Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model

Per Ganestam*
Lund University

Michael Doggett†
Lund University

ABSTRACT

This paper presents a method for auto-tuning interactive ray tracing on GPUs using a hardware model. Getting full performance from modern GPUs is a challenging task. Workloads which require a guaranteed performance over several runs must select parameters for the worst performance of all runs. Our method uses an analytical GPU performance model to predict the current frame's rendering time using a selected set of parameters. These parameters are then optimised for a selected frame rate performance on the particular GPU architecture. We use auto-tuning to determine parameters such as phong shading, shadow rays and the number of ambient occlusion rays. We sample a priori information about the current rendering load to estimate the frame workload. A GPU model is run iteratively using this information to tune rendering parameters for a target frame rate. We use the OpenCL API allowing tuning across different GPU architectures. Our auto-tuning enables the rendering of each frame to execute in a predicted time, so a target frame rate can be achieved even with widely varying scene complexities. Using this method we can select optimal parameters for the current execution taking into account the current viewpoint and scene, achieving performance improvements over predetermined parameters.

1. INTRODUCTION

Programming GPUs for high performance requires a careful balance of several hardware specific related factors that is typically only achieved by expert users through trial and error. GPUs are massively parallel devices with parallel compute capacity exceeding other single chip devices and are still the best device for high performance graphics [8]. There are currently many APIs for programming GPUs all with their respective advantages and disadvantages, but getting optimal performance from the GPU is still a challenging task that requires repetitive manual tuning. To reduce the amount of trial and error required to achieve optimal performance, general guidelines can be followed or different metrics can be considered to predict performance, but ultimately a trial and error process is still prevalent. In this paper, we present a method that makes this tuning process automatic using an analytical GPU model.

The current challenges of programming and getting efficient performance from GPUs is likely to increase in the future as new devices have increasily complex architectures. While new features, such as shader cache hierarchies, make programming easier, getting the best efficiency is still difficult. Also as pointed out by Owens et al. [8], the cost of memory bandwidth in comparison to computational power is ever increasing. On future devices applications will need to switch to modes that require more compute processing, whereas on older devices a better trade off between memory and compute is necessary. Power usage is also an important consideration and Dally [4] points out that the power cost of a memory operation is an order of magnitude greater than compute operations. Sometimes it is important to tune parameters for power usage instead of just performance. Using a single set of parameters for all devices is therefore inadequate and per device tuning is also needed, resulting in an ongoing maintenance task.

We present a method for automatically tuning the parameters of a parallel application running on massively parallel devices, GPUs. Automatic tuning is performed by estimating the run time for a given set of parameters using a GPU analytical model [7] and then changing parameters and re-estimating until a chosen performance target is met. A major advantage of this system over using feedback from the previous frame is that the rendering load based on viewpoint can change dramatically between frames, hence a previous frame's feedback loop could become inaccurate and highly unstable. We model GPU performance across a range of GPU architectures including older generations and different vendors by using the OpenCL API. We evaluate our auto-tuning method with a ray tracing application, because it has a range of parallel properties including a large amount of parallel work, high memory bandwith usage, and a workload that can be sometimes coherent and sometimes incoherent.

This paper contributes a new method for auto-tuning using an analtyical GPU model. Our method uses a feedforward controller to improve the performance estimate achieved over what is possible with just the GPU model. In addition, a simplified version of Hong and Kim's GPU model [7] is presented. This paper also presents GPU modelling across both GPU vendors, NVIDIA and AMD.

2. PREVIOUS WORK

Early efforts to tune rendering performance include work by Funkhouser et al. [6] which tunes the rendering algorithm using predetermined geometry properties and previous frame rates to ensure a user-specified frame rate. By using a cost and benefit calculation for each object a target frame rate can be achieved by setting

*perg@cs.lth.se
†mike@cs.lth.se

appropriate LOD levels and adjusting image quality for each object in the scene. In our approach we also control rendering parameters, but use a GPU model to get accurate estimates of the final performance for the current frame enabling much faster tuning.

