Numerical Simulation on GPUs

Rüdiger Westermann
Lehrstuhl für Computer Graphik und Visualisierung

Overview

- Numerical simulation techniques
  - Grid based numerical simulation techniques
    - From PDEs to difference equations
    - Discretization
    - Solution methods
  - Particle based numerical simulation techniques
    - SPH (Smoothed Particle Hydrodynamics)
    - The discrete kernel
    - Operations and data structures

Grid based simulation on GPUs

- Remember:
  Numerical solution methods of partial differential equations
  - Based on a discretization of the domain
  - Replace PDEs and closed form expression by approximate algebraic expressions
    - Partial derivatives become difference quotients
    - Involves only values at a discrete set of computational structures in the domain, at which the solution is determined

Grid based numerical simulation

- From PDEs to difference equations
  - Replace partial derivatives by finite differences on a grid
  - Example: The 2D wave equation

$$\frac{\partial^2 u}{\partial t^2} - c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

Partial Differential Equation

$$\frac{u_{ij}^{t+1} - 2u_{ij}^{t} + u_{ij}^{t-1}}{\Delta t^2} - c^2\left(\frac{u_{i+1\,j}^{t} + u_{i-1\,j}^{t} + u_{ij+1}^{t} + u_{ij-1}^{t} - 4u_{ij}^{t}}{(\Delta h)^2}\right) = 0$$

Difference Equation ($\Delta x = \Delta y = \Delta h$)

Grid based simulation on GPUs

$$\frac{\partial^2 u}{\partial t^2} - c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

- Remember numerical solution of the 2D wave equation
  - Can be solved explicitely for the unknown displacements u_{ij}

$$u_{ij}^{t+1} = C_1 \cdot \left(u_{i+1\,j}^{t} + u_{i-1\,j}^{t} + u_{ij+1}^{t} + u_{ij-1}^{t}\right) + C_2 \cdot u_{ij}^{t} - u_{ij}^{t-1}$$

  - Stepping through all interior points of the domain and updating u^{t+1} can be performed in parallel using a CUDA kernel
    - Needs special treatment of boundary (not in the code on next page)

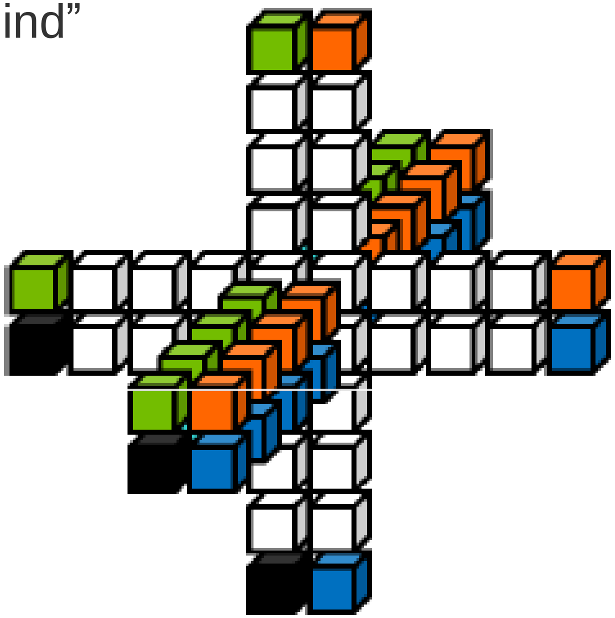
Grid based simulation on GPUs

- The CUDA kernel to numerically solve the 2D wave equation

```
   ...
   // allocate fast shared memory to cache the working set
   __shared__ float u_sh_last[BLOCKSIZEX][BLOCKSIZEY]; // need last time step, too
   __shared__ float u_sh[BLOCKSIZEX][BLOCKSIZEY];

   // fill the shared memory with the working set from the global device array
   u_sh[tX][tY] = u_sh_last[tX][tY] = u_last[i + j * Nx]; // need last time step, too

   __syncthreads(); // all threads wait at this barrier for all others

   if(tX > 0 && tX < BLOCKSIZEX-1 && tY > 0 && tY < BLOCKSIZEY-1) {

         // compute the stencil on the data in shared memory and write to out array
         out[i + j * Nx] = C1 * (u_sh[tX+1][tY] + u_sh[tX-1][tY] +
                                 u_sh[tX][tY+1] + u_sh[tX][tY-1]) +
                           C2 *  u_sh[tX][tY] - u_sh_last[tx][ty];
   }
} // end of computeStencilOnDevice
```

