Highly Parallel GPU-based Particle-in-Cell/MCC Plasma Simulation

ZOLTAN JUHASZ[1], PETER HARTMANN[2] AND ZOLTAN DONKO[2]

[1]DEPT. OF ELECTRICAL ENGINEERING AND INFORMATION SYSTEMS, UNIVERSITY OF PANNONIA, VESZPREM, HUNGARY

[2]DEPT. OF COMPLEX FLUIDS, INSTITUTE FOR SOLID STATE PHYSICS AND OPTICS, WIGNER RESEARCH CENTRE FOR PHYSICS, BUDAPEST, HUNGARY

OVERVIEW

Intro to plasma simulation

Structure of sequential execution

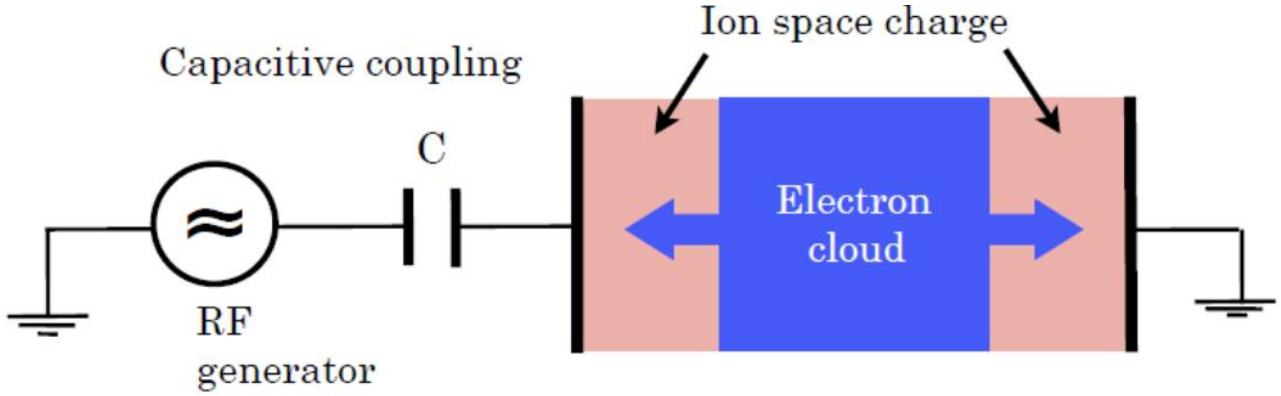
Parallel execution strategies

GPU implementation

Performance turning

Current results and conclusions

PLASMA SIMULATION



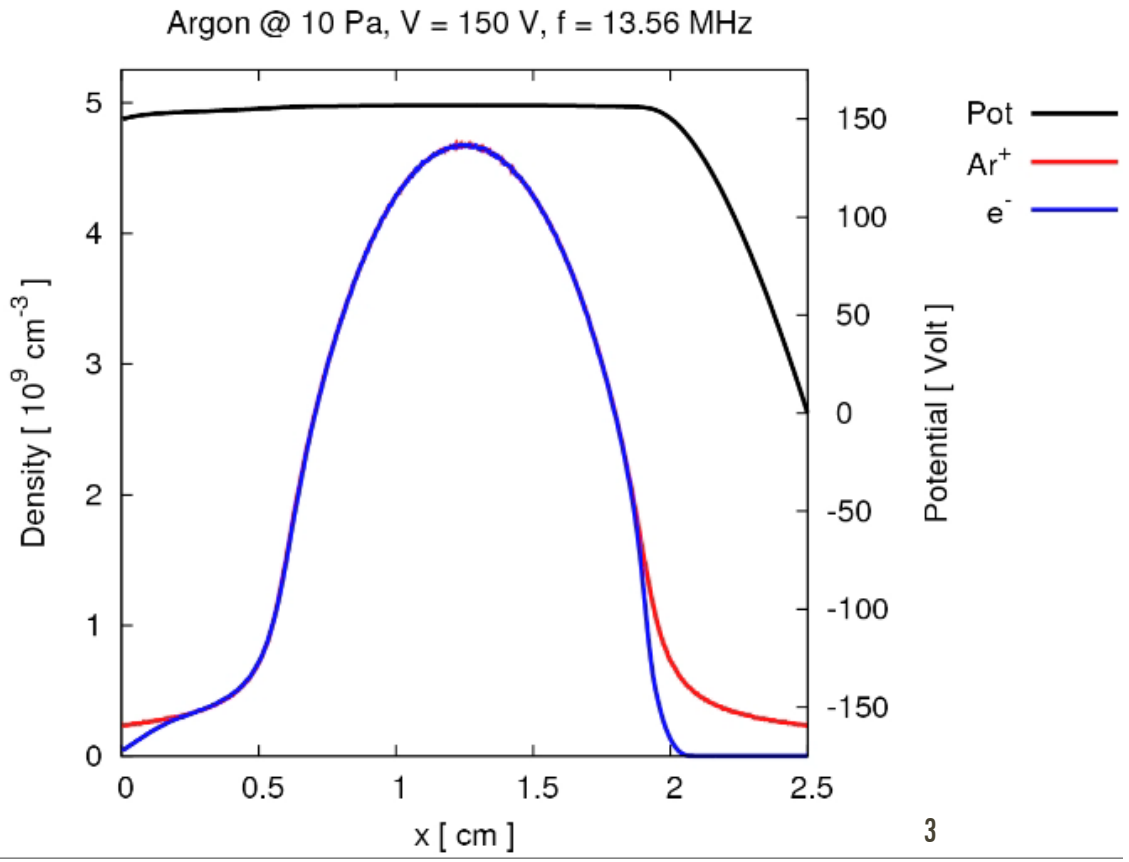
Understanding capacitively coupled radiofrequency discharges in plasma

