Parallel Hierarchical A* for Multi Agent-Based Simulation on the GPU

Giuseppe Caggianese and Ugo Erra

Università della Basilicata
Dipartimento di Matematica, Informatica ed Economia
{giuseppe.caggianese,ugo.erra}@unibas.it

Abstract. In this work, we describe a Parallel Hierarchical A* (PHA*) for path-finding in real-time using the Graphics Processor Units (GPUs). The technique aims to find potential paths for many hundreds of agents by building an abstraction graph of the input map in an off-line phase and then using this representation to speed up the path-finding during the on-line phase. The approach is appropriate in the case of scenarios based on grid maps and is independent on a specific obstacle configurations. In addition, we propose also a strategy to obtain smooth paths during the search. We show that this approach fits well with the programming model of the GPUs, enabling searching for many hundreds of agents in parallel in real-time applications such as simulations. The paper describes this implementation using the Compute Unified Device Architecture programming environment, and demonstrates its advantages in terms of performance and quality of the paths founded comparing PHA* with a GPUs implementation of Real-Time Adaptive A* and the classic A* algorithm.

Keywords: path-finding, GPUs acceleration, simulation, real-time search.

1 Introduction

Many types of simulations involve agents moving over maps. In particular, in some simulations the agents have different characteristics and behavior and a path planning algorithm is a key task for maintaining as an essential aspect of plausible agent behavior such as collision avoidance and goal satisfaction. Path planning also consumes a great part of the computation time during the simulations, particularly in highly dynamic environments where most of the agents are moving at the same time with varying goals. A fast path planning algorithm is therefore essential for the agents to move smoothly around obstacles and eventually with other moving agents.

Single-agent path planning, where the size of the search space is bounded by the map size, can be tackled with a search algorithm such as A*[5]. Multi-agent path planning is much more challenging than the single agent case in term of CPU resources and memory requirements. Hence, despite its completeness and

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 513–522, 2014.

solution optimality guarantees, a centralized A* search has little practical value in a multi-agent path planning problem. As the number of agents and the size of the search space increase, searching tends to become computationally burdensome or intractable even for relatively small maps and collections of agents. Trading the method completeness and the solution optimality for improved performance is a typical feature of decentralized approaches, including the method described in this paper.

In the last few years, Graphics Processor Units (GPUs) have increased rapidly in popularity because they offer an opportunity to accelerate many algorithms. In particular, applications that have large numbers of parallel threads that do similar work across many data points with limited synchronization are good candidates with which to exploit GPU acceleration. All of this means that in a multi-agent scenario, we can take into account the idea that exploration of different paths can be performed in parallel with large numbers of threads that explore simultaneous paths. In this way, scalability to larger maps can be achieved with decentralized approaches, which decompose the initial problem into a series of parallel searches. However, a GPU-based path planning must takes into account that it has limited access to the GPU and memory resources, which are allocated with higher priority to other modules such as the graphics engine and more recently to the physics engine.

In this paper, we describe Parallel Hierarchical A*, a path planning system for agents that exploits the GPU to achieve real-time performance. The proposed approach is appropriate in the simulation based on grid maps where a grid cell, called tile, is the smallest unit space that can be occupied by a single agent. Before the beginning of the simulation, the input map is decomposed in blocks. Because these blocks are not guaranteed to be contiguous, they are further divided into regions and each region within a block is represented by single tile. Once the tiles have been established we compute edges between regions. At the end of this process, we obtain an abstract graph which can be used at runtime to find in parallel the potential high-level paths for many hundreds of agents. The entire process from the building of the abstract graph to the planning path of the agents are implemented on the GPU. We show that our Parallel Hierarchical A* fits well with the parallel architecture of the GPU where many threads are necessary to exploit the GPU fully. The method is simple and easy to implement using the programming model of the GPU as for instance the platform chosen in this work NVIDIA's Compute Unified Device Architecture (CUDA). The empirical results demonstrate the performance scale advantage of the GPU for a large number of agents compared to GPU implementations of RTAA* and sub-optimal solutions compared to a CPU implementation of A*, then trading optimality for improved execution performance.

2 Related Works

The first part of this section summarizes hierarchical approaches used for path-finding which are more relevant for this work. The second part reviews related work on path-finding using the GPU in a more general context.

