12-2011

High Performance Geospatial Analysis on Emerging Parallel Architectures

Seth Warn
University of Arkansas, Fayetteville

High Performance Geospatial Analysis on Emerging Parallel Architectures

High Performance Geospatial Analysis on Emerging Parallel Architectures

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

By

Seth Warn
Columbia College
Bachelor of Science in Computer Science, 2006

December 2011
University of Arkansas

Abstract

Geographic information systems (GIS) are performing increasingly sophisticated analyses on growing data sets. These analyses demand high performance. At the same time, modern computing platforms increasingly derive their performance from several forms of parallelism.

This dissertation explores the available parallelism in several GIS-applied algorithms: viewshed calculation, image feature transform, and feature analysis. It presents implementations of these algorithms that exploit parallel processing to reduce execution time, and analyzes the effectiveness of the implementations in their use of parallel processing.

This dissertation is approved for recommendation
to the Graduate Council.

Dissertation Director:

_____

Dr. Amy Apon

Dissertation Committee:

_____

Dr. John Gauch

_____

Dr. Miaoqing Huang

_____

Dr. Jackson Cothren

Dissertation Duplication Release

I hereby authorize the University of Arkansas Libraries to duplicate this dissertation when needed for research and/or scholarship.

Agreed

_____

Seth Warn

Refused

_____

Seth Warn

Acknowledgements

I would like to thank the following people:

- Barbara Webb, my wife, for supporting me when I said "I think I'll go back to school," and in the years since then.
- Dr. Amy Apon, my advisor, for ceaseless support, encouragement, and advice.
- Stan Bobovych and Wesley Emeneker, for fruitful discussions and help with implementations and testing.

The work in this paper was supported in part by NSF grants #918970, #947679, and #072265.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Geospatial Applications Need Parallel Computing

The term *geospatial* describes information that is spatially located relative to the Earth. This broad definition includes a wide range of data, such as historical temperature records, aerial and satellite photography, property records, and more. This data is frequently stored in *geographic information systems*, or *GIS*, which include both the spatially-referenced data as well as the hardware and software used to store, analyze, and display that data. Geospatial applications analyze this data to perform a variety of tasks, such as predicting how a disease will spread, or determining the optimal location for a new cell phone tower.

The increasing capacity of computers to store data, the expanding bandwidth of networks to transmit data, and the growing number and resolution of sensors collecting data result in an explosion in the amount of information processed by geospatial applications. Existing applications designed for serial computer architectures cannot handle this growth in data size, because advances in computer performance are now based primarily on increasing parallelism [1], rather than increasing single-thread performance.

Static single-thread performance will not limit the exponential growth of computational power; for example, single-rack servers capable of petaflop performance will be available in the next five to ten years [2]. However, this level of performance will be provided by massively multi-core processors, with hundreds of cores per chip. Existing tools written for serial architectures, or even tools written for parallel architectures with few cores, must be replaced with algorithms and software designed to take advantage of these new, massively-parallel computers.

Current multi-core chip architectures essentially replicate a small number of traditional CPU cores in a single package. However, the massively multi-core processors mentioned above will

evolve, at least in part, towards a more data-parallel, task-based parallel architecture [3] that is currently exemplified by GPU hardware, described in Section 2.3. Writing applications to take advantage of GPU accelerators provides performance benefits now, while making them compatible with HPC systems of the future.

1.2 Target Applications

The field of geomatics is concerned with a wide variety of problems. The problems addressed by this dissertation were chosen primarily because they are immediately useful to ongoing projects at the University of Arkansas Center for Advanced Spacial Technologies (CAST). Each has proven amenable to some form of parallelization.

Viewshed analysis: A problem in computational geometry whose answer is useful for a variety of geospatial research questions. Briefly, it uses a model of a given terrain to answer questions about line-of-sight for observers within that terrain. It is described further in Chapter 3.

Feature Transformation: A method in computer vision that attempts to convert raw input images into a list of important "features" which are more amenable to further analysis. This dissertation describes a well-known feature transformation algorithm, SIFT, and its implementation in Chapter 4.

Feature dissimilarity: The high-performance SIFT implementation described in Chapter 4 generates large amounts of data, consisting of billions of feature descriptions. A common analysis to perform on those features is to look for similar features derived from other images, which requires a method of measuring feature similarity. One such method is the Circular Earth Mover's Distance, or CEMD. This similarity measurement and its uses are described in Chapter 5.

1.3 Contributions

The increase in the size of geospatial data sets requires faster tools to process that data, while the transition to parallel computer architectures means that existing tools, which are not designed for the new architectures, are not capable of the required performance. This dissertation describes an interdisciplinary project to create geospatial software tools capable of utilizing parallel computer architectures. This encompasses:

1. Examinations of the parallelism available in the three common geospatial applications listed above.
2. Software implementations of those applications designed to take advantage of the parallel architectures described in Chapter 2.
3. Evaluations of the effectiveness of said implementations.

These parallel implementations make it possible to use existing techniques on larger data sets, and enable the creation of new applications by making those techniques practical on a larger scale. Each of these implementations use novel methods to take advantage of parallel computers. In addition, the software developed has immediate applications, and is already used in geospatial research.

Summary of results: Each of the target applications is described in its own chapter; this is an overview of those results:

- Viewshed analysis is accelerated with novel algorithmic improvements and by exploiting multi-core architectures. The speed of the resulting implementation, which can calculate a 16k$\times$16k viewshed in 21 seconds on an inexpensive desktop computer, is orders of magnitude faster than available viewshed analysis software.
- SIFT feature extraction can benefit from all three forms of parallelism described in the next chapter: multi-core, cluster computing, and GPUs. The implementation described in Chapter 4 is a framework for SIFT and other scale-space based feature detectors, which uses a

novel partitioning and pipeline scheme to exploit these parallel architectures in a flexible manner. This is the only implementation of SIFT capable of operating directly on the very large images common in geospatial applications.

- Feature dissimilarity calculation is embarrassingly parallel, making it simple to use cluster computing for weak scaling. Chapter 5 focuses on the details of extracting performance from individual GPUs, and the resulting implementation is shown to be nearly optimal. The GPU code is embedded in an application that uses hybrid clusters to enable CEMD calculation for large numbers of features.

Chapter 2

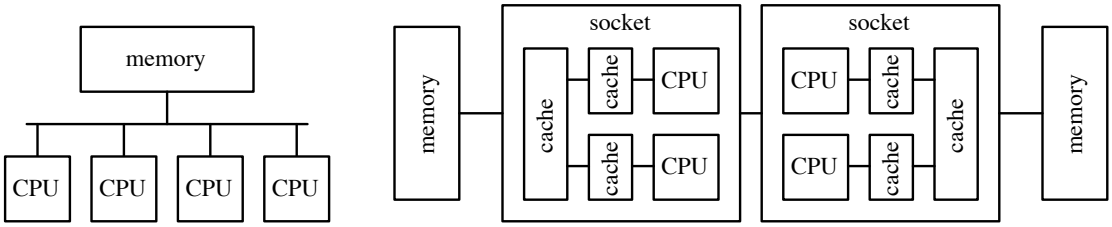
Parallel Computing

Parallel computing refers to wide variety of computing systems where multiple calculations occur at the same time. It includes both parallel computer architectures [4], and the programming models, languages, and APIs (application programming interfaces) used to write software that operates on parallel architectures. This is an enormous topic, and discussing it is made more difficult by the insufficiency of accepted terminology, for example Flynn's Taxonomy [5], to describe modern parallel systems, as well as disagreement over the definitions of such basic terms as *parallel* and *concurrent*. Therefore, this chapter restricts its description of parallel computing to only those architectures and software constructs directly relevant to the applications presented in later chapters.

2.1 Shared Memory Systems

Parallel computers may be divided into two groups, according to the configuration of their memory. The first group are *shared memory* computers; as the name suggests, all processing elements have shared access to the computer's memory. The most common examples of these are *symmetric multiprocessing*, or SMP, computers, where multiple identical CPUs are connected to a common system memory and controlled by a single operating system. The CPUs may be together in a single package and share a single motherboard socket, or they may be distributed among several sockets.

Threaded Programming: The canonical method of programming in order to take advantage of SMP systems is *threaded programming*. This is not, strictly speaking, a parallel programming model. Instead, some number of *threads*, which are sequences of instructions like a serial program, that share access to resources such as memory, and that execute *concurrently*: each thread proceeds independently of the others. This may be implemented by executing instructions from each thread in an interleaved fashion on a single CPU, or simultaneously executing instructions from multiple



(a) Logical View (b) A possible implementation

Figure 2.1: Shared memory, as it is modeled and implemented

threads on several CPUs. This nondeterminism is managed with a variety of constructs, such as mutexes, semaphores, and monitors.

POSIX threads [6] are a widely-used C API for threaded programming. Other APIs, such as Java threads [7], Boost/C++ threads [8], or Windows threads [9], allow similar operations. These APIs present a very simple abstraction of the underlying architecture, as shown in Figure 2.1(a): each CPU has equal access to a shared memory pool, thus the physical site of thread execution is irrelevant. A somewhat more realistic, but still simplified, depiction of an SMP system is shown in Figure 2.1(b). This system shows how threads may not have equal access to memory resources, and how thread placement may affect thread execution, for example by sharing cache resources. The effect on execution time can be significant [10]; libraries such as libNUMA [11] allow control over these sorts of execution details.

Alternatives to Threads: The threaded programming model mimics the shared-memory architecture, and is therefore natural, but is often difficult to use correctly [12]. A number of methods have been designed to make using shared memory systems easier. One of the most widespread is OpenMP [13], an API for C, C++, and Fortran. Typical OpenMP programs will not explicitly create a number of threads. Instead, the programmer specifies that some operation, such as a reduction or the iterations of a loop, should be done in parallel. The OpenMP will distribute this work to multiple threads to take advantage of multiple CPUs, but this is transparent to the programmer, who is thus freed from much of the concurrency management required in threaded programming.

A common pattern in threaded programming is the *thread pool*. In a thread pool, there are

a number of threads, typically equal to the amount of physical parallelism available, that is, the number of CPUs. The parallel application is broken into a number of *tasks*, which are units of work than can be performed concurrently. Typically, there are far more tasks than threads, an they will be stored in queues. Each thread dequeues tasks, completes them, and dequeues another task, until the pool of tasks is exhausted. Creating and destroying threads typically incurs significant operating system overhead; task queueing and dequeuing is typically far faster. This allows parallelism to be specified at a relatively fine level.

Instead of manually implementing thread pools, programmers can take advantage of libraries and compiler extensions that, like OpenMP, handle implementation details, allowing the programmer to simply specify the parallelism in the code. Cilk [14] was an early example of this approach, now commercialized as Cilk Plus [15]. More recent examples include Intel's Thread Building Blocks [16], Apple's Grand Central Dispatch [17], and Microsoft's Task Parallel Library [18]. Concurrent languages like Google's Go [19] allow specification of task-level parallelism with built-in language constructs.

2.2 Distributed Memory Systems

In contrast to shared memory systems, where all processors have access to the same memory, *distributed memory* describes a system where each processor has access to a private memory. In a distributed memory system, accessing memory other than the processor's own private memory requires communication with another processor. A common example of distributed memory is the computer *cluster*, shown in Figure 2.2: a collection of computers, typically linked by a high-speed local network, and working closely together.

The Message Passing Interface [20], or MPI, is the *de facto* standard for cluster programming. As with threaded programming, computation is specified as a set of instruction sequences running concurrently, with a number of mechanisms for synchronization. These sequences are called *processes*; it is usual for each process in an MPI application to be running identical code, so MPI is sometimes called *SPMD*, or single-program, multiple-data. Like OpenMP, the programmer typi-

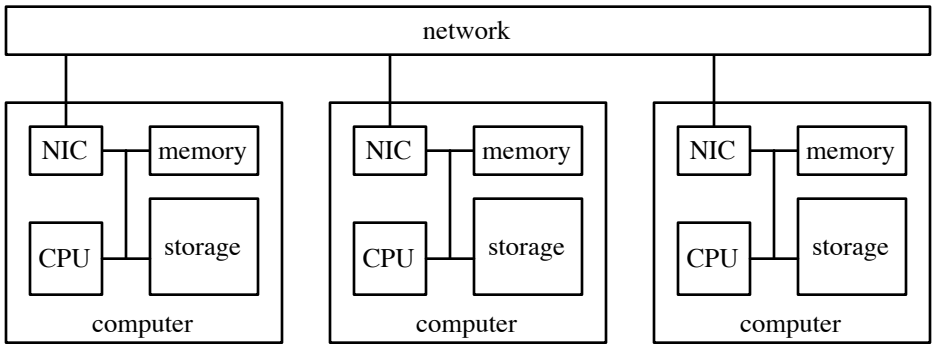


Figure 2.2: A cluster computer

cally does not specify a number of processes at compile time; and will instead write partitioning logic that detects the number of processes at runtime and divides work among them. MPI is an API that handles the management of, and communication between, these processes.

The division between shared memory systems and distributed memory system is not black-and-white. In modern clusters, there are multiple CPUs per compute node, which share memory within the node, but not with CPUs on other nodes. Frequently this is simply ignored; a single MPI process is created for each CPU, and the underlying MPI runtime may or may not optimize communications between processes executing on the same node. Shared memory and message-passing may also be explicitly combined, for example, by launching a single MPI process per node, then using OpenMP in each process to exploit the multiple CPUs in each node.

Another approach is to represent the collective memory resources of the cluster as a single, non-uniform memory, called a *partitioned global address space*, or PGAS. This programming model is the subject of ongoing research, and is used by a number of languages such as X10 [21] and Chapel [22].

2.3 GPGPU

Dedicated graphics processing hardware has existed for many years. Computer chips designed solely to handle graphics are called GPUs, or graphics processing units. GPUs exist in many forms; they are integrated with CPUs and other hardware in systems-on-chip designed for mo-

Table 2.1: Comparing GPU and CPU performance.

	NVIDIA GeForce 580	Intel Core i7 950	Intel Core i7 970
GFLOPS[a]	1581	98.2	154
Price[b]	$500	$300	$900
TDP	244 W	130 W	130 W
Memory BW	192 GB/s	25.6 GB/s	25.6 GB/s
GFLOPs/$	3.16	0.33	0.17
GFLOPs/W	6.48	0.76	1.18

[a] Theoretical peak, single-precision, multiply and add

[b] As of January, 2011

bile computing, as separate chips on a motherboard, and most commonly as daughterboards that combine a GPU, dedicated video memory, and video output hardware, called video cards.

The last decade has seen growing interest in the use of these video cards for non-graphical computation. This usage is now called GPGPU, or general-purpose computing on GPU. The rising popularity of GPGPU is due to the impressive performance of GPU processors, both in absolute terms and in relative efficiency. Table 2.1 above shows high-end, consumer-oriented GPU and CPU parts from NVIDIA and Intel, respectively. This sort of direct comparison ignores many important details, and even limited GPU–CPU comparisons are notoriously difficult; however, the order-of-magnitude differences in performance-per-dollar and performance-per-watt have driven widespread and growing adoption of GPUGPU programming.

2.3.1 CUDA

The Compute Unified Device Architecture, or CUDA, is a parallel architecture [23, 24] and programming model [25] designed by the NVIDIA corporation and implemented in their GPU product line. NVIDIA also provides a language called "C for CUDA" [26], compiler, and libraries used to program their GPUs; these are often informally referred to as part of CUDA.

Developing CUDA was NVIDIA's response to an already-active GPGPU programming community. While instances of using repurposed graphics hardware can be found going back many years, the current form of GPGPU programming began after 2000 with the addition of programmable shaders to commodity video cards. Programmers discovered that they could encode non-graphics

problems as pixels and shader operations using shader languages like OpenGL's GLSL or Microsoft's HLSL and execute them on a GPU. This proved useful but tedious, and by 2004, several higher-level languages [27, 28] had emerged. These languages looked more like traditional C or C++ and were compiled into shader code. NVIDIA released CUDA in early 2007 and has since release several updates to add features requested by the GPGPU community.

Programming Model: CUDA applications are based on the execution of kernels on the GPU device. The kernels are defined using C/C++ and are executed many times in parallel; each kernel instance is called a thread in CUDA, although these threads are not the same as POSIX-style threads. Like other SPMD models such as MPI, the threads have access at runtime to a unique identifier, called the "thread ID" in CUDA. The threads are grouped in a programmer-specified hierarchy, with the hundreds of threads grouped into blocks and potentially thousands of blocks grouped into a grid.

The thread hierarchy is a convenient way to organize execution: both the blocks and grids can be one-, two-, or three-dimensional to suit the needs of the application. Thread blocks are also reflected in CUDA's execution model and memory hierarchy. First, threads can use lightweight synchronization mechanisms to cooperate with other threads in the same block, but the blocks must be able to execute independently. Second, threads have access to a small, fast pool of memory, called "shared memory," that is visible to all threads in their block, and only persists for the duration of the block's execution.

Shared memory is one of several types of memory in the CUDA model. Global memory is the other primary memory pool, which is visible to all blocks, and is much larger and slower than shared memory. While researchers have implemented barriers [29] and work queues [30] at the grid level, CUDA has no built-in mechanism for grid-level synchronization, so each block will typically write to a separate region of global memory. Threads may either explicitly copy data between these memories, or the shared memory can act as a cache. There are another two memory types, constant and texture memory, that are read-only to threads and are used for more specialized

purposes.

The global memory pool is also the only memory accessible to the "host," the computer running the CUDA-accelerated application. The typical execution path of a CUDA program involves some host code execution, copying input data from the host memory into the CUDA device's global memory, and launching a kernel. The results of kernel execution are placed back into global memory, then copied into the host's memory, which proceeds with normal execution. These copies typically take place over a PCIe bus, which has far less bandwidth than the memory systems of the host or device, and minimizing these host–device copies is a primary design concern for CUDA developers.

Architecture and Optimization: The implementation of CUDA hardware depends heavily on data parallelism and latency-hiding, which affects how CUDA programs are written. CUDA GPUs consist of a number of streaming multiprocessors, or SMs, each of which has a number of instruction-locked processors called "CUDA cores." Nvida describes the CUDA architecture as "Single Instruction Multiple Thread," or *SIMT*. All threads in a block execute on the same SM; the instruction-locked cores mean that divergent behavior among threads in a block is simulated by masking off operations on some cores. This means that avoiding divergence is another design priority for developers.

The global and shared memory pools have their own quirks, as well. For example, the shared memory is arranged in a series of 32 banks, each of which can be accessed in parallel. The address space is striped across these banks in four-byte words; writes to the same bank are serialized. Data structures and algorithms must be designed to avoid these "bank conflicts." The global memory has its own set of rules governing which memory operations can occur in parallel, "coalescing" the operations into a single memory transaction. The performance penalty associated with suboptimal memory access patterns means that data structures must be carefully considered when writing GPU kernels, and additional computational work may be offset if it leads to better access patterns [31].

Typically, far more threads are resident on an SM that are currently executing. Each SM can

switch threads without any overhead; this mechanism is used to hide the latency of memory operations, and even regular instructions [32, 26]. Maximizing the architecture's ability to hide latency, by avoiding synchronization instructions and tuning the number of threads in a block, is another important design consideration for CUDA applications.

2.3.2 Alternatives to CUDA

Currently, CUDA is the most commonly-used GPU framework for high-performance computing. OpenCL [33] is a popular and growing alternative which defines a language and programming model similar to CUDA, but is vendor-agnostic. Code written using OpenCL is typically somewhat slower than code using CUDA [34]. Microsoft has also released their own framework, DirectCompute, as part of their DirectX [35] family of APIs. PGI's CUDA Fortran [36] replaces the C for CUDA language with a Fortran-based equivalent.

Some alternatives seek to add a level of abstraction above CUDA, rather than replace it. The Thrust [37] project is a C++ template library that allows the creation of CUDA code using data-parallel primitives, such as scan, sort, or reduce, with interfaces similar to the C++ STL (standard template library). The *hi*CUDA [38] language uses C pragmas and a source-to-source compiler to automatically handle some CUDA programming tasks. OpenMPC [39] also uses pragmas, building on OpenMP, to accomplish the same goal. Other alternatives run CUDA on a different underlying architecture, such as an FPGA [40, 41] (field programmable gate array), or even traditional x86 CPUS [42].

Other threaded, data-parallel architectures have been created in response to flat single-thread performance. AMD's Accelerated Parallel Processing (APP) framework [43] for GPGPU is the closest equivalent to CUDA. IBM's Cell [44] and Intel's Larrabee [45] both combine a large number of simple CPUs with wide vector units. Both AMD and Intel have started producing desktop CPUs that integrate GPU cores [46, 47], and Intel's AVX [48] is increasing the SIMD capabilities of their desktop CPUs. Several studies have compared performance among the architectures [49, 50].

2.3.3 Applications of GPGPU

Given the high FLOPs (floating point operations per second) capacity of GPUs and their data-parallel architecture, many have used GPUs for linear algebra and dense matrix operations. CUBLAS [51] and MAGMA [52] implement a CUDA-accelerated BLAS interface which can be used with HPL [53]; while the raw performance is impressive, the efficiency (measured flops vs. theoretical peak) is lower than CPU-based clusters [54, 55]. Sparse matrix operations have also been accelerated by finding data structures that efficiently encode such matrices for GPU processing [56, 57].

GPUs are being used for a wide variety of other applications. These range from sorting [58] and FFT (fast Fourier transform) calculation [59] to biomolecular simulation [60], sequence alignment [29], and quantum chromodynamics [61]. As GPUs become more common, they are also being used in non-HPC applications such as real-time systems [62] and databases [63].

2.4 Hybrid Clusters

Cluster computers and GPUs are both attractive platforms for high-performance computing due to their low price/performance ratios. Early GPU-equipped clusters [64] predate CUDA, but are identical to modern hybrid clusters in configuration. Where a typical cluster is group of commodity server nodes, frequently connected by a high-performance interconnect, a hybrid cluster attaches one or more GPUs to each node as daughterboards connected via the PCIe bus.

This model of normal-computer-plus extends to programming these clusters, as well. The GPUs cannot communicate directly with other GPUs, even on the same node, so hybrid-cluster applications function essentially identically to other cluster applications. They will typically use MPI; the individual MPI processes will accelerate selected functions on a GPU. Because moving data from GPU to GPU must always involve copying from a GPU, to a CPU, then a network transfer, then another CPU to GPU transfer, minimizing communication and hiding communication latency are even more important to hybrid applications than other cluster software. MPI libraries and cluster scheduling software are still largely GPU-agnostic, which causes some difficulties [65],

but these systems are being updated to be GPU-aware.

