Efficient Implementation of Parallel Path Planning Algorithms on GPUs

Ralf Seidler, Michael Schmidt, Andreas Schäfer and Dietmar Fey

Department of Computer Science, Chair of Computer Architecture, FAU Erlangen-Nuremberg, Germany

Abstract— In robot systems several computationally intensive tasks can be found, with path planning being one of them. Especially in dynamically changing environments, it is difficult to meet real-time constraints with a serial processing approach. For those systems using standard computers, a promising option is to employ a GPGPU as a coprocessor in order to offload those tasks which can be efficiently parallelized. We implemented selected parallel path planning algorithms on NVIDIA's CUDA platform and were able to accelerate all of these algorithms efficiently compared to a multi-core implementation. We present the results and more detailed information about the implementation of these algorithms.

***Keywords*—CUDA, GPGPU, parallel algorithms, path planning**

I. INTRODUCTION

Path planning for robots in dynamic environments is a highly challenging task [1] since the control units have to meet real time requirements, for instance in order to ensure the safety of human personnel [2]. This paper evaluates a number of approaches from the broad field of parallel path planning algorithms in order to assess their performance on GPGPUs.

A lot of robot systems are built of standard PC hardware for processing and controlling, especially in systems where a global view is necessary. Additionally, PCs offer the benefit of being compatible with standard software. In this setting it is sensible to harness the GPU of the PC as a fast coprocessor for tasks like image processing and especially path planning, which are both computationally intensive and can be efficiently parallelized.

Possible scenarios for such a system are some types of robot soccer applications, where scenes of the playing field are captured by a camera streaming the images to a PC for processing. Systems in use deliver frame rates of up to 100 fps[3]. Another conceivable use case of such systems are buildings, e.g. hospitals, with camera-controlled cleaning or service robots. The environment changes dynamically as humans moving through the same corridors.

In all these applications path planning is one of the most challenging tasks. A serial processing approach is rarely sufficient, since strict real-time requirements in the range of milliseconds have to be fulfilled. A parallel processing approach is more promising.

Several parallel path planning algorithms have been published. We evaluated the most promising ones which can be classified by following categories: *Visibility Graphs* [4] representing graph-based approaches, *Potential Field* [5], a special potential field planner denoted as *Wavefront* [6], *Autowaves [7]*, and an emergent approach invented by us, based on hardware agents called *Marching Pixels* [8].

We have implemented each of these path planning algorithms with NVIDIA's CUDA toolkit as well as OpenMP. Evaluation was carried out on a system featuring an Intel Core i7 quad-core and an NVIDIA Geforce GTX 480 GPGPU. In [9] we presented the comparative results and showed, that all path planning algorithms can be efficiently accelerated by the sourcing them out on a GPU. This paper focuses on the GPU implementations for details. We will highlight advantages and disadvantages of GPUs which should help in further applications to use specific features of GPUs efficiently.

II. RELATED WORK

A GPU accelerated path planning approach for multi-agents is presented in [10]. The main goal of this approach is to produce a collision free natural steering for many thousands of agents moving in virtual environments. Another GPU accelerated pathfinding algorithm is presented in [11], also dealing with autonomous navigation of many thousand agents. In [12], a planning algorithm for a flexible robot in complex medical environments is presented, e.g. for the insertion of catheters. Precision is the most important constraint for this application and it is not targeted at real-time applications.

All of these approaches have in common that they are focused on special applications. Their primary goal was to speed up a given serial algorithm by parallelizing it for the GPU.

Contrary to that, the focus of our work is to evaluate existing parallel path planning algorithms on a GPU, to compare them with a standard multi-core implementation and to show how they are implemented efficiently by the usage of GPU specific features.

III. PARALLEL PATH PLANNING ALGORITHMS

A. Abstractions and Assumptions

A path planning algorithm calculates a collision free path from a given start position to a given target position [1]. Prior to comparing different algorithms it is necessary to establish a common benchmark setup. First of all we consider the workspace to be two-dimensional.

