Proceedings of Australasian Conference on Robotics and Automation, 2-4 Dec 2013, University of New South Wales, Sydney Australia

An Optimised GPU-Based Robot Motion Planner

Stephen Cossell and José Guivant
School of Mechanical and Manufacturing Engineering,
The University of New South Wales, Australia
{macgyver,j.guivant}@unsw.edu.au

Abstract

This paper presents a novel approach for solving Dynamic Programming optimisation problems such as the generation of exhaustive cost-to-go functions for path planning in dense environments. Cost-to-go function generation on an occupancy grid is the core of many metric-based techniques for unmanned ground and aerial vehicle path planning. The main limitations of existing methods concern a trade off between properties such as physical size of occupancy grid, resolution of grid cells and required update frequency. This paper presents a concurrent version of a traditional dynamic programming algorithm used to evaluate a cost function on grid-based domains. The proposed algorithm provides an existing system with an order of magnitude more flexibility and leniency in the aforementioned properties as it provides mathematically equivalent results to the traditional algorithm in an order of magnitude less time on currently available hardware. Although this algorithm has been developed and tested in a robotics context, it may benefit other areas of optimisation and control.

1 Introduction

A current popular technique for robot path planning is a grid-based domain approach. For example, [Whitty *et al.*, 2010] uses laser range sensor data to generate 3D images that are simultaneously used to synthesise a two-dimensional occupancy grid. This grid is then used to create an exhaustive cost-go-to function of every cell in the environment to a chosen destination cell. An optimal[1] path is then extracted from the cost-to-go function and is approximated to a nonholonomic path under the constraints of the Unmanned Ground Vehicle's (UGV's) dynamic model. Although this implementation was adequate for the applications presented, the technique would greatly benefit from both a larger and more finely grained occupancy grid for larger scale applications that need to maintain detail in crucial areas such as doorways and corridors.

This paper begins by surveying current research and techniques in the fields of grid-based path planning as well as concurrent programming on graphics hardware. The paper continues by describing the existing and new *flavours* of the algorithm as they have evolved to alleviate some of the discovered implementation bottlenecks. The existing flavours include the initial *vanilla* and expanding texture implementations, and were initially presented in [Cossell and Guivant, 2011c]. Simulated results on two different occupancy grids are presented to compare graphics card implementation support as well as comparing the different evolutionary flavours of the algorithm. Experimental results are then given on the algorithm's implemented performance on a live UGV platform. The paper concludes by giving a brief outline of current directions of the proposed research that are still under development and analysis.

2 Background

Grid-based representations of a robot's surrounding environment such as occupancy grids have been utilised in applications for decades [Elfes, 1989][Guivant *et al.*, 2004][Magni *et al.*, 2001]. As a metric-based approach they provide precise[2] measurements of the spatial layout of an environment to a given resolution and can therefore, in theory, be used to plan closer to optimal[3] paths of traversal on. The main practical limitations, however, concern the trade off between occupancy grid coverage versus granularity, resulting in the overall number of cells used to represent an area. The required refresh or update rate of the grid when new sensor data is available is also a

[1] To the resolution of the grid representation.

[2] To within known operational errors of sensor measurements.

[3] As opposed to the open space road map, random search and other approximated classes of planning algorithms.

factor [Thrun and Bücken, 1996]. The current limitation of these three properties is determined by the computational power available to generate a cost-to-go function on an occupancy grid. A current partial solution to this limitation is to use a hybrid metric and topological approach [Tomatis *et al.*, 2003]. This approach maintains smaller localised metric-based representations of areas and aligns them globally using a topological representation. A similar hybrid approach has also been used in simultaneous localisation and mapping applications that need to be efficient for real time applications and cater for map correction and efficient loop closure [Whitty and Guivant, 2009]. Although these divide and conquer implementations offer a larger number of cells to represent an environment they are still limited by availability of computational power to plan a path on each of the local metric-based grids.

Over the last two decades one of the simplest methods of gaining more computational power was to acquire the next generation of CPU available. This would usually provide a dramatic increase of an algorithm's performance without modifying the implementation itself in any way. In recent years, however, consumer CPUs have reached a physical limit in the number of calculations they can perform per second [Patterson, 2010]. Processor manufacturers have begun to design multi-core processors that allow more calculations to be carried out per second, but as many computer scientists are becoming aware, multi-threaded software is an order of magnitude more complex to design and develop than its traditional sequential counterpart [Patterson, 2010][Andrews, 1991].

