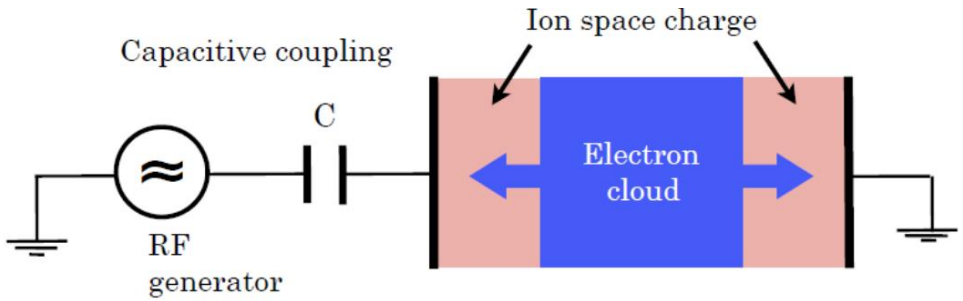
HIGH-PERFORMANCE IMPLEMENTATION TECHNIQUES OF CUDA-BASED 1D AND 2D PARTICLE-IN-CELL/MCC PLASMA SIMULATIONS

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

PLASMA SIMULATION



Understanding capacitively coupled radiofrequency discharges in plasma

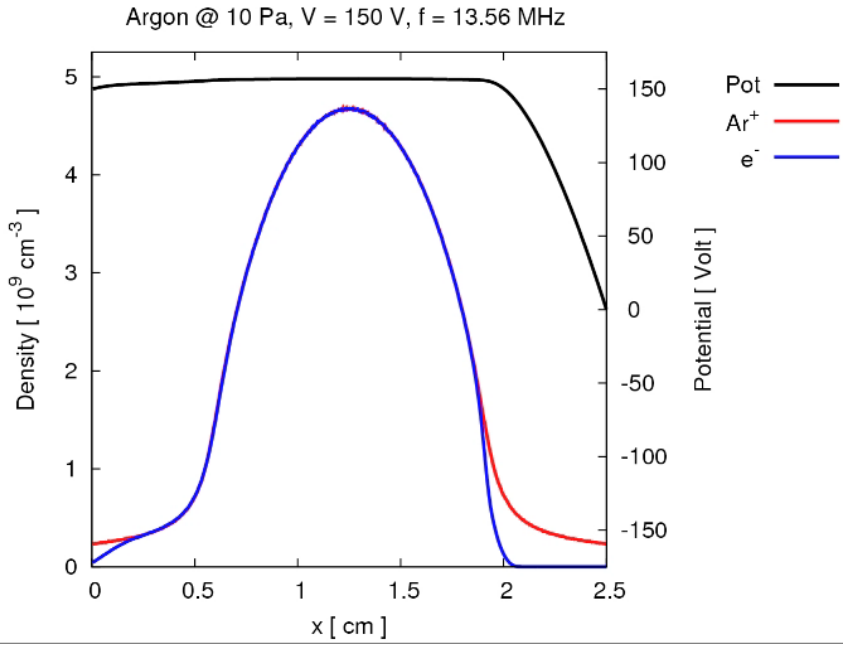
Spatiotemporal changes in electric field

Non-equilibrium transport of particles

Numerical simulation helps to understand the behaviour of particles

Uses kinetic theory for describing particle movement

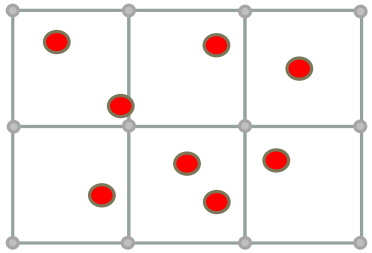
1D and 2D geometries



PLASMA SIMULATION

Approach: Particle-in-Cell (PIC) simulation, no direct particle interaction

- particles interact with the field
- place particle charges to grid
- solve grid for field – Poisson equation
- move particles based on field forces



Particle count: from 100k to 10m particles

Complication: collision!

CPU execution is long: ranging from days to months

- parallel solutions: OpenMP and/or MPI code
- irregular memory accesses make code difficult to parallelise efficiently

Initial goal: at least 10x speedup on GPU

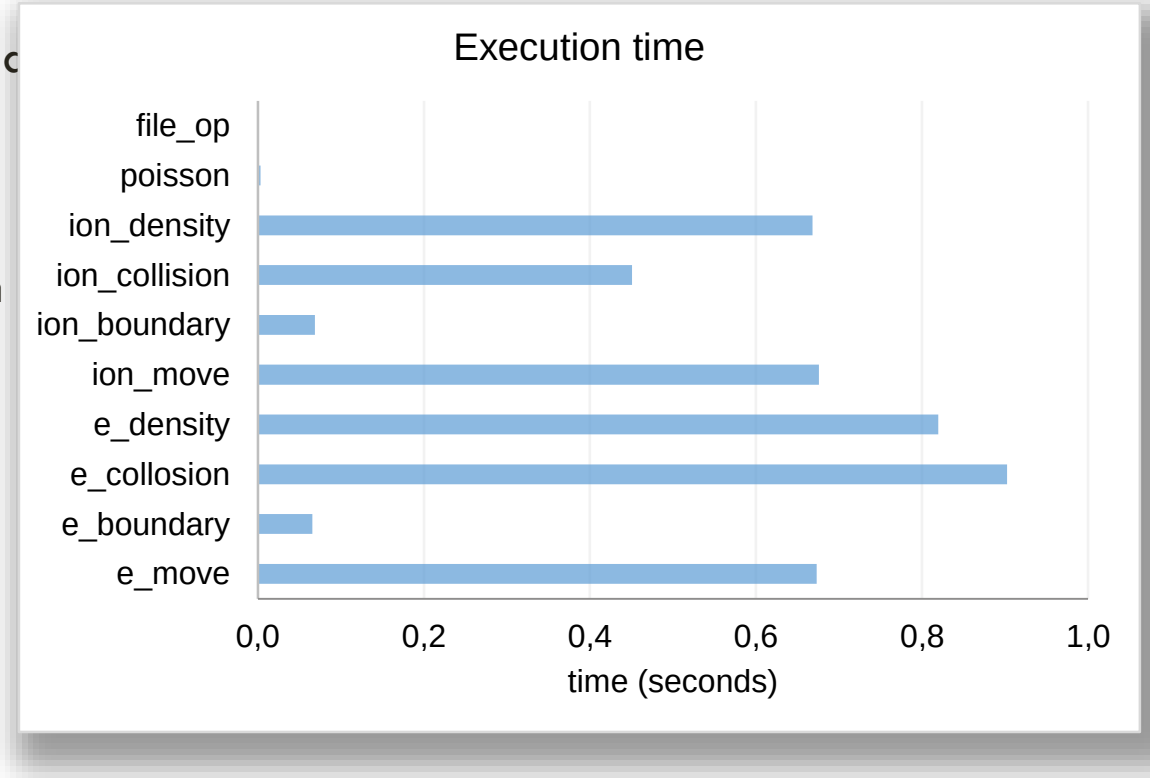
CPU BASELINE IMPLEMENTATION – 1D

Loop for simulation cycles (1000-3000)

