceeding that of traditional CPUs. For example, the Tesla M2070 GPU from NVIDIA features 448 cores, a 150GB/sec memory bandwidth, 6GB of GPU memory and 1030 Gigaflops of peak single precision floating point performance.

On the software side, CUDA provides a small C-like library for programming the GPU. GPU code is described in special GPU functions called *kernels*. The code contained within a kernel executes on all GPU cores simultaneously, reading and writing data from GPU memory. The CUDA architecture is designed to efficiently access GPU memory, particularly when sequential cores are accessing sequential data elements in memory. The basic design of a CUDA application is to copy data from CPU RAM to GPU memory, run one or more kernels on the GPU cores, and transfer the final result back to CPU memory for traditional, non-GPU post-processing.

To gain the best performance on GPGPU applications, it is important to keep CUDA cores busy and reduce the amount of data transfer between CPU memory and GPU memory. For small problem sizes, the overhead of copying data between the CPU and GPU may be greater than the time it takes to simply process the data sequentially on the CPU directly.

4.2 GPU Interpolation

Our approach uses the GPU to process a single work bundle (q, S_q) by executing three GPU kernels. The first kernel uses the n points in S_q to construct an $n+1 \times n+1$ matrix A, representing the coefficients of a_1 and $\lambda_i, 1 \leq i \leq n$ in Equation 2 and Equation 3. By copying the x and y values of the points in S_q to GPU, each GPU core can independently compute a single element in A. If there are more elements in A than there are GPU cores, we simply assign elements of A to cores in a round robin fashion. Using a special grid block and thread group model in CUDA, this assignment can be handled automatically by CUDA in a single kernel call. Note that since A is symmetric, we only need to compute elements on or above the diagonal and copy elements from above the diagonal to the lower side of A. The matrix A resides entirely in GPU memory.

Our second kernel solves the system $A\lambda = z$ for λ, where $\lambda = [a_1, \lambda_1, \ldots, \lambda_n]^T$ and $z = [0, z_1, \ldots, z_n]$, where z_i is the elevation of the point $(x_i, y_i, z_i) \in S_q$ for $1 \leq i \leq n$. Instead of developing a full parallel GPU accelerated LUP decomposition kernel, we leveraged the expertise of CULA [8], a set of GPU accelerated linear algebra routines. CULA provides GPU functionality similar to the CPU library LAPACK. Using the values for A and z stored in GPU memory, CULA solves for λ using the GPU and stores the result back in GPU memory.

Our final GPU kernel computes the values for cells in the region of the final grid DEM defined by the subregion q, by using λ, S_q, and the center coordinates of each cell in the DEM to evaluate Equation 1. Using the GPU, we can parallelize this computation by assigning each GPU core to single cell. As was the case in the computation of the matrix A, if there are more cells than cores, we can assign cells to cores in a round-robin fashion using readily available CUDA kernel semantics.

Once the cell values for a particular work bundle (q, S_q) are computed by the GPU, we copy the values back to the CPU for further processing. If, for example, we are using worker processes as in Algorithm 3 in Section 3, the worker would not need to compute cell values on the CPU. It would instead convert the GPU output cells to a sequence of $(\text{col}, \text{row}, z)$ tuples to send via MPI to the gatherer process.

Overall, our GPU accelerated interpolation is a simple adaptation of the standard RST algorithm implemented on the CPU in prior work. The assignment of work to GPU core is relatively natural in these matrix and grid-based applications as we have multiple matrix elements/cells performing independent but similar computations. As we show in our experimental results in the next section, this interpolation can be significantly accelerated by using GPU resources.

5 Experiments

We tested the effectiveness of our system through a series of experiments on a large GPU cluster available through XSEDE [9], formerly Teragrid. Our primary test cluster was the Dell NVIDIA Linux Cluster, Forge, available through NCSA. The primary work queue on Forge consists of 32 AMD Opteron Magny-Cours 2.4 GHz dual-socket eight-core nodes with 48GB of CPU memory. Each physical node is connected to six NVIDIA Fermi M2070 GPUs. Each GPU consists of 448 cores, 6GB of GPU memory and 1.03 Teraflops of single-precision floating point performance. InfiniBand QDR provides the network interconnect between physical nodes.