Grid based simulation on GPUs

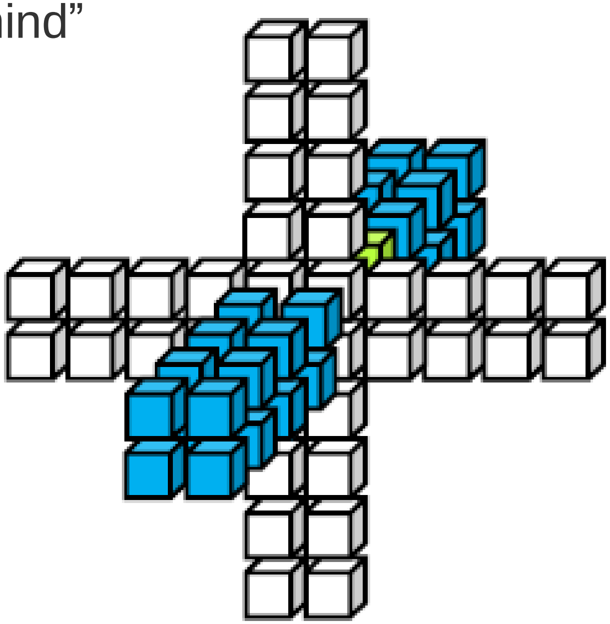
- Realizing a 3D FD kernel using CUDA
  - As in 2D, the *xy*-slices per thread block are stored in shared memory
  - Each thread keeps the required z-elements in registers – *n* "infront", *n* "behind"
  - The simulation moves slice by slice through the grid



http://developer.download.nvidia.com/CUDA/CUDA_Zone/papers/gpu_3dfd_rev.pdf

Grid based simulation on GPUs

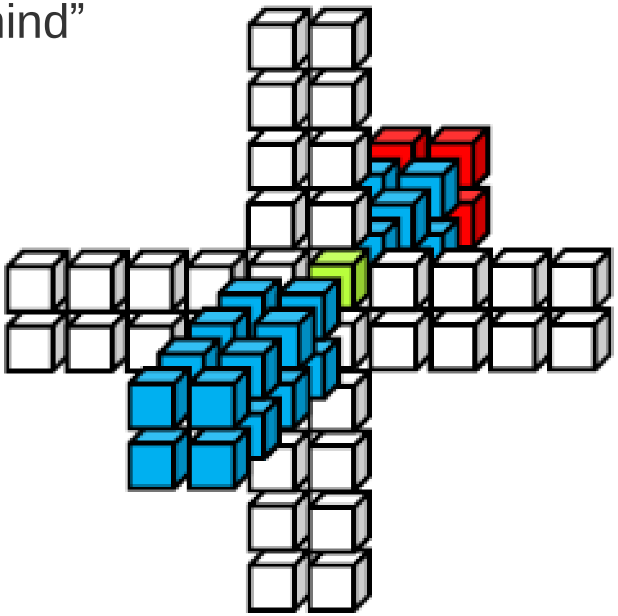
- Realizing a 3D FD kernel using CUDA
  - As in 2D, the *xy*-slices per thread block are stored in shared memory
  - Each thread keeps the required z-elements in registers – *n* “infront”, *n* “behind”
  - The simulation moves slice by slice through the grid



http://developer.download.nvidia.com/CUDA/CUDA_Zone/papers/gpu_3dfd_rev.pdf

Grid based simulation on GPUs

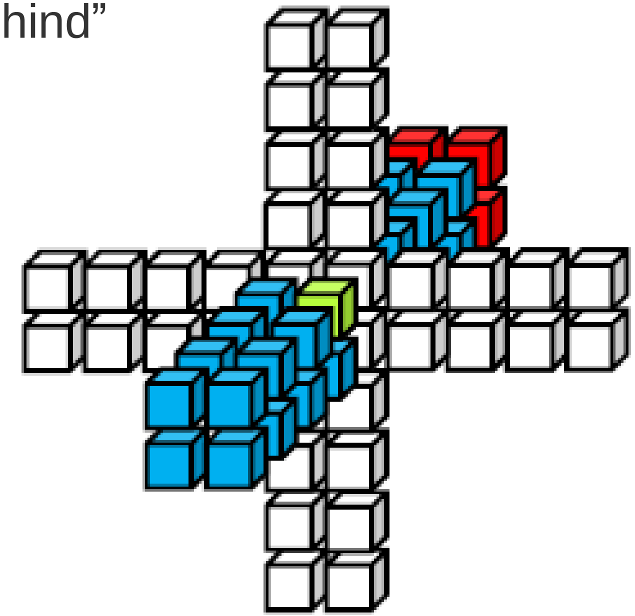
- Realizing a 3D FD kernel using CUDA
  - As in 2D, the *xy*-slices per thread block are stored in shared memory
  - Each thread keeps the required z-elements in registers – *n* “infront”, *n* “behind”
  - The simulation moves slice by slice through the grid