Loop for input samples (800)

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

runtime: 1.5 to 14 hours



RECAP FROM PAST GPU DAY TALKS

2017: Zoltan Juhasz, Peter Hartmann and Zoltan Donko, *Highly Parallel GPU-based Particle-in-Cell/MCC Plasma Simulation*

GPU port of a sequential 1D PIC/MCC simulation

- speedup: ~ 15x
- in line with literature

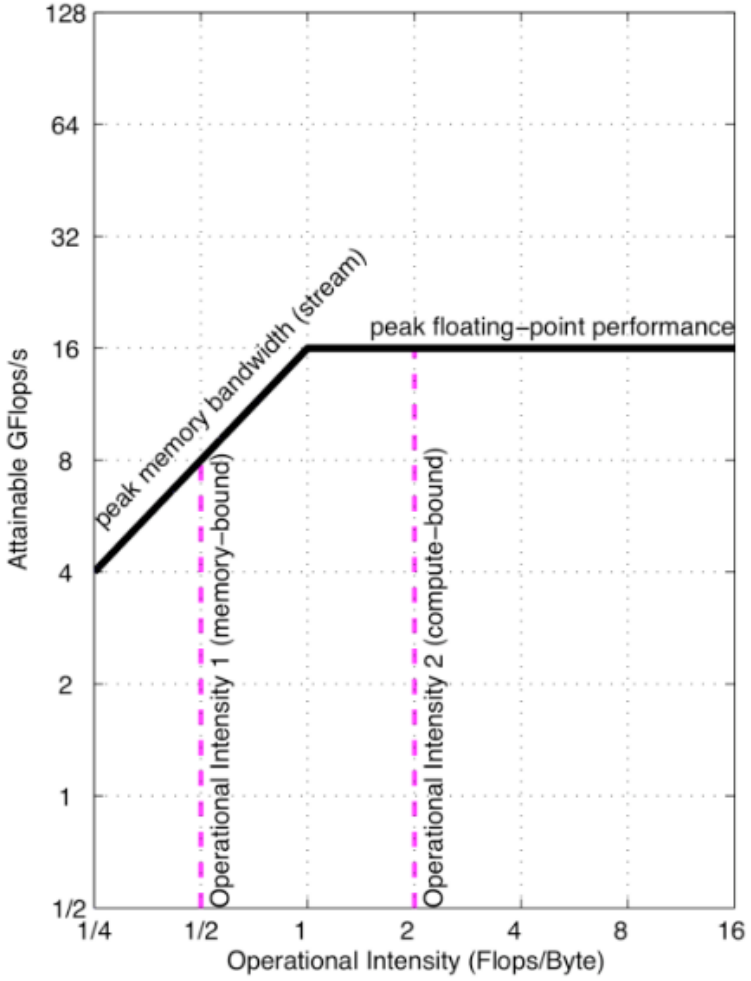
2018: P. Hartmann, Z. Juhász, Z. Donkó, *Accelerated Particle in Cell with Monte Carlo Collisions (PIC/MCC) for gas discharge modeling in realistic geometries*

New 2D PIC/MCC code

- speedup: ~ 90x
- first of its kind

So far so good, but…
Is this speedup good enough?
Can we improve performance further?

THE ROOFLINE PERFORMANCE MODEL



Hardware-algorithm match

The Roofline model:

Attainable Gflop/s =
min(peak floating-point performance,
peak bandwidth x operational intensity)

Peak float perf* $\approx f_{cpu} \times \text{cores} \times 2$

Peak bandwidth* $\approx f_{mem} \times \text{bus\_width} \times 2 / 8$

$$\text{Operational intensity} = \frac{\text{floating point operations}}{\text{bytes transferred}}$$

* or obtained from technical specifications

S. Williams and A. Waterman and D. Patterson, Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM, Vol 52, No 4, pp. 65–76, 2009.

GPU BASELINE IMPLEMENTATION (1D)

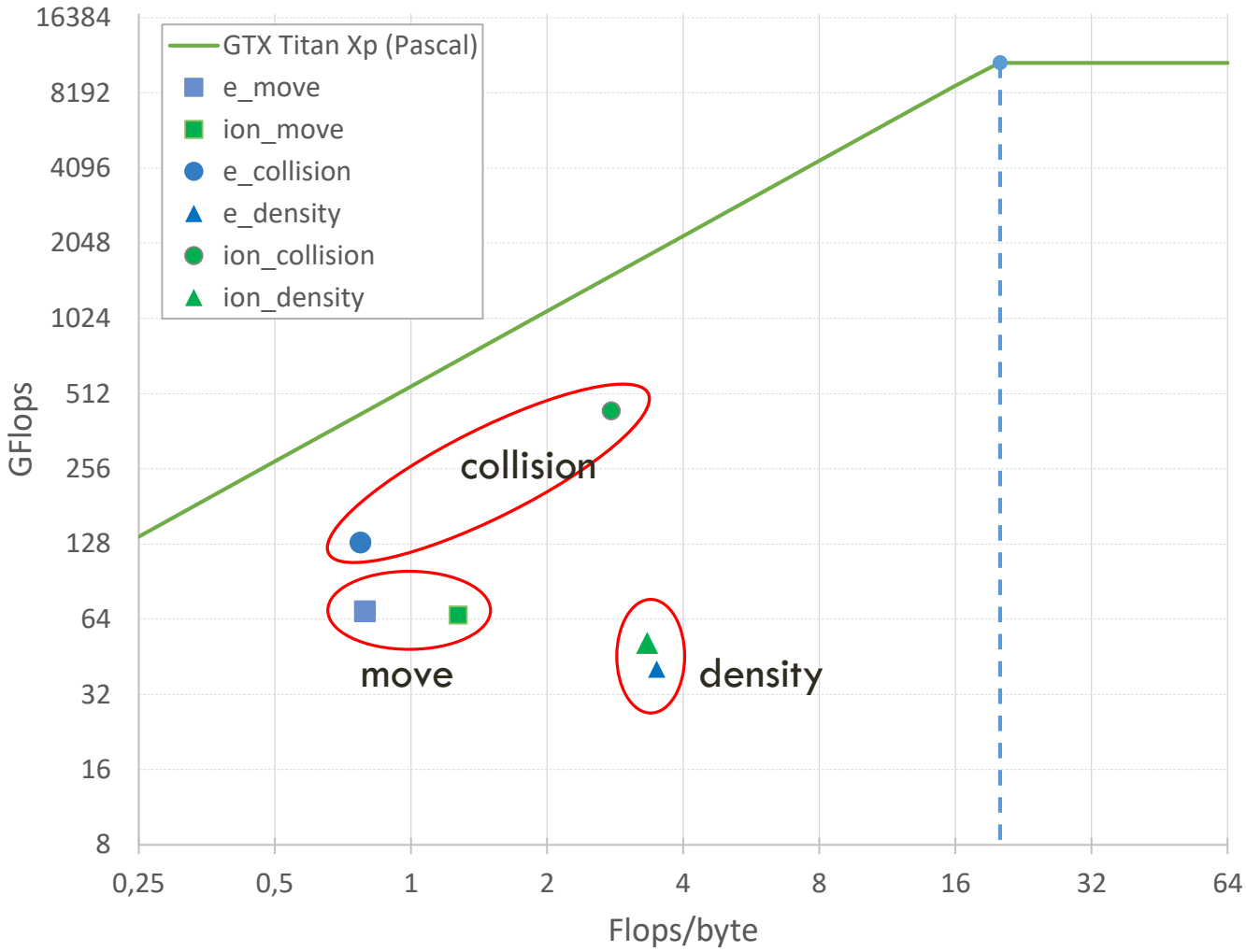
Loop for simulation cycles (1000-3000)