To create a uniform benchmark scenario we used a list of nodes of polygonal obstacles as input for all algorithms. Those algorithms needing a map assemble it with negligible overhead before processing. As sizes for the $n \times m \in N^2$ maps, we used standard resolutions of 320×240, 640×480, 800×600, 1024×768 and 1000×1000. Each point of such a map is referred as *configuration* $q(i,j)$, with $i = 1..n, j = 1..m$.

A problem is that the algorithms expect different inputs for processing: The approaches based on Autowaves, Wavefronts and Marching Pixels operate directly on a discretized map which is in our case a binary image representing obstacles and free space of the environment. The graph-based version reads a

DOI: 10.5176_2010-2283_1.2.47

©2011 GSTF

list of cartesian coordinates marking the corners of the obstacles. The Potential Field Method creates a map of potentials with the help of either obstacle coordinates or a binary map.

B. *Visibility Graph*

A *Visibility Map* is a type of a roadmap [1] and one of the earliest path planning methods [4]. It represents the map where the robot is moving in a topological structure, called the *Visibility Graph*. Visibility Graphs are most suitable for maps with polygonal obstacles.

Let $V = \{v_1, \ldots, v_l\}$ be the nodes of the graph which represents the vertices of the polygonal obstacle and the start and goal positions. At the beginning, the graph contains an edge for each edge of an obstacle polygon. Additional edges are being added during the construction of the graph when two vertices are in line of sight with each other for every vertex $v \in V$. It must be determined if it is visible for the other vertices $v_i \in V$. The most obvious way is to test for all line segments $\overline{vv_i}, v \neq v_i$, if they intersect with an edge of an obstacle polygon. A common simplification of this is not to test all edges of the polygon, but only the bounding circle. If no intersection exists, they are part of the Visibility Graph. The intersection tests can be done independently for every line segment and therefore realized efficiently in parallel. The edges of the graph are weighted with the euclidean distance of the two associated vertices because we want to find the shortest path from the start to the goal vertex.

Several search algorithms for weighted graphs have been published. We decided to use Visibility Graphs combined with *Dijkstra's* search algorithm for weighted graphs [13] as representation of graph-based path planning algorithms. The naive serial algorithm for constructing the graph requires $O(l^3)$ operations and the search algorithm needs $O(l \cdot log\ l)$ steps where l is the number of vertices.

C. *Potential Field Method*

In potential field approaches the robot is directed through a map with the help of a differentiable real-valued potential function U. It was originally developed as an approach with collision avoidance in dynamically changing environments [8]. The potential is typically defined as the sum of an attractive potential pulling the robot to the goal configuration and a repulsive potential pushing the robot away from obstacles. The function U_{att} attains its minimum in the goal configuration and should be monotonously increasing with the distance from goal. With the help of the repulsive potential U_{rep} a potential barrier around obstacles is created in order to avoid collisions. Because of the discrete representation of the map in our consideration (discretized configuration space) each configuration of the map is assigned to a discrete potential value which builds a potential field where the smallest potential value is found in the goal configuration. The potentials for every configuration q of the configuration space are realized by (1). The parameter d represents the distance $d = |q - q_{obj}|$ from a configuration to the centroid points q_{obj} of the obstacles. The attractive potential creates a conic well which is defined with a positive scaling factor σ. The well has its minimum in the goal configuration q_{goal}. The main idea of the repulsive potential is to create a potential barrier around obstacles. The barrier of the obstacle potentials is affected by the value rep and depends on the application. The value a is a high value for the potential inside obstacles which prevents a robot from moving into an obstacle. Around the obstacles a gradient potential is defined as a so called *distance of influence* which could be considered as a safety zone.

$$\begin{aligned} U(q) &= U_{att}(q) + U_{rep}(q) \\ U_{att}(q) &= \sigma|q - q_{goal}| \\ U_{rep}(q) &= \begin{cases} a\,\frac{1}{d} - \frac{1}{rep}, & if\ d < rep \\ a, & if\ d = 0 \\ 0, & else \end{cases} \end{aligned} \quad (1)$$

The robot follows the negated gradient of the potential values in the field. The resulting path is planned iteratively from the start configuration to the goal configuration with collision avoidance. Following such a path is called *gradient descent*.