Implementation. We implemented our solution in C++ and CUDA 4.0. For the quad tree construction, we used the implementation of Agarwal et al. [1], which also leverages TPIE [3, 21], a library for easing the development of I/O-efficient algorithms on large data sets. We implemented our MPI layer to interface with TPIE files generated by the quad tree construction phase. We used the openmpi 1.4.3 library implementation of the MPI interface. The implementation of the scatter node in the MPI layer used TPIE to write grid values to disk and sort the final set of grid points. Initial experiments indicated that there was sufficient I/O bandwidth to allow the scatter and gather processes to be implemented as a single MPI process, thus freeing an additional process for the computationally intensive interpolation.

Our implementation was designed to easily swap interpolation methods. Primarily, we wanted to explore the advantages of a GPU version of the regularized spline with tension (RST) method with an standard CPU implementation. The CPU implementation uses no external libraries, such as LAPACK. Initial experiments indicated that our hand coded implementation interfaced better with the output of the quad tree construction output and avoided the overhead of pre- and post-processing the input/output to conform to LAPACK formats. We developed our CUDA implementation of RST with the CULA [8] routines for LUP decomposition in mind so that we could leverage the expertise in GPU LUP decomposition provided by the CULA library.

Data. For experimental data, we used lidar data files in LAS format available from the Pennsylvania Spatial Data Access (PASDA) Geospatial Data Clearinghouse [20]. Lidar tiles are available in 10,000ft x 10,000ft tiles projected into PA State Plane coordinates. We used lidar points classified as ground points during a 2008 Survey of Delaware County in Southeastern Pennsylvania. The average point density was roughly one point per 50ft^2, suitable for grid DEMs with a spatial resolution of 10ft per cell or better.

Overview of Experiments. The performance of our implementation depends on a number of parameters. The two primary parameters that are independent of our implementation is the number of input lidar points N and the number of cells, G in the final output grid DEM. Given a fixed N and G however, there are number of parameters specific to our implementation. The first is the maximum number of points per quad tree leaf, k. A small value of k results in a large number of leaves, each with a small number of points. This means that the interpolation of each individual work bundle (q, S_q) will be less computationally intensive for small k as opposed to larger values of k. However, the number of work bundles increases as k decreases.

The second parameter we can vary in our implementation is the number of processors P used in the distributed computation. Increasing P should decrease overall runtime, provided we do not create a communication bottleneck in transferring data between an increasing number of processes. Our final parameter that we explore is the advantage of using the GPU to compute the interpolation as opposed to a standard CPU only implementation. We expect the GPU to outperform the CPU, provided we can provide enough data to the GPU to overcome the overhead of transferring the data from CPU memory to GPU memory when running our GPGPU interpolation.

As we are primarily interested in the scalability of our implementation, our experimental discussion focuses mostly on tuning the parameters k and P, and enabling or disabling interpolation via the GPU. We summarize our results below. Unless otherwise stated, our tests were run using a single 10,000ft by 10,000ft lidar file in LAS format containing $N = 2.2$ million ground points. Our final output DEM was constructed at 10ft resolution, containing 1000 rows, 1000 columns, and $G = 1$ million total cells. During quad-tree construction, points closer than half the grid resolution away from another point in the quad tree are discarded since the output grid will not be interpolated at sub-pixel resolution. For our experiments, a quad-tree for 10ft resolution on our 2.2 million point data set contained 1.5 million points after thinning.

5.1 Quad tree construction

Our first set of experiments was to create multiple sets of work bundles for various values of k, the maximum number of points per quad tree leaf. This process is independent of both the method of interpolation used and the number of processes used for interpolation. Results are shown in Table 1. As the quad tree is built in layers, processing all layers starting at a particular depth in the quad tree effectively reads and writes the input data set once. Thus, the run-time is roughly a function of the depth of the quad tree and the number of I/O passes over the data. With a reasonable 8GB of memory, the quad tree and all work bundles can be constructed in a single pass over the input and internal sort of work bundles. Thus the overall run-time for the quad tree construction is not significantly influenced by k.

