Accelerating Image Feature Comparisons using CUDA on Commodity Hardware

Seth Warn, Wesley Emeneker, John Gauch, Jackson Cothren, Amy Apon

I. Background

Given multiple images of the same scene, image registration is the process of determining the correct transformation to bring the images into a common coordinate system—i.e., how the images fit together. Feature-based registration applies a transformation function to the input images before performing the correlation step. The result of that transformation, also called feature extraction, is a list of significant points in the images, and the registration process will attempt to correlate these points, rather than directly comparing the input images.

The Scale-Invariant Feature Transform [1], or SIFT, is a popular feature extraction algorithm. After finding significant points, it generates a descriptor for each point. These descriptors are a collection of circular histograms describing the intensity gradients of small regions surrounding the point. The structure of these descriptors, combined with multi-scale extraction, makes SIFT feature descriptors invariant to the rotation and scale of the input images.

Georeferencing, which locates images in physical space, is one application of image registration. The location of an input image, for example an aerial photo, can be determined by registering it against other aerial images with known coordinates. When using feature-based registration, this is implemented by extracting features from the input image, searching for similar features in the existing images, and calculating a coordinate transformation that correlates the similar features.

Searching for similar features implies the existence of a function to measure feature dissimilarity. One such function is the circular earth mover's distance [2], or CEMD, which provides excellent feature comparison but requires more computation than other methods. Georeferencing frequently operates on large images, requiring searches among billions of features, so a method to quickly perform many CEMD comparisons is prerequisite to the use of CEMD in georeferencing applications.

II. Implementation

We present an implementation of CEMD accelerated with the use of graphical processing units (GPUs). Specifically, we use Nvidia's CUDA framework [3] and MPI to accelerate CEMD calculation with 24 GPUs spread across 6 nodes. This system is benchmarked performing over 1.2 billion CEMD calculations per second. A single GPU can perform these calculations 75 times faster than an optimized CEMD implementation running on a single CPU core.

We present this work in two parts: First, we discuss the process of creating a CEMD kernel in CUDA. We present a simple initial kernel, then show a series of refinements to the kernel and discuss their efficacy. Second, we demonstrate the kernel running in streaming/asynchronous mode on a single GPU to hide communication latency, and running across all GPUs and nodes in a cluster.

Unless otherwise noted, benchmarking 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)

Note that there are two GPUs on each GTX295 card; single-GPU results use only one of these two GPUs.

A. Building the CEMD Kernel

The initial GPU implementation (GPU-INITIAL) of CEMD copies data to and from the GPU device, and one CUDA thread is launched per result calculated. The code run by each thread is similar to the CPU implementation. This version of the GPU kernel shows a 7.96× speedup compared to the CPU.

Shared memory usage is the first refinement of the CEMD kernel. CUDA has several available memory spaces. Shared memory is on-chip and has 100× lower latency than the off-chip global memory. However, shared memory is local to a small "blocks" of threads and is only 16 KB. By fetching feature descriptors into this shared memory and reusing them, this version of the kernel (GPU-SHMEM) shows a 3.62× speedup as compared to the initial GPU implementation.

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

TABLE I: MEMD 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 MEMD 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
[d] The amortized number of measurements made every second, including overhead costs

an order of magnitude faster than non-coalesced operations. We refine our second kernel to coalesce memory operations, but only find a 1.03× speedup versus the previous kernel, because the CEMD algorithm is not bound by memory bandwidth—we calculate that this kernel is using less than 1% of the available global memory bandwidth.

GPU shared memory is divided into banks. 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. Profiling shows that the CEMD kernel generates many bank conflicts. We create a kernel (GPU-INDEX) which offsets the memory reads of each thread to avoid these conflicts. Profiling shows an 80% reduction in shared memory conflicts, but this modification results in slightly slower performance.