http://developer.download.nvidia.com/CUDA/CUDA_Zone/papers/gpu_3dfd_rev.pdf

Grid based simulation on GPUs

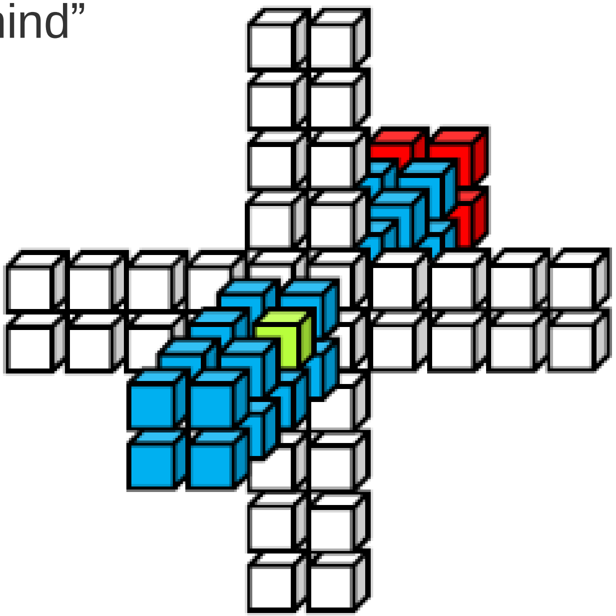
- Realizing a 3D FD kernel using CUDA
  - As in 2D, the *xy*-slices per thread block are stored in shared memory
  - Each thread keeps the required z-elements in registers – *n* "infront", *n* "behind"
  - The simulation moves slice by slice through the grid



http://developer.download.nvidia.com/CUDA/CUDA_Zone/papers/gpu_3dfd_rev.pdf

Grid based simulation on GPUs

- Realizing a 3D FD kernel using CUDA
  - As in 2D, the *xy*-slices per thread block are stored in shared memory
  - Each thread keeps the required z-elements in registers – *n* “infront”, *n* “behind”
  - The simulation moves slice by slice through the grid



http://developer.download.nvidia.com/CUDA/CUDA_Zone/papers/gpu_3dfd_rev.pdf

Grid based simulation on GPUs

- What if the discretization leads to an implicit solution approach
  - System of algebraic equations to be solved

$4\alpha+1$	$-\alpha$		$-\alpha$							x_1		b_1
$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$						x_2		b_2
	$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$					x_3		b_3
$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$				x_4		b_4
	$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$		•	x_5	=	b_5
		$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$		x_6		b_6
			$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$			x_7		b_7
				$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$		x_8		b_8
					$-\alpha$		$-\alpha$	$4\alpha+1$		x_{10}		b_{10}

$$\alpha = \frac{\Delta t^2 \cdot c^2}{2 \cdot \Delta h^2}$$

$$b_{i+j\cdot k} = \alpha \cdot \left(u^t_{i+1,j} + u^t_{i-1,j} + u^t_{i,j+1} + u^t_{i,j-1} - 4 \cdot u^t_{i,j}\right) + 2 \cdot u^t_{i,j} - u^{t-1}_{i,j}$$

