Mobile GPGPU Acceleration of Embodied Robot Simulation

Simon Jones[1,2], Matthew Studley[1], and Alan Winfield[1]

[1] Bristol Robotics Laboratory, University of the West of England, Bristol, UK
simon.jones@brl.ac.uk, matthew2.studley@uwe.ac.uk, alan.winfield@uwe.ac.uk
[2] University of Bristol, Bristol, UK

Abstract. It is desirable for a robot to be able to run on-board simulations of itself in a model of the world to evaluate action consequences and test new controller solutions, but simulation is computationally expensive. Modern mobile System-on-Chip devices have high performance at low power consumption levels and now incorporate powerful graphics processing units, making them good potential candidates to host on-board simulations. We use the parallel language OpenCL on two such devices to accelerate the widely-used Stage robot simulator and demonstrate both higher simulation speed and lower energy use on a multi-robot benchmark. To the best of our knowledge, this is the first time that GPGPU on mobile devices have been used to accelerate robot simulation, and moves towards providing an autonomous robot with an embodied *what-if* capability.

1 Introduction

The capability of an autonomous robot to perform on-board simulations of reality is desirable for a number of reasons.

In the area of swarm robotics [20] the design of controllers to produce a desired emergent collective behaviour is notoriously hard. Some successful approaches use an evolutionary algorithm where controller solutions are evolved off-line in repeated simulations of a swarm of robots prior to implementation in real robots but the resultant controller is not adaptive to changing environmental conditions. It is possible to have communication links between robots and off-board simulations to give adapability but at the cost of autonomy. One approach to provide both adaptability and autonomy is to move the evolutionary algorithm and simulation onto the robots so that controllers can be evolved in response to the environment. O'Dowd et al in [15] and [16] describe work in this area.

An on-board simulation might also be used to equip a robot with a 'functional imagination' [12] allowing a robot to evaluate courses of action or strategies in the safety of simulation, rather than in the real world where it may have potentially catastrophic consequences. Recent work by Winfield et al [25] extends this to provide a robot with a form of 'ethical' action selection, where a robot has an internal model which it can use to make predictions about the consequences

of both its own and others actions through simulation of multiple scenarios and even act to prevent danger to another robot. Currently this capability is dependent on a wifi link to a laptop due to the lack of sufficient on-board processing power. Clearly, where this *what-if* capability is safety critical or inherent in the behaviour of the robot, as in the 'ethical' robot above, it would not be possible to use an unreliable communications link and embodied simulation would be essential.

In both cases, the performance of the on-board simulation is critical in two ways. Firstly, simulation speed. Faster simulations allow larger numbers of robots, more scenarios, and longer simulated times within a given real time. Secondly, energy usage. Energy is a precious resource in a mobile robot and minimising the energy cost of performing a given simulation is an important goal.

Over the last decade, the performance of desktop PC graphics processors (GPU) in GFLOPS has outstripped that of CPUs and the emergence of parallel programming APIs such as CUDA [14], and more recently OpenCL [8], have made General Purpose Programming on the Graphics Processor (GPGPU) more accessible. GPGPU techniques are now widely used in scientific computing. This trend on the desktop is being mirrored on mobile platforms but within a far more restrictive power envelope; current mobile devices are as powerful as the desktop of around ten years ago but with power consumption at least an order of magnitude lower.[3]

Performing computation on a GPU is generally more energy efficient at a given performance level than performing the same computation on a CPU, provided the problem can be expressed in a suitably parallel way, because the CPU has to devote large amounts of silicon area to structures designed to extract instruction level parallelism while preserving the illusion of the semantically serial instruction stream, and will also generally run at a higher clock frequency. The GPU, on the other hand, is explicitly parallel and a much larger proportion of the silicon area can be devoted to performing computation rather than control. The design goal is massively multi-threaded throughput rather than single thread performance. See Keckler et al [7] for a good discussion of these trends.