The TSUBAME computers in Japan were some of the first to adopt GPU use for large, production compute clusters — TSUBAME 1.2 was the first GPU-equipped cluster on the TOP500 list, in 2008. Now, TSUBAME 2.0 is one of the three GPU-equipped computers in the of the top five systems listed by TOP500 [66].

Hybrid clusters are being applied to a range of computing problems, starting with basic tools such as Linpack [55], and on to N-body simulation [67, 68, 69], molecular dynamics [70], weather simulation [71], and more. These authors all report that the design considerations of typical GPGPU programming, such as minimizing PCIe communication, remain important; several also emphasize the importance of overlapping GPU and CPU computation, so that the considerable CPU power of these clusters is not wasted.

The term "hybrid clusters" applies not only to GPU-equipped clusters, but to any cluster that executes code on more than one type of architecture. Like the GPU clusters above, these computers will usually have CPU-equipped nodes with one or more accelerators. The most well-known is Roadrunner [72], which use Cell processors in addition to AMD CPUs. Other accelerators have been used, such as FPGAs [73] and specialized floating-point accelerators from ClearSpeed, which were also used in TSUBAME.

Chapter 3

Viewshed Analysis

This chapter describes a new viewshed analysis algorithm, and a parallel implementation of that algorithm. Viewshed analysis is described in Section 3.1, while the rest of the chapter describes the new algorithm; an implementation of it named "pvshed," made as part of this dissertation; and the performance of pvshed.

The new implementation described here is orders of magnitude faster than previous viewshed analysis software. This speedup is partially due to algorithmic improvements; additionally, pvshed can take advantage of the SMP systems described in Section 2.1. This additional speed is useful, both to allow current analysis methods to complete in reasonable times on larger data sets, and to enable the development of new algorithms that require repeated viewshed analysis, which are impractical with slower viewshed analysis.

3.1 What is Viewshed Analysis?

A *viewshed* is the terrain visible from a fixed viewpoint: "If I stand here, what can I see?" Viewshed analysis has many applications: placement of communication [74] or radar facilities [75, 76], which require good visibility; calculating the "zone of visual influence," areas where a development will have a visual impact; designing nature trails for maximum visibility [77, 78]; correcting aerial or satellite imagery for illumination variances [79]; and finding the fewest required observers to watch a given terrain [80] are among its uses.

Viewshed analysis processes a *digital elevation model*, or DEM, to determine the viewshed from a given viewpoint. DEMs are typically in one of three forms: Triangulated irregular networks, or TINs; *heightmaps*, also called raster or grid DEMs; and contours [81]. Heightmaps store elevation data for a terrain in a regularly-spaced array of samples, and are more common in modern geographic information systems (GIS) applications. The expansion of remote sensing

technologies is creating ever-growing amounts of DEM data. For example, the National Elevation Database [82] provides raster DEM data at 30-meter resolution or finer for the entire continental United States. LIDAR (light detection and ranging) uses laser pulses to measure range to a target; satellite- or aircraft-mounted LIDAR is now providing DEM data at 1-meter and even sub-meter resolution.

Strictly speaking, viewshed analysis calculates a boolean answer for each height sample, visible or obstructed, and may be represented by a matrix of single bits that correspond to the terrain samples. However, GIS systems are typically interested in not just a boolean answer, but the *maximum obscured height*; that is, how tall a pole can be placed at that point on the terrain and still be unseen from the viewpoint? Because visibility is bidirectional, viewshed analysis can also be used to answer the converse of the original question: "If I stand here, what can see me?"

3.2 Viewshed Background

The viewshed analysis problem was described at least as early as 1986 [83], and applications of viewshed analysis were explored over several years [84, 85]. In these papers, TIN models were preferred. Over time, the heightmap became the preferred form for DEMs. The viewshed for a grid DEM is as follows: The DEM defines the input terrain and consists of a set of height samples. Each sample is a three-dimensional point, and the samples are spaced in a regular grid above the xy plane. The viewshed is the set of all the height samples visible from the viewpoint. There are varying methods of determining visibility between two points, but in general, a target point T is visible to a viewpoint V if no part of the terrain intersects the line segment $\overline{VT}$.

3.2.1 Brute Force and Approximate Methods

The most straightforward approach to viewshed analysis of heightmaps is simple; visibility for each sample point is determined by checking all height samples between the viewpoint and the target sample point. If and only if all intervening samples are below the *sightline* connecting the viewpoint and the target, the target is visible. For a terrain S with $n \times n$ samples, this is an $O(n^3)$

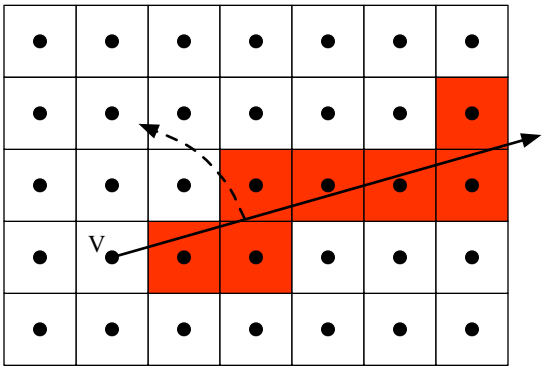


Figure 3.1: The planar sweep described by van Krevald. Looking down onto the terrain, a half-line is rotated around the viewpoint V. Pixels (shown in red) that intersect the line are stored in the status structure, ordered by distance from the viewpoint.

algorithm: for each of the n^2 samples, $O(n)$ intervening samples between the viewpoint and the target sample must be checked.

Because the brute force approach requires a great of computation time, several authors explored approximate methods. The Xdraw method [86] was highly influential. It operates by growing a "ring" of line-of-sight information around the viewpoint; the sightline to a target is determined by interpolating between the two samples immediately closer to the viewpoint. This approach has the disadvantage of diffusing irrelevant information — through repeated interpolation, samples far from the sightline contribute to the line-of-sight determination.

3.2.2 The Planar Sweep Approach

Later, van Krevald described [87] how to perform viewshed analysis in $O(n^2 \lg n)$ time using the *planar sweep* concept from computational geometry. The new approach described later in this chapter is also based on the planar sweep. Van Krevald considered the height samples as the centers of square pixels, as shown in Figure 3.1. Conceptually, van Krevald's algorithm rotates a half-line around the viewpoint in a full circle. The pixels beneath the sweep line are stored in a *status structure*, added as the sweep line first intersects the pixel, then removed as the sweep line leaves the pixel. The status structure is used to determine the visibility of each pixel; this is described in more detail, below.

The Status Structure: Key to this method is the use of a self-balancing binary search tree, such as a red-black tree, as a status structure to track the cells pixels under the sweep line. The pixels are ordered within the tree by their distance from the viewpoint. A node in the tree stores a single pixel. The tree is augmented by associating an extra number with each node. For the leaf nodes, this number is the slope from the viewpoint to the height sample represented by the event stored at that node. For internal nodes, the number stored is the maximum of the two slopes stored by its two children. These numbers are updated bottom-up after each update, so that internal nodes will always store the maximum slope of their subtree, and any update takes $O(\lg n)$ time.

This augmentation allows van Krevald's status structure to answer visibility questions in $O(\lg n)$ time. The tree is ordered so that leftmost events are closer to the viewpoint. During tree traversal, each time the right child is chosen, the left child is the root of a subtree whose members are all closer to the viewpoint. Each of these left child nodes records the maximum slope of its subtree, and there will be at most $O(\lg n)$ left nodes during tree traversal. The maximum of these values is the greatest slope between a target point and the viewpoint. This greatest slope can be used to determine visibility: the product of the slope and the horizontal distance to the target is the maximum obscured height at the location of the target. This is discussed in more detail in Subsection 3.3.1.

The Sweep: Using the status structure, van Krevald describes the sweep algorithm as follows:

1. Each sample is considered the center of a pixel. For each pixel, record three events: an "add" event at the point when the sweep line first crosses the pixel, a "search" event when the point when the sweep line crosses the center of the pixel, and a "remove" event at the point when the sweep line leaves the pixel. There are n^2 pixels, which means there are $3n^2$ events; listing these events takes $O(n^2)$ time.
2. The angle of each event is measured between a line defined by the viewpoint and the event, and a line parallel to the x-axis intersecting the viewpoint. Sort the events by their angle; sort events with the same angle by distance from the viewpoint. This has the effect of listing all events as they appear from the viewpoint, turning counter-clockwise. Sorting the $3n^2$ events

takes $O(n^2 \lg n)$ time.

3. Simulate the planar sweep by repeatedly removing events from the beginning of the list and processing them. When processing "add" or "remove" events, the pixel in question is added or removed from the status structure as described above. Processing a "search" event corresponds to the sweep line passing directly over the center of a pixel. As the sweep line crosses the center of pixel P, the status structure stores all the pixels between the viewpoint and P, and the maximum intervening slope is used to determine the visibility of P. Each of the $3n^2$ events requires a $O(n^2)$ status structure operation, so the sweep takes $O(n^2 \lg n)$ time.

The three phases of the algorithm, listing events, sorting events, and processing events, take $O(n^2)$, $O(n^2 \lg n)$, and $O(n^2 \lg n)$ time, respectively. The combined execution time is also $O(n^2 \lg n)$.

Van Krevald did not implement his algorithm, and several authors continued to explore alternatives, such as approximate methods [88, 89, 90, 91, 92] or GPU acceleration [93]. However, some work [94, 95, 96] has used the rotational-sweep approach while focusing on minimizing I/O operations for large DEMs.

3.3 A New Approach

The new viewshed analysis algorithm described in this chapter is based on the planar sweep and shares the $O(n^2 \lg n)$ runtime for $n \times n$ grid DEMs. However, it uses a number of techniques to increase performance. First, it avoids the use of computationally-expensive trigonometric and square root functions to determine visibility and for the planar sweep, as described in Subsection 3.3.1 and Subsection 3.3.5, respectively. Second, it uses a status structure, described in Subsection 3.3.6, which is optimized for for visibility calculations on grid DEMs. Third, it parallelizes the viewshed computation, as described in Subsection 3.3.8, to take advantage of SMP computers.

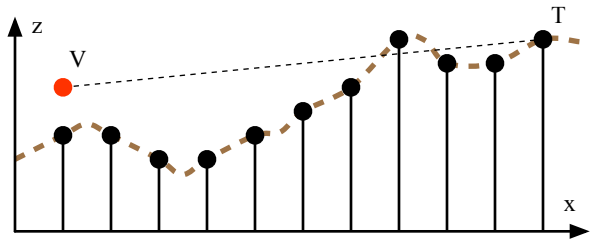


Figure 3.2: Calculating visibility of T from viewpoint V. The original terrain is shown in profile as a heavy dashed line; the terrain is represented as an array of height samples, shown as black dots.

3.3.1 Determining Visibility

Approaches to viewshed analysis vary in how they determine the visibility of a target point from the viewpoint. The method used in this chapter is described below, starting with a simplified version where the xy-projections of the viewpoint, target, and height samples are all collinear, then generalizing to visibility over a two-dimensional grid.

Simplified Visibility: Consider determining the visibility of a target point T from a viewpoint V. Figure 3.2 shows an example where T is not visible from V, because the line segment $\overline{VT}$, also known as the *sightline*, intersects the terrain between V and T. In general, T will be visible if and only if all samples between V and T are below the sightline.

This condition can be verified by comparing the slope of $\overline{VT}$ with the slope from V to the intervening samples. T is visible from V if and only if

$$\frac{T_z - V_z}{T_x - V_x} > \frac{S_z - V_z}{S_x - V_x}$$

for all samples S such that $V_x < S_x < T_x$. To perform this comparison, many viewshed algorithms will first calculate the angle of the slopes using the arctangent, then compare the values of the slopes. In effect, this is determining the result of the above comparison by evaluating:

$$\arctan\left(\frac{T_z - V_z}{T_x - V_x}\right) > \arctan\left(\frac{S_z - V_z}{S_x - V_x}\right).$$

However, the arctangent is a computationally expensive function, and there is no need to obtain the angle of the slope. The approach in this chapter avoids using the arctangent by storing the slopes as a rational data type; that is, it simply stores the vertical and horizontal differences. Slopes stored in this fashion can be compared by cross-multiplying:

$$(T_z - V_z) \times (S_x - V_x) > (S_z - V_z) \times (T_x - V_x).$$

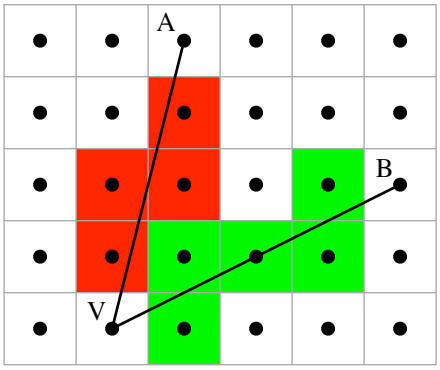
The actual angle of the slope is never calculated.

If the maximum of all the intervening sample slopes is known, then visibility can be determined with a single comparison. If the sightline is above the intervening sample with the greatest slope, it will also be above all other intervening samples. Also, the maximum intervening slope can be used to calculate the obscured height at a target; for a maximum slope m, the obscured height h at T_x is determined by

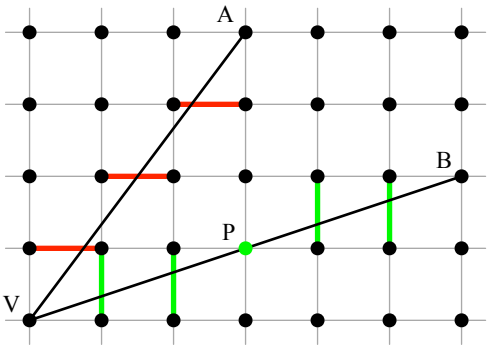
$$h = V_z + m \times (T_x - V_x) - T_z. \tag{3.1}$$

Visibility Over a Grid: Computing visibility over a grid of height samples adds complexity, because the sightline does not usually pass directly over intervening samples. This raises two questions: what points along the sightline should be checked to see if the obstruct the sightline, and what height values should be used at those locations? Here, it is useful to compare to van Krevald's approach, shown in Figure 3.3(a). He considered each sample point to be the center of a square cell. If the line segment $\overline{VT}$ crosses a cell with center P and the slope of $\overline{VP}$ is greater than the slope of $\overline{VT}$, then T is not visible from V.

In contrast, Figure 3.3(b) illustrates the strategy used in this chapter. It checks for obstructions where the xy projection of the sightline intersects the sample gridlines. For a given sightline, only intersections with either x- or y-gridlines are checked, choosing based on which results in the greater number of checks. Put another way, checks are performed at grid intersection along the longest axis of the sightline. Where the intersection with a gridline coincides with a sample, as occurs at P, the sample height is used to check for obstruction. However, in most cases the



(a) In van Krevald's approach, each height sample is interpreted as a cell. A cell may obstruct any sightline passing over it. Sightline $\overline{VA}$ is potentially blocked by any of the red cells; sightline $\overline{VB}$ can be blocked by any of the green cells.



(b) Top-down view of example sightlines $\overline{VA}$ and $\overline{VB}$. Intervening values are interpolated between the pairs of samples marked in red for $\overline{VA}$ and green for $\overline{VB}$. No interpolation is required at P, where the sightline passes directly over a height sample.

Figure 3.3: Interpolating heights over a raster DEM

intersection with a gridline will occur between two samples. In this case, a height is interpolated based on those two points, which are the nearest neighbors along the gridline. The interpolation method is discussed in Subsection 3.3.2.

Determining the true slope of a line segment over a grid requires calculating the Euclidean distance between sample points. Consider Figure 3.4: the slope of $\overline{VP}$ is

$$m_{\overline{VP}} = \frac{P_z - V_z}{\sqrt{(P_x - V_x)^2 + (P_y - V_y)^2}}.$$

Comparing $m_{\overline{VP}}$ and $m_{\overline{VT}}$ to determine visibility requires multiple uses of the square root function and is computationally expensive. Since the objective is only to compare the slopes, not to determine their actual values, only comparing the x- or y-components of the slopes is required.

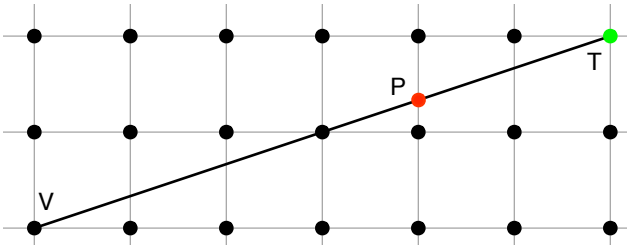


Figure 3.4: Component distances

For example, the x-components of the slopes can be compared using

$$\frac{P_z - V_z}{P_x - V_x} \stackrel{?}{\geq} \frac{T_z - V_z}{T_x - V_x}.$$

This correctly determines which slope is greater, while using considerably less arithmetic.

3.3.2 Interpolation

When a line of sight passes between two sample points, some form of interpolation is necessary. This chapter uses one of the simplest possible methods, the minimum function: the height between two neighbors is considered to be level with the lowest of those neighbors. This has several advantages. First, determining the minimum is computationally inexpensive, while also obviating any need to calculate the distance to the neighbors. Figure 3.5 demonstrates the second and more important advantage: all sightlines passing between two neighbors can reuse the same interpolated value.

Using minimum-based interpolation may seem like an oversimplification, but it is practical. Where interpolation occurs far from the viewpoint, or where there is a small difference between neighbors, the choice of interpolation method will make little difference in the viewshed. Where there are dramatic differences in elevation near the viewpoint, the interpolation method can make a large difference, but no interpolation method is guaranteed to be more accurate. The choice of minimum is meant to be conservative: any point that is found to be hidden is more likely to be hidden. The maximum function may be used instead, to bias the analysis so that visible points are

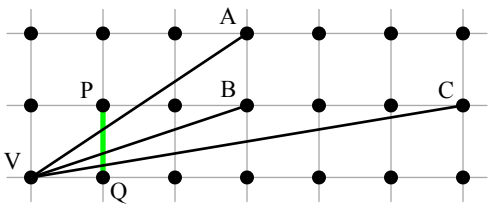


Figure 3.5: Sightlines $\overline{VA}$, $\overline{VB}$, and $\overline{VC}$ all require an interpolated value between P and Q. Interpolating using the the minimum of the heights at P and Q, all three can use the same value.

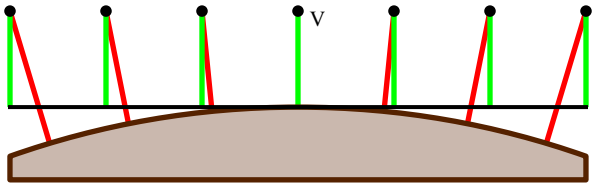


Figure 3.6: Correcting for Earth's curvature: The original heights, in red, are relative to the surface of the Earth. These heights are mapped to new heights, in green, relative to a plane tangent to the surface of the Earth at the viewpoint.

very likely to be visible. A rough margin of error can be established by performing the analysis twice, once with each method [86].

3.3.3 Corrections for Earth's Curvature and Light Refraction

Viewshed as described above considers the input heightmaps as a set of heights above a plane. In reality, height samples are relative to the curved surface of the Earth. For some applications, the error introduced by ignoring the Earth's curvature will be acceptable; the controlling factors are the accuracy required by the application and the maximum view distance. At one kilometer, the vertical error is 7.8 cm; at 16 km, the vertical error is 20 m.

For applications where this error is not acceptable, planar viewshed models can still be used. The heights relative to the surface of the Earth are mapped to height relative to the surface of a plane tangent to the Earth directly beneath the viewpoint. This is illustrated in Figure 3.6 and uses several approximations:

- The Earth is modeled as a sphere of radius R, where $R = 6371$ km. The accuracy of this approximation is discussed elsewhere and beyond the scope of this paper.
- The distance between samples along the plane is considered equal to the distance along the surface of the Earth. For a horizontal distance from the viewpoint of 16 km, this translates to a horizontal error of less than 3.4 cm.

Practically, this process takes an input heightmap and produces a new heightmap where samples further from the viewpoint are lower, simulating Earth's curvature. Viewshed calculation can proceed as if all heights are relative to a reference plane.

The adjustment for a given height sample is derived as follows: consider a height sample S, of height h from the Earth's surface, and which is a horizontal distance d from the viewpoint V. Let the angle θ be $\angle SEV$ where E is the center of the Earth, and let R be the radius of the Earth. A point on the Earth's surface at θ is $R\cos\theta$ distant from E along the $\overline{EV}$ radius, while the Earth's surface at V is R along the same radius. Therefore, the surface at V appears to be $R - R\cos\theta$ lower than the surface at V. Subtracting this amount from h gives the apparent height h':

$$h' = h - (R - R\cos\theta). \tag{3.2}$$

To avoid computationally expensive trigonometric calculations, the $\cos\theta$ term can be approximated with the first terms of the Maclaurin series of the cosine, $1 - (\theta^2)/2$.

$$h' = h - R\left(\frac{\theta^2}{2}\right).$$

The error introduced by this approximation is no larger than the product of R and the next term of the Maclaurin series, $\theta^4/24$. For a distance along the surface of 16 km, this is a maximum vertical error approximately equal to 0.01 mm. Because $\theta = d/R$, the equation can be rewritten as:

$$h' = h - \frac{d^2}{2R}.$$

Refraction has the effect of making the Earth appear to have less curvature by bending light and other EMR along the Earth's surface. This is represented by adding a coefficient n to the previous equation:

$$h' = h - n\left(\frac{d^2}{2R}\right). \tag{3.3}$$

The value of n varies based on atmospheric conditions and the wavelength of the radiation; the value of $6/7$ is typically used for visible light.

3.3.4 Integer Math

An interesting property of this approach to visibility is that it need not use floating-point arithmetic to maintain precision. The input heights can be represented using fixed-point numbers; for example, heights can be stored as an integral number of centimeters, rather than fractional meters. Horizontal distances are measured in terms of integral numbers of grid intervals. Slopes are stored as the ratio of these values. The output is calculated using Equation 3.1, where division by the denominator of m causes a single loss of precision due to integer division. This loss of precision will be less than a single unit of height. However, if the curvature correction described in the previous section is used, floating-point arithmetic may be required.

The implementation described later in this chapter showed little difference in performance between using fixed- or floating-point arithmetic, and fixed-point is used for the results shown in Section 3.4. Lack of floating-point arithmetic makes this approach well-suited for systems that lack FPUs.

