IN23C-1098

Computing the Delta-Eddington Approximation for Solar Radiation

Performance and Programmability on GPUs, FPGAs, and Microprocessors

Rory Kelly (rory@ucar.edu) and Jose Garcia (jgarcia@ucar.edu)
Computational & Information Systems Laboratory, National Center for Atmospheric Research

Introduction

The Delta-Eddington Approximation is used extensively in the NCAR Community Atmosphere Model (CAM) for calculating the effects of solar radiation on the atmosphere. The routine studied here, RADDEDMX, was taken from NCAR's CAM 3.0. It is a computationally expensive portion of the CAM model, and the embarrassingly parallel nature of the computation makes it a good candidate for acceleration on a GPU.

Problem Description

For the test case considered here the atmosphere is divided into 2000 blocks, with each block containing 16 columns, and each column having 27 vertical layers. The RADDEDMX routine is called independently on each layer of each column. Every subroutine call requires 19 floating point values as input, calculates 10 floating point values as output, and performs 419 floating point operations on average.

NVIDIA GPU Hardware

The graphics card used for this experiment was an NVIDIA GeForce 9800 GX2. The 9800 GX2 contains two G92 graphics processors, each with 128 individual scalar processor (SP) cores and 512 MB of fast DDR3 memory. The SPs are clocked at 1.5 GHz, and each can perform a fused multiply-add every clock cycle, which gives the card a theoretical peak performance of 768 GFlop/s. Data is moved to and from the host CPU memory by DMA transfers over a PCI Express bus. Although PCIe 2.0 data rates are supported by the GeForce 9800 GX2 card, the host machine only supported PCIe 1.0, limiting data bandwidths to 4.0 GB/s peak between the host CPU and the GPU memory. Internal bandwidth within the 9800 GX2 is much higher, with 64 GB/s between main global device memory and the SP cores, making the PCIe bus the major bottleneck.

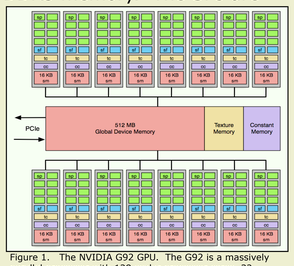

Figure 1. The NVIDIA G92 GPU. The G92 is a massively parallel processor with 128 scalar processor cores, 32 special function units, and multiple levels of global and local memory.

Implementation

The original subroutine was a Fortran module from CAM. The module was ported to C and a driver routine was created to call it using data from a sample run. The CPU version stores data in 2D arrays by vertical level and by column index, with 27 levels and 32000 columns (2000 blocks) in the test case. It iterates over levels, and at each level calls RADDEDMX to compute values in that level for each column. The CUDA version uses 1D arrays with data from adjacent columns contiguous in memory on each vertical level. The arrays are DMAed to the GPU, the CUDA kernel is called once, and results are DMAed back to the CPU. The arrays are discretized using CUDA threads, with one thread assigned to each level of each column (i.e. 27 x 32000 threads total). The threads are dispatched in 1D blocks, the number of threads per block is a tunable parameter and the total number of blocks is automatically adjusted to fit the problem size. This data arrangement causes neighboring threads to access contiguous memory locations, which allows coalesced memory accesses to maximize GPU memory efficiency. In addition local and temporary variables use registers to reduce traffic to and from global device memory. The CUDA code for the kernel itself is very similar to the original C, with just a few function calls traded for faster GPU intrinsics.

column 0 level 0	column 1 level 0	...	column N-1 level 0	column 0 level 1	column 1 level 1	...	column N-1 level 1	...	column N-1 level 27

Figure 2. Layout of variables in GPU memory for the CUDA version. All data arrays are 1D and contain data for adjacent columns in contiguous memory locations for each vertical level. In the present case N=32000. This arrangement guarantees memory coalescing on the GPU.

Performance

To evaluate performance the test case was run on the NVIDIA GeForce 9800 GX2 as well as some common microprocessor platforms. Results are shown in figure 3. The RADDEDMX code on the GPU is able to process columns 14x faster then the fastest microprocessor, achieving 967,000 columns/sec, vs 69,000 columns/sec for the Intel Core2. Although this may seem like great performance, in reality it is only a small fraction of the computing power available from the GPU. Because the computational intensity of the problem is low (only 5.5 Flops/byte loaded), performance is dominated by the data transfer bandwidth over the PCIe bus. The host system only supported PCIe 1.0 speeds, and therefore data transfer was limited to 4.0 GB/s. When PCIe protocol overhead and memory system effects are accounted for, in practice 3.0 GB.s - 3.5 GB/s is a more typical DMA transfer rate over the PCIe bus, depending on the transfer size. For this problem we are making a relatively small transfer (~63 MB) which takes 20.1 msec, giving a bandwidth of ~ 3.1 GB/s, showing that we are making good use of the limited bandwidth available. To show how limited the overall problem is by the PCIe bus, we can calculate the maximum compute rate possible with this bandwidth. In the limit of free computing, the data transfer alone takes 31.1 msec, giving a column processing rate of 1,029,000 columns / sec. Our implementation achieved 967,000 columns / sec, or roughly 94% of the maximum possible performance given the bandwidth constraint. A comparison of results between the test systems is shown in table 1. Ignoring the cost of the DMA, the GPU is able to make better use of its floating point units than the microprocessors, due to a greater ability to hide latency by using large numbers of very light weight threads.

