Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

GPU Accelerated Pathfinding

Avi Bleiweiss

NVIDIA Corporation

Abstract
In the past few years the graphics programmable processor (GPU) has evolved into an increasingly convincing computational resource for non graphics applications. The GPU is especially well suited to address problem sets expressed as data parallel computation with the same program executed on many data elements concurrently. In pursuing a scalable navigation planning approach for many thousands of agents in crowded game scenes, developers became more attracted to decomposable movement algorithms that lend to explicit parallelism. Pathfinding is one key computational intelligence action in games that is typified by intense search over sparse graph data structures. This paper describes an efficient GPU implementation of parallel global pathfinding using the CUDA programming environment, and demonstrates GPU performance scale advantage in executing an inherently irregular and divergent algorithm.

Categories and Subject Descriptors (according to ACM CCS): [Artificial Intelligence] I.2.8 Problem Solving, Control Methods, and Search – Graph and Tree Search Strategies I.3.1

1. 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. Agents must find their route to a particular goal position avoiding collisions with obstacles or other agents in the environment. The computational complexity of simulating multiple agents in crowded setup becomes intractable and arises from the fact that each moving agent is a dynamic obstacle to other agents [Lav06]. 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. This is analogous to the way individual humans navigate in a shared environment, where each of them makes its own observations and decisions, without explicit communication with others.

Navigation planning techniques for multi agents have been traditionally studied in the domain of robotics and in recent years have been increasingly applied to games. Centralized techniques [Lat91, Lav06] consider the sum of all agents to be a single agent and a solution is searched in a composite space. However, as the number of agents increases problem complexity becomes prohibitively high. Decoupled planners, on the contrary, are more distributed, but do require coordination space that may not always guarantee completeness. Their adaptation to dynamic environments [LH00] had either altered the pre-computed global static roadmap or modified the derived path itself.

A real-time motion planning technique using per-particle energy minimization and adopting a continuum perspective of the environment is discussed in Treuille et al. [TCP06] work. The formulation presented yields a set of dynamic potential and velocity fields over the domain that guide all individual motion simultaneously. This approach unifies global path planning and local collision avoidance into a single optimization framework. The authors found that the global planning assumption produces significantly smoother and more realistic crowd motion. Nonetheless, in avoiding agent based dynamics altogether the ability to succinctly express the computation in parallel becomes fairly constraint.

Several models devise a planning algorithm that is completely decoupled and localized [LG07, VPS*08] The method is a two-level planning technique; the first deals with the global path planning towards the goal, and the second addresses local collision avoidance and navigation. The global path is computed using the roadmap graph that represents static objects in the scene without the presence of agents. A classic search algorithm is then used to compute distances from a goal position to any graph node. Local planning infers velocity obstacles [FS98] from a set of neighboring agents and obstacles. Agents are modeled geometrically as a disc with a position and a radius. This simplified model avoids any orientation considerations in evaluating a collision free route for an agent. 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. The technique described is simple, robust, and easily parallelizable for implementation on multi- and many-core CPU architectures. It runs at interactive rates for environments consisting of several thousands to ten-thousands of agents and its performance scales well with the number of agents.

The feasibility of a parallel hardware based motion planning processor (MPP), running probabilistic roadmap method [KSLO96] for static environments, is studied by Atay and Bayazit [AB06]. The method first constructs the roadmap by determining connectivity along collision free paths. Then, the delimiting endpoints of an agent are being tied to roadmap locations and queried for the most desirable joining path. The MPP was realized as a Field Programmable Gate Array (FPGA) that utilizes configurable parallelism for both construction and query processing. It demonstrated roadmap generation and query speedup of over an order of magnitude compared to a high end CPU.

This explicit parallelism of navigation planning for many thousands of agents in a game lends itself well to data parallel compute paradigm. Essentially, a single program consults global connectivity data and resolves concurrently an agent's optimal path, bound by a start and goal positions. The program produces a pair of outputs: a scalar value for path total cost and a list of positions that constitutes a preferred track of plotted waypoints. Our work exploits a more generic form of nested data parallelism to global pathfinding, and its main contribution is an efficient GPU implementation of a highly irregular and divergent algorithm. Our empirical results demonstrate performance scale advantage of the GPU for medium and large number of agents compared to both an optimized scalar C++ and Single-Instruction Multiple-Data (SIMD) CPU implementations.

The remainder of the paper is structured as follows. Section 2 formulates global pathfinding and briefly discusses graph data structure and search algorithms. Section 3 sets the work objective, highlights the CUDA programming environment and presents the GPU implementation approach. Performance results are demonstrated and analyzed in section 4 and 5, respectively. In conclusion, possible forward looking game impact of GPU accelerated pathfinding is further addressed in section 6.