Spatiotemporal changes in electric field

Non-equilibrium transport of particles

Numerical simulation helps to understand the behaviour of particles

Uses kinetic theory for describing particle movement



PLASMA SIMULATION

Particle-in-Cell simulation method, Monte-Carlo collisions

Particles interact via the electric field

Sequential program written in C, verified and validated by real experiments

Total simulation execution time varies from **hours** to **days** (even **several weeks**)

Loop for simulation cycles (1000-3000)

Loop for input samples (800)

1. move electrons	2. check boundaries	3. electron collision	4. electron density	5. move ions	6. check boundaries	7. ion-electron collision	8. ion density	9. Poisson solver

PARALLEL APPROACHES, EXPECTATIONS

Per step execution time: **3.1-5.4 msec**, total time **3100+ seconds**

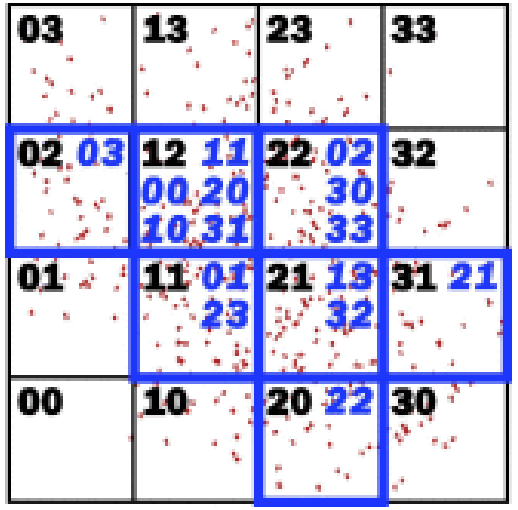
Outer loops strictly sequential, particle-level ops can be parallel

GOAL: minimum 10x speedup (310-540 usec)

Where and how to start?

- Keep existing program intact, no re-write, minimal alteration, ideally incremental changes
- OpenMP (ideal candidate from SW Eng point of view) ?
- GPUs ?

GPU STRATEGY: ISSUES TO CONSIDER



CPU-GPU interaction, division of work

How to map problem to kernels?

Many small kernels or few large kernels?

How to port/parallelise existing code?

