Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1847 – 1856

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

GPU Accelerated Multi-agent Path Planning based on Grid Space Decomposition

Giuseppe Caggianese, Ugo Erra

Dipartimento di Matematica e Informatica, Università della Basilicata, Viale Dell'Ateneo, Macchia Romana, 85100, Potenza, Italy

Abstract

In this work, we describe a simple and powerful method to implement real-time multi-agent path-finding on Graphics Processor Units (GPUs). The technique aims to find potential paths for many thousands of agents, using the A* algorithm and an input grid map partitioned into blocks. We propose an implementation for the GPU that uses a search space decomposition approach to break down the forward search A* algorithm into parallel independently forward sub-searches. We show that this approach fits well with the programming model of GPUs, enabling planning for many thousands of agents in parallel in real-time applications such as computer games and robotics. The paper describes this implementation using the Compute Unified Device Architecture programming environment, and demonstrates its advantages in GPU performance compared to GPU implementation of Real-Time Adaptive A*.

Keywords: path-finding, GPU acceleration, A* algorithm, real-time search, search space decomposition

1. Introduction

Navigation-planning techniques for multi-agents have been traditionally studied in the domain of robotics and in recent years have been increasingly applied to real-time strategy games and non-player characters in video games. In these applications, the principal challenge is the safe navigation of an agent to its target location while avoiding collision with static or dynamic obstacles and other moving agents. Agents are therefore not directly controlled by humans but rely instead on path-planning algorithms.

A*[1] is the most famous algorithm for finding cost-minimal paths in state spaces, which are usually represented as graphs. Given a start state (or start node) and a goal state (or goal node), A* finds the least-cost path by using a distance–cost heuristic function to determine the order in which the search visits states. The search performed by A* is ideal for off-line artificial intelligence applications where for each agent we perform a search from the start state to the goal state in turn. However, it is not suitable for search paths in real-time where the topology of the graph, its edge costs, the start state or the goal states change over time. Several approaches have been developed to tackle these problems, such as Fringe Saving A*, Generalized Adaptive A*, Lifelong Planning A*, and D*. Worthy of note for this paper is Real-Time Adaptive A*(RTAA*) [2], which we used as a comparison in our performance benchmarks.

In a scenario where an agent navigates to its goal position avoiding collisions, using for instance A*, the planning for many thousands of agents lends itself well to the parallel computing data paradigm. Essentially, for each agent a

Email addresses: giuseppe.caggianese@unibas.it (Giuseppe Caggianese), ugo.erra@unibas.it (Ugo Erra)

doi:10.1016/j.procs.2012.04.203

single program, A*, consults the global connectivity data and concurrently resolves an agent's optimal path, bound by a start and goal position. As the number of agents and the size of the search space increase, planning tends to become computationally burdensome. Hence, path-finding on large maps can result in serious performance bottlenecks. 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 with large maps, 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 subpaths. These subpaths will be likely shared between different agents' paths.

In this paper, we describe an approach for a path-planning system for many thousands of agents that uses A*. The approach is based on the search space decomposition of the input grid map into blocks. For all blocks, we perform simultaneous A* searches to obtain all potential subpaths of the input agents toward the goal state that traverse these blocks. In this way, given the start positions of agents and a goal position as input, we are able to break down the forward search A* algorithm into parallel independently forward sub-searches. This approach fits well with the parallel architecture of GPUs, where many hundreds of threads are necessary to exploit the GPU fully. In addition, our method is simple and easy to implement using a GPU programming model such as the platform chosen in this work, NVIDIA's Compute Unified Device Architecture (CUDA). The empirical results demonstrate the GPU performance-speed up advantages for large numbers of agents compared to GPU implementations of RTAA* and suboptimal solutions, thus trading optimality for improved execution performance. This offers a powerful way to accelerate simultaneous path planning in real-time application as for instance robotics and computer games.

The rest of the paper is organized as follow. Section 2 relates similar work on parallel path-planning problems. Section 3 introduces formally the A* algorithm and illustrates briefly the GPU programming model. Section 4 describes the fundamentals of our parallel path-planning method. In Section 5, we present some details about the implementation on the GPU. In Section 6, we assess performance trade-off and results on quality. Finally, in Section 7, we present our conclusions and future research directions.