Making efficient use of GPU hardware resources has always been a challenging task and a large amount of research is devoted to the discovery of improved tuning parameters for a particular algorithm that have been found through trial and error or developer tools. But recent efforts to find more general methods have been presented. Using metrics such as the number of instructions and threads running on a GPU at one time, Ryoo et al. [9] shows how measures of utilization and efficiency can be computed to predict which regions of the complete space of available optimizations need to be tested in order to find the optimal setting. This method reduces the search time considerably, but still requires iterating over variables to find the best optimizations. Their model also only works if the application does not have memory bandwidth issues.

In recent years several efforts have been made to understand and model GPU architectures. Hong et al. [7] propose an analytical GPU model that can be used to estimate the performance of algorithms by also taking into account the impact of memory operations. They count the number of memory transactions as well as the actual address to better model the amount of parallelism available when memory requests happen on GPUs. They measure GPU characteristics using microbenchmarking and can then make performance estimates using their analytical model. We use this model in this paper to tune our application. Bakhoda et al. [3] presents a detailed GPU simulator which takes PTX instructions and executes them to analyze execution performance. Their simulator gives accurate performance estimates but does not provide quick estimates that can be used to rapidly tune performance before execution. Baghsorkhi et al. [2] present a GPU model that also models important properties such as scratch-pad memory access and control flow divergence using a work flow graph. Further details of GPU architecture including cache sizing can also been determined via the use of micro benchmarking [10].

Recent results in auto-tuning show that using statically determined parameters for algorithms that run on GPUs always produce poorer results than tuning those parameters to the GPU capabilities [5]. Davidson et al. [5] also show that by running multiple benchmarks of an algorithm, optimal performance can be achieved. In this paper we improve upon this result by directly querying a GPU model to determine the best parameters. This enables a wider range of parameters to be checked, which is important for complex algorithms such as ray tracing.

In this paper we tune the ray tracing algorithm for image synthesis. Previous work in GPU ray tracing has also used statically determined parameters or algorithm changes to improve performance. Aila et al. [1] introduced the concept of persistent threading in order to improve GPU utilization beyond that achieved by the hardware work scheduler. They found that packet tracing was not much faster than per-ray tracing even though per-ray introduces more incoherent memory accesses. To improve performance they created scheduling threads on the GPU, called persistent threads, which improved the performance two-fold by moving scheduling into the GPU thread. Persistent threads are an example of a non-standard performance optimization that programmers may not be aware of, but could be incorporated into an auto-tuning system.

3. GPU PERFORMANCE MODEL

To estimate the performance of an algorithm on the GPU, we use a GPU model based on Hong and Kim's model. [7]. We use Hong and Kim's model on AMD GPUs as well as NVIDIA GPUs. The model is deterministic and straight forward to implement. It also executes quickly with practically no overhead, hence it is well suited for real-time applications. In this section we present a shorten version of their analytical GPU model. The properties of the program and the device are measured as shown in Table 1. The number of cores, D_c, is the number of Streaming Multicores (SM) for NVIDIA GPUs and SIMDs for AMD GPUs.

Program	
i_c	Number of compute instructions
i_m	Number of memory instructions
t_b	Number of threads per block
b_t	Number of blocks to run
b_o	Maximum blocks per core (occupancy)
Device	
M_d	Device total memory bandwidth
D_c	Number of cores
D_{tw}	Device number of threads per warp
W_d	Device resource limited number of warps $= \frac{b_o t_b}{D_{tw}}$
C_i	Cycles to execute one instruction
Benchmarked	
c_{ml}	Averaged memory latency cycles for one memory operation

Table 1: Input variables used for the GPU model.

To compute the total number of cycles, several intermediate values are also computed as shown in Table 2. These intermediate values are computed using the same equations as Hong and Kim [7] and the variable name from their model is also shown in the Table.

		Hong09 variable
M_w	Memory bandwidth per warp	BW_per_warp
c_c	Computation cycles	$Comp\_cycles$
c_m	Memory cycles	Mem_cycles
c_t	Total execution cycles	$Exec\_cycles\_app$
c_{ml}	Averaged memory latency	Mem_L
c_{dd}	Averaged departure delay to issue a memory instruction	$Departure\_delay$

Table 2: Variables computed by the GPU model.

