Particle in cell simulations on GPU clusters

Francesco Rossi [1]

[1] University of Bologna and INFN, LOASIS affiliate for the summer

AFRD Meeting, May 21st, 2013

Table of Contents

1 Introduction to GPUs

2 GPU PIC current deposition algorithm

3 Multi GPU parallelization

4 Benchmarks and simulations

Simulations for laser plasma acceleration

- Particle in cell (PIC) simulations are useful tools for designing and optimizing laser-driven, plasma-based accelerators.
- Such simulations may require a huge amount of computation ($\gtrsim 10^5$ CPU hours).

GPU Particle in cell codes

1. Viktor K. Decyk and Tajendra V. Singh ,"Adaptable Particle-in-Cell algorithms for graphical processing units," Computer Physics Communications 182, 641 (2011).
2. Burau,Widera,Hönig,Juckeland,Debus,Kluge,Schramm,Cowan,Sauerbrey,Bussmann. “PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster,” IEEE Transactions on Plasma Science 38(10)
3. Kong,Huang,Ren,Decyk. “Particle-in-cell simulations with charge-conserving current deposition on graphic processing units.” J.Comput.Phys.230,4(February2011)
4. Madduri, Kamesh and Ibrahim, Khaled Z. and Williams, Samuel and Im, Eun-Jin and Ethier, Stephane and Shalf, John and Oliker, Leonid “Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems” Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis

Motivation: why a GPU PIC code?

2012

Run simulations we need to run today on the most efficient parallel architectures available (GPUs) for PICs

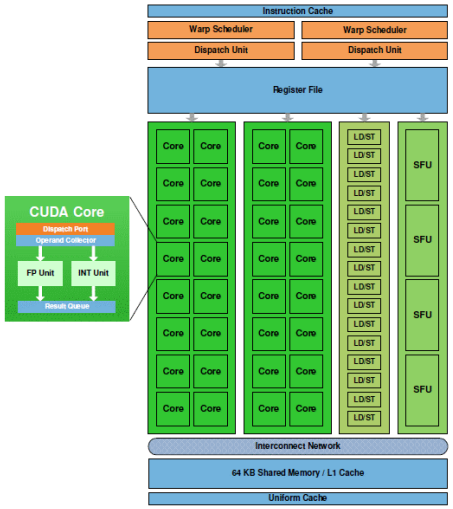
- Performance exploiting exposed parallelism
- Efficiency from data locality

2016

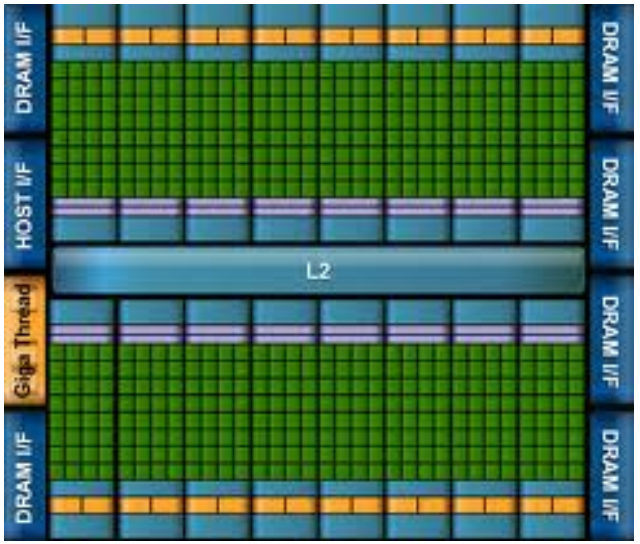
- Path to exascale computing era → dominated by manycore architectures
  - Prepare PIC algorithms for massively-manycore shared-memory node systems
  - Bigger subdomains → fundamental for scalability/load balancing
- GPUs roadmap promises 200% performance increase every year and a half (next generation out this fall)
  - Help to sustain the computational demand in LPA community

The NVIDIA CUDA GPU architecture

- On chip: ~15 Multiprocessors, each one:
  - 256 KB register files
  - 16-48 Kb manual cache: shared memory
  - Issuing instructions
    - executing “warps” of 32 threads in a SIMD fashion
    - divergent branches in a warp cause warp serialization
  - Hides latency keeping many thread warps in flight
- High-bandwidth memory bus (~200 Gb/s) connecting to device RAM
  - Prefers ordered access within a warp
  - Cannot rely on cache: number of cache bytes-per-thread is several orders of magnitude lower than on CPUs



Fermi Streaming Multiprocessor (SM)



The latency / throughput dilemma

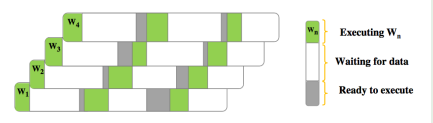
- Memory Latency and Bandwidth are often limiting performance. Two different strategies:

Single thread optimization

- In scalar processors
- Reduce latencies
- Use large caches (per thread)
- Predict branches

Throughput processors: GPUs

1. Provide high bandwidth/througputs
2. *Saturate it: Tolerate* latencies processing many threads in *parallel*
3. Space/energy saved removing scalar optimization used for having more computational power



Similar considerations also apply for other instructions, not only memory accesses.

Jasmine

- "*Jasmine*", a 3D GPU particle in cell code (PIC), featuring:
  - Second order explicit PIC algorithm (FDTD + Boris Pusher) in double precision
  - High order particles shape functions
  - Charge conserving simulations using Esirkepov shape factors
  - 3D multi-GPU simulations with high scalability
  - Dynamic load balancing
  - Moving window
  - Particle trajectory tracking
  - Simulation restart & asynchronous I/O
  - Integrated with a radiation generation computation code

Table of Contents

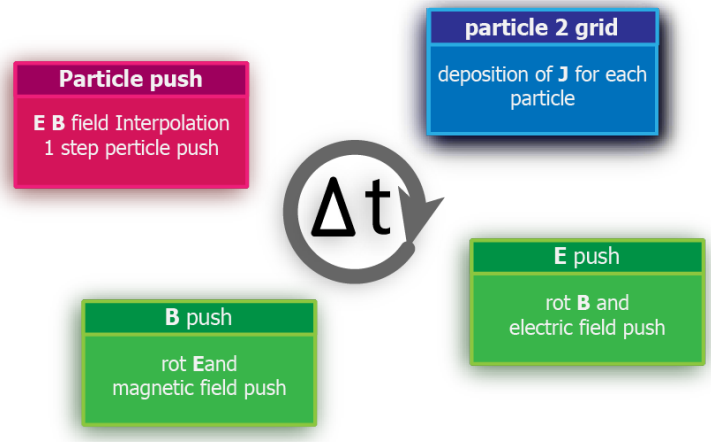
1 Introduction to GPUs

2 GPU PIC current deposition algorithm

3 Multi GPU parallelization

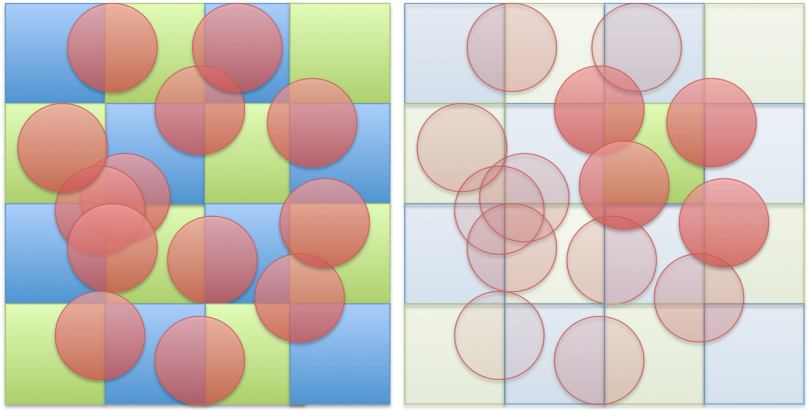
4 Benchmarks and simulations

Particle in cell method, algorithmically



- EM PIC: Maxwell+Lorentz+(Vlasov sampling)
- Particle-grid interactions:
  - Force on particles is interpoladed (averaging) *from fields grids*
  - Particle current/density is deposited *to grid*
    - scatter operation: a particle adds its density value the cells that it overlaps

PIC Deposition algorithm on massively parallel architectures



1. Naive, (1 particle $\longleftrightarrow$ 1 thread) parallelization $\rightarrow$ Race conditions on same memory cells: wrong results
   - Atomic operations or other synchronization methods are required
2. N particles per cell, particle shape function of total order K (4~27)
   - Density grid data is accessed $K \cdot N_{ppc}$ times
   - It's *worth caching* in GPU multiprocessor's shared memory

