CUDA KERNEL DESIGN FOR GPU-BASED BEAM DYMANICS SIMULATIONS*

K. Amyx, J. Balasalle, I. Pogorelov†, Tech-X Corporation, Boulder, CO 80303
M. Borland, R. Soliday, Y.Wang, Argonne National Laboratory, Lemont, IL 60439

Abstract

Efficient implementation of general purpose particle tracking on GPUs can result in significant performance benefits to large scale particle tracking and tracking-based accelerator optimization simulations. We present our work on accelerating Argonne National Lab's accelerator simulation code ELEGANT [1, 2] using CUDA-enabled GPUs [3]. In particular, we provide an overview of beamline elements ported to GPUs and briefly discuss optimization techniques for efficient utilization of the device memory space, with an emphasis on register usage. We also present a novel hardware-assisted technique for calculating a large histogram, and compare this to data-parallel implementations. In addition, we provide an overview of recent work on a new build system for ELEGANT and integrating GPU-accelerated elements with the existing codebase in a manner that allows 'silent' support of GPU acceleration. We conclude the paper with results of a realistic test simulation and comments on future work related to GPU-enabled version of ELEGANT.

EXTENDING ELEGANT'S LIBRARY OF BEAMLINE ELEMENT KERNELS

ELEGANT is an open source, multiplatform, massively parallel code used for design, simulation, and optimization of a wide variety of particle accelerator systems, including free electron laser (FEL) driver linacs, energy recovery linacs, and storage rings [1, 4]. Exploration and optimization of accelerator design requires massive amounts of processing power, and for that reason a substantial increase in the throughput of simulations that are conducted with thoroughly tested, widely used computational tools could result in significant savings of time and effort needed to produce optimal designs. In particular, accelerator optimization techniques such as direct, tracking based dynamic aperture (DA) and momentum aperture (MA), recently developed at Argonne National Lab (ANL) [5], can benefit from substantial reduction in simulation time. In what follows, we describe initial results of a project to enable GPU-accelerated simulations in ELEGANT.

Elements Ported to GPU

We have ported the following elements to CUDA and achieved the following performance improvements. (Acceleration figures compare implementations on NVIDIA Tesla C2070 to the serial ELEGANT reference implementations performed on a single core of an Intel Xeon 2.67 GHz.)

QUAD and DRIF: Quadrupole and drift elements, implemented as a transport matrix, up to 3^{rd} and 2^{nd} order, respectively. The computation involves contraction of the six-component vector of particle phase space coordinates with tensors of rank 2 and 3. Optimized kernels achieve 80 gb/s of particle data bandwidth (i.e., are bandwidth bound) while exceeding 200 GFLOPs in double precision for a total acceleration of nearly 100x.

CSBEND: A canonical kick sector dipole magnet with the exact Hamiltonian. Optimized kernels achieve roughly 90x acceleration.

KQUAD, KSEXT, and MULT: A canonical kick quadrupole, canonical kick sextupole, and a canonical kick multipole element using 4^{th} order symplectic integration. Similar in implementation to CSBEND, but with less arithmetic intensity and higher memory access requirements. These kernels achieve a modest 20x acceleration.

RCOL: A rectangular collimator that checks if particles should be removed from the simulation. Lower arithmetic intensity yields a bandwidth bound kernel, for accelerations on the order of 30x.

EDRIFT: An exact drift element that tracks particles through a drift with no approximations. Extremely low arithmetic intensities yield bandwidth bound kernels with accelerations on the order of 30x.

LSCDRIFT: A drift with longitudinal space charge element based on the longitudinal space charge impedance model in [7]. This is an example of a collective-effects element, where the first step of calculation of the voltage kick for each particle is an FFT performed on a histogram of particles' longitudinal coordinates. Costly computation of a histogram (described in detail below) is overshadowed by computationally intensive single particle effects, yielding a performance of 36x.

General Purpose Algorithms

Particle Data Transposes and Inverse Transposes: GPU-based transformation kernels for converting back and forth between CPU data structures (in ELEGANT, stored in Array-of-Structures format) and GPU data structures (stored in Structure-of-Arrays format) achieve bandwidths of 90 gb/s using shared memory to coalesce both reads and writes. By comparison, a simple bandwidth test kernel that reads and writes particle data in Structure-of-Arrays for-

* 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

ISBN 978-3-95450-115-1

mat in a manner typical of other GPU-accelerated element kernels achieves 88 gb/s of bandwidth.

