GPU-based Distributed Behavior Models with CUDA

Courtesy: YouTube, ISIS Lab, Universita degli Studi di Salerno

Bradly Alicea
Introduction

**Flocking: Reynolds “boids” algorithm.**
* models simple local behaviors (large # of elements).

* need to manage aggregate motion of elements.

**Related work:**
* Navier-Stokes equations (simulation of physical flows).
* robot motion planning.
* ray tracing (computer graphics).

**Implementation:**
* information about neighbor elements = stream data.
  * implemented as fragment programs.
* discrete simulation (behavioral and graphics).
Larger Context

Why do we care about flocking?
Models of complex, collective behavior (e.g. traffic modeling)
* local interactions, global order parameters.

Analogue for emergent processes (e.g. order from chaos, self-organized behavior)
* physical processes (B-Z reactions, morphogenesis).

Models of autonomous, interactive systems (e.g. social, comunication networks)
* local interactions (connectivity) contributes to global properties (stability).

Future work:
* extend to other implementations of k-nearest neighbors search algorithm.

* use scattering matrix, allows for position updates without centralized coordination.
Cellular Automata Models

Cellular Automata: discrete dynamical simulation.

Cells have properties and interaction rules, behave in parallel.

**Properties:** internal state.

**Interaction rules:** if \( n > 2 \) neighbors are red, turn red.

**Parallelism:** all cells use same set of rules, have same properties.

Example: Wolfram’s Rule 30 (1-D lattice)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>current pattern</td>
<td>111</td>
<td>110</td>
<td>101</td>
<td>100</td>
<td>011</td>
<td>010</td>
<td>001</td>
<td>000</td>
</tr>
<tr>
<td>new state for center cell</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Below: 2-D von Neumann neighborhood, order 1

Rule 30 - model  
Rule 30 - nature
Defining Nearest-Neighbor in Parallel

Agent-based, metric distance, and topological distance models:

Generative (agent-based) Model

Local parameters, data handling

Cellular Automata-like:
* coupled-map lattice.

Space partitioning data structure (CPU):

1) gather data on neighbors.
2) update at every frame (movement).

Synchronized aggregated motion of flocks:

1) fix one or more spatial goals (containment, obstacle avoidance).
2) leader following (whoever the leader is, determines where flock goes).
Data architecture

Left: architecture of behavior. “Personality” (profile of each boid) blended and determines global parameters.

Below: scattering matrix. Extracts global parameters from “uniform shape”.

Figure 3: Architecture of behavior model on the GPU.

Figure 4: Scattering matrix: it is used to track down how much the flock has a uniform shape after each frame calculation. The cells are numbered with triple meaning indicating in which direction the boid is moving.
Program Structure

The entire process takes as input four textures: 1) a constant texture $T_c$ to store scalar information as mass, maximum velocity and maximum acceleration, 2) a texture $T_o$ for orientation (three scalar values), 3) a texture $T_p$ for position (three scalar values) and current velocity (one scalar value), 4) a texture $T_n$ to store the four nearest boids.

We define the execution of a fragment program $[T_1, \ldots, T_n] \leftarrow $ FRAGMENT $\leftarrow [\hat{T}]$ as the operation that binds the fragment, sets up the texture parameters $T_1, \ldots, T_n$ and draws the geometry onto the output texture $\hat{T}$ using off-screen rendering (called p-buffer). The entire process is executed in the following steps:

1. Prepare the input texture $T_c, T_o, T_p, T_n$
2. $[T_p, T_o] \leftarrow $ CONTAINMENT $\leftarrow [T_s1]$
3. $[T_p, T_o] \leftarrow $ LEADER FOLLOWING $\leftarrow [T_s2]$
4. $[T_p, T_o] \leftarrow $ FLOCKING BEHAVIOR $\leftarrow [T_s3]$
5. For every obstacle $i$ with texture field $T_{f4}$
   - $[T_p, T_o, T_s4, T_{f4}] \leftarrow $ OBSTACLE AVOIDANCE $\leftarrow [T_s4]$
6. $[T_s1, T_s2, T_s3, T_s4] \leftarrow $ PERSONALITY BLEND $\leftarrow [T_s]$
7. $[T_c, T_s] \leftarrow $ ACCELERATION $\leftarrow [T_s]$
8. $[T_c, T_a] \leftarrow $ POSITION VELOCITY $\leftarrow [T_p]$
9. $[T_c, T_a] \leftarrow $ ORIENTATION $\leftarrow [T_o]$

GPU: runs fragment programs, can use “stream process”
* stream data arranged as textures.

