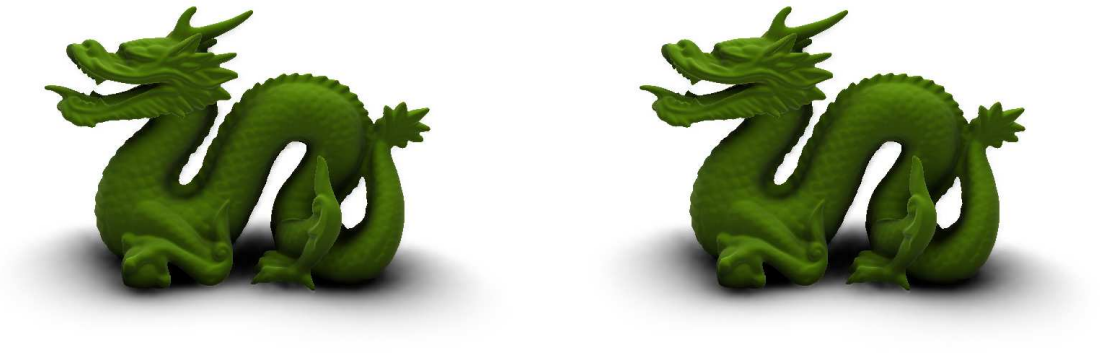
Precision Selection for Energy-Efficient Pixel Shaders

Jeff Pool*
Dept. of Computer Science
University of North Carolina
Chapel Hill, NC 27599

Anselmo Lastra†
Dept. of Computer Science
University of North Carolina
Chapel Hill, NC 27599

Montek Singh‡
Dept. of Computer Science
University of North Carolina
Chapel Hill, NC 27599



(a) Full Precision (b) Reduced Precision

Figure 1: *The image on the left was computed at full floating point precision (24 bits in the mantissa), while the image on the right was computed with an average precision of 12.5 bits in its pixel shaders. There are no perceptible differences between the two images, yet the reduced-precision image saved 71% of the energy in the pixel shader stage's arithmetic, or up to 20% of the GPU's overall energy.*

Abstract

In this work, we seek to realize energy savings in modern pixel shaders by reducing the precision of their arithmetic. We explore three schemes for controlling this reduction. The first is a static analysis technique, which analyzes shader programs to choose precision with guaranteed error bounds. This approach may be too conservative in practice since it cannot take advantage of run-time information, so we also examine two methods that take the actual data values into account - a programmer-directed approach and a closed-loop error-tracking approach, both of which can lead to higher savings. To use this last method, we developed several heuristics to control how the precisions will change over time. We simulate several series of frames from commercial applications to evaluate the performance of these different schemes. The average savings found by the static and dynamic approaches are 31%, 70%, and 62% in the pixel shader's arithmetic, respectively, which could result in as much as a 10-20% savings of the GPU's energy as a whole.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors;

*e-mail: jpool@cs.unc.edu
†e-mail:lastra@cs.unc.edu
‡e-mail:montek@cs.unc.edu

Keywords: variable precision, pixel shader, energy efficient

1 Introduction

Graphics processing units (GPUs) are increasingly limited by their energy consumption. Desktop GPUs combat power delivery and heat dissipation problems, while mobile GPUs contend with milliwatt-scale power budgets and shrinking batteries. We are approaching this problem from the microarchitectural level and advocate using variable-precision computation to lessen the power burden on the GPU's shader units.

This work is part of a larger plan: energy savings through variable-precision computation, communication, and storage. We have already explored precision and energy tradeoffs in the vertex shader [Pool et al. 2008; Pool et al. 2011], and here we take a similar approach to exploring these tradeoffs in the pixel shader. After showing that these two stages are able to save significant energy through reducing their arithmetic precision, we hope to be able to exploit the fact that the data stored in and transferred between these stages have fewer bits. This should permit energy and bandwidth savings in handling the data, both on and off chip.

