Designing a Domain-specific Language to Simulate Particles

dan bailey

double negative visual effects
Double Negative

- Largest Visual Effects studio in Europe
- Offices in London and Singapore
- Large and growing R & D team
Squirt Fluid Solver

- Introduced in 2006
- Grid / FLIP - 3D Navier Stokes
- Re-engineering for parallelisation
Dynamo Simulation Engine

- Based on data-flow programming model
- Split logic into data and algorithms
- Initial solver configuration performs setup
- Jet compiler provides parallelisation
Dynamo Data Model

- **Stencil**: Represents base data resolutions and tiling information.
- **Voxels**: Stores layout, indexing and access to data buffers.
- **Grids**: Provides world space transformation data.

* Diagram illustrates relationships and data flows among components.
Dynamo Operator Network

Operators pull data from Data Model

Operator network executed recursively

Operators pull parameters from Parameter Tree
Jet Language and Compiler

- Compiler can compile for CPU or GPU using LLVM 3.2
- Uses X86 backend or NVidia PTX backend
- Launch code also provided by the compiler

Jet Compiler

CPU
- 'parallel-aware' LLVM IR
  - X86 LLVM IR
  - Thread-pool CPU C++
  - clang
  - X86 Machine Code

GPU
- CUDA Runtime
  - Host GPU Launch C++
  - nvptx backend
  - clang
  - PTX Virtual Assembly
  - X86 Machine Code
Grid Simulation

Advection  Emission  Forces  Projection

Particle-in-Cell Simulation

Interpolation  Seeding  Advection  Splatting  Emission  Forces  Projection

FLIP Solve means quantities stored on grids are deltas
Particle Operators

Independent particle operations mean parallelisation is trivial

Presence of race conditions means care must be taken when parallelising
- Each grid tile maps to static contiguous memory
- Regular layout means predictable indexing
- Particle set maps to dynamic contiguous memory
- Irregular layout means frequent re-allocation
Particle Bucketing

- Common technique for handling particles in parallel
- Often used purely as a data acceleration structure (but we also want voxel data)
- Store start, end and length values per voxel
- Can now iterate over particles or over particles per voxel
Mandatory attributes are id and positional
Goals of Jet Language

• Modularity
• Scalability
• Domain-specific
• Simplicity
Example Jet kernel for performing a weighted blur on voxel data:

```c
Weighted_Blur : Voxel<Box> (output, input)
{
    limit(1, 1, 1, 1, 1, 1, 1, 1);

    value = input:(0, 0, 0) * 4
               + input:(-1, 0, 0) + input:(1, 0, 0)
               + input:(0, -1, 0) + input:(0, 1, 0)
               + input:(0, 0, -1) + input:(0, 0, 1);

    return value / 10;
}
```

- Type inference helps with simplicity.
- Colon-bracket operator for accessing relative voxels.
- Aspects of functional programming used to prevent data dependencies.
Interpolate : Particle<Voxel> (output, quantity, flip) {
    // retrieve particle positions
    x = flip:x(0);
    y = flip:y(0);
    z = flip:z(0);

    // calculate voxel indices
    i = x < 0.5f ? global_i() : global_i() - 1;
    j = y < 0.5f ? global_j() : global_j() - 1;
    k = z < 0.5f ? global_k() : global_k() - 1;

    // adjust offsets
    x = x < 0.5f ? x + 0.5f : x - 0.5f;
    y = y < 0.5f ? y + 0.5f : y - 0.5f;
    z = z < 0.5f ? z + 0.5f : z - 0.5f;

    return trilerp(quantity, i, j, k, x, y, z);
}
Splatting

Splat: $\text{Voxel<Particle>}$ (output, flip, attribute)

\begin{verbatim}
limit(1, 1, 1, 1, 1, 1);
neighbour(-1, -1, -1, 1, 1, 1);

x = flip:x(0);

// early exit if particle out of range (2 x 2 x 2)
if (local_i() == -1 && x < 0.5f)    return;

// calculate particle weighting
if (local_i() == -1)
else if (local_i() == 1)
else if (x < 0.5f)
else // (missing code for calculating ty and tz)

// sum the weighted results
return output:(0, 0, 0) + attribute:(0) * tx * ty * tz;
\end{verbatim}

Iterator means modify voxel data with reference to a child particle

Read from many, write to one to avoid race conditions

Neighbour modifier adjusts region in which to iterate over particles
Results

Interpolation

<table>
<thead>
<tr>
<th></th>
<th>CPU (unthreaded)</th>
<th>Jet CPU (8 threads)</th>
<th>Jet GPU (Q4000)</th>
<th>Jet GPU (Tesla K20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>506ms</td>
<td>(6.4x)</td>
<td>(7.2x)</td>
<td>(14.8x)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>500ms</th>
<th>250ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>79.0ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70.2ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34.1ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Splatting

<table>
<thead>
<tr>
<th></th>
<th>CPU (unthreaded)</th>
<th>Jet CPU (8 threads)</th>
<th>Jet GPU (Q4000)</th>
<th>Jet GPU (Tesla K20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6190ms</td>
<td>(7.8x)</td>
<td>(9.3x)</td>
<td>(22.2x)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>6000ms</th>
<th>3000ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>790ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>663ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>283ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Particle Sorting

- Particles must be sorted after being moved
- Rely on subsampling to maintain particle density
- Voxels with too many particles get starved
- Voxels with too few particles get spawned
Inter-tile Sorting

- Identify escaped particles
- Perform stream compaction to select them
- Deduce desired destination tile
- Move particles into spare tile scratch area
Intra-tile Sorting

- Sort each tile independently
- Include any particles in tile scratch area
- Deduce desired destination voxel
- Perform radix sort on voxel indices
Conclusion

- Complex data model compliments atomic algorithms
- Jet DSL provides efficient grid-particle interactions
- Particle sorting currently provided implicitly