Path-finding approaches based on topological abstraction have been explored in several domains such as computer games and robotics. Hierarchical Path-Finding A* is a hierarchical approach for reducing problem complexity on grid based maps. The works [6][11] are two approaches based on hierarchical representations of a space with the goal of reducing the overall search effort. In [2], Botea et al. presents and Hierarchical Path-Finding A* for reducing problem complexity in path-finding on grid-based maps. The technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing the cluster are pre-computed and cached. At the global level, an action is to cross a cluster in a single step rather than moving to an adjacent atomic location. In [13], Sturtevant presents an extension of the Botea et al. approach to build automated and minimal-memory state abstractions to speed path-finding. Experimental results show that with 3% more memory, path-finding is up to 100 times faster than A*. We extend significantly this work to massive path-finding scenario and also adapting it to the programming model of the GPU.

Researchers have recently developed parallel-based implementations of path-finding that use the computational power of the GPU. These approaches show that the GPU enables efficient path-finding to support large numbers of agents moving through expansive and increasingly large dynamic environments. In 2008, Bleiweiss [1] implemented the Dijkstra and the A* algorithms using CUDA. In [7], Katz et al. present a cache-efficient GPU implementation of the all-pairs shortest-path problem and demonstrate that it results in a significant improvement in performance. In [12], Stefan et al. obtained speedups for breadth-first search using a bit-vector representation of the search frontier on a GPU. In [8], Kider et al. present a novel implementation of a randomized heuristic search, namely R* search, that scales to higher-dimensional planning problems. They demonstrate how R* can be implemented on a GPU and show that it consistently produces lower-cost solutions, scales better in terms of memory, and runs faster than R* on a Central Processing Unit (CPU). In [3], Erra et al. propose an efficient multi-agent planning approach for GPUs based on an algorithm called RTAA*. The implementation of RTAA* enables the planning of many thousands of agents by using a limited memory footprint per agent.

3 The GPU Programming Model

In the last years, the increasing performance of the GPUs has led researchers to explore mapping general non-graphics computation onto these new parallel architectures. The GPGPU phenomenon has shown some impressive results and the demand to use the GPU as a more general parallel processor motivated NVIDIA to release in 2006 a new generation of graphics cards that significantly extended the GPU beyond graphics through a new unified graphics and computing GPU architecture and the CUDA programming model [9].

From the point of view of hardware model, modern NVIDIA GPUs are fully programmable many-core chips called CUDA cores. In a GPU, the number of streaming multiprocessors (SMXs) ranges from 4 SMX at the low end to 15

SMXs at the high end. For example, the GPU used in our experiments is an economic device of new Kepler generation and it consists of 4 SMXs, each one containing 192 CUDA cores. Each SMX is capable of supporting up to 2,048 threads. Current NVDIA GPUs manage up to 16,384 threads in real-time. All thread management, including creation, scheduling and barrier synchronization is performed entirely in hardware by the SMX with essentially zero overhead.

From the point of view of software model, CUDA is a minimal extension to the C language which permits the writing of a serial program called kernel. A kernel executes in parallel across a set of parallel threads. Following the representation in Figure 1, each thread has a private local memory. The programmer organizes these threads into a hierarchy of thread blocks and grids. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and have access to the shared memory with latency comparable to registers. The grid is a set of thread blocks that may each be executed independently. All threads have access to the same global, constant or texture memory. These three memory spaces are optimized for different memory usages and thus have different time access. For example, the read-only constant cache and texture cache are shared by all scalar processor cores and this speeds up reads from the texture memory space and constant memory space.

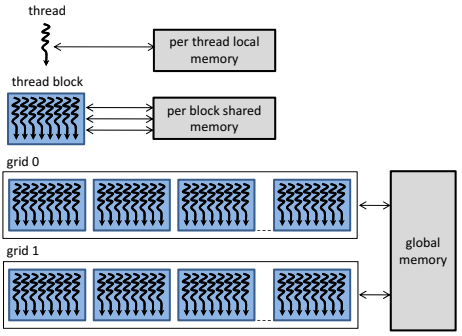

Fig. 1. Levels of parallel granularity and memory sharing on the GPU [10]

4 The Proposed Approach

Our Parallel Hierarchical A* (PHA*) is appropriate for simulation that are based on a grid map. In these simulations, the environment is subdivided into tiles that are the smallest unit space that can be occupied by an agent. Each tile represents a state of the search space and is connected to all nearby tiles. The cost of moving from a tile to each of its neighbors is specified by an integer. This can be used to model terrain elements that are difficult or impossible to pass, for example hills and lakes. A common metric used to measure distance on grid maps, which we adopt for this work, is the Manhattan distance.

