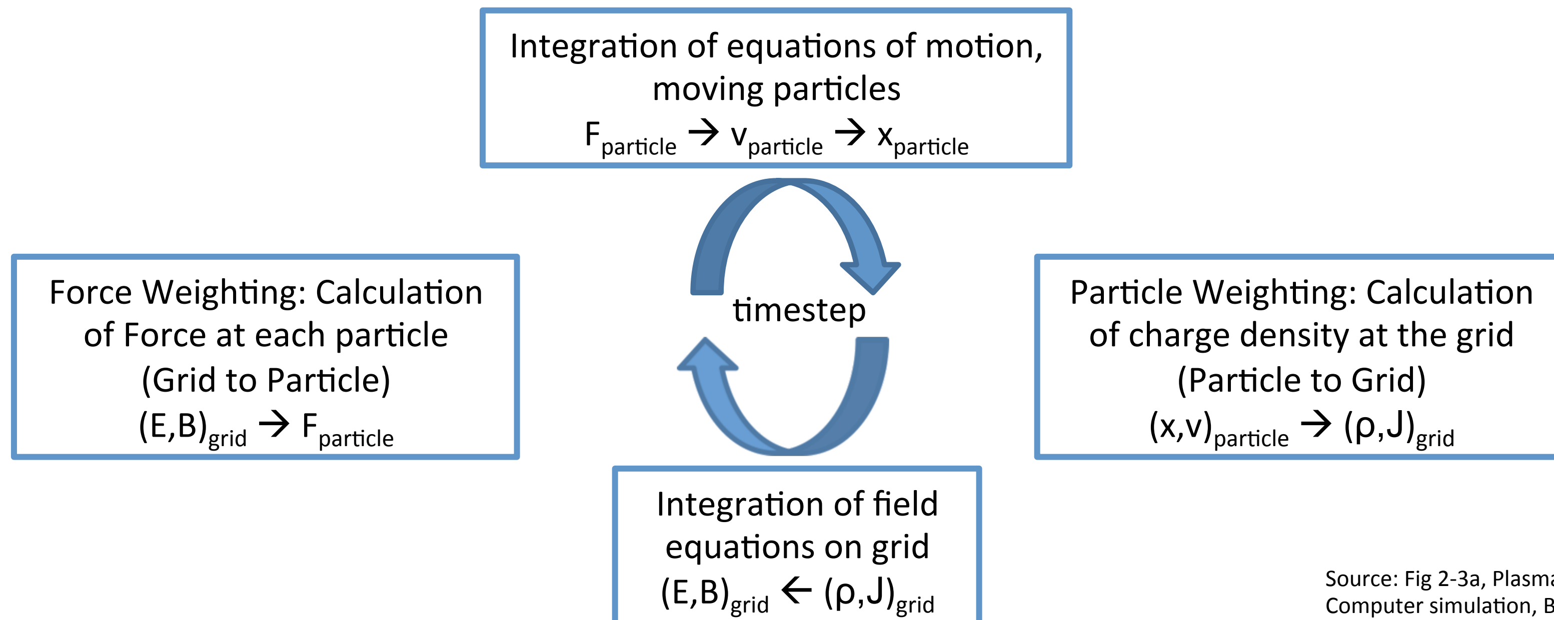
Beam Dynamics Simulations Using GPUs

Jutta Fitzek, GSI

Space Charge Workshop 2013

Simulation

- Longitudinal simulation (LOBO)



Source: Fig 2-3a, Plasma Physics via Computer simulation, Birdsall, Langdon

Space charge solver:

$$I_n(t^*) = FFT[I(Z_i, t^*)]$$

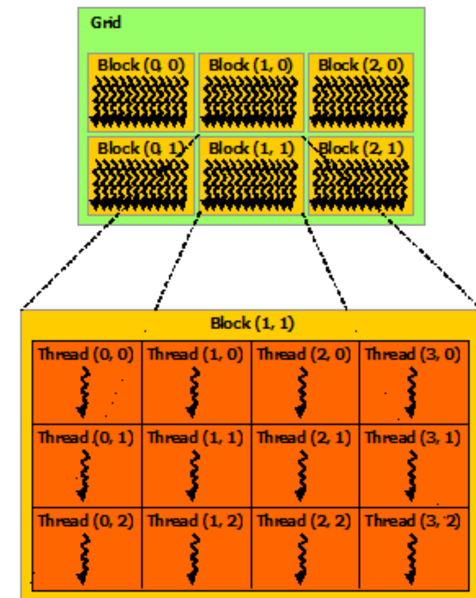
$$E(X_i, t^*) = FFT^{-1}[-Z_n I_n(t^*)]$$