Loop for input samples (800)

1. move electrons -- e_move kernel
2. check boundaries -- e_boundary kernel
3. electron collision -- e_collisions kernel
4. electron density calculation -- e_density kernel
5. move ions -- ion_move kernel
6. check boundaries -- ion_boundary kernel
7. ion collision -- ion_collisions kernel
8. ion density calculation -- ion_density kernel
9. Poisson solver -- CPU seq. solver (Thomas algorithm)

GPU BASELINE ROOFLINE

compute/memory/latency bound code?



GPU BASELINE IMPLEMENTATION (1D)

Loop for simulation cycles (1000-3000)

Loop for input samples (800)

1. move electrons -- e_move kernel
2. check boundaries -- e_boundary kernel
3. electron collision -- e_collisions kernel
4. electron density calculation -- e_density kernel
5. move ions -- ion_move kernel
6. check boundaries -- ion_boundary kernel
7. ion collision -- ion_collisions kernel
8. ion density calculation -- ion_density kernel
9. Poisson solver -- CPU seq. solver (Thomas algorithm)

Problems:

- too many small kernels with low op. intensity,
- memory bound kernels,
- kernel launch overhead,
- CPU Poisson solver, host-device data transfer

TRIVIAL OPTIMISATION RULES

Run everything on the GPU, if possible – avoid host/device data transfer

Use **float** instead of **double** everywhere possible – e.g. 1:32 penalty on Geforce cards

Use Structure of Arrays instead of Array of Structures – coalesced memory address access patterns

Have enough blocks and warps for execution

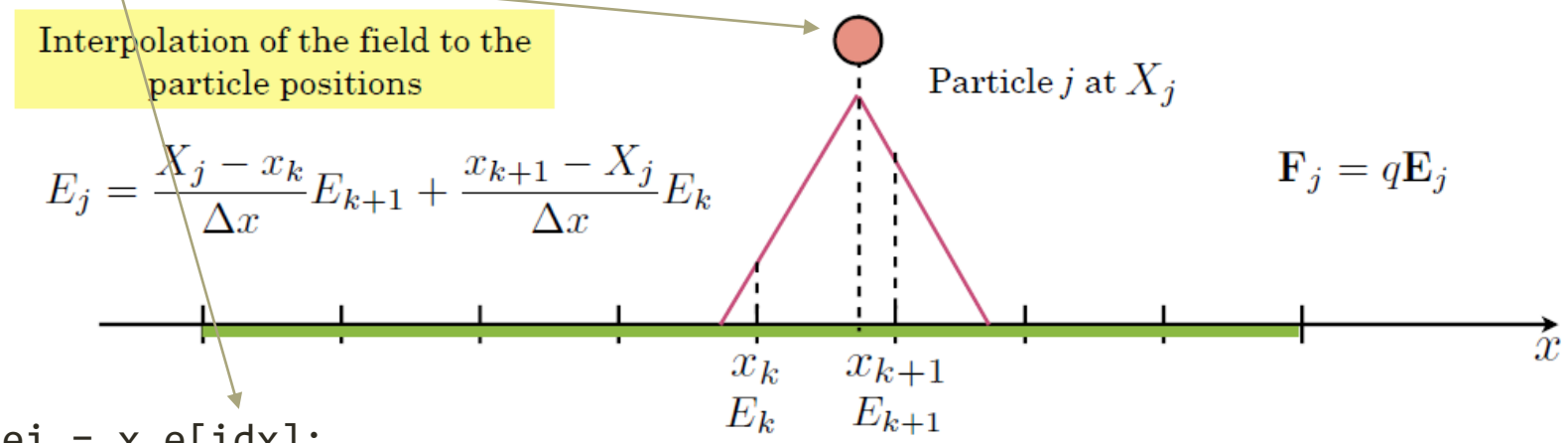
- Titan Xp (3840 cores): 30 SMs x 128 cores
- large blocks: more resources (shared mem, registers), potential scheduling stalls

GPU CARDS USED

card	K4200	GTX980	GTX1080	TitanXP	P100	V100
capability	3.0	5.2	6.1	6.1	6.0	7.0
SMs	7	16	20	30	56	84
cores/SM	192	128	128	128	64	64
total cores	1344	2048	2560	**3840**	3584	5376
memory (GB)	4	4	8	12	16	16
performance (GFlops)	2096.64	4612	8227.8	**10790.4**	9340	14899
max. resident grids per device	16	32	32	32	128	128
max threads per block	1024	1024	1024	1024	1024	1024
max resident blocks/SM	16	32	32	32	32	32
max resident threads/SM	2048	2048	2048	2048	2048	2048
max resident warps/SM	64	64	64	64	64	64
per device						
max resident blocks/device	112	512	640	960	1,792	2,688
max resident threads/device	14,336	32,768	40,960	61,440	114,688	172,032

PARTICLE MOVE – INDIRECT ADDRESSING

1 particle – 1 thread




```
float xei = x_e[idx];
float vxei = vx_e[idx];
float vrei = vr_e[idx];
int k = (int)(xei*c_deltax_inv);
float e_x = ((k+1)*deltax-xei) * c_deltax_inv*efield[k] +
            (xei-k*deltax) * c_deltax_inv * efield[k+1];
vxei -= s1*e_x;
xei  += vxei*dt_e;
```