In the subsequent sections, we first describe how to build an efficient abstraction graph of the input map by using an off-line phase, and then demonstrate how use this graph during a real-time phase for multi agent path-finding.

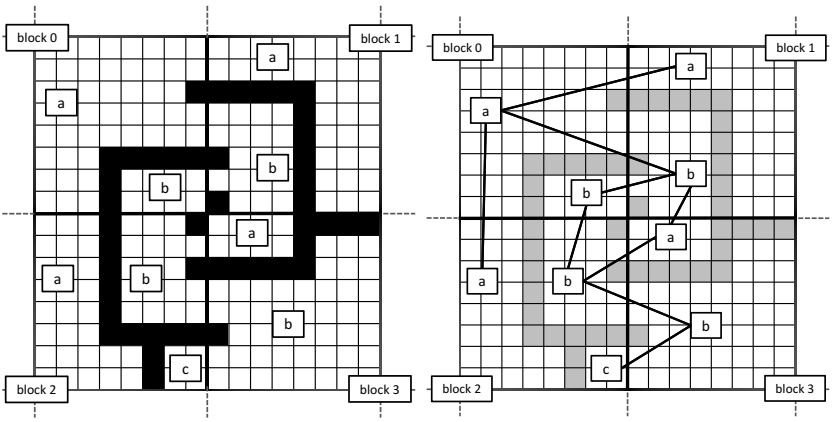

Fig. 2. A subdivision of a grid map in blocks. On the left, the regions identified using a BFS. On the right, the abstract graph.

4.1 Abstract Graph

In this phase, we build a high-level map abstracting away all the details within the tiles of the grid map. The grid map is divided into regular regions called blocks. These blocks are all of the same size and decompose the grid map into non-overlapping search sub-spaces. Because blocks are not guaranteed to be contiguous, we divide the bocks further into connected component called regions. The regions are identified by performing a breadth-first search (BFS) within a block using the GPU as described in Harish et al. [4]. Because there could be more than one region, the BFS is progressively invoked, with different start position, on the same block until all the tiles of the block will be associated with a region. Note that BFS is invoked in parallel over all blocks of the grid map. We illustrate blocks and regions on the left of Figure 2. As for instance, the Block 0 has two regions named 0.a and 0.b.

Each region within a block is represented by a single tile taken at the weighted center of the region. Once these regions have been established, we must compute edges between regions using the following process. For each side of the block, we have a thread that iterates along all the border tiles. The thread compares each border tile of a region with the adjacent border tiles belonging to the nearest region. This process enables to find all connected regions along the borders and hence whenever we find a pair of these regions, we add an edge to the abstract graph. As previously, this step is also performed in parallel and in particular for each block we have four threads one for each cardinal direction. The result of this process is illustrated on the right of Figure 2. As for instance, the region 0.a has three edges to 1.a, 1.b, and 2.a.

In order to use a data structure that fits well to the programming model of the GPU, we represent the graph in compact adjacency list form as illustrated in Figure 3. The array R contains the representative tile of each region. The

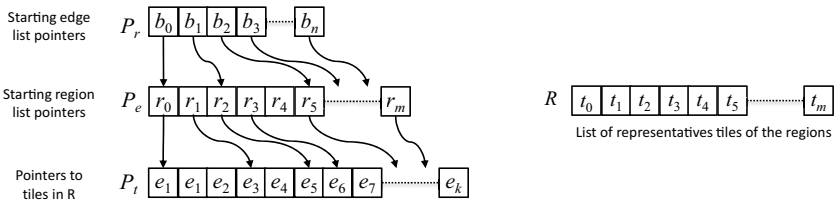

Fig. 3. Graph representation with vertex list pointing to a region list that pointing to an edge list