For the calculation of the potential field, $O(m \cdot n)$ steps are needed where $(m = width, n = height)$ is the grid size of the map. The gradient descent in maps with convex obstacles takes approximately $8(m + n)$ steps because all eight direct neighbors of every considered path point need to be calculated to determine the next path point. The main disadvantage of the Potential Field Method is that there may be local minima trapping the robot. This can be avoided by introducing an efficient escape strategy to overcome the local minima. We used the *Best-First* planning [4] which "fills up" a local minimum with penalty value, resulting in a worst case complexity of $O(2 \cdot r^2 \cdot log\ r)$, where $r = max(m, n)$.

D. *Wavefront*

The Wavefront Planner is a special potential-field approach developed for grid-based maps [1],[6]. In contrast to the "standard" Potential Field Method, the potential field is constructed iteratively. Only the free configurations of the map are considered and marked with a *zero*, all obstacle configurations are marked with a *one*. The goal configuration is first labeled with a *two*. In the next iteration, all *zero*-valued configurations neighboring the goal are labeled with a *three*. In the next step all *zero*-valued configurations neighboring a configuration labeled with a *three* are labeled with a *four* and so on. Hence a wave front is generated growing from the goal configuration until the start configuration is reached.

After that the planner starts the gradient descent from the start configuration to calculate the shortest path. Because of the different construction of the potential field it is guaranteed that there will always be a neighboring configuration with a value that is by one less than the value of the current configuration, thus avoiding local minima.

In our scenarios, the wave reaches the start configuration within $m + n$ iterations. Each iteration takes $m \cdot n$ cell updates in a naive implementation, resulting in a runtime complexity of $O\big(m \cdot n \cdot (m + n)\big) = O(r^3)$, with $r = max(m, n)$.

E. *Autowaves*

This approach was developed for real-time, wave-based robot navigation in dynamically changing environments [7]. It is based on reaction-diffusion (RD) processes which appear for example in chemical applications where the so called *Autowaves* are generated. During path planning the goal configuration generates attracting waves, while the obstacles are generating repulsive waves. On the basis of overlapping

©2011 GSTF

these waves at the robot's current position the next movement is determined. In particular the robot moves in the direction where the attracting wave comes from, while being pushed with the wave-direction of the repulsive wave.

The processing of such Autowaves can be implemented for instance via *Cellular Neuronal Networks* (CNN) [14], particularly with RD-CNN, which is a special two-layer model for the processing of discretized reaction-diffusion equations.

We used Equation (2) for the processing of Autowaves. It contains two sublayers which are necessary for an oscillation to generate waves. The two sublayers of the RD-CNN are labeled with u and v. The characteristics of the waves are defined by the coefficients a and b.

$$p_{t+1}(q) = \begin{pmatrix} u_{t+1} \\ v_{t+1} \end{pmatrix} = \begin{pmatrix} a \cdot u_t + b(\sum_{neighbor(p)} u_t) - v_t \\ a \cdot u_t + b(\sum_{neighbor(p)} v_t) - u_t \end{pmatrix} \quad (2)$$

For the calculation of the goal waves and the obstacle waves we need two layers of Equation (2). One layer generates the waves of the goal in the map, the other one generates the obstacles waves. Every configuration q is assigned two vectors of p which represent the values of the goal waves and obstacle waves. The robot can determine by these values in which direction it should move next.

To ensure the full propagation of the waves among all pixels of the image, $5r$ iterations, with $r = max(m,n)$, are taken in our scenarios. Each iteration also takes $m \cdot n$ cell updates, resulting in a complexity of $O(m \cdot n \cdot r) = O(r^3)$.

F. Marching Pixels

Marching Pixels (MP) are virtual agents akin to artificial ants. They collect information deposited from other MPs for future behavior. MPs are very simple units, but together they can tackle complex tasks[15],[16]. MPs can be modeled with cellular automata and can be used for many image processing tasks, e.g. parallel centroid detection of multiple objects or path planning. The algorithm presented in [8] combines a skeletonization operation [17] with the MP concept. The skeletonization is performed on the free space of the map where the robot can move freely. The resulting skeleton is an approximated Voronoi diagram [17]. After the skeletonization, an MP runs from the start position to the goal position across the skeleton. On a crossroad new MPs are created on not yet visited paths. If two MPs meet each other or if a MP reaches an already visited crossroad then this path of the skeleton can be safely deleted because it can not be part of the shortest path. The MP which reaches the goal position first succeeds and only this path remains. All other paths are deleted. An animation of the algorithm can be found in [19].