2. Pathfinding

Pathfinding is one of the more pivotal, low level core intelligence actions in a game [Buc04, Pat07]. Its main objective is to optimally navigate each of the game agents to its goal position avoiding collisions with other agents and obstacles in a constantly changing environment. The task involves a cost based search over a transitional roadmap graph data structure. Nodes in the roadmap represent the position of a key area or an object in the scene and edges attribute cost to connected locations. An agent movement is not restricted to roadmap edges and the navigation graph is consulted to negotiate the environment for forward planning. The next two sections provide mathematical characterization of graphs and search algorithms.

2.1 Graph

A graph (G) is formally defined as a pair of a set of nodes or vertices (N) binary linked with a set of edges (E): $G = \{N, E\}$. The ratio of edges to nodes expresses a graph as being dense (|E| is close to $|V|^2$) or sparse (|E| much less than $|V|^2$). Graphs are either directed or undirected. In a directed graph nodes define an edge are an ordered pair specifying edge direction. Edges in an undirected graph consist of unordered pair of nodes. Undirected graphs are often represented as directed acyclic graphs (DAG) by connecting each linked node pair with two bidirectional edges. Adjacency matrix and a collection of adjacency lists are the two main data structures for representing a graph [CLRS01], depicted in Figure 1:

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	0	0	0
3	0	0	0	0

0	1	2
1	3	

Figure 1: *Graph data structure representations: adjacency matrix (left) requires an $O(N^2)$ footprint, independent of the number of edges in the graph; and adjacency lists (right) consumes $O(N+E)$ memory space for both directed and undirected graphs.*

Adjacency matrix representation is a two dimensional array of Booleans that stores graph topology for a non weighted graph. It is simple, intuitive and more useful for dense graphs. 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. Each node stores a list of immediately connected edges. 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.

2.2 Search

Numerous algorithms have been devised to search and explore the topology of a graph [Nil86, BCF90, Sha92]. It is possible to visit every node of the graph, find any path between two nodes or find the best path between two nodes. Game navigation planning is ultimately concerned with the arriving at an optimal path that may be any of the shortest path between two nodes, the path that takes an agent between two points in the fastest time, or the path to avoid enemy line of sight. Searches are generally classified as being either uninformed or informed. Informed searches reduce overall amount of computations by making intelligent choices in selecting the graph region of interest. The problem set of pathfinding attends to informed searches that consider a cost weighted edge for traversing a graph. The cost alone is sufficient to determine the action sequence that leads to any visited node. Virtually any search algorithm is considered systematic, provided that it marks visited nodes to avoid revisiting the same nodes indefinitely.

Figure 2 depicts a general template of forward search algorithms expressed by using a state space [Lav06] representation. A planning state represents any of position, velocity or orientation of an agent. The definition of a state is an important component in formulating a planning problem and the design of an algorithm to solve it. A planning problem usually involves starting in some initial state and trying to arrive at a specified goal state. Feasibility and optimality are the major concerns of a plan that applies a search action. At any point during the search there is one of three possible node states: unvisited, dead or alive. Visited nodes with all possible next nodes already visited are considered dead. Alive are visited nodes with possible non visited adjacent nodes.

```
 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 ϵ 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
```

Figure 2: *A general template for forward search: alive nodes are stored in a priority queue Q; n_S and n_G are the start and goal positions, respectively; u is an action in a list of actions U for the current node n; and n’ is a next adjacent node derived from the state transition function f(n, u). Node that is reached multiple times in a cost based search requires the resolution step in line 12.*

Classical graph search algorithms include Best First (BFS), Dijkstra and A* (pronounced A Star) [CLRS01, Lav06, RN95]. Each algorithm is a special case of the general template above, obtained by defining a different sorting function to the priority queue Q. Given a path defined by its endpoint positions a cost during a search is evaluated from the current node to either the starting point, to the goal or to both. Incorporating a heuristic estimate [HNR68, Val84, DP85] of the cost to get to the goal from a given node reduces the overall graph space explored. The table in Figure 3 summarizes the properties of the search algorithms of concern:

Search	Start	Goal	Heuristic	Optimal	Speed
BFS	no	yes	yes	no	fair
Dijkstra	yes	no	no	yes	slow
A*	yes	yes	yes	yes°	fast

° assumes admissible heuristic

Figure 3: *Search algorithm comparative properties 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.*