This result indicates that the CUDA architecture is successfully hiding the shared memory latency in the previous kernel. The GPU hides these latencies through zero-overhead switching between groups of threads: while a group of threads is waiting on a memory operation, other threads continue to execute. The additional indexing required to avoid conflicts adds instruction overhead, slowing kernel execution, so this kernel refinement is discarded.

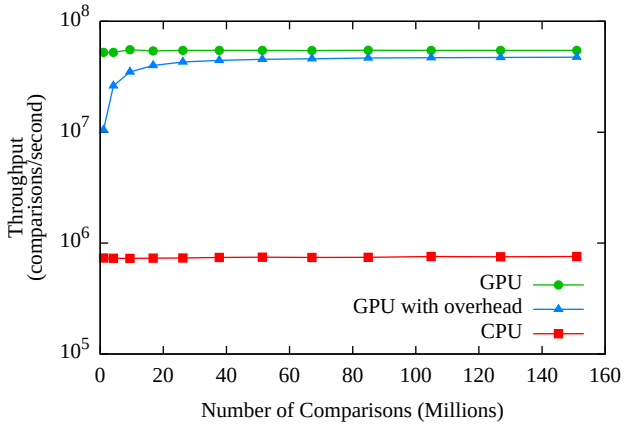

Fig. 1: Throughput of N Measurements

Eliminating instruction overhead is a potential optimization target. The simple processing cores in GPUs can benefit greatly from loop unrolling. The maximum size of a CUDA kernel is 2 million instructions, enough capacity for significant loop unrolling. We use a combination of C++ template and preprocessor metaprogramming to manually unroll the CEMD algorithm, flattening several levels of loops. This approach (GPU-UNROLL) proved very effective, producing a 5.10× speedup as compared to the previous kernel. By examining the GPU kernel assembly code, this kernel is shown to operate 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 GPU architecture is hiding most of that latency and this algorithm is unlikely to see substantial further speedups, though a final refinement—an improvement to the CEMD calculation algorithm—demonstrates a small performance gain (GPU-FINAL).

Loop unrolling and the improved algorithm can be profitably used in the CPU implementation of the MEMD kernel as well, doubling the throughput of comparison calculations. The final optimized version of the GPU kernel is 75.6× faster than the final optimized version of the CPU kernel.

B. Performance Comparison

Figure 1 shows the performance of both our final GPU kernel and the optimized CPU code. The algorithm is deterministic, and both implementations show a constant throughput regardless of the number of comparisons being performed. An additional performance curve, "GPU with overhead," is shown, which includes the cost of kernel launch, copying input data to the GPU, and output data from the GPU. For small input sizes, all three affect

the throughput, but as input size grows, only the size of the output data grows at the same rate as the amount of work done. For larger input sizes, copying the output from device to host results in a 13% communication overhead.

This overhead can be partially hidden by using multiple streams of execution to overlap asynchronous kernel launches. The GPU is capable of performing 54.7 million comparisons per second. Without overlapping streams of execution, the GPU can only sustain 47.5 million comparisons per second, due primarily to the overhead of copying the results back to the host memory. By overlapping execution and copies, a sustained rate of 51.2 million comparisons per second was achieved.

Nvidia's new "Fermi" architecture products became available after these experiments were run. Benchmarking on a single GTX 480 shows the MEMD kernel is capable of performing 114 million comparisons per second on the new Fermi-based devices.

C. Cluster Implementation

The cluster application distributes the comparison of an input set of features to a larger set of known features. Each node (represented by a single MPI process) is responsible for comparing the complete set of input features to a subset of the known features. The nodes use OpenMP to create a management thread for each GPU. The comparisons to be performed by the node are divided among the GPUs, then further divided into a number of kernel launches, and 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 6 nodes, the test cluster performed over 1.2 billion comparisons per second.

References

[1] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[2] J. Rabin, J. Delon, and Y. Gousseau, "Circular Earth Movers Distance for the Comparison of Local Features," in *International Conference on Pattern Recognition*, Dec. 2008.

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