GPUs are throughput oriented architectures capable of running thousands of program threads in parallel [8]. When a thread requests data from memory the GPU switches to other threads to ensure the GPU continues to do work while it waits for memory to respond. This switching is done on a warp (called wavefront for AMD hardware) basis. In the worst case, when data is not in any on-chip cache and must be read from off-chip global memory, the original thread must wait a memory latency period. This memory latency is measured using benchmarks for each chip and represented by the variable c_{ml} in graphics core clock cycles. While GPUs are capable of running multiple kernels, we assume that the device is running only multiple instances of the same kernel program. The maximum number of warps that are runnable on a core is determined by available resources such as OpenCL private memory, OpenCL local memory (NVIDIA shared memory) and other device specific features. We use the device specified parameter that limits the number of blocks (groups of warps) per core to determine this maximum which is called W_d.

Our objective is to compute the number of graphics core clock cycles that it takes to execute the specified program. When a warp issues a global memory request, it is put to sleep and other warps are run instead. While waiting for the sleeping warp's memory request, the GPU can run the compute instructions from other warps. We can compute the number of warps that run only compute instructions while waiting as

$$w_c = \frac{c_m + c_c}{c_c}. \quad (1)$$

This is the maximum amount of compute that can be done measured in warps. This must be capped by the maximum number of warps, W_d so we take $\min(w_c, W_d)$ and call it *Compute Warp Parallelism* (CWP) [7].

Next, we consider how many memory operations could be performed. First, we compute the maximum number of parallel warps that can issue a memory operation while the first warp is sleeping. The number of warps that can concurrently issue a memory operation is computed as follows:

$$w_{md} = \frac{c_{ml}}{c_{dd}}. \quad (2)$$

Each warp that makes memory requests uses some of the limited memory bandwidth resulting in a maximum number of warps that can run. This maximum number of warps is computed as

$$w_{mb} = \frac{M_d}{M_w D_c}. \quad (3)$$

Again, these numbers of memory warps must be limited by the maximum number of warps of the device, W_d, so we take $\min(w_{md}, w_{mb}, W_d)$ and call it *Memory Warp Parallelism* (MWP) [7].

From this, we can compute the total number of cycles for the kernel to run by multiplying the total of compute and memory cycles per block by the total number of block repetitions required on the device:

$$c_t = \frac{b_t}{b_o D_c}(C_i w_t(i_c + i_M) + c_b). \quad (4)$$

The total execution equation requires two variables which are determined by the limiting case for parallelism, compute warps w_t, and memory cycles per block c_b. For these two variables, there are three possible cases. The first case is when $MWP > CWP$, i.e., only CWP warps will be able to run because of the amount of compute instructions, so the device is *arithimetic* bound. In this case, the maximum number of warps that can run in parallel is determined by device properties, and this is the main contributor to the total number of cycles.

The second case is when $CWP > MWP$, i.e., only MWP warps will be able to run because of the amount of cycles memory instructions require, so the device is *memory* bound. The maximum number of parallel warps is determined by MWP. The third case is when the number of warps that can run in parallel for compute and memory instructions are the same and equal to the device limit of number of warps running, this case is refered to as *balanced*. Once the type of limitation is worked out, w_t and c_b are determined as shown in Table 3.

	Arithmetic	Memory	Balanced
w_t	W_d	$\frac{MWP-1}{i_m}$	$\frac{MWP-1}{i_m} + 1$
c_b	c_{ml}	$\frac{W_d c_m}{MWP}$	c_m

Table 3: Total execution time variables.

3.1 Parameters for Different GPUs

Several of the model parameters are measured using a series of synthetic benchmarks with known numbers of compute and memory operations. The memory operations are either consectutive memory accesses so that they will be combined, called coalescing or random addresses. More details about these benchmarks can be found in Hong and Kim [7].