A* is both admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic function. Admissible means optimistic in the sense that the true cost will be at least as great as the estimate. The use of heuristics trades off optimality vs. execution speed and often scales better to solve larger problems. The more notable heuristic methods used in a grid based graph search involve any of Manhattan, diagonal and Euclidian distance calculations. Various tie-breaking techniques are used for search optimization, but they are beyond the scope of this paper. Figure 4 illustrates the pseudo code of the A* algorithm.

```
 1:  f = priority queue element {node index, cost}
 2:  F = priority queue containing initial f (0,0)
 2:  G = g cost set initialized to zero
 3:  P, S = pending and shortest nullified edge sets
 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
```

Figure 4: *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).*

A* is fairly irregular and highly nested. For most part it is memory bound with very little math, largely embedded in the heuristic method. The outer loop of the algorithm commences as long as the priority queue contains live nodes. Queue elements are node index based and are sorted by their cost values in an ascending order. Element’s cost value is the sum of current-to-start and current-to-goal costs. 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. Else, adjacent nodes (or edges) are further evaluated to their cost; unvisited nodes or live nodes with lesser cost to start are pushed onto both a pending list of live nodes (or edges) and onto the priority queue. A successful search returns to the user both a total cost value of the resolved path, and a list of ordered nodes that provides plotted waypoints.

Optimal, non-weighted A* and Dijkstra search algorithms, running over undirected, sparse graphs that are stored in an adjacency lists format, are the concern of the GPU implementation described in the next section.

3. Implementation

The main objective of the work presented in this paper is to exploit general data parallelism in performing global navigation planning for many thousands of game agents. The irregular and deeply nested A* search algorithm imposes acceleration challenges on a device with a relatively large SIMD thread group. Ultimately, the goal is to demonstrate materialized GPU speedup compared to an equivalent single and multi threaded, both scalar and hand coded vector implementations running on the CPU.

NVIDIA's CUDA [NVI07] programming environment was the platform of choice, exposing hardware features that are essential to impact final performance of data parallel computations. The next section provides a brief, high level overview of CUDA's programming model followed by a detailed discussion of parallel global pathfinding realization on the GPU.

3.1 CUDA

CUDA stands for Compute Unified Device Architecture and is a relatively new hardware and software architecture for managing the GPU as a data parallel computing device. The CUDA programming environment inherits from the Brook [BH03, BFH*04] framework developed at Stanford. When programmed through CUDA, the GPU is viewed as a device capable of executing a very high number of threads in parallel. A single program, called a kernel, written in a C extended programming language is compiled to the device instruction set and operates on many data elements concurrently. To this extent the GPU is regarded as a coprocessor to the main CPU and data parallel, compute intensive portions of applications running on the host are offloaded onto the device. The CUDA software stack is largely partitioned into a low level hardware driver and a light weight runtime layer; the work presented here predominately communicates with the runtime application programming interface (API).

CUDA provides general DRAM memory addressing and supports both scatter and gather memory operations. From a programming perspective, this translates into the ability to read and write data at any location in DRAM, much like on a CPU. Both the host and the device maintain their own DRAM, referred to as host memory and device memory, respectively. One can copy data from one DRAM to the other through optimized API calls that utilize the device's high performance Direct Memory Access (DMA) engines.

The batch of threads that executes a kernel is organized as a grid of thread blocks. A thread block is a collection of threads that can cooperate together by efficiently sharing data through fast shared memory. In a block each thread is identified by its thread ID. A block can be specified as any of one, two or three dimensional arbitrary sized array, simplifying software thread indexing. Number of threads per block is finite and hence the provision for a grid extent of thread blocks, all performing the same kernel. The grid further insures efficient distribution of threads on the GPU yet trading off thread cooperation – threads of different blocks have no communication paths. A thread block in a grid has a unique ID and a grid of blocks can again be defined as one, two or three dimensional array.

The device is implemented as a set of multiprocessors, each of a SIMD architecture e.g. at any given clock cycle, a processor of the multiprocessor executes the same instruction, but operates on different data. The CUDA memory model defines a hierarchy ranking from per thread read-write registers and local memory, to per block read-write shared memory, and per grid read-write global memory, and read only constant and texture memory. On chip registers, shared memory, and constant and texture cache access is very fast; off chip local and global memory reads and writes are non-cached and hence much slower. The global, constant, and texture memory spaces can be read from or written to by the host and are persistent across kernel launches by the same application.