Caching using shared memory

Many accesses to same memory location →worth caching

Density grid data is accessed $2 * K * N_{ppc}$times

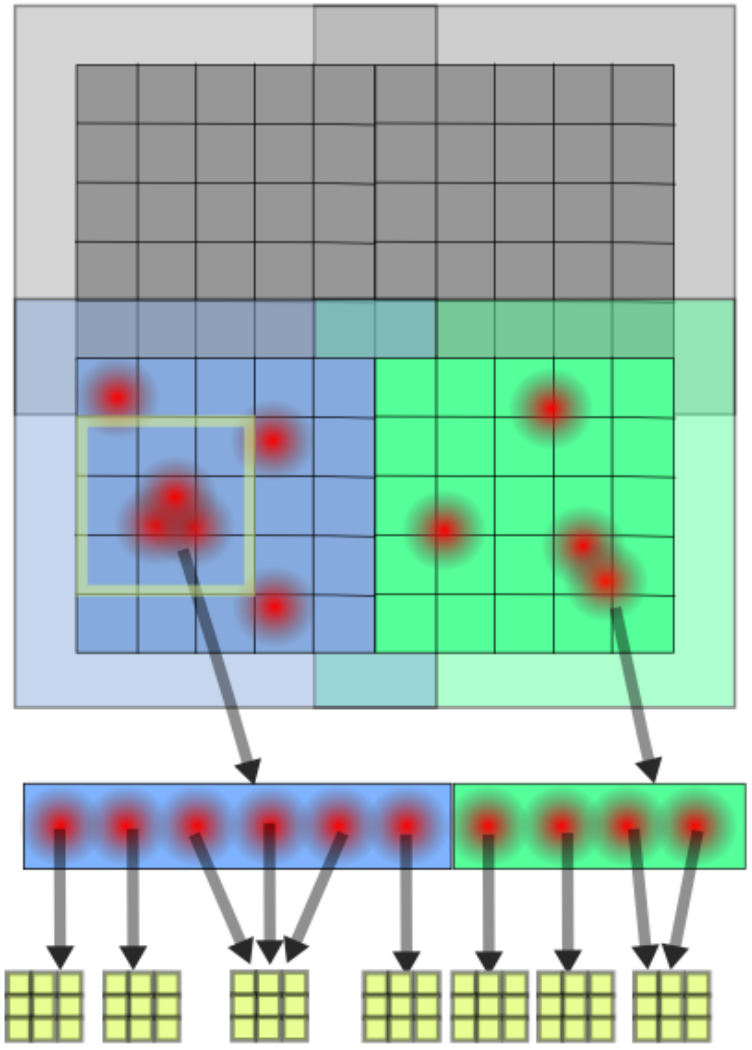
- Particles organized by *grid block* and then *grid cell*
  - One *block* processes particles having *center* inside the grid block
  - Grid block cells + halo (~ half the order of shape function)
- Cache electrical current grid *block* in multiprocessor's shared memory
  - Saves ~$N_{ppc} \times K$ global memory traffic
- Use *segmented reduction* for sums in shared memory
  - Rather than shared memory atomic operations
    - Not natively available for double precision
    - Can be slower than reduction

Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1



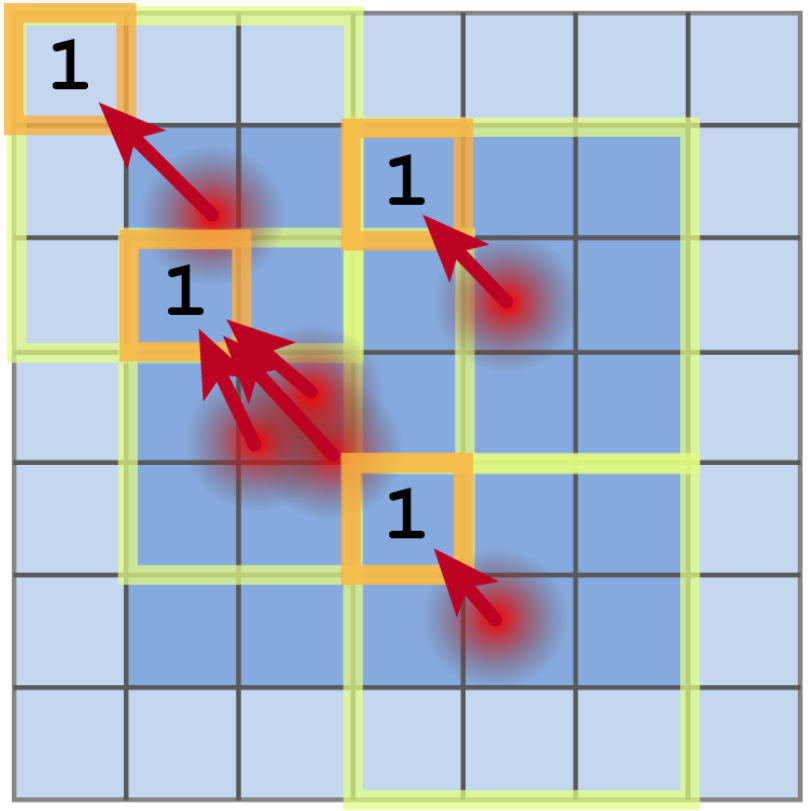
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



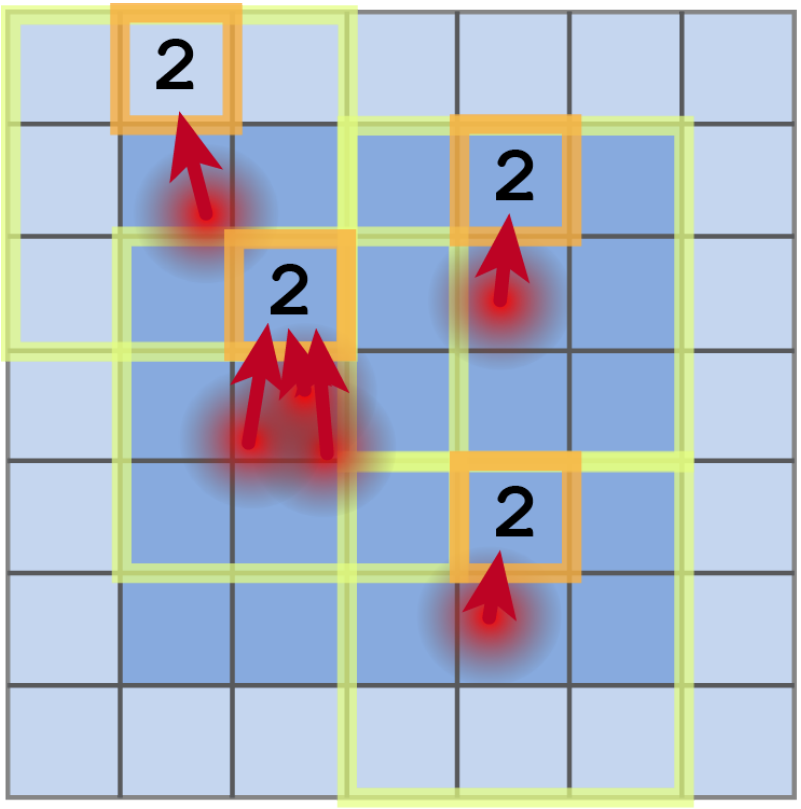
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



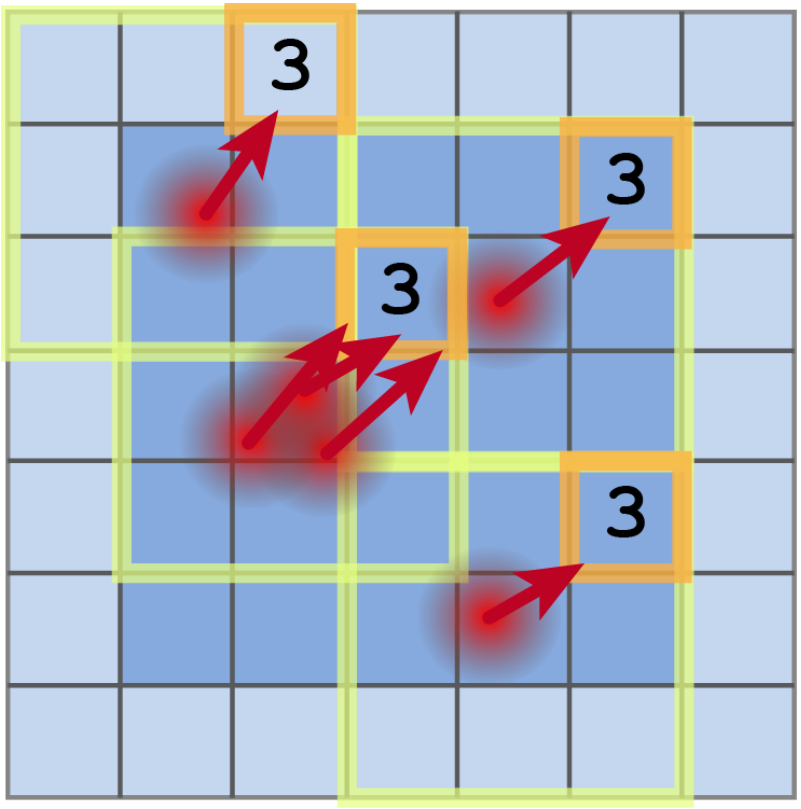
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



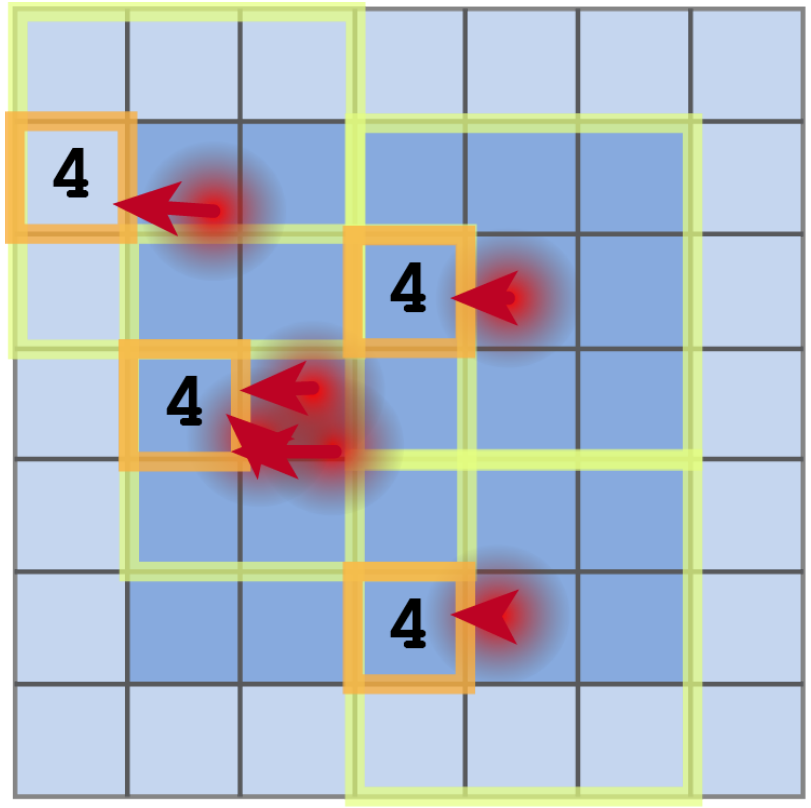
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