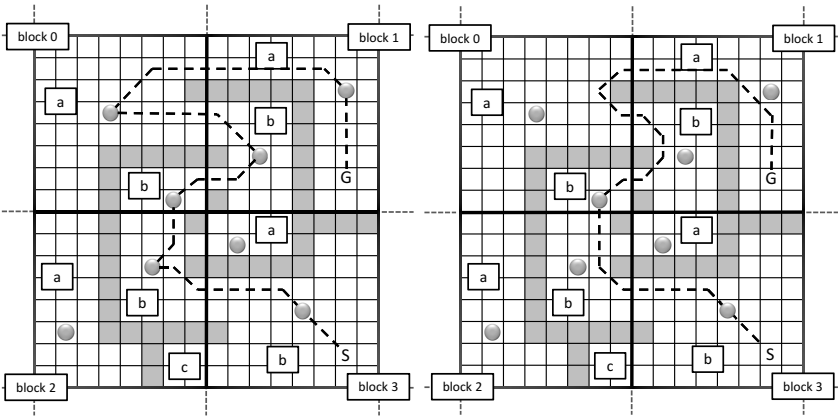

Fig. 4. A path from a start (S) to the goal (G) position. On the left, the path follows the representative tile of each region. On the right, the path enters within the regions and it heads toward the next region.

blocks of the grid map are represented as array P_r. Each entry $P_r[i]$ points to the regions of the block i. These regions are stored as an adjacency list in a large array named P_e from the starting position b_i. By using the index position j of a region in the array P_e we are able to obtain its own representative tile from $R[j]$. The array P_e contains the edges of the graph. Each entry $P_e[k]$ points to the edges of the tile k. These edges are stored as an adjacency list in an array named P_t from the starting positions r_k. Finally, the array P_t contains pointers to the representative tiles in the array R. This graph representation enables to obtain from a given block all the regions and its adjacent regions but also to trace all the adjacency regions from any given region. This feature is an essential requirement for computing paths with Dijkstra in the next phase.

4.2 Path-Finding

Given a set of agents with corresponding start tiles and goal tiles the next step of the path-finding is to find a path in the grid map using the abstract graph. We compute the paths in two steps. In the first step, we use a GPU implementation of Dijkstra[4] to find a path on the abstract graph. In particular, for each agent we

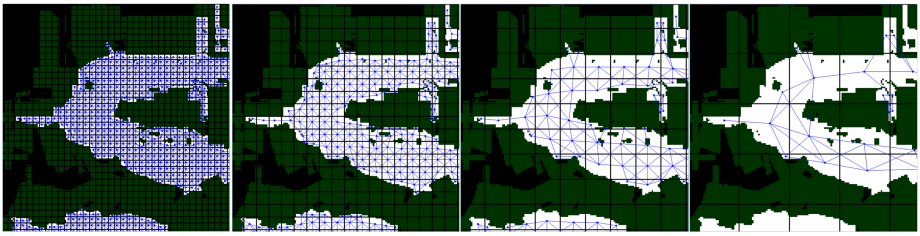

Fig. 5. Detail of the abstract graph for the map den520d

have a thread that computes a complete abstract path. Once a path is found, the second step is to refine this path crossing all the regional tiles using A*[5]. Also in this case, we exploit the GPU for refining the path of each agent allocating a thread per-agent. In both steps, we use the Manhattan distance as heuristic to measure distances on the grid map.

Given a complete abstract path, there are many approach to use this information for computing paths. A simple way to use the abstract path is to follow the representative tile of each region. First search from the starting tile to the first representative tile. Then, search between each region along the abstract path crossing all the representative tile, and finally search between the final region to the goal tile. The main drawback of this approach is that the path computed is much further long as illustrated on the left of Figure 4.

Our solution is to obtain a smoothed path during the searching. The idea is to follow the representative tile of each region but once an agent enters within a region, we force the agent to search immediately for the representative tile of the next region as illustrated on the right of Figure 4. Initially, for each start position we calculate the region where each agent begins to plan the path. Subsequently, the complete path is computed using iteratively search episodes. In each episode, the search is restricted to a small part of the grid map that can be reached from the current tile up to the next region using A*. In particular, each A* search has a representative tile as final position but the stop condition is to reach a new region. Once the agent has stopped within a region, it moves along the resulting trajectory and repeats the search episode process until it reaches the goal tile. This approach is suitable in an environment which tightly limit computational/temporal costs and thus the agents must alternate searching/planning and navigation as for instance in visual simulation.