2. Related Work

Researchers have recently developed parallel-based implementations of path-finding that use the computational power of the GPU. In 2008, Bleiweiss [3] implemented the Dijkstra and the A* algorithms using CUDA. After several benchmarks, he observed that the Dijkstra implementation reached a speed up of a factor 27 compared to a C++ implementation without SSE instructions, while A* implementation reached speed up of a factor 24 compared to a C++ implementation with SSE instructions. In [4], 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 [5], Stefan et al. obtained speedups for breadth-first search using a bit-vector representation of the search frontier on a GPU. In [6], 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 [7], 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. In addition, benchmarks support the GPU CUDA performance scale compared to multi-threading CPU implementation in running one, two, and four threads.

These approaches provide strong evidence that GPUs can substantially accelerate path-finding algorithms, particularly for real-time applications such as video games and real-time applications in robotics, which require efficient path-finding to support large numbers of agents moving through expansive and increasingly large dynamic environments.

3. Background

In this section, we introduce the reader to the A* algorithm and the GPU programming model.

3.1. A* Algorithm

Before explaining the proposed approach, we give a brief overview of the A*. In A* (Algorithm 1), for every state s, the user supplies a heuristic $h[s]$ that estimates the goal distance, which is the cost of a minimal path from the state s to a goal state. Classical heuristics are based on Manhattan, diagonal, or Euclidean distance calculations. During its execution, A* maintains two values, $g[s]$ and $f[s]$. The value $g[s]$ is the smallest cost of any discovered path from the start state s_{start} to state s. The value $f[s] = g[s]+h[s]$ estimates the distance from s_{start} to the goal state via state s. The algorithm maintains two lists, the open list and the closed list. The open list is a priority queue and contains the most recently discovered states. Initially it contains only the start state s_{start}. The closed list contains the expanded states, those from which all adjacent states have been explored and inserted into open list. At each iteration, A* removes the state s with the smallest $f[s]$ value from the open list. If state s is a goal state, it terminates. Otherwise, it explores the adjacent states and updates the g-value of each visited state. If the g-value decreases, it updates the g-value and the corresponding f-value in the open list. It then repeats the process. Finally, the g-value of every visited state s will be the distance from the start state s_{start} to state s.

Algorithm 1: A*

```
input : A start node, a goal node, and an heuristic function
output: A partition of the bitmap
ClosedList ⟵ ∅
OpenList ⟵ start
g[start] ⟵ 0
h[start] ⟵ HeuristicEstimate (start, goal)
f[start] ⟵ h[start]
while OpenList ≠ ∅ do
    x ⟵ ExtractLowerValue (OpenList)
    if x = goal then
        return (ReconstructPathFrom (goal))
    foreach s ∈ NeighborNodes(x) do
        newg ⟵ g[x] + Cost(x, s)
        if s ∈ OpenList and newg < g[s] then
            Remove (OpenList, s)
        if s ∈ ClosedList and newg < g[s] then
            Remove (ClosedList, s)
        if s ∉ OpenList and s ∉ ClosedList then
            parent[s] ⟵ x
            g[s] ⟵ newg
            h[s] ⟵ HeuristicEstimate (s, goal)
            f[s] ⟵ g[s] + h[s]
            Add (OpenList, s)
    Add (ClosedList, x)
```

For real-time applications, A* has two main drawbacks. The first is the computational time required to perform a search from the start state to the goal state. The second disadvantage relates to memory footprint during the execution of the algorithm. Each agent must store and update the $h[s]$, $g[s]$, and $f[s]$ values for each state s. This makes A* unsuitable for multi-agent path planning in large state spaces when memory is limited.

3.2. 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, but the limitations and difficulties of a mapping a problem via graphics APIs leaved these successful experimentations only to 3D graphics experts. The demand to use the GPU as a more general parallel processor motivated NVIDIA to release in 2006 a new generation of graphics cards (the so called G80 architecture or one of its successors) that significantly extended the GPU beyond graphics through a new unified graphics and computing GPU architecture and the CUDA programming model [8].