While current multi-core CPUs have between two and eight processors, current consumer graphics cards have evolved over the last decade to contain hundreds to thousands of shader processors [nVidia Corporation, 2011b][Gorder, 2007]. This evolution has been driven by the computer game and interactive entertainment industry to produce increasingly realistic visual effects, but a growing group of researchers have been using the architecture for nongraphical calculations [Harris, 2005][Furukawa *et al.*, 2010], in a technique called general purpose computation on graphical processing units (GPGPU) [Harris, 2005].

Dynamic programming methods such as those used to generate optimal paths through graphs and occupancy grids are inherently sequential in nature [Dijkstra, 1959][Hart *et al.*, 1968]. That is, each individual calculation depends recursively on the previous calculation being evaluated first, back to an initial base value. Using an occupancy grid representation, many of these calculations can actually be evaluated concurrently without degradation to the final result and this property of grid based configuration space representations is exploited by the proposed algorithm. A GPGPU design pattern known as a Ping-Ponging Texture [Harris, 2005] is used to keep track of the state of the algorithm's progress. This technique uses two memory locations to represent the state of the cost-to-go function as it is being evaluated. These buffers alternate between read and write states, where the read buffer represents the current state of the cost-to-go function and the write buffer represents the next state. As each cell's kernel[4] can be run in an arbitrary order under the concurrent programming paradigm, a read-only buffer is used in this way as the basis for all calculations. This ensures that, given any arbitrary order cells are evaluated in one iteration, the state of the cost-to-go function will be in a predictable state at the end of each iteration.

3 Optimising by Subregions

The hybrid method of using subregions is presented in this section. This method tries to find a balance between minimising wasted calculations in the GPU with low CPU overhead management costs to give a more efficient implementation than using the expanding texture method [Cossell and Guivant, 2011c]. Figure 1 shows the theoretical number of individual cell evaluations requested for execution on the GPU for the *campus emergency help point* test case outlined in Section 4.1. It highlights that the proposed method demands an order of magnitude less burden on the graphics hardware for mathematically identical results.

3.1 Definition

A subregion is defined as a square[5] grouping of cells within an occupancy grid. Each cell in a subregion is evaluated on the GPU together and managed via the CPU as a group. This approach tries to balance targeting cells that can be usefully evaluated on the GPU while minimising the number of "objects" to manage using the CPU. Each subregion can be in one of three states, as outlined below.

Unclassified A subregion where cells have not yet been classified with a value, nor are any of the cells currently being requested for evaluation by the GPU.

Maturing A subregion that is currently requesting all of its cells to be evaluated on the GPU. Cells within a maturing subregion may or may not have a value at a particular iteration, but are being put through the evaluation kernel program regardless of whether the calculation is useful or not.

[4] A kernel is a small program executed by each shader processor on the GPU for each cell submitted for evaluation.

[5] A subregion may be rectangular at the edges of a texture where the occupancy grid dimension is not divisible by the subregion dimension.

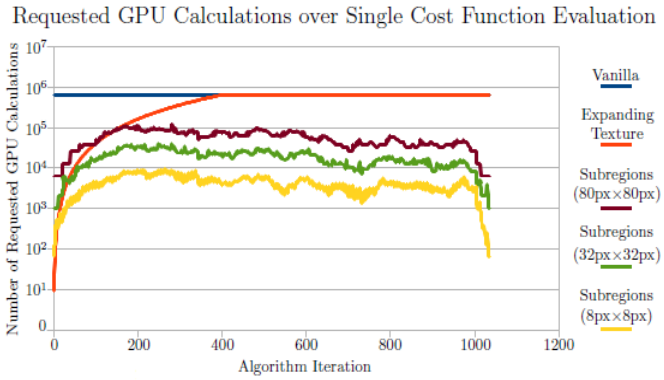


Figure 1: Logarithmically scaled graph of the number of cell evaluations requested for calculation on the GPU for different flavours of the algorithm. It should be noted that the *subregion* graphs are not an accurate measure of real world execution time as they do not include the CPU overhead of managing the subregions as discussed in the following sections. The comparison should highlight the fact that the GPU is requested to perform a number of orders of magnitude less work using subregions over the *expanding texture* and *vanilla* flavours of the algorithm.