3.3.5 Planar Sweep

Like van Krevald's algorithm, the approach in this chapter is based on a rotational planar sweep. Conceptually, the viewpoint and input DEM are projected onto the xy-plane, and a half-line is rotated around the viewpoint. This approach maintains a list of the interpolated values currently under the sweep line in the status structure. The status structure is updated when the sweep line crosses over DEM sample points. When the status structure changes, the change is called an *event*; the list of all events in the order they will be processed is the *event list*. The event list, the status structure, and how they are used, is described in more detail below.

Event List: First, an unsorted list of events is created. There is one event per height sample in the DEM, and each event is simply the row and column of the sample. Then, the samples are sorted based on their angle with the viewpoint, as shown in Figure 3.7. As before, avoiding expensive trigonometric functions will reduce execution time. Again, calculating the actual angle is

17	15	14	12	11	9	8
18	16	13	10	7	6	5
20	19	V	1	2	3	4
21	22	23	24	25	26	27

Figure 3.7: In the event list, samples in the grid are ordered as they appear from the viewpoint, rotating counter-clockwise. Collinear samples, as with samples 1–4, are sorted by ascending distance from the viewpoint.

unnecessary. Instead, the cross product $\overrightarrow{VA} \times \overrightarrow{VB}$ can be used to determine ordering, by calculating

$$(A_x - V_x) \times (B_y - V_y) - (B_x - V_x) \times (A_y - V_y).$$

If the result is positive, B is counter-clockwise from A; if the result is negative, B is clockwise from A.

The cross product alone cannot determine the ordering; there are several special cases to consider. First, the viewpoint divides the surrounding samples into four quadrants. A vector in the fourth quadrant is clockwise from a vector in the first quadrant, but the quadrants should be swept in order; that is, all events in the first quadrant should be swept before all events in the second quadrant, etc. The quadrant of any given sample can be determined by comparing the row and column indices of the samples with indices of the viewpoint. If they are in different quadrants, there is no need to calculate the cross product. Second, if the cross product is zero, then the samples are collinear with the viewpoint. In this case, the sample closer to the viewpoint is first in the event list ordering. As above, calculating the Euclidean distance, which requires calculating a square root, can be avoided by using the difference along the x- or y-axis to determine which sample is closer to the viewpoint.

In pvshed, this ordering logic is encapsulated in a *function object*, which is an object that can be called as though it were an ordinary function. In C++, function objects are instances of any class that overloads `operator()`. Function objects are frequently called *functors*, though they should

not be confused with the mathematical concept of the same name. The pvshed implementation of the sort described above uses a functor that stores the location of the viewpoint and can be used to compare two points to determine their ordering. The C++ STL provides several sorting functions that allow the use of user-defined functors for comparison, so pvshed is requires no additional sorting logic beyond that provided in the STL.

The viewshed area can be limited by omitting events from the initial, unsorted event list. One common example is restricting viewshed to a maximum visual radius. This can be accomplished by generating events for only the samples within the specified radius of the viewpoint, rather than for every sample in the input DEM. The rest of the algorithm remains unchanged.

3.3.6 Optimized Status Structure

The status structure holds a dynamically-maintained ordered list of the events as they occur along the sweep line. In most sweep algorithms, the status structure is some type of self-balancing binary search tree, such as a red-black tree, which allows updates and queries in logarithmic time. Self-balancing binary search trees have optimal asymptotic time complexities, but several factors can reduce the performance observed from real-world implementations of these data structures:

1. They are pointer-based: each node contains data and pointers to child nodes. The pointers inflate the size of the tree structure, making it less likely to fit into cache. Tree traversal is based on "pointer-chasing." It requires dereferencing a series of child pointers.
2. Their self-balancing behavior is not achieved without overhead. Additional data must be stored at each node to track tree structure, the structure must be checked whenever the tree is modified, and rebalancing operations occur whenever the tree becomes unbalanced.
3. In most implementations, some kind of allocation or deallocation, such as the C++ `new`/`delete` operations, are performed whenever adding or removing nodes to tree, adding additional overhead.

The approach to determining visibility described above — checking for obstructions at grid lines — does not require a self-balancing binary tree. This is for three reasons:

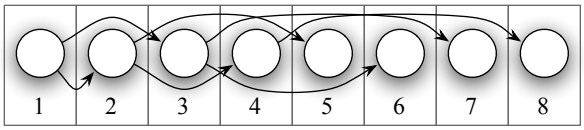


Figure 3.8: An implicit binary tree stores a tree structure in an array. The left child of node n is node $2n$; the right child is node $2n + 1$.

1. The size of the status structure can be determined before the sweep begins. As the sweep progresses, there will be an event one grid interval from the viewpoint, another event two grid intervals away, etc., up to the maximum distance from the viewpoint to an edge of the DEM. The maximum number of intervening points can be determined before the sweep starts: it will be the greatest component distance from the observer to an edge of the DEM. For example, given a 500×500 DEM and an observer at $(200, 100)$, there will be at most 400 intervening points to store in the status structure, when the sweep line is oriented in the positive-y direction.
2. The shape of the status structure never changes. During the sweep, intervening points are never removed, only updated. Put another way, the position of an intervening point in the structure never changes. For example, for an intervening point five units aways from the observer, there will always be four other points between it and the observer.
3. The status structure can use integer keys. Rather than sorting the events based on a real-valued distance from the viewpoint, they can by sorted based on the number of grid intervals between them and the viewpoint. This number is always unique for each event, that is, there is always a single event one interval away, another two away, etc. Unique integer keys map naturally to array offsets.

For these reasons, a dynamic data structure is not required to handle variable amounts of data, nor to maintain ordering as points are added and removed. Instead, an implicit binary tree can be used. As shown in Figure 3.8, in an implicit tree, the location of the nodes determines parent/child relationships, rather than pointers. Because of this, the structure of the tree cannot change. However, parent and child nodes can be found with simple integer operations (right and left shift, re-

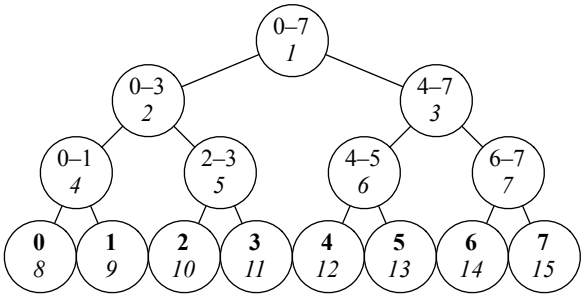


Figure 3.9: The new status structure is based on an implicit binary tree. Italic numbers are the index of the node in the underlying array. Bold numbers are leaf node indices. The internal nodes store the maximum slope over the indicated range of leaf nodes.

spectively), rather the dereferencing a pointer. Also, the implicit tree is optimally compact, because the nodes only store data, not keys or pointers.

The new status structure proposed here is built on top of an implicit binary tree is shown in Figure 3.9. The status structure records the index of the first leaf at initialization time. Notice that all the leaves are in a contiguous array, so any given leaf can be found by offsetting from the first. All intervening height samples are stored in the leaves, based on their distance (in grid intervals) from the viewpoint. Internal nodes store the maximum slope of all nodes in their subtree. Because is based on an implicit binary tree, the internal nodes must always form a complete binary tree. For trees with n leaves, there will be between $n - 1$ and $2n - 3$ internal nodes.

Any time a leaf value is modified, the internal nodes must be updated as well. For example, consider modifying leaf 3. First the leaf value itself is changed at index 11. The slopes at leaves 2 and 3 are compared and the maximum is placed in their parent node at index 5. In the same fashion, that node is compared with its sibling, and its parent is modified; the process repeats in a bottom-up fashion until reaching the tree root at index 1. Modifying the tree is an $O(\lg n)$ operation.

The internal nodes allow this data structure to act as an online variant of the *all-prefix-sums*, or *scan* operation. The scan operation takes a binary associative operator $\oplus$ and an ordered list $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, and returns the ordered list $\langle I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}) \rangle$, where I is the identity element for $\oplus$. The scan operation is useful for determining visibility [97] where maximum serves as the $\oplus$ operation, and the input list is the array of leaf values. The prefix of a

```
 1: procedure PREFIX(tree, n)
 2:     t ← I
 3:     i ← 2
 4:     m ← 2^k
 5:     while n ∧ (m − 1) do
 6:         m ← m ÷ 2
 7:         if m ∧ n then
 8:             t ← MAX(t, tree[i])
 9:             i ← i + 1
10:         end if
11:         i ← i × 2
12:     end while
13:     return t
14: end procedure
```

Figure 3.10: Finding the maximum slope before n in the status structure

leaf value is therefore the maximum intervening slope.

For example, to determine the visibility of a target point 6 grid intervals away from the viewpoint, the maximum slope of all intervening samples (0–5) is required. Figure 3.9 shows that node 2 holds the maximum slope of leaves 0 through 3, while node 6 holds the maximum of slope of leaves 4 and 5, so the maximum of values 0 through 5 can be found by taking the maximum of node 2 and node 6. As with van Krevald's algorithm, this is considering the left subtree each time the tree traversal chooses the right child. However, since there are no explicit child links, this is based on index arithmetic: each "1" in the binary representation of a node index corresponds to choosing the right child.

Figure 3.10 shows how to find the prefix of an arbitrary leaf n. If the tree is depth k, n has a k-bit binary representation. Let $\wedge$ be the bitwise-and operation. The main loop examines one bit of n at a time, corresponding to a single level of the tree. At the beginning of each iteration, index i is at the left child of a node. The bitmask m is used to determine if the bit for this level is set, which is equivalent to choosing the right child. If so, the temporary value t is updated with the maximum of its current value and the left child value, and the index i is moved to the right child. At the end of the loop, i is moved to next left child. The loop condition at line 5 checks if there are any further bits set.

In line 2 a temporary value t is initialized with the identity element I. However, there is no identity element for maximum, i.e. there is no smallest slope. Instead, a near-vertical negative slope — for example the ratio $-8000/1$ can be used. In practice, this will be lower than any measured slope. This I value is also used to initialize all internal nodes of the status structure. Note that on line 3, the index of the first left child is initialized. The algorithm never checks the value at the root, because the maximum slope of all values in the tree is never required.

In contrast with the disadvantages of self-balancing binary trees listed above, this status structure is far more compact, has far less overhead during tree traversal, and has no overhead due to rebalancing or resource allocation. Updates and reads from the status structure are inside the inner loop of the rotational sweep algorithm described below. The advantages of the new status structure do not change the asymptotic runtime, but will reduce the measured execution time substantially. The reduced size of the status structure also means that it will fit in the cache of modern CPUs for any realistic viewshed problem.

3.3.7 Sweep Algorithm

After the sorted event list is created, viewshed analysis can proceed to the planar sweep. First, consider sweeping only the points in the octant from zero degrees up to 45 degrees from the viewpoint. In this octant, interpolation will always occur on y grid lines. Where the input DEM is w wide, begin by allocating a status structure large enough to hold $w - V_x$ events, the number of gridlines from the viewpoint to the right edge of the DEM.

Figure 3.11 describes the sweep algorithm. The event list E is a sequence of n events, $\langle E_0, E_1, \ldots, E_{n-1} \rangle$. Each event is an (x, y) coordinate. G is the input DEM; the notation $G(E_i)$ is the height recorded at the grid coordinates specified by E_i. R is the output raster of maximum obscured heights. S is the status structure. On line 6, the status structure is used to determine the maximum intervening slope. On line 7, the MAX-HEIGHT function uses Equation 3.1 to determine the maximum obstructed height at the location of E_i from the viewpoint V.

In most cases, when an event is processed corresponding to a height sample at (x, y), interpola-

```
1:  procedure SWEEP(V, E, G, R)
2:      Initialize status structure S
3:      j ← 0
4:      for i ← 0 to n − 2 do
5:          (x, y) ← E_i
6:          slope ← PREFIX(S, x − V_x)
7:          R(E_i) ← MAX-HEIGHT(V_z, slope)
8:          if COLLINEAR(V, E_i, E_{i+1}) then
9:              S[x − V_x] ← (G(E_i) − V_z)/(x − V_x)
10:         else
11:             while j ≤ i do
12:                 (x, y) ← E_j
13:                 h ← MIN(G(E_j), G(x, y + 1))
14:                 S[x − V_x] ← (h − V_z)/(x − V_x)
15:                 j ← j + 1
16:             end while
17:         end if
18:     end for
19:     R(E_i) ← MAXHEIGHT(S, G, E_{n−1})
20: end procedure
```

Figure 3.11: Using the event list and status structure to implement the planar sweep

tion (using the minimum) is performed between that sample and at the sample at $(x, y+1)$. Using the interpolated height, the slope is calculated and inserted in the status structure at leaf $x - V_x$. However, when events are collinear with the viewpoint, for example P in 3.3(b), the sample value is used without interpolation. Line 8 checks for collinear points. The index j is used to track the next point requiring interpolation. When a series of collinear events are processed, j will lag behind; when the next non-collinear event is encountered, the while loop at line 11 will advance j to i while inserting the interpolated values.

The direction of interpolation depends on the location of the event relative to the viewshed. For example, events in the the octant between 0 and 45 degrees from the viewpoint will always interpolate in the positive-y direction as shown in Figure 3.11 and described in the previous paragraph. However, events in the next octant will interpolate in the negative-x direction. To avoid determining interpolation direction inside the sweep loop, the event list is partitioned before starting the sweep. Using a binary search, and the same functor used to sort the events, the events are

divided into eight octants. Because the events at octant boundaries are collinear, all values from a previous octant will be replaced at the boundary of a new octant, and the processing of each octant is completely independent.

The pvshed implementation uses C++ templates to create a family of four nearly-identical sweep functions, one for each interpolation direction. The correct sweep function is called for each octant. This avoids checking for interpolation direction, and several other checks. For example, in each of the sweep functions, the events will always be the same direction from the viewpoint, so each function can use $x - V_x$ or $V_x - x$ as appropriate, rather than computing the absolute value. Also, boundary-checking code, which is omitted from the description in Figure 3.11, only needs to check one direction.

3.3.8 Parallelization

The pvshed implementation takes advantage of multi-core processors by using the Intel Threading Building Blocks library described in Chapter 2. Task-level parallelism is identified in several areas, and the TBB runtime handles dividing those tasks among available CPU cores.

The event list sort is a substantial part of pvshed's execution time. Table 3.2 shows the sort comprises 47% of the total execution time in a serial implementation. Parallel sort algorithms are a well-explored topic, and no custom parallel sorting logic is required in pvshed, because TBB provides a `tbb::parallel_sort` function with the same interface as `std::sort` from the C++ STL. The pleasant consequence is that no additional code is required to parallelize the event list sorting.

The sweep itself is the majority of the remaining execution time. The sweep is already partitioned into independent octants, as described above. TBB provides a function, `tbb::parallel_invoke`, which launches a number of functions in parallel. The launched functions are required to have no arguments, so we wrap the sweep function and its arguments in another functor class. After creating a functor instance for each of the octant sweeps, all of the functors are launched in parallel.

Table 3.1: Measured performance.

DEM size ($N \times N$)	pvshed (seconds)	GRASS (seconds)	MATLAB (seconds)	ArcInfo[a] (seconds)
500	0.015	5	10.613	1
1000	0.056	115	30.616	3
2000	0.236	1668	203.294	8
4000	1.041	-	1446.834	26
8000	4.531	-	-	-
16000	20.992	-	-	-
19000	34.102	-	-	-

[a] Measured on a different machine; see text.

3.4 Experimental Results

The implementation of the viewshed analysis described above is a program called pvshed. It is written in C++, and the results in this section are based on using the GNU g++ 4.4.3 compiler and optimization flag -O3. The performance was measured on a computer with an Intel Core i7 930 2.8 Ghz CPU and 6 GB of RAM. Timing was performed with the `gettimeofday` function. Reported execution times include only the viewshed processing time; they do not include file I/O.

3.4.1 Performance

Table 3.1 shows the measured performance of pvshed under the "viewshed" column. The performance of several widely-available viewshed implementations are also shown:

- GRASS [98] is an open-source GIS which provides a viewshed analysis function named `r.los`. Execution times are as reported by the GRASS command interpreter.
- MATLAB has a built-in function `viewshed`; reported execution times were measured using the `tic` and `toc` functions.
- ArcInfo is a widely-used desktop GIS which provides viewshed analysis. The results for this application were measured on a different computer, with an Intel Core 2 Q6700 2.66 GHz CPU and 8 GB of RAM. Execution times are as reported by the ArcInfo software.

Table 3.2: Parallel scaling of the viewshed analysis of a 16k×16k DEM using 1–8 threads.

		serial version	parallel threads			
			1	2	4	8
Event Sort	time (s)	37.5	38.2	20.2	10.9	8.2
	speedup	-	0.98	1.86	3.44	4.57
	efficiency	-	98%	93%	86%	-
Planar Sweep	time (s)	45.6	45.6	23.0	11.8	10.3
	speedup	-	1.00	1.98	3.86	4.43
	efficiency	-	100%	99%	96%	-
Total	time (s)	85.6	86.3	45.7	25.3	21.1
	speedup	-	0.99	1.87	3.83	4.06
	efficiency	-	99%	94%	85%	-

Available memory limits the size of the DEM that can be analyzed with pvshed. Both the input heightmap and the output viewshed use $4n^2$ bytes: one four-byte integer for each input sample and output value. The event list also has n^2 events, each stored as a pair of integers, and uses $8n^2$ bytes, bringing the total memory usage to $16n^2$ bytes. The status structures used by the sweeps require only $O(n)$ memory and have little impact on overall memory usage. When $n = 19000$, the memory usage is 5.8 GB; beyond this size, pvshed uses more memory than is physically available, and performance drops precipitously. On another system with more memory, pvshed completed viewshed analysis of a 40000×40000 DEM in 262 seconds.

3.4.2 Parallel Scaling

Table 3.2 shows how pvshed execution time scales with the number of threads used. The third column, labeled "serial," shows the execution time of the sort and sweep phases of a serial implementation of `viewshed`, as well as the total execution time. This implementation is used as a baseline for comparison when examining parallel scaling.

The fourth through seventh columns show the execution times of the parallel version of `viewshed`, using between one and eight threads. The parallel efficiencies for each phase and the total runtime are also shown. The single-threaded operation of the parallel code demonstrates the relatively small overhead incurred by parallelizing pvshed. At four threads, `viewshed` is utilizing all four

Table 3.3: Time to calculate and apply offsets for curvature correction.

DEM size ($N \times N$)	calculation (seconds)	application (seconds)	total (seconds)
1000	0.005	0.001	0.007
2000	0.019	0.005	0.029
4000	0.074	0.022	0.118
8000	0.300	0.086	0.472
16000	1.241	0.341	1.923

cores of the i7 930 CPU used for testing, and shows an overall 85% efficiency.

By default, TBB uses eight threads on the test system, because each of the cores in the i7 processor uses two-way simultaneous multithreading (SMT). This allows each core to execute two threads at once, and to the operating system, the i7 processor appears to have eight cores. However, two threads in a core share execution resources, and will not execute as quickly as two threads on separate cores. Table 3.2 shows that `viewshed` can take advantage of SMT by using eight threads, gaining a 17% performance improvement versus four threads.

The viewpoint was located at the center of the input DEM for the tests above, so the octants were all equal in size. If the viewpoint is not at the center of the calculated viewshed area, the speedup seen during the sweep phase will be limited by load imbalance. In the worst case scenario, where the viewpoint is at the corner of a thin, rectangular DEM, most events will fall into a single octant, and the sweep phase will see little parallel speedup.

3.4.3 Offset Calculation

For applications of viewshed analysis where the curvature of the Earth is relevant, the approach described in Subsection 3.3.3 provides additional accuracy at the cost of additional execution time. Table 3.3 shows the additional time required for multiple DEM sizes.

In pvshed, vertical offsets for each sample are calculated and stored. These offsets are the $nd^2/2R$ term from Equation 3.3. The offsets are subtracted from the input DEM heights, after which viewshed calculation proceeds as usual. Afterwards, the offsets are added back to the resulting heightmap. The time to calculate the offsets is shown in the second column of Table 3.3,

and the time to add or subtract them is shown in the third column. The "total" time in the final column is the total additional time required for curvature correction, based on calculating the offsets once and applying them twice.

Because the offset calculation in Equation 3.3 is applied individually to each height sample, it is simple to parallelize. The pvshed code uses OpenMP and eight threads to achieve the results in Table 3.3.

3.5 Conclusion

Van Krevald described the use of a planar sweep to improve the asymptotic runtime of viewshed analysis. This chapter describs a refinement of his approach. Determining visibility as described in Subsection 3.3.1 permits the use of a new static data structure, described in Subsection 3.3.6, to serve as the sweep status structure. That data structure supports the required insert and prefix operations with little overhead, and is very compact. Additionally, a number of areas were identified where computationally expensive operations can be replaced with faster alternatives. This approach was implemented as pvshed, a viewshed analysis tool that is much faster that currently-available software, but does not use approximate methods.

Depending on the performance of secondary storage, the time required to read the input DEM from disk, or to write the results, can exceed the viewshed analysis time. This work does not address optimizing secondary storage access, but the speed of this viewshed algorithm does make it particularly well-suited to applications where multiple viewsheds are calculated for a given input. This includes examples such as determining the region visible from along a path [77], or finding the optimal arrangement of observers to cover a given terrain [80]. Both of these applications require calculation of viewshed from multiple viewpoints, and depended on heuristics to reduce the number of viewshed calculations.

Additional Parallelism: The pvshed tool exploits one type of parallel architecture: SMP systems. As described in Subsection 3.4.2, it makes efficient use of four CPU cores on a single

shared-memory system. Combined with algorithmic improvements, pvshed demonstrates ample performance for most current applications. However, there are several possible areas of future work that would allow pvshed to take advantage of additional parallelism:

1. The least efficient use of parallel resources in pvshed is the event sort. In the current implementation, the event list is generated by a single thread before being sorted in parallel. The even generation requires little time, so parallelizing it would have little direct benefit. However, the events could be generated so that they are already partially ordered. For example, a one thread could generate all the events in the first octant, another could generate the events in the second octant, etc. All the events generated by the first thread would precede all the events generated by the second thread, so the threads could generate and sort the events independently.
2. The current sorting implementation can take advantage of large numbers of CPUs, but the sweep phase can only take advantage of up to eight, because the sweep is divided into octants that performed in parallel. Additionally, the octants may be load-imbalanced if the viewpoint is near an edge of the input DEM. One solution would be to further subdivide the octants when it would be beneficial. This will add some complexity; the octants are a natural division, because they share no work. With further subdivisions, multiple partitions of the sweep will depend on the same events. This can be combined with the future work described above: with multiple event lists generated in parallel, the same event can appear in more than one list. Redundant events will add some overhead to the parallel sweep.
3. Taking advantage of distributed memory computers, e.g. clusters, would be more difficult. The parallel combined event generation and sorting described above removes the event list as a shared data structure, and would be useful step. Presumably, this use case would be motivated by a need to process very large DEMs, and each process could load input data for its events into a local sparse matrix format. However, the partitions and event list change if the viewpoint changes, which means every new viewshed calculation would be accompanied by I/O operations, likely negating any parallel speedup.