The device thread scheduler decomposes a thread block onto smaller SIMD thread groups called warps. Occupancy, the ratio of active warps per multiprocessor to the maximum number of warps allowed, is an execution efficiency criterion largely determined by the register and shared memory resources claimed by the kernel. An execution configuration with integral multiple of warps assigned to a thread block is a first line of order to insure adequate SIMD efficiency. CUDA's Occupancy Calculator tool further assists the programmer in finely matching threads per block to kernel usage of shared resources. The compute capability of the device exploited in the parallel pathfinding work presented in this paper complies with CUDA version 1.1. Asynchronous kernel launches, the use of the newly event API for timing measurements and a somewhat reduced register pressure per thread appeared to have benefited the implementation described next.

3.2 Software

We have implemented a navigation planning software library that has both a CPU and GPU invocation paths. The implementation of the A* search algorithm was made consistent on both processor types in order to deliver as credible as possible comparative performance data. The following sections discuss primarily the CUDA realization of global pathfinding alluding to design tradeoffs pertaining to roadmap textures allocation preferences, working set coalesced access constraints, optimal priority queue insert and extract operations and finally, parallel execution of a highly irregular and divergent kernel with extremely low arithmetic intensity .

3.2.1 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. Device memory reads through texture fetching has the benefit of being cached and potentially exhibit higher bandwidth for localized access. The A* inner loop exemplifies coherence in accessing adjacency list of edges, sequentially. In addition, texture reference bound to linear device memory has the added advantage of a much larger address extent when compared to a CUDA array; it sports a maximum width of 2^{27} (128M) components, well in line of accommodating large footprint graphs. 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. Texture component layout is depicted in Figure 5:

node			
id	position.x	position.y	position.z

edge			
from	to	cost	reserved

adjacency	
offset	offset+count

Figure 5: *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.*

The roadmap graph texture set is a single copy memory resource shared across a grid of threads. The presence of the adjacency directory texture brings the total graph memory footprint in the GPU to 16*(N+E)+8*N bytes. 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.

It is often the case in games for the roadmap to be constantly modified. An incremental change per time step carries little overhead in adding and removing nodes and edges of the currently loaded roadmap in host memory. A more substantial environment transformation could potentially require a new roadmap generation that is subject to a higher cost for loading the graph. Parallel, efficient roadmap generation techniques [AB06] are evolving and promising, but are outside the scope of this paper. The final step for copying the roadmap from host memory to the device's texture space is however a small percentage of the overall search computation workload.

3.2.2 Working Set

The working set for a CUDA launch is commonly referred to as per thread private local and shared memory resources. In the parallel pathfinding workload an agent constitutes a thread on the GPU. The A* kernel has five inputs and two outputs that collectively form the working set. The inputs are each in the form of an array and include:

- 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.

The memory space complexity for both the costs and edge input lists are $O(T*N)$, with T the number of agents participating in the game and N the number of roadmap graph nodes. The pair of outputs produced by the kernel follows:

- 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.

Kernel resources associated with both the inputs and outputs are being allocated in linear global memory regions. 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.

A strided and interleaved, per thread, working set access versions of kernels have been implemented with the goal of identifying the benefit of coalesced global memory access patterns. The strided version working set index for an element in any of the input lists is graph nodes (N) entries apart for each thread in a warp. This type of access is implicitly non-coalesced and for 4 byte reads and writes could experience an order of magnitude slower bandwidth compared to a coalesced access. On the other hand, the interleaved kernel has its element index only data structure size (4, 8 or 16 bytes) apart between consecutive threads in a wrap. Aside from the kernel being highly divergent, the interleaved version has the likelihood to benefit from coalescing global memory transactions. Performance data related to the behavior of strided vs. interleaved access are provided in section 5.

3.2.3 Priority Queue

The priority queue in the search algorithm maintains a set of element pairs composed of a float type cost and an integer node id. Elements with the smallest cost are placed on top of the queue, regardless of the insertion order. The queue is realized as an array of structure pointers in CUDA. Insertion of an element and extracting (and deleting) the element with a minimal cost are the queue operations of concern. The priority queue is the most accessed inside the search inner loop and its operation efficiency is critical to overall performance. The priority queue operates in a fixed size array, set to the number of graph nodes N, and avoids dynamic allocation. Both a naïve and a heap based operations were realized in CUDA giving a performance edge to the heap approach. Figure 6 lists the device heap based insert and extract methods.

```
__device__ void
insert(CUPriorityQ* pq, CUCost c)
{
   int i = ++(pq→size);
   CUCost* costs = pq→costs;
   while(i > 1 && costs[i>>1].cost >  c.cost) {
        costs[i] = costs[i>>1];
        i >>= 1;
    }
    pq→costs[i] = c;
}
```

```
__device__ CUCost
extract(CUPriorityQ* pq)
{
   CUCost cost;
   if(pq→size >= 1) {
      cost = pq→costs[1];
      pq→costs[1] = pq→costs[pq→size--];
      heapify(pq);
   }
   return cost;
}
```