* handles scene rendering, updates.

CPU: used for spatial sorting only.

Four textures:
$T_c = $ constant texture (global scalars – max. velocity, max. acceleration).

$T_o = $ orientation (3 scalar variables – forward, side, up).

$T_p = $ position (3 scalar values – x,y,z).

$T_n = $ nearest-neighbors (von Neumann neighborhood).
Example: N-body problem for CUDA

GPU Gems, #31:

Chapter 31

Fast N-Body Simulation with CUDA

Lars Nyland
NVIDIA Corporation
Mark Harris
NVIDIA Corporation
Ian Priml
University of North Carolina at Chapel Hill

Figure 31-1. Frames from an interactive 3D Rendering of a 16,384-Body System Simulated by Our Application.
We compute more than 10 billion gravitational forces per second on an NVIDIA GeForce 8800 GTX GPU, which is more than 50 times the performance of a highly tuned CPU implementation.
What is the n-body problem?

N-body = every body in set n interacts (potentially) with all other bodies.

Basic computational features:
1) all-pairs approach (unlike nearest-neighbor or structured topology).
2) $O(N^2)$ time.
3) encoded as a kernel, determines forces in close-range interactions.

Previous approaches:
* fast multipole methods (FMMs).
* particle mesh methods.
CUDA Implementation

* calculate each entry $f_{ij}$ in an $N \times N$ grid of all pair-wise forces.

* total force $F_i$ (acceleration $a_i$) on body $I \rightarrow \Sigma row_i$

* each entry computed independently, $O(N^2)$ available parallelism.

Code example #1: 31.1, Updating acceleration of one body as a result of its interaction with another body.

Code example #2: 31.2, Evaluating interactions in a $p \times p$ tile.

Code example #3: 31.3, Listing 31-3. The CUDA kernel executed by a thread block with $p$ threads to compute the gravitational acceleration for $p$ bodies as a result of all $N$ interactions.
Serialize some computations:
* achieve data reuse (bandwidth-performance tradeoff).
* computational tile (square region - pxp in size - subset of pairwise forces).
* 2p body descriptions, p^2 interaction within tile, total effect = update to p acceleration vectors.

Body-body Force:
* compute force on body i from interaction with body j updates acceleration a_i of body i resulting from interaction.
* float4 data type (CUDA-specific), stored in GPUs device memory -- body mass = w field (allows coalesced memory access to arrays of data -- efficient memory request transfers).
Other details

**myPosition**: input parameter that holds position of body for executing thread.

**shPosition**: array of body descriptions in shared memory.

**thread**: iterates over same $p$ bodies.

**$p$ threads**: execute function body in parallel.

**In summary:**
* acceleration computed on individual body as a result of interaction with $p$ other bodies.
* tile evaluated by $p$ threads -- same sequence of operations on different data.
* each thread updates acceleration of one body as a result of interactions with $p$ other bodies.
* load $p$ body descriptions from GPU device memory into shared memory provided to each thread block in CUDA model.
* each thread in block evaluates $p$ successive interactions -- $p$ updates accelerations.
Nearest-Neighbor Searches (instance of n-body problem) using kd-trees
NN Search Definitions

Vocabulary:

**NN** - Nearest Neighbor

**kNN** – ‘k’ nearest neighbors

Definitions:

- $d$ is the number of dimensions
- $S$ is a search set containing ‘$n$’ points
- $Q$ is a query set containing ‘$m$’ points
- $\text{dist}(a,b)$ is a distance metric between two points

$$\text{dist}(a,b) = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + \cdots + (b_d - a_d)^2}$$

All-NN: All Nearest Neighbor

Find the closest point in $S$ for each point in $S$ by $\text{dist}(p,q)$.