PARTICLE MOVE: OPTION 1

```
int p = (int)(xei*c_deltax_inv);

extern __shared__ float s_efield[];
int k = threadIdx.x;
while (k<n){
   s_efield[k] = efield[k];
   k += blockDim.x;
}
__syncthreads();

float e_x = ((p+1)*deltax-xei) * c_deltax_inv*s_efield[p] +
            (xei-p*deltax) * c_deltax_inv * s_efield[p+1];
```

PARTICLE MOVE: OPTION 2

```
__global__ void electrons_move_kernel(float* __restrict__ x_e,
      float* __restrict__ vx_e, float* __restrict__ vr_e,
      float* efield, float deltax, ...)
```

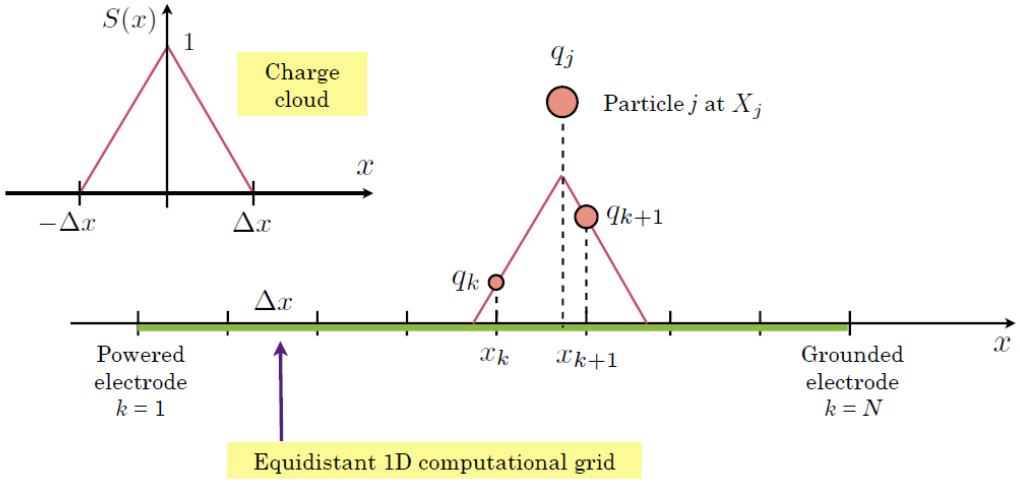
```
__global__ void electrons_move_kernel(float* __restrict__ x_e,
      float* __restrict__ vx_e, float* __restrict__ vr_e,
      const float* __restrict__ efield, float deltax, ...)
```

fixes pointer aliasing problems and caches read-only data

Shared memory or const __restrict__ achieves similar results:

29.4 μs to 6.4 μs, 68.9 to 325 Gflops!

DENSITY CALCULATION



Several options:

1. global memory version

   add charge in global memo

   T atomicAdd(T* address, T val);

2. shared memory version

   calculate local densities in shared memory

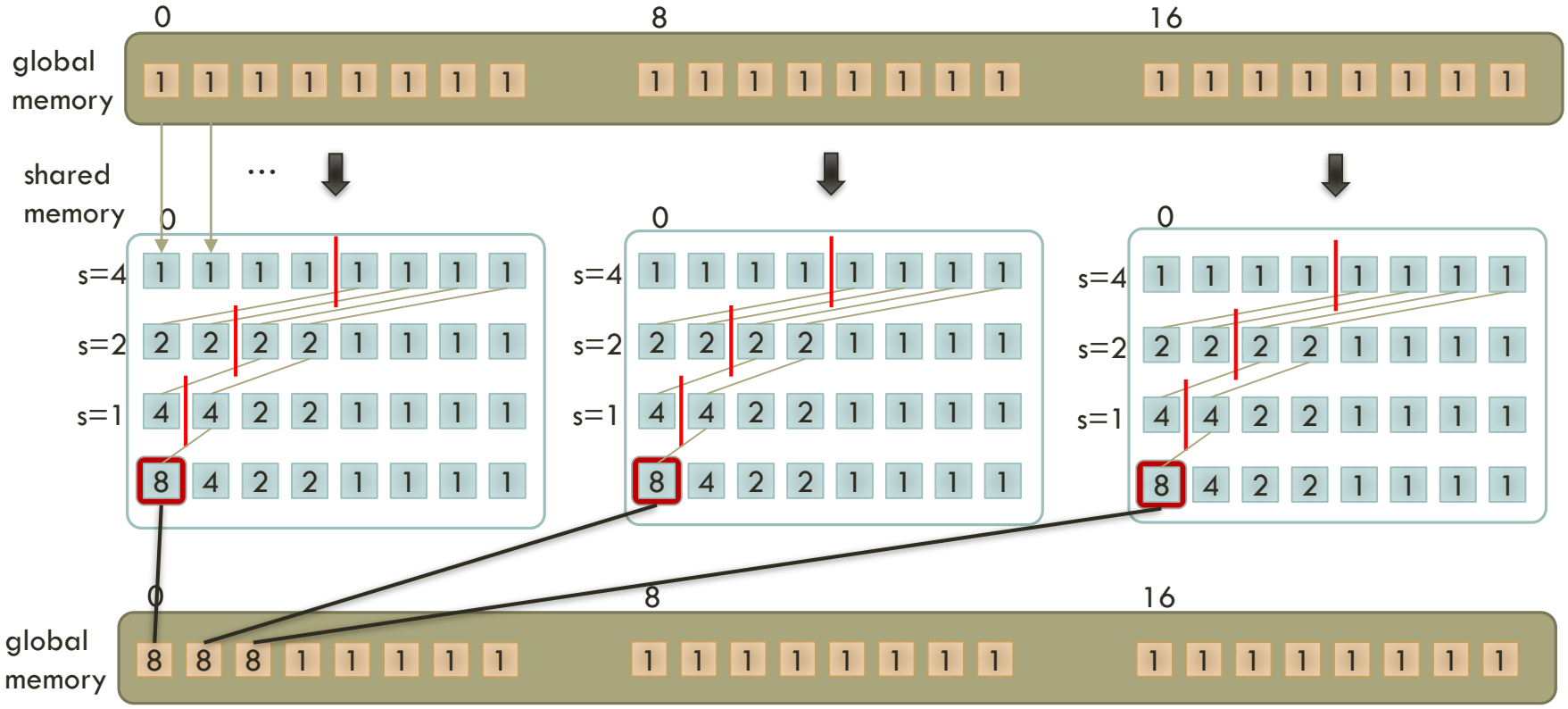
   merge local results into global memory

   a) merge using extra kernel to add local densities (reduction)