Stage is a widely used 2D robot sensorimotor simulator that is capable of simulating large populations of robots. Vaughan [21] introduces version 3 of Stage and examines its performance scalability, demonstrating near-linear execution time scaling with populations up to 100000 robots when each robot is running an identical simple controller. Vaughan also proposes a method of benchmarking the performance.

It is clear that accelerating Stage using GPGPU techniques could have wide applicability, both on and off robotic platforms. In this paper we present a method to apply OpenCL acceleration to the central time-consuming functionality of Stage without requiring a major re-write. We then evaluate its performance on the Samsung Exynos 5250 and 5420 SoCs, both mobile GPGPU capable de-

[3] Nvidia 6800 Ultra 40 GFLOPS, 100 W, Pentium 4 7 GFLOPS [11]. Chromebook with Samsung Exynos 5250 72 GFLOPS GPU, 27 GFLOPS CPU, $<$7 W

vices, demonstrating a speed increase of 82% and a drop in energy usage of 30% for some benchmarks, compared to the unaccelerated software on the same platform.

2 Previous Work

The scalability of Stage is measured and discussed in Vaughan [21], along with a good overview and some discussion of the internal structure and design choices. Piniciroli et al [19] describe a different robot simulator and also measure its performance using a similar methodology to that described by Vaughan.

An early demonstration of the use of evolutionary algorithms to design swarm robot controllers is given by Dorigo et al in [3] where controllers for two different collective tasks are evolved within a simplified simulation which are then tested within a high fidelity physics-based simulation. Hauert et al [5] tackle the problem of adaptability of evolved controllers by reverse engineering and parameterising them. O'Dowd et el in [15] and [16] move towards providing robustness to environmental change by using a distributed evolutionary algorithm on board a swarm of e-puck [13] robots, with simple reality simulations running on the Linux Extension Board [9]. This allows the co-evolution both of the simulated environment and the swarm controllers.

Bongard et al in [1] use a process of continuous self modelling to give a robot the ability to autonomously detect and compensate for damage. Vaughan and Zuluaga [22] introduce the use of self simulation to provide a form of imagination, whereby a robot can safely evaluate different courses of action in simulation before applying them in the real world. This is taken further by Winfield et al in [25] who describe using simulation to give a robot the ability to predict the consequence of both its own and others actions and then using this to provide an 'ethical' action selection mechanism.

Ohkura et al [17] demonstrate performance benefits from using CUDA on a desktop GPU to accelerate the evolution of a swarm robotics controller for a food-foraging problem. Wang et al [23] [24] and Kang et al [6] both demonstrate performance benefits through the use of OpenCL on mobile devices to accelerate image processing algorithms. Maghazeh et al [10] investigate the performance and energy efficiency of five different non-graphic benchmarks implemented in OpenCL on a mobile device, showing benefits with most but noting the need to consider different optimisation strategies compared to desktop GPUs. Grasso et al [4] evaluate the ARM Mali GPU of the Exynos 5250 SoC for energy efficient HPC usage, porting a number of benchmarks to OpenCL and demonstrating average speedups of 8.7x and energy consumption of only 32% compared to an ARM A15 CPU core.

3 Accelerating Stage

We briefly discuss the internals of Stage, particularly the ray tracing operation that is the most time consuming operation and outline how we used OpenCL to

accelerate this functionality. Our goal was to make as few changes to the code of Stage as possible because we wished to minimise both development risk and time, and demonstrate a proof-of-concept rather than an optimised solution[4].

3.1 Overview of Stage internals

Stage is a mature, well optimised piece of software, written in C++. All entities within the simulated world are based on the *Model* class and its derivatives, which include things like *ModelPosition* two-wheeled motion kinematics and *ModelRanger* range sensors. Each instance of *Model* can have physical characteristics such as geometry within the world, represented as *blocks*, which are polygons in the XY plane with Z extents ('two and a half D'). The space of the world is a discrete grid, and the presence of geometry within the world grid is represented internally with a sparse data structure.

