EG UK Theory and Practice of Computer Graphics (2012)
Hamish Carr and Silvester Czanner (Editors)

Path Tracing on Massively Parallel Neuromorphic Hardware

P. Richmond[1] and D.J. Allerton[1]

[1]Department of Automatic Control Systems Engineering, University of Sheffield

Abstract

Ray tracing on parallel hardware has recently benefit from significant advances in the graphics hardware and associated software tools. Despite this, the SIMD nature of graphics card architectures is only able to perform well on groups of coherent rays which exhibit little in the way of divergence. This paper presents SpiNNaker, a massively parallel system based on low power ARM cores, as an architecture suitable for ray tracing applications. The asynchronous design allows us to demonstrate a linear performance increase with respect to the number of cores. The performance per Watt ratio achieved within the fixed point path tracing example presented is far greater than that of a multi-core CPU and similar to that of a GPU under optimal conditions.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing

1. Introduction

Ray tracing offers a significant departure from traditional rasterized graphics with the promise of more naturally occurring lighting effects such as soft shadows, global illumination and caustics. Understandably this improved visual realism comes at a large computational cost. Traditionally ray tracing has been realised in software with high performance supercomputer solutions addressing the issue of performance [GP90]. The increased flexibility of the Graphics Processing Unit (GPU) has more recently opened up the possibly of massively parallel ray tracing on consumer hardware [PBMH02, PBD*10]. Essential to the success of this work has been the ability to exploit thread level coherency within the wide SIMD architecture. Typically this requires the grouping of similar threads via the use of generated hierarchical data structures [FS05]. Due to the issue of ray coherency and the requirement of efficient algorithms for generating hierarchical data structures on the fly, GPU ray-tracing has failed to achieve the same orders of magnitude speed-up achieved in other GPU accelerated domains such as molecular dynamics. This situation has led to the use of energy hungry supercomputing clusters of GPUs for high performance ray tracing applications.

Aside from pure supercomputing performance, energy efficiency and cooling requirements are quickly becoming a key consideration with respect to supercomputing metrics. The SpiNNaker architecture is a massively parallel (up to a million cores) and highly interconnected architecture which considers power efficiency as a primary design goal. Originally designed for the purpose of simulating large neuronal models in real time, the architecture is based on low power (passively cooled) asynchronous ARM processors with programmable cores. This paper examines the potential of the SpiNNaker architecture in its current form for the purposes of ray tracing. More specifically this paper describes the implementation and performance of a path tracer realised using fixed point integer arithmetic [HRB*09] suitable for the ARM based architecture of SpiNNaker. Consideration is given to the energy efficiency of the performance results which demonstrate a significant performance per Watt ratio.

2. The SpiNNaker architecture

The SpiNNaker hardware architecture consists of a a number of independently functional and identical Chip-Multiprocessors each consisting of 18 ARM 968 cores running at 200MHz. Each core has its own dedicated tightly coupled memory, holding 32KB of instructions and 64KB of data. Each chip also contains 1Gbit of shared SDRAM connected via a DMA controller which uses an asynchronous "system" Network on Chip (NoC) to replace the requirement of a traditional on chip bus. The system NoC implies a Globally Asynchronous and Locally Synchronous (GALS) timing model [PFT*07] allowing each chip to run in its own timing domain. Similarly, a second "Communication" NoC

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG12/025-028

provides a GALS multicast router, responsible for routing small packets between cores and out to 6 other chips on the SpiNNaker network. DMA events and packet inputs from the NoC interfaces are supported via interrupts within the overall event driven model of the architecture. A complete SpiNNaker network consists of SpiNNaker chips arranged in a 2D triangular torus with connections to the north, source, east, west, southwest and northeast.

A primary application of the SpiNNaker architecture is neural simulation. Each of the ARM 968 cores is sufficiently fast to provide simulation of up to 1000 simple integrate and fire neurons (or less neurons based on more advanced neuronal dynamics) in real time. The proposed SpiNNaker system of 1 million cores will therefore be sufficient to simulate a billion neurons (or 1 percent of the human brain). The choice of low power ARM cores suggests that the 1 million core system should generate only 250mW to 500mW per chip.