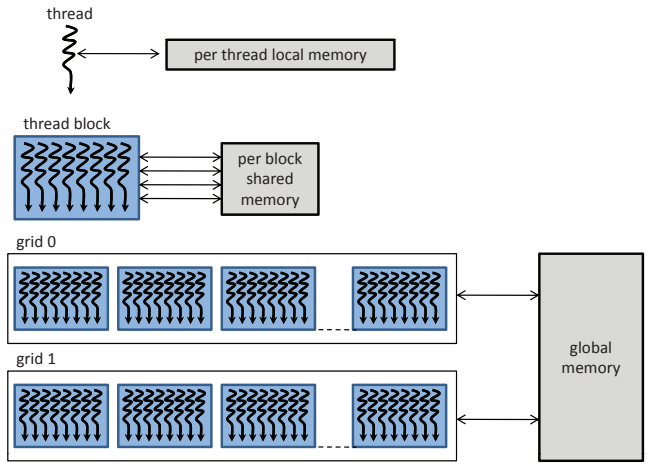


Figure 1: Levels of parallel granularity and memory sharing on the GPU [8].

From the point of view of hardware model, the GPU architecture is built as a scalable array of multithreaded multiprocessors. Each multiprocessor consist of a number of SIMD ALUs which one called processor. The processor executes at the same time the same instruction in a SIMD fashion and has access to local registers. On the multiprocessor level, all processors of a multiprocessor have read/write access to a shared memory.

From the point of view of software model, CUDA is a minimal extension to 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.

The grid and block sizes must be defined for every kernel invocation. Each block is mapped to one multiprocessor and then multiple thread blocks can be mapped on the same multiprocessor and are executed concurrently. Multiprocessor resources (registers and shared memory) are split among the mapped thread block. As a consequence, this limits the number of thread blocks that can be mapped onto the same multiprocessor. In order to maximize the number of threads supported by a multiprocessor it is important to take into account the resources required by each kernel. Then, the choice to design a framework by using several kernels is a crucial point to exploit the resources of the GPU and to maximize the amount of thread parallelism.

Further details on the GPU architecture and CUDA programming model are available in NVIDIA's CUDA Programming Guide [9].

4. The Proposed Approach

In this section, we describe an approach to compute in parallel simultaneous path planning in a multi-agent scenario. As inputs we have a set of start states, a goal state, and a search space that represents the environment in which the agents move. We first describe the reference scenario for our path planning and how we decompose the search space. We then describe the necessary steps to perform the parallel search from the start states to the goal state.

4.1. Search space decomposition

The approach we propose is suitable for scenarios that are based on a grid map. In these scenarios, the environment is subdivided into small regular zones called tiles. Each tile represents a state s 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

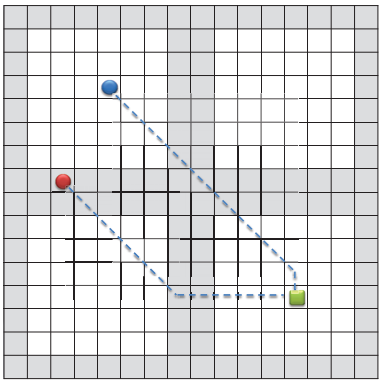
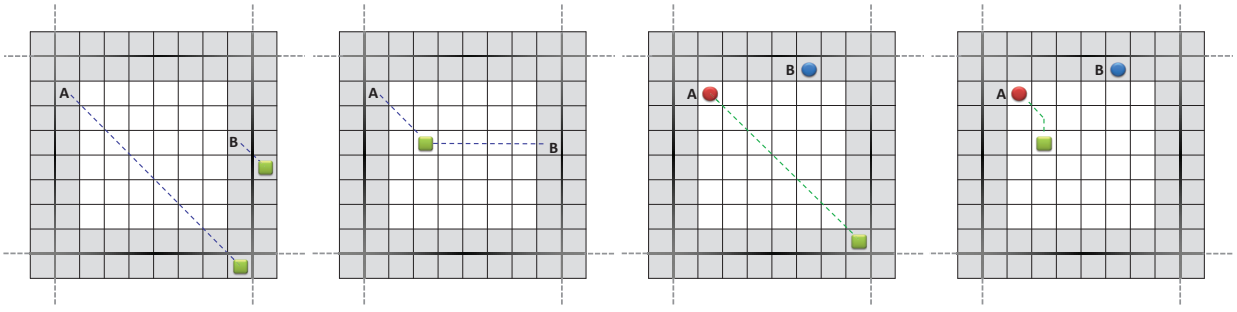


Figure 2: Example of a 16×16 grid map with four 8×8 planning blocks. Gray indicates border tiles. The circles are two agents in their start positions. The square is a goal position, and dotted lines are paths that traverse planning blocks.