The *ModelRanger* derivative class implements range sensing and is used to model sensors such as laser range finders and ultrasonic sensors. The process of modelling range sensing is implemented by performing a ray tracing operation using Cohen's algorithm [2] through the sparse occupancy grid from the location of the sensor. At every grid location, each block at that location is checked to see if it has Z extents that cover the Z position of the sensor, and then a predicate function is invoked on the block to ensure it both doesn't belong to the model the sensor belongs to, and is visible to the sensor. Other *Model* derivative classes such as *ModelBlobfinder* define this predicate function differently.

This ray tracing operation is the most time consuming part of the simulation, typically taking upwards of 90% of the execution time.

The sparse data structure representing the world grid divides the space into 32x32 squares of cells called *regions*, and 32x32 squares of regions called *superregions*. Only regions and superregions which actually contain geometry are represented, which saves memory and allows the ray tracing function to skip over known empty parts of the world.

Every simulation timestep, the following simplified sequence takes place: Firstly, all the *ModelPosition* models have their geometry moved within the world grid, being removed from old locations and redrawn into their new locations. Then all the *ModelRanger* models perform ray tracing through the world grid to create sensor data. Finally, all the robot controllers are updated. This sequence repeats until the end of the simulation.

3.2 OpenCL Acceleration Strategy

As illustrated in Fig. 1, the process of ray tracing involves checking every location within the world grid along the path of a ray from the sensor to the limit of the sensor range or until there is an intersection with an object. The sparse nature of the data structure means that known empty regions of the world can be skipped over, but checking for the presence of geometry dominates execution time.

[4] The modified source code is available at https://bitbucket.org/siteks/stage_opencl

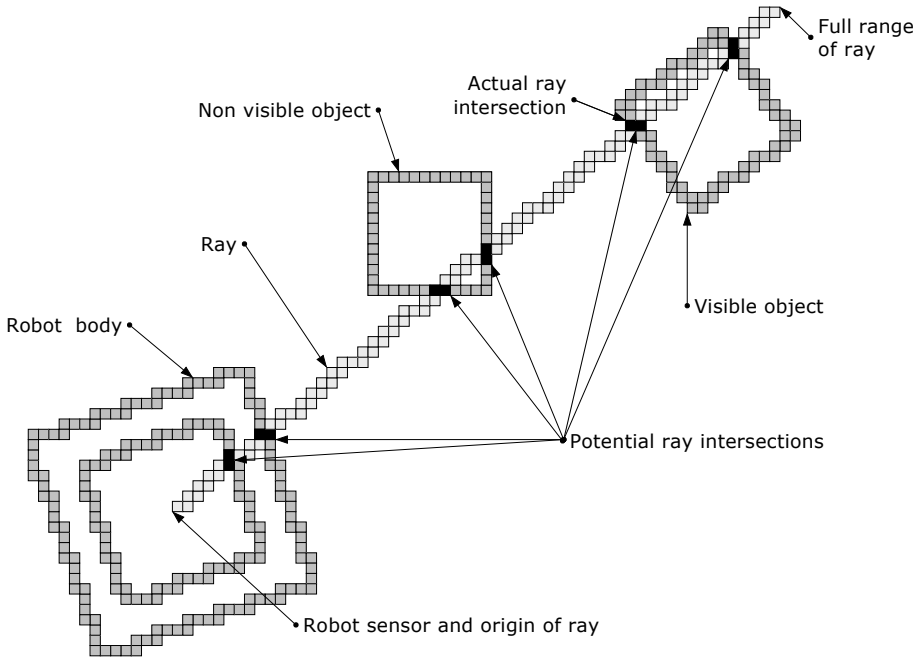

Fig. 1. Ray tracing process. In order to model a sensor, each cell of the world grid along the path of the ray is visited in turn to check if there is anything at that location. There may be many potential ray intersections before an actual intersection that corresponds to the sensor detecting an object. Objects within the world are shown as dark grey, the path of the ray as light grey, and potential intersections as black cells. The first four intersections are with the robots own geometry, which is not regarded as a hit, then there are four more with a non-visible object, perhaps because its Z position is below that of the sensor. Finally there is an actual intersection, at this point the ray trace function would normally terminate.