The measured memory parameters are memory latency, c_l, departure delay for coalesced memory access, c_{dc}, and for uncoalesced memory access, c_{du}. These three values are varied to find the best fit for the particular architecture. The coalesced memory access takes into account the lower memory access time required for shader loads and stores from different threads in the same warp. These memory values and also the architectural parameters of three different GPUs are shown in Table 4

	GF 8800	GF 580	Radeon 5870
M_d	86	192	153
D_c	16	16	20
D_{tw}	32	32	64
C_i	4	2	4
Max b_o	8	8	24
F (GHz)	1.35	1.54	0.88
Benchmarked			
c_l	420	550	500
c_{dc}	4	0.08	64
c_{du}	9.8	0.66	1

Table 4: GPU dependent variables.

In Table 4, for the Geforce580 architecture, we set C_i to 2 to account for the 2 instructions issued per SM resulting in a halving of the effective cycle time for each instruction. The blocks per core determined by occupancy is given as the maximum value for the architecture 'Max b_o', in Table 4.

4. ESTIMATING WORKLOAD

Ray tracing renders an image of a 3D scene by tracing a path from the eye into the geometry, intersecting with objects in the scene. For a ray tracer to handle large complex scenes, a hierarchical data structure is typically used to improve the performance of finding intersections. In particular we use a bounding volume hierarchy (BVH) constructed using a surface area heuristic. Each ray starts at the root node that contains a bounding box for the entire scene. For each node that is intersected, the two child nodes are read into memory, typically from global memory, and intersection testing is performed with their bounding boxes. This continues recursively until the leaf nodes are reached. Once the ray reaches a leaf node in the BVH, it intersects with the triangles contained there. To estimate performance of ray tracing the algorithm is divided into several major components such as the number of nodes

traversed, the number of nodes read in, and the number of triangles intersected.

Since every scene is different and even viewpoints within a scene vary greatly, we use a low resolution ray tracer to estimate the ray tracing specific parameters. One of the benefits of working with OpenCL is that we can easily make use of the CPU to quickly estimate these values and not put extra load on the GPU.

4.1 Ray Tracing Parameters

Beyond the basic surface of the objects in the scene, the actual lighting of the scene requires computation as well. To create a more realistic scene, more complex computation is required. Different techniques can be incrementally added to increase the realism and in this paper, we control the level of realism based on the available hardware. We add shadow rays and ambient occlusion (AO). Shadow rays trace a ray from the surface to each light source to determine if the surface is in light or shadow. AO attempts to approximate the light that is reflected by the scene to a point by calculating how much of a white hemisphere around the point it can 'see'. The visibility of the hemisphere is calculated by tracing a selected number of rays in random directions and terminating them at a set radius. We tune the performance of AO by adjusting the number of rays and the terminating radius.

4.2 Estimating Shader Cache Performance

The NVIDIA Fermi architecture used in the GeForce 4XX and 5XX series includes an L1 and L2 cache hierarchy for global memory loads and stores from a kernel program. This memory hierarchy improves performance significantly for our BVH based ray tracer, as the BVH nodes are frequently stored in this cache. When running the low resolution frame estimate, we store the final BVH node for each ray. We count the number of rays that end at the same node within a region of the screen and assume that rays that terminate at the same node are likely to have taken a similar path through the BVH and so when reading nodes from memory, the nodes are likely to be already in the cache. This estimate of cache hits is represented by the variable m and used to estimate the performance by modifying the instruction count.

4.3 Instruction Counting

We calculate the compute and memory instruction counts for our ray tracing kernel on NVIDIA GPUs, by using the ability to save the OpenCL kernel binary using clGetProgramInfo. The binary is compiled using `nvcc` for the target architecture and the assembler dumped using `cuobjdump`. For AMD GPUs we use the KernelAnalyzer application which compiles OpenCL directly into machine assembler code. To get the final number of instructions, we break the kernel into instruction counts inside loops and outside loops. These instruction counts are denoted by the variable i. Using the low resolution workload estimate, we compute an average of the number of times each loop runs and denote these variables as n.

The total number of compute instructions is calculated as:

$$i_c = i_{ct}(a_t a_r + p_t) + i_{ci}(p_i + a_w t_v a_r) + i_{cco},$$

where the variables are described in Table 5.