**Input:** $S$ ($Q \leftrightarrow S$)

**Output:** List of $n$ indices in $S$.

Note: Exclude zero distance results

All-kNN: All ‘$k$’ Nearest Neighbors

Find the $k$ closest points in $S$ for each point in $S$ by $\text{dist}(p,q)$.

**Input:** $S$ ($Q \leftrightarrow S$)

**Output:** List of $km$ indices in $S$.

Note: Exclude zero distance results

RNN: Range Query

ANN: Approximate Nearest Neighbor

Courtesy: Brown and Snoeyink, GPU Nearest-Neighbor searches using a minimal kd-tree. Slides on Internet.
NN search Solutions

Linear Search:
Brute force solution, compare each query point to all search points
\[ O(mn) \]

Spatial Partitioning Data Structures:
Divide space into smaller spatial cells. Use “branch and bound” to focus on productive cells.
Examples: kd-tree, Quad-tree, Grid, Voronoi Diagram, …

Spatial Partitioning:
subdivide space

Data Partitioning:
subdivide data into sets

kd-tree

Invented by J.L. Bentley, 1975

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Points (more complicated objects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical</td>
<td>Corresponds to a binary tree</td>
</tr>
<tr>
<td>Axis aligned spatial cells</td>
<td>Each cell (\rightarrow) node of the binary tree | The root cell contains the original bounds and all points</td>
</tr>
</tbody>
</table>
| Recursively defined | \\(\text{Divide each cell into left and right child cells starting from the root.}\\
| Splitting Heuristics | \\(\text{The points associated with each cell are also partitioned into the left and right child cells}\\
| Data Partitioning | Form a cutting plane (pick split axis & split value) |
| Space Partitioning | Median Split \| Empty space maximization \| Surface Area, Voxel volume, etc. |

Add root cell to build queue
While build queue not empty
- grab current cell from build queue
- Pick a cutting plane (via median split)
- Subdivide current cell
  - Termination “Do nothing” \(< m\) points in cell
  - Split parent bounds into left & right cells
  - Partition parent points into left & right cells
  - Add left & right cells to build queue

Storage: \(O(dn)\)
Build Time: \(O(dn \log n)\)
**GTX 285 Architecture**

**Device Resources**
- 30 GPU Cores
  - 240 total thread processors
- 1 GB on-board RAM
- 32 KB constant memory

**GPU Core**
- 8 physical thread processors per core
- 1 double precision unit per core
- 16 KB shared memory
- 8,192 shared 32-bit registers

**Thread Processor**
- Shares resources (memory, registers) in same GPU core

**Hardware**
- Thread blocks start & stay with initial core
- Thread block finishes when all threads finish
- Multiple blocks get mapped to each core
- One GPU core can execute several blocks concurrently depending on resources
- Maximum of 512 threads per thread block

**Thread Block**
- Threads blocks executed on GPU cores
- Supports syncing of threads within a block

**Grid**
- A kernel is launched as a 1D or 2D grid of thread blocks
- Only one kernel can execute on a GPU device at a time
- Syncing across blocks not supported*
**Timings (in ms)**

Uniform Random data
On range [0,1] for each axis

### QNN search on GPU (CPU)

<table>
<thead>
<tr>
<th>$n$</th>
<th>2D</th>
<th>3D</th>
<th>4D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.07 (0.10)</td>
<td>0.18</td>
<td>0.41</td>
</tr>
<tr>
<td>10,000</td>
<td>0.42 (12.46)</td>
<td>1.02</td>
<td>2.12</td>
</tr>
<tr>
<td>100,000</td>
<td>4.17 (156.20)</td>
<td>10.10</td>
<td>23.10</td>
</tr>
<tr>
<td>1,000,000</td>
<td>45.62 (2,001.20)</td>
<td>111.34</td>
<td>247.47</td>
</tr>
<tr>
<td>10,000,000</td>
<td>668.07 (26,971.21)</td>
<td>1,614.34</td>
<td>3,840.73</td>
</tr>
</tbody>
</table>