$$x_{i+j\cdot Nx} = u_{i,j}$$

N_x, N_y : size of the simulation domain

Grid based simulation on GPUs

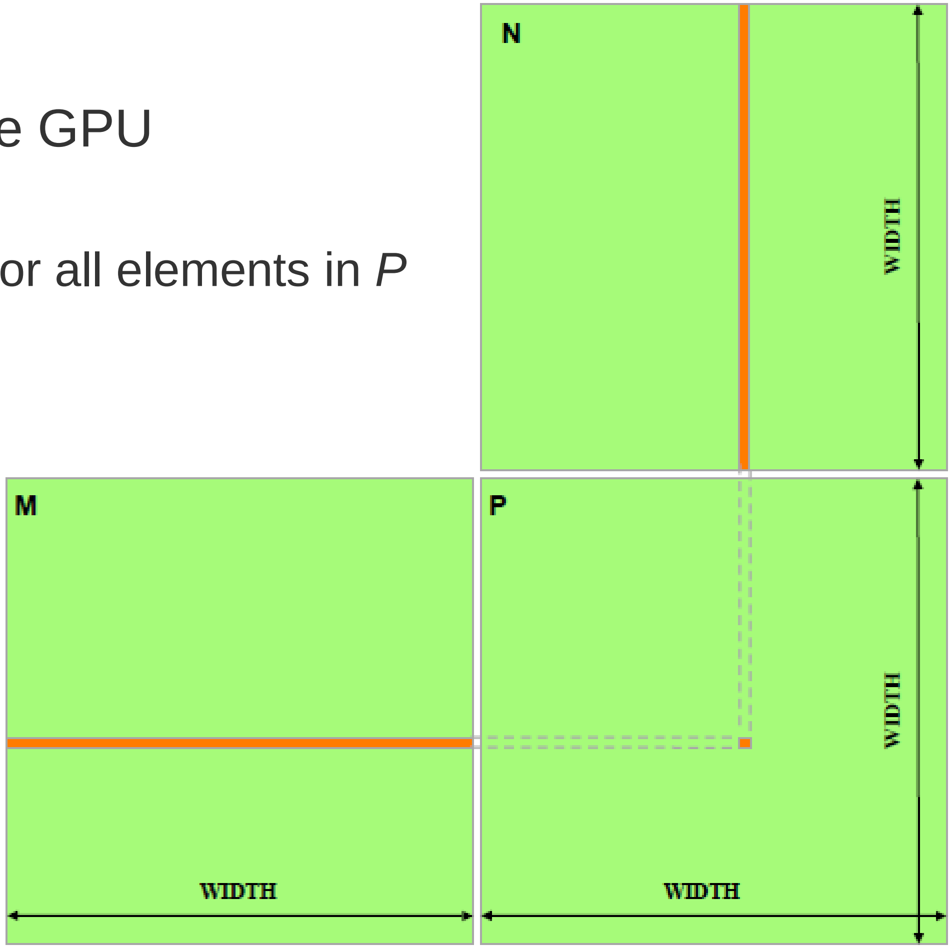
- The implicit solution method requires a numerical solver
  - For instance, a Conjugate Gradient solver would work
  - Requires linear algebra building blocks for matrix and vector operations

```
void clCGSolver::solveInit() {
	Matrix->matrixVectorOp(CL_SUB,X,B,R);     // R = A*x-b
	R->multiply(-1);                          // R = -R
	R->clone(P);                              // P =  R
	R->reduceAdd(R, Rho);                     // rho = sum(R*R);
}
void clCGSolver::solveIteration() {
	Matrix->matrixVectorOp(CL_NULL,P,NULL,Q); // Q = Ap;
	P->reduceAdd(Q,Temp);                     // temp  = sum(P*Q);
	Rho->div(Temp,Alpha);                     // alpha = rho/temp;
	X->addVector(P,X,1,Alpha);                // X = X + alpha*P
	R->subtractVector(Q,R,1,Alpha);           // R = R - alpha*Q
	R->reduceAdd(R,NewRho);                   // newrho = sum(R*R);
	NewRho->divZ(Rho,Beta);                   // beta = newrho/rho
	R->addVector(P,P,1,Beta);                 // P = R+beta*P;
	clFloat *temp; temp=NewRho;
	NewRho=Rho; Rho=temp;                     // swap rho and newrho pointers
}
void clCGSolver::solve(int maxI) {
	solveInit();
	for (int i=0;i< maxI;i++) solveIteration();
}
int clCGSolver::solve(float rhoTresh, int maxI) {
	solveInit(); Rho->clone(NewRho);
	for (int i = 0;i< maxI && NewRho.getData() > rhoTresh;i++) solveIteration();
	return i;
}
```

```
[…]
  clMatVec(CL_NOP,A,p,NULL,q);   // q = Ap
  a=r/clVecReduce(CL_ADD,p,q);   // a = r/dot(p,q)
  clVecOp(CL_ADD,1,a,x,p,s);     // s = x+ap
[…]
```