5 Evaluation and Results

In this section, we show the results of two types of experiments. The first experiment demonstrates the efficiency of our approach. We compare PHA* with RTAA* which are both used to perform searches for many thousands of agents. We chose to compare PHA* with a GPU parallel version of RTAA* because in previous work[3] this implementation was found to be faster than a parallel

Map den520d 256x256				
Agents	Block Size	Execution time PHA* in ms.		Execution time RTAA* in ms
		Off-line Phase	On-line Phase	
512	4	47.72	132.88	4400.75
	8	212.42	86.97	
	16	1269.38	103.91	
	32	9923.20	437.95	
1024	4	42.99	196.57	4699.8
	8	206.63	101.98	
	16	1260.80	121.09	
	32	9750.94	467.12	
2048	4	40.30	321.93	5384.99
	8	192.47	142.96	
	16	1215.06	155.81	
	32	9829.66	550.84	
4096	4	42.54	589.97	6057.33
	8	199.33	225.22	
	16	1212.22	229.37	
	32	9763.42	680.88	

Map tranquilpaths 512x512				
Agents	Block Size	Execution time PHA* in ms		Execution time RTAA* in ms
		Off-line Phase	On-line Phase	
512	4	78.15	338.60	6941.13
	8	387.48	168.86	
	16	2489.22	230.54	
	32	20418.30	563.05	
1024	4	68.17	548.14	8057.09
	8	381.87	234.81	
	16	2554.67	283.02	
	32	20508.40	696.79	
2048	4	67.18	980.09	9256.58
	8	389.24	338.69	
	16	2590.20	349.77	
	32	19638.40	800.46	
4096	4	72.42	1866.50	10896.43
	8	419.40	576.60	
	16	2599.50	490.15	
	32	19709.00	1058.40	

Fig. 6. GPU total times: PHA* vs RTAA*

Average Length Path and Percentage Increase									
Map	A*	PHA*							
		block size 4x4		block size 8x8		block size 16x16		block size 32x32	
	length	length	inc. in %	length	inc. in %	length	inc. in %	length	inc. in %
den520d	128.50	164.46	28.23	149.08	15.92	142.56	11.19	137.09	6.67
tranquilpaths	195.75	249.59	27.59	227.13	16.12	216.98	10.98	215.19	10.69

Fig. 7. Quality of the trajectories: PHA* vs A*

CPU implementation of A*. The second experiment is related to the quality of the trajectory found by using the abstract graph and concerns the length of the path obtained from PHA* compared to the A*[5] which is used as optimal path-finding algorithm.

The experiments were performed on two grid maps named den520d and tranquilpaths of sizes 256×256 and 512×512 respectively, coming from the public repository of Moving AI Lab[1]. An example map with its abstract graph is shown in Figure 5. Start and goal tiles were randomly chosen, and `lookahead`=3 for RTAA*, which is the optimal value for achieving best performance on these two maps. For each grid map, we launched several groups of agents ranging in size from 512 to 4096 with blocks measuring 4×4, 8×8, 16×16, and 32×32. Our tests were performed on an Intel Core i7-2600 3.40 GHz, NVIDIA Kepler GTX 650 Ti 1GB, Windows 7. All the kernels were written using CUDA 5.0 and Microsoft's Visual C++ 2010 compiler.

Figure 6 reports GPU total times of the PHA* compared to RTTA*. For the PHA*, we report both times for off-line and on-line phase. Note that off-line times are approximately the same because depends only to the size of the block. The results indicate that our approach is faster than RTAA*. The choice of the block size strongly influences the required time for the off-line phase. In fact

[1] `http://www.movingai.com/benchmarks/`

in this step, the greater effort is required to identify the representative tile of each region (node of the graph) and their connections (edges of the graph). Use smallest blocks mean small regions to explore, most of which corresponding to a single block, contrary biggest blocks lead to big irregular regions requiring more computation time. The same goes for the on-line phase in which the block size determines the behavior of the searches. In this case smallest block corresponds to a higher number of searches between consecutive regions in the abstract path but each region will be quickly explorable. The opposite happens using bigger block because there will be less regions to cross in the abstracted path but more tile to explore in each region. This aspect suggests the versatility of this approach, in the case of empty maps or with scattered obstacles should be useful use biggest blocks while that became absolutely discouraged for map like those used in the experiments. However, in this case our approach shows better results when the block size is 8×8 because this represents the right balance between region dimension and number of sub-paths in the abstract path.