Mature A subregion where cells have been classified and none of the cells in the subregion are being requested for evaluation on the GPU.

In short, maturing subregions are considered active and are requested for evaluation on the graphics card, while unclassified and mature subregions are inactive. A simulation of the algorithm running with maturing subregions specifically highlighted can be found at [Cossell and Guivant, 2011b].

3.2 Subregion Management Algorithm

Every iteration an active subregion is read back from the GPU memory into CPU memory to determine whether the subregion should be promoted to a *mature*, and therefore inactive, state. The assessment process comprises two forms of analysis using data from the current iteration with data from the previous iteration. The first of these iteratively checks each of the cells in the borders of the subregion. If a difference is detected between the cell's current value and its previous value then the subregion adjacent to the current subregion's border, against the wall containing the differing value cell, is added to the active list if it is not already there. This step is referred to as the *Unclassified to Maturing check* in later sections. At the beginning of the algorithm, the subregion containing the destination cell is the only subregion in the active list[6], so this is the mechanism by which subregions are gradually added to the list for evaluation.

The second form of analysis involves checking the entire subregion to detect any changes between the previous iteration and the current iteration's values for the entire subregion. If there is no change in the whole subregion then it is removed from the active list and is classified as a *mature* subregion. This step is referred to as the *Maturing to Mature check* in later sections. For efficiency, this check uses the results of the border checks mentioned above to rule out any single cell differences before continuing on to check all the nonborder cells in a subregion. This is the mechanism that removes subregions from the active list. As a result, when the active list is first detected to be completely empty then the algorithm is determined to be finished.

3.3 Subregion Management Data Structures

For efficiency, subregions are stored in two interdependent data structures. At initialisation each subregion is stored in a two-dimensional array in a manner matching their spatial layout on the occupancy grid. This data structure was chosen so that neighbouring subregions can be referenced and accessed efficiently by array index arithmetic. The active subregion list is implemented using a traditional C-style linked list with new entries added to the front of the list to minimise list traversal operations. A linked list is used as it provides an efficient and dynamic method for adding elements to a one-dimensional list type data structure as each subregion in the active list needs to be requested for evaluation on the GPU in turn.

3.4 Subregion size

As mentioned above, the main aim of breaking an occupancy grid into subregions is to find an optimal balance between minimising wasted GPU calculations and minimising CPU overhead in managing which particular cells should be requested for evaluation. One can then imagine the two extremes of having subregions containing just one cell against having the whole occupancy grid as one subregion. In practice, both of these extremes are inefficient and a balance between the two yields the most efficient results. Detailed optimal subregion sizes are outlined in Section 4 but for most environments a subregion size of 32×32 is most efficient for current low end graphics cards and 128×128 for current high end graphics cards.

3.5 Breakdown of Processing Time

An analysis of the breakdown of execution time spent on different components of the algorithm revealed that the reading of data back from GPU memory to CPU memory

[6] Alternatively, if there are many destination cells then each of their subregions is added to the active list at the beginning of execution.

Component	Avg. Time	%
`glReadPixels()`	3.846 s	97.19%
Unclassified to Maturing check	0.007 s	0.18%
Maturing to Mature check	0.006 s	0.14%
Other	0.098 s	2.48%

Table 1: Breakdown of execution time of different parts of the algorithm for a subregion size of 80×80.

was the largest burden on the system. The magnitude of this burden is shown in Table 1 for subregion size $s = 80$. The component labeled "`glReadPixels()`" is a single OpenGL function call used to read pixels from GPU memory to an array in CPU accessible memory. The CPU and GPU calculations are done side by side and evidence later in this section suggests that such pre-transfer computations contribute a noticeable, but minimal contribution to the measured execution time of this particular function call.

As an exploratory step, the algorithm was modified so that the active list membership checks were performed every second iteration in an attempt to reduce overall execution time. This saw the total execution time dedicated to "`glReadPixels()`" drop from 3.846 s to 2.283 s, with the overall execution time dropping to 60% of its original value. Even with this reduction, the proportion of time dedicated to executing "`glReadPixels()`" was still measured to be 96.15% of the total execution time.