For a even distribution of points, we would expect roughly N/k total leaves. We can see this general trend in Table 1, though for large values of k, we can see more heterogeneous density and deviations from the expected N/k distribution. For example compare the number of leaves for $k = 10$, 100, and 1000, or $k = 20$, 200, and 2000. For 1.5 million input points, we expect roughly 2200 leaves with $k = 1000$ and 1100 leaves with $k = 2000$. We see this behavior experimentally for $k = 2000$, but some local regions with a high density of points when $k = 1000$ results in some additional leaf splitting of the quad tree leaves, and a higher than expected leaf count when compared to a uniform distribution of points. This illustrates that lidar data indeed does not always have a uniform point density and that a quad tree decomposition which can adjust to non-uniform point density is an appropriate data structure for lidar data.

k	time	leaves
10	16	257889
20	14	165267
50	15	59715
100	15	25779
150	14	16401
200	14	15720
300	16	10755
400	13	5478
500	13	4233
800	16	4056
1000	14	3747
2000	11	1029
5000	18	582

Table 1: Quad tree construction time, shown in seconds, is mostly independent of the maximum number of points per quad tree leaf, k

5.2 Impact of quad tree leaf size

The real impact of the number of points in the quad tree leaf, k, is during the interpolation phase. Recall that the interpolation on each work bundle uses both the points within a quad tree leaf q and points in leaves which are neighbors of q. A quick analysis [1] shows that the expected number of neighbors for a quad tree leaf is eight, thus we expect the interpolation of each work bundle to run on $8k = O(k)$ points. In the case of RST interpolation and other related interpolation methods which solve a system of k linear equations, we expect the runtime per work bundle to be $O(k^3)$. Since the total number of work bundles/leaves would be roughly $O(N/k)$ for uniformly distributed data, we expect the total run time to be $O(Nk^2)$. To experimentally verify this, we ran our RST interpolation implementation on the work bundles generated by the quad tree construction for various values of k. We also compared the results using both a CPU version of RST and our GPU version implemented in CUDA. In each case, we used eight MPI processes. Using multiple processes helped conduct initial tests of our MPI implementation, and accelerated our run-times to test a wide range of values for k. Additionally, the cost model for running jobs

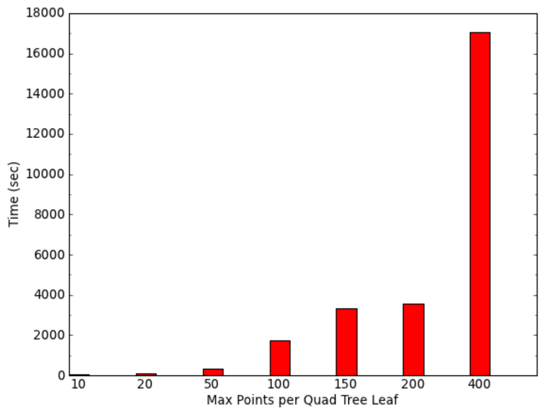

Figure 3: Run time vs k using CPU RST interpolant across eight MPI processes. Note, horizontal scale is not linear

on the Forge cluster charges users per physical node, not per number of processes. Since Forge contained one work queue with eight GPUs on one physical node, we could conduct tests both with and without the GPU using eight MPI processes for the same cost of using one process.

The results for our CPU RST implementation are shown in Figure 3. The timings only include the time for interpolation and for writing the final grid DEM, not the time for quad tree construction which is mostly independent of k and independent of the interpolation method. Note that the horizontal scale is not linear. The strong non-linear dependence on k is visible in the chart. Actual values for $k = 10, 20, 50, 100, 150, 200$ and 400 were 46, 73, 313, 1711, 3341, 3567, and 17051 seconds respectively. Compared to constructing the quad tree, interpolating dominates the overall run-time, even for small values of k. The final output DEM is the same size for each of these experiments, and constructing the final DEM takes only 2-3 seconds in each experiment.

