PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster

Heiko Burau, Renée Widera, Wolfgang Hönig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Tomas E. Cowan, Roland Sauerbrey, and Michael Bussmann

***Abstract*—The particle-in-cell (PIC) algorithm is one of the most widely used algorithms in computational plasma physics. With the advent of graphical processing units (GPUs), large-scale plasma simulations on inexpensive GPU clusters are in reach. We present an implementation of a fully relativistic plasma PIC algorithm for GPUs based on the NVIDIA CUDA library. It supports a hybrid architecture consisting of single computation nodes interconnected in a standard cluster topology, with each node carrying one or more GPUs. The internode communication is realized using the message-passing interface. The simulation code *PIConGPU* presented in this paper is, to our knowledge, the first scalable GPU cluster implementation of the PIC algorithm in plasma physics.**

***Index Terms*—Electron accelerators, parallel algorithms, parallel architectures, particle beams, plasma waves, simulation software.**

I. Introduction

THE PARTICLE-IN-CELL (PIC) algorithm [1] is used in plasma physics simulation if the dynamics of the plasma constituents cannot be modeled by ensemble averages, as it is done in fluid approaches. The average particle motion is described by the dynamics of macroparticles carrying both charge q and mass m. These macroparticles are defined by a particle density distribution function that is usually centered around the position $\vec{r}$ of the macroparticle on a mesh. The mesh is a domain decomposition of the plasma volume simulated. In its most simple form, a simulation of relativistic plasma dynamics using the PIC algorithm consists of the following four consecutive steps that are computed at each time step for all particles in each cell on the mesh.

1) Starting with a given distribution of the electric field $\vec{E}$ and magnetic field $\vec{B}$, the Lorentz force $\vec{F}_{\mathrm{Lorentz}}$ is calculated.
2) The Lorentz force is used to integrate the equation of motion $m\ddot{\vec{r}} = \vec{F}_{\mathrm{Lorentz}}$, changing the velocity $\vec{v}$ and the position $\vec{r}$ for all macroparticles. This step is usually called the *particle pusher*.
3) The new positions and momenta of the macroparticles are used to calculate the current $\vec{J}$.
4) Using the principle of superposition of fields, the change in field strength for both $\vec{B}$ and $\vec{E}$ is calculated from the local current following Maxwell's equations.

The Lorentz force is given by

$$\vec{F}_{\mathrm{Lorentz}} = q(\vec{E} + \vec{v} \times \vec{B})$$

while Maxwell's equations are given by

$$\text{Gauss (electric)} \quad \nabla \cdot \vec{E} = \frac{\rho}{\epsilon_0}$$

$$\text{Gauss (magnetic)} \quad \nabla \cdot \vec{B} = 0$$

$$\text{Faraday} \quad \frac{\delta \vec{B}}{\delta t} = -\nabla \times \vec{E}$$

$$\text{Ampere} \quad \frac{\delta \vec{E}}{\delta t} = \frac{1}{\epsilon_0}\left(\frac{1}{\mu_0}\nabla \times \vec{B} - \vec{J}\right).$$

For a relativistic plasma PIC simulation, usually, only Faraday's law of induction and Ampere's circuital law are considered, while Gauss' law for electric charges is only fulfilled for special versions of the algorithm.

A. Local Solution of the Field Equations—The Role of Particles

In a plasma PIC algorithm, Maxwell's equations are interpolated by a finite-difference scheme on the mesh, assigning field values to the individual cells on the mesh. Depending on the finite-difference scheme involved, the local integration of the equations requires information on the local fields and currents from the nearest neighbor cells or next to the nearest neighbor cells. Thus, the fields are fixed to the mesh lattice. With a constant mesh lattice, it is thus possible to project the mesh onto the data memory in a continuous sequence.

Since updating the fields in a cell is, for each time step, a local operation on the mesh that requires only information from adjacent cells, the field computation can be implemented as an independent computation on a local subset of the memory. The field update depends on the currents calculated from the particle motion. As long as particles do not pass boundaries between adjacent cells, the locality of the PIC algorithm is preserved, and particles can be unambiguously assigned to an individual cell. Thus, local memory mapping of both the cell's field data and the particle data is possible. This mapping allows

Manuscript received December 20, 2009; revised June 10, 2010; accepted July 29, 2010. Date of publication August 23, 2010; date of current version October 8, 2010.

H. Burau, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussmann are with the Forschungszentrum Dresden-Rossendorf e.V., 01328 Dresden, Germany.

R. Widera, W. Hönig, and G. Juckeland are with the Center for Information Services and High Performance Computing (ZIH), Technical University Dresden, 01062 Dresden, Germany.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TPS.2010.2064310

one to assign the local memory subset to a single processing unit without the need for data transfer between processing units. Particles crossing cell boundaries invalidate a simple local memory mapping of the PIC algorithm. This will be addressed in the discussion of the graphical processing unit (GPU) implementation.

Data transfer between processing units is required in two cases. First, the field data of adjacent cells must be transferred at each time step, and second, particle crossings require data transfers to retain data locality, associating the particles to the cell that they are in. As an example, consider the local data model using a simple square grid of dimension n with equally sized cells. With increasing size of memory per processing unit, the amount of data processed by a single unit becomes larger than the data transfer between neighboring processing units.