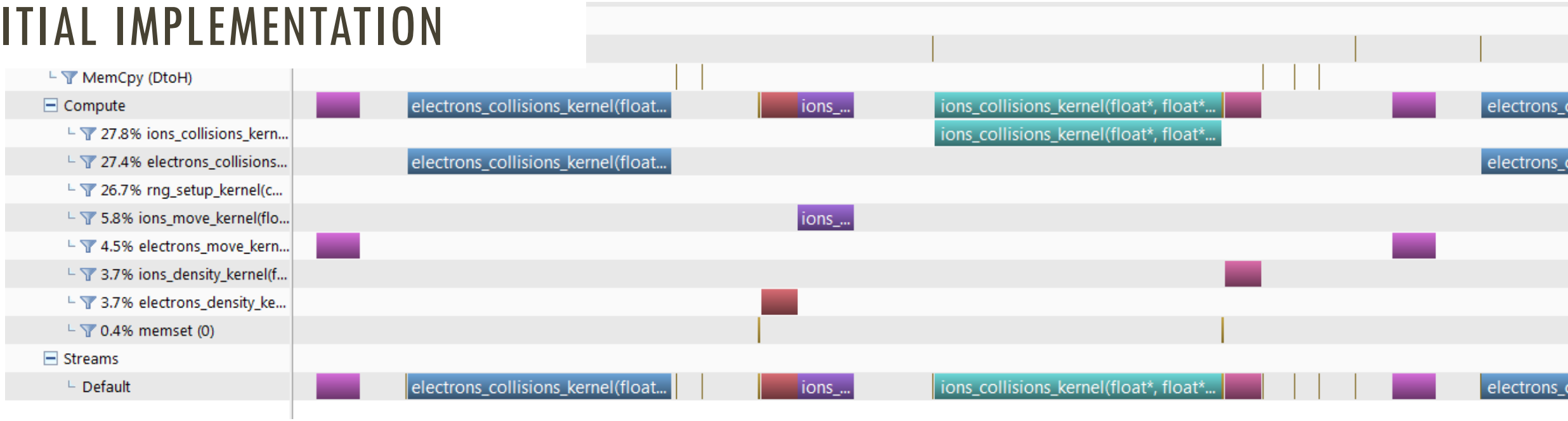
- how many threads to use? – PIC cell geometry!!!
- max # of resident threads? **14 336** (Kepler-3.0), **32 768** (Maxwell-5.2), **81 920** (Pascal-6.1)

Extreme scale processing – not necessarily a straight path from sequential code

Data transfer: host-device, device to chip

Other optimisation techniques?

INITIAL IMPLEMENTATION



One kernel for each step of the simulation cycle

Each kernel:

- reads input data
- compute/modify
- store results

Potential performance problems

- too many kernel launches
- low compute intensity
- host-device data transfer
- latency issues
- memory conflicts
- no kernel concurrency
- data structure updates
- random number generation
- Poisson solver

EXECUTION PROFILING (10 X 800 STEPS)

nvprof output (GTX 1080)

Time(%)	Time	Calls	Avg	Min	Max	Name
22.56%	329.65ms	8000	**41.206**us	38.913us	49.154us	electrons_density_kernel
18.85%	275.57ms	8000	**34.446**us	33.792us	39.937us	ions_density_kernel
13.22%	193.28ms					_collisions_kernel
12.88%	188.27ms					sions_kernel
11.68%	170.69ms					e_kernel
10.76%	157.20ms	8000	**19.650**us	18.208us	204.81us	electrons_move_kernel
4.63%	67.716ms	24132	2.8060us	0ns	27.379ms	[CUDA memcpy HtoD]
3.70%	54.091ms	32010	1.6890us	1.0240us	8.1930us	[CUDA memset]
1.71%	25.062ms	40102	624ns	256ns	39.905us	[CUDA memcpy DtoH]

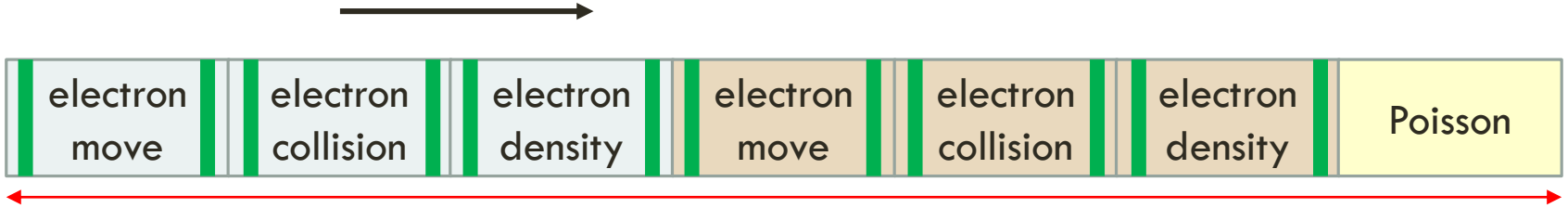
20x speedup on GTX 1080
from 1 hour down to 3 minutes