Each ray is completely independent, except for traversing the same world data structure, making ray tracing a parallel problem well suited to running on a GPU. The problem with using OpenCL to accelerate ray tracing in Stage is the use of an arbitrary predicate function for testing whether an occupied grid cell on the ray path is actually an intersection. An OpenCL kernel exists in a different memory space and has no knowledge of the data structures of the host, and no way to easily interpret them even if they were made available[5]. Making the intersection test a fixed function would radically and unacceptably change the behaviour and flexibility of Stage, keeping the functionality while performing all ray tracing on the GPU would require a major rewrite.

[5] The data structures are composed of C++ classes, while OpenCL is based on C99.

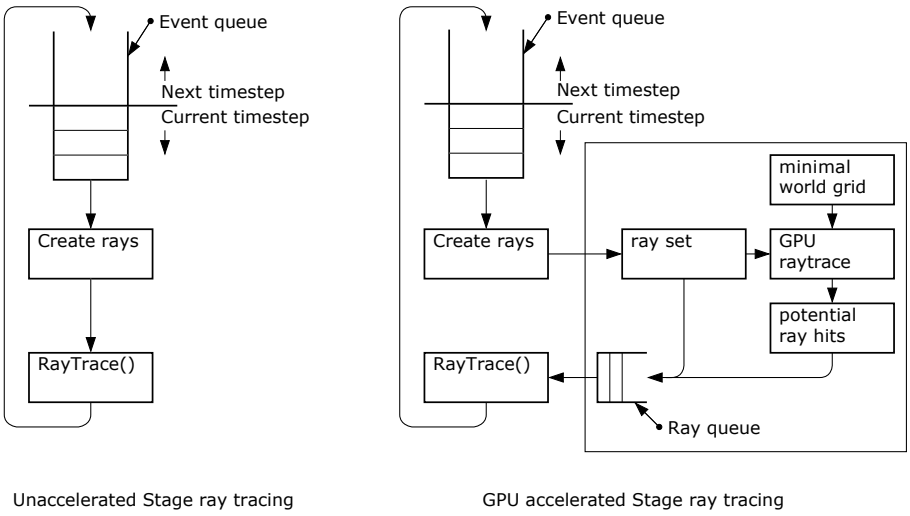

Fig. 2. Original and accelerated ray tracing data flow, box surrounds newly added functionality. Normally, each event associated with a sensor model is pulled off the queue, its rays created and then immediately fed to the RayTrace() method to create sensor data before being returned to the queue for the next timestep. In the accelerated version, all the rays are first created to make a complete set. These are then traced on the GPU using a minimal representation of the world occupancy grid to generate a set of potential ray intersections. The created rays are then fed to the RayTrace() method in the expected order but with the additional information allowing empty cells to be skipped.

The solution we chose was to perform the parallel ray tracing on the GPU using a minimal version of the world grid data structure, and create a list of *potential* intersections for each ray. This information is then be used by the normal Stage ray trace function to skip over all cells now known not to contain any geometry and only apply the predicate test to occupied cells. Complete functionality is preserved.

This is illustrated in Fig. 2. At each timestep, two data structures representing all the rays and a minimal world occupancy grid are prepared and made available to the GPU. The OpenCL kernel version of the ray trace algorithm is invoked on this data and runs in parallel across all the rays to the extent that the hardware allows, generating the potential ray hits data structure. This, and the rays, are fed back to the RayTrace() method, enhanced to allow it to skip over the cells now known to be empty.

