GPU Accelerated Pathfinding

Avi Bleiweiss
NVIDIA Corporation
Introduction

- Navigation planning
  - Global and local
- Crowded game scenes
  - Many thousands agents
- Decomposable movement
  - Explicit parallelism
- Dynamic environment

Roadmap Generation

Global Navigation

Local Navigation

environment

cost, path
Motivation

- CUDA compute enabling
- Nested data parallelism
- Irregular, divergent algorithms
  - Large thread SIMD challenge
- Extend GPU game computing
  - Core game AI actions

Flat Data Parallel
serial operation on bulk data

Nested Data Parallel
parallel operation on bulk data
Objective

- Optimally navigate agents
  - From start to goal state
- Roadmap representation
  - Graph data structure
- Parallel, arbitrary search
  - Varying topology complexity
- GPU performance scale
Outline

- Algorithm
- Implementation
- Performance
- Futures
Algorithm
Graph

- Linked set of nodes, edges
  - \( G = \{N, E\} \)
- Dense or sparse
  - Edges (E) to nodes (\(N^2\)) ratio
- Directed or undirected
  - Ordered, unordered node pairs
- Consistent data structure
Data Structure

- **Adjacency matrix**
  - Intuitive edge presence
  - Wasteful for sparse graphs $O(N^2)$

- **Adjacency lists**
  - Immediate node indices
  - Compact storage $O(N+E)$

- **Roadmap sparse graph**
  - Adjacency lists default
Search

- Feasibility and optimality
- Planning in state space
  - Unvisited, dead, or alive
- Priority queue alive states
- Cost based state
- Running time complexity
  - Worse than linear

Forward Search Algorithm Template

1: Q.Insert(n_S) and mark n_S as visited
2: while Q not empty do
3:     n ← Q.Extract()
4:     if(n == n_G) return SUCCESS
5:     for all u ∈ U(n) do
6:         n′ ← f(n, u)
7:         if n′ not visited then
8:             Mark n′ visited
9:             Q.Insert(n′)
10:        else
11:            Resolve duplicate n′
12:        return FAILURE

- alive nodes are placed on a priority queue Q
- n_S and n_G are start and goal positions, respectively
- u is an action in a list of actions U
- f(n, u), state transition function
- n is current node and n′ the next adjacent node.
Algorithms

- Cost based search
- Priority queue sort function
- Search properties:

<table>
<thead>
<tr>
<th>Search</th>
<th>Start</th>
<th>Goal</th>
<th>Heuristic</th>
<th>Optimal</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best First</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>fair</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>slow</td>
</tr>
<tr>
<td>A*</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes*</td>
<td>fast</td>
</tr>
</tbody>
</table>

* assumes admissible heuristic

- Dijkstra, A* without heuristic
Heuristic

- Admissible = optimistic
  - Never overestimate cost-to-goal
- A* with admissible heuristic
  - Guarantees optimal path
- Narrows search scope
- Suboptimal, weighted heuristic
  - Quality vs. efficiency tradeoff

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manhattan</td>
<td>$w \times \text{abs}(n_g - n)$</td>
</tr>
<tr>
<td>Diagonal</td>
<td>$w \times \text{max(abs}(n_g - n))$</td>
</tr>
<tr>
<td>Euclidian</td>
<td>$w \times \text{sqr}(\text{square}(n_g - n))$</td>
</tr>
</tbody>
</table>

$n$ – position vector state
**A***

- Irregular, highly nested
- Priority queue element
  - {node index, cost} pair
- Memory bound
  - Extensive scatter, gather
- Low arithmetic intensity
  - Embedded in heuristic
- Unrolling inner loop

```
1:  f = priority queue element {node index, cost}
2:  F = priority queue containing initial f (0,0)
3:  G = g cost set initialized to zero
4:  n = closest node index
5:  E = node adjacency list
6:  while F not empty do
7:      n ← F.Extract()
8:      S[n] ← P[n]
9:      if n is goal then return SUCCESS
10:  foreach edge e in E[n] do
11:      h ← heuristic(e.to, goal)
12:      g ← G[n] + e.cost
13:      f ← {e.to, g + h}
14:      if not in P or g < G[e.to] and not in S then
15:          F.Insert(f)
16:          G[e.to] ← g
17:          P[e.to] ← e
18:  return FAILURE
```