In our case, the data transfer roughly scales with the system size L as L^{n-1}, while the amount of computational operations scales as L^n. For a 2-D ($n = 2$) PIC simulation with a mesh size of 2048×2048 divided into four parts of 1024×1024 cells distributed onto four GPUs, one finds $L = 1024$. The amount of communication between the GPUs is then $4 \times L = 4096$ multiplied by the amount of data transferred between cells, whereas the computational amount is proportional to $4 \times L^2 = 4 \times 1024^2$, since the algorithmic steps are computed for each cell. Thus, computation scales quadratically with the number of cells, while communication only scales linearly. It is therefore desirable to address as much memory as possible to a single processing unit and minimize data transfer.

II. Implementation of a Relativistic Plasma PIC Algorithm on a Single GPU

We present a very simple version of a relativistic 2D3V PIC code designed for simulating laser–plasma interactions. Maxwell equations are solved on the mesh by a simple linear interpolation according to the Yee lattice scheme [2]. This scheme only requires information from the next-neighbor cells. The macroparticle distribution is chosen to be a point-like delta function. The equation of motion is integrated via the leapfrog scheme [3], integrating the particle momentum $d\vec{p}/dt = \vec{F}_{\mathrm{Lorentz}}$ rather than the velocity.

Although such a simple approach has problems with numerical heating [4], dispersion [5], and violation of charge conservation [6], further refinement of the code will not affect the general structure of our implementation. Thus, we have focused on developing a lightweight PIC algorithm that is scalable on a standard cluster of individual nodes, each equipped with one or more GPUs. In the following, we will introduce the key components of our object-oriented C++ implementation, which are essential for running large-scale simulations on GPU clusters.

A. Differences Between CPU- and GPU-Based Clusters Relevant for the Implementation

Today, high-performance computing clusters consist of individual nodes, each equipped with one or more central processing units (CPUs). The nodes are usually interconnected via a small-latency high-bandwidth network that allows for data transfer between the nodes. The weak scaling of the PIC algorithm on such systems is usually almost linear in both the number of particles and the number of mesh points in each dimension. This means that, for a linear increase of the simulated system size, the number of processors has to increase linearly to keep the computation time constant. This scaling behavior is a result of the optimized memory access architecture provided by modern CPUs. Throughout the text, we will mostly relate to the NVIDIA GPU architecture, as detailed in the CUDA Programming Guide [7]. However, most points outlined in the following also hold true for other architectures. For the implementation of the PIC algorithm, the following three major differences between the CPU and GPU architectures play a role:

1) the latency of local memory access compared to nonlocal memory access;
2) the ratio of available memory to the number of processing units;
3) the performance of single- and double-precision floating-point operations.

When considering a cluster of GPU-accelerated nodes, the latency of memory transfer between GPU and CPU, as well as the latency in internode communication, becomes important. A CPU usually has a large data cache that allows for fast random access to the data needed for the computation during a simulation time step. In addition, memory access is optimized by an instruction flow control that determines the most efficient order of independent instructions during the execution of the program. Memory access latency is thus minimized, allowing for fast random memory access. Nonlocal or random memory access introduced by particles crossing between cells is thus not as an important issue on CPU-based computing clusters as it is for GPUs.

In contrast to this, a GPU is designed for parallel data processing, meaning that GPUs are ideal for problems in which the same algorithm is executed on many data elements in parallel. Instead of using intensive caching and memory access optimization, the computing power of GPUs relies on massively parallel execution of many identical instances of the algorithm, the so-called *threads*, accessing different parts of the memory.

Efficient GPU programming is equivalent to providing a sequence of instructions wrapped in a routine, the so-called *kernel*, which can be replicated many times into individual threads. Each thread should require only a local subset of the total memory for its execution, which ideally is contiguous and not accessed by other threads[1]. Nonlocal memory access in which a thread writes to discontiguous parts of the memory or in which two different threads read or write to the same address in memory thus comes with a degradation in computing performance. With modern GPUs, such as the NVIDIA Tesla C1060 [8] with 240 1.3-GHz cores and 4-GB device memory, the ratio of memory per core, for example, is 17 MB/core. For comparison, each of the 2208 nodes of the JUROPA computing cluster at Forschungszentrum Jülich [9] is equipped

[1] A requirement also imposed by OpenMP implementations of the PIC algorithm. The solutions presented in this paper could thus serve as an example for efficient PIC implementations using OpenMP.

with two quad-core Intel Xeon X5570 (Nehalem-EP) CPUs and 24 GB of memory, resulting in a 4-GB/core ratio. The Tesla C1060 has an overall performance of 933 Gflop/s for single-precision floating-point operation and only 78 Gflop/s for double-precision floating-point operation, and the JUROPA double-precision floating-point performance is estimated to be 207 Tflop/s. In order to compete with PIC implementations on modern high-performance computers, implementations of the PIC algorithm on GPUs must use single-precision floating-point operations wherever numerical stability permits it.[2]

B. Memory Usage

The CUDA programming model provides several types of memory. First, one has to differentiate between the memory of the CPU host system, the so-called *host memory*, and the memory on the GPU device, the so-called *device memory*. In this section, we concentrate on *global*, *shared*, and *texture* device memory[3]. Shared memory provides the fastest memory access of these three, because it resides on chip, close to the processing unit. It is, however, limited in size and is not persistent across the lifetime of a kernel.

Global memory provides the slowest memory access but is persistent across the lifetime of a kernel. It can be accessed by all threads. Its size is limited only by the total physical memory on the GPU. Texture memory provides faster access to global memory for contiguous 1- or 2-D subsets of the memory through efficient caching. It is mandatory for efficient GPU programming to use single-precision floating-point operations as often as possible.