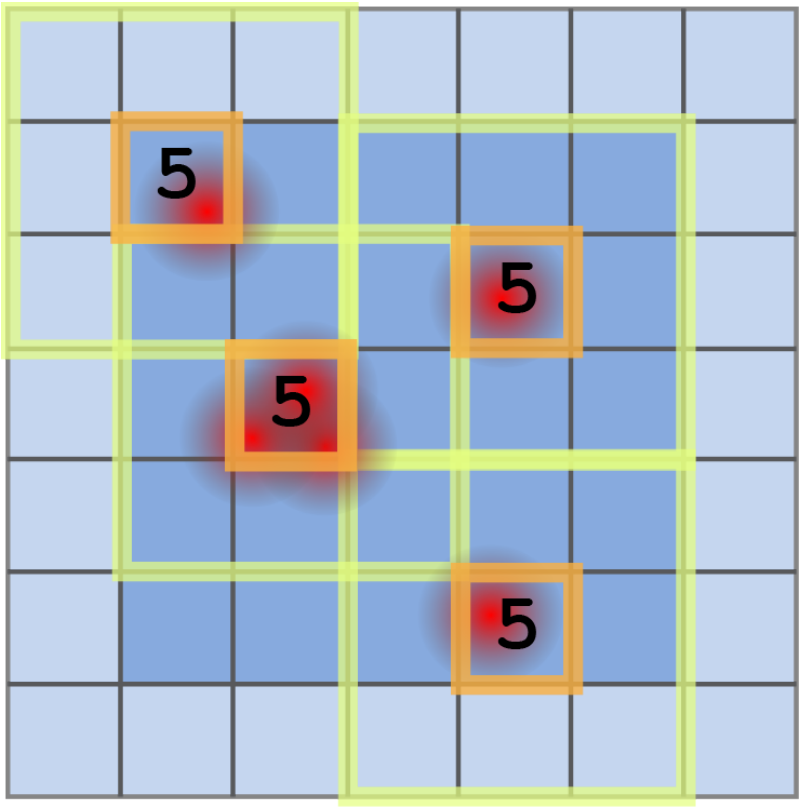
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



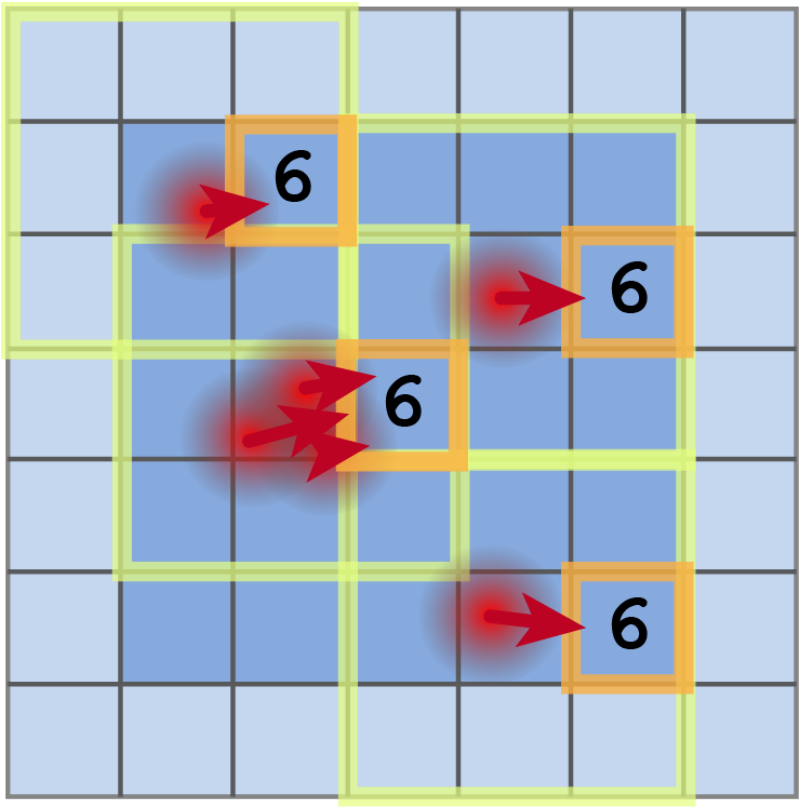
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



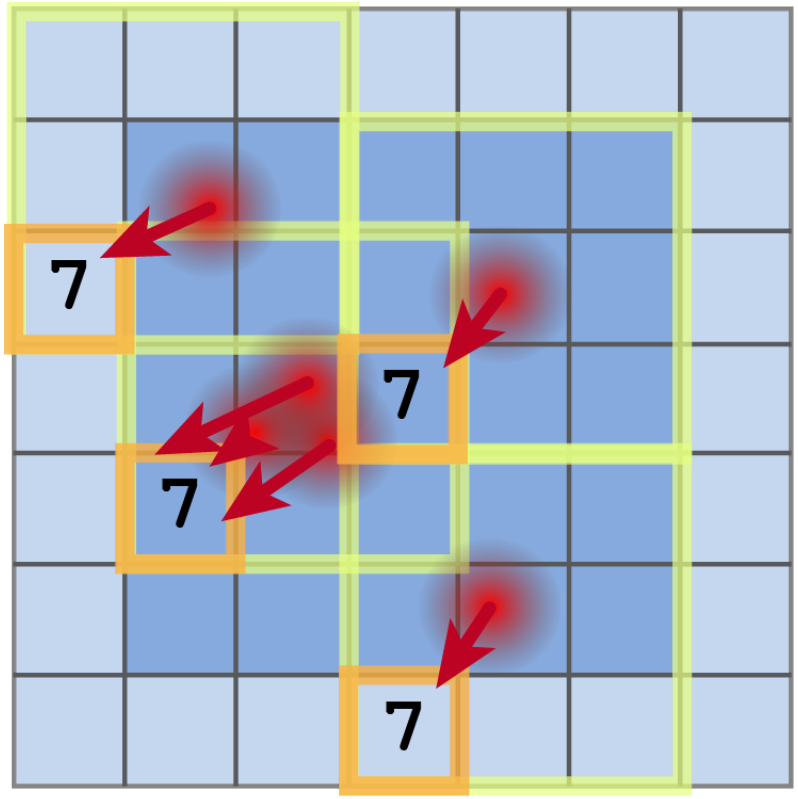
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



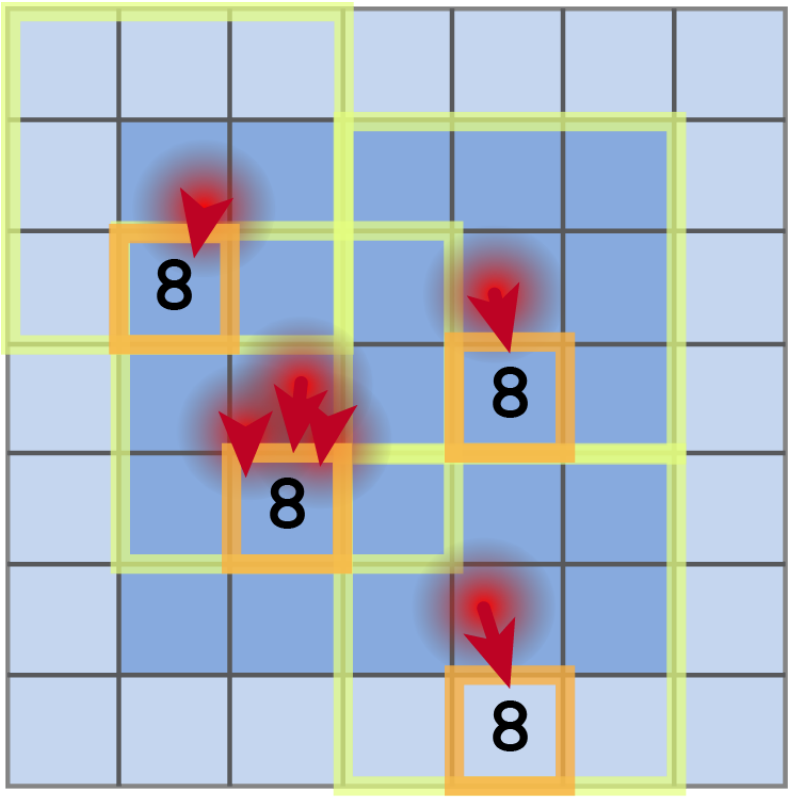
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

Scans:



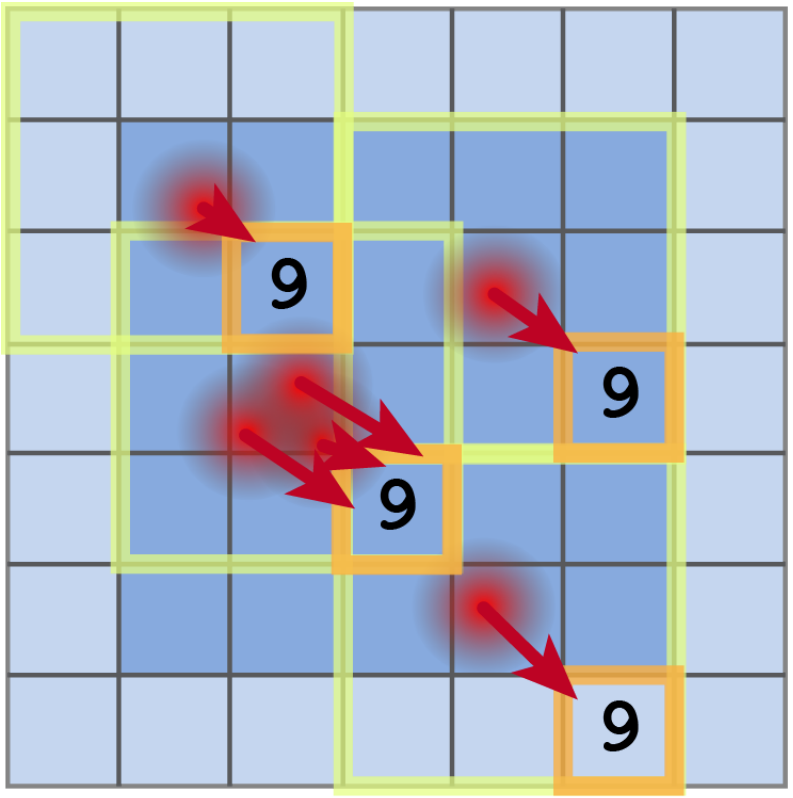
Deposition algorithm without atomics

Algorithm

- Sort by cell block and cell
- Assign a CUDA block to a cell block
- Perform a per-block, shared memory, *segmented scan* to compute density sum for each cell
- Sum cached copy to global grid

Cell start	1	0	0	0	1	0	0	1
Data	1	2	3	4	0	3	3	1
Pass 1	1	3	5	7	0	3	6	1
Pass 2	1	3	5	7	0	3	6	1
Pass 3	1	3	5	7	0	3	6	1

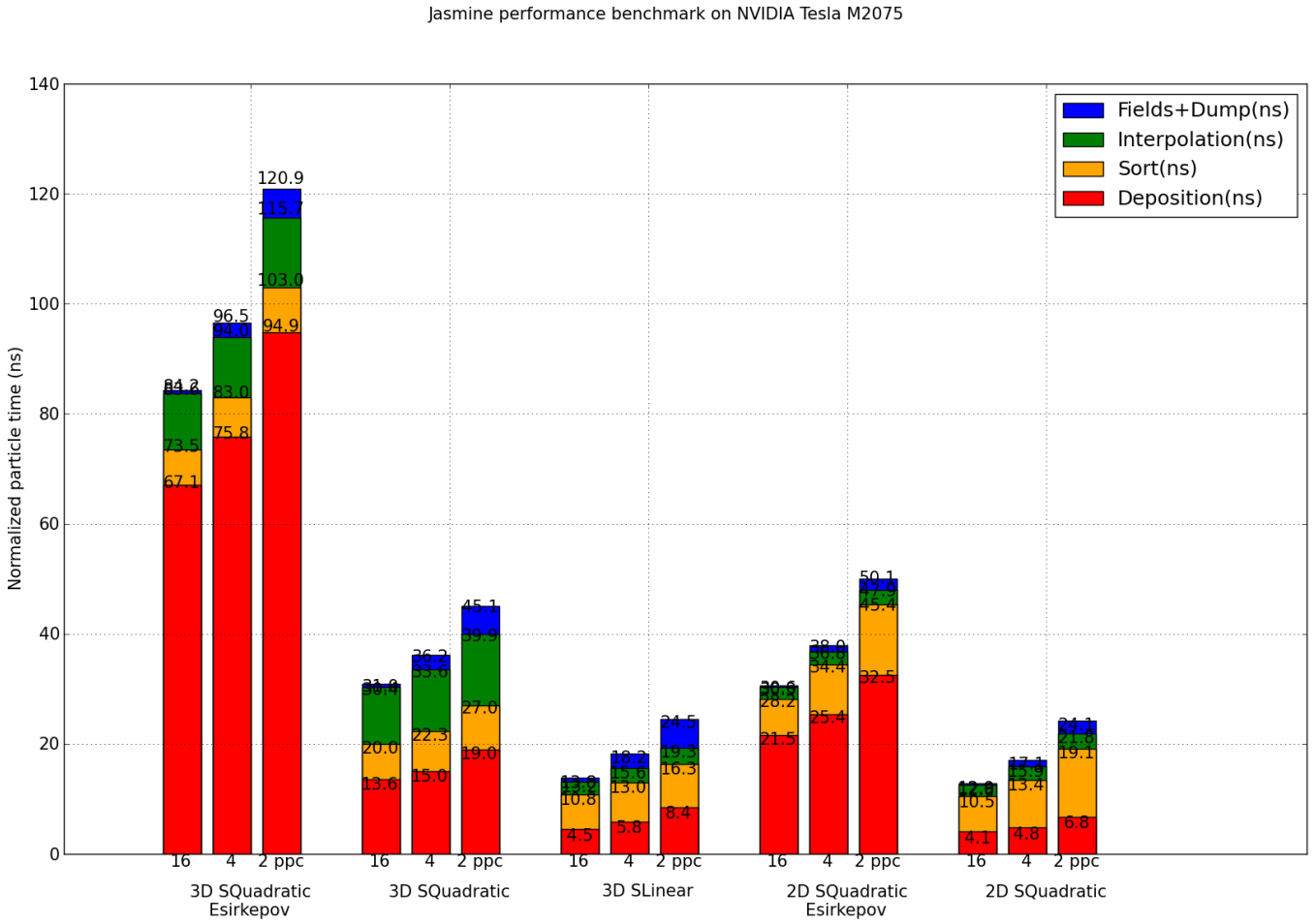
Scans:



Implementation of Esirkepov's method for charge conservation

- Can be implemented with no branches (ideal for GPUs):
  - large overhead: involves many more grid point per particle
  - e.g. for 3D quadratic shape function,$5x5x5 = 125$ instead of$3x3x3$
- With branches:
  - If no particle grid cell crossing $\rightarrow$ no overhead
  - If particle crossing in one direction $\rightarrow$ little overhead
  - 3^D possible cases to choose from!
- But:
  - Intra-warp divergent branches are serialized by multiprocessors
  - Ideally, one would assign particles moving up,down,right to different warps
    - Lots of extra synchronization required
    - There could be not enough particles moving to a particular direction to saturate a warp (3^D cases)
- Quick Fix for explicit simulations with grid stretching (transverse cell. dim ≥ 4):
  - In this case only a few particles cross transverse cell boundaries
  - Static branch: divide particle crossing transversally / not transversally
  - Optimal branch: most of the particles are not crossing cells transversally

Performance



LASER: a=7.7, waist=9.0 mum, fwhm=24.0 fs PLASMA: density=1.00e+19 1/cm^3. Simulation run for $ct = 60\mu m$, double precision. Note: 3D test with Esirkepov method runs stretched grid

Grid-based Esirkepov implementation

- Deposit ρ^{n+1} and compute $\mathbf{J}^{\mathbf{n}+\frac{1}{2}}{}_{n+\frac{1}{2}}$, using ρ^{n+1} . If particles' shape is separable $\hat{\rho} = \hat{\rho}_x \hat{\rho}_y \hat{\rho}_z$, the grid ρ variation per timestep can be decomposed into movement along axes:

$$[\rho^{n+1} - \rho^n] + \Delta t \, D\mathbf{J}^{\mathbf{n}+\frac{1}{2}} \rightarrow \Delta t \, D_{\{x,y,z\}} J_{\{x,y,z\}} = -\Delta\rho_{\{x,y,z\}}$$

- Pros:
  - Almost same cost of a standard, 3 components, current deposition
  - Simpler
  - Extendible to high order schemes (in 2nd order FDTD the spatial operators $D_{\{x,y,z\}}$ are solvable via a simple integral.
- Cons:
  - Auxillary buffer ρ_n must be allocated
- Details in Londrillo et al. "Charge preserving high order PIC schemes", NIMA 620-1, Aug 2010

Performance gain

- *Jasmine* vs *ALaDyn* (our CPU code), same exact simulation.
- Performance of a *single* NVIDIA Fermi GPU equates ~200x BlueGene cores or ~45x IBM SP6 cores.
  - Plus, since subdomains are much larger, load balancing and scalability are much easier
- In the simulation setups shown above (fair resolution), *jasmine* can simulate *~4mm of plasma* per day on a ~24 GPUs cluster