Cost notation:
- $g(n)$: cost from start to node $n$
- $h(n)$: heuristic cost from $n$ to goal
- $f(n, \text{cost})$: combined cost of $g(n)$ and $h(n)$
Implementation
Software

- Game AI workloads
- GPU, CPU invocation paths
- Orthogonal multi core semantics
- CUDA graceful multi launch
- Scalar C++, SIMD intrinsics (SSE)
Tradeoffs

- Shared roadmap caching
- Working set coalesced access
- Efficient priority queue operations
- Divergent kernel parallel execution
- CUDA profiler for optimization
Roadmap Textures

- Linear device memory
  - Texture reference binding
- Flattened edge list
  - With adjacency directory
- Adjacency list cacheable
- Loop control direct map
- 2 or 4, 32 bit components
Working Set

- Thread local storage
- Global memory regions
- \(O(T*N)\) storage complexity
- Node sized arrays
- 4, 8, 16 bytes data structures
- Exceeding available memory

\[ O(T^*N) \]

\[ O(T) \]

\[ O(T) \]

\[ O(T^*) \]

T - threads, N - roadmap nodes

<table>
<thead>
<tr>
<th>Inputs</th>
<th>List</th>
<th>Definition</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paths</td>
<td>start, goal positions</td>
<td>user defined</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>cost from-start</td>
<td>zero</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>sum of costs from-start, to-goal</td>
<td>zero</td>
<td></td>
</tr>
<tr>
<td>P, S</td>
<td>visited, dead node (edge)</td>
<td>zero</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>List</th>
<th>Definition</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costs</td>
<td>accumulated path cost</td>
<td>zero</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>subtree, plotted waypoints</td>
<td>zero</td>
<td></td>
</tr>
</tbody>
</table>
Coalescing

- Strided memory layout
  - Node id fast axis
- Interleaved organization
  - Thread id running index
- Contiguous memory access
  - Across a thread warp
- Array indexing inexpensive
Priority Queue

• Element pairs
  • Cost, node id
• Fixed size array
• Heap based
  • Logarithmic running cost
• Operation efficiency
  • Insertion, extraction
• Insertions dominates
  • Early success exit

```
1: __device__ void
2: insert(CUPriorityQ* pq, CUCost c)
3: {
  4:   int i = ++(pq->size);
  5:   CUCost* costs = pq->costs;
  6:   while (i > 1 && costs[i>>1].cost > c.cost) {
  7:     costs[i] = costs[i>>1];
  8:     i >>= 1;
  9:   }
10:   pq->costs[i] = c;
11: }
```

```
1: __device__ CUCost
2: extract(CUPriorityQ* pq)
3: {
  4:   CUCost cost;
  5:   if (pq->size >= 1) {
  6:     cost = pq->costs[1];
  7:     pq->costs[1] = pq->costs[pq->size--];
  8:     heapify(pq);
  9:   }
10:   return cost;
11: }
```
Execution

- CUDA launch scope
  - Consult device properties
- An agent constitutes a thread
- One dimensional grid of
  - One dimension thread blocks
- Kernel resource usage
  - 20 registers
  - 40 shared memory bytes

<table>
<thead>
<tr>
<th>CUDA Occupancy Tool Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per block</td>
</tr>
<tr>
<td>Registers per block</td>
</tr>
<tr>
<td>Warps per block</td>
</tr>
<tr>
<td>Threads per multiprocessor</td>
</tr>
<tr>
<td>Thread blocks per multiprocessor</td>
</tr>
<tr>
<td>Thread blocks per GPU (8800 GT)</td>
</tr>
</tbody>
</table>
Performance
Experiments

• Roadmap topology complexity (RTC)
• Fixed, varying agent count
• Dijkstra and non weighted, A* search
• SSE, multi core CPU scale
• CUDA interleaved kernel
• GPU timing includes copy
Benchmarks