Let $r = max(m,n)$ be the maximum of the height and width of the grid. According to [13] the skeletonization step takes $2r$ iterations. The runtime of the MP algorithm is empirically determined with $5r$ iterations as explained in [8], one iteration needs to update mn cells. The resulting complexity would be $O(m \cdot n \cdot r) = O(r^3)$.

IV. CUDA Implementation

In this section we present details related to the CUDA implementations of the different path planning algorithms. All algorithms receive the centroids of the obstacles as well as the start and end point as input. Then an obstacle growing is performed, so that the resulting path is not too close to objects. This is done by introducing a bounding box around the objects. To ensure a stable, reproducable setup, we did not use real input images from a camera system but generated the benchmark images directly on the test system.

For the implementations presented in section *IV-B - IV-E*, each pixel is mapped onto a single GPU thread. In the implementation described in section *IV-A*, every intersection test is mapped to one thread. In contrast to the solutions described in the section *IV-A* and *IV-B* the threads of *IV-C*, *IV-D* and *IV-E* have to communicate with each other because of local data dependencies.

The OpenMP solution does not differ much from the presented CUDA implementations, except that fewer threads are available than in CUDA and OpenMP parallel pragmas are used. A thread is not mapped to a pixel, but many pixels are mapped to a single thread.

A. Visibility Graph

The implementation of this algorithm is split into several stages. At first a complete graph is computed on the host where the obstacles are represented by the four corner-points of their bounding box which form a clique. For every edge of this complete graph we need to test if it belongs to the Visibility Graph. Therefore every edge - except for the obstacle edges, which are by definition part of the resulting graph - has to be checked for intersection with each circle of an obstacle. This can be done efficiently on the GPGPU. Every thread takes one edge of the whole graph and checks it against all obstacle circles for intersection. If there is one, a large penalty value is inserted into the adjacency matrix. Otherwise the Euclidean distance of this edge is inserted which is necessary for the further computation of the shortest path. For that we access the global memory by loading the current edge of the graph, each obstacle and save the distance in the adjacency matrix. Even large numbers of obstacles, fit perfectly into the constant memory of the device. With 64 kByte available, that would be 8192 obstacles. But since every obstacle has four edges, that would result in about 32.000 vertices for the graph, far more than necessary for our test-cases, so this seems to be a good choice. Unfortunately the new Nvidia Geforce GTX 480 does not yield any performance gain, but older GPUs based on the GT200 chipset can profit of that enhancement. We pointed out, that the availability of a large cache in the new architecture makes this constant memory for obsolete this application.

When the adjacency matrix is build, it is copied back to the host, where then the shortest path can be computed. The search of the shortest path with Dijkstra needs only a fraction of the time necessary for the construction of the Visibility Graph. That is why we decided to implement the search on the host. Other works [12] show, that even this search can be efficiently implemented on the GPGPU.

B. Potential Field Method

In this algorithm, the only off-loadable function is the creation of the potential field. For that, we decided to have at least 16×16 threads active per thread-block. If using less threads, more thread-blocks would be necessary and so the utilization of the given hardware would be worse, resulting in reduced latency-hiding and therefore reduced execution speed.

At first the object middle points are copied to the device.

©2011 GSTF

Then the computation kernel is started. In that, the attractive potential is computed pixel-wise according to Equation (1). Then all configurations are tested for being inside the radius of an obstacle. In such a case the potential is increased by the result of Equation (1). The last stage of the device function is to insert the potential of the walls, so that the robot cannot leave the field. This is done by adding the repulsive potential in the wall's range to the global potential field.