For large numbers of simulation steps, this poses a problem, since numerical stability becomes important. We have thus chosen particle positions to be relative to the cell position in the mesh instead of using absolute positions relative to a common point of origin. Particle momenta, as well as currents and electric and magnetic fields, are, however, stored as absolute values. The first tests show good numerical stability over several thousand time steps. A detailed analysis, however, will be the subject of an upcoming paper.

In the current version of the code, data are mostly stored in single-precision floating-point format in global device memory. Shared memory is used during the update of the fields and currents as a storage for temporary values. Texture read access to global memory is used for particle positions and the linked particle list—see the next section for details. A list of the most important data structures used in the simulation is found in Table I.

Electric and magnetic fields, as well as currents, are stored in objects derived from class `Field` that provides the mapping from the mesh lattice onto memory via two cell indexes. Particles are stored in an object of class `Particles` that holds both the particle data and information on the linked list. Electrons and ions are stored separately, and each step in the simulation accessing particle data is thus called two times, i.e., once for electrons and once for ions. This data separation enhances local data access, since particle crossings occur more often for lightweight electrons than for heavy ions.

TABLE I
MEMORY USAGE OF SIMULATION DATA

Simulation data	Data type	Associated memory
Electric field	float3	device global
Magnetic field	float3	device global
Current	float4	device global
Particle position	float2	device global
Particle momentum	float3	device global

C. Algorithmic Implementation—Linked Particle List

The linked particle list is one of the core features of our GPU PIC implementation. Similar techniques have already been used in high-performance PIC codes [11], [12] and have also been proposed for GPU architectures [13]. It is an approach toward localized data access for each kernel. Nonlocal memory access occurs when kernels access both mesh data, such as fields and currents, and particle data, such as positions and momenta. The execution order of the six main kernels is as follows.

1) Compute the electric field from Ampere's law.
2) Compute the magnetic field from Faraday's law.
3) Particle pusher (electrons).
4) Particle pusher (ions).
5) Compute the currents (electrons).
6) Compute the currents (ions).

The first two kernels only operate on current and field data and thus directly access the 2-D memory layout of the grid. For the time integration of Ampere's law, the local magnetic field is stored in shared memory before advancing the field—as is the electric field for the integration of Faraday's law. The linked particle list becomes important if both mesh and particle data are accessed in the same kernel. This holds true for both the particle pusher and the current computation. The first performance tests show that the kernel computing the current from the new particle momenta is the most time consuming of all kernel calls.

The memory layout for the cells and particles, as well as the linked particle list, is shown in Fig. 1. Each particle is associated to its current cell by a unique cell index. In addition, each particle carries a reference to its predecessor in the linked particle list. Instead of defining a global linked particle list, we have chosen to associate local lists to each cell by referencing the last particle in the cell's linked list to it. Thus, using the particle references to their predecessor, one can iterate the list for each cell.

As can be seen from the example of a border crossing shown in Fig. 1, the order of particles in the lists does not reflect the order in which particles are stored in memory. Therefore, border crossings of particles generally cause a different order in accessing the linked particle lists.

[2]Of course, using single-precision floating-point operations on high-performance computing systems is, in general, a good way to boost performance. With GPUs, the difference in execution time, however, demands the use of single-precision floating-point operation. This might change with new GPU architectures like NVIDIA Fermi [10].

[3]The very fast *register* memory is only used for single variables determining physical parameters used in the simulation. It is very limited in size, and we thus do not regard it as an alternative to shared memory, which provides almost comparable memory access times.

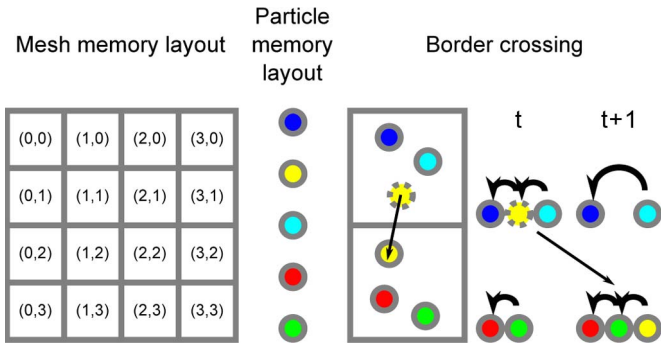

Fig. 1. For the 2D3V PIC implementation, cells are mapped directly onto memory in a continuous manner, following the indexing scheme shown on the left. Each cell is given a unique cell index. The same holds true for particle data, which are likewise mapped onto memory by a single particle index. Each particle has a reference to its current cell via the corresponding cell index, while each cell has a reference index to the last particle in its local linked particle list. In order to link the list elements, each particle has a reference to its predecessor in the list. A particle crossing borders is deleted from the particle list of its former cell and appended to the end of the particle list belonging to its new cell. This is depicted on the right for a border crossing between time steps t and $t+1$.

This reordered access pattern can reduce the performance in both the particle pusher and the current computation. However, the problem of noncontinuous memory access is most severe in the current computation. The particle pusher threads are called for individual particles. The fields needed to compute the Lorentz force are accessed via the cell index. Since a particle is associated to a unique cell, each thread will take about the same time to complete. This is due to the fact that the cell data are continuous in memory, so for each thread, memory access is local, and nonlocal memory access can only occur between two different threads. This is different for the current computation. Each thread is called for a unique cell and must iterate the particles linked to this cell. Thus, the threads can differ in execution time, since some cells have longer particle lists than others.

