ISBN: 978-3-95450-132-8 doi:10.18429/JACoW-IPAC2014-MOPME035

CURRENT STATUS OF THE GPU-ACCELERATED ELEGANT *

K. Amyx†, J.R. King, I.V. Pogorelov‡, Tech-X Corporation, Boulder, CO 80303, USA
M. Borland, R. Soliday, Argonne National Laboratory, Argonne, IL 60439, USA

Abstract

Efficient implementation of general-purpose particle tracking on GPUs can result in significant performance benefits to large-scale tracking simulations. This paper is an update on the current status of our work on accelerating Argonne National Lab's particle accelerator simulation code ELEGANT using CUDA-enabled GPUs. We summarize the performance of beamline elements ported to GPU, and discuss optimization techniques for some important collective effects kernels, in particular our methods of avoiding costly thread contention. We also present preliminary results of a scaling study of the GPU-accelerated version of the code.

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. Several "direct" methods of simultaneously optimizing the dynamic and momentum aperture of storage ring lattices have recently been developed at Argonne [5]. These new methods typically require various forms of tracking the distribution for over a thousand turns, and so can benefit significantly from faster tracking capabilities.

Graphics processing units (GPUs) offer unparalleled general purpose computing performance, at low cost and at high performance per watt, for large problems with high levels of parallelism. 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.

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, and (3) maintain 'silent support' so that GPU-accelerated elements can be used without additional input from the user.

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 K20c GPU to an Intel Core i7-3770K CPU in simulations with a few million particles.

QUAD and DRIFT: Quadrupole and drift elements, implemented as a transport matrix, up to 3^{rd} and 2^{nd} order, respectively: ~ 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 30x acceleration due to its high arithmetic intensity.

KQUAD, KSEXT, MULT: A canonical kick quadrupole, sextupole, and multipole elements using 4^{th} order symplectic integration: 45x acceleration.

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

RCOL: Rectangular collimator: 60x acceleration if particles are removed from simulation.

LSCDRIFT: Longitudinal space charge impedance: ~ 45x acceleration using optimized histogram calculation.

CSRCSBEND: A canonical kick sector dipole with coherent synchrotron radiation: Over 50x acceleration using optimized histogram calculation.

RFCW: RF cavity element, a combination of a first-order matrix RF cavity with exact phase dependence (RFCA), longitudinal wake (WAKE) and transverse wake (TRWAKE) specified as a function of time lag behind the particle, and LSCDRIFT: over 30x acceleration, convolution-based wake elements being the primary bottleneck.

OPTIMIZATION OF COLLECTIVE-EFFECTS KERNELS

Histogram Computation

Many collective-effects beamline elements in ELEGANT require binning of the particle distribution, *i.e.*, computing a histogram. Calculating a histogram on a GPU is difficult because multiple threads often need to update the same location in memory at the same time. This leads to thread contention issues that may cause either 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 suboptimal performance–roughly 10x of a reference CPU implementation.

Our improved algorithm for histogram computation is based on the following observations 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

* Work supported by the DOE Office of Science, Office of Basic Energy Sciences grant No. DE-SC0004585, and in part by Tech-X Corporation
† Current address: Sierra Nevada Corporation, Centennial, CO
‡ ilya@txcorp.com

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2014). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-132-8 doi:10.18429/JACoW-IPAC2014-MOPME035

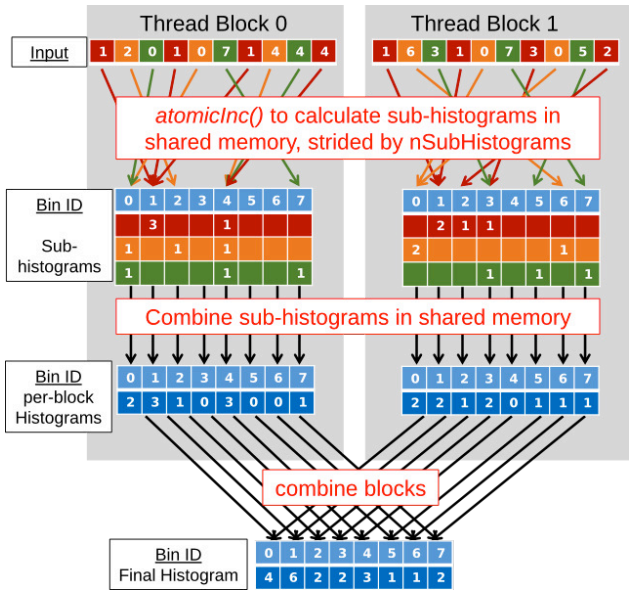

Figure 1: A schematic of the histogram computation algorithm. (See text for details.)