3. Path Tracing on SpiNNaker

In order to evaluate the suitability of SpiNNaker for ray tracing applications a simple path tracer has been implemented using the GCC cross-tools compiler for ARM. Our implementation is based upon the SmallPt (`http://kevinbeason.com/smallpt/`) path tracer, a minimal path tracer in 99 lines of C++. Whilst this is not particularly optimised it has a small instruction and data overhead which avoids having to use slower off chip SDRAM. The modified Cornel box scene described procedurally by the path tracer program consists entirely of spheres (9 in total) simplifying the ray collision code. A larger number of similar implementations exist on other parallel architectures such as the Graphics Processing Unit (GPU) which provide useful benchmarking.

Interfacing with the SpiNNaker hardware is limited, our test boards which consist of four 18 core chips, provides a single LED per chip (which is useful for debugging) and an Ethernet connection for transferring data packets in an internal packet format of up to 256 bytes. This connection is used initially to send packets to a rudimentary run-time system which allows memory values to be manipulated (including program loading) and chip execution to begin. During execution, packets can be routed back to the host via the use of a call to a system *printf* function. This configuration therefore requires a client server based design for our path tracer with the client periodically sending pixel samples to a visualisation server on the host machine.

The path tracing client application is identical for each core with the exception of a random state which is used throughout to Monte Carlo based processes. Each iteration of the algorithm calculates a specified *SPP* number of samples packet (each with 2x2 sub pixel samples for anti aliasing) before communicating the total pixel radiance back to the host and selecting a new pixel to proceed. Due to memory restrictions recursion is avoided and the main radiance calculation uses a loop to avoid branching. Pixel selection can be either stochastic (utilising the cores random state) or sequential with each core starting at a unique position based upon the SpiNNaker core ID. Path termination can be either based upon Russian Roulette or a fixed recursion depth criterion. Whilst the former is more computationally expensive, the asynchronous nature of the SpiNNaker allows different cores to follow fully divergent code paths without the noticeable performance penalties observable on SIMD architectures.

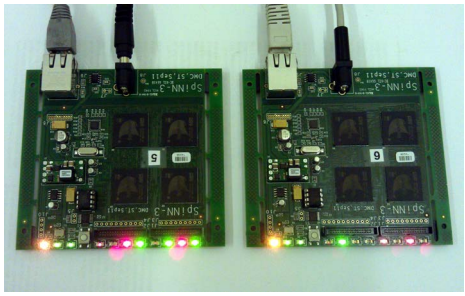

Figure 1: *Two SpiNNaker test boards each with 4 chips containing 18 cores each.*

4. A Fixed Point Path Tracing Implementation

The SpiNNaker hardware architecture does not provide any Floating Point Unit (FPU) and therefore all arithmetic requires a suitable fixed point integer storage format. We have chosen a fixed point format based on a 16 bit integer and 16 bit fractional part with a range of -32768 to +32767 and a resolution of approximately 1.526×10^{-5}. A library for fixed point arithmetic is provided including a a fixed point multiply which returns a 64 bit integer (in 48:16 format) which must be downcast and an ARM assembly implementation for fixed point division (`http://me.henri.net/fp-div.html`).

Implementations of common Mathematics functions for fixed point hardware often use a combination of either approximation functions, Taylor series expansions or lookup tables. Our implementation uses a "binary restoring square root extraction" [Tur94] to provide a fast square root implementation and a fast parabolic sin approximation which gives more accurate results than a Taylor expansion with less computational cost.

Whilst performing geometric operations within our path tracer it is possible to exceed the range and resolution of our fixed point format. This is most obvious when considering vector normalisation and the calculation of the determinant of the quadratic for a ray sphere intersection. In both cases the results from fixed point multiplications cannot be cast

back into the 16:16 32 bit range. Instead the 48:16 format is preserved until the values are square rooted. The implementation of a 48:16 version of the sqrt algorithm exploits the identity $\sqrt{xy} = \sqrt{x}\sqrt{y}$ to perform a 32 bit square root by shifting the 48:16 format number into the 16:16 range and then multiplying the result by the root of the divisor value (indicated by the shift amount).

Random number generation is achieved by using an implementation of a linear congruential generator. Each core implementing our path tracing client maintains its own seed value. To initialise the seed value of each core a shared seed is set in shared system RAM of each SpiNNaker chip. Sequential loading allows each seed to be generated using the shared seed value, itself iterated using the linear congruential generator. Experimentation has shown that a careful choice of multiplier and increment are necessary, especially as random numbers within the range of 0 and 1 utilise only the bottom 16 bits of the seed value.

