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 explicitly for the unknown displacements \( u_{ij} \)

\[
\begin{align*}
\dot{u}_{ij}^{t+1} &= C_1 \cdot \left( u_{i+1,j}^t + u_{i-1,j}^t + u_{ij+1}^t + u_{ij-1}^t \right) + C_2 \cdot u_{ij}^t - u_{ij}^{t-1} \\
\end{align*}
\]

- 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

```cuda
... // 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 arrayu_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 arrayout[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

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

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

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

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

Grid based simulation on GPUs

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

\[
\begin{vmatrix}
4\alpha+1 & -\alpha & -\alpha & \cdots & -\alpha \\
-\alpha & 4\alpha+1 & -\alpha & \cdots & -\alpha \\
-\alpha & -\alpha & 4\alpha+1 & \cdots & -\alpha \\
-\alpha & -\alpha & -\alpha & \cdots & -\alpha \\
-\alpha & -\alpha & -\alpha & \cdots & -\alpha \\
\end{vmatrix}
\begin{vmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
\end{vmatrix}
= 
\begin{vmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
\end{vmatrix}
\]

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

\[
b_{i+j,k} = \alpha \left( u_{i+1,j}^t + u_{i-1,j}^t + u_{i,j+1}^t + u_{i,j-1}^t - 4 \cdot u_{i,j}^t \right) + 2 \cdot u_{i,j}^t - u_{i,j}^{t-1}
\]

\[
x_{i+j, N_x} = u_{i,j}^t
\]

\(N_x, N_y\) : 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

```c
// 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(F, 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, F, 1, Beta); // P = R + beta*P;
NewRho=Rho; Rho=Temp; // swap rho and newrho pointers
}

// q = Ap
a=r/clVecReduce(CL_ADD, p, q);  // a = r/dot(p, q)
// s = x+ap
```

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

```c
…
clMatVec(CL_NOP, A, p, NULL, q);  // q = Ap
a=r/clVecReduce(CL_ADD, 1, a, x, p, s);
…
```
Grid based simulation on GPUs

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

Grid based simulation on GPUs

- Dense matrix operations on the GPU
  - Using CUDA, one thread block can process one $\text{BLOCK\_SIZE} \times \text{BLOCK\_SIZE}$ sub-matrix
  - $M$ and $N$ are only loaded $\text{WIDTH} / \text{BLOCK\_SIZE}$ times from global memory using coalescing memory accesses
  - Matrix/vector operation by setting $\text{WIDTH}$ and $\text{BLOCK\_SIZE}$ of $N$ to 1

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


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

Particle based simulation on GPUs

- Neighbor search on the GPU
  - Sorting and scanning

```
1  0  5  1  4  2
3  4  5
6  7  8
```
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