(85.54 s)

### All-$k$NN search on GPU (CPU), $k = 31$

<table>
<thead>
<tr>
<th>$n$</th>
<th>2D</th>
<th>3D</th>
<th>4D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1.01 (0.10)</td>
<td>1.64</td>
<td>2.64</td>
</tr>
<tr>
<td>10,000</td>
<td>5.88 (12.46)</td>
<td>12.57</td>
<td>28.73</td>
</tr>
<tr>
<td>100,000</td>
<td>57.04 (156.20)</td>
<td>123.74</td>
<td>291.26</td>
</tr>
<tr>
<td>1,000,000</td>
<td>579.57 (10,127.02)</td>
<td>1,270.45</td>
<td>2,999.02</td>
</tr>
</tbody>
</table>

---

**Increasing $n,m$; Increasing $k$**

---

**QNN, All-NN**
The optimal thread block size is $10x1$ for $n,m=1$ million points

---

**kNN, All-kNN**
The optimal thread block size is $4x1$ for $n,m=1$ million points, $k=31$

---

**Increasing $n,m$; $n \leq 100$, use CPU
$n \geq 1000$, use GPU**

---

**Increasing $k$ (kNN, All-kNN)**
Divergence on GPU gradually hurts performance
**SIMPLIFY**

**Future Directions**
- Streaming Neighborhood Tool
  - Apply operators on local neighborhoods (billions of points)
- Build on GPU
  - Attempted works but is slower than CPU solution
  - Use coalescence, increase # of threads
  - Need different approaches for startup, middle, and wind-down phases to get enough parallelism
- Compare & contrast against other NN solutions
  - CGAL, GPU Quadtree, GPU Morton Z-order sort
- Improve Search performance
  - Store top 5-10 levels of tree in constant memory
  - All-NN, All-kNN rewrite search to be bottom-up
- Improve code
  - Use ‘Templates’ to reduce total amount of code

**COMPRESS**

**GPU TIPS & Tricks**
- Develop methodically
- Minimize I/O’s
- Tweak kernels for better performance
- Use aligned data structures (4,8,16)
- Use **Locked** Memory I/O
- Compress Data Structures
- Structure of Arrays (SOA) vs. Array of Structures (AOS)

---

**BREAK APART**

- Consider the lowly Stack
  - 16K of **shared** memory
  - 16K/32 = 512 bytes per thread
  - 32 * 8 bytes = 256 bytes
  - We have just enough room for a simple 32 element stack with two 32-bit fields per stack object on each thread
  - This is enough to handle a binary tree of \(2^{32} = 4\) gig elements

- **Memory accesses are slow**
- **Local calculations are fast**
- Paying the cost of compression/decompression calculations to reduce memory I/O can increase performance.

```c
typedef struct __align__(16) {
    unsigned int nodedX;
    unsigned int splitAxis;
    unsigned int InOut;
    float splitValue;
} KDSearch_CPU;
```

```c
typedef struct __align__(8) {
    unsigned int nodeFlags;
    // Node ldx (29 bits)
    // split Axis (2 bits)
    // InOut (1 bit)
    float splitValue;
} KDSearch_GPU;
```

- **Structure of Arrays vs. Array of Structures**
  - Try both and use which ever gives you better performance
  - 8 field (64 byte) KDNode structure
  - Managed to compress it to 5 fields (40 bytes) but couldn’t compress further.
  - Broke it into 2 data structures
    - KDNode: 4 fields __align 16__(pos[x,y], left, right)
    - IDNode: 1 field __align 4__ (ID)
- **Surprising Result:**
  - The algorithm had a **3x-5x speed increase** as a result of this one change alone