The results of the CPU interpolation implementation show that having a small value of k is desirable to reduce overall runtime. However, having a small value of k creates more individual leaves, and while we use points from neighboring leaves to improve smoothness of the interpolation across boundaries of neighboring leaf regions, increasing k would result in fewer potential artifacts of our quad tree decomposition method as we transition across leaf boundaries.

GPU Interpolation. As we see in Figure 4, we can achieve significant reductions in overall run-time for large values of k by using a GPU interpolation routine. For a wide range of values of $k, 10 \leq k \leq 800$, the interpolation time is consistently around 140 seconds, and we see no quadratic dependence on k on the GPU. Furthermore, for $k = 10$, the GPU implementation at 147 seconds is more than three times *slower* than the corresponding CPU implementation. This comparison is a bit misleading, however. What is happening in this case is that we are seeing the overhead of copying data from CPU memory to GPU and getting little benefit from the hundreds of cores available on the GPU. Effectively, the runtime is I/O bound between CPU memory and GPU memory for small values of k. The actual computation is almost instantaneous for small value

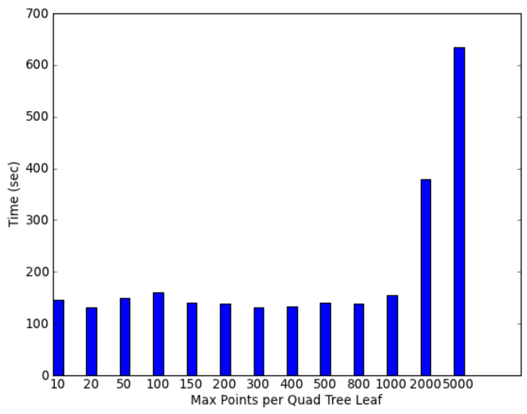

Figure 4: Run time vs k using GPU RST interpolant across eight MPI processes. Note, horizontal scale is not linear. Run time is I/O bound for a wide range of k until the amount of GPU work across 480 cores exceeds the cost of CPU-GPU memory transfers

of k on the GPU. So while we get no speedup for $k = 10$, our relative performance improves dramatically for larger values of k when compared to the CPU. We get speedups of $1.8\times, 10\times$, and $25\times$ for $k = 20, 100$, and 200, respectively. For $k = 200$, the CPU interpolation takes almost one hour compared to less than 140 seconds for the GPU implementation. For $k = 1000$, we could reasonably expect the CPU to take over 24 hours, based on the observed quadratic CPU performance. The GPU easily processes this data set in 156 seconds.

We can actually see some quadratic dependence on k when using the GPU, but we must use much larger values of k than we use on the CPU. Actual values for $k = 1000, 2000$, and 5000 were 156, 378, and 635 seconds respectively. Thus we can see the GPU RST interpolation has tremendous performance benefits over the CPU. It allows us to use higher values of k, reducing artifacts due to quad tree region boundaries as there are fewer leaves for larger k. Both our GPU and CPU implementations assume the entire matrix A representing the coefficient of Equation 2 fits in GPU/CPU memory. For $k = 5000$, using points in a leaf q and its surrounding neighbors can regularly result in matrices several Gigabytes in size. For this reason, we cannot currently process matrices larger than 20000 by 20000 on the GPU. Still, the GPU allows us to efficiently process work bundles two orders of magnitude larger than what is practical on the CPU. Overall, our experimental results demonstrate that, compared to a CPU RST implementation, using GPU to accelerate interpolation allows us to both process data more quickly and also use higher values of k, thus reducing artifacts.

5.3 Impact of number of processes

To test the effectiveness of our distributed approach to interpolation outlined in Section 3, we ran our implementation on for a varying number of processes, using both CPU interpolation and GPU interpolation. The results for the CPU interpolation are shown in Figure 5, while GPU/CUDA results are shown in Figure 6.

