A Generic Approach for Developing Highly Scalable Particle-Mesh Codes for GPUs

W. Höning, F. Schmitt, R. Widera, H. Burau, G. Juckeland, M. S. Müller, and M. Bussmann

I. INTRODUCTION

We present a general framework for GPU-based low-latency data transfer schemes that can be used for a variety of particle-mesh algorithms [8]. This framework allows to hide the latency of the data transfer between GPU-accelerated computing nodes by interleaving it with the kernel execution on the GPU. We discuss as an example the fully relativistic particle-in-cell (PIC) code PiConGPU [5] currently used to simulate particle acceleration by extremely short high-energy laser pulses. The PIC algorithm is a versatile algorithm used frequently in plasma physics—especially for large-scale simulations of fusion plasmas [13]—, in astrophysics [7], [9], or for the simulation of particle accelerators [11]. A special Cell processor version is used as a benchmark code for the Roadrunner system at Los Alamos National Lab [4].

In general, particle-in-cell codes show very weak scaling in both mesh size and particle number and thus are good candidates for massively-parallel computing approaches [10]. However, when considering vector parallelization as used for example by GPU accelerators one has to take into account that particles can cross the boundaries between mesh domains. Such particle crossings may cause non-local memory access every time mesh-related data such as fields and currents and particle data such as velocities and positions are accessed by the same kernel [7]. Random and non-local memory access can seriously degrade the overall performance on modern accelerator hardware [3], [12]. Thus, performant implementations of the PIC-algorithm with vector parallel memory access accelerator hardware have been sparse [1], [12]. The need for large-scale simulations of plasmas calls for a general approach towards vector-parallel PIC implementations.

The presented hybrid GPU-CPU data transfer and access framework is grid-based and can, furthermore, be used for general particle-mesh schemes. GPU memory access to particle data and regular grid data are efficiently separated, while data that has to be exchanged between domains located on different GPUs is transferred during computing steps using GPU-CPU memory transfers and MPI. A simulation of laser-wakefield acceleration of electrons in an underdense plasma serves as a real-world benchmark for the performance of the framework.

H. Burau and M. Bussmann are with the Forschungszentrum Dresden-Rossendorf e.V., D-01328 Dresden, Germany.
W. Höning, F. Schmitt, R. Widera, G. Juckeland, and M. Müller are with Technical University Dresden, Center for Information Services and High Performance Computing (ZIH), D-01062 Dresden, Germany.

II. SINGLE GPU OPTIMIZATIONS

A. Memory Model - Tiling Particles

For Particle-Mesh algorithms a mapping from cells to particles and vice versa is necessary. Our first basic solution was a particle list where each particle element holds a pointer to its owning cell and its successor [5]. Usually a PIC algorithm will run for a large number of iterations which leads to movement of many particles. This results in heavy fragmentation of the particle list which turns fast regular memory access patterns into slow random memory accesses and, thereby, to continuously increasing time needed for one iteration of the algorithm (figure 2).

The solution to this fragmentation problem is the introduction of a new data structure which uses a three stage memory hierarchy as shown in figure 1. This approach uses coalesced global memory accesses for all particle and cell data except tile domain border regions. Therefore, the global fragmentation does not affect the performance and we have no limit in the amount of particles per cell. Problem specific data has to be copied only if a particle moves to a neighboring supercell. Otherwise, only the cell index has to be adjusted.

The introduction of this data structure reduced the time needed for one iteration as well as the slowdown over time as shown in figure 2.

B. Comparing CUDA and OpenCL

Code written in OpenCL (Open Computing Language) is intended to work on multiple platforms with recompilation.
the underlying hardware architecture does not suit them. Between platforms, optimization strategies are not as long as conclude that though codes are portable in terms of correctness (figure 3). PIConGPU code on various NVIDIA and ATI graphic cards this topic by comparing two implementations of the described which utilized shared memory. We extend previous work on CUDA surpassed OpenCL by factor 8 for an optimized version identical results when only global memory was accessed while cations, CUDA and OpenCL on the Tesla S1070 produced testing matrix multiplication by running several synthetic benchmarks which showed that also compute on different data. We start by computing the fact that one iteration of the code has various steps solution communicates and computes in parallel exploiting the smallest communication/computation imbalance will ultimately limit the scalability of the program. Our presented the main problem of parallel applications is that even the smallest communication/computation imbalance will ultimately limit the scalability of the program. Our presented solution communicates and computes in parallel exploiting the fact that one iteration of the code has various steps that also compute on different data. We start by computing the first step (electrical field for PIConGPU) including the border regions. While the second step, the magnetic field, is processed without the borders, we simultaneously transfer the previously calculated borders for the electrical field. After that, the remaining magnetic field can be computed. The presented libGPUGrid library has an easy to use interface to accomplish this: Instantiate GridBuffer and use startSend() or startReceive() to exchange its borders. All internals (especially MPI) are hidden. The two functions return an EventTask that can be used to test or wait for the communication to finish. A schematic timeline visualization (Fig. 4) illustrates this overlap.

B. Computation Communication Overlap

The main problem of parallel applications is that even the smallest communication/computation imbalance will ultimately limit the scalability of the program. Our presented solution communicates and computes in parallel exploiting the fact that one iteration of the code has various steps that also compute on different data. We start by computing the first step (electrical field for PIConGPU) including the border regions. While the second step, the magnetic field, is processed without the borders, we simultaneously transfer the previously calculated borders for the electrical field. After that, the remaining magnetic field can be computed. The presented libGPUGrid library has an easy to use interface to accomplish this: Instantiate GridBuffer and use startSend() or startReceive() to exchange its borders. All internals (especially MPI) are hidden. The two functions return an EventTask that can be used to test or wait for the communication to finish. A schematic timeline visualization (Fig. 4) illustrates this overlap.

C. Decoupling MPI and CUDA

All MPI and low level CUDA details (i.e. memory management) are hidden in the presented grid library. The CUDA kernels (user code) need not to know that several GPUs are involved. The user must only specify possible transfer directions by defining a communication mask and can start all needed communication with one function call. Internally employing several design patterns like Observer, Strategy

![Figure 2. Performance comparison of old (linked list) [5] and new (tiles) memory models on Tesla S1070 and C2050 (Fermi) GPUs. The Simulation uses one GPU, a 1024 × 2048 grid, about 30 mill. particles, 40,000 iterations. The memory-fragmentation and resulting performance drop can be reduced to almost no influence using the new particle tiling data structure and the latest NVIDIA hardware.](image1)

![Figure 3. Average iteration times for different GPU platforms using one GPU. The simulation uses a 512 × 512 grid and about 700,000 particles. The OpenCL version of the code produces similar results compared to CUDA on NVIDIA platforms, while the same code takes about three times as long on and AMD/ATI GPU.](image2)
The capabilities of this framework include abstraction of all communication (between hosts and easily between host and accelerator devices), the possibility to easily interleave computation and communication, as well as a GPU oriented memory management for particle data. One particular implementation using this framework—PIConGPU—proves the efficiency of the approach by providing to our knowledge the lowest time to compute an iteration compared to other (accelerated) Particle-in-Cell codes. Additionally, we presented experience in porting our PIC code to heterogeneous systems using OpenCL.

Further work on the framework includes a library to abstract particle handling as well.

REFERENCES


