GPU Accelerated Pathfinding

By: Avi Bleiweiss

NVIDIA Corporation

Graphics Hardware (2008)

Editors: David Luebke and John D. Owens

NTNU, TDT24 Presentation by Lars Espen Nordhus
**Introduction**
One of the more challenging problems in real time games is autonomous navigation and planning of many thousands of agents in a scene with both static and dynamic moving obstacles.

Ideally, we would want each agent to navigate independently without implying any global coordination or synchronization of all or a subset of the agents involved.

The global path is computed using the roadmap graph that represents static objects in the scene without the presence of agents.

The integration of global and local planning is accomplished by computing a preferred velocity vector for each agent that is in the direction of the next node along the agent’s global path.

**Graph**
Adjacency matrix representation is a two dimensional array of Booleans that stores graph topology for a non weighted graph. The matrix has the added property of quickly identifying the presence of an edge in the graph. However, for large sparse graphs the adjacency matrix tends to be wasteful.

Adjacency lists are commonly preferred providing a compact storage for the more wide spread sparse graphs at the expense of a lesser traversal efficiency. Adjacency lists data structure is more economically extensible and adapted to represent weighted graphs for traversing an edge that is associated with a cost property for moving from one node to another.

**Search**

<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>BFS</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>

A* search appears more efficient in balancing both the cost from start and to the goal in determining the best path; A* without heuristic degenerates to Dijkstra’s algorithm.

For A* to be optimal the heuristic needs to admissible. Admissible means optimistic in the sense that the true cost will be at least as great as the estimate.
1: \( f = \text{priority queue element \{node index, cost\}} \)
2: \( F = \text{priority queue containing initial } f(0,0) \)
3: \( G = \text{g cost set initialized to zero} \)
4: \( P, S = \text{pending and shortest nullified edge sets} \)
5: \( n = \text{closest node index} \)
6: \( E = \text{node adjacency list} \)
7: while \( F \) not empty do
8: \( n \leftarrow F.\text{Extract()} \)
9: \( S[n] \leftarrow P[n] \)
10: if \( n \) is goal then return SUCCESS
11: foreach edge \( e \) in \( E[n] \) do
12: \( h \leftarrow \text{heuristic}(e.\text{to}, \text{goal}) \)
13: \( g \leftarrow G[n] + e.\text{cost} \)
14: \( f \leftarrow \{e.\text{to}, g + h\} \)
15: if not in \( P \) or \( g < G[e.\text{to}] \) and not in \( S \) then
16: \( F.\text{Insert}(f) \)
17: \( G[e.\text{to}] \leftarrow g \)
18: \( P[e.\text{to}] \leftarrow e \)
19: return FAILURE

Above you see the A* algorithm pseudo code: \( g(n) \) is the cost from start to node \( n \), \( h(n) \) is the heuristic cost from node \( n \) to goal; \( f \) is the entity to sort in the priority queue, its cost member is the sum of \( g(n) \) and \( h(n) \).

The top element of the queue is extracted, moved to the resolved shortest node (or edge) set and if the current node position matches the goal the search terminates successfully. A common mistake is to check for a positive match when adding a neighbour, thus destroying the optimality of the algorithm.

CUDA

From [http://www2.engr.arizona.edu/~yangsong/gpu.htm](http://www2.engr.arizona.edu/~yangsong/gpu.htm)
Roadmap Textures

The sparse roadmap graph is encapsulated in an adjacency lists data structure. Being read-only the graph is stored as a set of linear device memory regions bound to texture references.

Texture access in the pathfinding kernel uses consistently CUDA’s preferred and efficient `tex1Dfetch()` family of functions.

The roadmap graph storage set has been intentionally refitted to enhance GPU coherent access. The set of textures includes a node list, a single edge list that serializes all the adjacency lists into one collection of edges, and an adjacency directory that provides index and count for a specific node’s adjacency list.

The adjacency directory entry pair maps directly onto A*’s inner loop control parameters. As a result, one adjacency texture access is amortized across several fetches from the edge list texture. Nodes and edges are stored as four IEEE float components and the adjacency texture is a two integer component texture.

```
node
| id   | position.x | position.y | position.z |
edge
| from | to         | cost       | reserved   |
adjacency
| offset | offset+count |
```

Above you see that the roadmap graph texture set are of either four or two components to comply with CUDA’s `tex1Dfetch()` function. Component layout shown has the node with a unique identifier and a three component IEEE float position; an edge has a direction node identifier pair `{from, to}`, a float cost, and a reserved field; adjacency is composed of an offset into the edge list and a count of edges in the current adjacency list.

This layout incurs an extra cost of 8*N bytes compared to an equivalent CPU implementation; in return, it contributes to a more efficient roadmap traversal.

Working Set

The A* kernel has five inputs and two outputs that collectively form the working set.

Input:
  - A list of paths, each defined by a start and a goal node id, one path per agent.
  - A list of costs from the start position (G), initialized to zero.
  - A list of costs combined from start and to goal (F), initialized to zero.
  - A pair of lists of pointers for each the pending and the shortest edge collections P and S, respectively. Initialized to zero.

Output:
  - A list of accumulated costs for the kernel resolved optimal path, one scalar cost value for each agent.
  - A list of subtrees, each a collection of three dimensional node positions, that formulate the resolved plotted waypoints of an agent.