The estimated ratio m is used both to divide between coalesced and uncoalesced memory instructions and to measure cache performance. This is possible due to the similarity of probability to have a coalesced memory access and a cached memory fetch. The total number of memory instructions are calculated as follows:

Instruction count variables	
i_c	Total number of compute instructions
i_{ct}	Compute instructions per traversal
i_{ci}	Compute instructions per intersection
i_{cco}	Constant compute instructions
i_m	Total number of memory instructions
i_t	Initial number of memory instructions
i_{mt}	Memory instructions per traversal
i_{mi}	Memory instructions per intersection
i_{mco}	Constant memory instructions
i_{mu}	Uncoalesced memory instructions
i_{mc}	Coalesced memory instructions
m	Memory coalescing estimate
m^2	Memory cache performance estimate
p_t	Number of primary ray traversals
p_i	Number of primary ray intersections
t_v	Number of visible triangles
a_t	Number of AO node traversals
a_r	Number of AO rays
a_w	AO intersection cost
a_m	Total AO memory instructions
a_{ca}	AO cache performance estimate

Table 5: Compute and memory instruction count variables.

$$\begin{aligned} i_t &= (i_{mt}p_t + i_{mi}p_i + i_{mco})(1 - m^2), \\ a_{ca} &= 1 - \min(1, 2^{-(\frac{a_t + p_t}{23} - 1)}), \\ a_m &= a_{ca}(i_{mt}a_t a_r + \frac{t_v i_{mi} a_r}{2}), \\ i_{mu} &= i_t(1 - m) + a_m, \\ i_{mc} &= i_t m, \end{aligned}$$

The total number of memory instructions is $i_m = i_{mc} + i_{mu}$.

4.4 Tuning Ray Tracing Parameters

The GPU model is used as a feed forward controller which updates itself iteratively to find the best fitting parameters and then sends those to the actual ray tracer. The error is calculated between the current model estimated frame time and a target reference frame time and ray tracing parameters are adjusted to improve the error. Since the simulated ray tracer and model executes quickly they update several times within one frame of the ray tracer, hence it is possible to recover from sudden changes in view direction. As an example, viewing a plain wall with few node traversals and triangle intersections, the model adapts by increasing the number of AO rays. If the view is turned around 180 degrees to view some more complex geometry the actual frame rate would drop severely, but since the model updates several times before the frame is rendered the number of AO rays are matched so that frame rate stays constant. Ray tracing features that are adjusted to match the reference frame rate are shadow rays and the number of AO rays.

Figure 1 illustrates how the GPU model is utilized to tune the ray tracer as a feed forward controller. The model control loop runs several times per frame and iteratively updates x until the error e, from model output y to reference r, is as small as possible. Changes in x result in switching shadow and AO on or off and adapting the number of AO rays so that a fixed frame rate is maintained.

The feed forward model simulates the ray tracer with some error due to unmodeled behaviours. This error can be greatly reduced

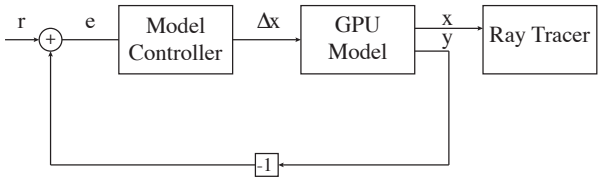

Figure 1: Auto tuning the ray tracer using the GPU model and a feed forward controller.

with help of a slow outer feedback loop. A controller compensates the model by comparing the model predicted execution time with real ray tracer execution time. In figure 2 the inner loop containing the GPU model also receives real execution time y_2 and a feedback controller compensates for the error e_2 between model execution time and real execution time reducing the error from model y_1 to reference r further.

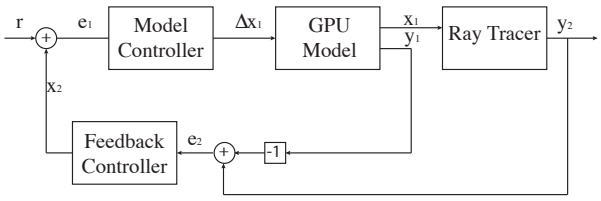

Figure 2: Model errors are improved by introducing a slow outer feed back loop.