In order to minimize these differences, global memory access to particle data is provided using texture read access. We use `cudaBindTexture` to bind the particle data residing in global memory to a readable texture memory location. For the computationally most expensive task of iterating the particle list when computing the current deposition, fast memory reads are crucial. It should be made clear that texture memory is only used for reading particle data. Thus, latency due to binding the global memory to the texture memory is small compared to the performance gain when iterating the particle list via texture memory read access.

Particle list element deletion and insertion are handled via atomic operations, since more than one particle can enter or leave a cell. When increasing the number of macroparticles per cell, atomic operations can take considerable time because boundary crossings occur more frequently. However, it is not the atomic operations that limit the execution speed of our code. Instead, the routine calculating the current deposition by iterating over the linked list of particles currently takes most of the execution time for a single time advance. Even at high particle numbers, calls for atomic particle deletion and insertion due to particles crossing cell boundaries occur less often than random unordered memory accesses due to particle list iteration. We are currently testing a new memory model replacing the linked-list approach discussed here, which might allow for faster calculation of the current deposition. Please note that performance tests show that border crossings of particles and the corresponding unordered memory access patterns significantly change the execution time for the current computation only after several thousand time steps [see Fig. 6 (right)]. A resorting of the particle list every thousand time steps, meaning after few microseconds of execution time, could reduce the linear increase in single-step duration observed. Finding an optimum frequency for calling a suitable resorting algorithm strongly depends on the simulated physical scenario. For large particle numbers and highly perturbed plasmas, resorting must be invoked more often than for homogeneous plasmas with small particle density.

III. Implementation of a Relativistic Plasma PIC Algorithm on a Cluster of GPUs

Copying data from the GPU device to the CPU host system and transferring data from one host system to the other via network are both tasks with high latency. However, it is important to overcome the problem of high latency in order to scale the PIC algorithm to a cluster of GPU-accelerated CPU nodes. The need for GPU cluster scaling becomes apparent when considering the very limited GPU on-board memory. Large-scale plasma simulations that would profit from GPU acceleration cannot be handled by a maximum of four GPU cards that can be placed in a single node. Instead, a scalable PIC implementation must rely on a domain decomposition of the simulation volume that spans over several CPU host systems, each equipped with one or more GPU devices. The simulation volume is then divided so as to maximize the memory usage on each single GPU. As pointed out in Section I-A, an increase of local GPU memory while keeping the number of GPU processing cores constant increases the ratio of computational steps performed on a single GPU over the amount of data transferred between host systems. Thus, an increase in on-board memory size with respect to the computation power will reduce the problem of high-latency memory transfers. This, of course, also holds true for message-passing interface (MPI) and OpenMP implementations of the PIC algorithm. For the case of a GPU that usually has much less local memory compared to a CPU cluster of similar computing power, the ratio is much smaller, and thus, communication latency is of much greater importance compared to a standard cluster implementation.

A. Domain Decomposition and Locality of the PIC Algorithm

In our implementation, the host CPUs are utilized solely for memory transfer between CPU and GPU, as well as internode communication via MPI [14]. We exploit the domain locality of the PIC algorithm to interleave computation on the GPU with data exchange.

We define guarding cells on each border of a subdomain connected to an adjacent subdomain. In the following, we will call the subdomain without guarding cells the *core* and the guarding cells the *border*, as shown in Fig. 2.

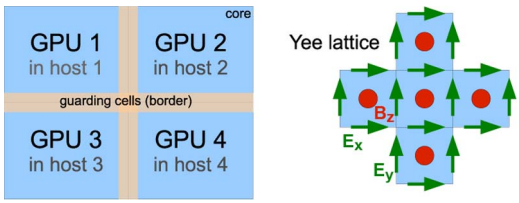

Fig. 2. Simulation domain is decomposed into subdomains. Each GPU device manages its own subdomain. Each subdomain is composed of (blue) a core and (gray) a border. The border consists of guarding cells that are mirrored on the host. Internode communication is handled by the CPU on each node using MPI. Since the PIC algorithm relies on local field updates following the Yee lattice layout depicted on the right, it is possible to compute fields for core and border independently. Data then can be exchanged between nodes via the guarding cells without interfering with GPU computations on the core.

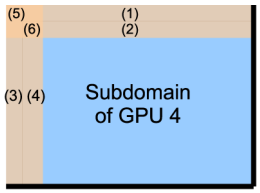

Fig. 3. Communication scheme depicted before can be implemented independently of the interpolation scheme used to integrate the field equations. It consists of two guarding-cell arrays defining the border, i.e., one allowing for read access and the other allowing for write access. The communication then follows the pattern: (1) Incoming data from GPU 2, (2) outgoing data to GPU 2, (3) incoming data from GPU 3, (4) outgoing data to GPU 3, (5) incoming data from GPU 1, and (6) outgoing data to GPU 1 (see Fig. 2).

The locality of the PIC algorithm allows one to execute the steps in the algorithm for all cells in the core while data exchange is performed between borders. It should be noted that the communication pattern depends on the lattice layout chosen for the interpolation of Maxwell's equations. In the case of the standard Yee lattice, one can compute the electric field solely from the local magnetic field and current, while for the computation of magnetic field and current, data from adjacent cells are required.

The algorithmic structure can be found hereinafter. The individual PIC steps are interleaved by communication calls, as visualized by printing the communication in italic and indented by a leading arrow.

