ACCELERATING BEAM DYNAMICS SIMULATIONS WITH GPUs*

I.V. Pogorelov†, K. Amyx, P. Messmer, Tech-X Corporation, Boulder, CO 80303, USA

Abstract

We present recent results of prototyping general-purpose particle tracking on GPUs and discuss our CUDA implementation of transfer maps for single-particle dynamics and collective effects. The objective of this work being incorporation of the GPU-accelerated tracking into ANL's accelerator code ELEGANT [1], we used the code's quadrupole and drift-with-LSC elements as test cases, achieving 80x and 36x speedups over CPU implementations, respectively. We discuss quadrupole kernel optimizations, as well as data-parallel and hardware-assisted approaches to avoiding thread contention at the charge deposition stage of algorithms for modeling collective effects.

GENERAL PURPOSE COMPUTING ON GPUS

In recent years, general purpose computing on graphics processing units (GPUs) has attracted significant interest from the scientific computing community because these devices offer a large amount of computing power at very low cost. Unlike general purpose processors, which are designed to address a variety of tasks ranging from control flow and bit manipulation operations to floating-point operations, GPUs are optimized to perform floating-point operations on large data sets. Instead of allocating a large amount of the on-chip real estate for large cache memory and control flow logic, GPUs dedicate a lot of resources to floating-point units. A GPU like the one found in the NVIDIA Tesla C2050 series consists of 15 vector processors with a vector length of 32 elements. Using predicated execution enables each vector element to execute its own flow through the program, providing the impression of 448 independent execution units.

The introduction of the Compute Unified Device Architecture (CUDA) by NVIDIA has made it possible for computational scientists without a deep knowledge of graphics-oriented programming interfaces like OpenGL to take advantage of the high processing power offered by GPUs.

CUDA enables users to develop algorithms in C++ with a small set of language extensions. The developers write so-called *kernels*, code that executes on the GPU defining the behavior of a single thread of execution. Typically, a kernel is executed by thousands of threads concurrently and the GPU's thread manager maps them to the physical thread processors. The kernel is invoked on the host side, at which time it is determined how many threads will be executed. Memory management, data transfer and kernel invocations are all controlled by the host CPU. A special compiler, *nvcc*, translates kernels and host programs into code that executes on the CPU and on the GPU. This architecture simplifies significantly the software development process for CUDA-enabled GPUs, but it still requires a detailed knowledge of the GPU's architecture in order to obtain good performance. For example, while the threads can be treated independently of each other, they are in fact executed on a Single-Instruction-Multiple-Data (SIMD) type architecture. Thus, on a C2050 card, 32 threads are executed using the same instruction stream, which means that diverging threads can lead to a large amount of stalled threads and result in performance degradation. Also, one of the benefits of GPUs is their large memory bandwidth, but in order to take advantage of it, memory access of different threads has to be carefully aligned. Finally, many inherently sequential algorithms, such as cumulative sums of a vector, are straightforward to implement on a serial processor. In contrast, optimization on a massively parallel system like GPUs requires carefully crafted routines.

GOALS OF THE CURRENT WORK

Our primary goal is to provide a set of fast particle tracking kernels for ELEGANT. ELEGANT is an open-source, multi-platform code used for design, simulation, and optimization of FEL driver linacs, ERLs, and storage rings [1, 2]. The parallel version, PELEGANT [3], uses MPI for parallelization and shares all source code with the serial version.

Several new ”direct” methods of simultaneously optimizing the dynamic and momentum aperture of storage ring lattices have recently been developed at Argonne [4]. These powerful new methods typically require various forms of tracking the distribution for over a thousand turns, and so can benefit significantly from faster tracking capabilities. Because the ability to create fully scripted simulations is essential in this approach, ELEGANT is used for these optimization computations.

ELEGANT is fundamentally a lumped-element particle accelerator tracking code utilizing 6D phase space, and is written entirely in C. A variety of numerical techniques are used for particle propagation, including transport matrices (up to third order), symplectic integration, and adaptive numerical integration. Collective effects are also available, including CSR, wakefields, and resonant impedances [1, 2]. Recently, we prototyped key ELEGANT particle tracking algorithms on NVIDIA GPUs and demonstrated that such accelerated implementations can be incorporated into ELEGANT. To achieve this goal, we

* Work supported by the DOE/BES Grant No. DE-SC0004585 and by Tech-X Corp.

† ilya@txcorp.com

focused on one element described by a transfer map (a quadrupole), and one collective-effect element (a drift with 1D longitudinal space charge). Our longer-term goal is to expand the kernel library to include optimized implementation on GPUs of most of the ELEGANT elements, starting with the most time consuming ELEGANT kernels.