5. Path Tracing Visualiser

Visualisation of the path tracing results is provided through an interactive server application which runs on the "host" machine connected to the SpiNNaker hardware. The server is initialised by the receipt of packets which arrive in a fixed format composed of a pixel location followed by an *r*,*g* and *b* radiance value. The *rgb* values are stored in discrete buffers which hold cumulative average values for each pixel with a *samples* buffer storing the running total number of samples per pixel. A separate pixel buffer maps the *rgb* radiance values into a single 32 bit RGBA format (after clamping and gamma correction) which is continuously rendered by a separate display loop thread. A final activity buffer holds an RGBA colour value (set to green with zero alpha when a packet is received for a given pixel). The render loop decays each activity buffer value by increasing the alpha value to give a visual heat map used to display an overlay indicating pixel activity. As each packet is received a number of statistic are also updated and optionally displayed including values such as total packets, total samples, running time, average packets/samples per pixel and average packets/samples per second.

6. Results and Discussion

Figure 2 shows a visual comparison of the fixed point path tracer in comparison with the results rendered using full double precision and results from a floating point implementation of the path tracer rendered within SmallptGPU (`http://davibu.interfree.it/opencl/smallptgpu/`) using OpenCL for the GPU. As with floating point implementations, numerical precision causes an issue for self intersections and as such a small ε value is used to offset rays from their originating surface. Figure 3 shows that for small ε values of 0.01 the numerical

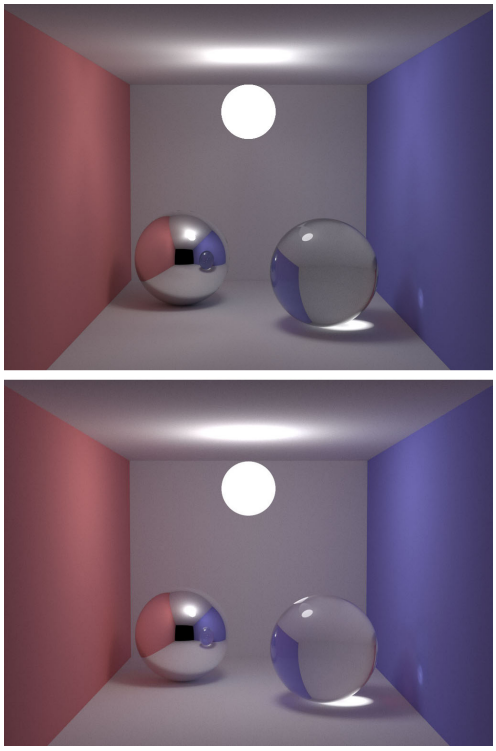

Figure 2: *Comparison between our fixed point path tracer on SpiNNaker(top) and double precision CPU result (bottom) both rendered using 20,000 samples per pixel.*

precision of our 16:16 format is insufficient to avoid significant rendering artefacts during intersection. An ε of 0.06 still shows some minor artefacts which are not observable for an ε of 0.1 (2). Unfortunately large ε values prohibits the generation of the reflectance based light hat visible on the glass sphere (right).

The performance of the fixed point path tracer running on the SpiNNaker hardware has been evaluated by considering the average samples per second for a number of hardware configurations at varying *SPP* values. Figure 4 shows this performance using both "Russian Roulette" path termination and a fixed number of 4 bounces per ray. The results show a linear scaling of performance for both ray termination techniques as the number of SpiNNaker cores are increased.

In contrast with alternative parallel architectures, a multi core CPU implementation of the Smallpt path tracer is able to render a scene of 1024×768 pixels with $25,000$ samples per pixel (spp) using Russian Roulette path termination in 10.3 hours on a 2.4GHz Intel Core 2 Quad CPU. This implies a rate of roughly $530,226$ total samples per second (*sps*). In direct compassion this is approximately 4 times faster than our 136 core implementation. Considering that the dual test card set-up used for our calculations consumes