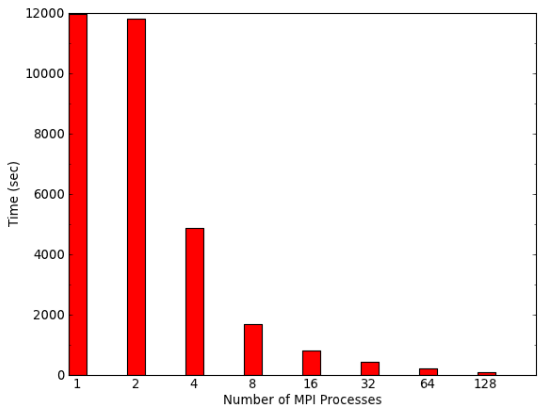

Figure 5: Run time in seconds vs total number of MPI processes, P, using CPU RST interpolant and $k = 100$. Note, horizontal scale is not linear. For $P > 1$, one process serves as a scatterer/gatherer while $P - 1$ processes serve as workers interpolating work bundles. Results for $P = 1$ do not use MPI and instead correspond to Algorithm 1.

For the CPU results, we fixed the value of k at 100. Relative performance appears similar for other values of k. We repeatedly ran our experiments while doubling the number of processes used each time from 1 to 128 total processes. We ran up to eight processes on a single physical host on the Forge Cluster. As each physical host has two eight core processors, each host could comfortably run eight interpolation processes while still managing the operating system with the remaining unused cores. For $P \leq 8$, all processes resided on a single physical host machine.

As Figure 5 shows, our solution scales linearly over a large number of processes. Doubling the number of processes reduces the total run-time by roughly half, as we would expect under ideal circumstances. Such performance indicates that our solution has low MPI communication overhead, allowing the scatterer and gatherer processes to maintain a steady flow of data to all worker processes even when there are over 120 separate workers distributed over sixteen physical hosts. The result for 128 processes, 107 seconds, is 111 times faster than the serial result for no MPI (P=1), 11996 seconds, over three hours.

In addition to running a completely serial version with no MPI, we ran experiments using only $P = 2$ MPI processes. Since one node is dedicated to being the scattering/gathering process leaving only one worker, we expected the results for $P = 2$ to be very similar to the results for $P = 1$. In fact, one might expect that results for $P = 2$ would be slower as a result of MPI communication overhead, which is completely absent in the serial case. Somewhat surprisingly however, the $P = 2$ case is roughly 1–2% *faster* than the serial version. This was confirmed over multiple runs of these two cases. Even though the two processes communicate over MPI, the two processes are physically located on the same hosts and do not need to communicate over InfiniBand to exchange messages. Additionally, there is a small benefit because the I/O of the first process can overlap somewhat with the computation of the worker process. The gatherer prefetches the next work bundle from

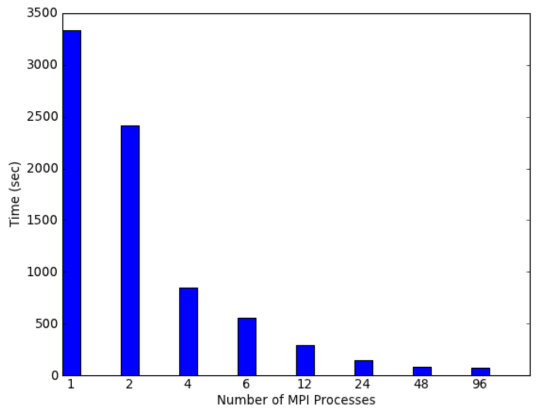

Figure 6: Run time in seconds vs total number of MPI processes, P, using GPU RST interpolant and $k = 2000$. Note, horizontal scale is not linear.

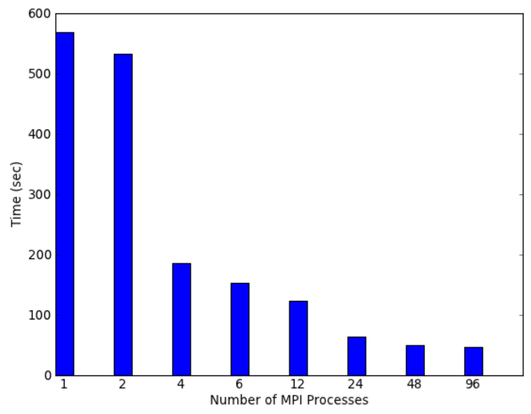

Figure 7: Run time in seconds vs total number of MPI processes, P, using GPU RST interpolant and $k = 1000$. Note, horizontal scale is not linear.