The involved data structures are memory aligned with the size of any of 4, 8 or a maximum of 16 bytes to limit multiple load and store instructions per memory transfer. Arranging global memory
addresses, simultaneously issued by each thread of a warp, into a single contiguous, memory aligned transaction is highly desirable for yielding optimal memory bandwidth. Coalesced 4 byte accesses deliver the highest bandwidth, with 8 byte and 16 byte accesses contributing a little lower to a noticeably lower bandwidth, respectively. Fulfilling coalescing requirements in a highly divergent A* kernel, remains a programming challenge.

**Execution**

<table>
<thead>
<tr>
<th>Threads per Block</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers per Block</td>
<td>2560</td>
</tr>
<tr>
<td>Warps per Block</td>
<td>4</td>
</tr>
<tr>
<td>Threads per Multiprocessor</td>
<td>384</td>
</tr>
<tr>
<td>Thread Blocks per Multiprocessor</td>
<td>3</td>
</tr>
<tr>
<td>Thread Blocks per GPU</td>
<td>48</td>
</tr>
</tbody>
</table>

NVIDIA’s CUDA Occupancy Calculator tool generated output for the default pathfinding block of 128 threads, running on current generation GPU.

The available global memory is an attribute of the device properties provided by CUDA. The pathfinding software validates the total memory required for the grid of threads and automatically splits the computation into multi launch tasks. Each launch in the sequence is synchronized and partial search results are copied from the device to the host in a predefined offset into the output lists.

**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>
</tbody>
</table>

Above you see a list of parallel pathfinding benchmarks; depicting for each test graph number of nodes and edges, number of agents (threads), and the number of thread blocks (128 threads per block).

In our benchmarks the CPU was a dual core 2.11 GHz AMD Athlon™ 64 X2 4000+ in a system of 2 GBytes of memory. The GPU was an NVIDIA 8800 GT running at shader clock of 1.5 GHz and has attached 512 MBytes of global memory. The 8800 GT we used had 112 shader processors that amount to 14 multiprocessors (a more latent version of the chip sports 16 multiprocessors). The GPU performance was compared to running on the CPU single threaded both an optimized scalar C++ code and an embedded hand-compiler, tuned SIMD intrinsics (SSE) program with potential vector arithmetic acceleration. In addition, we have validated the CPU performance scale running two threads, one on each core of a 2.0 GHz Intel Core Duo T7300 processor in a system of 2 GBytes of memory and a 4 MBytes of L2 cache; the front-side-bus (FSB) speed was 1.12 GHz. The pathfinding software ran in a Windows XP environment and speedup figures shown reflect wall-to-wall running time measured using Windows high performance counters for both processor types.
Results

<table>
<thead>
<tr>
<th>Graph</th>
<th>Roadmap (KBytes)</th>
<th>Working Set (MBytes)</th>
<th>Total global memory (MBytes)</th>
<th>Launches</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>0.576</td>
<td>0.021</td>
<td>0.021</td>
<td>1</td>
</tr>
<tr>
<td>G1</td>
<td>3.616</td>
<td>1.319</td>
<td>1.322</td>
<td>1</td>
</tr>
<tr>
<td>G2</td>
<td>6.368</td>
<td>10.518</td>
<td>10.519</td>
<td>1</td>
</tr>
<tr>
<td>G3</td>
<td>13.848</td>
<td>86.001</td>
<td>86.001</td>
<td>1</td>
</tr>
<tr>
<td>G4</td>
<td>27.672</td>
<td>588.726</td>
<td>588.726</td>
<td>2</td>
</tr>
<tr>
<td>G5</td>
<td>42.560</td>
<td>1573.086</td>
<td>1573.086</td>
<td>3</td>
</tr>
</tbody>
</table>

G4 and G5 global memory capacity surpasses the available GPU memory (512MBytes) and are thereby broken into multiple pathfinding compute launches, each responsible for a subset of the total agents.

Comparative performance of GPU running CUDA Dijkstra search algorithm vs. CPU scalar C++ compiled with optimization.

Performance of two-threaded A* search algorithm using Euclidian heuristic, one thread per CPU core with hand-compiler tuned SIMD intrinsics (SSE), compared against a single threaded run.
Performance of GPU running CUDA A* search algorithm using Euclidian heuristic, compared to CPU plain optimized C++ code and to hand-compiler tuned SIMD intrinsics (SSE) implementation.

Current GPU running time logarithmic scale, normalized to G0, demonstrates a (close to) linear ascend with growing roadmap complexity

Analysis

The A* search kernel exhibits a larger GPU performance scale compared to Dijkstra, mostly attributed to the elevated arithmetic intensity rate of the former.

The A* CPU implementation incorporates SIMD intrinsic (SSE) calls in the heuristic methods and as a result contributed to an average of 2.3X speedup across all benchmarks, compared to the scalar C++ code.

GPU performance speedup for Dijkstra (against scalar C++) and A* (compared to the SSE implementation) searches reached up to 27X and 24X, respectively.

- Total memory copies from device-to-host and host-to-device incurred an overhead comparable to kernel running time for some of the workloads. Roadmap copy to device and device-to-host copy appears a small percentage of overall copy cost (less than 1%).
- Non-coalesced global memory loads and stores by far exceed coalesced accesses. Nonetheless, many of the non-coherent accesses are of 8 or 16 bytes and suffer lesser bandwidth fallout.
- The interleaved kernel exhibited a 1.15X performance edge over the strided accessed working set. The interleaved thread indexing improved coalesced loads and stores substantially, but remained a small share of overall global memory transactions after all, and hence the mild speedup.