the resulting distribution. Thus, creating additional sub-histograms (as memory size permits) will reduce the thread contention and increase the performance, while the cost of combining these sub-histograms in shared memory will be negligible. Our optimized kernel, illustrated in Fig. 1, utilizes the Fermi GPU's reconfigurable L1/Shared memory cache to prefer shared memory. Because the most costly thread contentions arise from threads within the same warp, we minimize the chances of intra-warp threads accessing the same memory location: sub-histogram access is strided by *nWarpsPerBlock*. The histogram kernel 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.) The final step in our histogram computation is a *__threadfence()*-based reduction that combines the results of multiple thread blocks.

Convolution

Array convolution computation is required by several wakefield elements. In a "typical" ELEGANT simulation, the array size is several hundred to a few thousand, which is not large enough to benefit from a convolution theorem-based approach. At the same time, relying on a serial CPU implementation would negatively impact performance even in large particle count runs, due to the algorithm scaling as the product of array sizes, $O(N_1 N_2)$.

Our optimized convolution kernel achieves good acceleration by buffering sub-sections of each array in shared memory while performing O([buffer size]) computations. This operation computes part of the final result for a given array index. As the convolution is a linear operation, each thread block then applies an atomic addition operation to produce the final result of the convolution.

Reductions with Asynchronous Execution

Reduction operations on particle phase space coordinate arrays are present in many ELEGANT elements and diagnostics. We template standard reduction algorithms over the reduction operation (*e.g.* sum, minimum, maximum, etc.). In these algorithms, thread blocks concurrently apply the reduction operation to subsections of the data array and place the result in global memory. The last block to finish the sub-reduction then reduces the results from the previous step.

Certain functions (i.e. *accumulate_beam_sums* and *compute_centroids*) that compute the beam properties may be called from multiple elements or the main ELEGANT do_tracking loop. These functions reduce quantities from separate data arrays, for example during computations of the mean and standard deviation of beam's phase space coordinates. By launching these functions asynchronously on separate CUDA streams we can take advantage of concurrency during the last reduction operation and the transfer of the reduction result(s), in addition to moving towards achieving the maximum device memory bandwidth during the concurrent reductions. Asynchronous reductions are 40% faster than their synchronous counterparts with data arrays of size one million on a NVIDIA Telsa K20c.

PARTICLE LOSS AND SORTING

Many beamline elements allow for particle loss. When a particle is lost on the CPU, it is swapped with the particle at the end of the particle array and the particle count is decremented. This algorithm is not amenable to the GPU where concurrent particle-update operations are performed. A straightforward GPU algorithm is to fill an array with the particle index plus the number of particles if the particle is lost, and just the particle index otherwise, and then sort the particle array by this key. This is somewhat inefficient still, given that sort algorithms are not amenable to the concurrency of the GPU. However, when the fraction of lost particles for any given element is below 10% or so–as is often the case in practice–a more efficient algorithm than a straightforward sort-by-key can be designed, as we now discuss.

Our particle-loss algorithm is illustrated in Fig. 2. A computational kernel does two things to incorporate particle losses: 1) it returns an unsigned integer (zero if the particle is lost and unity otherwise); and 2) it fills a particle-sort index with the particle index plus the number of particles if the particle is lost, and the particle index otherwise. The particle-loss algorithm then performs a sum reduction over the return value. If the result is equal to the number of particles, no particles are lost and the remainder of the loss computation is skipped. If particles are lost, the end of particle array (size of the number of lost particles) is sorted with *Thrust::sort*, and then sort index is converted to unity if the particle is lost, and zero otherwise. An inclusive scan is performed which creates a particle linear index array. When two subsequent elements of this array are different, a particle

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2014). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-132-8 doi:10.18429/JACoW-IPAC2014-MOPME035

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2014). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