200-600 usec, depending on GPU generation (Kepler, Maxwell, Pascal)

Analysis with the Profiler

- **Relevant profiler counters:**
  - instructions_issued
    - Incremented by 1 per warp, counter is for one SM
  - dram_reads, dram_writes
    - Incremented by 1 per 32B access to DRAM
    - Note that the VisualProfiler converts each of the above to 2 counters
      - These simply get added together, refer to the Visual Profiler User Guide for details
      - You'll need to do this yourself if you're using command-line profiling
  - If your code hits in L2 cache a lot, you may want to look at L2 counters instead (accesses to L2 are still expensive compared to arithmetic)
- **Compute instruction:byte ratio and compare to the balanced one:**
  - (number of SMs) * 32 * instructions_issued : 32B * (dram_reads + dram_writes)
- **Example: vector add**
  - 1.49:1, lower than 3.76 so memory-bound

PERFORMANCE TUNING: KERNEL EXECUTION



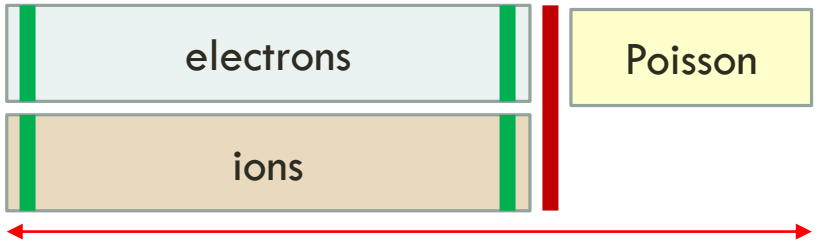
fused kernels

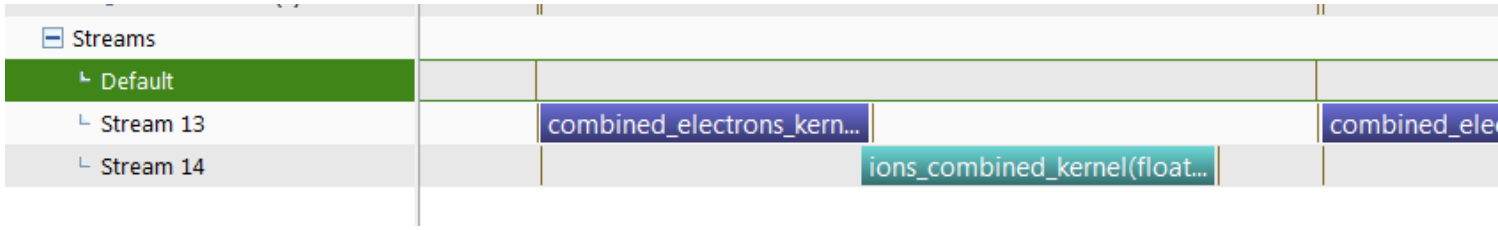
electrons improvement:

625.5 usec → 520 usec

1.2x faster on a K2000M

concurrent kernels





ITERATIVE IMPROVEMENTS

Analyse what to change, why – Nsight, Visual Profiler, nvprof

Check effects of occupancy (block/grid size), instruction latency, use of concurrent kernels, streams, etc.

Instruction pipeline – more particles per kernel, unrolled loops

Use of shared memory – electric field, can be read by all threads in a block without memory bank conflict

Data structure manipulation – removing/inserting particles; not trivial on GPU!

Random number generation – cuRAND or not cuRAND?

Density calculation – 'histogram' code using Maxwell shared mem atomic support

Poisson solver – CPU or GPU?