Chapter 4

Accelerating SIFT on Hybrid Clusters

This chapter describes an approach to parallelizing SIFT and other feature transformation algorithms that utilize scale-space approaches. By partitioning the workload in a novel fashion, this approach can take advantage of all forms of parallelism listed in Chapter 2: the shared-memory parallelism of threaded programming, the distributed-memory approach of cluster programming, and GPU-based acceleration.

Also described is an implementation of this approach called SOHC, or SIFT on hybrid clusters, which can take advantage of hybrid clusters to accelerate the transformation of arbitrarily large images into sets of features. SOHC is both portable and scalable: it can run on systems ranging from a desktop without any GPU hardware, to a cluster of multi-GPU nodes, with the only difference being time to complete the extraction. It is the only implementation of SIFT capable of operating directly on gigapixel-sized images.

4.1 What is SIFT?

The Scale-Invariant Feature Transform [99, 100, 101], or SIFT, is a popular feature transformation algorithm. Essentially, the goal of SIFT is to identify important points in an image, then describe those points in a way such that a computer can identify similar points in other images. Alternatives such as GLOH [102] and SURF [103] exist, but SIFT remains in wide use.

While SIFT was originally created to facilitate object recognition, by matching features from an input image to features from an image of a known object. It is also widely used for *image registration*. This process establishes a correspondence between two or more images, allowing them to be aligned, overlapped, and transformed into a common coordinate system. The images may be photographs of the same scene taken at different times, with different cameras, and from different viewpoints. Image registration is discussed in more detail in Chapter 5.

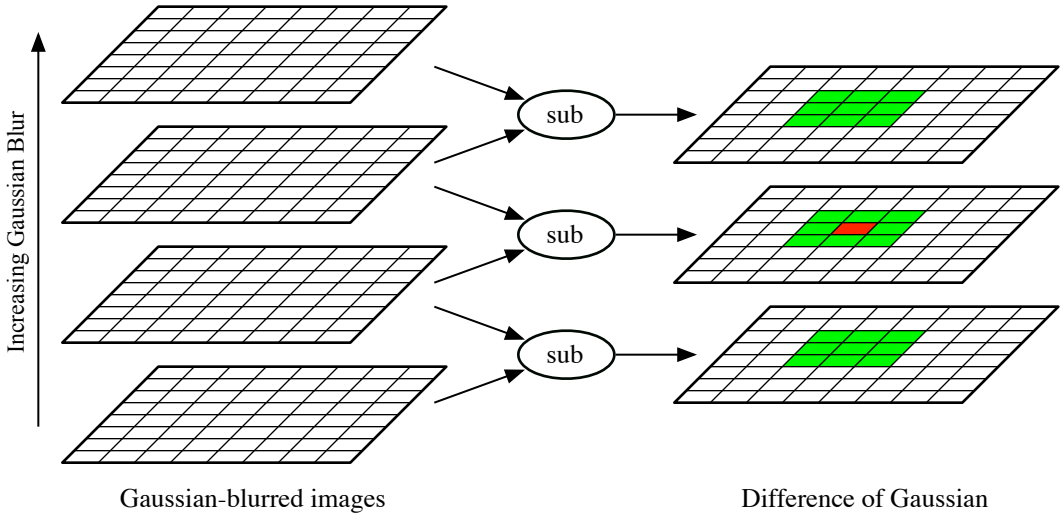


Figure 4.1: The Difference of Gaussian images are produced by subtracting a Gaussian-blurred image from the previous, less-blurred image. The red pixel is a potential feature location if its value is either greater than, or less than, the values of all the green pixels.

SIFT can be broken into two independent phases. During the first phase, feature detection, the location and scale of the features is identified. During the second phase, feature description, the features are characterized so that they can be matched with features from other images. These phases are described in more detail below.

4.1.1 Feature Detection

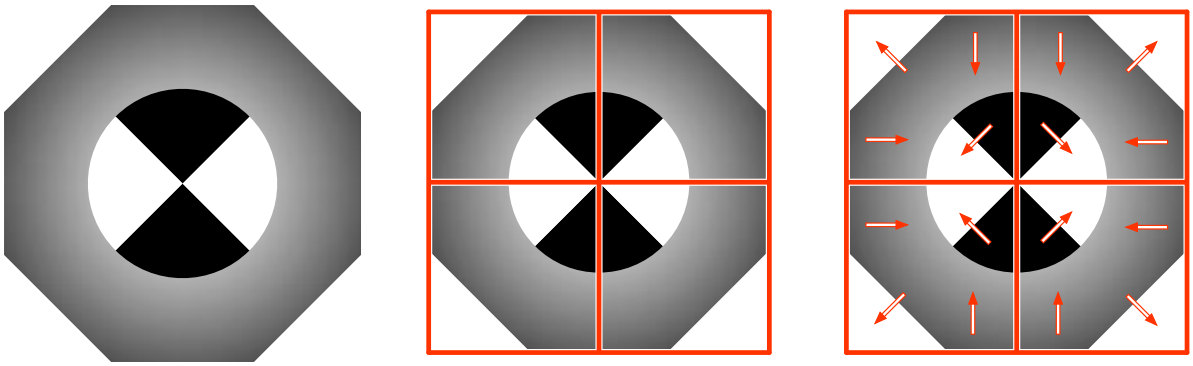
Scale Space: SIFT's feature detection mechanism is based on finding extrema in the *Difference of Gaussians*, or DoG. This process is depicted in Figure 4.1. The input image, at the bottom of the stack of images on the left of the figure, is repeatedly blurred by convolution with Gaussian kernels. The standard deviation of the Gaussian distribution used, σ, controls the level of blur: a higher σ is more blurred. Gaussian convolution acts as a low-pass filter, repeated convolution effectively increases σ, lowering the cutoff frequency and discarding more information from the original image. The series of Gaussian images is called a *scale-space representation* of the original image.

The DoG images are created by subtracting a Gaussian-blurred image from the previous, less-blurred image. Because the Gaussian images are effectively low-pass filtered versions of the orig-

inal image, the DoG images are band-pass filtered versions. Intuitively, the progressive blurring is removing larger features from the image with each step, and the DoG images show when the blurring has completely removed a feature. Extrema in the DoG images are those pixels with a greater or lesser intensity than all surrounding pixels in the same DoG image and in the two adjacent DoG images, as shown in Figure 4.1. These extrema show the location of a "removed" feature, and all such features corresponding to the extrema in a single DoG image will be of roughly the same size, corresponding to the band-pass frequency of the DoG image. This is used to identify the size, or *scale* of the feature.

Pyramid Representation: One way of approximating scale-space is by repeatedly downsampling the input image. Downsampling by a factor of two, which creates an image four times smaller than the original, is an approximation of doubling σ, or halving of the cutoff frequency. Each doubling of σ is called an *octave*. SIFT uses a fixed number of samples per octave, typically three. This pyramid approach greatly reduces the computational overhead of scale-space representation for two reasons. First, images higher in scale space are much smaller, and faster to process. Second, Gaussian kernels with higher σ must be wider, and therefore more expensive to compute; pyramid representation limits the magnitude of σ.

Filtering: The goal of feature detection is not only to identify these features, but to do so in a repeatable way. That is, the feature detection should reliably identify two features that look the same in two different images. To help accomplish this, features that are less likely to be repeated are filtered out after the DoG detection. Features found in low-contrast regions are removed because they tend to look similar to any other feature found in a low-contrast area. Features found along edges are removed because they are "unstable," that is, the same feature may be detected elsewhere along the same edge in different images.



(a) An example of a portion of an image identified as a feature.

(b) The local neighborhood is sized according the scale of the feature and subdivided into regions.

(c) Within each region, the dominant gradient is sampled.

Figure 4.2: Creating a SIFT descriptor

4.1.2 Feature Description

For the features that remain, SIFT generates a feature descriptor. The list of points and their descriptors is the output of the SIFT algorithm. The SIFT feature descriptors are invariant to rotation, scale, and translation. The are also partially invariant to affine changes such as viewpoint change, and have been demonstrated to be distinct and robust enough to match features for a wide range of image transformations [102, 104].

The descriptors characterize the local image neighborhood around a feature. The size and orientation of the neighborhood are determined by the source image. The size is based on where in the sequence of DoG images the feature was found. Images later in the sequence are more blurred, so features found in those images will be larger. A larger feature is described using a larger neighborhood. The goal is that each feature will be the same size relative to its neighborhood, so a feature that appears in different images at varying scales will have the same descriptor; this is the basis of SIFT's scale-invariance. The rotation of the neighborhood is determined by the dominant image gradient in the neighborhood. The angles of the orientation histogram described below are measured relative to the angle of dominant gradient; SIFT uses this mechanism to achieve rotational invariance.

This local area is divided into subregions; Figure 4.2(b) shows the neighborhood divided into

four regions; SIFT uses 16 regions by default. In each subregion, the image gradient is sampled in a grid pattern. Figure 4.2(c) shows a 2×2 sample grid for each region; SIFT uses a 4×4 grid by default. The gradient magnitudes are weighted by a Gaussian window centered on the feature, with the weights falling off with greater distance from the center. Finally, these samples are accumulated into a single 8-bin histogram for each subregion. Each bin covers a different forty-five degree sector of orientation; the value in each bin is the sum of the weighted gradient magnitudes whose orientation falls within that bin.

With an orientation histogram for each of the 16 subregions of the local neighborhood, and eight bins per histogram, SIFT uses a $16 \times 8 = 128$ element vector for each feature descriptor. The number of gradient samples and bins per histogram are variable, but the values above are the most common. The full descriptor for each feature consists of the coordinates, scale, orientation angle, and the 128-element vector describing the feature's neighborhood.

4.1.3 Implementations

SIFT's widespread use means there are many existing SIFT implementations. David Lowe's original implementation is only available in binary form [105]. SIFT++ [106] and the newer SIFT code in VLFeat [107] are commonly-used open-source solutions. Several authors have investigated accelerating SIFT using multi-core architectures [108, 109]. Others have accelerated portions of the SIFT algorithm using GPGPU [110, 111, 112]; these efforts have focused on computer vision applications, which emphasize fast throughput of many small images, rather than the larger images typical in geospatial applications.

4.2 Parallelizing SIFT

To take advantage of hybrid clusters as described in Chapter 2, SOHC addresses several issues:

1. Input images from the target domain are very large, on the order of tens of gigapixels or more. Therefore, the input images must be partitioned. The partitioning scheme must have good locality, so that distributed nodes can work on limited areas of the image without frequently

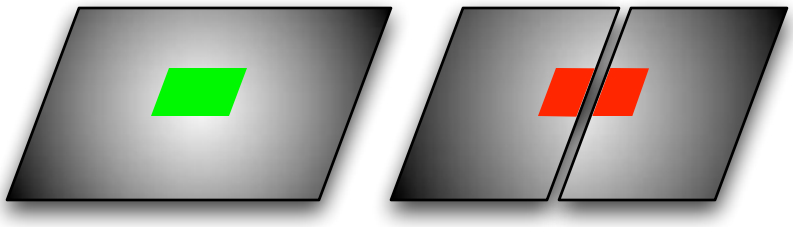


Figure 4.3: Dividing the input image into tiles and using SIFT on each tile. A feature in the original image is lost because its neighborhood (shown whole on the left and divided on the right) is split across tiles.

communication with other nodes or additional file I/O, which will limit performance. Additionally, SOHC scales down to a single node, which will typically not have enough memory to entirely load the input image and its generated scale space and DoG images. Therefore, the partitioning scheme must accommodate *out-of-core* processing — that is, it must be able to stage the work such that it can load smaller portions of the work during execution time.

2. SOHC can use GPUs to accelerate the scale-space generation step of the SIFT algorithm, while later steps are executed on CPU cores. SOHC runs on a wide variety of architectures, with one or more distributed memory nodes, each node having varying numbers of CPU cores and GPU accelerators. SOHC must be specified with sufficient concurrency to run efficiently on a variety of systems.

SOHC uses a novel partitioning scheme, described in Subsection 4.2.1, to address the first issue. It uses a pipelined thread pool scheme, described in Subsection 4.2.4, to address the second issue.

4.2.1 Work Partitioning

Both the out-of-core and distributed memory issues can be solved with a partitioning scheme that divides the SIFT computation into independent tiles. The tiles can be completed serially or in parallel, allowing SOHC to use multiple CPUs and cluster nodes in parallel. Each tile must be able to independently load its work and write its results to allow the required out-of-core operation: to process a tile, a node can load a small portion of the input at a time. This also allows smooth parallel scaling. The mechanism for independent I/O is described in Subsection 4.2.2.

The most straightforward method of input tiling is to simply divide the input image into multiple pieces. SIFT can run on each sub-image, and the location of generated features corrected from the coordinates of the sub-image to the coordinates of the original image. This approach has the drawback [113] of losing features, for the reason illustrated in Figure 4.3. Essentially, features whose local neighborhoods overlap the boundaries between sub-images will be lost or altered. If the image is being aligned with another image of similar size, these lost features may not be significant. However, if the other image is much smaller in scale and located near the tile boundary, this can prevent correct alignment. This can be mitigated by overlapping the tiles, but the overlap leads to a duplication in work. Also, overlapping the tiles can never completely remedy the problem of missing and altered features, as feature neighborhoods grow with the scale of the detected feature, so there will always be some features missed with any amount of overlap.

Octave Partitioning: Instead of partitioning the input image in its original coordinate space, the partitioning can occur in scale space. This is shown in Figure 4.4. Instead of a work partition consisting of all levels of scale space for a subregion of the original image, it consists of a set of fixed-sized images in a single octave of scale space. For example, in octave 0 no downsampling has been performed. A partition in octave 0 will be some size $n \times n$; partition sizing is discussed in further detail below. A partition in octave 1 will also be $n \times n$, so there will be roughly four times as many partitions in octave 0 as in octave 1, but each partition represents a roughly equivalent amount of work.

Partitioning in scale space also allows SOHC to get all original features without excessive overhead. Because each octave has logically increased the scale, rather than physically increasing it, there is a relatively small maximum neighborhood size for any feature detected in an octave. The partitions can be overlapped by this maximum neighborhood size to guarantee that all required information is available in scale space for feature detection and description.

In addition to preserving the neighborhood area itself, maintaining perfect fidelity would also require overlapping all pixels that contribute to the final value within the neighborhood area. Put

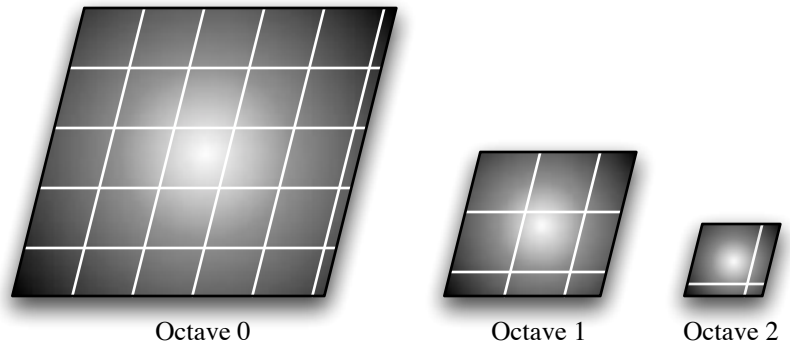


Figure 4.4: Rather than divide the source image into equally-sized tiles, instead each octave of scale space can be divided into equally-size regions.

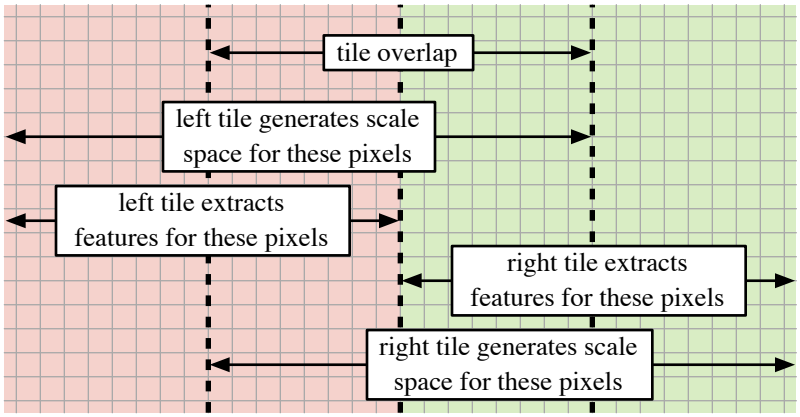


Figure 4.5: Dividing an octave base into multiple tiles. Both adjacent tiles will generate scale space for the overlapped region, but each only generates features for half of the region.

another way, tiles would have to be enlarged by the total radii of all Gaussian convolution kernels used for scale space generation with the octave of scale space. However, the values from the edges of the filter kernels contribute little to the final value. SIFT further reduces the impact of these edge values by applying a Gaussian weighting window to the gradient histogram values during descriptor generation.

Because of these factors, the tiles can be overlapped less than is required for guaranteed fidelity, with little impact to the final result. Empirical testing showed that an overlap of 40 pixels in the tile boundaries led to the alteration of less than 0.1% of the generated features. Approximately 50% of the altered features only differed in one out of the 128 histogram bin values. SOHC uses this value as its default tile overlap parameter.

Figure 4.5 shows how this overlap is implemented. Some overhead is incurred because, as shown, processing of adjacent tiles involves redundantly generating scale space for overlapped regions. However, there is no redundant calculation of feature descriptors; half of the overlapped region will be handled during the processing of each adjacent tile. The overhead of scale space generation is mitigated by two factors:

1. The tile size can be maximized to reduce overlap as much as possible. The tile size is primarily limited by available memory: memory on the GPUs for scale space generation, and system memory which stores scale space, DoG images, and generated features before they are written to disk. While larger tiles reduce the proportion of overlap, they also increase the granularity of parallelism available to SOHC and can make less efficient use of system resources. Tile size selection is discussed further in Subsection 4.3.1
2. SOHC accelerates scale-space generation with GPUs, which can complete this task with high throughput. It is fast enough that scale-space generation is not a bottleneck in SIFT processing, and because of SOHC's pipelined design (described below), additional time spent in this phase of SIFT does not increase the overall runtime of SOHC. This effect is discussed in more detail in Subsection 4.3.2.

4.2.2 Octave File I/O

As mentioned above, SOHC needs to be able to load work tiles independently. Each octave must load a base image, which is convolved with Gaussian filters to create the scale space images within the octave. Octaves beyond the first required downsampled versions of the original image to use as the base image. The downsampling for each tile could be done independently, but this requires redundant calculations. For example, to load an $n \times n$ tile in octave 3, a $8n \times 8n$ region of the original image (which is the base of octave 0) would be loaded and downsampled by a factor of 8. This same region must be downsampled to produce areas of octaves 1 and 2.

To reuse this computation, SOHC adds a pre-processing phase before SIFT feature extraction begins. It generates a number of temporary files to act as octave bases. SOHC divides the rows of

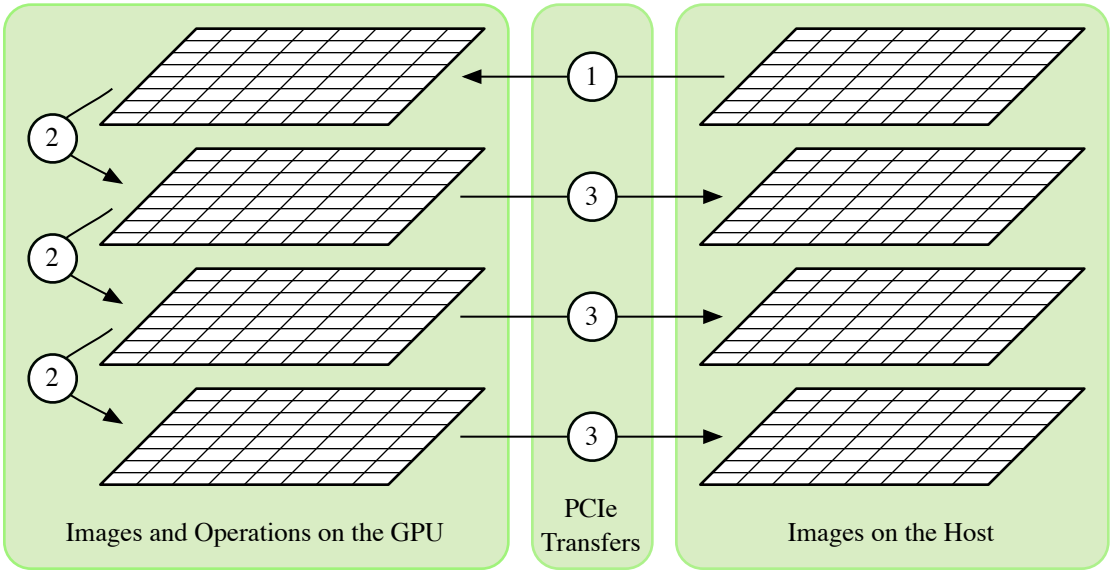


Figure 4.6: Acceleration of scale space generation on the GPU: 1) A tile of the octave base is transferred to the GPU. 2) Filter kernels are used to generate scale space images in GPU memory. 3) Generated images are copied back to the host.

the input image among the MPI processes, in a granularity of 2^o, where o is the highest-numbered octave. For example, if there are three octaves, each process reads a multiple of four rows, because there are octaves 0, 1, and 2, and $2^2 = 4$. Each process writes the base of the first octave in parallel by writing each pixel in floating-point format. Continuing the above example, each process writes a multiple of four rows. For octave 1, each process downsamples by a factor of two and writes the new base file. This continues until the last octave, when each process writes a single row. By reading and writing long, contiguous strips of data, SOHC maximizes file I/O throughput. After the process is complete, each temporary octave base file can be read independently.

4.2.3 GPU Acceleration

After the reading the base file, SOHC creates the scale-space images for the octave using GPU acceleration. GPUs are ideal for this type of operation [114], which has excellent data-parallelism. That work showed that GPUs can perform the Gaussian filtering required for scale-space generation several orders of magnitude faster than a serial CPU implementation. The primary speed bottleneck was communication of input and results between the GPU and the host computer.