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
<th>Agents</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>G1</td>
<td>32</td>
<td>178</td>
<td>1024</td>
<td>8</td>
</tr>
<tr>
<td>G2</td>
<td>64</td>
<td>302</td>
<td>4096</td>
<td>32</td>
</tr>
<tr>
<td>G3</td>
<td>129</td>
<td>672</td>
<td>16641</td>
<td>131</td>
</tr>
<tr>
<td>G4</td>
<td>245</td>
<td>1362</td>
<td>60025</td>
<td>469</td>
</tr>
<tr>
<td>G5</td>
<td>340</td>
<td>2150</td>
<td>115600</td>
<td>904</td>
</tr>
<tr>
<td>G6</td>
<td>5706</td>
<td>39156</td>
<td>64–9216</td>
<td>1–72</td>
</tr>
</tbody>
</table>

- G0–G5: small to moderate RTC (<500 nodes)
- G6: large graph (>5000 nodes)
## Processors

- **Processor properties:**

<table>
<thead>
<tr>
<th>Property</th>
<th>Intel Core 2 Duo</th>
<th>AMD Athlon 64 X2</th>
<th>8400 M</th>
<th>8800 GT</th>
<th>GTX 280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Clock</td>
<td>2000</td>
<td>2110</td>
<td>400</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>Shader Clock</td>
<td>NA</td>
<td>NA</td>
<td>550</td>
<td>1500</td>
<td>1300</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>1180</td>
<td>667</td>
<td>400</td>
<td>900</td>
<td>1000</td>
</tr>
<tr>
<td>Global Memory</td>
<td>2048</td>
<td>2048</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Memory Bus</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Multiprocessor</td>
<td>1 per core</td>
<td>1 per core</td>
<td>8</td>
<td>14</td>
<td>30</td>
</tr>
</tbody>
</table>

Notes: Clocks and memory size in millions.
Footprint

- Working set $O(T*N)$ dominates roadmap $O(N+E)$
- Per thread local 0.33–13.6 KB (G0–G5), 230 KB (G6)
A* higher arithmetic intensity improves speedup
Multi Core

- Quad core vs. dual core speedup: 1.05X (G0–G5), 1.2X (G6)
**Cross GPU**

- **A*, Euclidian**

  ![Graphs](image)

  GPU speedup vs. a single core CPU, SSE optimized code for RTC G0–G5, fixed agent #

- **A*, Euclidian**

  ![Agents](image)

  GPU speedup vs. a single core CPU, SSE optimized code for RTC G6, ascending agent #

- GTX 280 vs. 8800 GT speedup up to 2X
Running Time

<table>
<thead>
<tr>
<th>Parameter</th>
<th>G5</th>
<th>G6°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total running time (seconds)</td>
<td>2.495</td>
<td>6.136</td>
</tr>
<tr>
<td>Average search time (seconds)</td>
<td>0.000021</td>
<td>0.000665</td>
</tr>
<tr>
<td>Average points per path</td>
<td>12.6576</td>
<td>15.8503</td>
</tr>
</tbody>
</table>

° agent count: 9216

- **Unlocked copy overhead**
  - Host-to-Device (up to 50%)
  - Device-to-Host (less than 5%)
Limitations

- Small agent count
- Unlocked copy expensive
  - Pinned memory 1.6X overall speedup
- Software memory coalescing
  - Limited, 1.15X performance scale
- Multi GPU linear scale
  - Replicated roadmap expense
- Weighted A* oscillating
Futures
Futures

- Working set greedy allocation
  - Dynamic, CUDA kernel malloc
- Global memory caching
- Kernel spawning threads
  - Unrolled A* inner loop
- Realigning agent blocks
- Local navigation
Conclusions

• Global navigation scalable
• GPU efficient search for
  • Many thousands agents
• Nested data parallelism
  • Evolving GPU opportunity
• GPU preferred platform
  • Integrating core game AI
Thank You!

Questions?