A possible next step would logically be to perform the check every n-th iteration for $n \geq 3$. This method reaches an asymptotic best case, whereas the following section discusses a more directed solution to this data transfer bottleneck problem as well as other methods of optimisation.

3.6 Active list membership optimisations

As stated in the previous section, the reading back of data from GPU memory to CPU memory for checking membership of a subregion in the active list, is the most costly step of the algorithm. A number of optimisations have been implemented to reduce this bottleneck. The first method is to cache data read from one iteration so it can be used on the next iteration. That is, when both adding and removing subregions from the active list, data from the current and previous iteration are compared. Storing data in a pair of buffers on CPU memory using a double buffering method reduces total execution time of a single complete evaluation of cost-to-go function by 60% on average. Even though this method significantly reduces the execution time of the algorithm, reading data from GPU to CPU memory is still a large proportion of execution time.

Statistical analysis of how many iterations a subregion

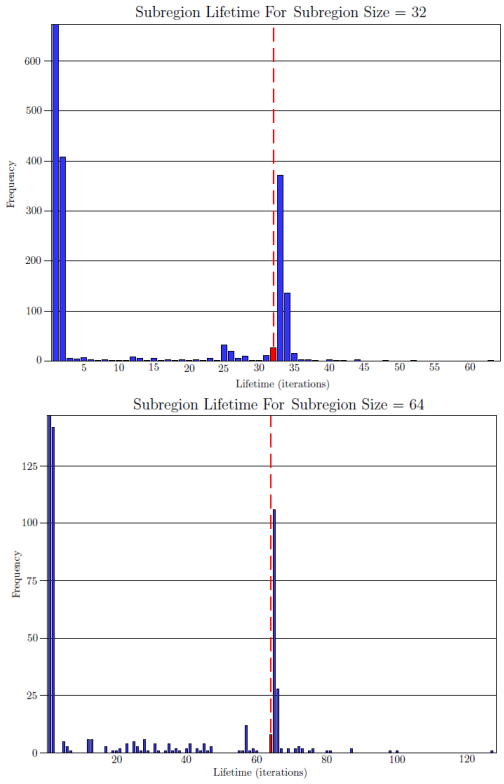


Figure 2: Histograms of the number of iterations a subregion is active or maturing for, for different sized subregions. The lifetime equal to the subregion size is highlighted by a vertical red dashed line.

is active for revealed that, although subregion lifetimes exist for a range of iterations, there are statistically significant "spikes" in the number of low values such as those less than three as well as a similar "spike" with gradual decay after the number of iterations equal to the subregion size used. Figure 2 shows a frequency histogram of the subregion active lifetimes for subregion sizes of 32×32 and 64×64 for the *campus emergency help points* simulated case outlined in more detail in Section 4.1. Other occupancy grids used during simulated and experimental testing gave similar results.

Using the gathered statistical results, the following schedule of checking the status of a subregion membership in the active list was then employed. For a subregion size s and individual subregion iteration counter i, checks were performed when:

- $i = 1$ and 2,
- $i = \frac{s}{2}$ and $\frac{s}{2} + 1$,
- $i = s + 2n$ for $n \in \mathbb{N}$.

Using this schedule to perform the active list membership checks gave the shortest execution times, as outlined in the following section.

Property	Desktop
Name	nVidia GeForce GTX 480
Shader processors	448
Peak FLOPs	1344.96 GigaFLOPs
Bus transfer	8GB/s (PCIe 2.0 x16)
References	[nVidia Corporation, 2011a] [nVidia Corporation, 2010]

Table 2: Outline of graphics hardware used in the following simulations. FLOP stands for floating point operations per second.

4 Simulated analysis and benchmarking

This section provides the performance characteristics of a set of simulated and real world examples to show how the algorithm performs in certain conditions. As it is difficult to measure real world running time of mixed processor implementations, the execution times mentioned below are based on the following formula:

$$T = \frac{E}{100} = \frac{s_1 + 100 \times A + s_2}{100} \quad (1)$$

where T is the estimated execution time of the algorithm itself for one complete cost-to-go function generation, E is the real world (clock) execution time of the program running the simulation. Because the program requires some setup and shutdown time, the core calculation itself, A, is repeated 100 times to attempt to dilute the setup and shutdown times, s_1 and s_2 respectively.