PCIe communication still dominates GPU scale space creation in SOHC, but the effect is minimized by generating an entire octave of scale space at once, as shown in Figure 4.6. This reduces the total number of PCIe transfers versus performing the transfers for each convolution separately, as in [114]. SOHC uses the `GaussianBlur` function from the OpenCV library [115] to implement the GPU-based scale space creation in a way that is both portable and performant.

4.2.4 Pipelined Concurrency

After the MPI processes have built the octave base files, all octave tiles are divided evenly among the MPI processes, which can begin SIFT processing. The basic steps of processing a tile are:

1. Load the tile input data from the octave base files. The MPI-2 standard includes "MPI-IO," a file-handling API that was provided by NASA. SOHC uses MPI-IO because it is portable, has built-in functionality for reading subarrays from regular files, and can be optimized by MPI implementations based on the underlying system.
2. Generate scale space using GPU acceleration as described above, or using a CPU-based implementation. In either case, the result of this step is all scale-space images derived from the tile's region of an octave base file.
3. Identify and filter features on the CPU. Identification is based on extrema detection, and is more difficult to implement efficiently on the GPU than scale space creation. A serial CPU, examining each pixel sequentially, can short-circuit the check before examining all 26 neighbors and move to the next pixel. This approach can also be used on the GPU, with each thread checking a pixel; however, the short-circuiting causes thread divergence, as each pixel may be removed from consideration after examining a different number of neighbors. Like extrema detection, the feature filtering is unlikely to see the same speedup as scale-space generation; the scattered nature of detected features prevents regular memory access patterns required for maximum performance. SOHC uses the feature identification and filtering implementation from SIFT++ for this step of processing. The result of this step is a list of feature locations, orientations, and scales.

4. Generate descriptors on the CPU. Like the previous step, the pattern of memory accesses due to the irregularly-distributed nature of detected features makes GPU acceleration difficult, and SOHC again uses SIFT++ code for this step in SIFT processing. The location and scale of the resulting descriptors must be corrected based on the location of the tile, both in image and scale space. For example, a tile with a top left corner of (x, y) in octave 0 will add that offset to the coordinates of detected features. A tile with the (x, y) top left in octave 3 will multiply the scale of detected features by 4 and add $(4x, 4y)$ to feature coordinates, translating from the scale and image space of the tile back to those of the original image. The scale space images are no longer needed after this step and the memory they use can be released. The result of this step is the completed set of SIFT descriptors for the tile.
5. Write results to disk. After the descriptors generated by the previous step are saved to an output file, the memory they occupied can be released, which frees all memory resources allocated while processing the tile. The order of the features is unimportant, but they must be written atomically — that is, feature descriptors should not be interleaved. SOHC uses MPI-IO again to write the result file, and all writes are performed by a single thread to serialize output and prevent interleaving.

In a serial implementation, these steps would be performed sequentially for each tile. Figure 4.7 shows this sequential execution for a single tile in the row labeled "serial processing."

Figure 4.7 also shows SOHC's use of a pipeline to parallelize these steps. In MPI programming, it is typical to launch one MPI process per CPU core in the cluster. The fact that some processes are sharing the same node is abstracted away by the MPI interface, although the underlying implementation may take advantage of shared memory to optimize communication between processes on the same node. With hybrid clusters, this abstraction is less appropriate — processes will compete for GPU resources. Instead, SOHC launches a single MPI process per node, then launches threads corresponding with the physical resources of the node: one "gpu worker" thread per GPU and one "cpu worker" thread per CPU core, as well as one "MPI worker" thread.

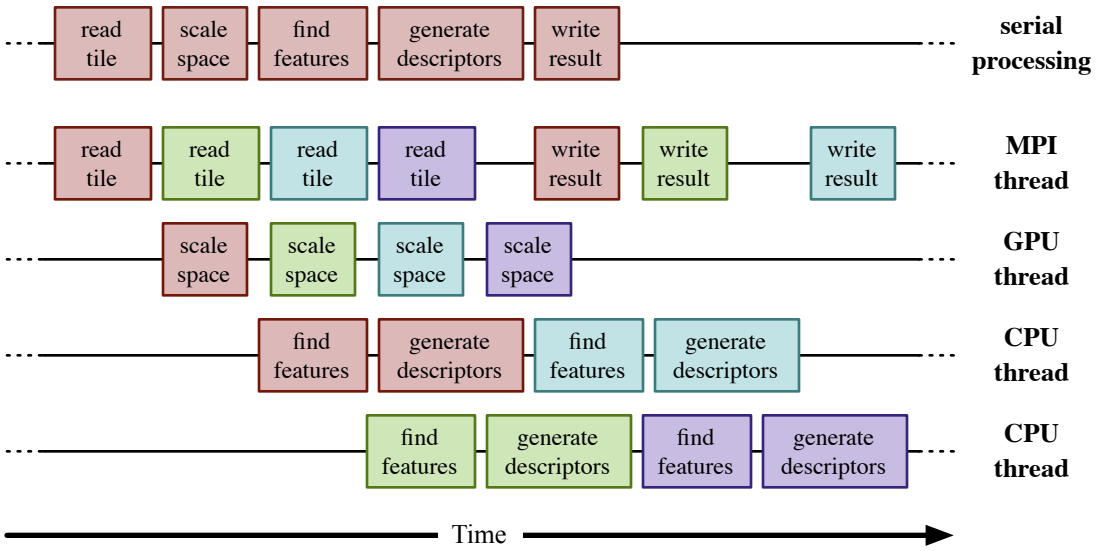


Figure 4.7: Concurrent task execution in SOHC. The top row shows serial execution of the tasks that comprise processing a single tile. The next four rows show four tiles moving through the SOHC pipeline; tasks for a given tile are all shown in the same color.

The single MPI thread compensates for a software limitation: most current MPI implementations are not thread-safe. Only one thread — or only one thread at a time — can be performing MPI operations. In SOHC, all MPI operations are "funneled" into a single thread. This includes loading tile data and writing the features extracted from tiles to disk. These operations are independent and can theoretically be done in parallel. However, serializing them in a single thread is not overly restrictive, because the MPI-IO operations are all sharing the same underlying network and storage subsystems. While these can be logically independent, they will naturally slow to near-serial speed as they contend for network and storage resources.

Once the MPI thread completes step one, loading the tile data, a pointer to the data is passed via a shared-memory queue to the GPU threads. Any available GPU thread can dequeue the pointer and generate scale space images. When complete, the scale space images are passed via another queue to the CPU worker threads, which perform steps three and four. This phase of SIFT computation is inherently unbalanced. Different regions and levels of scale space will have varying numbers features, corresponding to areas of the image with varying levels of detail. Given a static work partitioning, this would lead to load imbalance and inefficient use of the available hardware.

Instead, the pipelined thread pool scheme used by SOHC is inherently load-balancing within a node. Any of the CPU worker threads can dequeue a tile's scale space images produced by any of the GPU worker threads. If a given tile requires more processing than average, another CPU thread will pick up the next tile. After the CPU thread has completed feature extraction on the tile, the result is passed through a final queue back to the MPI thread, which writes the results.

In addition to balancing varying tile workloads, this pipelined approach also makes the SOHC application portable across a variety of systems. In any given system, the storage, GPU, or CPU resources will be a theoretical limiting factor to performance, creating a bottleneck in the pipeline. Which resource causes the bottleneck will vary depending on the relative capabilities of those systems. The pipelined thread pool decouples the performance of each stage as much as possible, allowing them to proceed independently and in parallel, so that the slowest stage will not limit the performance of other pipeline stages beyond effects of its own throughput.

4.3 Experimental Results

The software implementation of the feature extraction described above is a program called SOHC. It is written in C++, and the results in this section are based on using the GNU g++ 4.4 compiler with optimization flag -O3. Two computer systems were used to benchmark SOHC's performance:

- The "fermi" computer has an Intel Core i7 930 2.8 Ghz CPU, 6 GB of RAM, and two Nvidia GTX 480 GPU cards.
- The "star" computer is a distributed-memory cluster where each node has two Intel Xeon E5520 processors, 12 GB of RAM, and two Nvidia GTX295 graphics cards.

Performance measurements were obtained using the unix "time" command and thus include all program overhead and file I/O time.

The input image used for much of the testing is shown in Figure 4.8. At 758 megapixels, it is relatively small — some geospatial images are many gigapixels — but it is large enough to demonstrate the parallelism and out-of-core capability of the SOHC application. It is also large

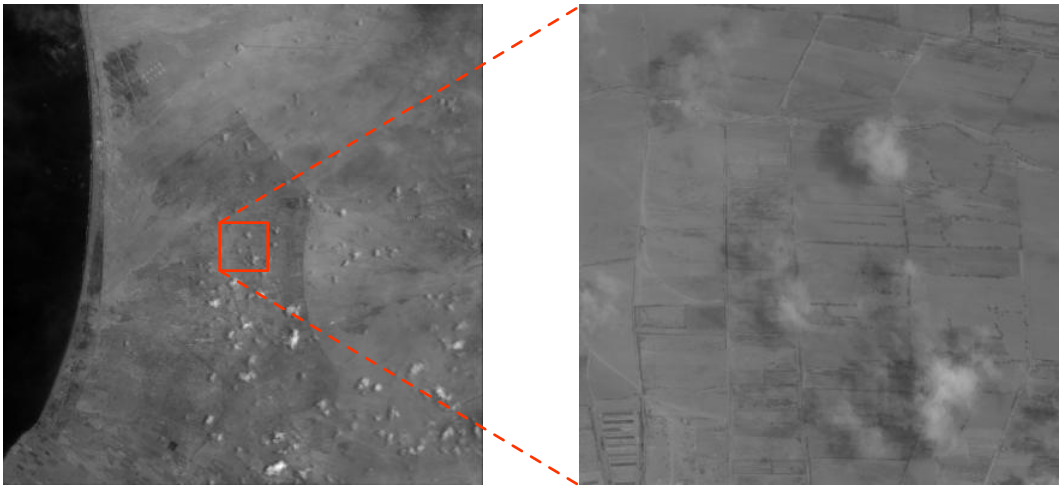


Figure 4.8: The 27552×27504 input image shown on the left was used as input for the results in this section. The image on the right is a 2750×2750 sample from the center of the input image, magnified 10×.

enough that traditional SIFT implementations will either exit with errors because they cannot allocate enough memory, or demonstrate unacceptably long execution times due to excessive virtual memory swapping.

4.3.1 Shared and Distributed Memory Scaling

The SOHC application has a number of parameters that can be selected at runtime to control parallel execution. The most important are:

- The size of the octave space partitions, or tiles. These tiles form the fundamental unit of work in SOHC: the worker threads always operate on one tile at a time.
- The number and type of worker threads. There is always a single mpi worker thread, but the number of scale-space threads and feature-extraction threads can be varied.
- The number of "in-flight" tiles. This is the total number of tiles in the pipeline, at any stage.

Multi-Core CPU-only Scaling: The ability of the SOHC application to utilize multi-core computers can be demonstrated by varying the number of in-flight tiles. The results in Figure 4.9 were obtained on the fermi test system. Because this computer has four cpu cores, SOHC was run with

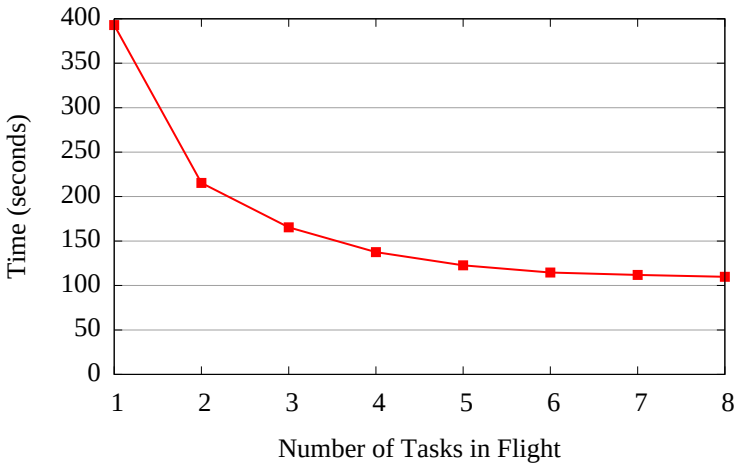


Figure 4.9: Execution time on a 27552×27504 image for different numbers of in-flight tiles. See text for execution parameters.

four scale-space and four feature-extraction worker threads. This number insures that there is always a thread available for all cores, and any extra threads stay blocked and do not add overhead. The execution time shown is the minimum of several test runs, to minimize the effects of system noise.

Having a single tile in-flight is equivalent to serial operation: one tile is loaded and passed to each type of worker thread in turn. That tile's results are written before the next tile is loaded. Operating in this fashion, SOHC completes SIFT on the input image in 393.0 seconds. This time is measured with the unix `time` command and includes all overhead, including program initialization and file I/O. Over 305,000 features are generated.

As additional in-flight tiles are added, they are processed in parallel, and execution time decreases. At five tiles, the execution time reaches 122.7 seconds, a 3.2× speed-up over the serial serial execution time. Additional in-flight tiles add only modest gains. Eight tiles reduces the execution time to 109.8 seconds, a 3.6× speedup. This additional speed is the result of exploiting the "hyper-threading" capability [116] of the i7 processors in fermi, which can extract some additional performance from the additional threads.

Table 4.1: Breakdown of tile processing time for a 27552×27504 image.[a]

generate octave bases (s)	load tile data (s)		scale space generation (s)		feature extraction (s)		write tile results (s)	
	total	per tile	total	per tile	total	per tile	total	per tile
29.6	30.5	0.029	264.9	0.255	170.3	0.164	0.231	0.0002

[a] Tile size is 1000×1000; times are based only on full-size tiles. See text for details.

Execution time by function: Each tile is processed by a stage in the SOHC pipeline. Table 4.1 shows the time spent in each stage. These results were obtained on the fermi computer, with four scale-space and four feature-extraction worker threads, and a maximum of six in-flight tiles. The total and per-tile times listed on the table are based only on full tiles. That is, the "total" time does not include the time spent processing the partial tiles that occur at the right and bottom edges of the octave images. For the given input image, there are 1039 full-size tiles, and the "average" time is the average time to process one of these tiles. These times are wall-clock times measured by the `gettimeofday` function.

The first column, "generate octave space," is the time spent reading the original input images and generating the temporary octave base files. This phase of the program is not part of the processing pipeline and does not benefit from multi-threading. The average total execution time was 116.7 seconds, so the octave space generation represents 25.4% of the overall execution time. By subtracting the octave generation time from the total execution time, the pipelined portion took 363.4 seconds for the serialized run described above and 87.1 seconds for this, parallel run. This speedup of 4.2× is fully exploiting the available parallelism.

Table 4.1 also shows that the most expensive tile processing stage is the scale-space generation, at 0.255 seconds per 1000×1000 tile. Feature extraction takes roughly 64% as long, at 0.164 seconds per tile. The final column is deceptive; writing tile results takes longer than the measured 0.2 ms. Instead, this number is a result of the semantics of MPI's file writing operation. The buffer to be written to disk, which here is the set of features extracted from a tile, is copied into memory in a separate MPI process. The timing numbers are based on the execution of the MPI file write function, which returns immediately after the memory copy, while the results are written to disk

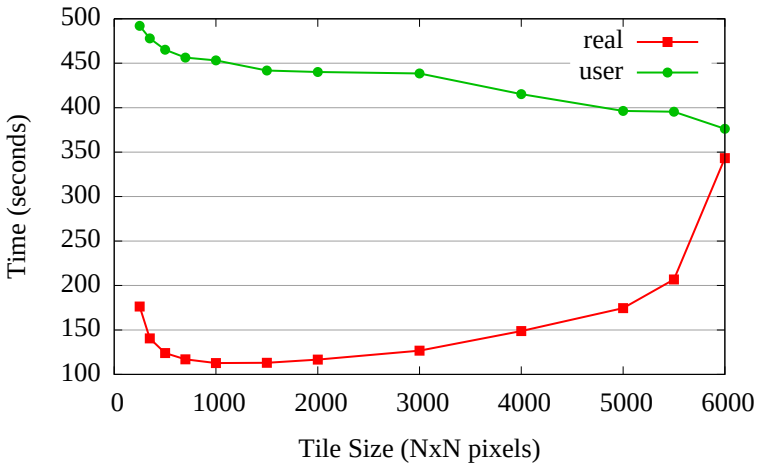


Figure 4.10: How tile size affects the processing time of a 27552×27504 image. The "real" and "user" lines show wall-clock and user CPU time, respectively; see text for details.

asynchronously.

Effects of Partition Size: The tile size of 1000×1000 used in the previous tests is not an arbitrarily-chosen value. Figure 4.10 shows how tile size affects execution time. These results were obtained on the fermi computer, with four scale-space and four feature-extraction worker threads, and a maximum of six in-flight tiles, using the unix `time` command.

Tile size selection is influenced by three factors: overhead, granularity, and available resources. Larger tile size leads to less overhead. In the overlapped region at tile borders, scale space is generated redundantly for all adjacent tiles. The same features may also be detected redundantly, though SOHC ensures that the descriptor for a given feature is only calculated in a single tile. As an example, consider a tile size of 1000 and an overlap of 40 pixels, which are the SOHC default values. An area 1000×40 pixels at the bottom of the tile is redundant with the tile below, an area 40×1000 is redundant with the tile to the right, and a 40×40 area is redundant with the tile to the bottom right. In general, with t-sized tiles and r-sized overlap, the ratio of redundant pixels to all pixels processed is:

$$\frac{t \times r + r \times t + r \times r}{t \times t}.$$

With a fixed r, the numerator increases linearly with t, while the denominator increases quadrati-

cally, so increasing tile size reduces the proportion of redundant work — that is, overhead.

The "user" line in Figure 4.10 demonstrates this effect. This is total user CPU time reported by the unix `time` command. This is the time spent by all threads in user code (as opposed to in system calls), and indicates the "total work" done. As tile size is increased, the total work done decreases. For smaller tile sizes the effect is dramatic, but as the ratio of redundant to all pixels asymptotically approaches zero, the reduction in real work is more gradual.

The benefits of increased tile size are offset by decreased granularity. Larger tiles suffer from poorer load-balancing, increase starvation among threads by dividing the work into fewer tiles, and increases the latency of tile-processing operations. The "real" line in Figure 4.10 shows the total wall-clock time for SOHC execution. For tile sizes smaller than 1000×1000 pixels, the real time decreases with decreasing user time. Above this size, the total execution time increases even as SOHC does less work, because the decreasing granularity limits SOHC's ability to exploit multiple cores. The effect is most dramatic at 6000×6000 pixels, where execution is nearly serialized.

Finally, the available resources limit the maximum tile size. The tile size dictates the amount of memory required to process each tile; this number is multiplied by the number of tiles in flight. For large tile sizes, SOHC uses more than the available physical memory on fermi. This causes frequent virtual memory swaps to disk, known as "thrashing." Running with a tile size of 7000×7000, SOHC execution was manually terminated at 1000 seconds before the program completed.

Cluster Scaling: Figure 4.11 shows how SOHC's performance scales on a cluster computer. These results were obtained by running SOHC against the same input image. The serial results were obtained, as above, by running SOHC with a maximum of one in-flight tile. While this serializes the execution at each node, each node processes tiles in parallel. The threaded results were obtained with a maximum of nine in-flight tiles and eight worker threads of each type on every node, allowing SOHC to fully exploit all CPU resources in the cluster.

SOHC's use of multiple nodes of the star cluster is not as efficient as its ability to use multiple cores. With serial execution on each node, it SOHC goes from 464.3 seconds on a single node to

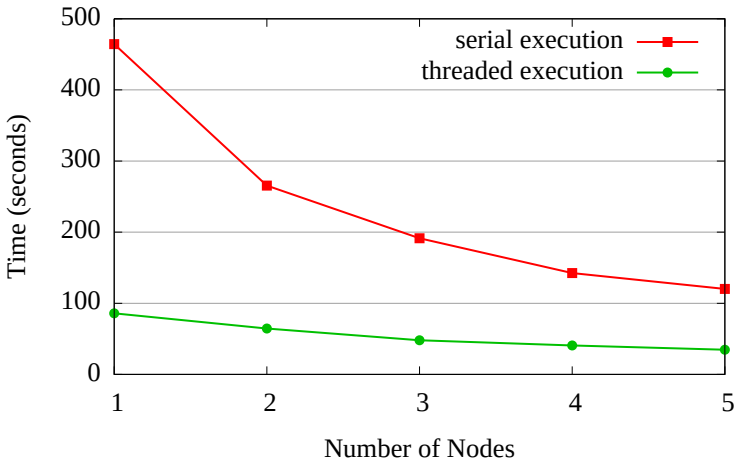


Figure 4.11: Execution time of SOHC using multiple nodes. Serial execution uses a single CPU core on each node. Threaded execution uses all CPU cores on all nodes.

120.1 seconds on five nodes, a 3.9× speedup. When fully utilizing each node, SOHC completes in 85.8 seconds using a single node, and 34.7 seconds on five nodes, which is only a 2.5× speedup.

This reduced parallelism is due to starvation; the worker threads on each node cannot load tile data fast enough to keep up with the throughput of the other worker threads. All nodes on star communicate with a file server via gigabit Ethernet. The interactions of the storage subsystem, network, and MPI I/O libraries on the star cluster are complex, the speed of gigabit Ethernet gives an approximate indication of the maximum throughput for tile processing. Processing the 27552×27504 input image results in 1039 full-size 1000×1000 tiles. Each tile is 4 MB; the total size of the tiles to be loaded is 4.2 GB. That volume of data requires approximately 33 seconds to transfer at 1 Gbps, which is the same time that the parallel execution time in Figure 4.11 is asymptotically approaching.

4.3.2 GPU Acceleration

Table 4.2 shows the results of enabling GPU accelerations. These results were obtained on the fermi computer, with two GPU worker threads and two CPU worker threads, with a maximum of six tiles in-flight. Comparing with the results in Table 4.1, the time to generate scale space for a 1000×1000 tile is reduced from 255 ms to 15 ms, a 17× speedup. As before, this is measured

Table 4.2: Breakdown of tile processing time with GPU-accelerated scale-space generation for a 27552×27504 image.[a]

generate octave bases (s)	load tile data (s)		scale space generation (s)		feature extraction (s)		write tile results (s)	
	total	per tile	total	per tile	total	per tile	total	per tile
27.3	31.0	0.030	15.7	0.015	152.7	0.147	0.303	0.0003

[a] Tile size is 1000×1000; times are based only on full-size tiles. See text for details.

wall-clock time, and includes all overhead, notably the time required to transfer data to and from the GPU across the PCIe bus.