(a) Traversing subpath. (b) Incoming subpath. (c) Outgoing subpath. (d) Internal subpath.

Figure 3: Left (a–b): border-to-border step. All the simultaneous A* searches start from the border tiles and terminate in a border tile of a neighboring planning block or in a goal state. Right (c–d): start-to-border step. The simultaneous A* searches start from the agent's positions and terminate on a border tile of the same planning block or in a goal state.

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.

The grid map used as search space S is further divided into k regular regions B_i called *planning blocks*. Planning blocks are all of the same size and decompose the search space into non-overlapping search sub-spaces such that $S = B_1 \cup B_2 \cup \ldots \cup B_k$. We refer to the edge states of a planning block as *border tiles*, which enable a state transition from a planning block to a neighboring block, as illustrated in Fig. 2.

Given two distinct planning blocks B_i and B_j, let $p_i = \langle s_1, s_2, \ldots, s_m \rangle$ a subpath with $s_i \in B_i$ for $i = 1, \ldots, m-1$. We name p_i a *traversing subpath* of B_i if $s_1, s_m \in B_j$ are border tiles (Fig. 3a). We call p_i an *incoming subpath* of B_i if $s_m \in B_i$ and s_1 is a border tile (Fig. 3b). If $s_m \in B_i$ is a border tile we call p_i an *outgoing subpath* of B_i (Fig. 3c). Finally, if $s_m \in B_i$ and none of the states s_i is a border tile then p_i is an *internal subpath* of B_i (Fig. 3d). A path p from a start state to a goal state has subsequent subpaths p_i with $i = 1, 2, \ldots, n$. If $n = 1$, we have only an internal subpath. With $n \geq 3$, p_1 is an outgoing subpath, $p_2, p_3, \ldots, p_{n-1}$ are traversing subpaths, and p_n is an incoming subpath.

The rationale behind the planning blocks is to break down the search of a single path, computing independently all its subpaths. By using simultaneous A* searches for all planning blocks, we compute in parallel all potential subpaths. This may pose a problem because of the dependence of the planning blocks. During an A* search a path is discovered sequentially, and if a subpath p_{i-1} inside the planning block B_{i-1} then precedes a subpath p_i inside the planning block B_i, we are unable to launch simultaneous A* searches for B_{i-1} and B_i. However, in a multi-agent scenario it is natural to expect portions of a path to be shared between agents. Thus, we can take advantage of this scenario, computing in parallel all subpaths potentially able to traverse all planning blocks.

4.2. The parallel search

To perform searches for many thousands of agents in parallel, we need to find a way of using all planning blocks simultaneously. Our solution exploits the fact that, given a set of start states and one goal state, it is likely that the discovered paths share subpaths. We attempt to determine these shared subpaths, computing in parallel all potential subpath types inside the planning blocks, taking into account that they must converge toward the goal direction. This strategy is achieved in two steps: *border-to-border* search and *start-to-border* search.

In the border-to-border step (Fig. 3a–b), we compute for all planning blocks the traversing subpaths and incoming subpaths using multiple A* searches. In particular, for a given planning block, A* searches have as start states the border tiles, and the searches determine all states along the way toward the goal position. A single A* search terminates when a border tile belonging to a neighboring planning block is discovered or when the search discovers a goal position. At the end of this step, we can assemble a path from any border tile toward the goal position, assembling a sequence of zero or more traversing paths and an incoming path. Note that this step is independent from the start positions and can be performed off-line if the goal is not expected to move anywhere.

In the start-to-border step (Fig. 3c–d), we compute the outgoing subpaths and internal subpaths using multiple A* searches. In this case, for all the input agents, A* searches have as start states the start positions of all agents, and the searches determine, as described above, all states along the way toward the goal position. A single A* search terminates when a border tile belonging to the same planning block is discovered or when the search discovers a goal position. Thus, the objective of this step is to search the paths of all agents from their start positions to the nearest border tiles.