The computation of the potential field can be efficiently done in parallel, but the search of the shortest path with the gradient descent is a serial process. Therefore the potential field is transferred back to the host for final processing. Beginning with the target point, we compute the gradient for all neighbor points. The point with the largest gradient now continues to be our next point, inserted into a list of path-points. After some iterations a complete path is planned. To avoid a local minimum, we need to implement an escape strategy: when we reach a local minimum, a best first search is taken where all previously visited points are discarded. This is done by inserting the cardinal neighbors, sorted according to their potential value, into a list. Then the point with the smallest potential is extracted and all its neighbors are also inserted into the sorted list. Afterwards the gradient is determined. If it points towards an already visited point, the point is inserted into a visitation list. Otherwise the gradient is traced as usual. In this circumstance, when we speak of a worst-case scenario for this algorithm, an obstacle constellation that forms an L between the start and the goal is meant.

C. Wavefront

Since all the algorithms receive only the middle points of the obstacles as input, for this cellular automaton based algorithm, we need to generate a binary map, representing the current constellation of the obstacles in the configuration space. This map is used as fast lookup table (LUT), determining if a thread belongs to an object or not. Furthermore, as an initial state of the automaton we introduce another array, where every cell except the start point (this is set to *zero*) is set to the largest possible value. All of that is done in a separate kernel function.

After this is done, the initial state array is bound to a texture and the kernel for computing one iteration is started, writing the resulting states to another array. These arrays are now exchanged in every iteration of the algorithm. The final result is generated by restarting the kernel, which is required due to the local data dependencies of the algorithm. Every thread of the kernel function peeks in every cardinal direction if there is a value smaller than the current one is has. For that we need five loads of the surrounding cells as well as one for the LUT. The smallest value is copied into the local cell and increased by one. So we only need one write operation on the global memory. After the iterations, the complete image is filled and can then be transferred back to the host, where a gradient descent is applied. Since there are no local minima problems, we do not need a special escape strategy.

D. Autowaves

Autowaves are generated separately for the attracting and repulsive waves. Therefore two initial 2D-arrays are set up, one with the goal, the other one with all obstacles. This is done by a kernel that uses the obstacle centroids as well as the target position as input. Every thread handles one cell of both 2D-arrays and decides if it is inside the bounds of an object respectively target and updates the corresponding array. So we get again a LUT for our computation kernel.

Afterwards, the first array is bound to a texture, used by the RD-CNN kernel as input. The results are computed according to Equation (2) with the parameters $a = 1.3, b = 0.1$ where the neighboring values are fetched from the texture. These parameters guarantee a stable and fast wave propagation. Since the results grow rapidly in size, we decided to define an upper/lower limit so that the result is always well defined.

Textures are always read-only and, since no inter block synchronization is available, the kernel is run once per iteration. The textures are rebound to the new result and the old texture becomes the target buffer for the next iteration.

After the waves have spread over the whole image, the last and the current result are transferred to the host. According to the presented technique, a path is planned by comparing these results. This is done in a dynamical way, which means that the complete path to the goal is not calculated in one step, only the direction is determined for the next image taken.

E. Marching Pixels

Our experiences with straightforward implementations have shown, that the most common problem for solving marching pixel algorithms is the use of a large cellular automaton where each update of a cell involves a lot of branches, leading to divergent behavior within warps. Another problem we encountered, is the disadvantageous ratio of memory bandwidth to computational requirements.

The first issue can be solved by computing a LUT, which stores the resulting new state for any combination of cell and neighbor states. The states are combined by masking and shifting bits in order to obtain an address which in turn is used to load the new state from the LUT.

The LUT can be computed once, before path planning starts and can be reused for future invocations. However, due to the number of states a cell can take on (eight) and the number of cells to be considered (eight neighbors plus the cell's own state), the LUT for the given algorithm is about 135 MBytes is size. On a GPGPU, this would require an additional, most likely non-coalesced read from global memory per update, thus seriously hurting performance.

Therefore we did split the Marching Pixels algorithm into two phases: first the skeletonization, then the Marching Pixels phase. Since the skeletonization can be implemented with basically two states (a little precaution has to be taken when considering a path's start and end points), the LUT can be exponentially shrunken to 4096 Bytes. This is more than the expected 2^9 Bytes since we have to consider the direction of the skeletonization and if the current cell is the start or end point.