Table of Contents

1 Introduction to GPUs

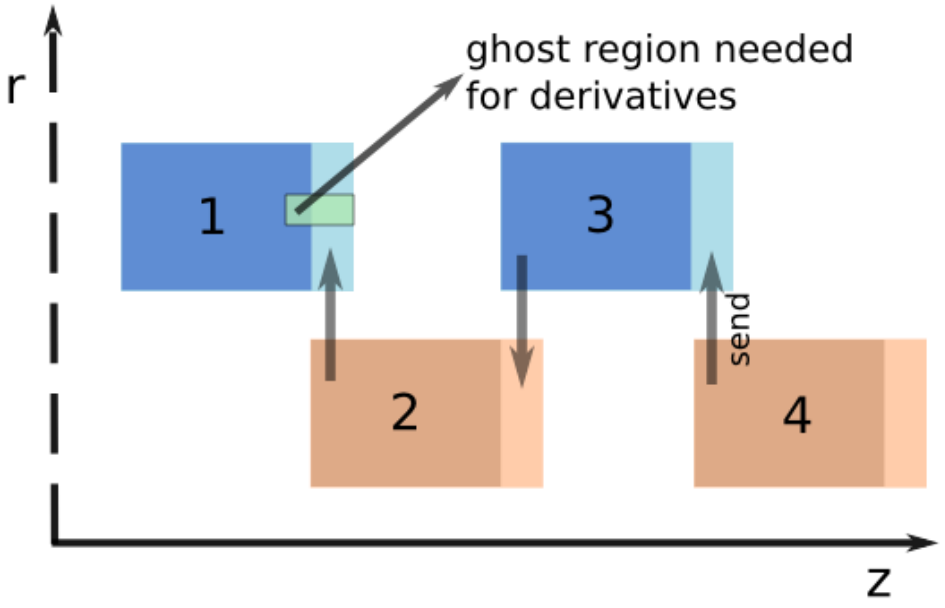
2 GPU PIC current deposition algorithm

3 Multi GPU parallelization

4 Benchmarks and simulations

Scaling to multiple GPUs: Hiding network transfer

- Exchange:
  1. Fields' halos
  2. Particles leaving subdomain
- Cluster nodes communicate using standard MPI



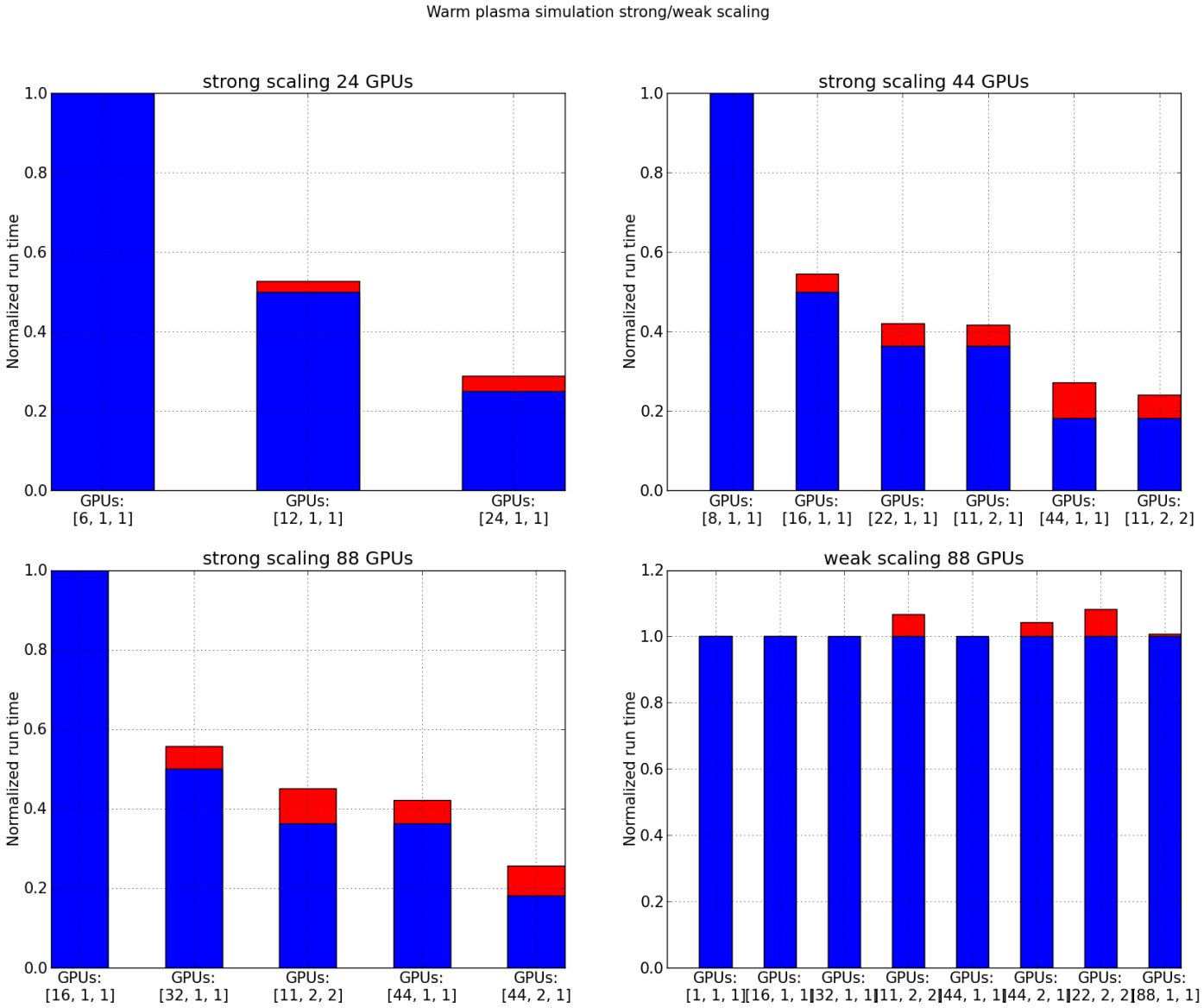
Scaling to multiple GPUs: Hiding network transfer

- Exchange:
  1. Fields' halos
  2. Particles leaving subdomain
- Cluster nodes communicate using standard MPI

- Transfer particles concurrently with current deposition.
  - Communication can be hidden almost completely.

- Scalability test: warm plasma simulation on INFN APE cluster @ "ULa Sapienza" and PLX machine @ CINECA

Scaling


Warm plasma simulation strong/weak scaling
strong scaling 24 GPUs
strong scaling 44 GPUs
strong scaling 88 GPUs
weak scaling 88 GPUs
Normalized run time
GPUs: [6, 1, 1]
GPUs: [12, 1, 1]
GPUs: [24, 1, 1]
GPUs: [8, 1, 1]
GPUs: [16, 1, 1]
GPUs: [22, 1, 1]
GPUs: [11, 2, 1]
GPUs: [44, 1, 1]
GPUs: [11, 2, 2]
GPUs: [16, 1, 1]
GPUs: [32, 1, 1]
GPUs: [11, 2, 2]
GPUs: [44, 1, 1]
GPUs: [44, 2, 1]
GPUs: [1, 1, 1]
GPUs: [16, 1, 1]
GPUs: [32, 1, 1]
GPUs: [11, 2, 2]
GPUs: [44, 1, 1]
GPUs: [44, 2, 1]
GPUs: [22, 2, 2]
GPUs: [88, 1, 1]

Simple algorithm for load balancing

- The particle motion easily leads to inhomogeneous distribution of the load
- Shrink the volume of heavy-loaded nodes:
  - Each few timesteps select a subdomain (and its row) and the direction where to shrink
  - Subdomains topology remains intact (vertices conservation)
  - Choice is done trying to minimize the cost function:

$$k_1 \times Max(Load)/Average(Load) + k_2 \times Variance(Load)/Average(Load)$$

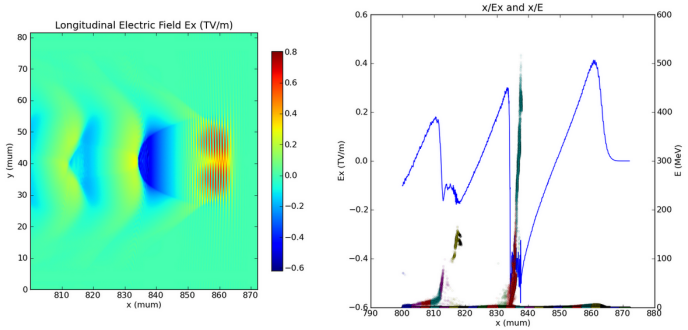
Load balancer test case

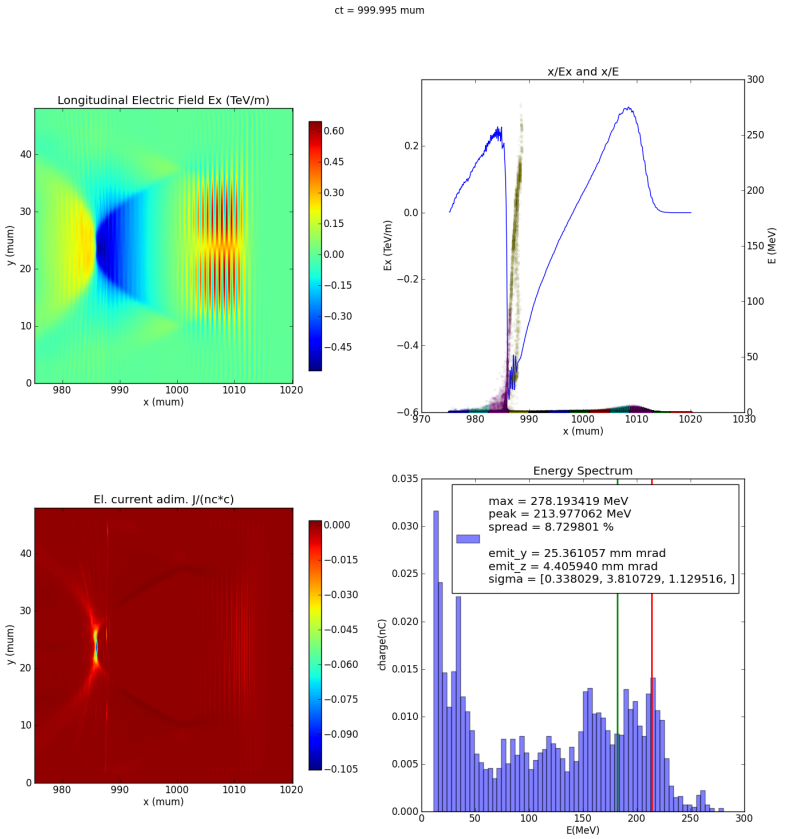
Setup (coarse test)