5. RESULTS

We auto tune ray tracing parameters for three different GPUs, namely, NVIDIA Geforce 580, NVIDIA Geforce 8800 and the AMD Radeon 5870. We use two data sets, a fairy scene shown in Figure 3 and a cabin scene shown in Figure 4. With our implementation of surface area heuristics BVH the fairy scene contains 174,117 triangles and requires a BVH depth of 28. The cabin scene contains 422,635 triangles and requires a BVH depth of 34. The increased complexity of the cabin scene results in more traversal iterations and triangle intersections resulting in longer rendering times. In particular on the Radeon 5870 the cabin scene results in ray tracing stack nodes spilling from the shader's on-chip registers out to global memory, resulting in slower performance.

Figure 5 shows the frame time results of an animation of 100 frames of our two scenes. The times shown in the graphs are the actual measured frame time (red), the GPU model predicted frame time (green) and the outer feedback loop corrected model time (blue). For each GPU and scene a different reference frame time is set. This reference frame time is user selected and we set it to values to enable a reasonable number of AO rays. Lower reference times are possible, but if the frame time is too low, auto-tuning will switch to the lowest possible settings. The GPU model times follow the curve shape of the actual times accurately, but in some cases with a significant offset. The average model to execution time error for the fairy scene is 6.3, 4.8 and 2.1 percent and for the cabin scene it is 11.1, 12.8, and 19.9, for the three GPUs used. The GPU model alone does reasonable well with the fairy scene, but the error increases with the cabin scene. The offset between frame time and model time is removed when the outer loop feedback is used

Figure 3: Fairy scene.

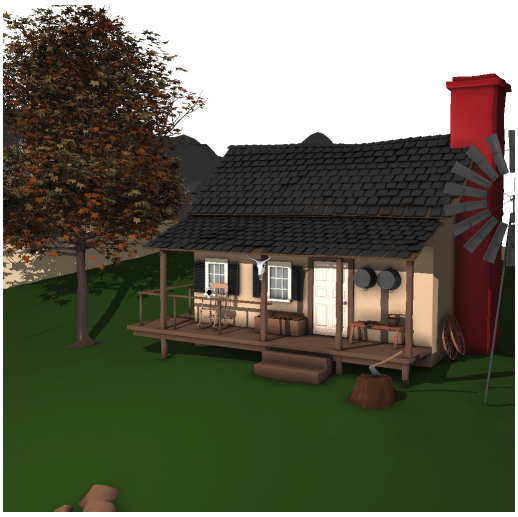

Figure 4: Cabin scene.

resulting in the new 'Model FB' estimated time. The average error of this improved feedback model time compared to the original execution time for the fairy scene is 1.1, 0.7, and 0.2 percent and for the cabin scene it is 1.7, 0.2, and 1.3, for the three GPUs used. Now with the feedback the error is reduced significantly for both scenes.

As a comparison we also ran the animation on both scenes on the GF 580 without any auto-tuning. Before measuring we manually tuned the first frame to execute at 100 ms. These settings were then kept during the animation. For both the fairy and the cabin scene the initial errors from target to real execution time were within 1 percent, however as the animation progressed the errors changed and for the fairy execution time increased and for the cabin execution time decreased. At the last frame both scenes had an error close to 20 percent and the average errors over the animation are for the fairy 10.5 percent and for the cabin 13.2 percent. These measurements are only a comparison for this animation since without auto-tuning the errors can grow arbitrarily large depending on