There is obviously a certain amount of additional processing overhead that didn't exist before; preparing the world grid and ray set, making buffers available to the GPU, and bringing the potential intersection data back again. In a desktop GPU, the overhead is exacerbated by the need to copy data to the

distinct memory of the GPU, but mobile SoCs typically have a unified memory architecture. In addition, each individual ray tracing thread of execution on the GPU will be much slower than on the CPU, we gain only when there are enough rays to trace in parallel. What might that number be? The ARM Mali T604 GPU has four cores, each with 256 threads, so we expect that we will need thousands of rays to show performance gains.

4 Testing Methodology

In order to evaluate the effectiveness of the acceleration of Stage, we propose two figures of merit and a series of benchmark scenarios. We then run the benchmarks on the target systems, measuring the power consumption and run times for both normal and accelerated versions of Stage.

4.1 Figures of Merit

Since we are interested in both the speed and the energy cost of simulation, as well as the scalability of the our acceleration with numbers of robots, we use two figures of merit. The first, r_{ACC}, is a measure of how much faster than real time an individual robot is simulated, defined as:

$$r_{ACC} = \frac{n \cdot t_{SIM}}{t_{MEAS}} \tag{1}$$

where n is the number of robots, t_{SIM} is the simulated time, and t_{MEAS} is the measured run time. The second, r_{EPSS}, is a measure of how much energy is consumed to simulate each robot for one simulated second. This is defined as:

$$r_{EPSS} = \frac{P_{RUN} \cdot t_{MEAS}}{n \cdot t_{SIM}} = \frac{P_{RUN}}{r_{ACC}} \tag{2}$$

where P_{RUN} is the average power consumption of the system while running the benchmark.

Some previous work on mobile GPGPU, Maghazeh et al [10], uses the difference between idle and running power when making energy cost measurements, while other work, Pathania et al [18], considers the total system energy cost. We take the latter approach since it is more conservative, taking the view that the entire system is necessary in order to run the benchmark. A system designer may be able to reduce this overhead but never eliminate it. An on-board simulation can only be of use to a robot if there is enough power to run the robot too.

4.2 Benchmarks

We use a similar methodology to that described by Vaughan [21], using two worlds, *cave*, and *hospital*, populated with an increasing number of robots, each running an identical maximum dispersal controller. As Vaughan points out, this

represents a worst-case scenario for a ray-tracing simulator like Stage, since it maximises the space through which rays must propagate. The characteristics of the two benchmark series are summarised in Table 1.

The *cave* series uses the simple Pioneer 2DX robot model supplied with stage, which has a laser scanner range finder with 180 samples, and 16 ultrasonic range finders, each with a single sample. The robot body geometry is modelled with two polygons. The maximum number of robots simulated is 1000.

The *hospital* series uses a much larger world based on the hospital section bitmap supplied with Stage, with a smaller simpler robot. The body is only a single polygon with fewer sides, and just a laser sensor, though extended to a 350 degree field of view with a sample per degree. The maximum number of robots with *hospital* is 10000.

Table 1. Benchmarks

	cave	*hospital*
Size	64 m x 64 m	540 m x 220 m
Resolution	0.02 m	0.1 m
Grid locations	1×10^7	1.2×10^7
Robots	1-1000	1-10000
Robot size	0.4 m x 0.4 m	0.24 m x 0.24 m
Sensors per robot	16 sonar + 180 sample laser	350 sample laser
Rays per robot	196	350

In both series, we measure the real time taken to simulate 600 seconds of simulated time, and the total energy consumed, for each of the population numbers. In all cases, the tests are run with graphics disabled.

4.3 Target Devices

We targeted two mobile devices; the Samsung Chromebook and the Arndale Octa development board. We used these devices because they both have a System-on-Chip (SoC) with a GPU that supports the OpenCL language. The Samsung Chromebook is a low-cost lightweight laptop that runs the Chrome browser-based operating system. It is based on a Samsung Exynos 5250 SoC. The Arndale Octa is based around a more recent Samsung Exynos 5420 SoC. Some relevant specifications are shown in Table 2.