INITIAL RESULTS

Prototype Kernels for Single-Particle-Dynamics in ELEGANT

As a first step toward enabling particle tracking with ELEGANT on GPUs, we implemented in CUDA the 2nd order map for the quadrupole beamline element (QUAD in ELEGANT notation). We implemented algorithms in both single and double precision, with an emphasis on optimizing the kernels for the NVIDIA "Fermi" GPU architecture. In our implementation, we stored particles in linear memory, and investigated schemes in which one particle is computed per thread. We utilize the high-bandwidth and low-latency constant memory cache to access the map parameters used to update the particle information: constant memory gives register-time access for cache hits when all threads of a warp access the same value, as happens in a quadrupole map computation. Alternatively, on Fermi-capable devices the LDU (LoaD Uniform) instruction may be employed to accelerate reads from global memory that do not depend on thread index. In our highest-performing prototype kernels (see below), we observed a performance degradation of roughly 12 percent when employing the LDU instruction instead of explicitly utilizing constant memory.

For such a kernel with a high density of floating point computations, it is extremely important to ensure that all per-thread data is stored not as local memory (on the L1 cache of a GPU multiprocessor) or shared memory but in registers. By default, a small per-thread array is mapped to cached memory when the compiler cannot determine all array indexing. There are two options to force the kernels to use registers: either by manually unrolling all loops and using scalar variables (for a second-order quadrupole map, this corresponds to 6 scalars and 258 computations), or by proper loop ordering with the **#pragma unroll** compiler directive. Inspection of the PTX assembly code verifies the type of memory used for each thread: kernels that use local memory instead of registers will employ the **ld.local** and **st.local** mnemonics.

For testing purposes, we generated a realistic 6D phase space distribution function that was propagated through a lattice consisting of a small number (~ 20) of quadrupoles and drifts (without space charge). In a simulation with 20 million double-precision particles with kernels that utilize constant memory to access map elements and registers for per-thread data, we observed a 80x speedup on a C2070 Fermi GPU compared to a single core Intel Xeon X5650 @ 2.67GHz CPU, with a comparable speedup seen when traversing a single quadrupole. This corresponds to 200 GFLOPS, out of a peak theoretical throughput of 500 GFLOPS for a Tesla C2070. Kernels utilizing local memory and shared memory achieved 90 and 115 GFLOPS, respectively.

In addition to developing kernels for other beamline elements, we plan to explore additional efficient ways of accessing the map parameters. The latter becomes more important in simulations with higher order maps, as the number of Taylor series coefficients that describe the map goes up.

Prototype Kernels for Collective Effects in ELEGANT

Investigation into kernels for collective effects elements in ELEGANT focused on the LSCDRIFT (drift with longitudinal space charge). The core computation steps of the LSCDRIFT are as follows:

1. The particle temporal coordinates must be calculated from the particle spatial coordinates and momenta
2. The maximum and minimum temporal values must be found
3. A histogram is generated from all particles' temporal coordinates
4. The maximum of the binned charge is computed
5. An FFT is performed on the binned charge
6. The RMS of the particle coordinates is computed
7. Low- and High-pass filters are applied to a rescaled-FFT of current, which is then multiplied by impedance
8. An inverse FFT is performed to compute the voltage kick
9. The voltage kick is interpolated to each particle
10. All particle coordinates are updated based on the interpolated voltage kick

The most difficult component of LSCDRIFT to port to a GPU is the charge binning (step 3), in which a histogram with 200-500 bins is computed based on a large number of particles. This is challenging to implement on a GPU due to contention between individual threads attempting to update the same location in memory simultaneously. We have investigated multiple approaches to avoiding such race conditions.

The data-parallel approach to implementing the charge binning utilizes the Thrust template library's **thrust::sort** function to sort an array of particle bin indices. A call to the **thrust::lower_bounds** function is applied to calculate the number of particles per charge bin **i** by subtracting **lower_bounds[i]** from **lower_bounds[i+1]**. The maximum of the binned charge histogram is then computed with a secondary kernel that computes the maximum via a reduction in shared memory of a single block. This solution achieves an acceleration 5.3x for 20 million particles and 256 charge bins.

A second approach involves utilizing the Fermi architectures ability to perform thread-safe atomic memory updates in the L1 cache of each multiprocessor. Calling **atomicAdd()** to a global array for every particle results in terrible performance (ten times slower than a CPU implementation); however, atomic updates to per-thread-block copies of bin arrays are quite fast because the per-bin arrays can be kept on the L1 cache of each multiprocessor. This kernel then utilizes the **__threadfence()** instruction to enforce a degree of block-ordering to combine the results of all blocks and perform a reduction in shared memory to calculate the maximum binned value. This approach achieves an acceleration of 8.7x.