Figure 6: *CUDA device implementation of heap based priority queue insertion (top) and extraction (bottom); both are of complexity O(*log*C) time (C is the number of elements enqueued); heapify runs one round of a heapsort loop after removing the top of the heap and swapping the last element into its place.*

In examining the A* kernel workload it was evident priority queue insertions dominate extractions for the early exit cases, once the search reaches the goal position. Nonetheless, a naïve, linear time cost extraction appeared to have hurt performance. In our implementation the extraction (and deletion) operation received equal efficiency attention and performs a heap sort in CUDA resulting in a logarithmic running cost. Finally, both insertion and extraction operations are in-place and avoid any recursion.

3.2.4 Execution

This section addresses parallel execution considerations of global pathfinding running on the GPU. An agent defines a start and goal end points and constitutes a disjoint thread entity for performing an optimal search, operating over the shared roadmap graph. The total number of participating agents in a game defines the CUDA launch scope.

The navigation software lays out threads in a one dimensional grid of one dimensional thread blocks. The software initially consults the device properties provided by CUDA and sets not to exceed caps for the dimensions of each the grid and the block. The A* kernel has no explicit allocation of device shared memory and any shared memory usage is an implicit assignment by the CUDA compiler. The number of threads allocated per block is therefore largely dependent on the register usage by the kernel. The CUDA occupancy metrics for thread efficiency scales well for regular algorithms with high arithmetic intensity. A memory bound, irregular and divergent A* kernel with little to no arithmetic presence appears to be peaking at a 0.5 occupancy rate, yielding the parameters listed in Figure 7 (with 20 registers and 40 bytes of shared memory used per thread):

Threads per Block	128
Registers per Block	2560
Warps per Block	4
Threads per Multiprocessor	384
Thread Blocks per Multiprocessor	3
Thread Blocks per GPU	48

Figure 7: NVIDIA's *CUDA Occupancy Calculator tool generated output for the default pathfinding block of 128 threads, running on current generation GPU.*

The total roadmap graph texture and the working set memory space allocated for the entire game agents are liable to exceed the global memory available on a given 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. Per launch allocation is always guaranteed to be in bounds providing a graceful path for running parallel pathfinding on lower end GPU platforms.

Finally, the last thread block in a grid is likely to be only partially occupied with active threads. The A* kernel compares the current thread id against the total number of threads (agents), provided to the kernel as an input argument, and bails out before committing to any search computation. This is not much of a performance gain and is more to account towards not overflowing the total working set space allocated for active threads.

4. Performance

We instrumented the performance of parallel global pathfinding on the GPU by benchmarking half a dozen test roadmaps with varying number of agents from several tens to many thousands. The test graphs were generated using the Raven game graph editor [Buc04] and they largely represent small to moderate topology complexity for the roadmaps. We ran both Dijkstra and A* search kernels using the more efficient interleaved version for working set access. At the time our software was compatible with CUDA 1.1 compute type devices. The following sections discuss benchmark parameters and experiment methodology, and demonstrate memory footprint distribution, speedup and running time results.

4.1 Benchmarks

Figure 8 illustrates the topological characteristics of each of the test roadmaps used in our benchmark. The roadmap graphs are undirected and the number of agents used for each benchmark is the graph nodes squared (N^2), each exercising any possible pair of start and goal endpoint positions. The table also highlights the number of thread blocks deployed for each of the benchmarks.

Graph	Nodes	Edges	Agents	Blocks
G0	8	24	64	1
G1	32	178	1024	8
G2	64	302	4096	32
G3	129	672	16641	131
G4	245	1362	60025	469
G5	340	2150	115600	904

Figure 8: *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.

4.2 Results

In this section we present our experimental results for running the benchmarks listed above. Figure 9 shows consumed GPU global memory footprint figures for each of the benchmarks, broken into roadmap textures (KBytes), working set (MBytes) and the total global memory (MBytes). Expectedly, the working set memory space allocated by far exceeds the roadmap set share. 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.