LASER: a=5.8, waist=13.2 mum, fwhm=24.0 fs

PLASMA: density=3.80e+18 1/cm^3

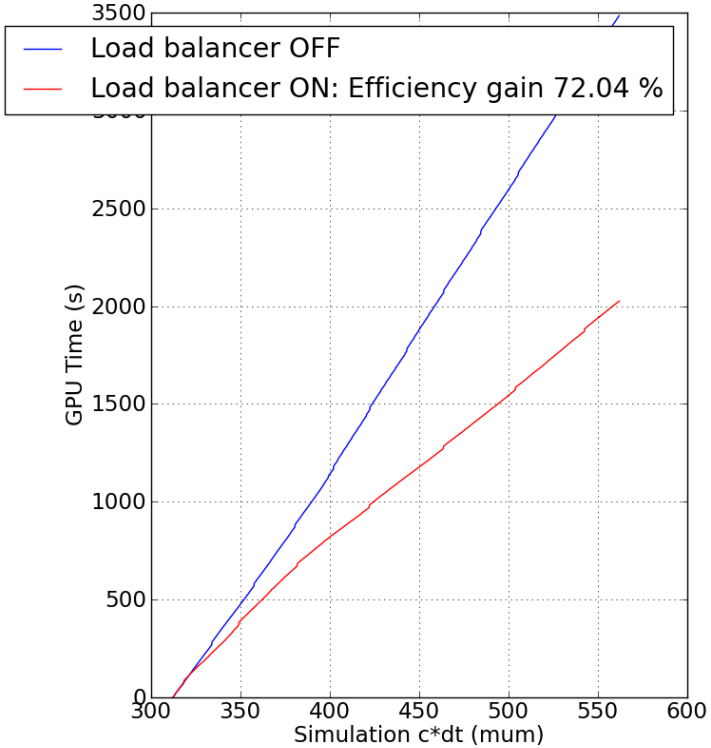
GRID: n=[729, 96, 96], dx=['6.25e-02', '5.00e-01', '5.00e-01'] mum

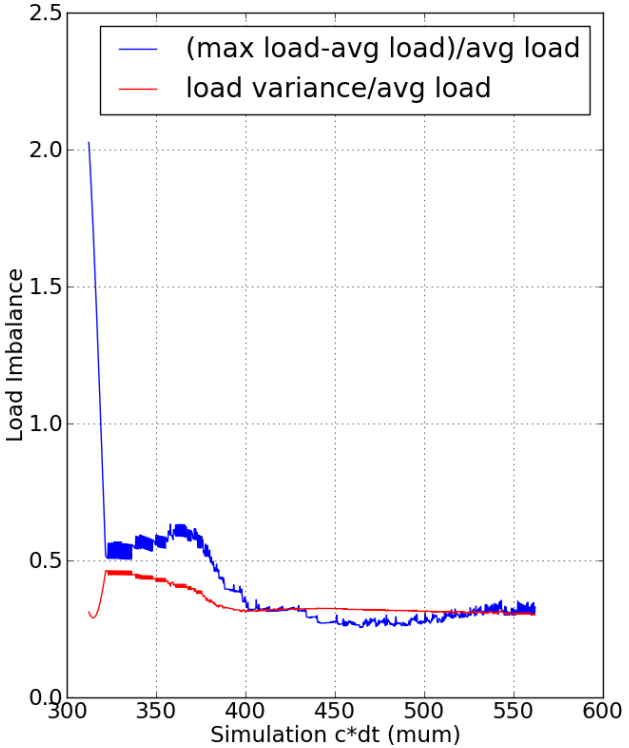




Test case: scaling with load balancing

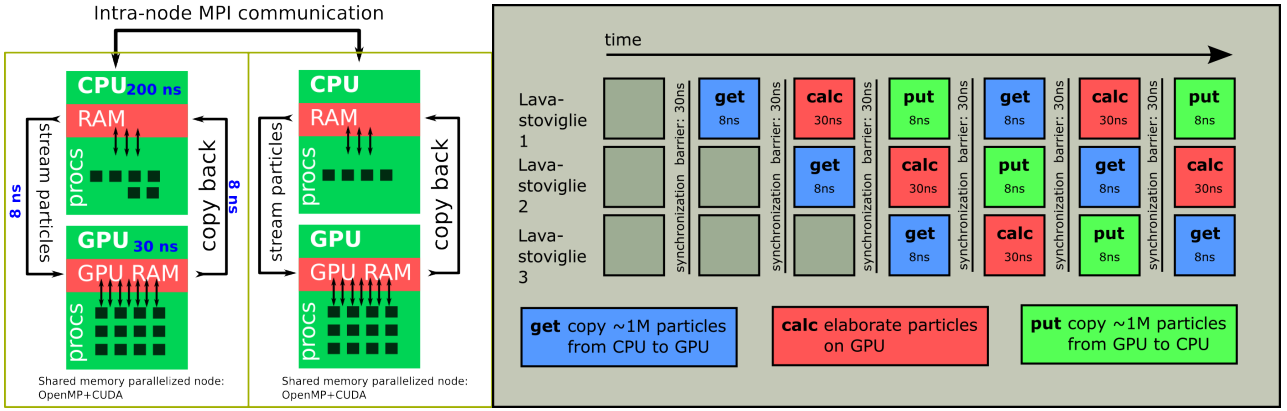
- With 72 subdomains:





Memory intensive simulations

- Total memory availability represents a constraint for many simulations (for example ion acceleration ones).
  - In a node, GPU memory is often much less than the total host memory available
  - Using host memory to store simulation data make larger simulations possible on a cluster of fixed size
- Asynchronous stream of particle chunks stored in main CPU memory overlapped with computation using CUDA streams
  - Slower but no longer memory bound to the GPU device memory
  - Currently in testing stage



Implementation note: Meta-programming

- Meta-programming can be used:
  - for writing maintainable code for all particle weighting / numerical schemes
  - tweak parameters for optimization of each case
  - implementing different numerical schemes using the same core algorithms (deposition and intepolation)
- Attempts:
  1. C++ template meta-programming
  2. Python-based code template engine (code becomes more linear, but non standard)
- Python also used for simulation initial conditions definition, plotting (numpy+matplotlib) and basic automated data-analysis

Meta-programming: examples

- Particle deposition: unrolled loops in particle shape, warp scans and compiler-computed shared memory offsets.
- Preprocessing for scan integrals for grid-based Esirkepov method: generic n-dimensional, memory access efficient, grid transpose kernel to move the integration coordinate to the fastest index.