Each simulation was run on a near forefront Desktop based gaming graphics card, part of which forms the basis for the Ground Control Station (GCS) used in [Whitty *et al.*, 2010] and used for experimental results given in Section 5. The relevant specifications for this graphics card are given in Table 2.

4.1 Campus emergency help point

The authors' campus has a number of emergency help points where staff and students can contact security via radio terminals. A map of campus was used as the occupancy grid for this experiment, with one of the help points set as the destination point for the algorithm. The final cost-to-go function is given in Figure 3 with algorithm performance measured against differing subregion sizes seen in Figure 5.

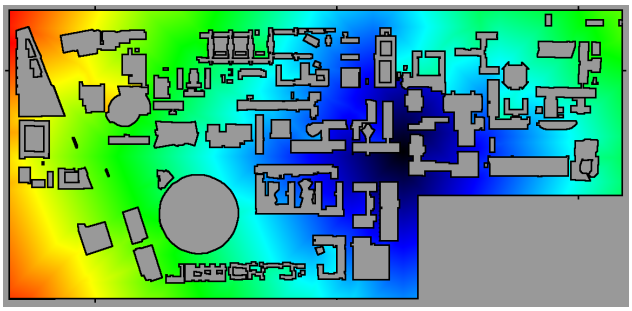


Figure 3: The cost-to-go function for the *Campus emergency help point* test case. The global cost is shown ranging from indigo at the zero cost destination cell to red at the highest cost. Obstacle cells are represented by black and unreachable cells are represented by gray.

4.2 Multiple destination points: campus help points

As the algorithm strongly benefits from a high level of concurrency, a second set of tests were run on the campus help point occupancy grid. A destination point was assigned to each of the 13 help points and the algorithm was run with a wavefront emanating from each of the destination points concurrently. The occupancy grid consists of 1158px ×559px and the cost-to-go function generated on this grid can be seen in Figure 4. The performance of this algorithm can be seen in Figure 6. A simulation of the algorithm running with multiple destinations points can be found at [Cossell and Guivant, 2011a].

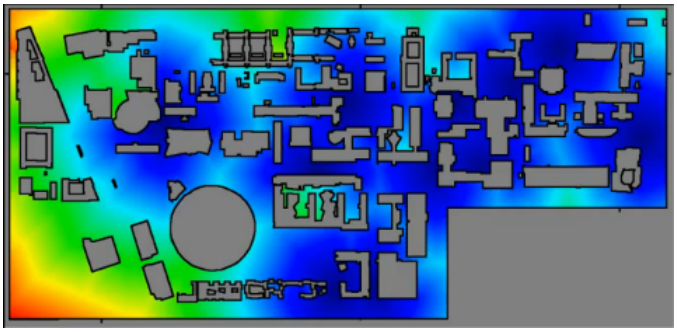


Figure 4: The cost-to-go function for the *campus help point* test case with multiple destination points. The global cost is shown ranging from indigo at the zero cost destination cell to red at the highest cost. Obstacle cells are represented by black and unreachable cells are represented by gray.

4.3 Comparison of each flavour of the algorithm

For the single campus emergency help point (Section 4.1) and multiple help point (Section 4.2) examples, benchmarking was carried out on each evolutionary flavour of the algorithm. For each graph given in this section, the vanilla and expanding texture (*exp.tex.*) execution times are given as a reference point and are based on those given in [Cossell and Guivant, 2011c]. The horizontal axis of each graph uses a range of subregion sizes to show the implemented algorithm's performance over a range of values. As the vanilla and expanding texture flavours do not involve subregions, their execution times are given as a constant horizontal line on each graph.

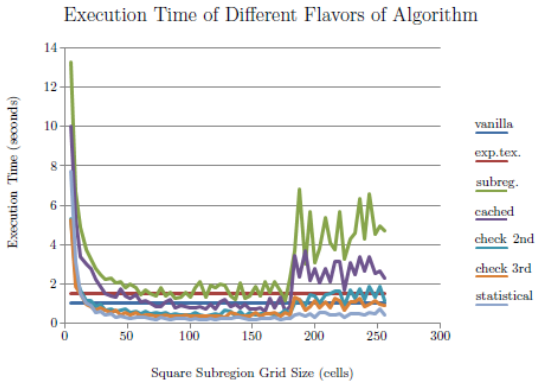
In each graph, *subreg.* designates the basic subregion implementation with no optimisations. The *cached* label represents the next evolution of the algorithm where subregion values read back to the CPU's memory are cached for later comparison in the active list check process. The labels *check 2nd* and *check 3rd* represent the cached subregion data flavour where the active list check is done every second and third iteration, respectively. Finally the *statistical* label designates the cached subregion data method where active subregion list membership checks are carried out on the schedule specified in Section 3.6.