Notably, this speed means that with GPU acceleration, a single scale-space thread can process 66.7 tiles per second. In comparison, the MPI thread can only load 33.3 tiles per second, and a single CPU thread processes 6.8 tiles per second, while four CPU threads process 27.2 tiles per second. Using only a single GPU thread, the results in Table 4.2 remain unchanged, because that GPU thread can process tiles faster than they can be loaded from disk.

A slight decrease from 164 ms to 147 ms per tile can also be observed in the CPU thread processing time. With six tasks in flight and the hyperthreading architecture of the Core i7 CPU in fermi, the CPU-only execution sometimes shares resources between the scale-space and feature-extraction threads, increasing the observed wall-clock time to process a tile. Using GPU-based scale-space creation, the CPU resources are almost entirely dedicated to the CPU worker threads.

The result of these increases in throughput is that the total runtime of SOHC on the 27552×27504 input image is 73.1 seconds with GPU processing enabled, vs. 114.6 seconds without GPU processing, a $1.6\times$ speedup.

4.4 Conclusions

The SOHC application is a demonstration of the partitioning-and-pipeline approach to parallelizing feature extraction described in Section 4.2. Rather than attempting to accelerate most of the individual steps involved in feature extraction, the octave partitioning approach processes regions of scale space in parallel. This has several advantages:

- It requires little change to existing feature extraction code, as each "tile" of scale space can be processed a separate image.
- Scale-space partitioning — versus more straightforward image-space partitioning — doesn't affect the extracted features, and adds little overhead.

The scale-space generation itself is a step where more fine-grained parallelism can be exploited by using GPUs. Results showed the GPU generating scale space 17× faster than a single CPU. Other limitations prevented this from increasing overall speedup more than 1.6×; these are further addressed below.

One of these limitations is secondary storage throughput. This proved particularly limiting on the star cluster computer, where large computational resources quickly saturated the available storage bandwidth. While SOHC is capable of using GPUs in hybrid clusters, in most cases this will not result in a decrease in overall run time.

However, SOHC did demonstrate good use of computational resources in a single multi-core computer, showing effectively linear speedup with the number of CPUs. The pipelined approach provides load balancing across the CPUs in a single node, and there are no artificial constraints: the entire system has the throughput of the slowest component.

Other scale-based feature extraction algorithms can make use of the SOHC framework with little modification to either SOHC or the feature extraction algorithm. In particular, SURF requires less computation for feature detection and description. The additional work required by SIFT for these steps means that GPUs are underutilized in SOHC; SURF may be a better fit for GPU-equipped computers.

4.4.1 Future Work

As noted above in Subsection 4.3.2, the GPUs in fermi computer used for testing were underutilized during SIFT processing. With two GTX 480 GPUs, the scale-space generation stage is capable of processing over 130 1000×1000 tiles per second. At the rates measured in Table 4.2, the system would require 20 CPU cores for the feature extraction stage to match this speed, assum-

ing that feature extraction would scale linearly to that number of cores.

While not all computers will have the high GPU-to-CPU compute ratio of fermi, this suggests that additional steps of SIFT processing can be profitably moved to the GPU, even if those steps do not see the same speedup as scale-space generation. One option is to replace SIFT with SURF, as mentioned above. The OpenCV library includes most steps of the SURF algorithm implemented on GPU, which can be combined with SOHC's partition-and-pipeline scheme. Additional testing, and possibly run-time profiling and tuning, would be needed to ensure both GPUs and CPUs are close to fully-utilized.

Better use of system resources would require a faster storage system, as well. The secondary storage on fermi's is a single SATA drive, and SOHC was limited to loading roughly 33 tiles per second. More expensive storage solutions, such as RAID arrays or solid-state disks, make increasing storage throughput relatively straightforward. Another option is to increase the work done per input file. For example, in addition to SIFT, an application might begin image registration, generating nearest matches for features to another image during the feature extraction process.

In SOHC, all GPU operations are performed synchronously and serially. The scale-space images are created, then copied to the host. CUDA supports overlapping these operations; scale space image n could be copied to the host during the creation of scale space image $n + 1$. This requires the use of "pinned" memory: unlike the memory usually provided by allocation requests, pinned memory describes physical memory resources, not virtual memory. SOHC has facilities for managing a pool of pinned memory resources that can be used in the processing pipeline. This feature is not used, because additional GPU performance is unnecessary and the asynchronous operation adds complexity. If the improvements described above are implemented, asynchronous operation can further improve the GPU processing time.

Chapter 5

Circular Earth Mover's Distance

This chapter describes a tool, gpu-cemd, that runs on hybrid clusters and aids in processing the SIFT features generated by SOHC, described in Chapter 4. This means that, in contrast with SOHC, gpu-cemd operates not on image sensor data, but on generated data. SIFT feature extraction from large images generates an equally large number of features — large satellite images can generate tens of millions of features. Few of these features can be discarded; the images to be referenced against these large images may be at a much smaller scale and overlap only a small number of the features. High-performance analysis of the generated features is therefore a practical requirement of using SIFT on large images.

The gpu-cemd tool uses the CEMD algorithm, described below, to compare SIFT features. Written using CUDA, it is is 75× faster than a single-threaded version of the algorithm running on the CPU. Low-level programming techniques required to fully exploit GPU processors are discussed, and gpu-cemd is shown to take full advantage of GPU processors — that is, no implementation of the same algorithm could run significantly faster.

5.1 Image Registration

Image registration establishes a correspondence between two or more images, allowing them to be aligned, overlapped, and transformed into a common coordinate system. The images may be photographs of the same scene taken at different times, with different cameras, and from different viewpoints.

Image registration techniques can be broadly categorized as direct or feature-based [117]. While direct algorithms may attempt to correlate images based on pixel intensity, feature-based registration requires that images first be transformed into a set of features. As in SIFT, these features are typically points in the image, but can be other entities such as edges or gradients. Once

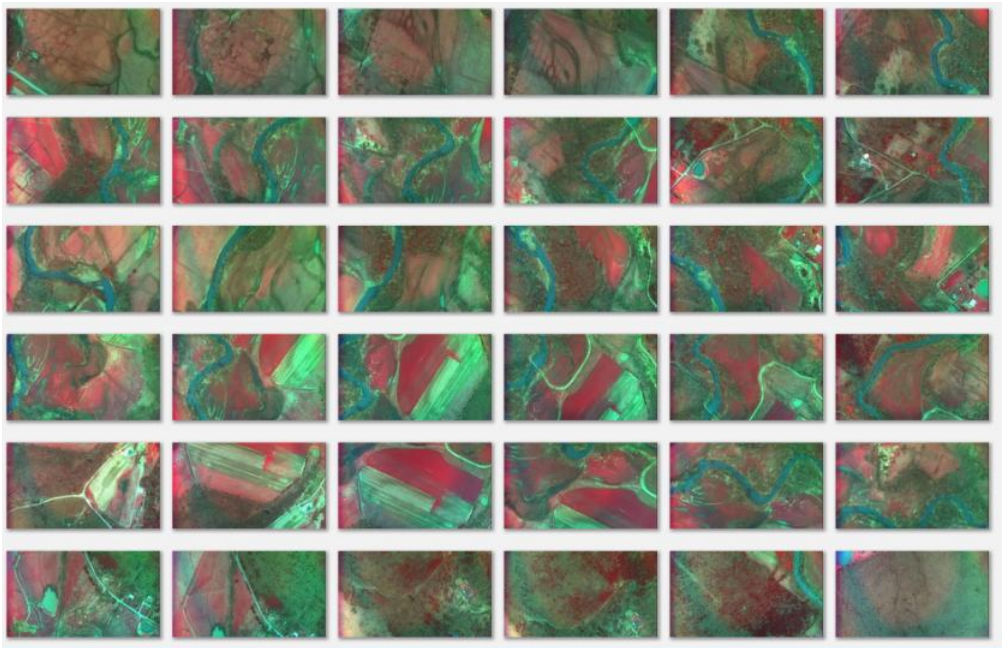


Figure 5.1: Unorganized Block of Images

the features in an image are found, feature-based image registration will search for matching features from other images, rather than directly comparing the images. If matching features can be correlated, then the images can be transformed into a common coordinate system.

Matching features across images requires a method of describing those features and a way to quantify their similarity. The widely-used Scale-Invariant Feature Transform (SIFT), described in Section 4.1, provides a method of finding and describing image features. The Circular Earth Mover's Distance (CEMD), which can be used to measure the similarity of these features, is described in Section 5.2.

Georeferencing: One application of image registration is *georeferencing*, which establishes the location of an photograph can be determining its relationship to photographs with known geographical coordinates. Figures 5.1 and 5.2 show an example; Figure 5.1 is a group of overlapping aerial photos of the same scene, but the relative location of each photo was not recorded. Using image registration, the photos can be oriented to each other, producing the mosaic shown in Figure 5.2.

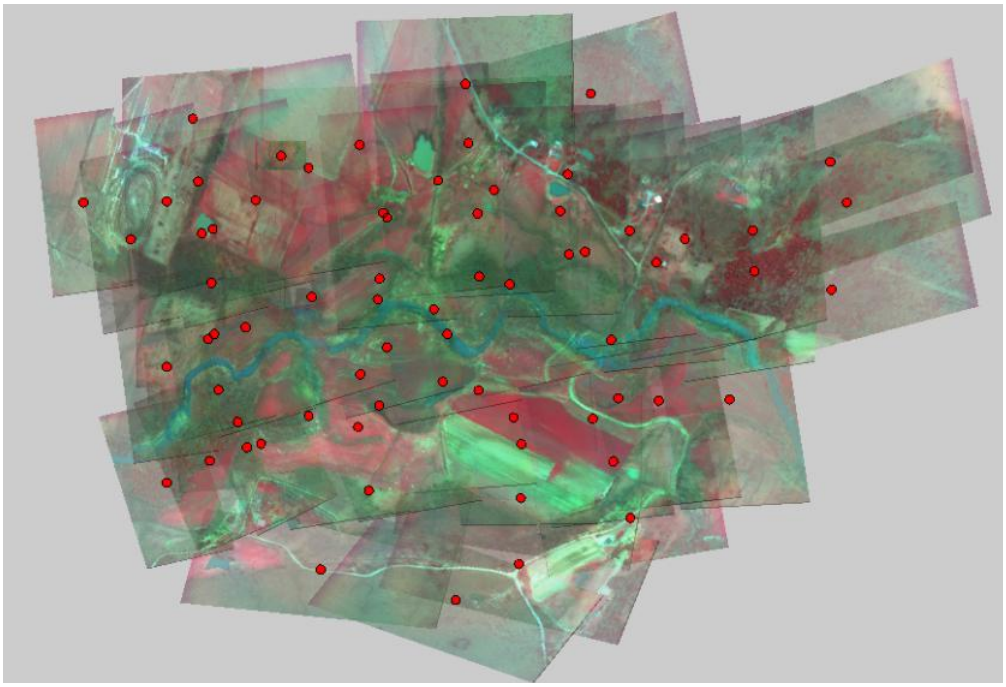


Figure 5.2: Georeferenced Image Mosaic

GIS systems can provide a source of reference images. For example, GeoStor [118] is an online system, maintained by the state of Arkansas, that provides aerial photos of the state with known locations. A new aerial photo, taken somewhere in the state but with unknown exact coordinates, could be matched with the GeoStor images to determine its location.

Image georeferencing can require the calculation of many similarity measurements. For example, a satellite image may have billions of pixels and generate millions of features. If this satellite image is a reference image, then registering a new image requires searching those millions of features for the nearest match to each feature in the new image.

Image Alignment: Aligning the images requires identifying corresponding features in several images, then estimating the *homography*, that is, the parameters that transform coordinates in one image to coordinates in the other. Although SIFT was designed such that images of the same scene should generate similar features, the actual features generated will rarely be identical for many reasons:

1. Differences in scale and orientation mean that while images may overlap, only a subregion

of each image records the same scene. Features from outside these subregions will not be matched.

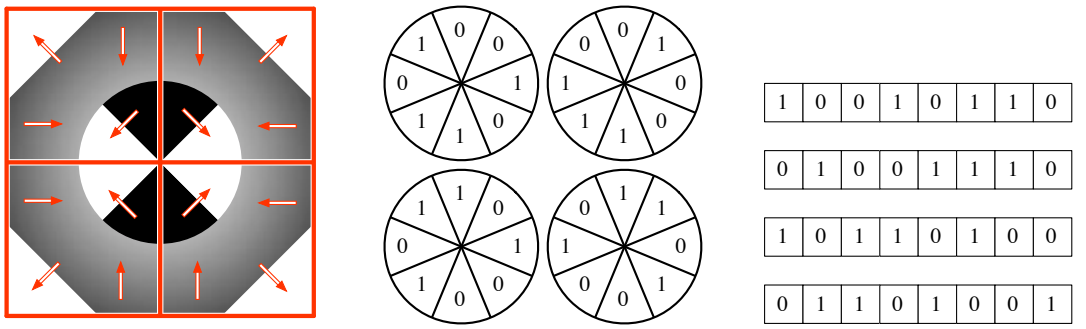
2. Some portions of the scene may be obscured, such as by clouds, smoke, or flood waters. This means that even otherwise-identical images may not share all features.
3. Environmental differences such as lighting and atmospheric conditions change the recorded image, and different sensors — that is, different kinds of cameras — also affect the recorded image.

Given a number of corresponding features, estimating the homography is straightforward, but identifying those corresponding features can be difficult in the face of the problems listed above. Pairs of possibly-corresponding features, sometimes called *putative matches*, are identified by using various measures of similarity, described further in Section 5.2. The "most similar" features will identified as putative matches. Filtering is applied based on the uniqueness of the match: if a feature is roughly similar to several features from the other image, that feature is ignored.

Even with that filtering, not all of the putative matches are actual matches. The random sample consensus, or *RANSAC* [119] algorithm is used to identify the actual matches. RANSAC is an iterative method, applied to feature-based image registration with the following steps:

1. Seven putative matches are chosen at random. These matches are called *hypothetical inliers*; they are assumed to be actual matches and used to generate a homography.
2. The other putative matches are evaluated according to the homography. In other words, features from one image are translated according to parameters of homography. If the translated features are located closely to their putative match in the other image, that match is considered an inlier.
3. If the model is sufficiently good, i.e. if there is a sufficient proportion of inliers, the homography is fitted to the set of all inliers. Otherwise, the model is discarded.

These steps are repeated a fixed number of times, generating a number of homographies with each repetition of the final step. The homography with the least error is identified as the correct image alignment.



(a) A single SIFT feature, with four gradient samples per subregion

(b) Each subregion is divided into eight bins. Each gradient sample is added to a bin based on its direction.

(c) The histogram bin values for all subregions are recorded as a set of arrays.

Figure 5.3: Translating the subregion sampling of a SIFT feature neighborhood into a descriptor.

5.2 Comparing SIFT Features

5.2.1 More on SIFT Feature Descriptors

SIFT was described earlier in Section 4.1. It transforms an input image into a set of features; these features have a location and size in the input image, as well as a descriptor to characterize them. The SIFT feature descriptors are invariant to rotation, scale, and translation. The are also partially invariant to affine changes such as viewpoint change, and have been demonstrated to be distinct and robust enough to match features for a wide range of image transformations [102, 104].

The earlier description of SIFT described how the descriptors characterize the local image neighborhood around a feature. This local area is divided into subregions, and the gradients within each subregion are sampled as shown in Figure 5.3(a).The number of subregions can be varied. The figure shows four subregions, but the SIFT default and the value used in the implementation in this chapter 4×4 subregions. The sampling density is another SIFT parameter; the figure shows 2×2 samples per region, while gpu-cemd assumes 4×4 samples, which is commonly used.

The orientation of the neighborhood is determined by the dominant image gradient within the area of the neighborhood. The positive x-axis of the neighborhood is rotated to match the dominant gradient. Put another way, the dominant gradient is considered to be angle 0. This is the mechanism that SIFT uses to achieve rotation invariance.

The gradient magnitudes are weighted by a Gaussian window centered on the feature, with the weights falling off with greater distance from the center. Finally, these samples are accumulated into a single histogram for each subregion, as shown in Figure 5.3(b). The number of bins is also variable; the implementation uses and the figure shows the SIFT default of eight. Each of the eight bins in the histogram corresponds to a different forty-five degree orientation. The value in each bin is the sum of the weighted gradient magnitudes whose orientation falls within that bin.

The circular histograms for each subregion are unrolled and stored sequentially for each feature, as shown in Figure 5.3(c). Each bin value is stored as a single byte, with a value from 0 to 255. With an orientation histogram for each of the 16 subregions of the local neighborhood, and eight bins per histogram, SIFT uses a $16 \times 8 = 128$ element vector for each feature descriptor.

5.2.2 Measuring Feature Dissimilarity

The dissimilarity between two feature descriptors is commonly measured with distance metrics such as the Euclidean or Manhattan distance [120, 121, 122]. The distance metric is used in a nearest-neighbor search for each feature from an input image among all the features in a reference image. The nearest neighbor is considered a match if it is closer than the second nearest neighbor by a specified ratio, often 40–60% [99]. The dissimilarity measurement used to find the nearest neighbors has a significant effect on the matching process [123].

It is straightforward to use this type of bin-to-bin comparison to measure dissimilarity between SIFT descriptors. However, these comparisons fail to account for the structure of SIFT descriptors, whose elements are not orthogonal dimensions, but orientation histograms. Consider three, eight-bin orientation histograms a, b, and c, where

$$a = (0, 0, 0, 0, 0, 0, 0, 10), \quad b = (0, 0, 0, 0, 0, 0, 5, 5), \quad c = (0, 0, 0, 5, 0, 0, 0, 5).$$

Using the Euclidean distance, Distance(a, b) = Distance(a, c) = $\sqrt{50}$. However, a intuitively seems closer to b than to c, because a and b have more closely-aligned gradient orientations. The

Earth Mover's Distance [124, 125] (EMD) gives a more reasonable result. For the above example, EMD$(a, b) = 0.625$ and EMD$(a, c) = 2.5$, agreeing with the intuition that a is closer to b than it is to c.

The distance EMD(a, b) can be thought of as the minimum work required to change distribution a into distribution b. If both distributions are imagined as piles of dirt, one pile can be transformed into the other pile by pushing some of the dirt. The work done is the product of the amount of dirt moved and how far it had to be pushed.

The EMD defines the cost of moving a unit of dirt from bin i to bin j as $c(i, j)$, where i and j are in a one-dimensional histogram with N bins and

$$c(i, j) = \frac{|i - j|}{N}. \tag{5.1}$$

For example, the cost to move five units from bin 6 to bin 7 in the example above is $5 \times |6 - 7|/8 = 5/8$. Given this definition of the cost function, the EMD can be efficiently computed using

$$\text{EMD}(a, b) = \frac{1}{N} \sum_{i=1}^{N} |A[i] - B[i]|, \tag{5.2}$$

where A and B are the cumulative histograms of a and b.

The EMD is defined between histograms, but SIFT descriptors are a collection of 16 histograms. The EMD can be used to define the difference between descriptors by summing the EMD for each pair of corresponding pair of histograms in the descriptors.

While the EMD is an improvement over the Euclidean distance, it still fails to account for the periodic nature of SIFT descriptors. Each histogram in the descriptor is circular. As shown in Figure 5.3(b), each bin corresponds to a different forty-five degree orientation segment. The last bin is adjacent to the first bin.

The Circular Earth Mover's Distance (CEMD) is a variation on EMD that takes into account the periodic nature of SIFT descriptor histograms [123]. Essentially, it connects the first bin to the last bin, allowing dirt to be pushed directly between them. More formally, it alters the cost function

shown in (5.1) to

$$c(i,j) = \min\left\{\frac{|i-j|}{N}, 1 - \frac{|i-j|}{N}\right\}. \tag{5.3}$$

In other words, it costs $1/N$ to move a single unit of dirt to an adjacent bin. Because the bins are now circular, the maximum distance dirt will be moved is $N/2$, and the maximum result of $c(i,j)$ is $1/2$.

As shown in (5.2), the EMD can be calculated by generating a cumulative histogram X for each input histogram x. The CEMD instead uses N cumulative histograms $X_0, \ldots, X_{N-1}$, defined as

$$X_j[i] = \begin{cases} \sum_{k=j}^{i} x[k] & \text{if } i \geq j, \\ \sum_{k=j}^{N-1} x[k] + \sum_{k=1}^{j} x[k] & \text{if } i < j. \end{cases} \tag{5.4}$$

Essentially, the cumulative histogram X_j begins at index j of the input histogram and wraps around to finish at index $(j-1)$. Given these cumulative histograms, the CEMD can be calculated as

$$\text{CEMD}(a,b) = \min_{j \in \{0,1,\ldots,N-1\}} \left\{\frac{1}{N}\sum_{i=0}^{N-1} |A_j[i] - B_j[i]|\right\}. \tag{5.5}$$

An example for two four-bin histograms is shown in Table 5.1. Intuitively, three units in v_1 should be moved from bin 0 to bin 1 at a cost of $3 \times (1/4)$, and another two units should be moved from bin 0 to bin 3, at a cost of $2 \times (1/4)$, for a total cost of $5/4$. The table shows how four pairs of cumulative histograms are generated, one starting at each index. The differences between each pair of cumulative histograms are summed. The minimum sum for this example is five, and N for this example is four, so $\text{CEMD}(v_1, v_2) = 5/4$.

Research has shown that CEMD is better than many other dissimilarity measures (Euclidean, Manhattan, χ^2) for SIFT feature matching [123, 126]. Because CEMD takes into account the structure of the data, comparisons between SIFT descriptors are more likely to return meaningful results than Euclidean or Manhattan distances.

CEMD is computationally expensive compared to these other measures, requiring significantly

Table 5.1: CEMD Example with 4-bin Histogram

Starting Index		v_1 / v_2			
		v_1	1	9	2	4
		v_2	6	6	2	2
0	v_1 Cumulative Histogram		**1**	10	12	16
	v_2 Cumulative Histogram		**6**	12	14	16
	Absolute Difference		5	2	2	0
1	v_1 Cumulative Histogram		16	**9**	11	15
	v_2 Cumulative Histogram		16	**6**	8	10
	Absolute Difference		0	3	3	5
2	v_1 Cumulative Histogram		7	16	**2**	6
	v_2 Cumulative Histogram		10	16	**2**	4
	Absolute Difference		3	0	0	2
3	v_1 Cumulative Histogram		5	14	16	**4**
	v_2 Cumulative Histogram		8	14	16	**2**
	Absolute Difference		3	0	0	2

more computation to find the difference between two vectors. When viewed from a computational cost perspective, the CEMD is a poor choices for comparing image features. However, the advantage of CEMD is that neighboring bins in a histogram are given consideration in the distance calculation.