Stream compaction of killed particles: Serial implementations of rectangular collimator (RCOL) elements, along with a variety of other elements (e.g., CSBEND), remove particles from the simulation using a data swapping algorithm not suitable for GPUs. Stream compaction operations in GPU-accelerated elements rely on the `Thrust` template library's [6] `remove_if` and `gather` kernels to remove particles. These operations are roughly twice as costly as, for example, the rectangular collimator kernel itself, which determines if the particles should be removed. Applying the stream compaction sparingly will be critical to maintaining high performance in simulations which feature large numbers of elements that can remove particles.

Kahan Summation: ELEGANT, by default, uses the Kahan summation algorithm [8] to reduce floating point roundoff error in large summations. We have developed kernels that utilize Kahan summation along with tree-based reductions to minimize the accumulation of roundoff error. Note that, because RMS error is proportional to the square root of the number of elements summed, typical tree based GPU accelerated reductions are actually more accurate than their CPU counterparts.

Optimization Techniques

Many of the straightforward CUDA implementations discussed above were extremely prone to two similar problems: use of local memory instead of registers, and spillage of registers to both local and global memory.

Use of local memory instead of registers, especially for the 6D vectors that describe each particle, causes a very large performance hit because registers are orders of magnitude faster than local memory. The NVCC compiler will, by default, store per thread arrays in local memory (verified by inspecting the PTX assembly for instances of the `str.local` mnemonic), unless the compiler can completely determine all thread access patterns. Use of the `#pragma unroll` directive, along with manual unrolling of more complicated inner loops, was key in increasing the performance of the QUAD and DRIF kernels from 30x to 100x.

Initial implementations of the CSBEND, KQUAD, KSEXT, and MULT kernels all exhibited signs of spilling registers. Use of constant memory for parameters reduced the amount of register spillage in some cases, but the intensive calculations still spilled from register space to local memory, and even from local memory to global memory. This was alleviated by reconfiguring the Fermi Shared / L1 cache to favor L1 cache size at the expense of shared memory, via `cudaFuncSecCacheConfig`.

For other kernels, particularly those in LSCDRIFT, kernel fusion was applied to reduce memory bandwidth requirements when possible.

Histogram Calculation in the LSCDRIFT Element

The 1D longitudinal space charge calculation in the LSCDRIFT element is based on the LSC impedance model described in [7]. Schematically, the space charge kick calculation is performed by binning current distribution based on longitudinal particle positions (this is essentially the same as binning charge), performing a Fourier transform on the discretized charge distribution, multiplying the result in transform space by an analytically or numerically specified impedance, and then performing the inverse Fourier transform and interpolation to obtain the voltage of the kick experienced by particles in the original (physical) space. Computation of the beam current histogram (charge binning) is the most challenging of the calculations in LSCDRIFT to implement on a GPU.

Three alternative algorithms were implemented and tested: a data parallel version utilizing the `sort` and `lower_bounds` functions of the open source template library Thrust; a data parallel algorithm based on a per block bitonic sort and segmented scan; and a hardware assisted implementation based on per-block `atomicAdd()` operations. Surprisingly, the per-block `atomicAdd()` method yielded the best performance, because atomic operations on Fermi GPUs are capable of utilizing the L1 cache. We ensure that all atomic operations are 'on cache' by restricting each thread block to operating only on preallocated arrays, thus avoiding costly global memory accesses. A secondary kernel combines the subsequent partial histograms.

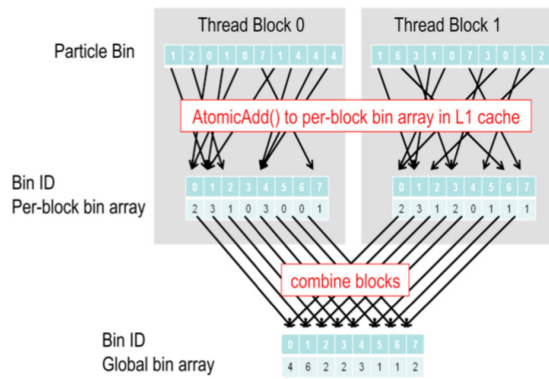

Figure 1: Charge binning based on per-block cached atomic operations

Proper choice of kernel parameters (relatively few threads per block and a relatively low number of total blocks) reduces the tendency of this kernel to serialize memory accesses. As this kernel is extremely work efficient and memory access efficient (provided the atomic updates are themselves efficient), and does not rely heavily on tree based data parallel sort or scan algorithms, this approach outperforms the other implementations, achieving an acceleration of 8.7x over the reference serial implementations. The data-parallel algorithm using the `thrust::sort` and `thrust::lower_bounds` func-

ISBN 978-3-95450-115-1

tions achieves 5.3x acceleration, and the per-warp sort and segmented scan algorithm achieves a disappointing 3.6x acceleration, comparing an NVIDIA Tesla C2070 with a single core of an Intel Xeon 2.67GHz processor.