In this paper, we look at reducing precision in the pixel shader stage because pixel shaders currently perform all their operations with 32 bit floating point numbers, which have 24 bits of precision in the mantissa. However, the final colors are displayed with only 8 to 12 bits of precision in each channel. In this work, we show that it is possible to reduce the precision of the intermediate operations without incurring noticeable differences in the final image, allowing us to use variable-precision hardware to save energy [Pool et al. 2011; Huang and Ercegovac 2002; Lee et al. 2007]. We then explore several approaches to choosing the final operating precision

of the hardware for maximizing energy savings.

The specific contributions we make in this paper are as follows:

- Simulations of modern applications showing the errors induced in their pixel shaders as computational precision is reduced,
- a static analysis technique to determine the lowest safe precision for a given shader program,
- a framework for the application programmer to choose the precisions for the application's shaders concurrently with the design of the shaders themselves,
- a closed-loop system for dynamically monitoring induced errors, thus exploiting knowledge of the actual data to reduce the operating precisions as much as possible, and
- an estimate of the energy savings possible by using variable-precision arithmetic in the pixel shader stage of modern GPUs.

The rest of the paper is organized as follows. Section 2 details related research in the areas of variable-precision rendering and hardware. Section 3 discusses different sources of error in reduced-precision pixel shaders and their implications. Section 4 presents our static and dynamic approaches to controlling the precision used in the pixel shader stage. Our experimental setup and data sets are described in Section 5. Section 6 presents the error and energy savings results, and Section 7 offers our conclusions and future work.

2 Related Research

2.1 Variable-Precision Rendering

Variable-precision rendering was first covered in-depth by Hao and Varshney [2001], who presented a thorough error analysis of the fixed-function vertex transformation and lighting pipelines. Since their work, however, graphics hardware has changed significantly, allowing for these stages to be programmed by the application developer. While their work is accurate, it is no longer current, since pixels can be subject to any arbitrary stream of instructions, not just a fixed series of steps known a priori. We have presented work detailing the errors and possible energy savings seen by recent vertex shader programs [Pool et al. 2008; Pool et al. 2011]. To our knowledge, there has been no work in variable-precision operation in modern pixel shaders.

Others have explored the effects of varying precision in other, specific parts of the pipeline. For example, Akeley and Su [2006] have examined the minimum triangle separation necessary to ensure that surfaces do not incorrectly occlude one another at a given depth buffer precision. Chittamuru et al. [2003] presented work exploring varying the wordlength of texture mapping operations.

2.2 Variable-Precision Hardware

In previous work [Pool et al. 2011], we designed variable-precision integer adders and multipliers and predicted their energy savings in vertex shaders. We now use the energy-saving trends of this hardware to estimate the energy saved by our new explorations in variable-precision pixel shaders, though our designs are not the only efforts in this area. Liu and Furber [2004] presented a low-power multiplier, while Calloway and Schwartzlander [1997] detailed the relationship between power and operand size in CMOS multipliers. Tong et al. [2000] suggested a digit-serial multiplier with three discrete operating precisions, resulting in a linear power savings. Lee et al. [2009] proposed a variable-precision constant multiplier that replaces multiplication by powers of two with a simpler shift operation, realizing a savings between 16% and 56%. Huang and Ercegovac [2002] developed circuits that are most similar to ours, with two-dimensional signal gating for variable bit-width multipliers; they achieved savings of up to 66% over a baseline implementation.

Our approach, in contrast with others, offers fine-grained precision control *and* power gating, which will save leakage as well as dynamic power due to our use of sleep transistors to eliminate power delivery to unused circuitry. Table 1 gives a summary of our circuit overheads. A full discussion of their energy savings is beyond the scope of this work, but in short, the energy savings are approximately linear for adders: compared to full 24-bit computations, an 18-bit addition saves 26% of the energy and a 12-bit addition saves 51% of the energy. For multiplications, on the other hand, the savings are approximately quadratic: 18-bit multiplications save 67% and 12-bit multiplications save 87% of the energy compared to 24-bit multiplications. Leakage power is also reduced: halving the precision of an addition or multiplication saves approximately 12% or 42% of the leakage power, respectively. Please refer to our past work [Pool et al. 2011] for a detailed discussion on each of our designs.