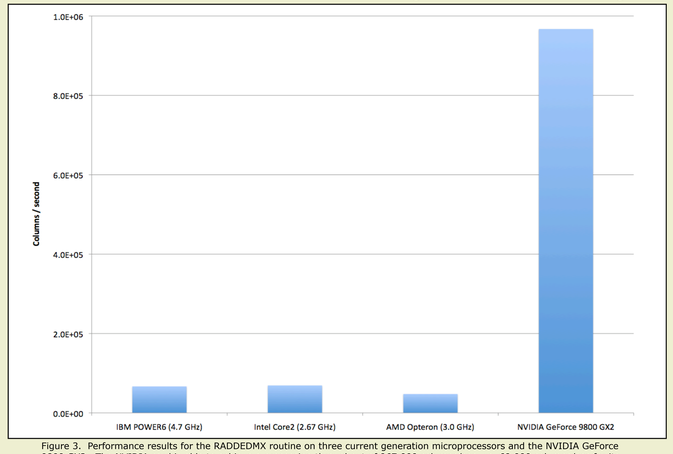

Figure 3. Performance results for the RADDEDMX routine on three current generation microprocessors and the NVIDIA GeForce 9800 GX2. The NVIDIA card is able to achieve a processing throughput of 967,000 columns/sec, vs. 69,000 columns/sec for its closest competitor, the Intel Core2. This represents a speedup of 14x - 20x for the subroutine, which includes time taken to transfer data between the CPU and GPU main memories.

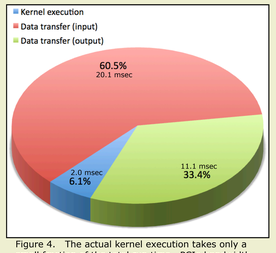

Figure 4. The actual kernel execution takes only a small fraction of the total run time. PCIe bandwidth is the major performance limiter for this application.

	IBM POWER6	Intel Core2	AMD Opteron	NVIDIA GPU GPU+DMA	NVIDIA GPU GPU Only
Time (msec)	479.22	462.64	672.25	33.08	2.02
Rate (MFlop/s)	754.92	781.97	538.15	10,934.95	179,095.05
Rate (column/s)	66,775.46	69,168.10	47,601.55	967,234.92	15,841,584.16
Percent of Peak	4.02	7.32	8.97	1.42	46.64

Table 1. Results comparison across the test systems. Note that when looking only at compute time and ignoring DMA transfer time, the GPU achieves a large percentage of its peak performance relative to the other platforms.

Conclusions

Bandwidth limitations turned out to be the major performance bottleneck for this code, although, given its relatively low computational intensity, such a result was not unexpected. The upside to being bandwidth limited is that we can achieve a large increase in overall application performance simply by increasing the bandwidth between the GPU and the host system, without having to make any changes to the code. In particular, since PCIe 2.0 is supported on the NVIDIA GeForce 9800 GX2, we should be able to plug the card into a machine with PCIe 2.0 support and get nearly a factor of two improvement in overall performance, provided that the system has a fast enough memory system to saturate a PCIe 2.0 bus.

Another interesting consequence of implementing a routine with low computational intensity for the 9800 GX2 card is that the best performing implementation turned out to only use one of the two G92 chips available on the card. The reason for this is that using both G92s could halve the computation time from roughly 2 msec, to roughly 1 msec, but the lower DMA bandwidth resulting from performing two DMAs of half the size (one to each chip), as well as the additional control overhead of scheduling computations on both chips, added significantly more than 1 msec to the run time. One of the biggest lessons of this exercise is: when looking at using GPUs and other accelerators to speed up an application, it is critical to consider the *system* as a whole, including data transfer overhead, in order understand what sort of performance is possible for a given application.

In the end we were able to demonstrate an efficient CUDA implementation of the RADDEDMX routine which, despite bandwidth constraints, still achieved speedups in excess of 10x over current generation microprocessors.

FPGA Footnote

Originally we had intended to include some promising results from a research project at Xilinx to create compilers that produced pipelined FPGA code from standard C and Fortran source. Although in relatively early stages, the CHiMPS project was able to take our C code and produce an FPGA executable which, on a simulator, produced speedups of order 10x, with little change to the source code. Unfortunately the project is no longer being actively pursued at Xilinx, although we hope it might be revived at some point in the future.

Acknowledgements

This research was supported by the NSF and made possible by an equipment grant from the NVIDIA corporation through the NVIDIA Professor Partnership Program. Special thanks also to Jeff Mason from Xilinx for supplying data on FPGA performance.

For Further Information

Please Contact:
Rory Kelly (rory@ucar.edu)
Jose Garcia (jgarcia@ucar.edu)

More information about this project and related projects can be found at:
http://www.cisl.ucar.edu/css/

NCAR NVIDIA NSF