BUILD SYSTEMS AND INTEGRATION

We have revamped the original ELEGANT build system, based on Argonne's Epics package, to use CMake. CMake is a widely used, cross-platform, open-source build system that natively supports such features as CUDA integration and streamlines finding and including external libraries, executables, and includes. In addition to building ELEGANT with CMake, we have elected to build the Epics Extensions libraries with CMake, as well, which further simplifies the overall build process of ELEGANT. It also enables fast and easy switching between compilers (such as going from serial compilers to MPI compilers), which in turn allows for hybrid MPI-CUDA parallelism schemes.

Our core philosophy for integrating GPU accelerated elements into ELEGANT is one of "do no harm": attempt to minimize impact on non-GPU ELEGANT codebase by performing conditional compilation using `#define`, and call GPU-accelerated functions instead of their equivalent CPU functions without any user intervention. That is to say, our goal is for ELEGANT to offer 'silent support' of GPU acceleration when possible, and utilize CPU implementations otherwise. In this way, a user stands to benefit from any GPU implementations of elements without needing to change any input files.

Particles are kept on the GPU whenever possible to reduce costly memory traffic between the host and the device. We accomplish this by registering GPU enabled elements in a globally accessible function, and looking ahead after every element is computed to check if the next element is also GPU accelerated. If not, we perform a high-performance transpose from GPU data structures to CPU data structures on the device and copy the particle data back to the CPU. This approach allows for piecemeal porting of elements to GPUs, as whenever particles encounter an element that is not on the GPU, they can simply be transferred back to the CPU. Of course, per Amdahl's law, the performance of any simulation will be better if all elements in a simulation are GPU enabled.

TEST SIMULATIONS

We have successfully run two GPU-accelerated test simulations in ELEGANT, tracking 10 million particles through a beamline ten times sequentially. For a beamline consisting of (in order) DRIF, DRIF, KSEXT, KSEXT, DRIF, and KQUAD, the reference serial CPU implementation takes 362 seconds to complete, whereas the equivalent GPU implementation takes 68 seconds. In both cases, approximately 49 seconds of run time is actually dedicated to file I/O. Discounting the I/O time (e.g., assuming a larger beamline), this corresponds to a net reduction in runtime of the computational portion of ELEGANT by a factor of 16x. An alternative simulation involving DRIF, DRIF, KSEXT, KSEXT, DRIF, DRIF, KQUAD, CSBEND yields a reduction in the computation portion of the program runtime by a factor of 27x, from 819 seconds to 77 seconds, where once again 49 seconds was dedicated to file I/O.

CONCLUSION

GPU-accelerated implementations of various beamline elements in ELEGANT achieve between 10- and 100-times speed-up compared to reference serial implementations. Refactoring the ELEGANT build system and integrating element kernel implementations into ELEGANT allows for 'silent' support of these GPU-accelerated elements for users of ELEGANT: no changes need be made to input files. A short but realistic GPU-accelerated simulation was run that indicated a reduction in the computation portion of ELEGANT by a factor of 16x to 27x compared to a serial version, running on a single Tesla C2070. Longer simulations, or simulations that utilize mostly beamline elements with better on-GPU performance (such as CSBEND or QUAD), stand to show substantially greater performance improvements.

In the near future our work will be focused in the following areas: adding additional GPU accelerated elements, further optimizing existing elements, ensuring that the build system supports additional platforms, and investigating OpenCL support in addition to CUDA support.

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

[3] CUDA home page: http://www.nvidia.com/cuda

[4] 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)

[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)

[6] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA", GPU Computing Gems: Jade Edition (2011)

[7] Z. Huang et al., Phys. Rev. ST Accel. Beams 7 074401 (2004)

[8] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," ACM Computing Surveys, 23(1): 5-48 (1991)

ISBN 978-3-95450-115-1