GPU PIC algorithm

1: Compute the electric field (core and borders)
2: $\mapsto$ *Start the exchange of electric field data*
3: Compute the magnetic field (core)
4: $\mapsto$ *Finalize the exchange of electric field data*
5: Compute the magnetic field (border)

6: $\mapsto$ *Start the exchange of magnetic field data*
7: Particle pusher for electrons
8: Particle pusher for ions
9: $\mapsto$ *Finalize the exchange of magnetic field data*

10: $\mapsto$ *Start the exchange of electron data*
11: $\mapsto$ *Start the exchange of ion data*
12: Compute the currents for electrons (core)
13: Compute the currents for ions (core)
14: $\mapsto$ *Finalize the exchange of electron data*
15: $\mapsto$ *Finalize the exchange of ion data*

16: Compute the currents for electrons (border)
17: Compute the currents for ions (border)

18: $\mapsto$ *Start the exchange of current data*
19: Sum up the currents over adjacent cells (core)
20: $\mapsto$ *Finalize the exchange of current data*
21: Sum up the currents over adjacent cells (border)

For other interpolation schemes, data exchange patterns might differ from the steps shown earlier. This is particularly true if other combinations of higher order current deposition and particle pusher schemes are used. However, these can and will be incorporated in future versions of the code. Our approach to hide the communication latency by interleaving communication with the calculation steps on the GPU can still be applied to these more complex schemes, since the algorithmic listing must only be altered at a few places to match these schemes and should be regarded as a general scheme to incorporate communication into any GPU implementation of the PIC algorithm. To take this into account, our approach to create an independent modular library for communication helps one to build customized PIC implementations. As a second approach, we present an alternative scheme that can be used almost independently from the choice of the interpolation scheme (see Fig. 3). It uses a border of two guarding-cell arrays that allows for independent read and write access to the memory. We currently compare this scheme to the interleaved scheme described in the text.

Data synchronization between the host and device is managed by class `HostAndDeviceData`. Memory is allocated for data exchange both on the device—by instantiating an object of type `DeviceData` and the host by instantiating an object of type `HostData`. The routine `cudaMemcpy` is used to copy the data. For the latter, asynchronous data exchange might speed up the algorithm, but care has to be taken to ensure data integrity after each time step. As explained in the following section, the memory transfer from GPU to CPU executes simultaneously to the internode communication using MPI, so asynchronous memory copy must preserve the data used by MPI communication calls. A complete analysis of the benefits and problems when using asynchronous memory transfer is beyond the scope of this paper.

While field and current data can be exchanged following the simple guarding-cell approach, particles crossing boundaries between two host systems are treated using a so-called *graveyard* list. This list provides device global memory for particles entering the subdomain associated to a GPU device. Instead of time-consuming per-particle allocation and deallocation of global memory, a dynamically growing chunk of memory, i.e., the graveyard, is allocated, from which memory space for particles entering the device subdomain can be borrowed. If a particle leaves the device subdomain, the borrowed memory is passed back to the graveyard. Dynamic memory management ensures that graveyard memory is available at every time.

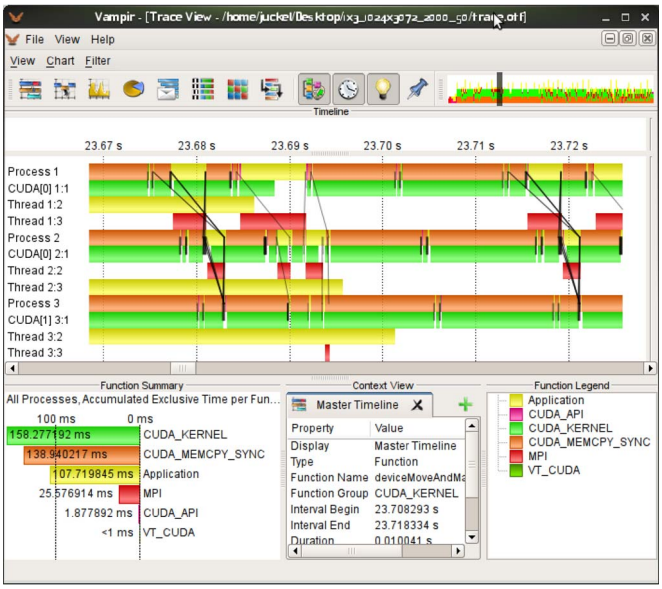

Fig. 4. VAMPIR trace of a 1024×3072 cells simulation run with three GPUs. Processes 1, 2, and 3 refer to the main program thread that transfers data between the GPU and the main memory. It is dominated by `CUDA_MEMCPY_SYNC` calls because the main program waits for the execution of the GPU kernels when calling a blocking operation, such as `CUDA_MEMCPY_SYNC`. Threads 1:2, 2:3, and 3:2 are responsible for the visual data output of the GPU memory, while threads 1:3, 2:2, and 3:3 are solely responsible for internode communication via MPI. The execution state of the CUDA kernels is visualized by the green bars named CUDA[0] 1:1, CUDA[0] 2:1, and CUDA[1] 3:1, where the number in brackets identifies the corresponding node. As can be seen from the VAMPIR trace, the execution of the CUDA kernels, meaning the computation of the PIC algorithm on GPUs, is barely interrupted by communication calls, and the computational load on GPUs is well balanced.

B. Optimizing MPI Internode Data Exchange and Kernel Execution

While latency in memory transfer between the device and host is important, latency in internode communication finally determines the feasibility of porting a single-GPU PIC code onto a GPU cluster. With the newest version of the VAMPIR [15] performance tracing suite maintained by the Center for Information Services and High Performance Computing (ZIH), Technical University Dresden, Dresden, Germany, it has become possible to trace the execution of the GPU kernels parallel to the MPI data transfer.