The data-parallel approach relies on a global $NlogN$ sort, and the cached atomic update approach relies on the hardware to resolve concurrent memory accesses. As an alternative approach, we attempted to take advantage of the hierarchical memory architecture of the GPU by computing sub-histograms in shared memory. This is done by performing a per-block bitonic sort in shared memory, and then performing a per-block segmented prefix sum. The sort and scan operations allow a thread-safe stream compaction in shared memory to calculate a per-block sub-histogram without relying on either hardware methods or global sort operations. As in the cached atomic kernel, a **__threadfence()** reduction is used to combine the results of all blocks and calculate the maximum binned value. Our implementation for this kernel achieved a modest 3.2x acceleration. We attribute this to low occupancy as a result of significant shared memory usage, as well as the inherent cost of the sorting and scanning algorithms.

High-performance kernels were implemented for the remaining operations in the LSCDRIFT computation. The RMS calculation kernel was redesigned to maximize use of shared memory reductions. In some cases, separate operations were combined into a single kernel to reduce memory traffic on the device. This was done for the following cases:

1. The time coordinates computation kernel (step 1) was augmented to also perform shared memory reductions to calculate the minimum and maximum values per block. A second kernel performs the reduction of these per-block max and min values.
2. As mentioned, the maximum binned value is computed inside the cached atomic histogram kernel via a **__threadfence()** reduction.
3. The voltage kick interpolation and final particle update kernels were combined

The entirety of the GPU-accelerated LSCDRIFT algorithm achieves a 36x speedup relative to a CPU implementation in double precision for 20 million particles and 256 charge bins.

The LSCDRIFT computation on the CPU spent 13% of the total time computing time coordinates and min/max; 6% on binning time coordinates and max; 9% on calculating RMS measures of the distribution; 1% on forward FFT, voltage calculation, and inverse FFT; and 71% of total time on applying voltage kicks to particles. The breakdown was very different for the GPU-accelerated LSCDRIFT: 11% of total time compute time coordinates and min/max; 32% binning time coordinates and max; 7% calculating RMS measures of the distribution; 2% on forward FFT, voltage calculation, and inverse FFT; and 48% on applying voltage kicks to particles.

For a Tesla C2070 compared to Xeon 5650, accelerations for these computations was 42x for computing time coordinates and min/max; 8x for binning time coordinates and max; 45x for calculation of RMS measures of the distribution; $<$ 1x for forward FFT, voltage calculation, and inverse FFT; and 51x for applying voltage kicks to particles.

ONGOING DEVELOPMENT

Future development will focus adding full parallel multiple-GPU support to ELEGANT, supporting arbitrary number of particles in a simulation (rather than being limited by GPU memory), supporting GPU-acceleration for additional types of both collective and single-particle elements, and offering transparent and backwards-compatible support for GPU acceleration without requiring users to modify their simulations.

We will also further optimize existing implementations. Examples of such optimizations include: taking advantage of higher-order quadrupole map symmetries to reduce memory accesses and maximize cache use, allowing a kernel to perform multiple elements' map computations in a single call to minimize memory traffic, and redesigning the CPU-to-GPU particle data structure conversion to utilize GPU-accelerated matrix transposes to minimize CPU-to-GPU transfer time.

ACKNOWLEDGEMENTS

This work is funded by the DOE/BES Grant No. DE-SC0004585 and Tech-X Corp. Michael Borland and Yusong Wang have offered us valuable advice on various aspects of ELEGANT.

REFERENCES

[1] M. Borland, "elegant: A Flexible SDDS-compliant Code for Accelerator Simulation", APS LS-287, September 2000.

[2] M. Borland, V. Sajaev, H. Shang, R. Soliday, Y. Wang, A. Xiao, W. Guo, "Recent Progress and Plans for the Code ELEGANT," in Proceedings of 2009 International Computational Accelerator Physics conference, San Francisco, CA, WE3IOpk02 (2009).

[3] Y. Wang, M. Borland. "Implementation and Performance of Parallelized Elegant", in Proceedings of PAC07, THPAN095 (2007).

[4] M. Borland, V. Sajaev, L. Emery, and A. Xiao, "Direct Methods of Optimization of Storage Ring Dynamic and Momentum Aperture", in Proceedings of PAC09, TH6PFP062 (2009).