Simulation parameters

- Grid: 128 – 512 points, equidistant
- 500000 macro particles
- elliptic or gaussian distribution
- RF not switched on
- Cut-off frequency as input parameter

GPU computing – basics

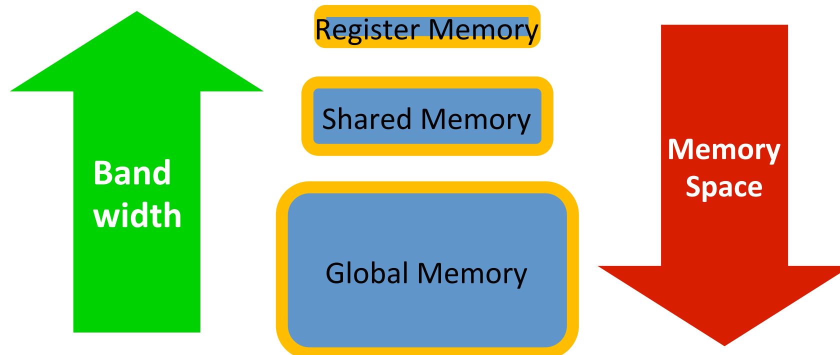
- since the end of the 1990's, GPUs are explored as basis to solve compute intensive problems
- massive parallel execution of the same code with different data (data parallelism)
- structuring achieved by mapping the problem onto threads, blocks and grids
- drawback: bottleneck is the time-consuming copying of data



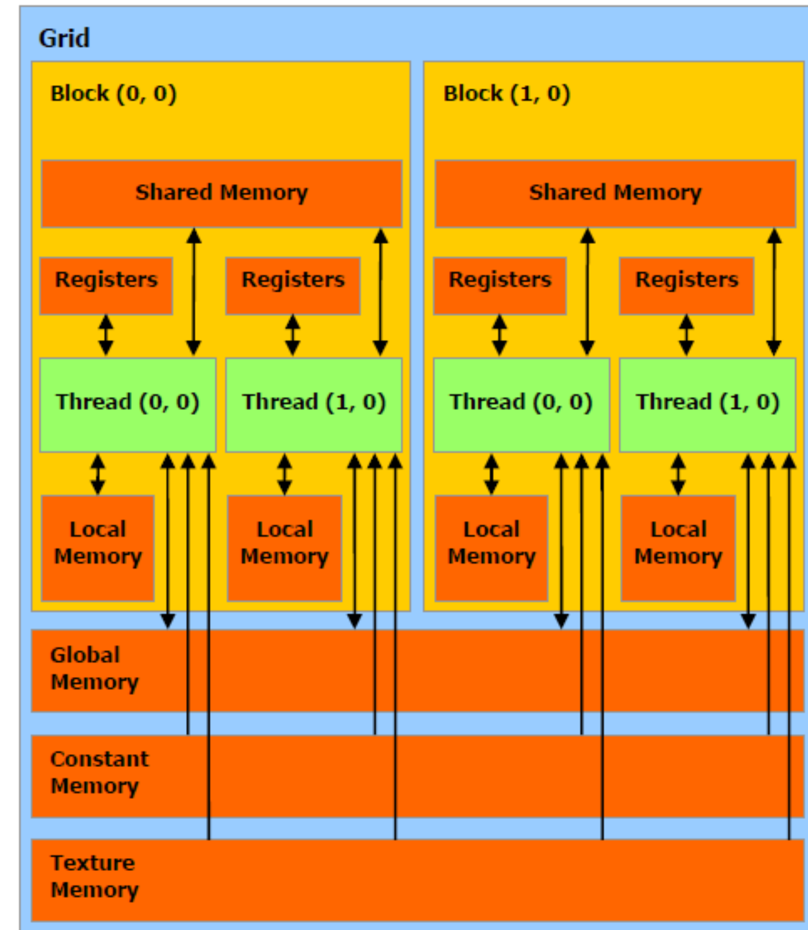
©CUDA Programming Guide, NVIDIA

GPU Computing – Memory Model

- Read-write per-thread *registers*
- Read-write per-thread *local memory*
- Read-write per-block *shared memory*
- Read-write per-grid *global memory*
- Read-only per-grid *constant memory*
- Read-only per-grid *texture memory*



