BEAM DYNAMICS SIMULATIONS WITH A GPU-ACCELERATED VERSION OF ELEGANT*

K. Amyx, J. Balasalle, J. King, I.V. Pogorelov†, Tech-X Corporation, Boulder, CO, USA
M. Borland, R. Soliday, Argonne National Laboratory, Lemont, IL, USA

Abstract

Large scale beam dynamics simulations can derive significant benefit from efficient implementation of general-purpose particle tracking on GPUs. We present the latest results of our work on accelerating Argonne National Lab's accelerator simulation code ELEGANT, using CUDA-enabled GPUs. We summarize the performance of beamline elements ported to GPU, and discuss optimization techniques for some core collective effects kernels, in particular our methods of avoiding costly thread contention. We also outline briefly our testing and code validation infrastructure within ELEGANT as well as a new template meta-programming infrastructure for streamlining code development.

INTRODUCTION

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, 4], 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 [5]. 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.

In recent years, general purpose computing on graphics processing units (GPUs) has attracted significant interest from the scientific computing community because these devices offer unparalleled performance at low cost and at high performance per watt. Unlike general purpose processors, which devote significant on-chip resources to command and control, pre-fetching, caching, instruction-level parallelism, and instruction cache parallelism, GPUs devote a much larger amount of silicon to maximizing memory bandwidth and raw floating point computation power. This comes at the expense of shifting the burden towards developers and away from on-chip command and control logic, and additionally requires relatively large problems with high levels of parallelism.

Our main goals for this project are (1) to port a wide variety of beamline elements to GPUs so that ELEGANT users can take advantage of the high performance that GPUs can provide, (2) support CUDA-MPI hybrid parallelism to leverage existing GPU clusters, (3) transition ELEGANTs build system to the open-source, cross-platform CMake build system, (4) maintain silent support so that GPU-accelerated beamline elements are used when possible without user input such, and (5) provide a framework and infrastructure that reduces development time and allows developers without CUDA experience to contribute to ELEGANT.

BEAMLINE ELEMENT PERFORMANCE

In this section we present a list of the particle beamline elements fully ported to the GPU, and rough estimates of their acceleration compared to the reference CPU code, comparing an NVIDIA Tesla C2070 GPU to an Intel Xeon X5650 CPU.

Accelerated Single-Particle Dynamics Elements

QUAD and DRIFT: Quadrupole and drift elements, implemented as a transport matrix, up to 3^{rd} and 2^{nd} order, respectively: nearly 100x acceleration, achieving particle data bandwidth of 80 gb/s and over 200 GFLOPS in double precision.

CSBEND: A canonical kick sector dipole magnet with exact Hamiltonian (computationally intensive): Nearly 90x acceleration due to its high arithmetic intensity.

KQUAD, KSEXT, MULT: A canonical kick quadrupole, sextupole, and multipole elements using 4^{th} order symplectic integration: 20x acceleration (lower arithmetic intensity and higher memory requirements than CSBEND).

EDRIFT: An exact drift element: Less than 20x acceleration (purely bandwidth bound).

RCOL: Rectangular collimator: 10x acceleration if particles are removed from simulation.

Accelerated Collective Effects Elements

LSCDRIFT: Longitudinal space charge impedance. Over 50x acceleration using optimized histogram calculation.

* Work supported by the DOE Office of Science, Office of Basic Energy Sciences grant No. DE-SC0004585, and in part by Tech-X Corporation

† ilya@txcorp.com

CSRCSBEND: A canonical kick sector dipole with coherent synchrotron radiation. Over 65x acceleration using optimized histogram calculation.

Beamline elements that are at the time of writing in debugging and final optimization stages include the RF cavity elements and WAKE-based elements.

OPTIMIZATION OF COLLECTIVE-EFFECTS KERNELS

Most of the collective effects beamline elements in ELEGANT utilize a histogram-based approach. Calculating a histogram on a GPU is very difficult because multiple threads need to update the same location in memory at the same time: this leads to thread contention issues that may cause extreme performance problems (if thread-safe atomic operations are used) or race conditions (otherwise). A baseline implementation that creates a sub-histogram in shared memory per CUDA thread block and which relies on atomic memory transactions to fill the histogram yields mediocre performance–roughly 10x of a reference CPU implementation.

However, we observe the following about any kernel that relies on atomics to shared memory: atomics to shared memory are costly, and the cost of such atomics roughly scales with the level of thread contention. The level of thread contention is mostly a product of the number of bins and the resulting distribution. Thus, creating additional sub-histograms (as memory permits) will reduce the thread contention and increase the performance, at the negligible cost of combining sub-histograms in shared memory. Our optimized kernel utilizes the Fermi GPU's reconfigurable L1/Shared memory cache to prefer shared memory size, and tries to fit as many sub-histograms as possible per thread block while maintaining high block occupancy. The number of thread blocks is limited to the block occupancy multiplied by the GPU's number of multi-processors, and a __threadfence()-based reduction combines the results of multiple thread blocks.

We achieve a performance of 23x over CPU for a wide range of number of bins (100s-1000s of bins) for millions of particles, with an effective bandwidth of 24 gb/s–the same effective bandwidth as a global sum reduction kernel.