   b) merge in histogram kernel using global atomicAdd

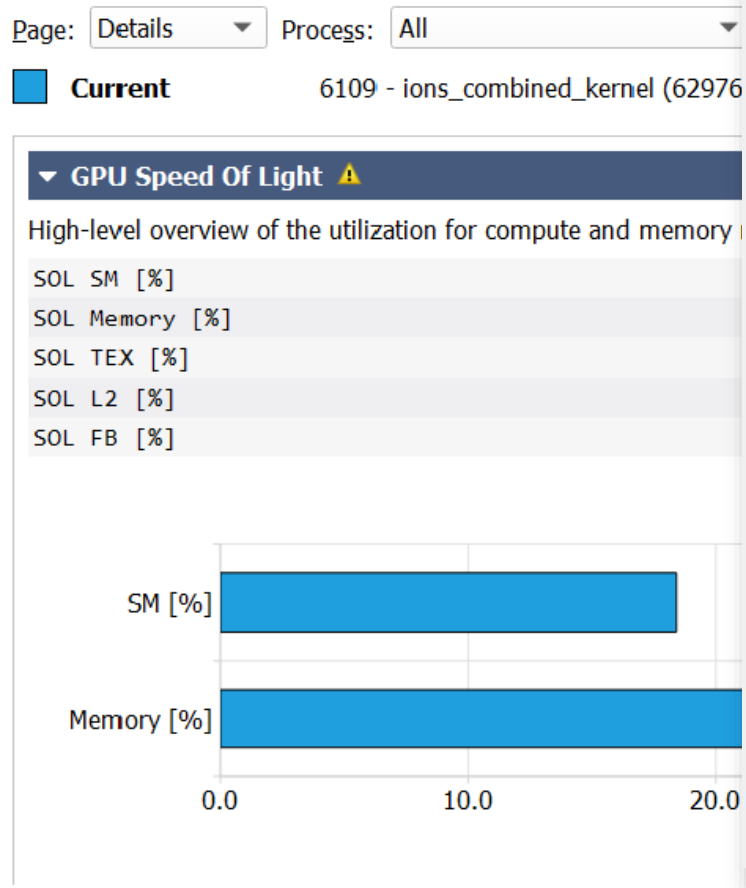
Performance may vary depending on hardware architecture (atomic support)

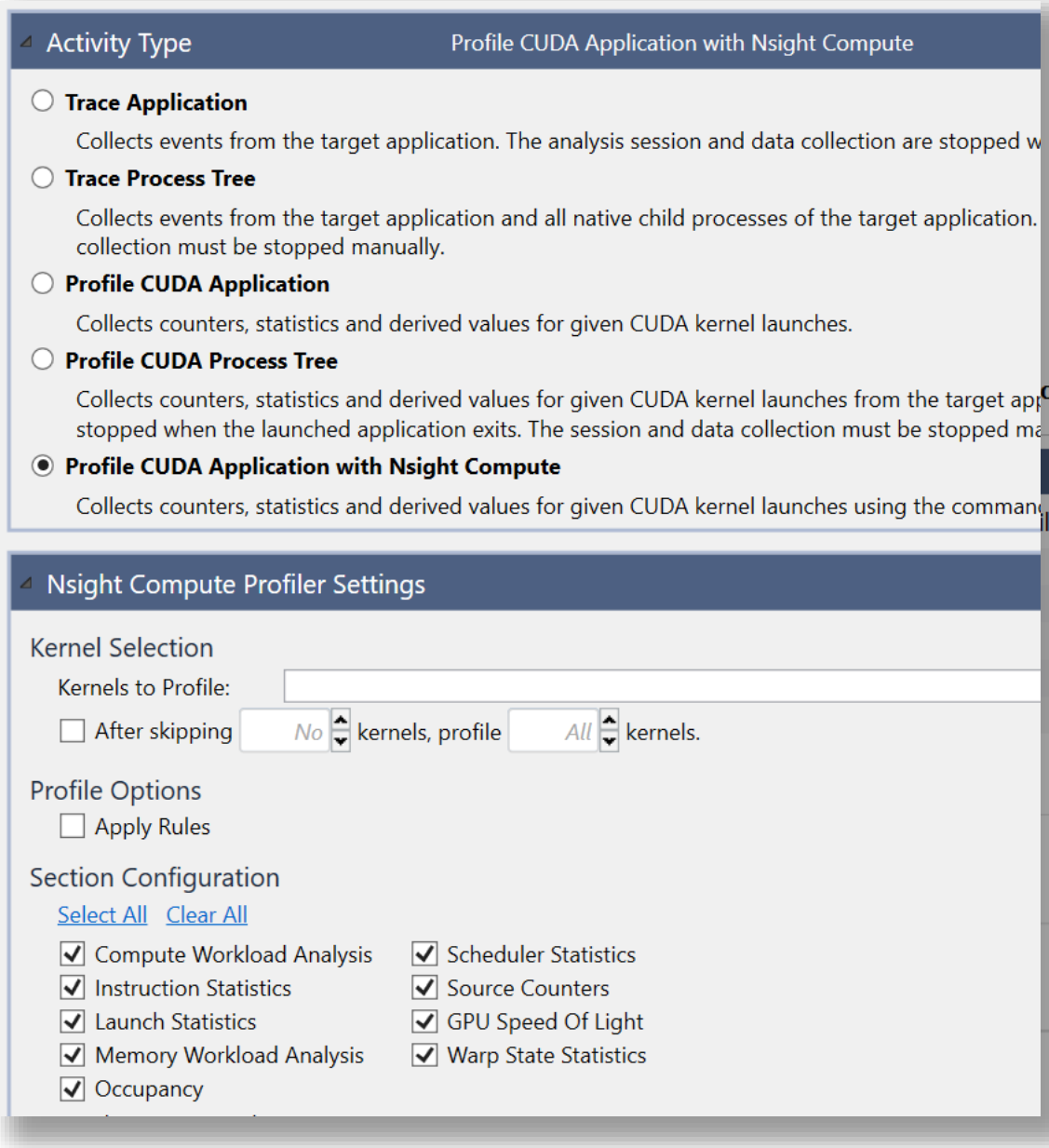
REDUCTION USING SHARED MEMORY



KERNEL FUSION

move + collision + density co





POISSON SOLVER

1D: CPU: direct solver, Thomas algorithm

- execution time: 2 μs
- exec. time with data transfer: 220 μs

GPU implementation:

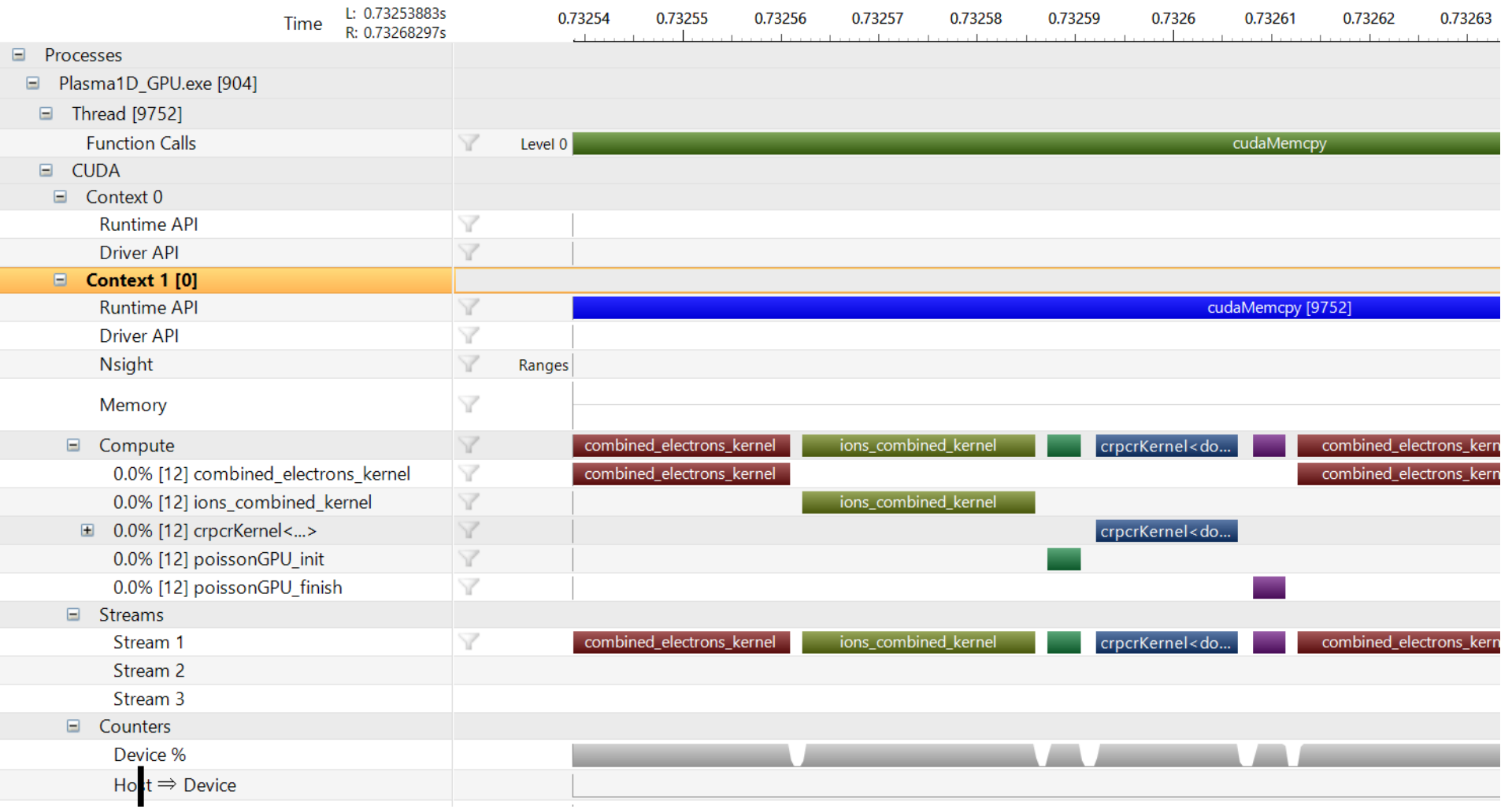
- CR-PCR (Cyclic reduction – Parallel Cyclic Reduction)
- kernel implementation from CUDA Data Parallel Primitives Library (https://github.com/cudpp/cudpp)

Yao Zhang, Jonathan Cohen, and John D. Owens. Fast Tridiagonal Solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 127–136, January 2010