Figures 5 and 6 show execution times for the single and multiple help points occupancy grids, respectively. The desktop graphics hardware performed the raw GPU calculations steps in shorter time and did not appear to be as hindered by a large number of redundant GPU calculations being requested. This is clearly demonstrated by the vanilla and expanding texture execution times being in the same order of magnitude as the various subregion flavour execution times. However, given that in practice a certain subregion size can be chosen to fit the application's requirements, the statistically scheduled CPU memory cached implementation of the subregions algorithm still out performs all other previous methods.

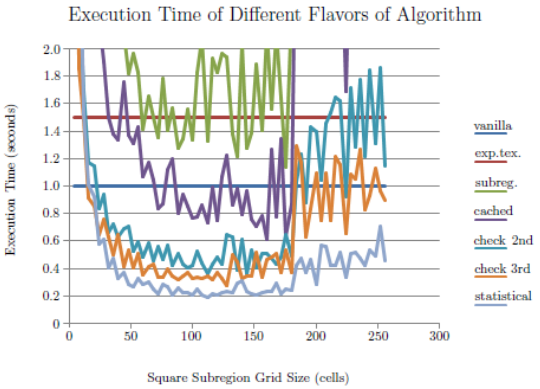
Table 3 summarises the fastest execution times of each of the flavours of the algorithm over the single and multi-destination help point occupancy grids.

5 Experimental results

The algorithm was implemented for interfacing with an existing autonomous ground vehicle system with technical specifications outlined in [Whitty *et al.*, 2010]. The existing module reads in a live occupancy grid from a centralised real-time round robin database system [Guivant, 2012], along with the pose and destination location. A cost-to-go function is calculated based on the occupancy grid and destination, after which a path is planned based on the vehicle's position relative to the desired destination location. This high level path is then pushed back into the centralised database for another module to



(a) Complete comparison.

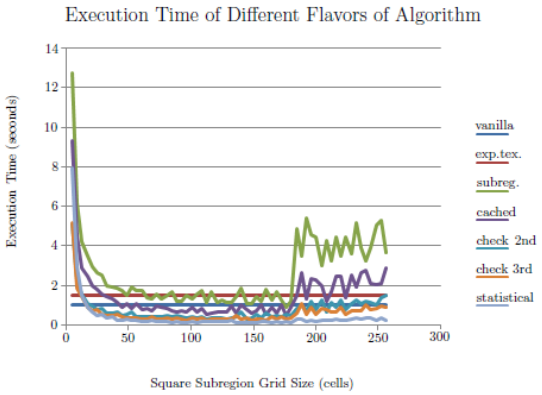


(b) Comparison highlighting the 0 to 2 second range.

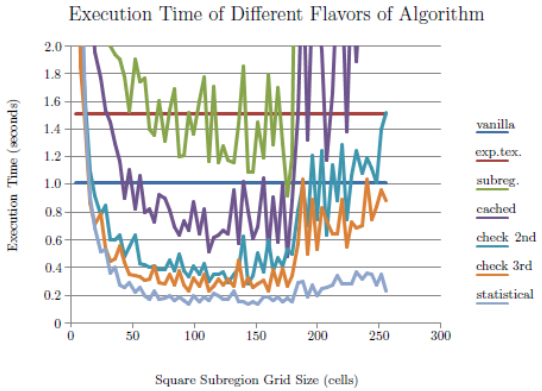
Figure 5: The execution time of the algorithm implemented in different evolutionary flavours. The results are from the desktop machine running the single destination help point occupancy grid. Graph (b) shows identical results to the graph (a), only concentrated on the 0 to 2 second range of execution time to show greater detail.