Before optimization, VAMPIR performance traces had revealed that GPU utilization was still below 50%, which was mainly due to MPI communication delays. Although data exchange used nonblocking `MPI_Isend`/`MPI_Irecv` pairs, the `MPI_Waitall` routines invoked by the host CPU to synchronize the data for all hosts at each time step blocked execution of the host process until the last sender had sent its data. This prevented concurrent `cudaMemcpys`.

In order to allow concurrence on the host CPU side, `MPI_Waitall` has been moved to its own thread spawned on the host CPU. This allows MPI communication and CUDA memory transfers to occur at the same time (see Fig. 4) and thus serves as the key component to port the single-GPU implementation depicted previously to a cluster of GPU nodes. In our current implementation, the data for both fields and particles are sent at once, so every MPI communication handles a large amount of data. Thus, moving the MPI communication to an independent thread allowing for simultaneous calls of `cudaMemcpy` (see Fig. 4) was mandatory to increase the GPU load close to 90%, depending on the number of GPUs used. While it seems a waste of CPU cores to use two cores for communication, the overall performance gain makes it worthwhile mainly for two reasons. First, the ratio of GPU devices to CPU cores is in favor of CPU cores on a multicore CPU node; hence, CPU cores need to be occupied with work anyway. Second, the ratio of computational capability of the CPU and GPU is in favor of the GPU by an order of magnitude. An increase of GPU load by almost a factor of two compared to the single-thread version should therefore be weighed against the loss of overall computational power when occupying an additional CPU solely for communication. In the future, we will test a variation of the communication scheme outlined earlier, in which small amounts of data are streamed continuously between the GPU hosts, so that the high CPU load due to sending large amounts of data via MPI and `cudaMemcpy` at once is reduced. This might allow one to use a single thread for communication.

C. Performance Tests

We use our PIC code to simulate the acceleration of electrons in an underdense plasma by a laser-driven wakefield in the so-called *bubble regime* [16] (see Fig. 5). The initial laser pulse has a central wavelength of 800 nm and a Gaussian form in both space and time. The pulse has a width of 8 μm, a duration of 4:7 fs (full-width at half-maximum), and a maximum field amplitude of 5×10^{14} V $\cdot$ m^{-1}. It enters a fully ionized pure hydrogen plasma with an initial linear density gradient of 64 mesh cells. The plasma density was set to 1.745×10^{19} cm^{-3}, which relates to 1.57% of the critical density. The number of cells, cell width Δx, height Δy, simulation time step Δt, and number N_{pic} of macroparticles per cell were chosen according to the number of GPUs to keep the ratio between the size of the system simulated and the number of GPUs constant.[4] Two configurations are tested. In the first configuration, the simulation box is extended along the laser propagation axis by appending an area of 1024×1024 cells to the end of the existing box with each new GPU added. In the second configuration, the simulation box is widened first, so that the box width perpendicular to the laser propagation axis is covered by two GPUs, and extended afterward along the laser axis using two more GPUs. The set of simulation parameters listed in Table II is tested for performance for one, two, four, and eight macroparticles per cell and particle species.

The algorithmic performance is tested on two nodes (Core i7 920, 2.66 GHz, 8-GB RAM) connected via Gigabit Ethernet with two GPUs per node (first node: 2xGTX 280; second node: 1xGTX 280, 1xTESLA C1060) running Ubuntu Linux 2.6.28-15, OpenMPI 1.3.3, and CUDA 2.2.

Heat-related stability problems were encountered during the simulation runs, causing temporary memory faults on GPUs,

[4]When changing the cell size or the number of macroparticles, we made sure to keep the plasma density constant by reweighing the individual macroparticles.

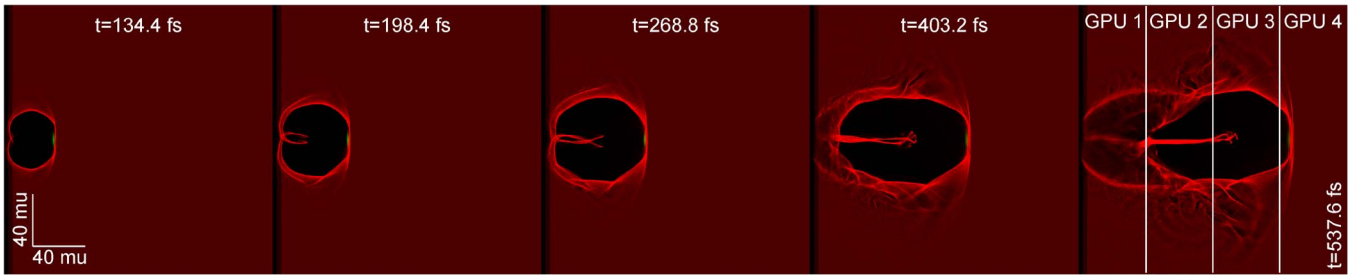

Fig. 5. Upper figure shows the electron dynamics during laser wakefield acceleration in the so-called bubble regime at five consecutive time steps (1024 × 4096 cells and $N_{\mathrm{pic}} = 2$; see Table II). (Green) A high-intensity ultrashort laser pulse enters an underdense hydrogen plasma from the left, pushing aside (red) electrons. The positive hydrogen ions (not shown in the photograph) remain almost at rest, and thus, a positively charged cavity that is void of electrons, the so-called *bubble*, forms behind the laser pulse. Electrons from the back of the cavity get accelerated in the strong field created by the hydrogen ions inside the bubble. They enter the cavity from the back and form a large stem of electrons inside the bubble. Please note that the figure shows the individual electron positions and not a density distribution. The distribution of the simulation volume onto the GPUs is shown in the rightmost panel.