Graph	Roadmap	Working Set	Total	Launches
G0	0.576	0.021	0.021	1
G1	3.616	1.319	1.322	1
G2	6.368	10.518	10.519	1
G3	13.848	86.001	86.001	1
G4	27.672	588.726	588.726	2
G5	42.560	1573.086	1573.086	3

Figure 9: *Benchmark's GPU global memory footprint for each the roadmap (KBytes), working set (MBytes) and total (MBytes). Multiple launches are the result of exceeding available GPU global memory.*

The chart of Figure 10 compares GPU's performance vs. scalar C++ (compiled with O2 level optimization on Microsoft's Visual C++ 2005 compiler) for the benchmarks running the Dijkstra search:

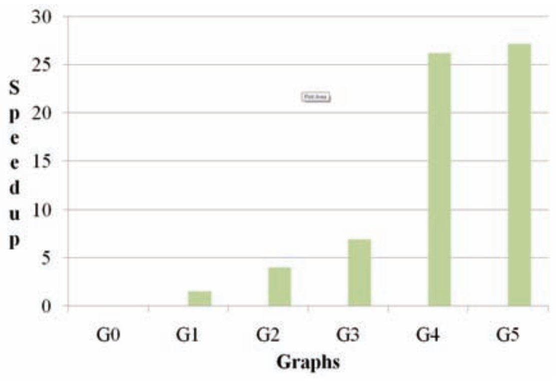

Figure 10: *Comparative performance of GPU running CUDA Dijkstra search algorithm vs. CPU scalar C++ compiled with optimization.*

Figures 11 and 12 provide performance statistics for running the A* search algorithm using a Euclidian heuristic. Figure 11 demonstrates CPU performance scale for a cost comparable, dual core CPU in running two threads, one on each core executing code with SIMD intrinsic (SSE) calls. Figure 12 shows GPU CUDA performance compared to both plain, optimized C++ code and hand-compiler tuned SIMD intrinsics (SSE) program.

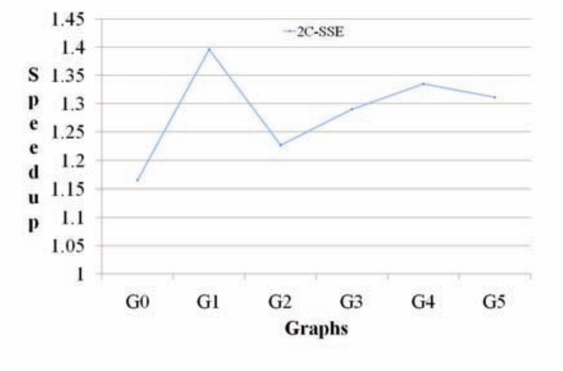

Figure 11: *Performance of two-threaded A* search algorithm using Euclidian heuristic, one thread per CPU core with hand-compiler tuned SIMD intrinsics (SSE), compared against a single threaded run.*

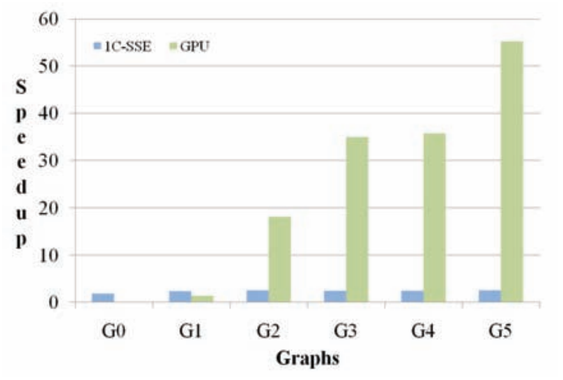

Figure 12: *Performance of GPU running CUDA A* search algorithm using Euclidian heuristic, compared to CPU plain optimized C++ code and to hand-compiler tuned SIMD intrinsics (SSE) implementation.*

The absolute running time for the benchmarks executing on the GPU ranged from 30 milliseconds for G0 up to 2.5 seconds for G5 performing for the latter an average search time of 21 microseconds and 12.6576 average points per resolved agent path. The start and goal positions across agents of a thread block were intentionally spatially non coherent in the roadmap and hence expose high degree of execution divergence within a multiprocessor. The running time logarithmic scale as a function of the benchmark's topology complexity, normalized to G0 benchmark, is shown in Figure 13:

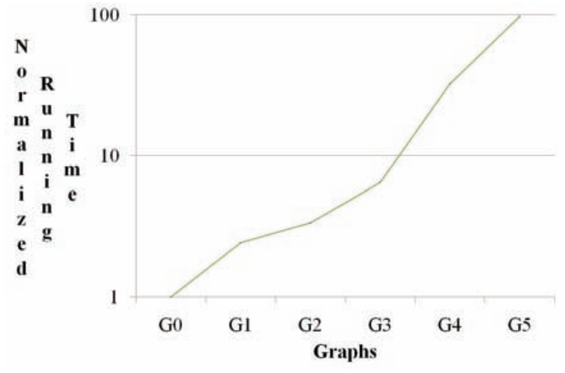

Figure 13: *Current GPU running time logarithmic scale, normalized to G0, demonstrates a (close to) linear ascend with growing roadmap complexity*