Table of Contents

1 Introduction to GPUs

2 GPU PIC current deposition algorithm

3 Multi GPU parallelization

4 Benchmarks and simulations

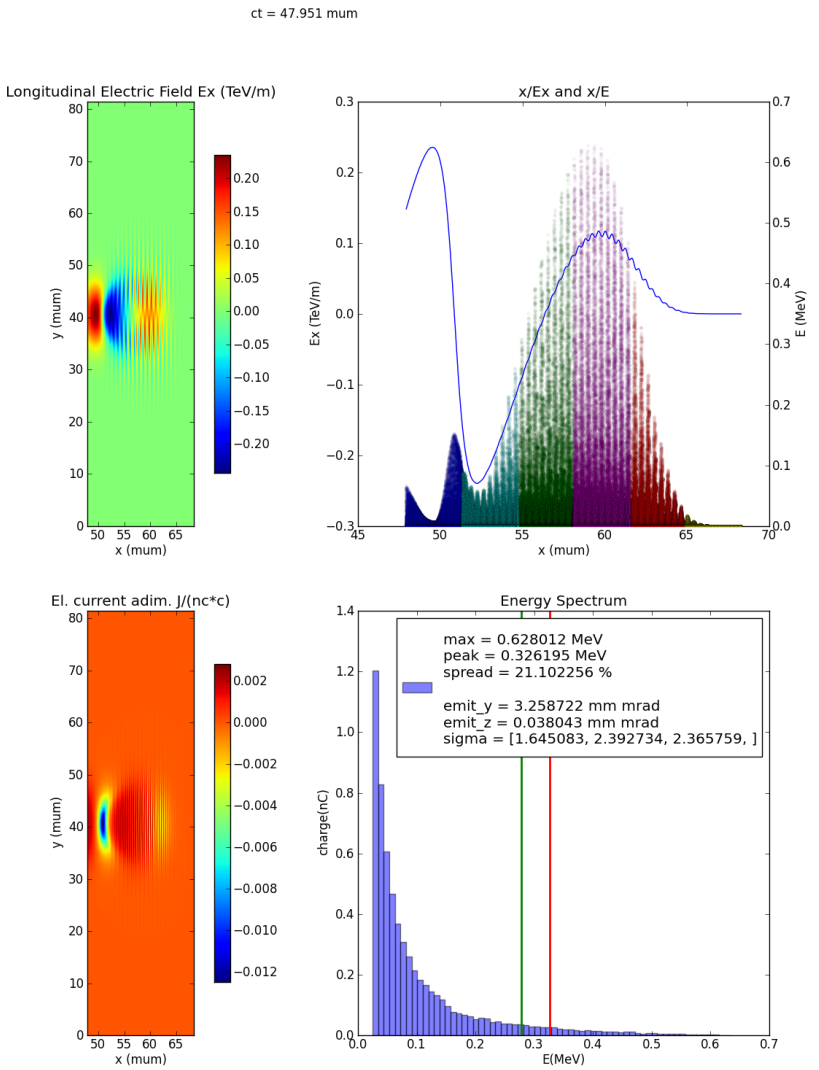
LWFA benchmark simulation

Setup

LASER: a=2.0, waist=8.2 mum, fwhm=21.0 fs

PLASMA: density=1.38e+19 1/cm^3

GRID: n=[512, 256, 256], dx=['4.00e-02', '3.18e-01', '3.18e-01'] mum



LWFA benchmark simulation

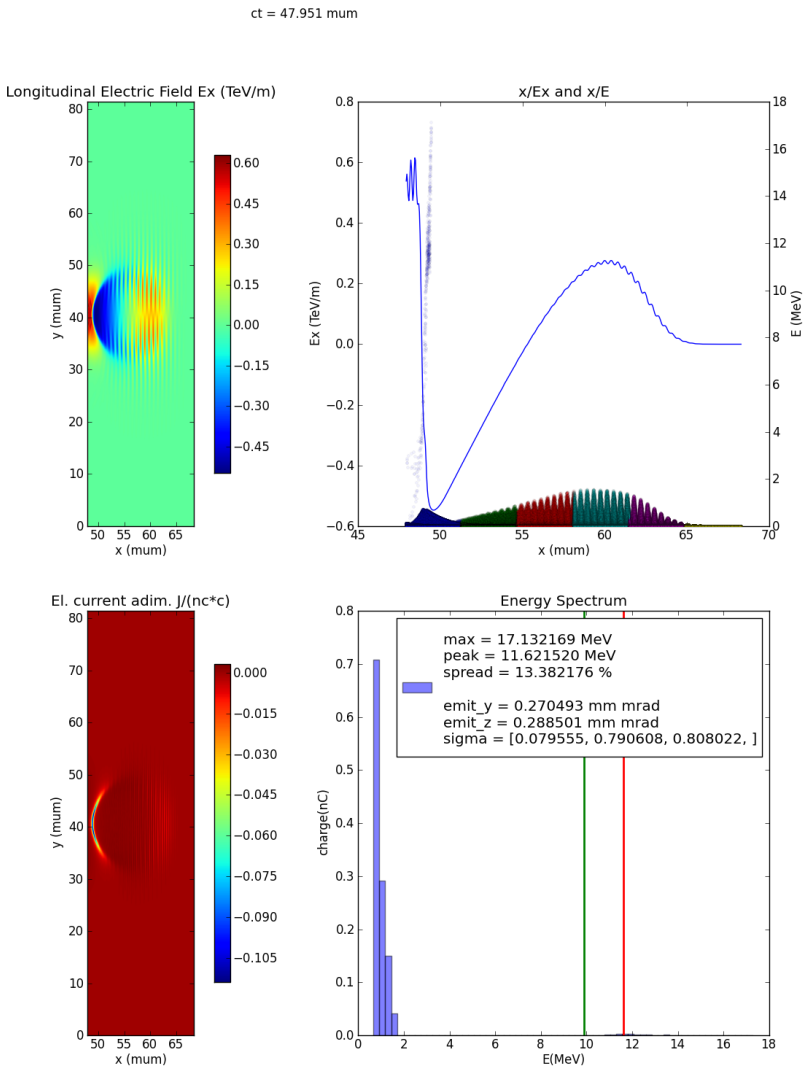
Setup

Parameters from Paul et al. *Benchmarking the codes VORPAL, OSIRIS, and QuickPIC with Laser Wakefield Acceleration Simulations,* AIP Conference Proceedings;1/22/2008

LASER: a=4.0, waist=8.2 mum, fwhm=21.0 fs

PLASMA: density=1.38e+19 1/cm^3

GRID: n=[512, 256, 256], dx=['4.00e-02', '3.18e-01', '3.18e-01'] mum



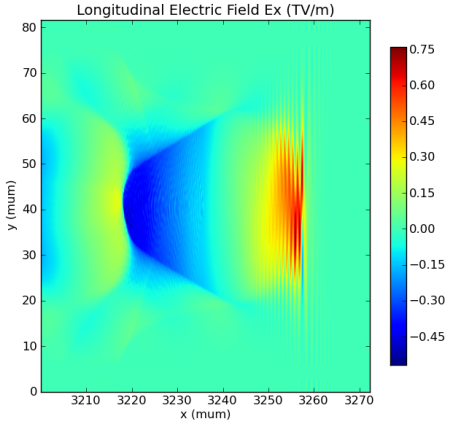
SPARCLab electron acceleration simulation

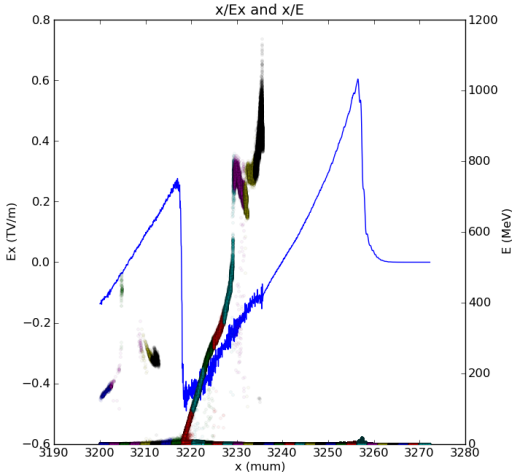
Setup

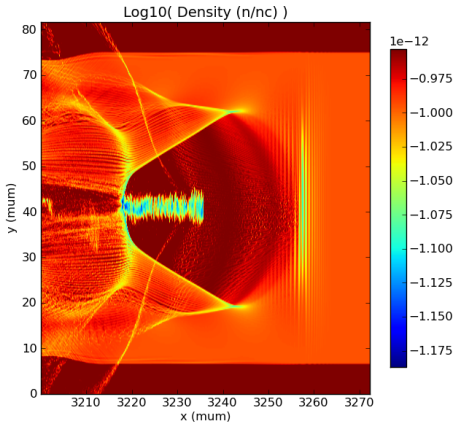
LASER: a=4.9, waist=15.5 mum,

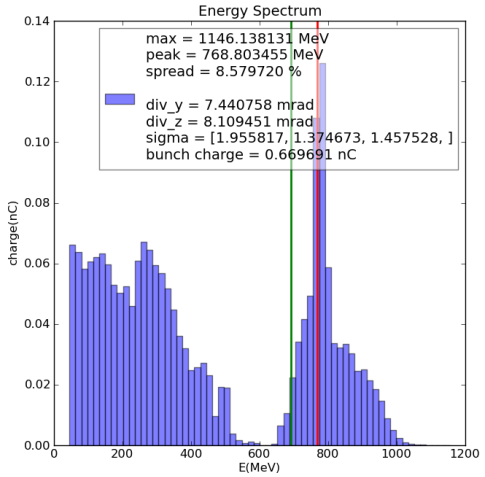
fwhm=30.0 fs

PLASMA: density=3.0e+18 1/cm^3









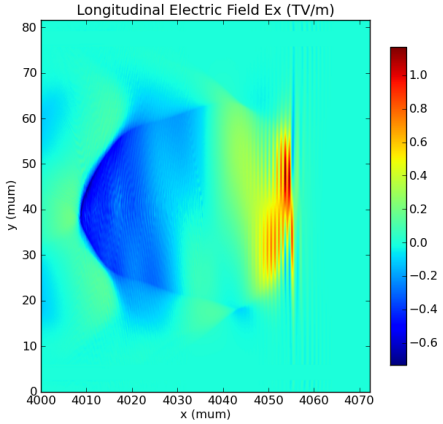
SPARCLab electron acceleration simulation

