9-2009

Accelerating SIFT on Parallel Architectures

Amy Apon
Clemson University, aapon@clemson.edu

Seth Warn
University of Arkansas - Main Campus

Wesley Emeneker
University of Arkansas - Main Campus

Jackson Cothren
University of Arkansas - Main Campus

Recommended Citation

Accelerating SIFT on Parallel Architectures

Seth Warn [#1], Wesley Emeneker [#2], Jackson Cothren [*3], Amy Apon [#4]

[#] *Computer Science and Computer Engineering, University of Arkansas*
504 JBHT, University of Arkansas, Fayetteville AR, 72701, USA
[1] swarn@uark.edu
[2] ewe@uark.edu
[4] aapon@uark.edu

[*] *Center for Advanced Spatial Technologies, University of Arkansas*
304 JBHT, University of Arkansas, Fayetteville AR, 72701, USA
[3] jcothren@cast.uark.edu

Abstract—SIFT is a widely-used algorithm that extracts features from images; using it to extract information from hundreds of terabytes of aerial and satellite photographs requires parallelization in order to be feasible. We explore accelerating an existing serial SIFT implementation with OpenMP parallelization and GPU execution.

I. INTRODUCTION

Computer vision attempts to extract features and discern information about images. Its use is well-known in near-real-time applications like robot maneuvering or object tracking, but extracting information from images is also useful in other types of problems. For example, photogrammetry uses computer vision algorithms to extract geometric and geographic information from images. We want to apply some of those techniques to a large archive of aerial and satellite imagery, determining if and where those images overlap, and the correct rotation, translation, and scale to "stitch" those images together. Because our archive has more than one hundred terabytes of raw images, we are investigating a number of avenues to parallelize and accelerate the process.

Matching and stitching images is performed by an application with two major, computationally-intensive phases. The first phase extracts features using David Lowe's Scale Invariant Feature Transform (SIFT) algorithm [1]. The second phase of computation takes combinations of images and uses the results from the first phase to find any overlapping regions. Our work focuses on accelerating the SIFT algorithm's feature extraction. There are a number of operations in the SIFT algorithm:

- Scale space construction: The image is repeatedly convolved with a Gaussian convolution kernel. This produces a series of increasingly-blurred versions of the original image.
- Difference of Gaussian calculation: The difference between adjacent images in Gaussian scale space is calculated. This approximates the scale-normalized Laplacian of Gaussian.

- Keypoint identification: Local extrema in the Laplacian are found by comparing difference-of-Guassian values with their eight neighbors at the same scale, and nine neighbors at scales above and below. Extrema are recorded as possible keypoints.
- Keypoint filtering: Potentially unstable keypoints, such as those in areas of low contrast or along edges, are removed.
- Keypoint orientation: The dominant gradient of the neighborhood around the point is determined. The description of the point is relative to this gradient, making it rotation-invariant.
- Keypoint descriptor creation: Histograms are calculated that describe the neighborhood of the point. These histograms form a vector that serves as the descriptor of the keypoint.

The purpose of this project is to determine the efficacy of two acceleration techniques, applied to the same code base. We use an existing implementation of the SIFT algorithm, SIFT++ [2], as the starting point for our work. To identify the portions of the application that will benefit most from parallelization, we use a code profiler to identify compute-intensive functions within SIFT++. Then, we create two separate, accelerated versions of original application.

First, we implement a traditional and simple parallelization with OpenMP. This allows the application to take advantage of all the CPUs in an SMP architecture, and is a relatively straightforward method of parallelizing code. Second, we create a version that executes portions of the SIFT algorithm on a NVIDIA Graphics Processing Unit (GPU) with CUDA [3] capabilities. General-purpose processing on GPUs (GPGPU) is a relatively new technique than can accelerate some applications by several orders of magnitude, but is typically more complex to implement.

We benchmark each version and compare the results. Because GPGPU requires a GPU and extra programming effort, we are interested in whether OpenMP is "good enough," i.e., if SIFT++ accelerated with OpenMP scales well enough for the purposes of the larger project. Additionally, we are interested in the return on investment of a GPGPU solution; is a GPU-enhanced SIFT++ accelerated enough to justify the extra effort

and expense required to implement it?

II. METHODOLOGY

A. SIFT++

As mentioned above, we use an existing application, SIFT++, as a starting point for our work. It is an implementation of SIFT in C++ with a single binary `sift`. Our tests use it with the default command line options, reading a single image and generating a list of keypoint descriptors. Given the existing code, our desire is to reduce the runtime while generating identical results.