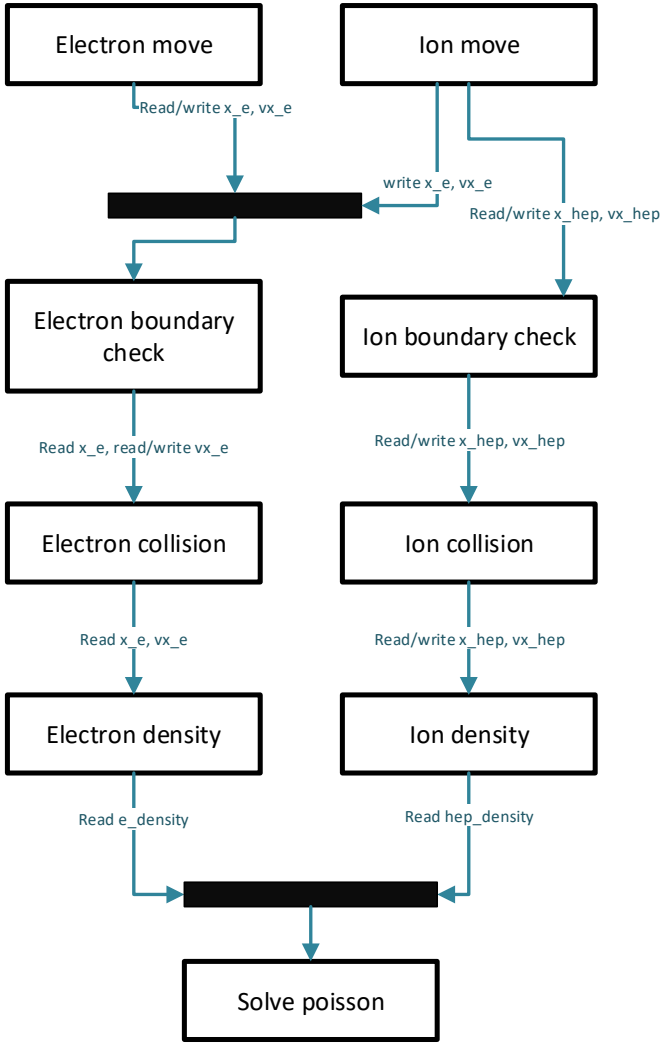
Execution time: 14.3-16 μs

2D: Successive Overrelaxation

EXECUTION TIMELINE



USING STREAMS FOR INTERNAL PARALLELISM

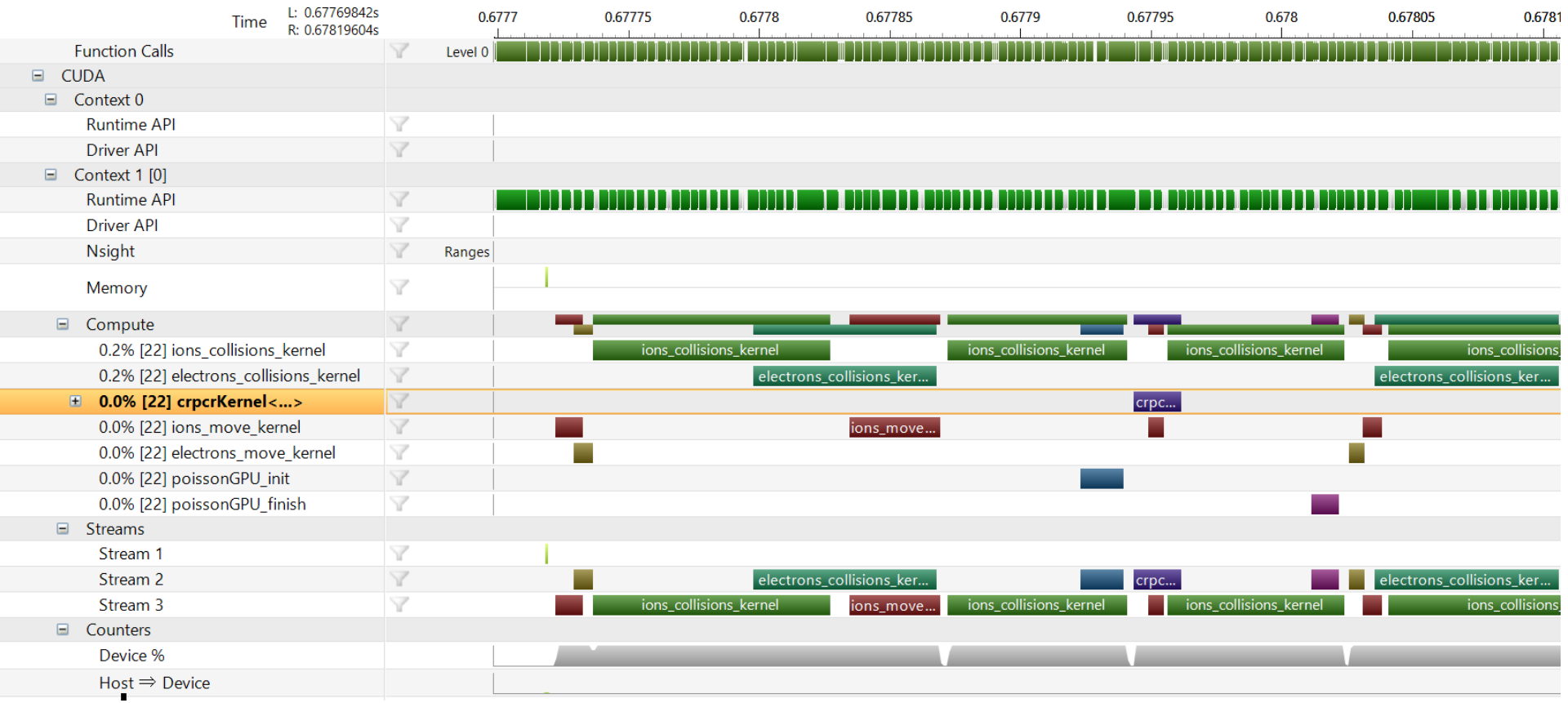



```
cudaStreamWaitEvent(stream[1], solver_event, 0);
ions_move_kernel<<<igrid, iblock,
                   n*sizeof(float), stream[1]>>> (...);
cudaEventRecord(ion_move_event, stream[1]);
electrons_move_kernel_unrolled<<<egrid, eblock,
                   n*sizeof(float), stream[0]>>>(...);

cudaStreamWaitEvent(stream[0], ion_move_event, 0);
electrons_collisions_kernel<<<egrid, eblock,
                   0, stream[0] >>>(...);
cudaEventRecord(electron_density_event, stream[0]);
ions_collisions_kernel<<<igrid, iblock,
                   0, stream[1]>>>(...);
cudaEventRecord(ion_density_event, stream[1]);

cudaStreamWaitEvent(stream[0], electron_density_event, 0);
cudaStreamWaitEvent(stream[0], ion_density_event, 0);

poissonGPU_init <<<1, n, 0, stream[0]>>>(...);
crpcrKernel<<<pgrid, pthreads,
                   p_smemSize, stream[0]>>>(...);
poissonGPU_finish<<<1, n, 0, stream[0]>>>(...);
cudaEventRecord(solver_event, stream[0]);
```