Grid based simulation on GPUs

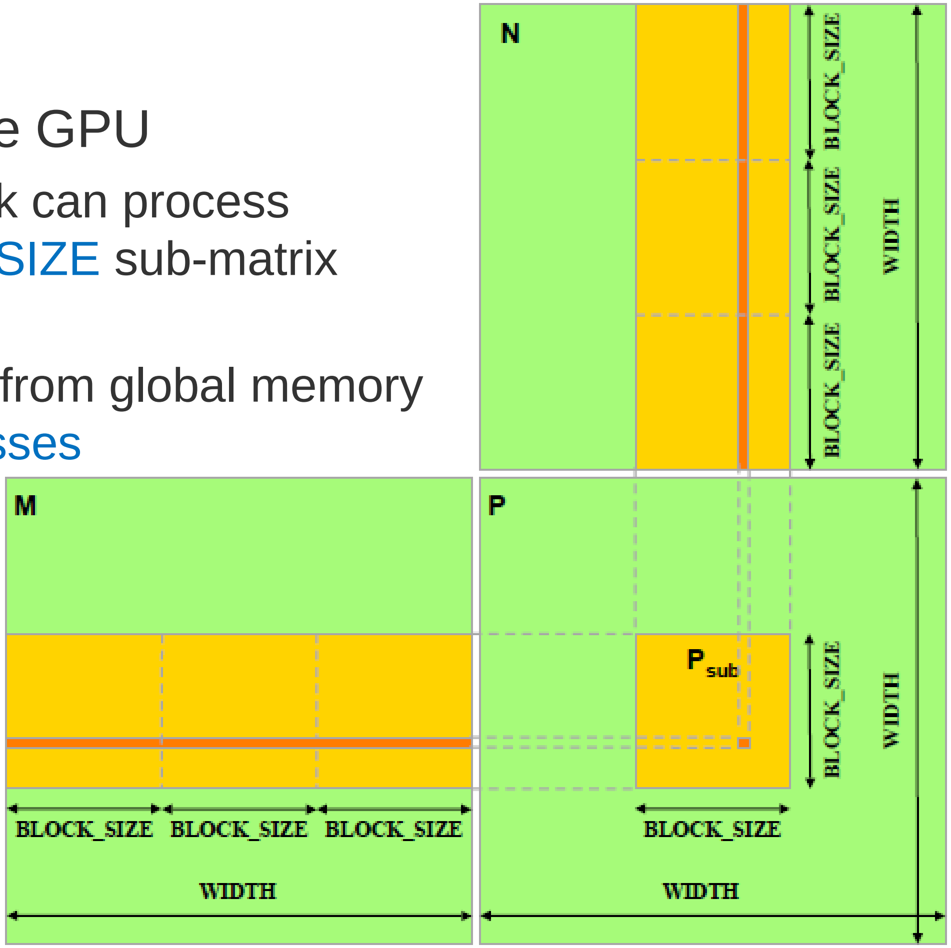
- Dense matrix operations on the GPU
  - Performs $P_{r,c} = M_r \bullet N_c$
  - Can be performed in parallel for all elements in P



http://gpgpu.org/wp/wp-content/uploads/2009/11/SC09_Irregular_Data_Structures_Owens.pdf

Grid based simulation on GPUs

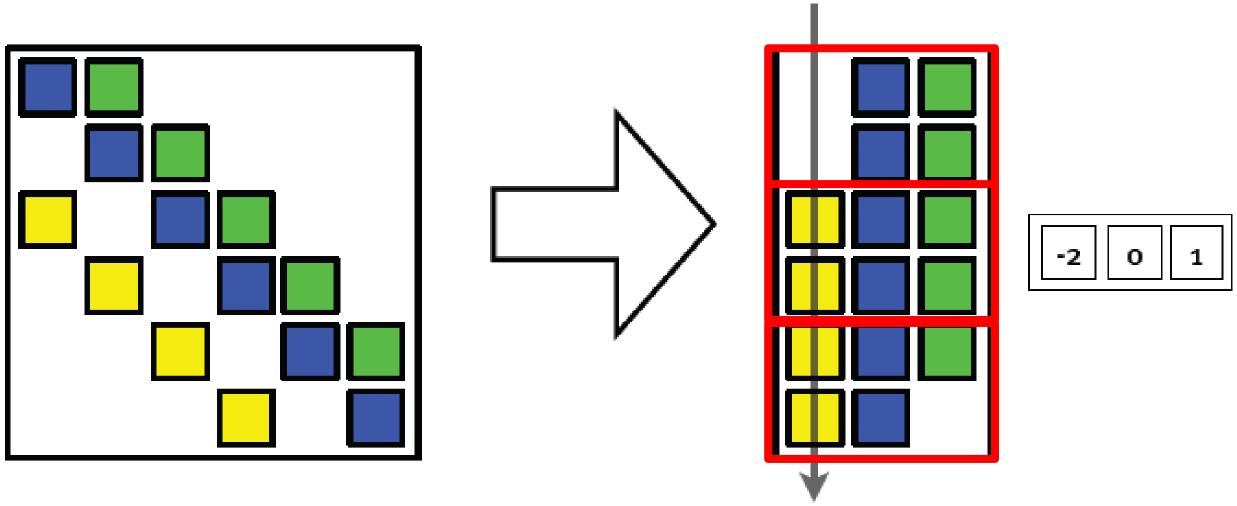
- Dense matrix operations on the GPU
  - Using CUDA, one thread block can process one BLOCK_SIZE x BLOCK_SIZE sub-matrix
  - *M* and *N* are only loaded WIDTH / BLOCK_SIZE times from global memory using coalesce memory accesses
  - Matrix/vector operation by setting WIDTH and BLOCK_SIZE of *N* to 1



http://gpgpu.org/wp/wp-content/uploads/2009/11/SC09_Irregular_Data_Structures_Owens.pdf