4.4 Energy Measurement

We measured the power by using 50 mR current sensing resistor in series with the system power supply, and measuring both the voltage drop across it and the voltage of the supply. The voltages were sampled at 10 ms intervals while a simulation was running and the product integrated to give a value for the total energy used for the simulation.

Table 2. Technical specifications of the two systems used. Note that due to limitations in the available Linux kernel it was only possible to run the Octa CPU at 800 MHz. System power values are typical, measured with the screen turned off for the Chromebook and with an accelerated Stage simulation running for the busy power.

	Samsung Chromebook	Arndale Octa
System-on-chip	Samsung Exynos 5250	Samsung Exynos 5420
CPU	Dual A15	Quad A15 + Quad A7
Max CPU frequency	1.7 GHz	1.8 GHz
Max CPU GFLOPS	27	58
GPU	ARM Mali T604	ARM Mali T628 MP6
Max GPU frequency	533 MHz	600 MHz
Max GPU GFLOPS	72	122
System idle power	1.8 W	1.2 W
System busy power	3.5 W	2.7 W

5 Results

Performance of unaccelerated Stage across the range of robot populations on all platforms and benchmarks showed the expected linear execution time. Energy usage was also relatively flat across the range. The GPU accelerated Stage performs poorly at low robot numbers, particularly for energy use, which is expected, but then overtakes the CPU-only version at higher robot population numbers. Figure 3 shows the results for the *hospital* series running on the Arndale Octa board. The other three results are omitted for brevity but show the same general picture, see Fig. 4 for a comparison.

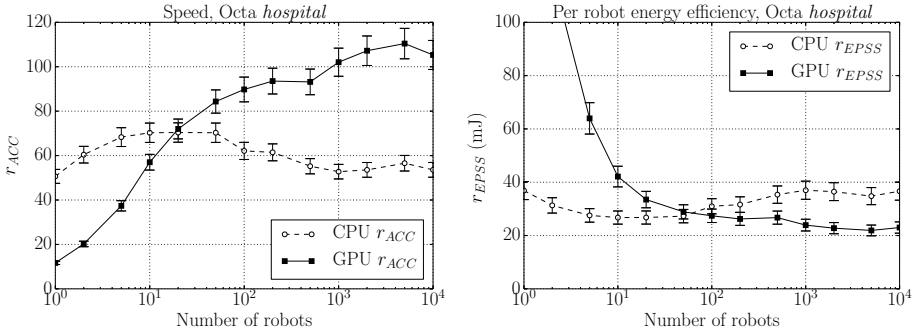

Fig. 3. Arndale Octa *hospital*. The CPU performance is broadly flat across the whole range of robot populations, demonstrating roughly linear scaling as described by Vaughan [21]. GPU performance is poor at low robot populations but exceeds the CPU in both speed and energy efficiency once above a population of 100 robots, or 35000 rays.

Figure 4 shows the relative performance between the GPU and CPU versions across all combinations of benchmark and hardware platform demonstrating the expected characteristics of a massively parallel throughput engine in that performance gains are not apparent until the level of parallelisation is high. Table 3 shows the points where the GPU performance reaches that of the CPU. The Arndale Octa and Chromebook show almost identical behaviour with regard to energy efficiency, but the break-even points for speed are much higher with the Chromebook than the Octa, probably due to the higher relative performance of the GPU compared to the CPU on the Octa.