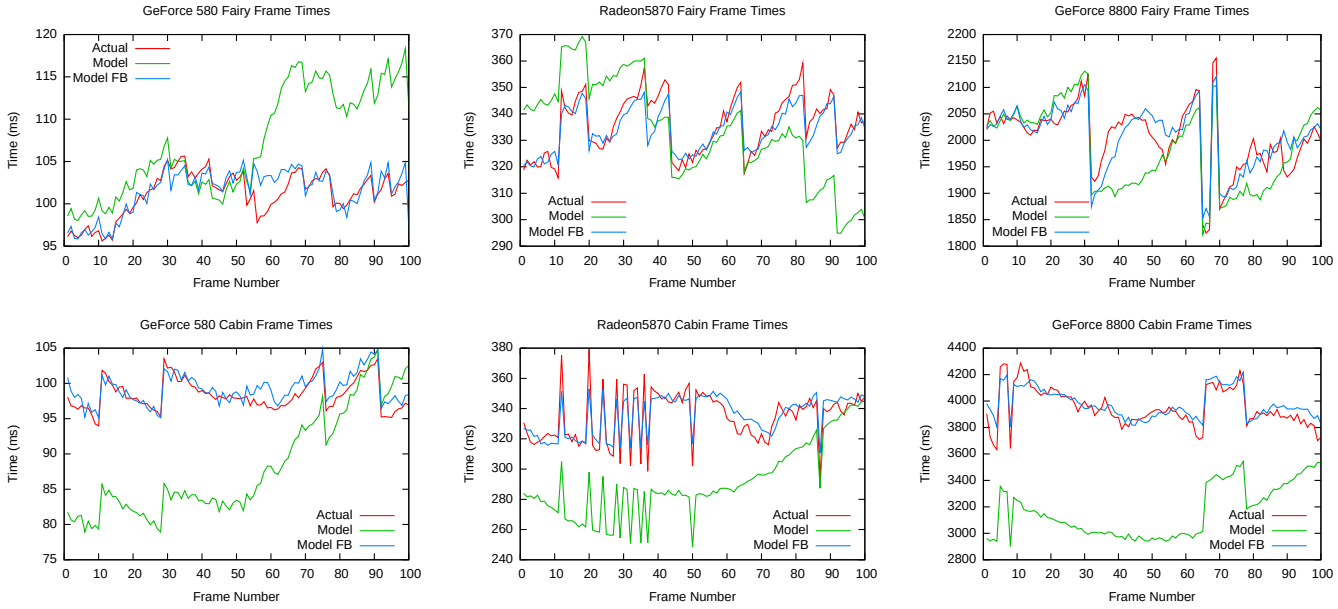

Figure 5: Frame times for different GPUs and scenes for a 100 frame animation. The reference time for the graphs for the fairy scene on the top row are 100ms, 333ms and 2000ms, and for the cabin scene on the bottom row, they are 100ms, 333ms and 4000ms.

the initial view direction and tuning.

Figure 6: Frames from fairy animation.

Three frames from 1st, 50th and 100th frame of the two animations are shown in Figure 6 and Figure 7. Both animations rotate around the scene while moving in towards the center. The cabin animation starts with the tree filling a small part of the scene and finishes with the camera right next to the tree. When rendering the tree, the prediction of the rendering works well, due to the very constant distribution of primitives across the screen.

Figure 7: Frames from cabin animation.

Figure 8 shows the auto-tuned number of AO rays parameter, GPU model to execution time percentage error and the outer loop feedback corrected model percentage error. The number of AO rays is tuned in order to ensure the target frame rate as specfied in Figure 5. The number of AO rays is similar across the different GPUs because the rendering time is determined by the current view point which is the same for each GPU. The GPU model error follows a similar curve for the two scenes even though different GPUs are used, showing that the model works well, but estimating the workload of ray tracing is still challenging for some views. This could be improved by increasing the resolution of our low resolution presampling pass, which on multi-core CPUs would not affect the performance of the GPU rendering time if run in parallel with the previous frame rendering on the GPU. The outer feedback loop corrected time improves upon the original GPU model estimate by removing unmodelled behaviour in the GPU.

6. CONCLUSION

We have used an analytical GPU model to tune a complex application, ray tracing, on a variety of GPU hardware. The model was originally designed only to target NVIDIA GPUs but with its general construct we have managed to estimate performance on AMD GPUs as well.

Using the model and a feed forward controller we have shown that it is possible to estimate GPU workload and to tune a complex application using the workload information. We also introduced a slow outer feedback loop that can be used to improve the GPU models errors by compensating for unmodelled behaviours. Using this approach it is possible to estimate and auto-tune applications with different levels of complexity, given a model of the application's run time.

We believe that our approach should be applicable to other complex compute applications as well. Performance tuning is possible as long as the number of instructions executed can be estimated for the application. Even if the problem space is large with many tuning parameters, the cost of executing the model is low and can easily be executed hundreds or more times per iteration.