Though SIFT++ is considered superseded by the VLFeat suite of computer vision algorithms [4], a brief test found that SIFT++ is both faster and uses less memory than the VLFeat implementation of SIFT for the large images typically used in our workloads.

B. Profiling

We used code profiling to to identify the most computationally intense portions of SIFT++. Accelerating these portions of the code will presumably show the greatest effect on runtime. We used the GNU `gprof` utility to perform the profiling; abbreviated profile output from is shown in Table I on the following page. The results in this table were generated from an image *paris.pgm* with dimensions 4136x1424; SIFT++ generates 42605 keypoints for *paris.pgm*.

The results of the profiling show that three functions consume more than ninety percent of the computation time, and are the best candidates for parallelization:

1) `prepareGrad` performs the keypoint orientation.
2) `econvolve` is used for scale space construction.
3) `computeKeypointDescriptor` generates keypoint descriptors for detected features.

We chose the function `econvolve` to parallelize first. It consumes 40% of the application runtime. It implements a simple, one-dimensional convolution pass; it is used twice with Guassian kernel values to accomplish the Gaussian smoothing required for scale space construction. The convolution is straightforward to parallelize, consisting primarily of a multiply-accumulate operation inside a nested loop.

C. OpenMP

OpenMP is an established way of modifying serial code to take advantage of parallelism on an SMP system. Essentially, OpenMP works by transforming sections of serial code into regions that can execute in parallel. The programmer specifies which regions of code are to be parallelized, and tells OpenMP which variables are private, public, shared between threads, etc. With this knowledge, the compiler can take loops (for example), and split iterations among threads. With proper separation, the end result of an OpenMP loop is the same as the serialized loop. In OpenMP, any variable that is dependent upon the results of previous iterations must be removed or computed independently. Additionally, any parallelized loops must have stopping conditions not dependent on functions. (OpenMP 3.0 can support iterators and function calls for loop conditions, but at the time of implementation and writing, the gcc compiler did not support the new functionality.)

Adding OpenMP parallelism to each of the three functions `econvolve`, `computeKeypointDescriptor`, and `prepareGrad` requires less than 20 new or changed lines of code in total. That amounts to a less than one percent difference between the original and the OpenMP parallelized code. Additionally, the changes implemented required approximately 12 hours to implement and test for correctness.

D. GPU

Accelerating an application with CUDA is a more complex process. NVIDIA describes to their device architecture as "Single-Instruction, Multiple Thread," or SIMT, referring to how it is programmed with multiple, instruction-locked threads. Additionally, the device has a complex memory hierarchy, with multiple disjoint memories and special requirements to maximize memory bandwidth. Unlike with OpenMP, serial code can not be extended to make efficient use of GPGPU capabilities; it must be rewritten to take advantage of the GPU architecture.

There are two types of code in a CUDA application. First, there is the "host" code, which is essentially unmodified C/C++ that runs on the CPU. Second is the "device" code, written in a subset of C with a number of CUDA-specific extensions. Typically, the host code will copy input data into the devices memory, then make a call to a function specified in the device code. The CUDA runtime translates this call, downloading the device code to the GPU as a compiled kernel and executing it, then returning execution to the host code when the kernel has completed. Then, the host will copy the output data from the device into system memory.

Of the two most time-consuming functions, `econvolve` is best-suited to acceleration on the GPU. It uses memory buffers for the input image and resulting output images, making it easy to match with the typical operation described above. Also, it is called relatively few times, incurring the overhead of the host/device transition less often. The descriptor computation (`computeKeypointDescriptor`) requires access to more, less cleanly-separated input data, and is called many more times, making it less suited to GPU acceleration.

Our accelerated convolution kernel is based on code available in the CUDA SDK. The kernel addresses the two major issues that arise from the architectural features described above: using the GPU at maximum efficiency, and coalesced memory access.

Using the GPU efficiently – by keeping all of its resources productively working – is more difficult on the GPU than a multi-CPU system, because of its SIMT architecture. CUDA uses "threads" to program the many scalar processors on the GPU (240 on the FX 5800). These threads do not execute independently, like POSIX threads. Instead, they execute the instructions of a thread in lockstep. Threads can take different branches; in practice, this divides the pool of processors into several sets, one set per branch. The processors from only one set at a time will be executing instructions, while the

% time	cum. seconds	self seconds	calls	name
41.90	12.56	12.56	42605	computeKeypointDescriptor(double*, ...
40.13	24.59	12.03	82	void econvolve<float>(float*, ...
10.48	27.73	3.14	76770	prepareGrad(int)
3.50	28.78	1.05	1	detectKeypoints(double, double)
2.27	29.46	0.68	34165	computeKeypointOrientations(double*, ...
1.50	29.91	0.45	1	process(float const*, int, int)
0.20	29.97	0.06	1	extractPgm(std::istream&, VL::PgmBuffer&)
0.03	29.98	0.01	42605	insertDescriptor(std::ostream\&, ...