©Michael Bussmann, HZDR, 2007



©CUDA programming guide, NVIDIA, 2007

GPU Computing – applied

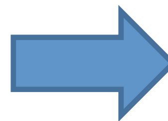
- Statements:
  - The number of Grid points is smaller compared to the number of particles, e.g. 256x256 vs. 500000+ => exchange/share the Grid
- Setup using the GPU:
  - Every thread treats 1 particle
  - Grid resides in global memory and is updated
  - Static data (if any) is kept in constant memory

GPU computing – code snippet

- Fits well for data parallelism
- Single step, calculations without FFT (cufft): 0.124 sec (serial) down to 0.00029 sec (GPU) version (GPU time 0.298016 sec)

```
67 //  calculated line current density to 1D grid
68 void Beam::line_current_density(SynParticle& sp, Grid1D& ldy) {
69     long j;
70     double pic_quantity = Q * sp.beta0 * clight / ldy.get_dz();
71     for (j = 0; j < ldy.get_size(); j++)
72         ldy[j] = 0.0;
73     for (j = 0; j < pics.size(); j++)
74         ldy.Pic2Field(pic_quantity, pics[j].z);
75 }
```

```
57 // ! Interpolates PIC particle to grid:
58 void Grid1D::Pic2Field(double pic_quantity, double pic_pos) {
59     int j1, j2, n = grid.size();
60     double f1, f2;
61     double dist0 = pic_pos - begin;
62     j1 = (int) floor(dist0 / dz - 0.5);
63     j2 = j1 + 1;
64     f1 = ((j2 + 0.5) * dz - dist0) / dz;
65     f2 = (dist0 - (j1 + 0.5) * dz) / dz;
66
67     this->operator[](j1) += pic_quantity * f1;
68     this->operator[](j2) += pic_quantity * f2;
69 }
```



```
74 __global__ void kernel_line_current_density( StructGrid1D ldy, StructBeam beam, SynParticle sp ) {
75
76     int threadNumber = blockIdx.x * blockDim.x + threadIdx.x;
77
78     // pic to field
79     double pic_quantity;
80     if (threadNumber < beam.numberOfParticles) {
81         pic_quantity = beam.charge * sp.beta0 * clight / ldy.dz;
82
83
84         int j1, j2;
85         double f1, f2;
86         double dist0 = beam.particles_z[threadNumber] - ldy.begin;
87         j1 = (int) floor(dist0 / ldy.dz - 0.5);
88         j2 = j1 + 1;
89         f1 = ((j2 + 0.5) * ldy.dz - dist0) / ldy.dz;
90         f2 = (dist0 - (j1 + 0.5) * ldy.dz) / ldy.dz;
91
92         ldy.gridValues[j1] += pic_quantity * f1;
93         ldy.gridValues[j2] += pic_quantity * f2;
94     }
95
96 }
```

GPU computing - analysis

- How it is done
  - Theoretical analysis on each single kernel, treated as if the GPU really was SIMD / PRAM
  - e.g. updating particle positions in O(1) instead of O(n)
  - Data transfer host <-> device can be calculated by data amount
- However..
  - Blocks are executed in an undefined order when the GPU has time for it!
  - Overhead of the operating system when issuing GPU calls?
  - Different GPU Cards behave differently? How can we derive general predicates?

First findings

- Tips for GPUs:
  - Keep particles on the GPU, use the GPU only for long-term calculations with little/seldom output
  - Different (old?) programming paradigm, try to structure the code well
  - 10^6 particles is ok, but.. (GPU should be kept busy)
- General tips for parallelization:

  (1) localize the calculation, avoid data moving

  (2) parallelize the most time-consuming routines (!?)

  (3) for GPUs: use faster memory where possible

Outlook

- Next steps: integrate MPI again
  - Longitudinal (1D): particles can be arbitrarily distributed, but FFT needs to be done over all particles
  - Transversal (2D/3D, PATRIC): additional complexity: distribute "slices" of the beam to have nearby particles reside on the same node, particles move between the nodes
    → look at integrating an MPI framework that allows data to be transferred between GPUs directly: "CUDA-aware MPI frameworks"