Occupancy Grid	Sgl. Dest.	Multi-Dest.
Vanilla	0.999s	1.013s
Expanding Texture	1.496s	1.5063s
Subregion	1.1357s	0.9156s
Cached	0.6128s	0.4789s
Check 2nd	0.3591s	0.2841s
Check 3rd	0.2765s	0.2283s
Statistical	0.1872s	0.1323s

Table 3: A comparison of the fastest execution times for each of the implemented algorithm flavours. *Sgl. Dest.* labels results from the single destination campus help point example. *Multi-Dest.* labels results from the multiple destination campus help points example. Times given in this table are in seconds.



(a) Complete comparison.



(b) Comparison highlighting the 0 to 2 second range.

Figure 6: The execution time of the algorithm implemented in different evolutionary flavours. The results are from the desktop machine running the multiple destination help points occupancy grid. Graph (b) shows identical results to the graph (a), only concentrated on the 0 to 2 second range of execution time to show greater detail.

interpret the planned path into actual motor commands onboard.

The proposed algorithm was implemented in the form of a module that has the same application programming interface with the centralised database as the existing module. The module was run on the GCS to aid the user in monitoring the UGV's motion remotely and as a result the planner used a subregion size of 128×128, for reasons stated in Section 3.4. From a black-box point of view, the traditional sequential implementation provided mathematically identical results to the proposed concurrent version of the module. Figure 7 shows the instantaneous cost-to-go function generated on a live occupancy grid during a outdoor campus experiment.

6 Future and related work

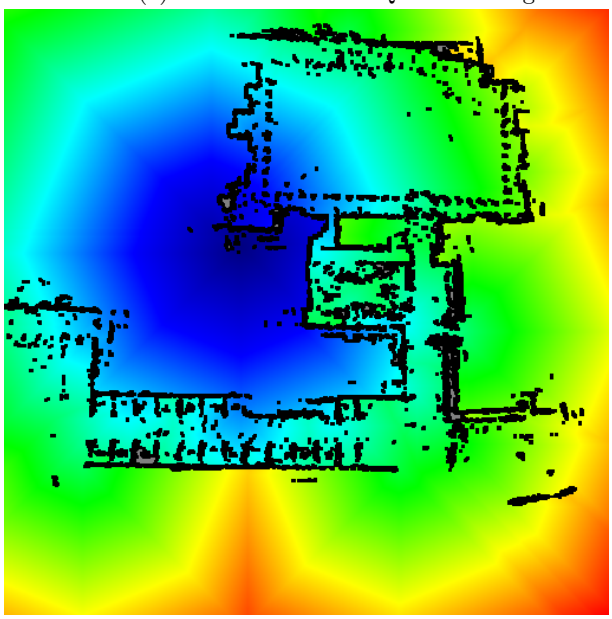
A more in depth analysis of the proposed algorithm and implementation have been submitted for journal review [Cossell and Guivant, 2013]. This work involves analysis over a wider range of occupancy grid types, multiple generations of graphics card architecture and provides greater analysis of the effect the subregion size has on certain metrics used to measure the performance of this approach.

Additional research is being finalised on extending this 2D implementation to 3D configuration space representations. Theoretical predictions of the concurrent benefits of such an approach are even more favourable than the 2D implementation presented here. Traditionally, as the dimensionality of the configuration space increases, the computational burden on path planning in such a domain increases exponentially. This burden is partially but significantly negated by the fact that a higher dimensionality in the configuration space allows the proposed algorithm to expand into more cells within a given initial number of iterations. Initial experiments have also been conducted into implementing the algorithm in configuration spaces in $\mathbb{R}^d$ for $d > 3$, specifically targeted at applications in high degree-of-freedom robot manipulators.

7 Conclusion

A concurrent grid-based cost-to-go function generator has been presented that relies on the emerging and improving technology found on modern graphics hardware. The evolutionary steps of the development of the implemented algorithm have been presented showing bottlenecks faced by real world use cases and subsequent techniques developed to minimise or mitigate those bottlenecks. Under current GPU and CPU hardware characteristics, the proposed algorithm performs on par with existing forefront CPU-based algorithms for most real world cases, while it excels in configuration spaces with

(a) Aerial view of the Quad Building.



(b) Generated cost-to-go function.