TABLE II
SIMULATION PARAMETERS

GPUs(x) × GPUs(y)	cells ($x \times y$)	Δx	Δy	Δt
1×1	1024×1024	200 nm	50 nm	16 as
1×2	1024×2048	200 nm	50 nm	32 as
1×3	1024×3072	200 nm	50 nm	48 as
1×4	1024×4096	200 nm	50 nm	64 as
2×1	2048×1024	100 nm	50 nm	16 as
2×2	2048×2048	100 nm	50 nm	32 as

which could only be resolved by rebooting the system. In the algorithm, the calculation of the relativistic γ-factor, which is needed for the conversion of relativistic momentum to velocity when solving the equation of motion, relies on a `1/sqrt` operation. Not-a-number errors were observed when using the optimized `rsqrt` to boost performance, and we thus had to use a simple `1/sqrt` operation instead, loosing in overall performance by a factor of 1.45 compared to the optimized version. A first test on a NVIDIA Tesla S1070 system did not show the same behavior, and we expect it to be able to use the optimized version in the future.

In order to compare the results for the various runs, we have chosen to increase the physical size of the simulated system according to the number of GPUs used. This means that, for every GPU added, we increase the total number of cells simulated accordingly (see Table II). The time τ shown in Fig. 6 (left) is computed by dividing the overall simulation run time t_{run} for $N_{\mathrm{step}} = 1000$ time steps[5] by the number of steps and the total number N_{part} of macroparticles and multiplying it with the number N_{GPU} of GPUs used

$$\tau = \frac{t_{\mathrm{run}} \times N_{\mathrm{GPU}}}{N_{\mathrm{step}} \times N_{\mathrm{part}}}$$

where the total number N_{part} of particles is roughly[6] given by the number of mesh cells multiplied by N_{pic}.

The duration of a single time step on a GPU is therefore given by $\tau \times N_{\mathrm{part}}$ and can be used to compute the weak scaling of our implementation. The weak scaling shows a decrease in performance by a factor of almost 1.4 ($N_{\mathrm{pic}} = 1$ comparing the simulation using a single GPU (1×1) to the 1×4 GPUs case). It is interesting to compare the latter case to the 2×2 GPUs setup, which only shows a performance decrease by a factor of 1.2. This difference can be attributed to the differences in computational load on GPUs. In the 1×4 GPUs case, as shown in Fig. 5, the plasma simulated on GPU 1 is perturbed earlier by the laser than that for GPU 4. This means that the local particle list ordering on GPU 1 is affected earlier in the simulation than the particle list ordering on GPU 4. For the 2×2 GPUs case, the load is more evenly distributed among the GPUs.

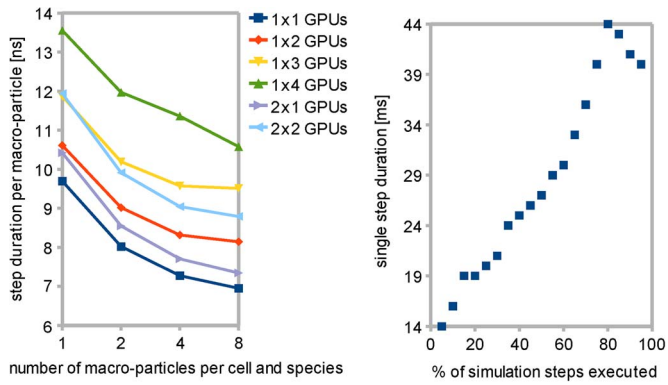

Fig. 6. (Left) Step duration τ versus the number N_{pic} of macroparticles per cell, as measured for the simulation runs listed in Table II. The increase of τ observed when increasing the system size can be attributed to load balancing, as explained in the text. Increasing N_{pic} leads to a decrease in τ. For low N_{pic}, the computation of the fields dominates the overall computation load, while with increasing N_{pic}, the computation load of both particle pusher and current computation becomes dominant. (Right) Single-step duration t_{step} for a 1×4 GPUs run with $\Delta t = 20$ and $N_{\mathrm{step}} = 45\,000$ versus the amount of time steps processed. With increasing step number, t_{step} increases linearly until the laser pulse has left the simulation box. At this point, t_{step} saturates because reordering of the plasma due to laser–plasma interaction has almost completed.

This interpretation is supported by comparing the corresponding increase in τ for $N_{\mathrm{pic}} = 8$. Here, the increase in τ for the 1×4 GPUs is almost a factor of 1.5 compared to a factor of 1.3 for the 2×2 GPUs setup. An increase in N_{pic} means that the overall time spent by the algorithm to go through the particle list is increased, and thus, the ratio between the time spent for computation to the time spent for communication is increased. Thus, one expects a decrease in τ, as seen in Fig. 6 (left). If the latency of network communication would be the main reason for the weak scaling measured, one would expect a

[5]except for the 1×4 GPUs run with $N_{\mathrm{pic}} = 8$ for which we only ran 9000 time steps.

[6]Please note that the computational steps in the algorithm are performed for both electrons and ions. We thus normalize the results to the total particle number, including both electrons and ions. We also take into account the density gradient at the beginning of the simulation volume.

better weak scaling when increasing N_{pic}, which is opposite to what is observed. This is shown in Fig. 6 (right), which shows the increase in the duration of a single time step due to changes in the ordering of the particle list by almost a factor of three between the case of an unperturbed homogeneous plasma and the final situation of a plasma strongly perturbed by the laser pulse, as shown in Fig. 5. A complete analysis of load balancing will be the subject of an upcoming publication and is beyond the scope of this paper.