1D VERSION RESULTS

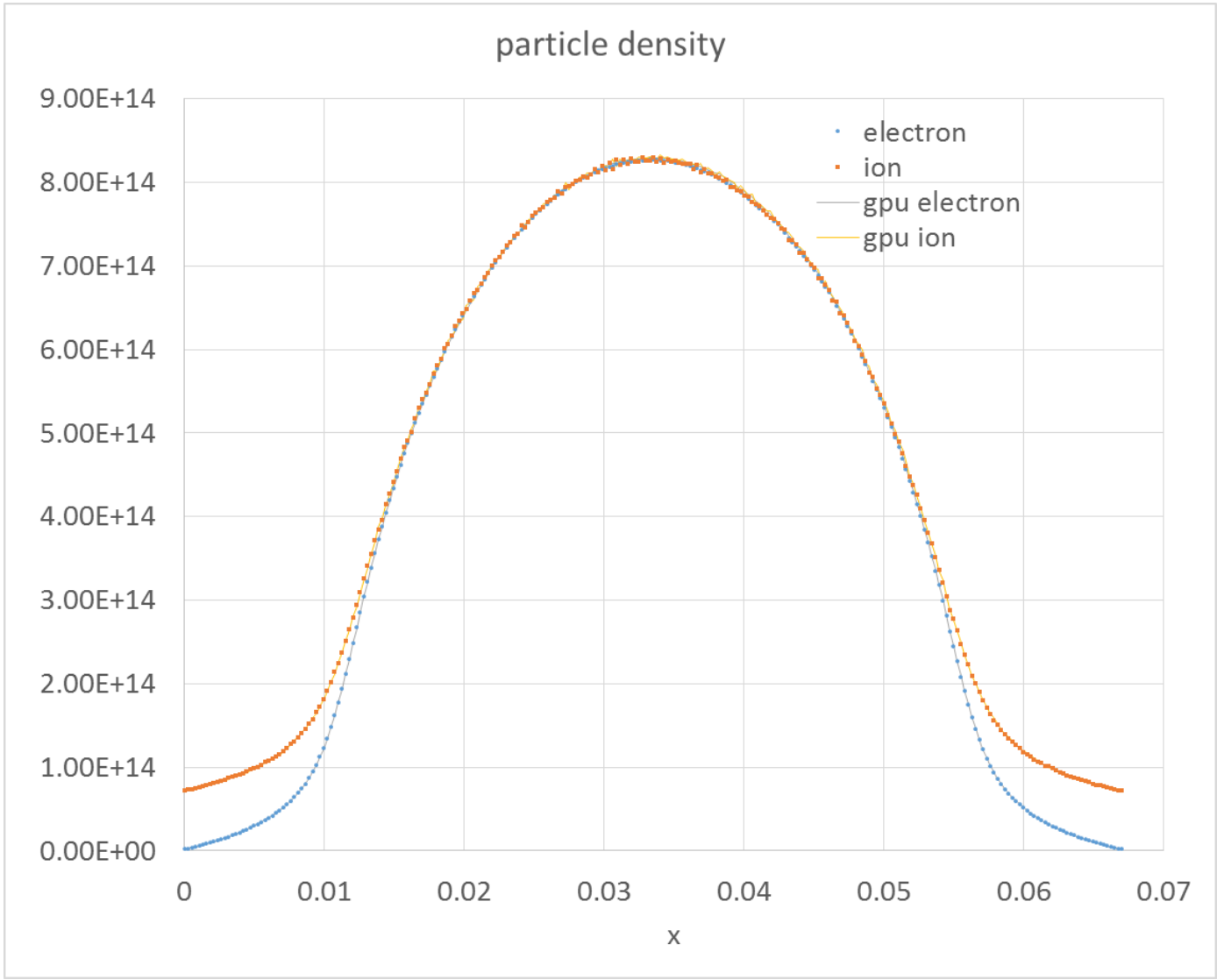
20x speedup compared to original code

- 1000 cycles – 3 minutes
- in-kernel optimisations
- new random number generator

Accuracy is satisfactory

Further optimisation is in progress

- program-level optimisation
- fused and concurrent kernels
- new GPU Poisson solver



CONCLUSIONS

Think how to re-design algorithms for extreme scale parallelism (100k-1million threads)

Architecture internals and versions are important (Kepler, Maxwell, Pascal)

- better memory performance
- improved atomics (global and shared)

1D simulation performance better than expected (20x)

PLUS, first 2D version is operational

- 50 million particles
- 90x speedup
- 1 day instead of 90 days
- may improve eve more with further optimisations