GPU Performance Assessment with the HPEC Challenge

Andrew Kerr, Dan Campbell, Mark Richards
Georgia Institute of Technology, Georgia Tech Research Institute
{andrew.kerr, dan.campbell}@gtri.gatech.edu, mark.richards@ece.gatech.edu

Introduction

Commodity graphics processing units (GPUs) are highly parallel programmable microprocessors. The current high end GPUs offer a peak performance of 500 GFLOP/s in single precision. Previous methods of performing general-purpose computation on GPUs required algorithm implementations be cast as 3D graphics operations with prohibitive limitations on programmable vertex and pixel shader length, control flow, and arthmetic capabilities. In 2006, NVIDIA released the Compute Unified Device Architecture (CUDA) [1] development platform for the GeForce 8 series of GPUs. CUDA specifies extensions to the C programming language for writing program "kernels" directly targeting GPUs. This enhances the viability of GPUs as a general-purpose computing solution by providing a straightforward programming model and language exposing the parallel data paths of the GeForce 8. Achieving high performance requires careful consideration of GPU architectures and rigorous optimization efforts. Efficient implementations are nontrivial, and certain classes of algorithms are better suited to GPU architectures than others. Our implementation demonstrates the performance characteristics of GPUs for various kernel-level benchmarks selected from the High Performance Embedded Computing Challenge [2]. We will discuss the suitability of each benchmark to GPU architectures and describe optimization techniques that result in maximum speedup.

Benchmarks and Performance Metrics

The HPEC Challenge defines nine kernel-level benchmarks consisting of fundamental matrix operations. We have implemented time-domain FIR filter bank, frequency-domain FIR filter bank, QR factorization, constant false-alarm rate detection, pattern matching, graph optimization via genetic algorithms, and corner turn.

The HPEC Challenge specifies several performance metrics. Latency, $L_1(k,d_i)$, is the total time required to perform one kernel k for data set size d_i. The HPEC Challenge specifies that input data sets should reside initially in system memory, and that latency measurements should include time required to transfer data to coprocessor memory spaces such as the GPU's global memory. Because GPUs have enough on-board memory to allow many real-world applications to perform several operations consecutively before transmitting final results to system memory, we also provide a strict kernel latency measurement denoted $L'_1(k,d_i)$ that excludes time spent transferring data between GPU and system memory. The HPEC Challenge defines asymptotic throughput $T(k,d_i)$ in terms of a predefined workload associated with each kernel and the latency $L_1(k,d_i)$. Additional metrics derived from latency and throughput include efficiency with respect to peak theoretical performance, performance stability, and performance per unit power. Details are given in [2]. We will present metrics for each kernel benchmark implemented and contrast these results with performance of the HPEC Challenge reference implementation [3]. Results of our implementation of HPEC Challenge demonstrate GPUs are a suitable class of architectures for applications requiring high performance and energy efficiency.

Optimization for GPUs

The NVIDIA GeForce 8800 GTX possesses sixteen multiprocessors. Each multiprocessor is structured as a SIMD processor with eight data paths. To achieve maximum performance, all data paths must perform computation while at the same time maximizing memory data transfer rates. This is accomplished with careful parallelization with regard to concurrency, interthread communication, and efficient memory access patterns. Because GPUs lack a method for multiprocessors to exchange data or synchronization messages within kernels, multiple kernels must be invoked. This forces a flush of "shared memory" and limits throughput of algorithms that depend on fine-grain synchronization among concurrent blocks. Though the GeForce 8800 lacks a large cache, it interfaces global memory with a 384-bit bus capable of bandwidth up to 86 GB/s. Reads and writes are coalesced among adjacent threads accessing consecutive locations. Latency is also high, however, requiring several hundred cycles to satisfy a request in the absence of other traffic. Many concurrent threads may effectively hide memory latency by performing computation while others wait for memory transactions to complete. This may be accomplished by structuring algorithms to employ data pipelines as in the time-domain FIR implementation. Skewed storage of vectors in shared memory avoids bank conflicts between adjacent threads. Performing serial operations in blocks promotes coarse-grain synchronization as demonstrated in the QR factorization benchmark.

Performance Results

The latencies and speedup of selected benchmarks implemented appear in Table 1. Speedup is expressed in terms of the strict kernel latency metric $L'_1(k,d_i)$ for the GPU implementation. For the CPU reference implementation, the data is already in the address space of the CPU. Benchmarks

were run on a 2.4 GHz Intel Core2 Q6600 with an NVIDIA GeForce 8800 GTX. The OS is Windows XP Professional.