Grid based simulation on GPUs

- Banded sparse matrix representation
  - Store “full” rows in linear memory segments, together with “offsets” from the main diagonal
  - Map one (or many) thread blocks per diagonal
    - Coalesced memory reads to load the matrix into shared memory
    - Combination of per-block results in device memory

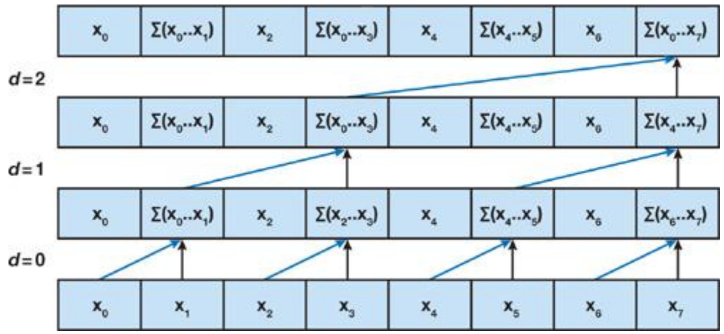


Grid based simulation on GPUs

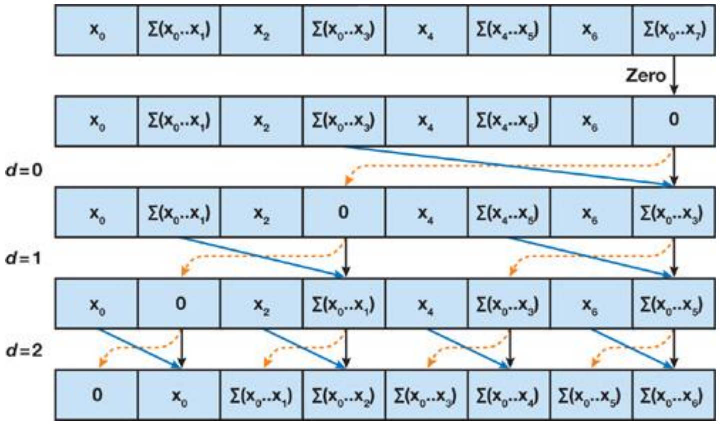
- Sparse matrix representation
  – Store “full” rows in linear memory segments, together with “offsets” from the main diagonal
  – Map one (or many) thread blocks per diagonal
    - Coalesced memory reads to load the matrix into shared memory
    - Combination of per-block results in device memory

Grid based simulation on GPUs

- Vector vector operations
  - Vector-vector multiply via a simple CUDA multiply kernel
  - Scalar product/component sum via parallel log-reduce scan operations: $[a_0, a_1, \ldots, a_{n-1}] \rightarrow [a_0, (a_0 \otimes a_1), \ldots, (a_0 \otimes a_1 \otimes \ldots \otimes a_{n-1})]$
    - Only requires up-sweep phase in our case



up-sweep phase



down-sweep phase

http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Approx.: 4M elements/msec

Grid based simulation on GPUs

- Given the CUDA linear algebra building blocks for matrix and vector operations, a numerical solver like CG can be implemented efficiently on the GPU