5.3 Implementation and Discussion

Registering the new image requires searching those millions of features, and possible features from other images, for matches. The CEMD has been shown to perform well, but is computationally expensive.

CEMD is a good candidate for GPU acceleration. Where there are two sets of features n and m, a single thread can be assigned to calculate each CEMD(n_i, m_j), using C code very similar to a CPU implementation. Where n and m are too large to calculate all results with a single kernel launch, the $|n| \times |m|$ comparisons can be tiled into multiple kernel launches. This can also be used to split the comparison among multiple GPUs and even multiple cluster nodes. The results have

no interdependencies, and thus should see linear speedup with the number of GPUs used.

This section describes the initial implementation and several refinements of a CEMD kernel in CUDA. A summary of the results is shown in Table 5.3. All refinements of the kernel share the same basic form:

1. The input consists of two arrays of descriptors, A and B. These descriptors are in the default SIFT format: they consist of 16 histograms, each having 8 bins. Each bin is stored as an unsigned byte, so each descriptor is 128 bytes. Before the CEMD kernel can execute, A and B must be copied from the host's memory into the device's global memory. For the results shown in Table 5.3, A and B each have 2048 descriptors, and each uses 256 KiB[1] of memory.
2. The output is a two-dimensional array D, where

$$D[I][J] = \text{CEMD}(A_I, B_J)$$

   After the GPU kernel has computed all results, D must be copied from device memory to host memory. For the results reported in Table 5.3, D holds 4×2^{20} results and uses 16 MiB of memory. This copy is the dominant source of overhead in the CEMD kernel.
3. The kernel executes in blocks of 16×16 threads. Each thread has an index (i, j) within the block and an index (I, J) within the grid of all threads. Thread (I, J) calculates the CEMD between A_I and B_J, writes the result to $D[I][J]$ in global memory.

5.3.1 Experimental Setup

All performance results were obtained on a computer with the following specifications:

- Two Intel Xeon E5520 processors
- 12GBytes 1333Mhz DDR3 RAM
- Two NVIDIA GTX295 graphics cards (compute capability 1.3)

[1] 1 KiB $= 2^{10}$ bytes; 1 MiB $= 2^{20}$ bytes

Require: a and b are feature descriptors
Require: a and b are each composed of H histograms

function DESCRIPTORDISTANCE(a, b)
  $distance \leftarrow 0$
  for h **from** 0 **to** H **do**
    $distance \leftarrow distance +$ CEMD(a_h, b_h)
  end for
  return $distance$
end function

Require: x and y are circular histograms
Require: x and y are each composed of B bins

function CEMD(x, y)
  $memd \leftarrow \infty$
  for i **from** 0 **to** B **do**
    $acc_x \leftarrow 0$; $acc_y \leftarrow 0$; $d \leftarrow 0$
    for k **from** i **to** B, **from** 0 **to** i **do**
      $acc_x \leftarrow acc_x + x_k$
      $acc_y \leftarrow acc_y + y_k$
      $d \leftarrow d + |acc_x - acc_y|$
    end for
    $memd \leftarrow \min(memd, d)$
  end for
  return $memd$
end function

Figure 5.4: Descriptor CEMD Algorithm

The software used in development and testing is:

- Rocks 5.2 (Based on CentOS 5.2)
- GNU g++ 4.1.2 (code compiled with -O2 flag)
- CUDA Toolkit 2.3 (code compiled with -O2 and -arch=sm_13)
- NVIDIA Display Driver 190.53

5.3.2 Runtime of CEMD and L2

The initial algorithm used to calculate CEMD on both the CPU and GPU is shown in Figure 5.4. Calculating the CEMD between two descriptors requires more work than computing the Euclidean distance between the same descriptors. Table 5.2 illustrates the resulting difference in runtime

Table 5.2: Performance of CEMD vs. Euclidean

	seconds[a]	meas/sec[b]
CEMD	12.8	328 K
Euclidean	2.78	1.51 M

[a] Time to compute 4×2^{20} difference measurements
[b] The number of difference measurements made per second

Table 5.3: CEMD Kernel Performance

	kernel[a] seconds	total[b] seconds	kernel[c] meas/sec	total[d] meas/sec
CPU-INITIAL	-	12.8	-	328 K
CPU-FINAL	-	5.82	-	720 K
GPU-INITIAL	1.61	1.69	2.61 M	2.48 M
GPU-SHMEM	0.444	0.527	9.45 M	7.95 M
GPU-MEMOPT	0.429	0.510	9.77 M	8.22 M
GPU-INDEX	0.535	0.619	7.85 M	6.78 M
GPU-UNROLL	0.0843	0.166	49.8 M	25.2 M
GPU-FINAL	0.0771	0.165	54.4 M	25.5 M

[a] The kernel execution time on the GPU to calculate 4×2^{20} measurements with the CEMD algorithm
[b] The wall clock time with all overhead of GPU kernel execution, including copying input data from the host to the device, actual kernel run time, and copying the results back to the host
[c] The amortized number of measurements made every second, i.e. 4×2^{20} divided by column 2
[b] The amortized number of measurements made every second, including overhead costs

speed. Both the CEMD and the Euclidean algorithms in this benchmark were implemented without heavy optimizations.

Note that the CEMD algorithm in Figure 5.4 does not normalize with the number of bins as shown in Equation 5.5. Its results will be larger by a constant factor equal to the number of bins per histogram.

5.3.3 Initial Design: CPU-INITIAL and GPU-INITIAL

Both the initial CPU and GPU implementations use the CEMD algorithm shown in Figure 5.4. The CPU implementation serves as a basis of comparison for the GPU kernel performance and is also used to confirm the correctness of GPU kernel results. The performance of this CPU implementation is shown as CPU-INITIAL in Table 5.3.

The first GPU implementation of CEMD includes the copies between device and host memory

described above, and is otherwise a direct port of the CPU code. A thread $t_{I,J}$ reads in descriptors A_I and B_J directly from global memory, calculates CEMD(A_I, B_J), and writes the result to D. Its performance is shown as GPU-INITIAL in Table 5.3. This version of the GPU kernel shows a 7.96× speedup as compared to the reference CPU code.

5.3.4 Using Shared Memory: GPU-SHMEM

CUDA kernels can operate on data in several memory spaces. GPU shared memory is on-chip and has 100× lower latency than the off-chip global memory. However, shared memory is local to a thread block and is limited to 16 KiB.

Using shared memory is the first refinement to the GPU kernel. Recall that thread blocks for this kernel are 256 threads arranged in 16 rows of 16 columns. For this version of the kernel, the 16 threads $(i, 0)$ each load a descriptor A_I into shared memory, then the 16 threads $(0, j)$ each load a descriptor B_J into shared memory. To ensure the reads are complete before proceeding, they are followed by a barrier (the CUDA function `__syncthreads()`). Then, the threads calculate the CEMD as before, but now operating on the descriptors stored in shared memory.

The results of using shared memory are shown in Table 5.3 as GPU-SHMEM. This version of the kernel shows a 3.62× speedup as compared to the initial GPU implementation.

5.3.5 Optimizing Global Memory Reads: GPU-MEMOPT

Global memory performance can vary substantially based on the access pattern of memory reads. Memory operations with the right pattern can be coalesced into a single transaction, and can achieve a bandwidth roughly an order of magnitude faster than non-coalesced operations.

The global memory reads of GPU-SHMEM are not coalesced. Each thread copying a descriptor from global to shared memory loads one bin at a time, performing 128 consecutive load operations. As thread t_i is reading bin b from descriptor B_i, thread t_{i+1} is reading bin b from descriptor B_{i+1}. These loads are separated by the 128-byte length of a descriptor, preventing them from being coalesced.

The reads can be coalesced with a straightforward change to the copy procedure. The subarrays of 16 descriptors to be read from each input array are cast as arrays of 512 4-byte words. Each thread t_i in the thread block reads two words, i and $i + 256$. This guarantees that consecutive threads read consecutive words, and the global memory accesses are fully coalesced.

The result of this optimization is shown in Table 5.3 as GPU-MEMOPT. This refinement produces a 1.03× speedup versus the previous kernel, GPU-SHMEM. Only a small benefit is obtained because the CEMD kernel is not bound by memory bandwidth. The memory requirements of CEMD are as follows:

- Each block of threads computes 256 descriptor distances.
- Each block reads 32 descriptors, for a total of 4 KiB, so the kernel reads 16 bytes for each distance computed.
- At 9.77 million distances computed per second, the kernel is reading 156 MB per second.
- The GTX 295 has a 448-bit memory interface to 1 GHz GDDR3 memory. This provides a theoretical 112 GB/second of bandwidth.

Because the GPU kernel is only using a small fraction (0.14%) of the global memory bandwidth, optimizations in this area have little effect.

5.3.6 Reducing Shared Memory Bank Conflicts: GPU-INDEX

Shared memory on the GTX 295 is divided into 16 banks. Each bank holds successive 4-byte words. That is, bank 0 holds words 0, 16, 32..., bank 1 holds words 1, 17, 33..., etc. Operations on different banks can be serviced simultaneously, as can multiple reads of the same data in a single bank. However, reads from multiple locations within a single bank are referred to as a bank conflict and must be served sequentially, increasing shared memory latency.

The shared memory reads of GPU-MEMOPT generate many bank conflicts. While a given thread t_i is reading bin b from descriptor A_n, thread t_{i+1} is reading bin b of descriptor A_{n+1} at the same time. Recall that these descriptors are 128 bytes in length, so these two reads are 128 bytes apart, or 32 4-byte words apart. Since there are 16 banks and $32 \bmod 16 = 0$, both reads are

coming from the same bank and generate a bank conflict. As long as consecutive threads read the same bin from consecutive descriptors at the same time, all of the bin reads create bank conflicts. This was verified by using the CUDA profiler, which counts the warp serialization events that occur during bank conflicts. A single run of GPU-MEMOPT causes more than 156 million warp serializations.

The GPU-INDEX kernel avoids these warp serializations. GPU-INDEX is modified from the original CEMD algorithm by a change to the starting bin and histogram for each thread. When a thread is in column j within its thread block, it starts with bin b of histogram h, where

$$h = j/2 \quad \text{and} \quad b = (j \bmod 2) \times 4$$

For CUDA devices of compute capability 1.x, shared memory requests are batched by half-warps, or groups of 16 threads. Changing the starting index of CEMD computation as shown above reduces the bank conflicts between threads in a half-warp.

While GPU-INDEX shows many fewer bank conflicts—the profiler records 31 million serialization events—its performance is slightly slower, as shown in Table 5.3. This result indicates that the SIMT CUDA architecture is successfully hiding the shared memory latency in the previous kernel refinement, GPU-MEMOPT. For compute capability 1.3 devices,

- All threads within a block execute on the same streaming multiprocessor (SM). More than one block of threads can be resident in an SM at one time.
- There are 16384 32-bit registers per SM. These are shared among all thread blocks currently resident on the SM.
- GPU-MEMOPT threads use 31 registers, so a 256-thread block uses 7936 registers. The CUDA architecture allocates registers to thread blocks in multiples of 512, so this value is rounded up to 8192.
- At 8192 registers per block, two blocks can be executing on an SM at the same time. This is a total of 512 threads, 50% of the maximum of 1024 resident threads. This percentage is

referred to as the thread "occupancy."

- The SM has 8 scalar processors (SP). This means that, of the 512 threads resident on the SM, 8 can execute at the same time. If a thread cannot execute, for example because the next instruction has an operand that has not yet arrived from memory, the SM can execute instructions from any of the 504 other resident threads. This mechanism is used to hide the latency of any operation, including shared memory loads.

The additional calculations to compute h and b only add overhead, so this refinement to the kernel was discarded.

5.3.7 Loop Unrolling: GPU-UNROLL

Experience with GPU-INDEX shows that eliminating instruction overhead is a potential optimization target. The Scalar Processor (SP) cores in the GT200 chip that powers the GTX 295 can benefit greatly from loop unrolling. The maximum size of a CUDA kernel is 2 million instructions, enough capacity for significant loop unrolling.

For the GPU-UNROLL kernel a combination of C++ template and preprocessor metaprogramming is used to manually unroll the CEMD algorithm. After unrolling, there is a single loop that executes 16 times for each distance calculated, once per histogram. The two levels of nested loops below this, to calculate cumulative histograms starting at an index and to repeat that process for each bin in a histogram, are completely flattened. This approach is very effective at reducing the computation time. The results are shown in the GPU-UNROLL line of Table 5.3. The unrolling produces a 5.10× speedup as compared to the GPU-MEMOPT kernel.

This large speedup was not entirely the result of unrolling. Profiling shows that only 20 million warp serializations occur during the execution of GPU-UNROLL—almost 8× less than GPU-MEMOPT. This surprising result occurs because of compiler optimizations that only occur after loop unrolling. Consider that for each pair of 8-bin histograms, CEMD calculates 8 cumulative histograms. Each bin value is read once per cumulative histogram, but the bin values do not change. After unrolling, the compiler recognized the redundant reads and eliminates them by

keeping the bin values in registers.

After this refinement, it is unlikely that further changes can greatly accelerate the execution of this algorithm. With the redundant reads eliminated as described above, and 16-histogram descriptors with 8 bins per descriptor:

- There are 16 load instructions per pair of histograms
- There are 8 subtractions, 8 absolute values, and 21 additions per cumulative histogram.
- There are 8 cumulative histograms and 7 minimum operations to find the smallest of them, requiring 319 operations per histogram.
- Each descriptor distance has 16 pairs of histograms and 15 add instructions. Ignoring the overhead of the histogram loop, there are 5119 operations per descriptor distance.
- The 4×2^{20} distance calculations performed by the kernels in Table 5.3 require 21.47 billion operations, ignoring some overhead such as the global to shared memory reads.
- The GT200 chip in the GTX 295 has 240 processors running at 1.24 GHz, for a total of 297.6 billion instructions per second. The minimum time to complete the test calculations is $21.47/297.6 = 0.072$ seconds

The measured time of 0.084 seconds indicates that this kernel is operating at roughly 85% of the theoretical maximum based only on the time required to issue instructions; i.e., ignoring all instruction latency. This means that the SIMT execution model is hiding most of that latency, and this algorithm is unlikely to see substantial further speedups.

5.3.8 Algorithmic Improvements: GPU-FINAL and CPU-FINAL

Further speedups require a better CEMD algorithm. Consider the definition of CEMD labeled Equation 5.5. It requires repeated summation of the elements of D_j for $0 \leq j < N$, where N is the number of bins per histogram and

$$D_j[i] = X_j[i] - Y_j[i]$$

```
function CEMD(x, y)
    acc_x ← 0; acc_y ← 0
    for i from 0 to B do
        acc_x ← acc_x + x_i
        acc_y ← acc_y + y_i
        c[i] ← acc_x − acc_y
    end for
    memd ← ∞
    for i from 0 to B do
        d ← 0
        for k from i to B do
            d ← d + |c[i] − c[k − 1]|
        end for
        for k from 0 to i do
            d ← d + |c[i] − c[k − 1] + c[B − 1]|
        end for
        memd ← min(memd, d)
    end for
    return memd
end function
```

Figure 5.5: Modified CEMD Algorithm

To find the values of D_j, the original CEMD algorithm, shown as MEMDISTANCE in Figure 5.4, calculates each histogram X_j and Y_j for $0 \leq j < N$.

However, only the initial cumulative histograms X_0 and Y_0 need to be calculated. These can be subtracted to find D_0, and further difference-of-cumulative values D_k for $k > 0$ can be derived from D_0 with

$$D_k[i] = \begin{cases} D_0[i] - D_0[k-1] & \text{if } i \geq k, \\ D_0[i] - D_0[k-1] + D_0[B-1] & \text{if } i < k. \end{cases}$$

The updated function MEMDISTANCE in Figure 5.5 uses this method to calculate the CEMD, and is similar to the implementation described in [123].

As mentioned above, the original function requires 319 operations per histogram. The new function only requires 210, leading to the performance shown as GPU-FINAL in Table 5.3. This is a 1.09× speedup as compared to the GPU-UNROLL kernel. Given the 34% reduction in work done, the limited 9% increase in performance indicates that the GPU-FINAL kernel is being limited

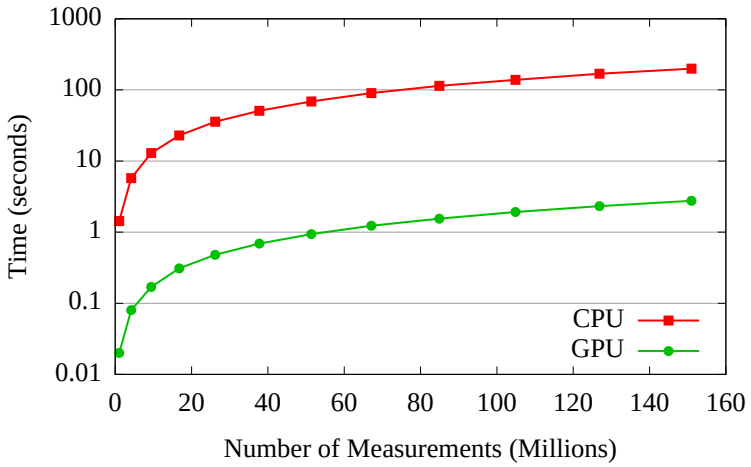


Figure 5.6: Time to Complete N Measurements

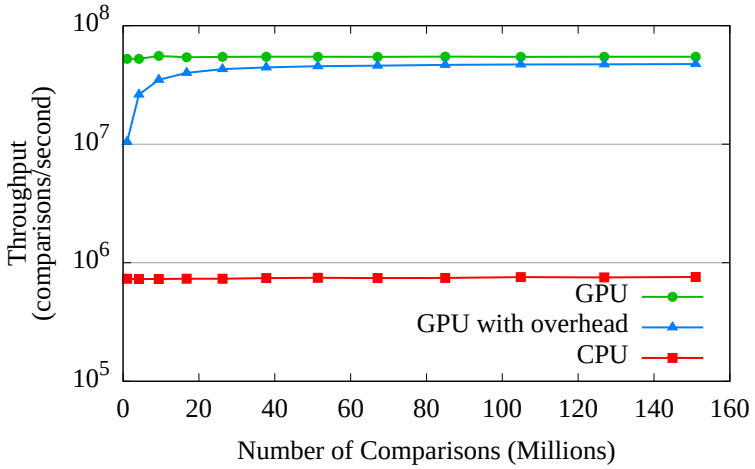


Figure 5.7: Throughput of N Measurements

by another bottleneck, such as shared memory latency.

Loop unrolling and the improved algorithm can be profitably used in the CPU implementation of the CEMD kernel as well. After updating the CPU function to incorporate the improved algorithm and loop unrolling, the performance is shown as CPU-FINAL in Table 5.3. The final optimized version of the GPU kernel is 75.56× faster than the final optimized version of the CPU kernel.

5.3.9 Performance Scaling

The figures in Table 5.3 all show the time to complete about four million CEMD calculations. The performance of the final CPU and final GPU implementations on larger input sizes is shown in Figure 5.6. The largest problem size shown is 151 million comparisons, which requires a 576 MiB result array in addition to the storage required for the input descriptor arrays. The GTX 295 has 896 MiB available to each GPU, which limits the problem size that can be computed in a single kernel invocation.

GPU-FINAL is consistently 70–75$\times$ faster than CPU-FINAL for all problem sizes. The same results in terms of calculations per second are shown in Figure 5.7. Both the CPU and GPU kernels have very consistent performance. Figure 5.7 has an additional line showing the performance of the GPU kernel including overhead costs. The primary source of overhead are the transfers of input data to the GPU and the CEMD results back to host memory.

For a given launch, there is $O(|n| + |m|)$ input data transferred and $O(|n| \times |m|)$ computations performed, so the overhead of data input becomes negligible for large enough data sets. However, there are $O(|n| \times |m|)$ results to transfer back to the host, so the output time remains relatively important for all data set sizes. For larger problem sizes, the throughput including overhead is 13% slower than the GPU kernel alone. For smaller problem sizes, including the size shown in Table 5.3, the input data transfer time, and constant factors such as kernel invocation time, are relevant. These can reduce the throughput by up to 50%.

5.3.10 Streaming Kernel Execution

If the sum total of memory required by the input arrays of descriptors and the output array of results is larger than device memory, result calculation must be broken into multiple kernel launches. There is no dependence between the results, so the result array can be broken into tiles and one kernel can be launched to calculate the results in each tile. For each tile, a subset of the input arrays are copied to the device, and the results are copied out.

Devices with compute capability 1.1 or higher can overlap the execution of a kernel with a

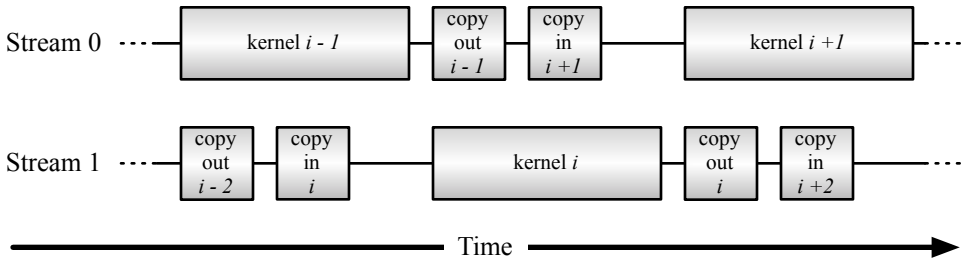


Figure 5.8: Streaming Execution in CUDA

single device-host memory operation. Figure 5.8 shows the typical flow of execution. While a kernel is executing, the results of the last kernel are copied to the host, and the input to the next kernel is copied to the device. This double-buffering imposes several restrictions:

1. Because memory must be allocated for two kernel launches at a time, each kernel launch is limited to calculating a smaller number of results.
2. Asynchronous operation requires page-locked or "pinned" host memory. Systems make a limited amount of this memory available, possibly further limiting buffer size, especially in multi-GPU systems.

This style of concurrent execution was utilized in a cluster-based system, and is described below.