The advantage of this approach is that it can be easily implemented in parallel because all the searches in the border-to-border step and in the start-to-border step can be performed simultaneously. We have removed the dependence between planning blocks taking into account all possible subpaths inside the planning blocks. Furthermore, the ability to use small search areas ensures faster searches and limits memory use. Finally, in the case of a change in start position, we need only to perform a start-to-border search where the change occurred.

A problem associated with this approach is the formation of loops at the end of the border-to-border step, as illustrated in Fig. 4. There are two possible strategies to tackling this problem. The first is to handle the loop during the movements of agents, e.g., prevent the agent from becoming trapped in the loop. The second is to increment the heuristic associated with the border tiles that form the loop and then perform a new border-to-border step only for the planning blocks involved. In this way, these border tiles will not be taken into account in the new paths because of their higher movement cost.

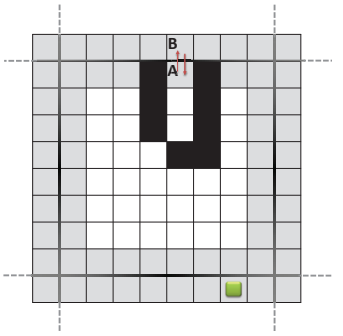


Figure 4: A dead-end causes the A* search from border tile A to stop in border tile B. Conversely, A* search from border tile B stops in border tile A.

5. GPU Implementation

Implementation of the approach described above on a GPU programming model is straightforward. To execute all searches simultaneously, we associate a single GPU thread with each search; in fact, each search executes the same instructions but with different data. In the border-to-border step, we couple a single thread block for each planning block, as illustrated in Fig. 5. This decision enables us to execute concurrently all border-to-border searches and to

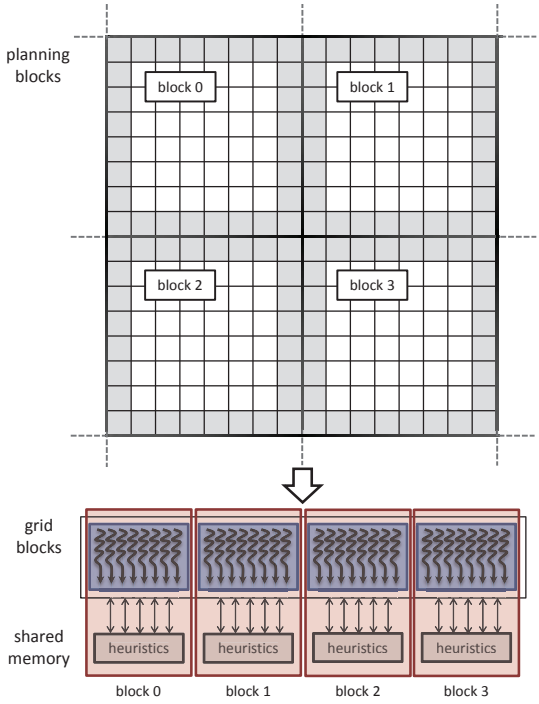


Figure 5: Parallelizing the border-to-border step using GPU. A planning block is associated with a thread block, and a thread executes a search for each border tile. This mapping enables us to use the shared memory in a thread block to store the estimated heuristics of a planning block. Finally, all planning blocks are gathered in a single CUDA grid.

share information through shared memory. Indeed, heuristic values $h[s]$, used to estimate the goal distance for each state s, are stored as a single array in the large shared memory as a planning block. This is possible because in the border-to-border step all the searches are local, i.e., a search starts and stops in the same planning block.

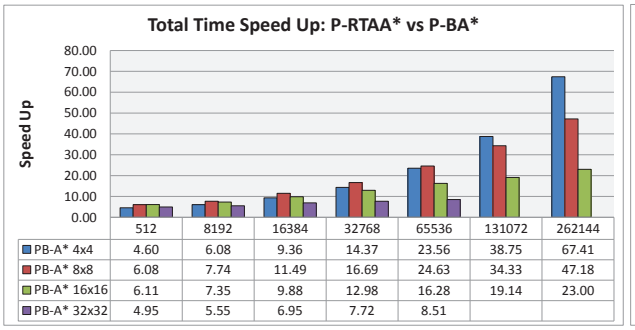
Conversely, in the start-to-border step, agents' start positions are not all located in the same planning block. However, even in this case, the array for heuristic values can be shared between agents. In fact, the array may become as large as the entire map and must then be stored in a global memory because its dimensions are too large for shared memory. For both types of steps, another element that is shared between all agents and stored in a global memory is the map that retains the cost of movement from each tile to its neighbors and therefore also the positions of obstacles.