TABLE I
SiftPP Profiling Data

others idle. To maximize efficiency, the convolution code must minimize divergent thread behavior.

Compared to a CPU, the GPU has relatively little on-chip memory (i.e., cache), while it has much greater bandwidth to off-chip DRAM. This bandwidth is only available when "coalescing" memory reads. Threads are executed in groups of 32, called "warps". If the threads access consecutive address, and the first address is aligned to 64 bytes, then 16 memory accesses (a "half-warp" of threads) will be coalesced into a single operation. This can change the execution time of device code by over an order of magnitude, so the convolution code is carefully written to ensure the correct alignment of memory operations.

E. Testing Setup

The OpenMP code was tested on a machine with dual quad-core Intel Xeon "Gainestown" processors running at 2.66 GHz. These processors have a 8 MB L3 cache shared by all four cores, and private 256 KB L2 caches for each core.

The GPU device used for the CUDA benchmarking is an Nvidia FX 5800, which has 4GB of GDDR memory. The execution time of the CUDA code is compared to execution on an Intel E6550 "Conroe" processor running at 2.33 GHz.

III. Results

A. OpenMP

Figure 1 shows the runtime of our OpenMP-enhanced SIFT++, as measured by `time` command, running with between one and eight threads. The lines represent the original image, and three additional images obtained by scaling the original down to half, quarter, and eighth size. This is done to illustrate the homogeneity of the work done by the code, and show how speedup changes relative to problem size.

Figure 2 shows the speedup gained with the OpenMP, based on the execution times in Figure 1. An ideal linear speedup is also shown. Our results fall well below this, with a speedup just over 2x when running threads on all eight processors.

B. CUDA

Convolution on the CPU and GPU demonstrated the behavior shown in Table II on the next page. The "Compute Time" column describes how long it takes to complete a convolution with a kernel width of 15 on an 8272 x 2848 image. The "GPU" row describes execution time on a FX 5800 graphics card, and the "CPU" is an Intel E6550 running at 2.33 GHz.
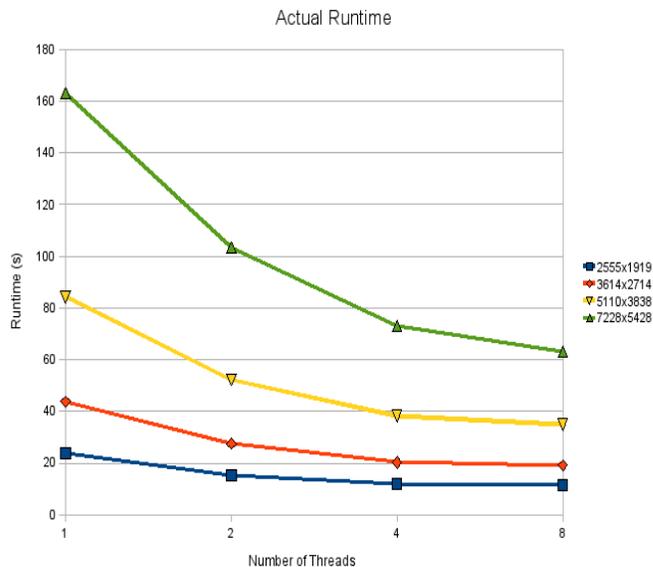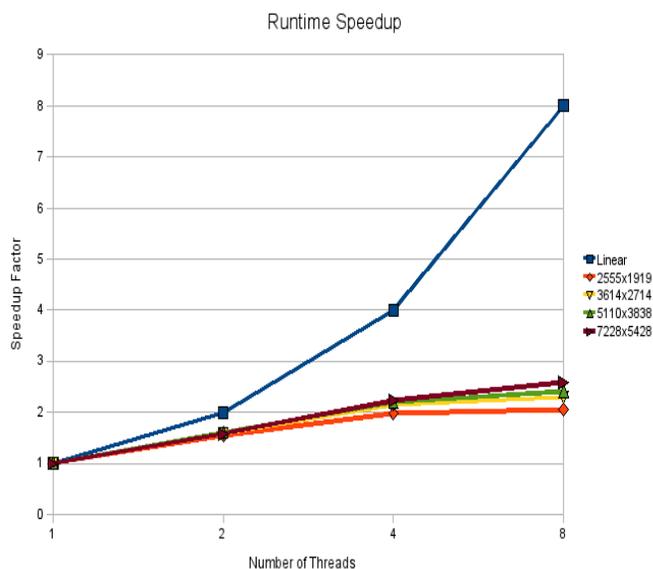