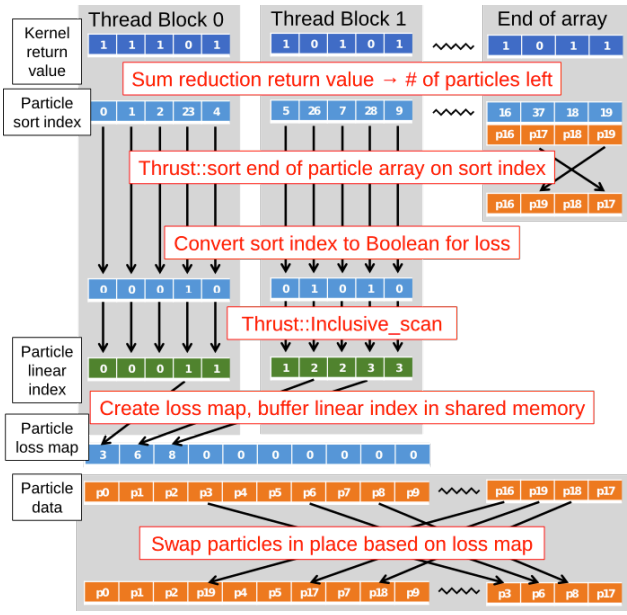

Figure 2: A schematic of the particle-loss sorting algorithm in GPU-accelerated ELEGANT for a simplified case with 20 total and 4 lost particles. (See text for details.)

is lost and the value of the second element in the pair, i, indicates that it is the i^{th} particle that is lost. This information is used to produce a contiguous particle loss map which contains indexing information on the lost particles. A final step uses the particle loss map to swap particles to the end of the particle array, and the particle count is decremented by number of lost particles. This final step is launched with a different CUDA thread block decomposition that accounts for the sparsity of operations. Although the final two steps of this algorithm contain uncoalesced reads and writes, it is still more efficient than a straightforward sort-by-key algorithm due to the sparsity of operations.

Relative to the sort-by-key algorithm, this optimized algorithm is 4× faster with 0.5% losses, 3× faster with 5% losses, 2.5× faster with 10% losses and roughly equivalent with 50% losses (benchmarking on an NVIDIA Tesla K20c).

DISTRIBUTED-MEMORY SCALING

In this section we present preliminary results from performance and scaling studies of the GPU-accelerated ELEGANT relative to the CPU-only version of the code. These studies are performed on the 18,688-node, hybrid-architecture, Titan Cray XK-7 supercomputer at the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory. We use the LCLS linac beam delivery system as our test lattice, so that these studies represent end-to-end application performance in a realistic setting, as opposed to the kernel- and function-specific results from previous sections. This choice presents a more reasonable comparison of the GPU to CPU performance, where the performance of two 8-core AMD Opteron CPUs are compared to the performance of a single NVIDIA Tesla K20x (as there are 16 CPU cores and a single GPU per Titan node). With 8M particles, a K20x GPU is approximately 6 times faster than 16 Opteron CPU cores.

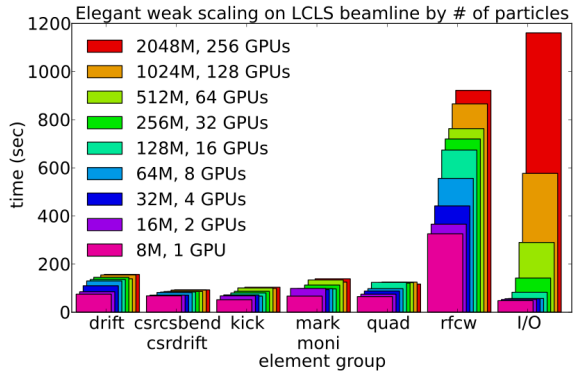

Figure 3: Weak scaling results for the GPU-accelerated version of ELEGANT, arranged by beamline element. LCLS driver linac lattice was used as the test case for this study.

Results of the weak scaling studies (where the number of cores is increased in proportion to the problem size) are shown in Fig. 3. One can see that most beamline elements exhibit nearly perfect scaling over the explored range of the problem sizes (up to 2 billion macroparticles). Exceptions are the I/O (we did not use the ELEGANT's SDDS parallel I/O capabilities in this test), and the RFCW, which has only recently been implemented and still undergoes optimization. One observation from these weak scaling studies is that, for the full LCLS test case, and relative to a job with 8 million particles, it takes the GPU-accelerated code only 4 times longer to run a 256 times bigger job to completion, a very good scaling performance in an important-in-practice range of problem sizes. In particular, the full LCLS beamline simulation was done within 45 minutes when using 2×10^9 particles, which is comparable to the number of actual electrons in the beam.

At the time of writing, we are working on further optimization of the GPU code, and the results presented in this section are likely to be superseded as this work progresses.

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 ICAP'09, 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 ICAP'09, 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).