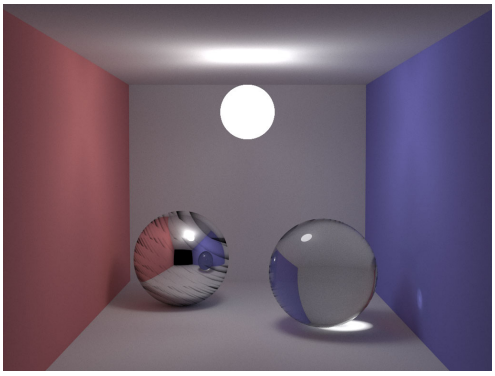

Figure 3: *Using an insufficient epsilon value (in this case* 0.01*) causes visual artefacts as a result of the fixed point precision used to implement the path tracer on SpiNNaker hardware.*

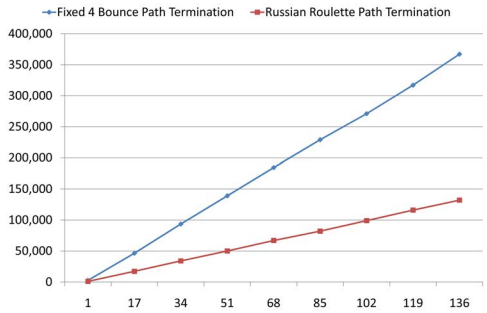

Figure 4: *Results show linear scaling of performance (i.e. samples per second shown on Y axis) vs. the number of SpiNNaker cores (X axis) for both fixed bounce and Russian Roulette path termination.*

at most $4W$ of power this is considerably more cost effective in terms of sps/W than that of the Quad Core set-up which requires a potential of $105W$ (excluding power requirements of other system components).

GPU implementations of the Smallpt path tracer boast a considerably greater performance than that of multi core CPU implementations. Tokaspt (`http://code.google.com/p/tokaspt/`) is significantly faster than SmallptGPU and boasts a performance of $185.6M$ sps (for a fixed maximum of 4 bounces per sample) using an NVIDIA Quadro FX 5800. In terms of performance per Watt a conservative estimate of $500W$ for a high end PC system with a fully loaded Quadro FX 5800 implies a value of $371,200sps/W$. In direct comparison with the SpiNNaker path tracer this is roughly 4 times better than our implementation on SpiNNaker. It is worth noting however that using a fixed number of bounces per ray presents an almost ideal case for the GPU in that there is no thread level divergence across threads. In more realistic situations or comparing our method with Russian Roulette would likely demonstrate a swing in our favour with respect to overall performance per Watt.

7. Future Work

Future work will concentrate on the use of SpiNNaker in more robust ray tracing applications involving large datasets requiring access to SDRAM memory. The most obvious mapping of a pixel per core in the million core system seems the most intuitive way to achieve real time performance. Despite this we are also interested in how massive distributed datasets may be mapped to and rendered efficiently using the hardware. There is also significant scope to explore caching techniques and intelligent scaling of the fixed point format. From a practical perspective, we have observed that the largest number of packet tracer packets which can be communicated via ethernet is roughly 1500 per second. Any more than this causes active cores to interfere with the loading and starting of new cores on the same test board. This therefore requires that the value of *SPP* is increased at the same rate as the core count. In future work it will be necessary to consider the use of higher bandwidth output mechanisms or a more direct hardware framebuffer implementation. Likewise future variations of SpiNNaker may consider alternative ARM cores capable of SIMD instructions or the possible addition of a FPU if necessary.

References

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 15–22. URL: http://doi.acm.org/10.1145/1071866.1071869, doi:http://doi.acm.org/10.1145/1071866.1071869. 1

[GP90] GREEN S. A., PADDON D. J.: A highly flexible multiprocessor solution for ray tracing. *The Visual Computer 6* (1990), 62–73. 10.1007/BF01901067. URL: http://dx.doi.org/10.1007/BF01901067. 1

[HRB*09] HEINLY J., RECKER S., BENSEMA K., PORCH J., GRIBBLE C.: Integer ray tracing. *journal of graphics, gpu, and game tools 14*, 4 (2009), 31–56. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics* (August 2010). 1

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002). 1

[PFT*07] PLANA L., FURBER S., TEMPLE S., KHAN M., SHI Y., WU J., YANG S.: A gals infrastructure for a massively parallel multiprocessor. *IEEE Design & Test of Computers 24(5)* (2007), 454–463. 1

[Tur94] TURKOWSKI K.: *Fixed Point Square Root (Tech Report 96)*. Tech. rep., Apple Computers, 1994. 2