Another costly computation that is present in wake-based collective effects elements is an array convolution that, if implemented by coding up its definition, scales as $O(N^2)$, with N representing the number of bins in a given wake file. Given that N is typically in the range from several hundred to a few thousand, the convolution's N-squared scaling indicates that we cannot afford to rely on the CPU calculation without negatively impacting performance of the simulation as a whole, even for millions of particles.

Our optimized kernel achieves a 40x acceleration by buffering sub-sections of each array in shared memory and performing O(bufferSize) computations in shared memory, thus computing part of the final result for a given array index. As the convolution is linear, each thread block then applies an atomicAdd() update to the final result.

FACILITATING DEVELOPMENT THROUGH TEMPLATE META-PROGRAMMING

CUDA, NVIDIAs programming language for GPU computing, allows developers to access the raw throughput of GPUs. Recent versions of CUDA have wide support for C++ template constructs, which is important because unlike typical CPU code, GPU kernels must be almost completely defined at compile-time: run-time polymorphism is very limited in CUDA. Furthermore, compile-time polymorphism allows additional abstraction layers that can hide the more close-to-the-metal implementation details from application developers.

We exploit compile-time polymorphism through template meta-programming with the following aims:

- Create extendable kernels to ease code maintenance and reduce programming errors
- Provide data-parallel abstractions that hide CUDA-specific implementation details
- Ease workflow by avoiding CUDA-related boilerplate such as thread and block configurations, thread index computations, conversion to or from array-of-structures vs. structure-of-arrays data formats , etc

As ELEGANT is fundamentally a particle-based code, the programming abstraction is based heavily on a GPU particle accessor class that behaves as though it were a CPU-style array. Given such an accessor, a developer need only define a functor that acts on individual particles by creating a basic class and overloading the proper operator. Any non-particle data needed by the functor–including physical constants and auxiliary arrays–is placed in the classs member variables. The developer then passes the class to a template GPU driver function and thus replaces a complicated for-loop over particles with a small functor class and a call to a GPU driver. No explicit CUDA code is ever written–only per-particle update classes that clearly encapsulate what data is needed for each particle update step.

In situations that involve reductions over particle-generated data (for example, computing minimum and maximum particle temporal coordinates), the developer simply has operator() return a value, and passes a relevant reduction functor to the template GPU driver (for example, Add(), Min(), MinMax(), MaxIndex(), etc).

This leads to clean, transparent code, and relies only on requiring developers to correctly identify variables that are updated by all particles, rather than variables that can be scoped as per-particle variables. Furthermore, because we are generally replacing individual for-loops, MPI-CUDA hybrid parallelism is maintained.

ISBN 978-3-95450-122-9

TESTING AND VALIDATION

Testing and validating the CUDA implementation to ensure accuracy of the results is an essential part of the project. This is a complicated task, given that the potential ELEGANT use cases are widely varied, and full coverage of all cases is difficult. In order to validate the code with a wide variety of use cases, runtime validation of the CUDA routines is now implemented via a preprocessor flag (-DGPU_VALIDATE=1). Compilation with GPU validation is not intended for production runs, but rather it is to be used as an aid in development and as a check for the user.

With respect to programming details, CUDA function calls are embedded within the equivalent CPU routines, such that the CUDA version is used when ELEGANT is compiled with GPU acceleration. When validation is enabled, timings are computed with the CUDA event timers where, in addition to the CUDA version of the routine, the CPU routine is also recursively called and timed. The result from both routines is then copied to the host memory and compared for accuracy. At the end of the Elegant run, aggregate timing statistics are displayed for each routine called.

This run-time testing has several advantages and disadvantages relative to unit tests. Unit tests can be quickly run just after compilation and provide simple test cases for debugging - both advantages over the run-time testing. However, run-time testing allows for full coverage of possible use cases and can be more easily integrated within the existing ELEGANT regression testing system than unit tests. The implementation of the run-time testing is vastly more straight-forward than unit testing, as all the element information is already present and does not need to be configured by the testing framework.

FUTURE WORK

Once the work on the RF cavity and wake function elements is completed, we will move on to full-scale simulations of realistic lattices (e.g., the APS Ring and LCLS beam delivery system), so as to gain a better understanding of the GPU-accelerated version's performance in the large-scale real-world simulations and inform further optimization efforts. With our template meta-programming infrastructure and our validation and testing infrastructure in place, the bulk of our future work relates to porting additional beamline elements to GPU. Once sufficient elements are ported to the GPU and validated, we plan to merge our GPU-enabled version of ELEGANT into Argonnes main ELEGANT source code, allowing users worldwide access to significantly accelerated serial and parallel versions of ELEGANT.

ACKNOWLEDGEMENTS

This work was supported by the US DOE Office of Science, Office of Basic Energy Sciences under grant number DE-SC0004585, and in part by Tech-X Corporation, Boulder, CO.

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] H. Shang, M. Borland, R. Soliday, Y. Wang, Parallel SDDS: A Scientific High-Performance I/O Interface, in Proceedings of 2009 International Computational Accelerator Physics Conference, San Francisco, CA, THPsc050 (2009)

[5] 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)