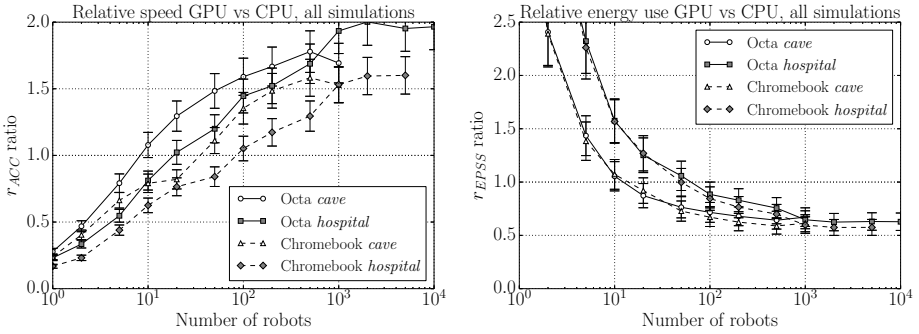

Fig. 4. The relative performance between the GPU accelerated and the CPU versions of Stage shows broad similarities across all four combinations of simulation series and hardware targets. Somewhere between a population of 10 to 100 robots, performance in both speed and energy efficiency on the GPU exceeds that of the unaccelerated software. The Chromebook demonstrates lower speed gains but almost identical energy efficiency gains with the GPU. The best improvement is the Octa *cave* series at 2000 robots, with at least 82% increase in speed and 30% drop in energy use.

Table 3. Break-even points for GPU performance versus CPU performance in number of rays.

	r_{ACC}	r_{EPSS}
Octa *cave*	1800	2200
Octa *hospital*	7000	21000
Chromebook *cave*	7800	2200
Chromebook *hospital*	32000	18000

We summarise the performance gains from GPU acceleration in Fig. 5. Taking the average for all data points with a population of 100 robots or more, there

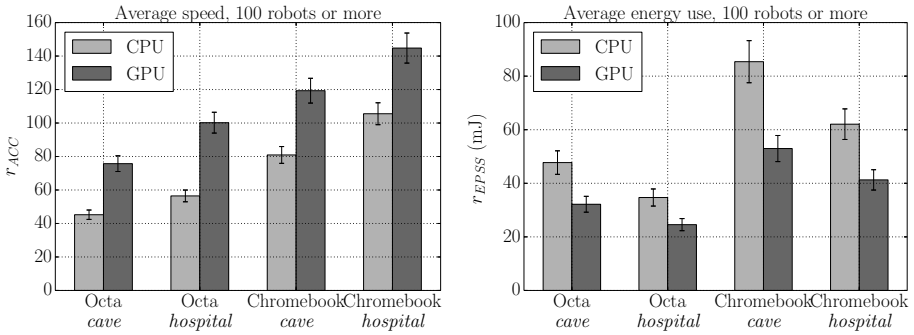

Fig. 5. Average performance at 100 robots or more.

are clear benefits across all benchmark and hardware combinations with both figures of merit.

6 Conclusions and Further Work

We have demonstrated a proof-of-concept GPU acceleration of the robot simulator Stage showing both simulation speed and energy efficiency gains. There are many further avenues down which this work can be taken.

In this initial proof-of-concept, we focussed on maximising benefit for minimal development risk. We intend to investigate many further optimisations of this approach. The execution on GPU and CPU can be overlapped, the construction of the data structures for the GPU could be made much faster, and a smart allocation of rays to GPU cores could increase speed by improving cache behaviour through increasing locality of access within the world grid data structure. In addition, alternative ray tracing algorithms may be a better fit for the characteristics of a GPU.

We intend to adapt our approach to support further work on the 'consequence engine' described in Winfield et al [25] in which each of the simulation scenarios contain only a few robots. By constructing a world containing many such scenarios arranged in a grid, we can have a single simulation with many robots, such that GPU acceleration will be beneficial. An essential requirement for such an 'ethical' robot is that *what-if* simulations are conducted embodied in the robot, rather than at the other end of an unreliable communications link. Accelerated simulation on a low-power mobile platform moves towards that goal, and we also intend to equip e-puck robots with mobile GPU hardware to demonstrate embodied simulation.

The use of mobile System-on-Chip devices with GPUs opens new possibilities for robot self simulation. This paper demonstrates the viability of one possible approach and points the way towards autonomous robots with *what-if* capability.