Example:
3D Navier-Stokes equations

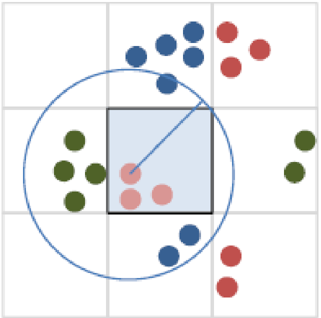
Grid size:
128x128x512

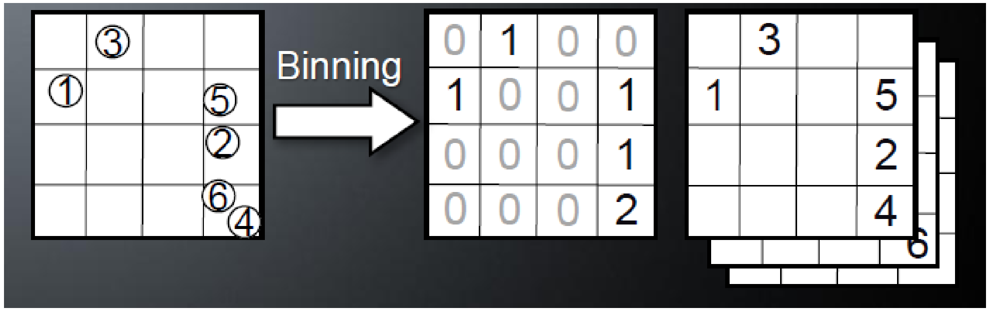
CG iterations
6 it/time step

Simulation rate:
20 time steps/ second

Particle based simulation on GPUs

- Neighbor search on the GPU
  - First attempt using regular space partitions and particle binning
    - A spatial grid divides the domain into uniform cells
    - Particles compute the cell they are contained in and accumulate a 1 at this cell via a scattered write operation
    - For each cell a container large enough to hold all contained particles is allocated and filled
    - Neighbor search involves lookup of (adjacent) cells





Particle based simulation on GPUs

- Neighbor search on the GPU
  - Optimization via scan operation and sorting
    - Sorting (e.g. radix-sort):

      A) Sorting of i-digit numbers

      B) i-th iteration (sorting wrt. the i-th digit):

      1) binning the numbers into buckets 0,..,9 depending on i-th digit

      2)

      $\forall$i:

      compute #numbers in bucket$_i$ $\rightarrow Z_i$

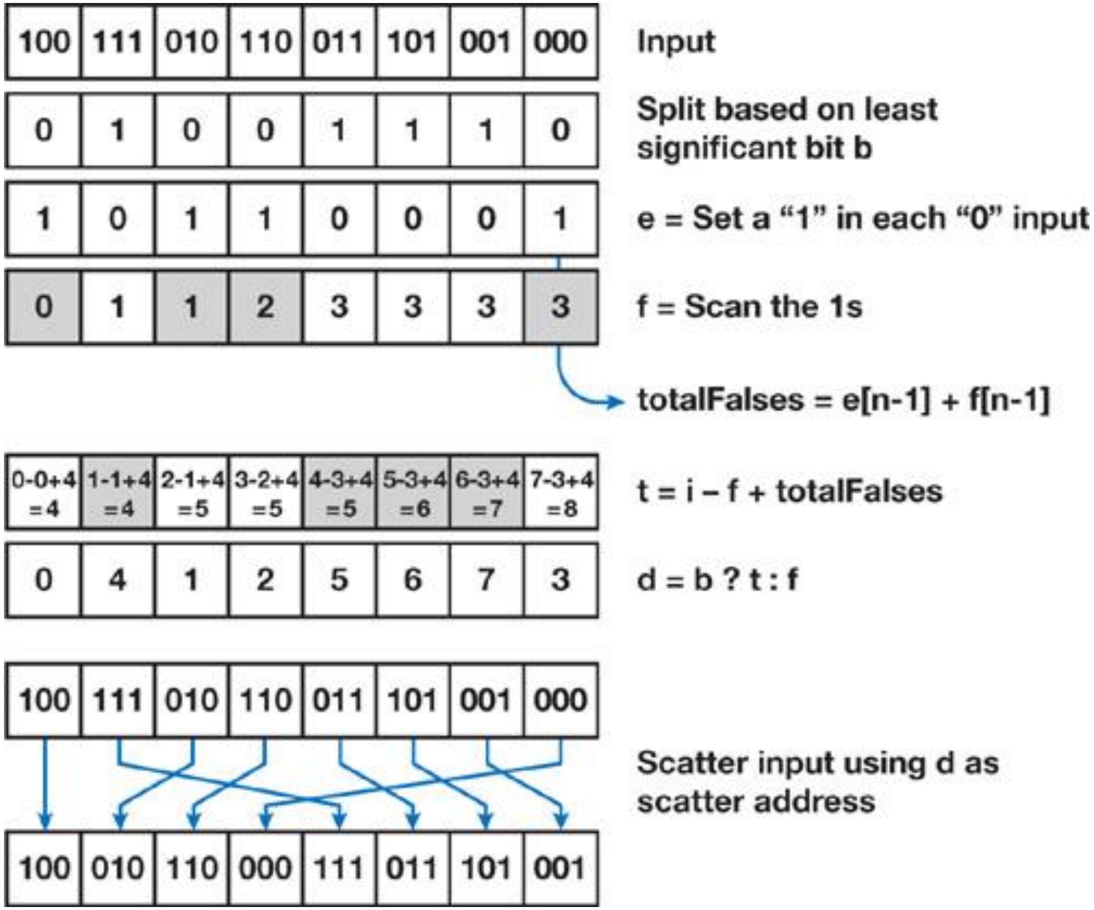
      compute $\Sigma_{(0,i-1)} Z_i$ via scan operation