The planning-block size affects the behavior of searches because it determines the number of states in a single planning block, the array dimensions for shared memory used to maintain the heurists, and the number of border tiles. If the planning block is too large, the A* algorithm is required to explore too many states, while if it is too small, there are fewer states to explore but many searches to execute in the same length of time. However, to optimize time execution, we select the planning-block size to always be a power of 2 so that we can use CUDA bitwise operations to replace integer division and modulo operation, which are too expensive to run and are necessary to retrieve tile coordinates and consequently the planning-block ids.

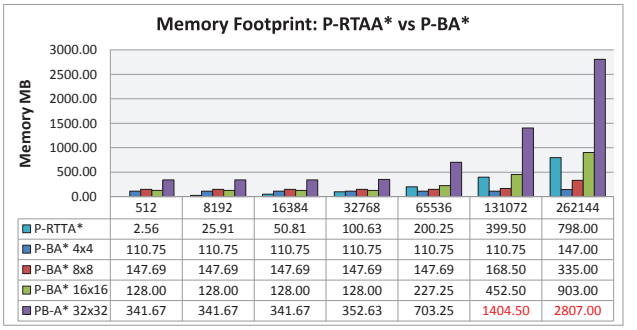
6. Experiments and Results

In this section, we provide the results of two experiments. The first type of experiment demonstrates the efficiency of our approach, while the second is related to the quality of the trajectory found using the search space decomposition. Our tests were performed on an Intel Core i7 CPU 1.6GHz, NVIDIA Fermi GTX 480 1.5GB, Windows 7. All the kernels were written in CUDA 4.0 and using Microsoft's Visual C++ 2010 compiler.

The first experiment compares our GPU parallel approach based on planning blocks (P-BA*) with a GPU implementation of RTAA* (P-RTAA*) [2]. RTAA* is a real-time heuristic search method that selects its local search spaces in a very fine-grained way. The basic principle is to update the heuristics of all states in the local search space swiftly

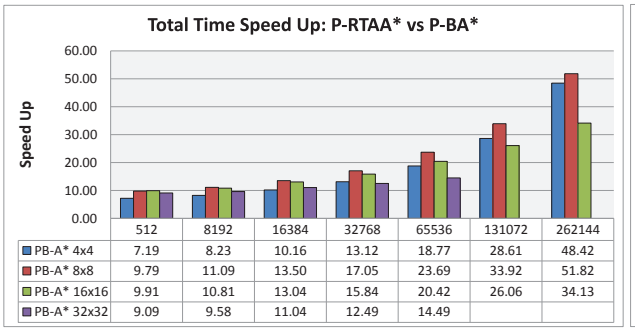


	512	8192	16384	32768	65536	131072	262144
PB-A* 4x4	4.60	6.08	9.36	14.37	23.56	38.75	67.41
PB-A* 8x8	6.08	7.74	11.49	16.69	24.63	34.33	47.18
PB-A* 16x16	6.11	7.35	9.88	12.98	16.28	19.14	23.00
PB-A* 32x32	4.95	5.55	6.95	7.72	8.51		