disk after sending the current work bundle to the worker for computation, resulting in small overlap in reading while the worker computes the result for the current work bundle. A similar overlap occurs after the worker sends back the resulting DEM values back to the scatter/gather process. The worker can receive the next bundle over MPI while the gather process writes DEM values to disk. The final result is a small, but noticeable performance boost even when using an MPI solution with two processes. The performance benefits compared to the serial implementation are naturally more noticeable as we increase the number of workers interpolating grid values in parallel.

In addition to comparing the effect of varying the number of processes using a CPU interpolation, we can perform the same analysis while using the GPU. Our setup is slightly modified from the CPU approach as each physical host has only six individual GPU cards. As each worker process needs exclusive access to a GPU, we limit the maximum number of processes per physical host to six, capping the maximum number of processes over sixteen physical hosts at 96. As was the case for the CPU experiments, for $P \leq 6$, all MPI processes run on a single physical host. For the GPU experiments, we fixed k at 2000, since higher k yields fewer work bundles. Furthermore, we found that using a smaller k with GPU interpolation results in greater MPI communication overhead and diminishing returns for large number of MPI processes. As seen in Figure 6, the performance trend is similar to that of Figure 5. We see the same linear scalability with increasing number of MPI processes, at least up to 48 processes. We also see a somewhat more pronounced performance gain when using $P = 2$ processes over the completely serial implementation.

The overhead of MPI is more noticeable in this set of experiments than in the CPU interpolation case. Because the GPU interpolation is so efficient, there is less elapsed time between sending the work bundle out to a worker processes and being able to receive the final DEM values from that worker. When distributing work bundles to a large number of GPU enabled workers, we noticed that some workers sat idle while the scatter processes was distributing bundles to other processes, leading to diminishing returns on performance for larger values of P. In an ideal setup with no communication overhead, we would expect a speedup of roughly $P-1$ when using P processors, as computation is distributed over $P-1$ workers. When $P = 4$ and there are 3 worker processes, the speedup over the serial version $P = 1$ is 2.8×. For $P = 24$, the speedup is only 16.2×, for $P = 48$, 28×, and for $P = 96$, 34×. This indicates that for such high computational throughput we cannot keep the workers sufficiently busy for such a large number of GPUs. For comparison, we include results for using the GPU when $k = 1000$ in Figure 7. The reduced computational workload results in even a greater percentage of communication overhead and diminished performance returns on $P \geq 24$ processes. Still, we continue to get greater speedups using more GPU processes.

Ultimately, we would expect this diminished performance behavior in the CPU interpolation results as well if we increased the number of worker processes sufficiently. At some point, the time it takes to communicate with all $P-1$ worker processes exceeds the time it takes to process a work bundle on a single worker. Given the slow interpolation of the CPU, we do not see this limit for 128 workers, but we definitely begin to see the communication overhead even using a few as six GPU accelerated workers.

5.4 Large Scale Test

As a final experiment to demonstrate scalability, we constructed a 10ft resolution DEM containing 121 million cells from 162 million lidar points, occupying 9GB of disk space. Constructing the quad tree on a single host took 13 minutes using $k = 1000$. Using 24 GPUs, we could construct the DEM using RST in 2 hours and 4 minutes. The final result is shown in Figure 8.

6 Discussion

Overall, our solution for grid DEM construction leverages many modern computational techniques to quickly compute a grid DEM from large lidar point clouds. Interpolation techniques such as RST have typically been replaced by simpler methods when used with lidar due the computational

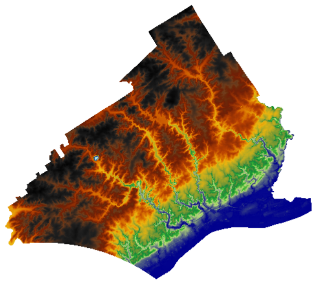

Figure 8: 10ft DEM of Delaware County, Pennsylvania, 121 million grid cells

complexity of RST. However, by using I/O-efficient algorithms, multiple cores, GPU computing, and modern cluster resources such a XSEDE, we can reduce the computation time of complex interpolation methods. Furthermore, since RST has nice geomorphological properties such as the ability to easily evaluate derivatives of terrains, our solution could have applications in areas including flow modeling, flood risk assessment and coastal erosion.