5. Analysis

The pathfinding software ran the six benchmarks introduced above. G0 and G1 workloads are of relatively low agent count and GPU performance scale is either none or insignificant. Speedup is substantially more noticeable for tens to hundreds of thousands of agents, as evidenced in G3 to G5 scenarios, with active number of thread blocks exceeding one hundred and thereby sustaining a higher rate of GPU thread efficiency. Overall, the A* search kernel exhibits a larger GPU performance scale compared to Dijkstra, mostly attributed to the elevated arithmetic intensity rate of the former. A* math is embedded in the Euclidian heuristic function and is invoked in the inner loop of the search at a frequency that balances out part of global memory access cost. The math is composed of a vector subtract and a dot product followed by a scalar square root. 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. In addition to vector math acceleration the implementation leverages an efficient SSE square root instruction in contrast to a slow C runtime function. GPU performance speedup for Dijkstra (against scalar C++) and A* (compared to the SSE implementation) searches reached up to 27X and 24X, respectively.

In the course of running the benchmarks we have collected performance statistics emitted by the CUDA Profiler tool that helped understand our current limitations of both the navigation software implementation and the hardware. The tool assisted us in identifying performance bottlenecks and quantifying the benefit of kernel optimizations. The profiler queries device performance counters state inline with the execution of the code in a non intrusive manner. The performance counter values do not correspond to an individual thread and rather represent events within a thread warp. The profiler only targets a single multiprocessor on the GPU and we found it highly useful to analyze relative performance of the strided vs. the interleaved kernel versions. The following list accounts for some of the more important profiler data to affect overall efficiency:

- 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.

The GPU absolute running time for the benchmarks appears to be consistent with the increased topological complexity of the roadmap graph. Running time is somewhat sub linear with the number of thread blocks being less than one hundred and (close to) linear for a higher block count when sustaining an increased overall GPU thread utilization. We were encouraged to find CUDA launch and synchronization toll in the multi-launch benchmarks to be non critical to overall performance.

The pathfinding software supports multi core parallelism with a core semantic orthogonal to any of the CPU and GPU processor types. In a multi core run computation load is partitioned by assigning evenly a subset of the agents to each core. Multi core GPU system assumes the roadmap texture set replicated across devices. In our experiments we have observed an average speedup of 1.3X across benchmarks in comparing two CPU threads, one per core, to a single thread each running the A* search with SSE vector optimization (illustrated in Figure 11). Relative to the two threaded, dual core CPU running A* the GPU performance scale is up to 18X.

6. Conclusion and Future Work

This paper demonstrated an efficient implementation of global pathfinding on the GPU, challenging the irregularity and a highly divergent core algorithm. The work presented exploits nested data parallelism on the GPU and proved its performance scale to be over an order of magnitude compared to a single and two threaded, optimized plain C++ and SSE accelerated CPU implementations for the classic Dijkstra and A* search algorithms, respectively. The method presented adapts seamlessly to a multi GPU core system, anticipating close to a linear performance scale. CUDA programming environment has played an important role in achieving this speedup level by providing direct access to conventionally invisible GPU hardware resources. Many thousands of agents participating in a game is a near term reality and this work reaffirms the GPU as the preferred platform for off loading game computational intelligence workloads. The scaling of navigation planning in crowded scenes holds the prospect of elevating interactive game play credibility.

We look forward to GPU devices that support efficient double precision math computation and result in a consistent behavior for floating point accumulation of search cost values. The sequential fetch of an adjacency list proved the binding of the roadmap graph to texture memory favorable, resulting in cache locality when mattered most, and is likely to scale well for larger roadmaps. We feel our working set allocation is somewhat greedy and could potentially be relaxed with dynamic allocation support in CUDA. Caching global memory reads, especially for priority queue insertion and extraction, is expected to enhance our efficiency further. We also anticipate the improved auto coalescing of global memory accesses by the device to yield a higher effective memory bandwidth for four bytes scatter. Using host type CUDA allocation and leveraging the GPU faster DMA transfers is likely to reduce the overall copy overhead incurred in the current implementation. The option of spawning threads within a kernel with a relatively low fork and join expense will allow the A* inner loop to be completely unrolled and parallelized [CN90], potentially reducing iteration cost.