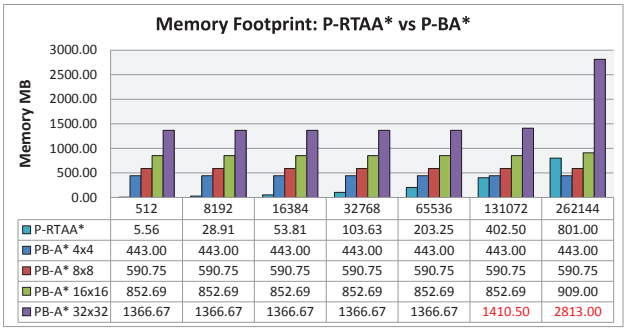


	512	8192	16384	32768	65536	131072	262144
P-RTTA*	2.56	25.91	50.81	100.63	200.25	399.50	798.00
P-BA* 4x4	110.75	110.75	110.75	110.75	110.75	110.75	147.00
P-BA* 8x8	147.69	147.69	147.69	147.69	147.69	168.50	335.00
P-BA* 16x16	128.00	128.00	128.00	128.00	227.25	452.50	903.00
PB-A* 32x32	341.67	341.67	341.67	352.63	703.25	1404.50	2807.00

(a) GPU total time speed-up and memory footprint for a 512×512 map.



	512	8192	16384	32768	65536	131072	262144
PB-A* 4x4	7.19	8.23	10.16	13.12	18.77	28.61	48.42
PB-A* 8x8	9.79	11.09	13.50	17.05	23.69	33.92	51.82
PB-A* 16x16	9.91	10.81	13.04	15.84	20.42	26.06	34.13
PB-A* 32x32	9.09	9.58	11.04	12.49	14.49		



	512	8192	16384	32768	65536	131072	262144
P-RTAA*	5.56	28.91	53.81	103.63	203.25	402.50	801.00
PB-A* 4x4	443.00	443.00	443.00	443.00	443.00	443.00	443.00
PB-A* 8x8	590.75	590.75	590.75	590.75	590.75	590.75	590.75
PB-A* 16x16	852.69	852.69	852.69	852.69	852.69	852.69	909.00
PB-A* 32x32	1366.67	1366.67	1366.67	1366.67	1366.67	1410.50	2813.00

(b) GPU total time speed-up and memory footprint for a 1024×1024 map.

Figure 6: GPU total time speed-up values and memory footprint compared to GPU implementation of RTAA* with groups of agents ranging in size from 512 to 262144. We also include the time to transfer data from CPU to GPU and vice versa. Memory footprints are for border-to-border and start-to-border searches. Note that in two cases the required memory is higher than the memory available for computation.

and to save the heuristics so as to speed up future A* searches. This approach uses a variable called `lookahead`, which specifies the largest number of states to expand during A* searches, and was used in the GPU implementation to reduce the memory footprint required for each agent. We chose to compare our approach with a GPU parallel version of RTAA* because in previous work [7] this implementation was found to be faster than a parallel GPU implementation of A* [3].

Two grid maps measuring 512×512 and 1024×1024 with several groups of agents ranging in size from 512 to 262144 were used to assess performance and memory footprint with planning blocks measuring 4×4, 8×8, 16×16, and 32×32. In P-RTAA* the number of searches and then threads run on the GPU is always equal to number of agents. Conversely, in P-BA* the number of agents determines only the number of start-to-border searches because border-to-border searches depend on the planning-block dimensions and on the number of border tiles. For instance, in the 1024×1024 map we have 786432, 458752, 245760, and 126976 threads for all planning-block sizes tested.

In all configurations, start positions were randomly chosen in the grid map, whereas the stop tile was always the center tile of the map. Also, the heuristic values $h[s]$ are precomputed off-line and stored in a matrix large as the grid maps. Figure 6 reports GPU total speed-up time and memory footprint compared to P-RTAA*. GPU implementation of RTAA* is always executed with `lookahead` = 3, which is the optimal value for achieving best performance as described in [7]. The results indicate that our approach is faster than P-RTAA*. The average speed up acceleration for each group of agents ranging from 5X to 45X in the 512×512 map and from 9X to 44X in the 1024×1024 map; measured time values include memory transfer time (CPU to GPU and vice versa) and kernel execution time. However, although we observed that shared memory improves performance, its use implies a degree of variability across the tested configurations. Conversely, P-RTTA* exhibited a better memory footprint in most cases, because of the greater amount of memory required to store the searches generated in the border-to-border step compared with P-RTAA*. However, as the number of agents increases, the number of searches is expected to rise considerably, and so also the memory footprint. Note that in general a planning block measuring 8×8 offers better performance in terms