5.3.11 Using Multiple GPUs on Multiple Machines

The GPU kernel was used as part of a cluster-based implementation of gpu-cemd. This system can use multiple nodes, each with several GPU devices, to quickly calculate many CEMD results. Each node was managed by a single MPI process, and was responsible for comparing the complete set of input features to a subset of the reference features. The nodes use OpenMP to create a management thread for each GPU. The comparisons to be performed by the node were divided among the GPUs, then further divided into a number of kernel launches which were then dispatched in two overlapping streams to each GPU. Because there are no dependencies between CEMD calculations, the throughput of those calculations scales linearly with the number of GPUs in the cluster. With 24 GPUs in six nodes, the test cluster performed over 1.2 billion comparisons per second.

5.4 Conclusion

The gpu-cemd implementation of the CEMD measurement for SIFT feature descriptors is the first to use GPUs for this purpose. Given the large number of features generated from geospatial images, and the computational expense of CEMD relative to simpler measurements, high-performance tools are needed to make CEMD practical in geospatial applications.

The instruction throughput of gpu-cemd, versus the theoretical peak of the GPU it runs on, indicates that gpu-cemd is taking full advantage of the SIMT hardware of the GPU — no implementation of CEMD on the GPU can be faster. This results in over a 70× speedup over a single-threaded CPU implementation of the same algorithm. However, the process of developing gpu-cemd, or any GPU kernel, is not straightforward because the CUDA platform exposes a large design space to developers.

Where there can be data reuse, developers should always utilize the shared memory pool. Simple use of shared memory lead to a 3.6× speedup for the GPU CEMD kernel. Note that the GPU kernel was 8× faster than the CPU without using shared memory, so effective use of the shared memory space may not be prerequisite to seeing significant gains with a GPU accelerator. While virtually all GPU kernels should implement shared memory usage, the utility of other optimizations will vary widely between different kernels. Coalescing global memory access can have dramatic effects on bandwidth-limited algorithms, but had little effect on the CEMD kernel.

Instead, loop unrolling was one of the most useful refinements for the CEMD implementation. Because CUDA uses a toolchain based on C/C++, this loop unrolling was implemented using a combination of C preprocessor and C++ template metaprogramming. This type of rigorous, manual loop unrolling is more likely to be useful in GPU programming than with traditional CPU programming. In general, identifying useful useful approaches to performance optimization of GPU code requires knowledge of the GPU architecture and the use of profiling and timing to confirm the efficacy of various optimizations.

Chapter 6

Conclusions

6.1 Results

The work described in this dissertation has a number of useful results:

- The "prefix tree" data structure described in Chapter 3 uses an implicit binary tree to store the intermediate calculations of a traditional parallel prefix operation. This allows online updates to parallel prefix result in logarithmic time with very low overhead.
- Carefully-chosen methods of calculating line-of-sight over a rasterized height map allows the use of the prefix tree data structure to be used in place of a traditional binary tree for viewshed analysis with a planar sweep approach.
- The planar sweep can be partitioned radially to facilitate parallel calculation on multi-core computers. Combined with the low memory usage of the prefix tree, this currently allows essentially linear speedup with up to eight CPUs.
- The three items above are implemented in a software application called "pvshed" which is orders of magnitude faster than available viewshed analysis software.
- Scale-spaced based feature extraction algorithms can be partitioned in scale space, rather than in image space, as described in Chapter 4. This allows the algorithm to exploit multi-core and cluster computers with minimal change to the algorithm itself, and little change to the results.
- By processing the scale space partitions in a thread-pool-based pipeline, feature extraction can accommodate a wide variety of underlying hardware. This also facilitates out-of-core processing, decoupling the maximum input image size from the available amount of system memory.
- The previous two items are implemented in a software application named SOHC. This is

the first SIFT implementation capable of processing arbitrarily large images, and it can take advantage of GPU-based systems as well as multi-core and cluster computers.

- A GPU-based implementation of the CEMD algorithm which greatly accelerates feature dissimilarity calculations using the algorithm. The implementation, gpu-cemd, makes nearly-optimal use of the GPU hardware.

These results have already been used as the basis for further geospatial applications. The Center for Advanced Spatial Technologies at the University of Arkansas has used pvshed as the basis of an interactive web-based viewshed analysis application, which would be difficult or impossible to develop with previous viewshed algorithms. The gpu-cemd code was used in research attempting to modify SIFT for better multi-modal performance, where its speed made parameter sweeps more practical. Finally, SOHC is planned for use in future research on object tracking in large images.

6.2 Applicability of Parallel Computing to Given Applications

Multi-core Computing: Both pvshed and SOHC make efficient use of multi-core architectures. In general, the spatial nature of their input data should make it possible to partition geospatial applications to take advantage of the coarse-grained parallelism of multi-core computers, while the shared memory of these systems means that locality is not as great a concern as with distributed-memory systems. Increasing input size and the recent ubiquity of multi-core processors will drive most geospatial applications to utilize this type of parallelism. Most of these applications will also be able to use the task-based approach to concurrent programming demonstrated by libraries and frameworks such as TBB, which was used in pvshed, to avoid the traditional difficulties of using threaded programming to utilize multi-core computers.

GPU Computing: The gpu-cemd software makes great use of GPUs, and other applications that process large arrays of SIFT or other generated features will likely be able to use this sort of parallel computer. Locality of computation is even more important for hybrid clusters than other distributed-memory systems, because the GPUs have no direct access to network resources,

increasing the overhead of communication.

Viewshed analysis is unlikely to benefit from GPU acceleration. The radial partitions around the viewpoint can be swept in parallel, and thus are a good fit for multi-core parallelism. However, the sweep of each partition is inherently sequential, because updates to the status structure for a single event must be performed sequentially, and events must be processed in order, as well. The sweeps also exhibit non-regular data access patterns as they process the event list. This presupposes an approach based on a planar sweep; other approaches might achieve faster speeds if they are a better match to the GPU architecture.

Because SOHC's partitioning approach allows it to use mostly-unchanged feature extraction algorithms, it can also take advantage of efforts by the computer vision community to accelerate those algorithms with GPUs. The data-parallel nature of these algorithms makes this a likely-fruitful avenue of research.

Cluster Computing: In general, cluster computers are useful for applications where there is a high ratio of computation to data, and good locality. Simulations of fluids or particles are good examples, because an initial state is loaded, then an arbitrary number of simulation iterations can be performed, based on mostly local interactions.

In contrast, feature extraction involves a fixed amount of computation for given inputs. The computational power of a cluster computer, for these problems, quickly exceeds the aggregate secondary storage bandwidth of a cluster. With enough network bandwidth, SOHC might take advantage of cluster in a "capacity computing" fashion; that is, by simultaneously processing many jobs, with input images loaded directly to node-resident secondary storage. For viewshed analysis, data can be reused many times for the calculation of multiple viewsheds. However, there is poor locality: the computation is divided radially around the viewpoint; these partitions change with each viewshed calculation. Much like SOHC, the gpu-cemd application could also operate in a capacity-based fashion with enough aggregate storage bandwidth.

Given these factors, distributed-memory clusters are the most difficult parallel computing plat-

form to use effectively in geospatial applications.

Bibliography

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[2] P. Kogge *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," Sep. 2008.

[3] V. Sarkar *et al.*, "Exascale software study: Software challenges in extreme scale systems," Sep. 2009.

[4] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, Feb. 1990.

[5] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[6] *POSIX.1c, Threads extensions*, IEEE Std. 1003.1c, 1995.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Longman, 2005.

[8] A. Williams, "The Boost thread library," http://boost.org/libs/thread, 2008.

[9] "Microsoft developer's network: Processes and threads," http://msdn.microsoft.com/en-us/library/ms684841.aspx, 2011.

[10] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors," *IEEE Workload Characterization Symposium*, vol. 0, pp. 225–236, 2006.

[11] A. Kleen, "A NUMA API for linux," White Paper, Novell, Apr. 2005.

[12] E. A. Lee, "The problem with threads," University of California at Berkeley, Tech. Rep. UCB/EECS-2006-1, Jan. 2006.

[13] The OpenMP API specification for parallel programming. [Online]. Available: http://openmp.org/wp/

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216.

[15] R. Geva, "Elemental functions: Writing data-parallel code in C/C++ using Intel Cilk Plus," White Paper, 2011.

[16] T. Willhalm and N. Popovici, "Putting Intel threading building blocks to work," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*. New York, NY, USA: ACM, 2008, pp. 3–4.

[17] (2011) Grand Central Dispatch (GCD) reference. Apple. [Online]. Available: http://developer.apple.com/mac/library/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html

[18] (2011) Parallel computing. Microsoft. [Online]. Available: http://msdn.microsoft.com/en-us/concurrency/default.aspx

[19] (2011) The go programming language. Google. [Online]. Available: http://golang.org/

[20] *MPI: A Message Passing Interface*, Message Passing Interface Forum Std. 2.2, 2009.

[21] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: Programming for hierarchical parallelism and non-uniform data access," in *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[22] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," in *International Journal of High Performance Computing Applications*, vol. 21, Aug. 2007, pp. 291–312.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, 2008.

[24] "NVIDIA's next generation CUDA compute architecture: Fermi," White Paper, NVIDIA, Jul. 2010.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[26] *CUDA C Progamming Guide 3.2*, NVIDIA, 2010.

[27] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, pp. 777–786, Aug. 2004.

[28] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," *ACM Trans. Graph.*, vol. 23, pp. 787–795, Aug. 2004.

[29] S. Xiao, A. M. Aji, and W. chun Feng, "On the robust mapping of dynamic programming onto a graphics processing unit," *Parallel and Distributed Systems, International Conference on*, vol. 0, pp. 26–33, 2009.

[30] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*. New York, NY, USA: ACM, 2009, pp. 145–149.

[31] A. Gharaibeh and M. Ripeanu, "Size matters: Space/time tradeoffs to improve GPGPU applications performance," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.

[32] D. Tarjan, J. Meng, and K. Skadron, "Increasing memory miss tolerance for SIMD cores," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[33] (2010) OpenCL. Khronos Group. [Online]. Available: http://www.khronos.org/opencl/

[34] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL," *Computing Research Repository*, vol. abs/1005.2581, 2010.

[35] (2010) DirectX developer center. Microsoft. [Online]. Available: http://msdn.microsoft.com/en-us/directx/default

[36] (2011) CUDA fortran programming guide and reference. The Portland Group. [Online]. Available: http://www.pgroup.com/doc/pgicudaforug.pdf

[37] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. [Online]. Available: http://www.meganewtons.com/

[38] T. D. Han and T. S. Abdelrahman, "hiCUDA: A high-level directive-based language for GPU programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 52–61.

[39] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, 2010, pp. 1–11.

[40] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *IEEE 7th Symposium on Application Specific Processors (SASP '09)*, Jul. 2009, pp. 35–42.

[41] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "High-performance CUDA kernel execution on FPGAs," in *Proceedings of the 23rd international Conference on Supercomputing (ICS '09)*. New York, NY, USA: ACM, 2009, pp. 515–516.

[42] "PGI CUDA-x86: CUDA programming for multi-core CPUs," PGI Insider, The Portland Group, Nov. 2010.

[43] (2010) AMD accelerated parallel processing (APP) SDK. AMD. [Online]. Available: http://developer.amd.com/gpu/amdappsdk

[44] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," ISSCC, 2005.

[45] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, Aug. 2008.

[46] N. Brookwood, "AMD fusion family of APUs," White Paper, AMD, Mar. 2010.

[47] (2010) Intel microarchitecture codename sandy bridge. Intel. [Online]. Available: http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm

[48] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New frontiers in performance improvements and energy efficiency," White Paper, Intel, May 2008.

[49] S. Kang, D. A. Bader, and R. Vuduc, "Understanding the design trade-offs among current multicore systems for numerical computations," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–12, 2009.

[50] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, 2010.

[51] (2010, Aug.) CUBLAS library. NVIDIA. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf

[52] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. Vol. 180. Innovative Computing Laboratory, University of Tennessee, 2009.

[53] A. Petitet, C. Whaley, J. Dongarra, and A. Cleary. (2008) HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. [Online]. Available: http://www.netlib.org/benchmark/hpl/

[54] M. Fatica, "Accelerating linpack with CUDA on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 46–51.

[55] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Atlanta, Apr. 2010, pp. 1–8.

[56] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[57] M. Hussein and W. Abd-Almageed, "Efficient band approximation of gram matrices for large scale kernel methods on gpus," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–10.

[58] D. Merrill and A. Grimshaw, "Revisiting sorting for GPGPU stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Tech. Rep. CS2010-03, 2010.

[59] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–10.

[60] S. S. Hampton, S. R. Alam, P. S. Crozier, and P. K. Agarwal, "Optimal utilization of heterogeneous resources for biomolecular simulations," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[61] R. Babich, M. A. Clark, and B. Joó, "Parallelizing the QUDA library for multi-GPU calculations in lattice quantum chromodnamics," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[62] G. A. Elliott and J. H. Anderson, "Real-time multiprocessor systems with GPUs," In submission, Jun. 2010.

[63] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units*, Mar. 2010.

[64] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 47–.

[65] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. mei Hwu, "GPU clusters for high-performance computing," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, Sep. 2009, pp. 1–8.

[66] (2011) Top500 supercomputing sites. TOP500 Project. [Online]. Available: http://top500.org

[67] T. Hamada and K. Nitadori, "190 TFlops astrophysical n-body simulation on a cluster of GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9.

[68] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical N-body simulations on GPU clusters," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[69] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *SC '09: Proceedings of the*

Conference on High Performance Computing, Networking, Storage, and Analysis. New York, NY, USA: ACM, 2009, pp. 1–12.

[70] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[71] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-fold speedup, 15.0 tflops full GPU acceleration of non-hydrostatic weather model ASUCA production code," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[72] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: The architecture and performance of Roadrunner," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1:1–1:11.

[73] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13–13, 2007.

[74] E. Boci, S. Sarkani, and T. Mazzuchi, "Optimizing ADS-B RF coverage," in *Integrated Communications, Navigation and Surveillance Conference (ICNS)*, May 2009, pp. 1–10.

[75] J. Lubczonek, "Application of GIS techniques in VTS radar stations planning," in *Radar Symposium, 2008 International*, May 2008, pp. 1–4.

[76] A. Xiao, H. Bertoni, C. Chrysanthou, and J. Boksiner, "Fast prediction of scattering in mountainous terrain using commonly visible surfaces," in *Sarnoff Symposium, 2010 IEEE*, Apr. 2010, pp. 1–5.

[77] M. Lu, J. Zhang, P. Lv, and Z. Fan, "Max/min path visual coverage problems in raster terrain," in *Computer-Aided Design and Computer Graphics, 2007 10th IEEE International Conference on*, Oct. 2007, pp. 497–500.

[78] J. He, Y. Tian, X. Ming, and Y. Xu, "The optimal path planning based on landscape ecology and health science: A case study of JinYun Mountain in ChongQing, China," in *Geoinformatics, 2010 18th International Conference on*, Jun. 2010, pp. 1–4.

[79] Y. Iikura, "Precise evaluation of topographic effects in satellite imagery for illumination correction," in *Geoscience and Remote Sensing Symposium, 2008. IGARSS 2008. IEEE International*, vol. 3, Jul. 2008, pp. 1091–1094.

[80] S. V. G. Magalhães, M. V. A. Andrade, and W. R. Franklin, "An optimization heuristic for siting observers in huge terrains stored in external memory," in *Hybrid Intelligent Systems (HIS), 2010 10th International Conference on*, Aug. 2010, pp. 135–140.

[81] J. P. Wilson and J. C. Gallant, *Terrain Analysis: Principles and Applications*. Wiley, 2000, ch. 1.

[82] D. B. Gesch, *Digital Elevation Model Technologies and Applications: The DEM Users Manual*, 2nd ed., American Society for Photogrammetry and Remote Sensing, Bethesda, Maryland, USA, 2007.

[83] L. De Floriani, B. Falcidieno, C. Pienovi, D. Allen, and G. Nagy, "A visibility-based model for terrain features." in *Proceedings of the Second Interational Symposium on Spatial Data Handling*, 1986, pp. 235–250.

[84] R. Cole and M. Sharir, "Visibility problems for polyhedral terrains," *J. Symb. Comput.*, vol. 7, pp. 11–30, Jan. 1989.

[85] J. Lee, "Analyses of visibility sites on topographic surfaces," *International Journal of Geographical Information Science*, vol. 5, no. 4, pp. 413–429, 1991.

[86] W. R. Franklin and C. K. Ray, "Higher isn't necessarily better: Visibility algorithms and experiments," in *Advances in GIS Research: Sixth International Symposium on Spatial Data Handling*, 1994, pp. 751–770.

[87] M. van Kreveld, "Variations on sweep algorithms: Efficient computation of extended viewsheds and class intervals," in *Proc. 7th Int. Symp. on Spatial Data Handling*, 1996, pp. 15–27.

[88] B. Kaučič and B. Zalik, "Comparison of viewshed algorithms on regular spaced points," in *SCCG '02: Proceedings of the 18th Spring Conference on Computer Graphics*. New York, NY, USA: ACM, 2002, pp. 177–183.

[89] D. Izraelevitz, "A fast algorithm for approximate viewshed computation," *Photogrammetric Engineering and Remote Sensing*, vol. 69, no. 7, pp. 767–774, Jul. 2003.

[90] L. Liu, L. Zhang, C. Chen, and H. Chen, "An improved LOS method for implementing visibility analysis of 3D complex landscapes," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 2, Dec. 2008, pp. 874–877.

[91] Y. Shen, L. Lin, M. Yang, and G. Yurong, "Viewshed computation based on LOS scanning," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 2, Dec. 2008, pp. 984–987.

[92] Z.-Y. Xu and Q. Yao, "A novel algorithm for viewshed based on digital elevation model," in *Information Processing, 2009. APCIP 2009. Asia-Pacific Conference on*, vol. 2, Jul. 2009, pp. 294–297.

[93] Y. Xia, Y. Li, and X. Shi, "Parallel viewshed analysis on GPU using CUDA," in *Computational Science and Optimization (CSO), 2010 Third International Joint Conference on*, vol. 1, May 2010, pp. 373–374.

[94] M. Magalhães, S. Magalhães, M. Andrade, and J. Filho, "An efficient algorithm to compute the viewshed on DEM terrains stored in external memory," *Brazilian Symposium on GeoInformatics*, pp. 183–194, 2007.

[95] H. Haverkort, L. Toma, and Y. Zhuang, "Computing visibility on terrains in external memory," *J. Exp. Algorithmics*, vol. 13, pp. 1.5–1.23, 2009.

[96] J. Fishman, H. Haverkort, and L. Toma, "Improved visibility computation on massive grid terrains," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '09)*. New York, NY, USA: ACM, 2009, pp. 121–130.

[97] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.

[98] GRASS Development Team, *Geographic Resources Analysis Support System (GRASS GIS) Software*, Open Source Geospatial Foundation, 2010. [Online]. Available: http://grass.osgeo.org

[99] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[100] ——, "Object recognition from local scale-invariant features," in *Proc. Seventh IEEE International Conference on Computer Vision*, vol. 2, 1999, pp. 1150–1157.

[101] ——, "Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image," U.S. Patent 6,711,293, Mar., 2004.

[102] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 10, pp. 1615–1630, 2005.

[103] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (SURF)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, 2008.

[104] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. V. Gool, "A comparison of affine region detectors," *Int. J. Comput. Vision*, vol. 65, no. 1-2, pp. 43–72, 2005.

[105] D. G. Lowe. (2011) Keypoint detector. [Online]. Available: http://www.cs.ubc.ca/~lowe/keypoints

[106] A. Vedaldi. (2009) SIFT++ source code and documentation. [Online]. Available: http://www.vlfeat.org/~vedaldi/code/siftpp.html

[107] A. Vedaldi and B. Fulkerson, "VLFeat: An open and portable library of computer vision algorithms," http://www.vlfeat.org/, 2008.

[108] H. Feng, E. Li, Y. Chen, and Y. Zhang, "Parallelization and characterization of SIFT on multi-core systems." in *IISWC*, D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, Eds. IEEE, 2008, pp. 14–23.

[109] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "SIFT implementation and optimization for multi-core systems," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, Apr. 2008, pp. 1–8.

[110] S. Heymann, K. Muller, A. Smolic, B. Froehlich, and T. Wiegand, "SIFT implementation and optimization for general-purpose GPU," in *WSCG'07*, 2007.

[111] C. Wu, "SiftGPU: A GPU implementation of scale invariant feature transform (SIFT)," http://cs.unc.edu/~ccwu/siftgpu, 2007.

[112] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," *Machine Vision and Applications*, Mar. 2007.

[113] S. Bobovych, W. Emeneker, S. Warn, J. Cothren, and A. Apon, "Parallelization of the scale invariant feature transform (poster)," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2010.

[114] S. Warn, W. Emeneker, J. Cothren, and A. Apon, "Accelerating SIFT on parallel architectures," in *IEEE International Conference on Cluster Computing and Workshops (CLUSTER '09)*, Sep. 2009, pp. 1–4.

[115] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, Oct. 2008.

[116] L. Chao, "Hyper-threading technology," *Intel Technology Journal*, vol. 6, no. 1, Feb. 2002.

[117] R. Szeliski, "Image alignment and stitching: A tutorial," *Found. Trends. Comput. Graph. Vis.*, vol. 2, no. 1, pp. 1–104, 2006.

[118] (2010) GeoStor: Arkansas' official GIS platform. [Online]. Available: http://www.geostor.arkansas.gov/

[119] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, pp. 381–395, June 1981.

[120] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Applications*, 2009.

[121] H. Jegou, H. Harzallah, and C. Schmid, "A contextual dissimilarity measure for accurate and efficient image search," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR '07)*, Jun. 2007, pp. 1–8.

[122] Q. Wang and S. You, "Fast similarity search for high-dimensional dataset," in *ISM '06: Proceedings of the Eighth IEEE International Symposium on Multimedia*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 799–804.

[123] J. Rabin, J. Delon, and Y. Gousseau, "Circular earth mover's distance for the comparison of local features," in *International Conference on Pattern Recognition*, Dec. 2008.

[124] Y. Rubner, C. Tomasi, and L. J. Guibas, "A metric for distributions with applications to image databases," in *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*. Washington, DC, USA: IEEE Computer Society, 1998, p. 59.

[125] ——, "The earth mover's distance as a metric for image retrieval," *International Journal of Computer Vision*, vol. 40, pp. 99–121, 2000.

[126] J. Rabin, J. Delon, and Y. Gousseau, "A statistical approach to the matching of local features," *SIAM Journal on Imaging Sciences*, vol. 2, no. 3, pp. 931–958, 2009.