      3) output (scattered write) into sorted field

Particle based simulation on GPUs

- One Iteration of radix-sort in CUDA

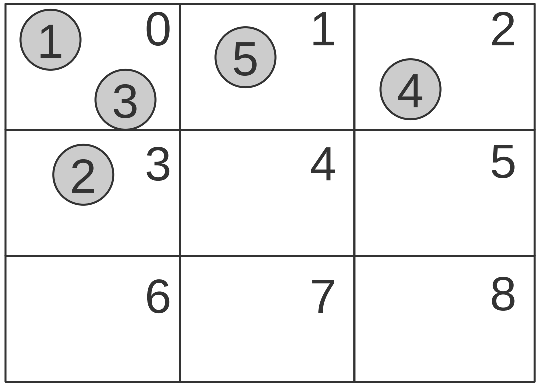
Approx.
20M 32-bit keys/sec

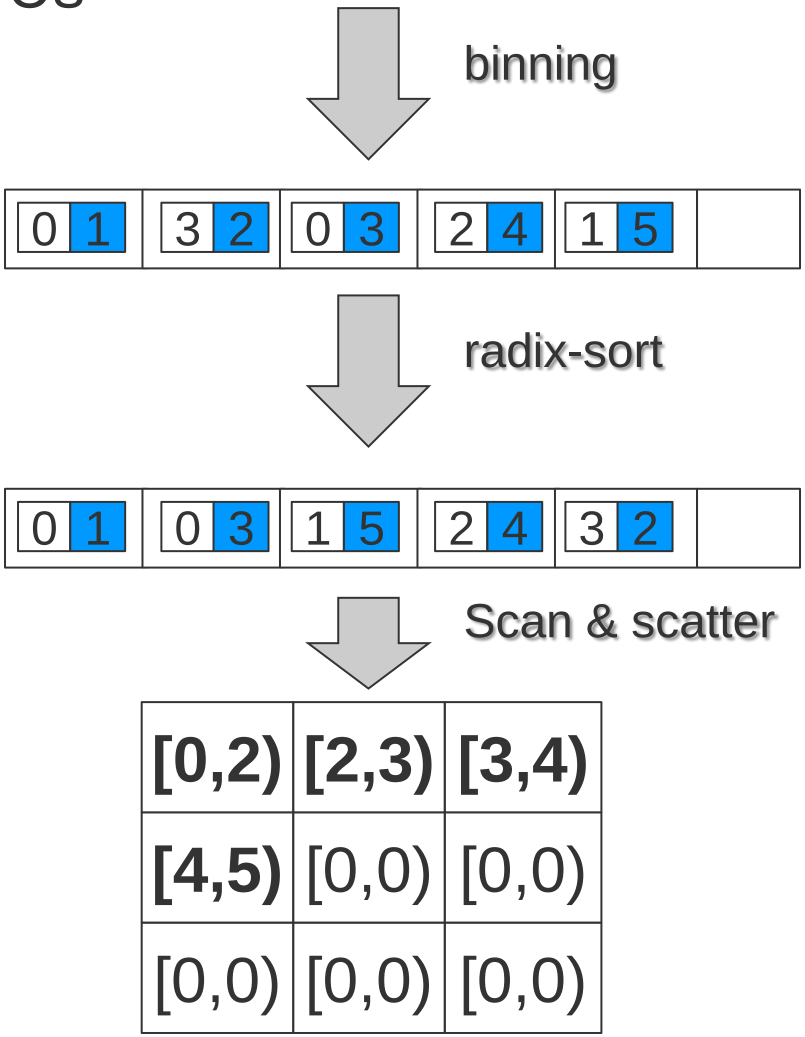


http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Particle based simulation on GPUs

- Neighbor search on the GPU
  - Sorting and scanning





Particle based numerical simulation

- Lessons learned:
  - Numerical simulation via moving particles (Lagrangian approach)
  - Comes down to neighbor search and simple averaging
  - Neighbor search on GPUs using CUDA scan primitives and sorting
  - Efficient parallelization
  - Due to heterogeneous particle density, unbalanced computational load for averaging