IV. Summary and Outlook

We have shown that an implementation of the PIC algorithm for a single GPU can be scaled to run on a cluster of GPU notes despite the high latency of data exchange between the host and device and between host nodes. The implementation presented here features a linked particle list to optimize local memory access on the GPU. Internode data exchange is handled via threaded, an nonblocking MPI communication and memory copies between the host and device are interleaved with the computation of individual steps of the PIC algorithm. It should be noted that both the communication scheme and the linked-list approach can be easily adopted to a full 3D3V PIC implementation, although, for such a case, the ratio between data communicated and data computed is greater than that for the 2D3V case.

The locality of the PIC algorithm makes it possible to mask data exchange between nodes by computations updating fields and particles on the core of a subdomain mapped to a single-GPU device. Guarding cells defining the border of such a subdomain provide temporary memory for data exchange between devices.

The first tests have shown numerical stability of all single-precision floating-point operations for several thousand time steps. Still, long-term tests are necessary to fully test for numerical stability. The implementation presented here has been used to simulate laser acceleration of electrons in underdense plasmas in the so-called bubble regime. For this application, plasma densities are low, and time-consuming operations on particles do not undo the performance gained with GPU acceleration. Further optimization of the particle pusher and the current computation can help one to simulate high-density plasmas, as required for the realistic simulation of laser ion acceleration and fusion plasmas.

With GPU devices, such as the new NVIDIA Fermi architecture [10], higher double-precision performance that is comparable to single-precision performance will be available. Both global and shared device memories will be increased, and memory access will be optimized by hierarchical memory caching. Furthermore, faster and more versatile atomic operations, IEEE-conforming floating-point operations, and error-correcting code memory access are foreseen.

With these developments, large-scale plasma PIC simulations on a GPU cluster using small-latency network transfer and high-performance file access are in reach. With the techniques developed here, we have shown for the first time that both nonlocal memory access due to particles crossing subdomain borders and high latency of data exchange can be controlled and that GPU acceleration can be exploited for particle plasma simulations.

References

[1] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation (Series in Plasma Physics)*., 1st ed. New York: Taylor & Francis, Oct. 2004.

[2] K. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. Antennas Propag.*, vol. AP-14, no. 3, pp. 302–307, May 1966.

[3] R. Hockney and J. Eastwood, *Computer Simulation Using Particles*. New York: Taylor & Francis, Jan. 1988.

[4] F. S. Tsung, T. Antonsen, D. L. Bruhwiler, J. R. Cary, V. K. Decyk, E. Esarey, C. G. R. Geddes, C. Huang, A. Hakim, T. Katsouleas, W. Lu, P. Messmer, W. B. Mori, M. Tzoufras, and J. Vieira, "Three-dimensional particle-in-cell simulations of laser wakefield experiments," *J. Phys.: Conf. Ser.*, vol. 78, no. 1, p. 012 077, 2007.

[5] A. Pukhov, "Three-dimensional electromagnetic relativistic particle-in-cell code VLPL (Virtual Laser Plasma Lab)," *J. Plasma Phys.*, vol. 61, no. 3, pp. 425–433, Apr. 1999.

[6] T. Esirkepov, "Exact charge conservation scheme for particle-in-cell simulation with an arbitrary form-factor," *Comput. Phys. Commun.*, vol. 135, no. 2, pp. 144–153, Apr. 2001.

[7] NVIDIA CUDA Programming Guide, NVIDIA Corporation, Santa Clara, CA, Aug. 2009.

[8] Tesla C1060 Data Sheet, NVIDIA Corporation, Santa Clara, CA, Jun. 2008.

[9] Jülich Supercomputing Center, Juropa System Configuration, Jülich, Germany, Jul. 2009. [Online]. Available: http://www.fz-juelich.de/jsc/juropa/configuration/

[10] NVIDIA Corporation, NVIDIAs Next Generation CUDA Compute Architecture: Fermi, Santa Clara, CA, 2009.

[11] K. Bowers, "Accelerating a particle-in-cell simulation using a hybrid counting sort," *J. Comput. Phys.*, vol. 173, no. 2, pp. 393–411, Nov. 2001.

[12] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proc. ACM/IEEE SC*, 2008, pp. 1–11.

[13] G. Stantchev, W. Dorland, and N. Gumerov, "Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1339–1349, Oct. 2008.

[14] MPI: A Message-Passing Interface Standard, Univ. Tennessee, Knoxville, TN, Sep. 2009.

[15] Vampir Website. [Online]. Available: http://www.vampir.eu/

[16] A. Pukhov and J. Meyer-Ter-Vehn, "Laser wake field acceleration: The highly non-linear broken-wave regime," *Appl. Phys. B: Lasers Opt.*, vol. 74, no. 4, pp. 355–361, Apr. 2002.

Heiko Burau, photograph and biography not available at the time of publication.

Renée Widera, photograph and biography not available at the time of publication.

Wolfgang Hönig, photograph and biography not available at the time of publication.

Guido Juckeland, photograph and biography not available at the time of publication.

Alexander Debus, photograph and biography not available at the time of publication.

Thomas Kluge, photograph and biography not available at the time of publication.

Ulrich Schramm, photograph and biography not available at the time of publication.

Tomas E. Cowan, photograph and biography not available at the time of publication.

Roland Sauerbrey, photograph and biography not available at the time of publication.

Michael Bussmann, photograph and biography not available at the time of publication.