References

1. Josh Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121, 2006.
2. Daniel Cohen-Or and Arie Kaufman. 3d line voxelization and connectivity control. *Computer Graphics and Applications, IEEE*, 17(6):80–87, 1997.
3. Marco Dorigo, Vito Trianni, Erol Şahin, Roderich Groß, Thomas H Labella, Gianluca Baldassarre, Stefano Nolfi, Jean-Louis Deneubourg, Francesco Mondada, Dario Floreano, et al. Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17(2-3):223–245, 2004.
4. Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Alex Ramirez. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 123–132. IEEE, 2014.
5. Sabine Hauert, J-C Zufferey, and Dario Floreano. Reverse-engineering of artificially evolved controllers for swarms of robots. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 55–61. IEEE, 2009.
6. Seung Heon Kang, Seung-Jae Lee, and In Kyu Park. Parallelization and optimization of feature detection algorithms on embedded gpu. *International Workshop on Advanced Image Technology*, 2014.
7. Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.
8. Khronos OpenCL Working Group et al. *The OpenCL Specification, Version 1.1*, 2010.
9. Wenguo Liu and Alan FT Winfield. Open-hardware e-puck linux extension board for experimental swarm robotics research. *Microprocessors and Microsystems*, 35(1):60–67, 2011.
10. Arian Maghazeh, Unmesh D Bordoloi, Petru Eles, and Zebo Peng. General purpose computing on low-power embedded gpus: Has it come of age? In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 1–10. IEEE, 2013.
11. Dinesh Manocha. General-purpose computations using graphics processors. *Computer*, 38(8):85–88, 2005.
12. Hugo Gravato Marques and Owen Holland. Architectures for functional imagination. *Neurocomputing*, 72(4):743–759, 2009.
13. Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stéphane Magnenat, Jean-Christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on autonomous robot systems and competitions*, volume 1, pages 59–65, 2009.
14. Nvidia. *NVIDIA CUDA, Compute Unified Device Architecture Programming Guide*. NVIDIA, Santa Clara, CA, USA, 2007.
15. Paul O'Dowd, Alan FT Winfield, and Matthew Studley. Towards accelerated distributed evolution for adaptive behaviours in swarm robotics. In T. Belpaeme, G. Bugmann, C. Melhuish, and M. Witkowski, editors, *Towards Autonomous Robotic Systems*, pages 169–175. University of Plymouth, 2010.
16. Paul J O'Dowd, Alan FT Winfield, and Matthew Studley. The distributed coevolution of an embodied simulator and controller for swarm robot behaviours. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4995–5000. IEEE, 2011.

17. Kazuhiro Ohkura, Toshiyuki Yasuda, Yoshiyuki Matsumura, and Masaki Kadota. Gpu implementation of food-foraging problem for evolutionary swarm robotics systems. In *Swarm Intelligence*, pages 238–245. Springer, 2014.
18. Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 40:1–40:6, New York, NY, USA, 2014. ACM.
19. Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, et al. Argos: a modular, multi-engine simulator for heterogeneous swarm robotics. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 5027–5034. IEEE, 2011.
20. Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm robotics*, pages 10–20. Springer, 2005.
21. Richard Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2-4):189–208, 2008.
22. Richard Vaughan and Mauricio Zuluaga. Use your illusion: Sensorimotor self-simulation allows complex agents to plan with incomplete self-knowledge. In *From Animals to Animats 9*, pages 298–309. Springer, 2006.
23. Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R Cavallaro. Accelerating computer vision algorithms using opencl framework on the mobile gpu-a case study. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2629–2633. IEEE, 2013.
24. Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R Cavallaro. Computer vision accelerators for mobile systems based on opencl gpgpu co-processing. *Journal of Signal Processing Systems*, pages 1–17, 2014.
25. Alan FT Winfield, Christian Blum, and Wenguo Liu. Towards an ethical robot: internal models, consequences and ethical action selection. In Michael Mistry, Aleš Leonardis, Mark Witkowski, and Chris Melhuish, editors, *TAROS 2014 - Towards Autonomous Robotic Systems*, volume 8717 of *Lecture Notes in Computer Science*, pages 85–96. Springer International Publishing, 2014.