The design of our system is independent of the interpolation method used. While we demonstrated that RST in particular benefits from GPU computation, it would be possible to accelerate simpler interpolated methods as well by using only the quad-tree decomposition and distributed computing component using MPI.

When designing such a complex system, it is important to carefully analyze bottlenecks and attempt to balance I/O, computation, and network and inter-process communication. Our approach proposes a framework which helps to decouple some of these concerns and allow us to reduce bottlenecks when processing large geospatial data sets.

References

[1] P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In A. Riedl, W. Kainz, and G. Elmes, editors, *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling*, pages 771–788. Springer-Verlag, 2006.

[2] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.

[3] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.

[4] A. Beutel, T. Mølhave, and P. K. Agarwal. Natural neighbor interpolation based grid dem construction using a gpu. In *ACM GIS '10: Proceedings of the 18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems*, pages 172–181, November 2010.

[5] A. Beutel, T. Mølhave, P. K. Agarwal, A. P. Boedihardjo, and J. A. Shine. Terranni: Natural neighbor interpolation on a 3d grid using a gpu. In *ACM GIS '11: Proceedings of the 19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems*, pages 64–74, November 2011.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications.* Springer Verlag, Berlin, 1997.

[7] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for mpp and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[8] EM Photonics. CULA Dense Version R12, 2012. `http://www.culatools.com`.

[9] Extreme Science and Engineering Discovery Environment (XSEDE). `https://www.xsede.org/` 2012.

[10] Q. Fan, A. Efrat, V. Koltun, S. Krishnan, and S. Venkatasubramanian. Hardware-assisted natural neighbor interpolation. In *ALENEX '05: Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*, pages 111–120, 2005.

[11] G. R. Hjaltason and H. Samet. Speeding up construction of quadtrees for spatial indexing. *VLDB*, 11(2):109–137, 2002.

[12] I. K. E. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proc. of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[13] Khronos Group. OpenCL Version 1.1, 2012. `http://www.khronos.org/opencl/`.

[14] M. McKenney, S. Hill, G. D. Luna, and L. Lowell. Geospatial overlay computation on the gpu. In *ACM GIS '11: Proceedings of the 19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems*, pages 473–476, November 2011.

[15] L. Mitas and H. Mitasova. Spatial interpolation. In P. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, editors, *Geographic Information Systems - Principles, Techniques, Management, and Applications.* Wiley, 1999.

[16] H. Mitasova and L. Mitas. Interpolation by regularized spline with tension: I. theory and implementation. *Mathematical Geology*, 25:641–655, 1993.

[17] H. Mitasova, L. Mitas, W. Brown, D. Gerdes, L. Kosinovsky, and T. Baker. Modelling spatially and temporally distributed phenomena: new methods and tools for grass gis. *International Journal of Geographical Information Systems*, 9:433–446, 1995.

[18] H. Mitasova, L. Mitas, and R. S. Harmon. Simultaneous spline interpolation and topographic analysis for lidar elevation data: methods for open source gis. *IEEE Geoscience and Remote Sensing Letters*, 2(4):375–379, 2005.

[19] NVIDA. CUDA Version 4.1, 2012. `http://developer.nvidia.com/what-cuda`.

[20] Pennsylvania Office for Information Technology, Geospatial Technologies Office, and Pennsylvania State University. Pennsylvania spatial data access (pasda) geospatial data clearinghouse. `http://www.pasda.psu.edu/`.

[21] The Center for Massive Data Algorithmics (MADALGO), Aarhus University, Aarhus, Denmark. TPIE, 2012. `http://www.madalgo.au.dk/tpie/`.

[22] The National Center for Supercomputing Applications (NCSA). Forge dell nvidia linux cluster 2012. `https://www.xsede.org/web/guest/ncsa-forge`.