Fig. 1. SIFT++ runtime as measured by "time"

Fig. 2. Speedup of sift++ with OpenMP parallelization

	Compute Time	Comm. Time	Total Time
GPU	10.4 ms	158 ms	168 ms
CPU	2180 ms	-	2180 ms

TABLE II
CONVOLUTION ON GPU AND CPU

This result demonstrates more than just the floating-point power of the FX 5800 GPU. The theoretical (single-precision) peak performance of the FX 5800 is 933 GFlops, while a single core of the E6550 can do four floating point operations per clock cycle, for a theoretical peak performance of 9.32 GFlops. With 100x the theoretical FPU power of the CPU, the GPU achieves 200x the performance; this is due to the novel architecture and high on-device memory bandwidth of the FX 5800.

The second column describes the overhead incurred in the CUDA implementation. It includes the communication time required to transfer the input image to the card and copy the output image from the card, as well as other overhead, such as the time required to allocate memory on the GPU device before memory transfers. This overhead far exceeds the computation time, so the total execution time of the convolution is 168 ms, only 13x faster than the CPU execution time.

The entire execution time of SIFT++, including file I/O, is 33.9 seconds for the original siftpp binary, and 17.8 seconds for the accelerated version. This is a 1.9x speedup is on a 4136 by 1424 image that generates over forty thousand keypoints. The convolution time and runtime overhead of CUDA are dependent only on the size of the input image; the results shown in Table II will not vary with image content.

IV. RELATED WORK

A number of authors have previous explored accelerating SIFT. This previous work has been in a different domain than work presented in this paper: these authors have focused on real-time computer vision applications, typically processing 640x480 images as quickly as possible. At four bytes per pixel, these images occupy roughly 1.2 MB of memory and may fit entirely in the cache of a modern CPU. Processing the larger images we are using will be more affected by system memory performance.

Previous work on accelerating SIFT with a GPU typically measures performance in Hz, the number of 640x480 frames per second that SIFT feature extraction can process. Sinha, et. al. [5] built "GPU-SIFT" and benchmarked it running at 10 Hz on an GeForce 7800 GTX, roughly 10x faster than a CPU implementation. Heymann, et. al. [6] presented a GPU-accelerated SIFT implementation that achieved an approximately 5x speedup over an SSE-optimized CPU implementation using a QuadroFX 3400. These video cards, released in 2005 and 2004 respectively, were less-suited to GPGPU and were programmed without the benefit of CUDA.

Feng, Zhang, et. al. published several papers [7], [8] on implementing and measuring SIFT on SMP systems. They explore three different optimizations: OpenMP parallelization, cache optimizations, and SIMD operations (via Intel SSE instructions). In [8], they show a 6.2x speedup from parallelization on an 8-core machine; in [7] they show a 10x to 11x speedup on a 16 core machine. Their analysis shows that memory bandwidth is a determining factor of SIFT performance.

V. CONCLUSIONS

Using OpenMP, it is straightforward to accelerate existing serial code on SMP hardware. Our implementation required minimal changes to the original code in order see a speedup from parallel hardware. However, the performance of SIFT is very dependent on the target machine's memory performance, and a more careful (and time-consuming) approach is necessary to make optimal use of SMP hardware.

GPGPU can offer immense performance gains, but at the cost of programmer effort. Attempting a direct port of existing code is a suboptimal approach. The device code must be written with the architectural features mentioned above, SIMT and memory coalescence, firmly in mind. Also, because of the comparatively slow communication between the device and the host, a GPU-based applications as a whole must be written from the beginning to minimize host-device communication. If there is a relatively discrete portion of the existing code that is computationally intensive and requires little input or output from the rest of the application, then it may successfully moved to the GPU.

In SIFT++, only the convolution function matched that description, and the data transfer time to and from the device was still an order of magnitude higher than the computation time for each call to the function. Part of our planned future work is a from-scratch implementation of SIFT targeted at the GPU. By copying only the original image (or tiles thereof) into the device memory, performing most work there, and copying only the resulting keypoint descriptors back into system memory, performance gains up to 100x faster than the a single-CPU implementation are possible.

REFERENCES

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

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

[3] NVIDIA, "CUDA technology," http://http://www.nvidia.com/CUDA, 2009.

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

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

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

[7] 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. [Online]. Available: http://dblp.uni-trier.de/db/conf/iiswc/iiswc2008. html#FengLCZ08

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