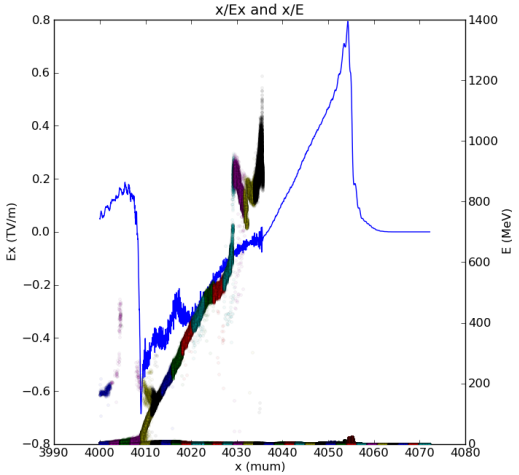
Setup

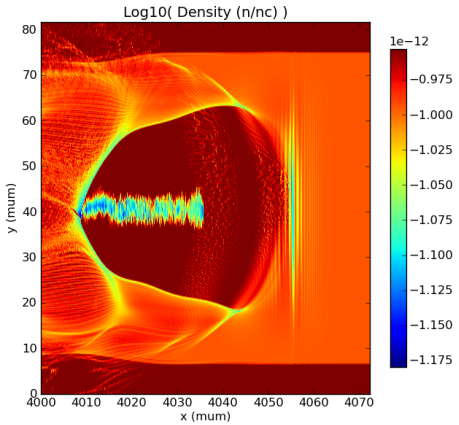
LASER: a=4.9, waist=15.5 mum,

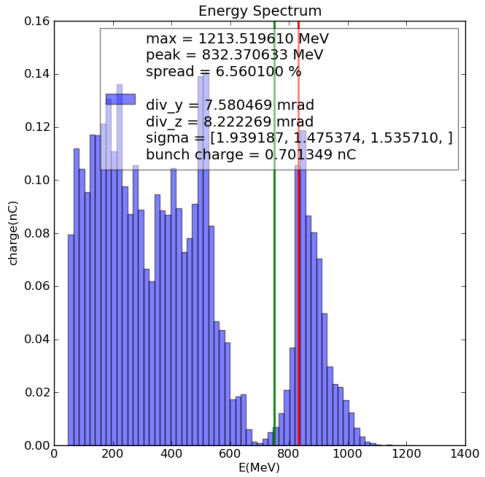
fwhm=30.0 fs

PLASMA: density=3.0e+18 1/cm^3









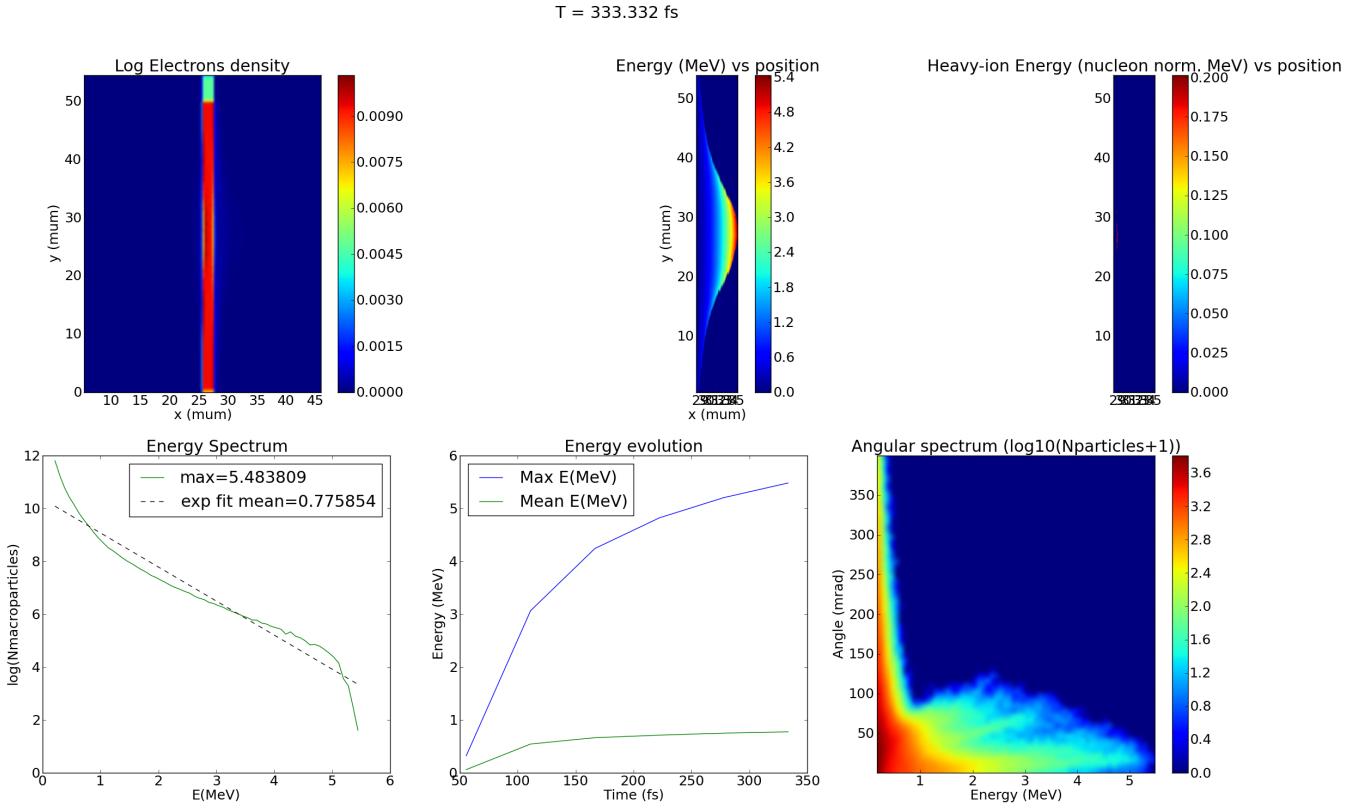
TNSA ion acceleration: Frascati Flame parameters

Setup

LASER: a=7, waist=10.0 mum, fwhm=30.0 fs

TARGET: 2.0μm thick

Double layer: aluminium, (n/nc = 100) + back side contaminants layer



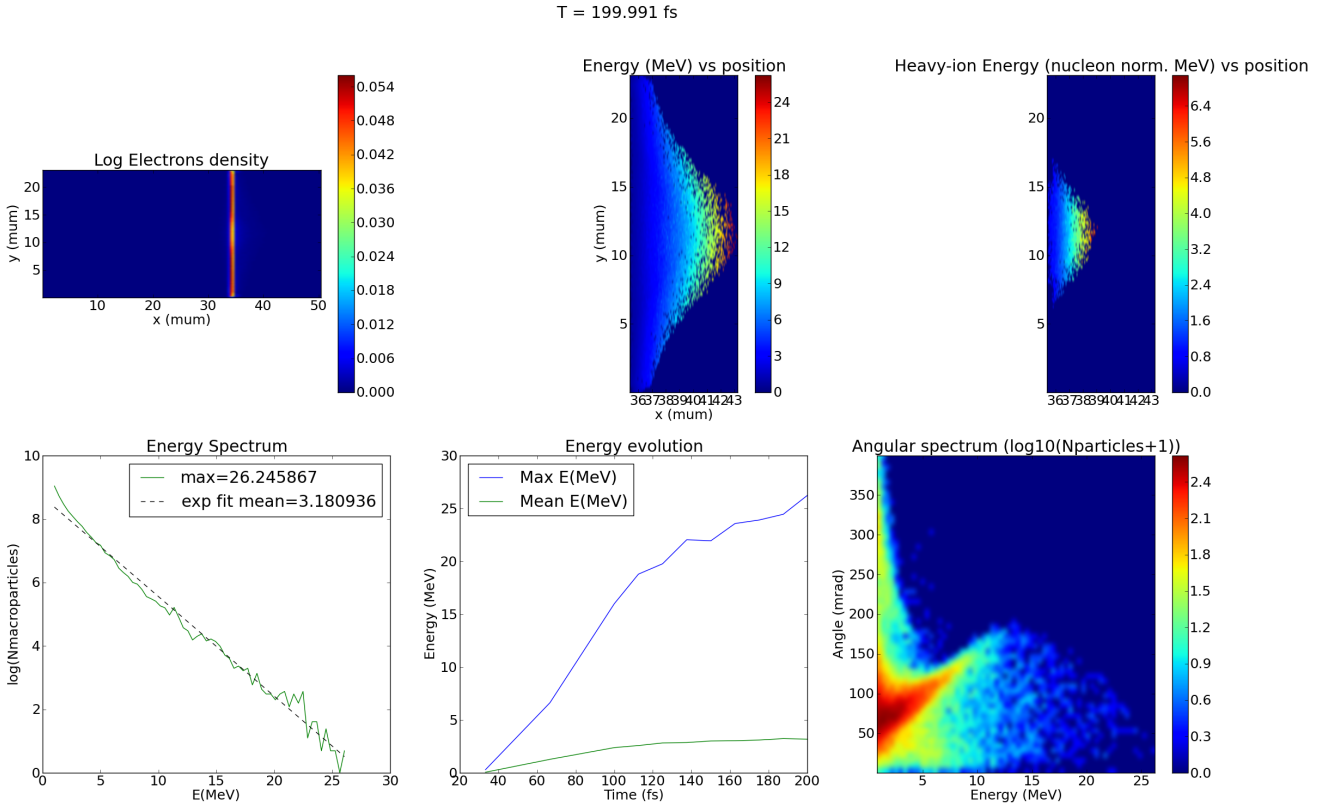
TNSA ion acceleration: Nara-like parameters

Setup

LASER: a=22, waist=3.5 mum, fwhm=40.0 fs

TARGET: 0.8μm thick

Double layer: aluminium (n/nc = 60) + back side contaminants layer



Future work

- Performance tuning for Kepler architecture
- Implement more accurate and/or optimized numerical schemes.
  - GPUs alone are not of enough for satisfying all the computational requirements of the experimental groups (e.g. simulating a 10GeV electron acceleration stage).
- Lot of work to do!

References

1. F.Rossi et al., Towards robust algorithms for current deposition and dynamic load-balancing in a GPU particle in cell code. AIP Conference Proceedings Vol. 1507 Issue 1, p184
2. Birdsall,Langdon. Plasma physics via computer simulation.
3. Benedetti,Sgattoni,Turchetti,Londrillo. ALaDyn: A High-Accuracy PIC Code for the Maxwell-Vlasov Equations. IEEE Transactions on Plasma Science,36(4),2008.
4. Burau,Widera,Hönig,Juckeland,Debus,Kluge,Schramm,Cowan,Sauerbrey,Bussmann. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster, IEEE Transactions on Plasma Science 38(10)
5. Abreu,Fonseca Pereira,Silva. PIC Codes in New Processors:A Full Relativistic PIC Code in CUDA-Enabled Hardware With Direct Visualization.IEEE Transactions on Plasma Science,vol.39,issue 2
6. Kong,Huang,Ren,Decyk.Particle-in-cell simulations with charge-conserving current deposition on graphic processing units. J.Comput.Phys.230,4(February2011)
7. Sengupta,Harris,Zhang,Owens.Scan primitives for GPU computing.In Graphics Hardware 2007, Aug.2007.
8. Hoberock,Bell.Thrust:A Parallel Template Library