Several areas for future work arise from our initial work. More

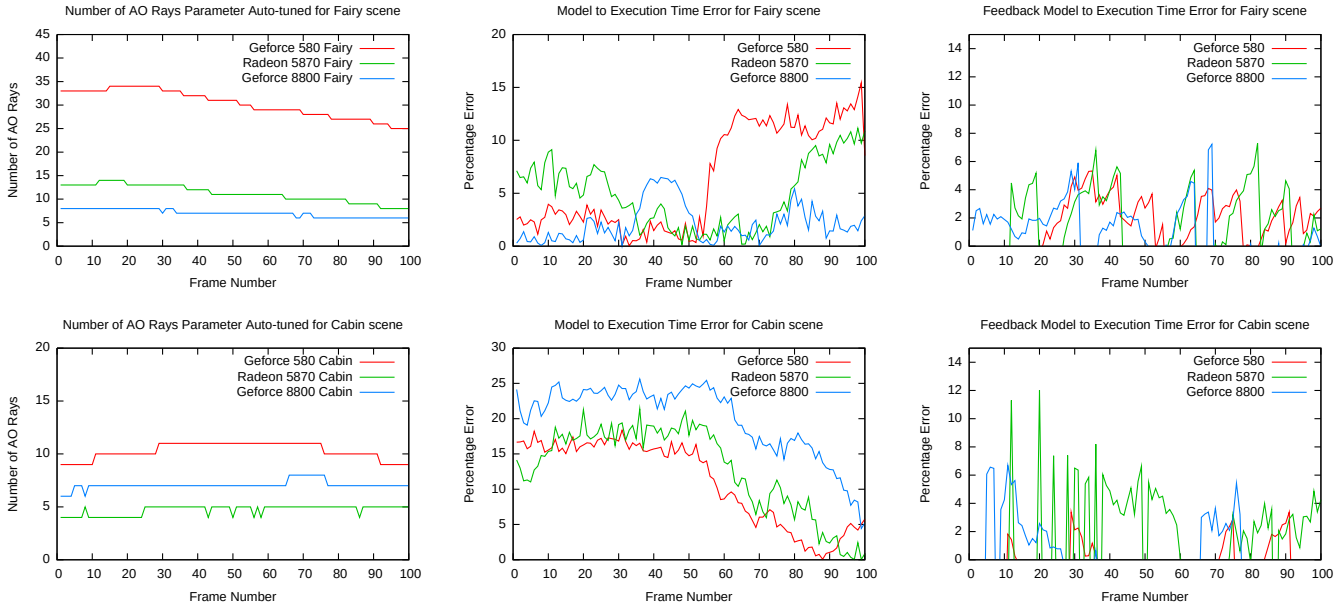

Figure 8: Auto-tuned number of AO rays, and error for different GPUs and scenes over a 100 frame animation. The top row is for the fairy scene and the bottom row for the cabin scene. The two left graphs show the number of AO rays auto-tuned over the animation, the middle two graphs show the percentage error between the GPU model and actual execution time and the right two graphs show the percentage error between the GPU model and the outer loop feedback corrected model time.

ray tracing features such as reflection and refraction can be added to the ray tracer and their parameters auto-tuned. We modelled the Fermi shader cache architecture inside our workload estimates. A more general model of the cache architecture in the GPU model would make it useful for other applications. The GPU model could also be generalised to work with multi-core CPUs and Intel CPU SIMD extensions such as AVX.

Acknowledgements

We acknowledge support from the Intel Visual Computing Institute, Saarbrücken, Germany and the ELLIIT Excellence Center at Linköping-Lund in Information Technology. We thank NVIDIA and AMD for the generous donation of GPUs. Thanks to Andrew Kin Fun Chan and Dan Konieczka for allowing us to use 'The Cabin' model.

7. REFERENCES

[1] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45:105–114, January 2010.

[3] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[4] W. Dally. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation), 2009.

[5] A. Davidson, Y. Zhang, and J. D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[6] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings ACM SIGGRAPH*, pages 247–254, 1993.

[7] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[8] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[9] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.