Table 1: Performance results in milliseconds

Benchmark	Data set	$\mathbf{L}_1(k,d_i)$	$\mathbf{L}'_1(k,d_i)$	Speedup
Corner Turn	Set 1	2.49	0.30	8.32
	Set 2	28.2	4.60	11.4
TD FIR	Set 1	3.92	2.54	151
	Set 2	0.76	0.09	22.2
FD FIR	Set 1	9.02	3.25	19.7
	Set 2	2.0	0.26	11.5
Pattern Matching	Set 1	2.00	0.24	12.7
	Set 2	3.99	1.65	23.1
CFAR	Set 1	2.43	0.29	2.3
	Set 2	56.6	3.5	166
	Set 3	19.1	3.4	46.8
	Set 4	10.5	2.7	25.6
Genetic Algorithms	Set 1	2.12	0.5	15.6
	Set 2	13.6	11.7	33.3
	Set 3	2.65	1.0	21.9
	Set 4	6.4	4.1	23.7
QR	Set 1	23.5	20.3	4.6
	Set 2	6.54	4.5	1.5
	Set 3	3.91	1.8	5.6

Performance Analysis

Several benchmarks have straightforward parallelization schemes that result in high speedup without significant optimization effort. The time-domain FIR filtering benchmark assigns one block per filter bank and performs convolution with an unrolled looping structure and data pipeline through shared memory. CFAR is embarrassingly parallel and specifies a large data cube. Each doppler bin may be processed independently of the others, and overall runtime is bounded by global memory bandwidth.

The graph optimization benchmark simulates the search of a problem space using a genetic algorithm. Candidate solutions are coded as "chromosomes" consisting of a tuple of "genes" and evaluated based on a table look up. For each iteration, selection, crossover, and mutation are performed. While each of these steps requires the population be initially held in global memory, each procedure is highly parallel and independent. Each generation may be performed as a batched sequence of kernel calls. The relatively small number of distinct code words for all genes of a data set permits the lookup table be held entirely in shared memory during the evaluation step. Due to inherent coarse-grain parallelism, genetic algorithms are favorable to GPUs and exhibit high speedup.

The QR factorization benchmark specifes the Fast Givens algorithm defined in [4] for factoring a matrix A such that $A = QR$, where Q is orthogonal and R is upper triangular. The benchmark was implemented for complex-valued matrices of up to 512 rows. To achieve coarse-grain synchronization, a set of Givens rotation matrices is computed for consecutive rows along a set of adjacent "active" columns. This is performed by a kernel of one block with only as many threads as the number of full columns that can fit into shared memory. Once all threads have computed Givens rotations and applied them across the columns of this block, the matrices are written to global memory and the kernel terminated. A second kernel is launched with many blocks and applies these rotations to all of the columns of the matrix. Additionally, Q is transformed. This sequence is repeated until all of the input matrix has been triangularized.

While Fast Givens is well-suited to CPU architectures, it is not necessarily the best algorithm for performing QR factorization on GPUs. Fast Givens reduces the number of square roots performed by constructing a diagonal matrix of squares that delays the normalization of the Givens rotations. After R is triangularized, the inverse square root of this diagonal matrix is computed and used to multiply Q and R. On a CPU with a large cache, the diagonal matrix is likely to remain in cache permitting fast accesses. Because square root is a computationally intensive operation, this method is preferable to the Givens method of QR factorization for CPU implementations if accuracy requirements are relaxed. GPUs, however, are capable of performing many square root calculations in parallel with relatively low latency. Avoiding updates to the diagonal matrix and computing square roots for each Givens rotation is faster during the triangularization step and avoids a large matrix multiply afterward. QR factorization with Givens rotations is likely to be faster than the Fast Givens algorithm on GPUs. Moreover, the Givens rotation method is typically more accurate than Fast Givens implemented for the same architecture and should be selected for future GPU implementations if the implementer is free to choose the underlying algorithm.

Conclusion

GPUs perform well when applications can be partitioned into disjoint blocks of threads that do not require fine-grain communication or synchronization. The wide SIMD nature of each multiprocessor successfully achieves high performance without hindering any class of applications. The fast shared memory in each multiprocessor is of sufficient capacity for the kernel benchmarks covered. Additional memory in future architectures will improve scalability of certain implementations. Lack of a high-speed interconnect between multiprocessors does limit throughput for certain classes of algorithms, but this latency may be effectively hidden with a large number of threads and by batching kernel invocations.

References

[1] *NVIDIA CUDA Programming Guide 1.1*, `http://www.nvidia.com/object/cuda_get.html`

[2] *HPEC Challenge*, `http://www.ll.mit.edu/HPECchallenge`

[3] *HPEC Challenge Reference Implementation*, `http://www.ll.mit.edu/HPECchallenge/software.html`

[4] Golub and Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.