The other problem can be mitigated by using textures, which cache redundant accesses. We further optimized this by storing a two times two pixel block in each texture byte and letting each thread update a total of four times four pixels per step. This way a thread has to load 16 Bytes (four for own cells and twelve neighboring bytes) and write four bytes per step while performing 16 updates, leading to a ratio of 1.25 Bytes per update. This is a significant improvement over a naive implementation in which each thread would load nine bytes,

©2011 GSTF

perform one update and write one byte back, resulting in a ratio of 10 Bytes per update. Of course this packed storage requires a lot of bit masking operations, but calculations are available in abundance on a GPU. Together with using textures to cache redundant reads, this allowed us to significantly reduce the impact of memory latency and improve our bandwidth usage.

For the Marching Pixels phase we did not use a LUT, but did perform the necessary computations directly in the code. We were able to speed up this phase however, by not considering the whole image in each time step, but rather only those pixels who did form the resulting skeleton in the first phase. Cells are stored in two two-dimensional arrays, one for the old states and one for the new states which are to be computed. After each time step, the newly generated image array is bound to a texture, while the old texture becomes the result for the next iteration. As discussed in Chapter *III* the algorithm is stopped after at most $5 \cdot n$ iterations and the result is copied back to the host.

V. RESULTS

Our system consists of an Intel Core i7 920 quad-core@ 2.66GHz with 12GB RAM and a Geforce GTX 480 with 1.5 GB GDDR2 RAM. For programming NVIDIA Toolkit version 3.0 with NVIDIA SDK 3.0 was used. The results are given in two figures since the Visibility Graph's performance depends on the number of obstacle nodes and not on the maps resolution. The other approaches are oblivious to the number of object nodes, but depend on the map resolution.

Figure 1(a) shows absolute run times of the resolution-dependent algorithms on the Core i7 multi-core and Figure 1(b) shows the run times of the same algorithms on our GPU system. The Visibility Graph results for the multi-core and GPU run times are shown in Figure 2. The time for the data transfer from and to the GPU is included in the run times of the GPU implementations. The comparison of the multi-core implementations of Figure 1(a) to their GPU counterparts of Figure 1(b) shows that all algorithms can be accelerated on the GPU, by a factor between 2 and 400, depending on the algorithm and map resolution. The approaches based on

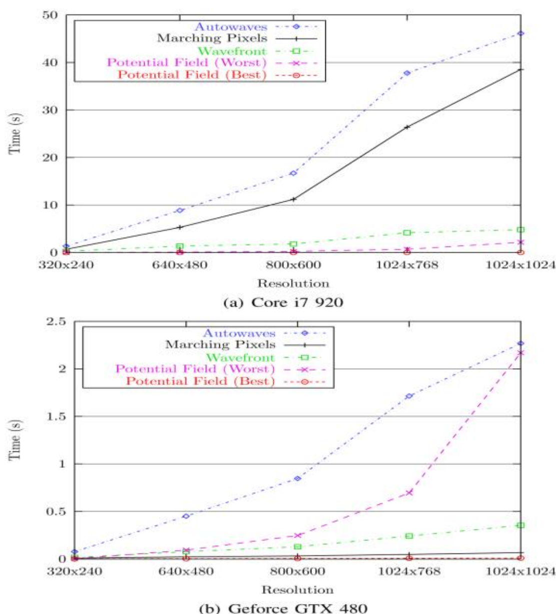

Fig. 1 Timings of pathfinding algorithms for different image sizes. Please observe the different scalings.

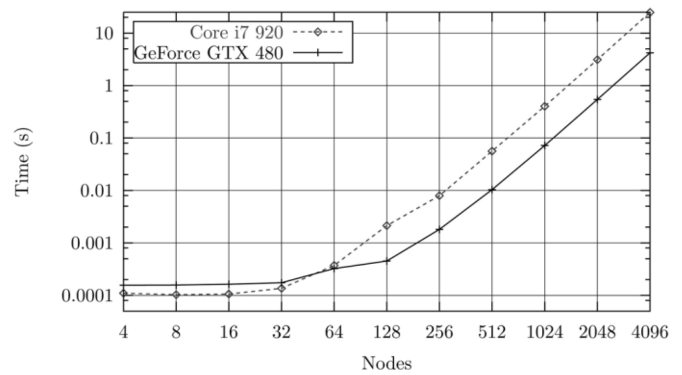

Fig. 2. Timings of pathfinding algorithms for different image sizes. Please observe the different scalings.