Figure 7 lists the results of the quality test that compare PHA* and A*. This test was executed with 100 searches on both maps using different block size. The results indicate that the length of the paths retrieved with PHA* are worse than the optimal solution. This deficiency caused by the fact that the final path is composed by using the refining of different sub-path but is compensated for in terms of efficiency as shown in performance experiments. However, as illustrate in Figure 7 increasing the size of the blocks involves a path length near the optimal solution because the final path is less affected by the presence of intermediate representative tile and the A* search used for the on-line phase mitigates the result. This different behavior still plays in favor of user that tuning the block size parameter should reach the desired trade-off between speed and better path. For example, in a real-time application, speed is the highest priority and suboptimal paths may be acceptable.

6 Conclusions

In this work, we have demonstrated a parallel implementation of path-finding for agent-based simulation. Through a hierarchical representation of the input map, PHA* offers a simple and powerful way of planning trajectories for many hundreds of agents using a thread to explore each potential path. We propose also a strategy to obtain smooth paths during the search which enables to improve path lengths. Our results show that the GPU implementation enables the real-time use of this technique even in scenarios with a vast number of agents which is common in applications such as simulation. The execution times showed that both phases could be used also in real-time as for instance in scenario with dynamic obstacles that occur in the grid map.

This aspect offers a starting point for interesting future works. Once the system recognizes the presence of dynamic obstacles, our approach should be used to rebuild the entire graph or only the portion of the abstract graph which corresponds to the blocks where changes occurred. Thus, all retrieved paths combining

multiple sub-paths should be able to adapt swiftly to changes in the map. For this purpose, we are interested to further enhance the performance of the off-line phase. Finally, we want to improve the quality of the path by exploiting in a better way the knowledge of the map acquired during the off-line phase.

References

1. Bleiweiss, A.: GPU accelerated pathfinding. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH 2008, Aire-la-Ville, Switzerland, pp. 65–74. Eurographics Association (2008)
2. Botea, A., Müller, M., Schaeffer, J.: Near Optimal Hierarchical Path-Finding. Journal of Game Development 1(1), 7–28 (2004)
3. Erra, U., Caggianese, G.: Real-time Adaptive GPU multi-agent path planning, vol. 2, ch. 22, pp. 295–308. Morgan Kaufmann Publishers Inc. Computing Gems Jade Edition edition (2011)
4. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
5. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (1968)
6. Holte, R.C., Perez, M.B., Zimmer, R.M., MacDonald, A.J.: Hierarchical A*: searching abstraction hierarchies efficiently. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI 1996, vol. 1, pp. 530–535. AAAI Press (1996)
7. Katz, G.J., Kider Jr., J.T.: All-pairs shortest-paths for large graphs on the GPU. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH 2008, Aire-la-Ville, Switzerland, pp. 47–55. Eurographics Association (2008)
8. Kider, J., Henderson, M., Likhachev, M., Safonova, A.: High-dimensional planning on the GPU. In: 2010 IEEE International Conference on Robotics and Automation (ICRA), pp. 2515–2522 (May 2010)
9. Kirk, D.B., Hwu, W.-M.W.: Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2010)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6(2), 40–53 (2008)
11. Shekhar, S., Fetterer, A., Goyal, B.: Materialization trade-offs in hierarchical shortest path algorithms. In: Scholl, M.O., Voisard, A. (eds.) SSD 1997. LNCS, vol. 1262, pp. 94–111. Springer, Heidelberg (1997)
12. Stefan, E., Damian, S.: Parallel state space search on the GPU. In: International Symposium on Combinatorial Search, SoCS (2009)
13. Sturtevant, N.R.: Memory-efficient abstractions for pathfinding. In: Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference, Stanford, California, USA, June 6-8, pp. 31–36. The AAAI Press (2007)