Figure 7: A ©2013 Google Maps overview of a section of campus along side the cost function generated on a live occupancy grid of the same region.

wide open areas. As CPU clock cycle rates have begun to reach their physical limit, GPU hardware continues to increase in terms of individual processor clock cycle rate, but more significantly in the number of processors available — a particularly favourable property for the proposed algorithm. The algorithm has been proven to be at least an order of magnitude more efficient from a mathematical point of view and as graphics hardware capabilities continue to improve this theoretical advantage will become more evident in practical applications.

Acknowledgments

The authors would like to thank Mark Whitty and Jayantha Katupitiya, colleagues in the development of the UGV platform, for this algorithm to be tested in a practical context.

References

[Andrews, 1991] G. R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, 1991.

[Cossell and Guivant, 2011a] Stephen Cossell and José Guivant. GPGPU cost function generation with multiple destination points. `https://www.youtube.com/watch?v=xeqOCPCZU_Q`, May 2011.

[Cossell and Guivant, 2011b] Stephen Cossell and José Guivant. GPGPU cost function generation with subregion optimisation. `https://www.youtube.com/watch?v=yk-pnRXu1aU`, July 2011.

[Cossell and Guivant, 2011c] Stephen Cossell and José Guivant. Parallel evaluation of a spatial traversability cost function on gpu for efficient path planning. *Journal of Intelligent Learning Systems and Applications*, 3(4):191–200, November 2011.

[Cossell and Guivant, 2013] Stephen Cossell and José Guivant. Concurrent dynamic programming for grid-based problems and its application for real-time path planning. *Robotics and Autonomous Systems (under review revision)*, 2013.

[Dijkstra, 1959] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[Elfes, 1989] Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, June 1989.

[Furukawa *et al.*, 2010] Tomonari Furukawa, Banjamin Lavis, and Hugh F. Durrant-Whyte. Parallel grid-based recursive Bayesian estimation using GPU for real-time autonomous navigation. In *International Conference on Robotics and Automation*, Anchorage, Alaska, May 2010.

[Gorder, 2007] P. F Gorder. Multicore processors for science and engineering. *Computing in Science and Engineering*, 9(2):3–7, 2007.

[Guivant *et al.*, 2004] José Guivant, Eduardo Nebot, Juan Neito, and Favio Masson. Navigation and mapping in large unstructured environments. *International Journal of Robotics Research*, 23(4-5):449–472, April-May 2004.

[Guivant, 2012] José Guivant. Possum robot. `http://www.possumrobot.com`, 2012.

[Harris, 2005] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH*, Los Angeles, California, USA, July 2005.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[Magni *et al.*, 2001] L. Magni, G. D. Nicolao, L. Magnani, and R. Scattolini. A stabilizing model-based predictive control algorithm for nonlinear systems. *Automatica*, 37(9):1351–1362, 2001.

[nVidia Corporation, 2010] nVidia Corporation. New nvidia geforce gtx 480 gpu cranks up pc gaming to new heights, March 2010.

[nVidia Corporation, 2011a] nVidia Corporation. GeForce GTX 480. `http://www.nvidia.com/object/product_geforce_gtx_480_us.html`, 2011.

[nVidia Corporation, 2011b] nVidia Corporation. High performance computing - supercomputing with tesla GPUs. `http://www.nvidia.com/object/tesla_computing_solutions.html`, 2011.

[Patterson, 2010] David Patterson. The trouble with multicore. *IEEE Spectrum*, July 2010. http://spectrum.ieee.org/computing/software/the-trouble-with-multicore.

[Thrun and Bücken, 1996] Sebastian Thrun and Arno Bücken. Integrating grid-based and topological maps for mobile robot navigation. In *Proceedings of the Thirteenth National Conference on Artificial Intellegence*, Portland, Oregon, USA, August 1996.

[Tomatis *et al.*, 2003] Nicola Tomatis, Illah Nourbakhsh, and Roland Siegwart. Hybrid simultaneous localization and map building: a natural integration of topological and metric. *Robotics and Autonomous Systems*, 44(1):3–14, February 2003.

[Whitty and Guivant, 2009] Mark A. Whitty and Joseé E. Guivant. Efficient path planning in deformable maps. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, St. Louis, USA, October 2009.

[Whitty *et al.*, 2010] Mark Whitty, Stephen Cossell, Kim Son Dang, Jose Guivant, and Jayantha Katupitiya. Autonomous navigation using a real-time 3d point cloud. In *Australasian Conference of Robotics and Automation*, Brisbane, Australia, December 2010.