Autowaves, Wavefronts and MPs are iterative algorithms in contrast to the Visibility Graph and Potential Field Methods. The Visibility Graph results shown in Figure 2 shows a quadratic runtime behavior when more and more objects are considered. The algorithm is very fast, but slows down when complex polygonal obstacles with many nodes are present in the environment. The performance of the iterative algorithms can be expressed in GLUPS (Giga Lattice Updates per Second). For this the resolution is multiplied with the number of iterations of an algorithm, divided by the time taken for the whole processing. The Autowave algorithm was the slowest on the GPU. We observed a performance of only 2.2 GLUPS. This is due to the high memory-bandwidth needs of the CNN-data and the complex computations necessary, compared to them of the other algorithms. The Wavefront algorithm resides in the midrange of the lineup with a performance of 6.0 GLUPS. Here we have much less data per cell and the computation is significantly faster, so this results in a three times higher GLUPS number. The Marching Pixels algorithm is the fastest of the iterative algorithms on the GPU with a performance of 16.5 GLUPS. This high count comes from the very small amount of data (only two bits) necessary and the possibility of using a LUT for the state transitions.A lot of areas can be masked out per iteration where states of map points are constant. Only the Potential Field Method in its best case is faster because it is not an iterative approach. But the disadvantage of the Potential Field Method is its susceptibility to being stuck in local minima. For this reason, two Potential Field curves are shown in Figure 1, one with the best case, the other one with the L-shaped obstacles of the worst-case scenario: The calculation of a path out of a local minimum is expensive and has to be done in serial on the host. Because this search is not parallelizable it inflicts serious runtime penalties for worst case setups. This means in best case the algorithm is fully real time capable, with a run time of less than ten milliseconds. But in the worst case the algorithm consumes over two seconds. For that, all other algorithms (except the autowaves) are much faster.

We observed several GPU related advantages/disadvantages when we did the implementations. First of all, a massive parallel approach is possible for path planning tasks and it is ideal when there are no data-dependencies between the threads. But not parallelizable tasks, such as the gradient descent of the Potential Field Method can massively hamper the whole computation. Another important fact is, that if there are data-dependencies, the ratio of computation to memory transfers needs to be reduced where possible. If an algorithm is compute bound, then a larger memory afford can speed-up the computation. Out of this idea the state-transition of the

©2011 GSTF

Marching Pixels algorithm is speeded up with the use of a LUT, that fits into the constant memory. Finally, when there are access-patterns, that fit into the limitations of the texture-access scheme, these should be used whenever possible.

VI. CONCLUSION

In this paper, we presented an evaluation of selected parallel path planning algorithms, compared them with multi-core implementations and presented detailed information about the implementation on GPUs using NVIDIA's CUDA platform.

As the results show, there is no overall winner, but nevertheless, we could show that the GPU is an efficient coprocessor for parallel algorithms in robot systems. Depending on the given application, a graph-based algorithm like the Visibility Graph should be used if only a few objects occur. In the best case the Potential Field Method is very fast and flexible, but should only be favored when being trapped in a local minimum is not critical for the application. In contrast the Marching Pixels algorithm is a rather efficient solution which is immune to adverse environment setups. Furthermore, tight upper bounds for its runtime can be given [8], making it an ideal choice for real-time applications whose setups do not fit into the niches of graph-based or potential field approaches.

REFERENCES

[1] H. Choset et al., Principles of Robot Motion. Cambridge, Massachusetts, USA: The MIT Press, 2005.

[2] R. Schraft et al., "Man-machine-interaction and cooperation for mobile and assisting robots," in Proceedings of EIS 2004, 2004, pp. 1025–1032.

[3] W. Kubinger et al., "An embedded vision sensor for robot soccer," Lecture Notes in C

[4] Computer Science, vol. 4739/2007, pp. 1025–1032, 2007. J.-C. Latombe, Robot Motion Planning, 7th ed. Norwell, USA: Kluwer Academic Publishers, 1991.