Map	Obstacle Rate	A*	P-BA*-4	P-BA*-8	P-BA*-16
32×32	10%	33,850	33,897	33,838	33,778
	15%	35,102	35,500	35,489	35,277
	20%	37,553	36,822	37,016	37,181
	25%	39,338	38,000	38,441	38,625
64×64	10%	68,53	68,67	68,29	68,45
	15%	70,66	71,88	71,32	71,44
	20%	74,66	73,50	73,83	73,75
	25%	79,78	77,40	77,29	78,24
128×128	10%	138,12	138,88	138,48	138,09
	15%	142,84	143,38	142,84	142,92
	20%	151,02	148,60	149,32	149,41
	25%	161,43	-	155,00	153,00
256×256	10%	277,10	277,93	276,73	276,49
	15%	287,48	288,60	287,93	286,56
	20%	301,57	292,00	296,40	298,64
	25%	324,21	-	314,00	309,75

Table 1: Average steps of A* and P-BA*.

of acceleration and memory footprint.

The second type of experiment concerns the lengths of paths obtained via our approach through the introduction of planning blocks. We measured the average path length using 32×32, 64×64, 128×128, and 256×256 grid maps with an increasing rate of obstacles. One A* search and three P-BA* searches were performed with planning blocks measuring 4×4, 8×8, and 16×16 with the upper-left corner as the start position and the lower-right corner as the goal position. Table 1 lists the average path lengths for 100 runs. For each run, we placed obstacles chosen randomly, and because of this, there may have been maps where there was no path from start to goal positions. The results indicate that the length of the path retrieved with our approach is substantially the same as calculated with sequential A* and increasing the size of planning blocks involves a path length near the optimal solution.

These experiments suggest that our approach finds paths whose difference from the optimal path length is not significant. This deficiency is compensated for in terms of efficiency, as shown in the performance experiments. Fine-tuning the planning-block dimensions allows the user to trade off speed against path optimality. For example, in real-time applications, speed is the highest priority and suboptimal paths may thus be acceptable.

7. Conclusion

In this work, we have demonstrated a parallel implementation based on the A* algorithm that fits well with GPU parallel architecture. By using it to explore each potential subpath per thread, the method offers a simple and powerful way of planning trajectories for many thousands of agents in parallel. Our results show that the GPU implementation improves by up to 45 times on that of RTAA*, allowing the real-time use of this technique even in scenarios with a vast number of agents, which is common in applications such as video games.

Future work may explore further the shared memory and in particular explore how to improve its impact in the border-to-border step. It would also be interesting to investigate the management of dynamic obstacles that occur in the grid map. Once the system recognizes the presence of dynamic obstacles, only in the planning blocks where they occur will a new border-to-border and/or start-to-border step be performed. Thus, all retrieved paths combining multiple subpaths should be able to adapt swiftly to changes in the map.

We also propose an extension of this technique to manage very large maps using an out-of-core technique to reduce memory footprint requirements. Finally, we intend to release this implementation over an open source license.

References

[1] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, Systems Science and Cybernetics, IEEE Transactions on 4 (2) (1968) 100 –107.

[2] S. Koenig, M. Likhachev, Real-Time Adaptive A*, in: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, AAMAS '06, ACM, New York, NY, USA, 2006, pp. 281–288.
[3] A. Bleiweiss, GPU accelerated pathfinding, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '08, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008, pp. 65–74.
[4] G. J. Katz, J. T. Kider, Jr, All-pairs shortest-paths for large graphs on the GPU, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '08, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008, pp. 47–55.
[5] E. Stefan, S. Damian, Parallel state space search on the GPU, in: International Symposium on Combinatorial Search (SoCS), 2009.
[6] J. Kider, M. Henderson, M. Likhachev, A. Safonova, High-dimensional planning on the GPU, in: Robotics and Automation (ICRA), 2010 IEEE International Conference on, 2010, pp. 2515 –2522. doi:10.1109/ROBOT.2010.5509470.
[7] U. Erra, G. Caggianese, Real-time Adaptive GPU multi-agent path planning, GPU Computing Gems Jade Edition Edition, Vol. 2, Morgan Kaufmann Publishers Inc., 2011, Ch. 22, pp. 295–308.
[8] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2) (2008) 40–53.
[9] Nvdia, NVIDIA CUDA Compute Unified Device Architecture - Programming guide.