STREAMED SIMULATION TIMELINE

Concurrent kernel support

Might better utilise GPU, if kernel occupancy is below 60%

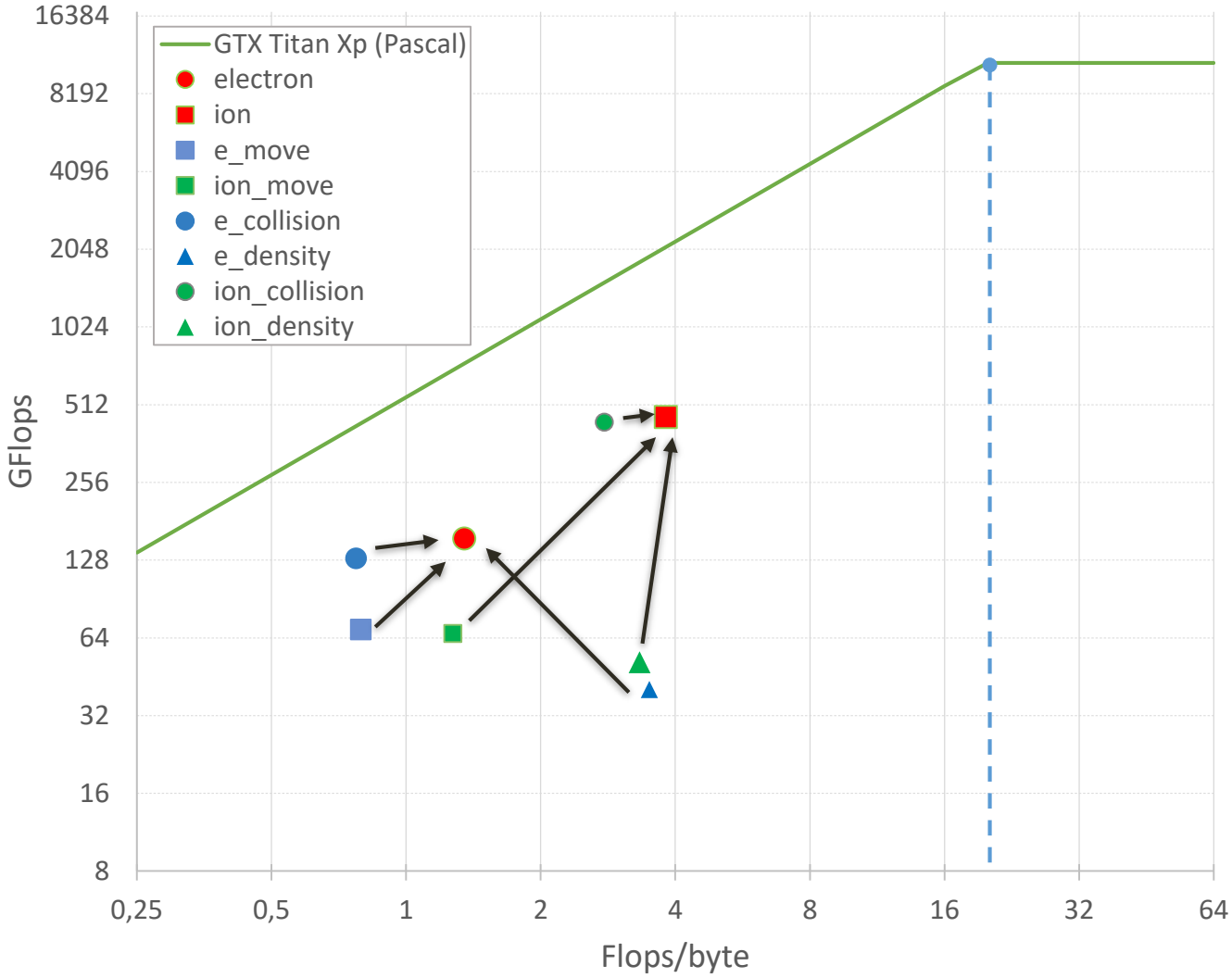


OPTIMISATION SUMMARY

Version	Notes	Time (seconds)
v0	baseline – separate kernels, CPU Poisson solver, extra host-device copies	46.879
v1	fused kernels, CPU Poisson	19.828
v2	fused kernels, GPU Poisson prototype	8.411
v3	Shared-mem combined kernels, CR-PCR Poisson kernel, block size=256	6.409
v4	further float fixes (sqrtf, expf), block size=512	5.113

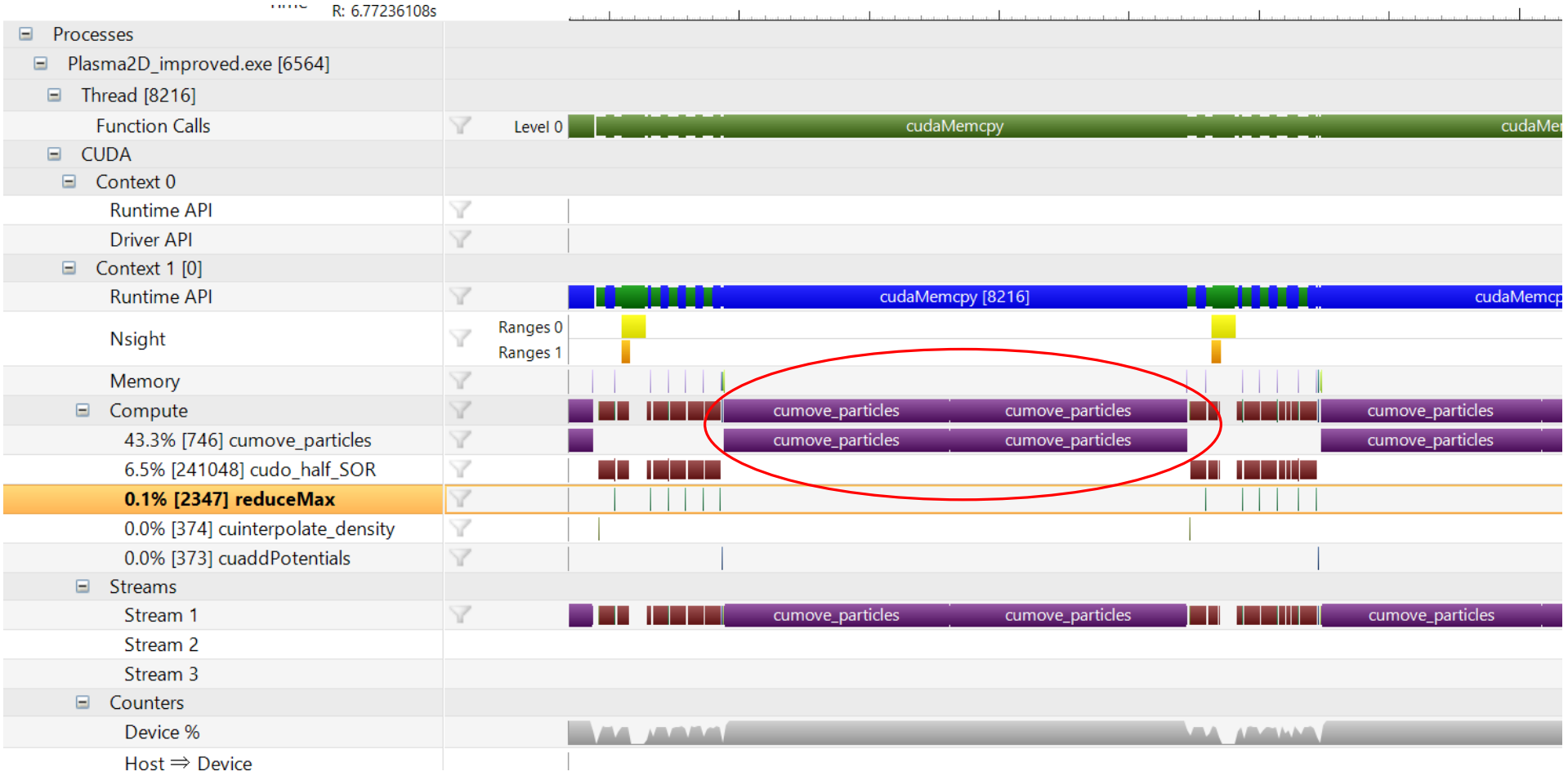
speedup: 80-100x, depending on CPU

ROOFLINE MODEL: FINAL VERSION



TIMELINE OF 2D VERSION

performance limiter: particle move



CONCLUSIONS

1D version

- 10x performance increase to GPU baseline
- ~ 100x speedup to original CPU code

2D version

- 2x performance increase to GPU baseline
- ~ 200x speedup to original CPU code

New results can be obtained within minutes or hours

Key optimisation steps

- using floats instead of doubles
- minimise memory reads, store values in registers and reuse, recalculation may be cheaper
- extensive use of shared memory
- fusing kernels to reduce kernel launch and data fetch overheads
- no CPU code, no host-device data transfer

Future work

- cooperative groups/threads – fine grain execution control
- cuda graphs – more efficient task graph launch within loops