[5] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," Int. J. Rob. Res., vol. 5, no. 1, pp. 90–98, 1986.

[6] J. Barraquand et al., "Numerical potential field techniques for robot path planning," IEEE Transactions on Systems, Man and Cybernetics, vol. 22, no. 2, pp. 224–241, 1992.

[7] A. Adamatzky et al., "Reaction-diffusion navigation robot control: From chemical to vlsi analogic processors," IEEE Trans. on Circuits and System-I, vol. 51, no. 5, pp. 926–938, 2004.

[8] M. Schmidt and D. Fey, "A parallel path planning approach based on organic computing principles," in 20th IASTED International Conf. on PDCS, 2008, pp. 176–181.

[9] R. Seidler, M. Schmidt, A. Schäfer and D. Fey, "Comparison of selected parallel path planning algorithms on gpgpus and multi-core processors," in Proceedings of the ADPC, 2010, pp. A133–A139.

[10] L. Fischer et al., "Gpu accelerated path-planning for multiagents in virtual environments," SB Games II, 2009.

[11] A. Bleiweiss, "Gpu accelerated pathfinding," in GH '08: Proceedings of the 23rd ACM SIGGRAPH / EUROGRAPHICS symposium on Graphics hardware, Aire-la-Ville, Switzerland, 2008, pp. 65–74.

[12] R. Gayle et al., "Path planning for deformable robots in complex environments," in Robotics: Systems and Science, 2005.

[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, pp. 269–271, 1959.

[14] L. O. Chua et al., "Cellular neuronal networks:theory," IEEE Trans. on Circuits and Systems, vol. 35, pp. 1257–1272, 1988.

[15] D. Fey and D. Schmidt, "Marching Pixels: A new organic computing principle for high speed cmos camera chips," in Proceeding ACM International Conference on Computing Frontiers 2005, 2005, pp. 1–9.

[16] M. Komann and D. Fey, "Realising emergent image preprocessing tasks in cellular-automaton-alike massively parallel hardware," IJPEDS, vol. 22, no. 2, pp. 79–89, 2007.

[17] R. Stefanelli and A. Rosenfeld, "Some parallel thinning algorithms for digital pictures," Journal of the ACM, vol. 18, no. 2, pp. 255–264, 1971.

[18] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," ACM Comput. Surv., vol. 23, no. 3, pp. 345–405, 1991.

[19] Animation of the marching pixel path planning algorithm. [Online]. Available: http://www3.informatik.unierlangen. de/Research/PathPlanning/

Ralf Seidler started his university studies in 2004 and received the masters-degree in computer science (Diplom Informatiker) from the Friedrich Schiller University of Jena in 2010. Currently he is a staff-member at the chair of computer architecture, department of computer science at the university of Erlangen-Nuremberg since June 2010. His main research interest is GPGPU computing for image processing tasks.

Michael Schmidt received his diploma degree in computer science from the University of Jena, Germany in 2005. From 2005 to 2009 he was a member of the research staff at the University of Jena. Since 2009 he is working at the University of Erlangen, Germany. His main research interests are embedded systems and FPGAs. He is currently working on the efficient realization of path planning algorithms for robot systems based on FPGAs.

Andreas Schäfer graduated in Computer Science at the Friedrich-Schiller-Universität Jena, Germany in 2006. He received a PhD scholarship for three years and is now a staff member of the Friedrich-Alexander-Universität Erlangen, Germany. Andreas has specialized in high performance computing. His research focuses on the flexible parallelization of stencil codes.

Prof. Dr.-Ing. Dietmar Fey studied computer science at University Erlangen-Nuremberg. The topic of his Ph.D. thesis in 1992 was about Optical Computing Architectures. From 1999 to 2001 he was researcher and lecturer at the Universities in Jena and Siegen in Germany. In 2001 he became professor for Computer Engineering at the University of Jena. Since 2009 he has the Chair of Computer Architecture at the University of Erlangen-Nuremberg, Germany. His research interests are in the areas of parallel embedded processor architectures, heterogeneous parallel architectures, Custer and Grid computing, and nanocomputing.

©2011 GSTF