Finally, we would like to evolve our software into local navigation planning and account for inter-agent constraints and dynamic obstacles [VPS*08]. We think we might improve our overall performance and benefit from searching abstraction hierarchy as demonstrated in [HPZM96, Stu07]. We also want to understand the performance tradeoff for a much larger roadmap graph (N > 10000), and the influence of agents with coherent start and goal endpoint positions, coalesced as much as possible into distinct thread blocks, to possibly lessen thread divergence.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive and helpful comments.

References

[AB06] ATAY N., BAYAZIT B.: A Motion Planning Processor on Reconfigurable Hardware. *Proceedings of the International Conference on Robotics and Automation*, (May 2006), 125–132.

[BCF90] BARR A., COHEN P. R., FEIGENBAUM E. A.: Search. *The Handbook of Artificial Intelligence*. Addison-Wesley, 1990.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions of Graphics*, (Aug. 2004), 777–786.

[BH03] BUCK I., HANRAHAN P.: Data Parallel Computation on Graphics Hardware. *Tech. Report 2003-03, Stanford University Computer Science Department*, (Dec. 2003).

[Buc04] BUCKLAND M.: Programming Game AI by Example. Wordware Publishing, Inc, 2004.

[CLRS01] CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C.: Introduction to Algorithms. MIT Press, Cambridge, MA, 2001.

[CN90] CVETANOVIC Z., NOFSINGER C.: Parallel AStar Search on Message-Passing Architectures. *System Sciences, Proceedings of the Twenty-Third Annual Hawaii Conference*, 1 (1990), 82–90.

[DP85] DECHTER R., PEARL J.: Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, 32 (Jul. 1985), 505–536.

[FS98] FIORINI P., SHILLER Z.: Motion Planning in Dynamic Environments using Velocity Obstacles. *International Journal of Robotics Research* (1998), 760–772.

[HNR68] HART, P. E., NILSSON, N. J., RAPHAEL, B.: (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on System Science and Cybernetics*, 4 (Jul. 1968), 100–107.

[HPZM96] HOLTE R. C., PEREZ M. B., ZIMMER R. M., MACDONALD A. J.: Hierarchical A*: Searching Abstraction Hierarchies Efficiently. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, (Aug. 1996), 530–536.

[KSLO96] KAVARAKI L., SVESTKA P., LATOMBE J. C., OVERMARS M.: Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," *Proceedings of the International Conference of. Robotics and Automation*, (Aug. 1996), 566–580.

[Lat91] LATOMBE, J.: Robot Motion Planning. Kluwer Academic Publishers, 1991.

[Lav06] LAVALLE S. M.: Planning Algorithms. Cambridge University Press. http://msl.cs.uiuc.edu/planning/, (2006).

[LG07] LI, Y., GUPTA, K.: Motion Planning of Multiple Agents in Virtual Environments on Parallel Architectures. *IEEE International Conference on Robotics and Automation*, (Apr. 2007), 1009–1014.

[LH00] LEVEN P., HUTCHINSON S.: Toward Real-Time Path Planning in Changing Environments. *Proceedings of the Fourth International Workshop on the Algorithmic Foundations of Robotics*, (Mar. 2000), 363–376.

[Nil86] NILLSSON N. J.: Search Strategies for AI Production Systems. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, 1986, 53–96.

[NVI07] NVIDIA CORPORATION: CUDA Compute Unified Device Architecture Programming Guide. http://developer.nvidia.com/cuda, (Jan. 2007).

[Pat07] PATEL A.: Amit's A* Pages. http://theory.stanford.edu/~amitp/GameProgramming/, 2007.

[RN95] RUSSELL S. J., NORVIG P.: *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995, 97–104.

[Sha92] SHAPIRO S. C.: Artificial Intelligence. *Encyclopedia of Artificial Intelligence.* John Wiley & Sons, 1992, 54–57.

[Stu07] STURTEVANT N. R.: Memory-Efficient Abstractions for Pathfinding. *Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference*, (Jun. 2007), 31–36.

[TCP06] TREUILLE A., COOPER S., POPOVIC Z.: Continuum Crowds. *In* ACM Transactions on Graphics, (Jul. 2006), 1160–1168

[Val84] VALTORTA M.: A Result on the Computational Complexity of Heuristic Estimates for the A* Algorithm. *Information Sciences*, 34 (Jan. 1983), 48–59.

[VPS*08] VAN DEN BERG J., PATIL S., SEWALL J., MANOCHA D., LIN M.: Interactive Navigation of Multiple Agents in Crowded Environments. *Symposium on Interactive 3D Graphics and Games*, (Feb. 2008), 139–147.