Table 1: *Variable-Precision Circuit Savings and Overheads*

Circuit	Area Overhead	Delay Penalty
Adders		
Ripple-Carry	16.9%	6.9%
Carry-Select	12.6%	6.9%
Brent-Kung	9.3%	0.4%
Multipliers		
Single Gating	34%	3.8%
Double Gating	34%	3.8%
Column Gating	16%	3.8%

3 Precision-Induced Errors in the Pixel Shader

The various operations in modern pixel shaders have very different sensitivities to precision reduction. Arithmetic instructions will give a result whose imprecision can be statically determined by the imprecision of the operands. The results of other instructions, such as texture fetches, can be much more sensitive to variations in input precision. So, maximum energy savings will be seen only with control over the precision of these two groups of instructions independently, when neither precision will limit the other. It is this observation that makes the problem of controlling the precision used in a pixel shader more complicated than might be immmediately apparent. Figure 2 demonstrates these two types of errors. Here, we discuss these characteristics so that we can take them into account in our algorithmic and experimental sections.

3.1 Texture Fetches

Texture fetches behave very differently from arithmetic instructions, since texture coordinates are effectively indices into an array. Using slightly incorrect indices to index an array can lead to results that are very wrong, correct, or anywhere in between. The behavior is dependent on such parameters as the frequency of the texture data, size of the texture (or mip level accessed), and type of filtering used - information that may only be available at run time. Reduced precision texture coordinates will lead to neighboring pixels fetching the same texel. In some pathological cases, texture coordinates

(a) Full Precision

(b) Reduced Precision Texture Fetches

(c) Reduced Precision Color Computations

Figure 2: *Figure 2(a) is the reference frame produced by full-precision computation (24 bits) throughout the pixel shader. Figure 2(b) shows an exaggerated result due to reducing the precision of texture coordinates to 8 bits, and Figure 2(c) shows similarly exaggerated results of reducing the precision of color computations to 6 bits. Errors of this magnitude are never seen in our test applications when using our techniques to select precisions; these images are to demonstrate the types of errors that are possible.*

for entire triangles may collapse to the same value when using a slightly reduced precision, giving the triangle a single color.

3.2 Arithmetic Operations

The errors that arise in simple arithmetic operations (*add, mul, div,* etc.) are quantifiable, and a discussion of these errors is readily available [Wilkinson 1959]. For complex operations, such as *rsq, sin/cos,* etc., the errors incurred will depend upon the implementation. We assume that these operations have an error bound of no greater than one unit in the lowest place. With these error characteristics, we are able to apply our static analysis technique to the instructions in shader programs that do not contribute to a value used as a texture coordinate. Arithmetic imprecisions generally manifest themselves in the computation of color values in two ways: they gently darken the scene overall as LSBs are dropped, and smooth color gradients can appear blocky as nearby values are quantized to the same result.

4 Precision Selection Methods

4.1 Static Program Analysis

A static analysis of the shader programs used by an application will determine reduced precisions with guaranteed error bounds. Our approach is to build a dependency graph for the final output value and to propagate the acceptable output error back towards the beginning of the shader program. This procedure yields a conservative estimate of the precision for each instruction. As we noted above, though, the error characteristics of texture fetch instructions are non-linear and impossible to predict without knowledge of the data stored in the textures in use. In the worst case, reducing the precision of a texture coordinate by a single bit could cause an unbounded error in the resulting value. For this reason, we are not able to safely change the precision of instructions that modify texture coordinates. The output of our static analysis, then, is a single precision for each shader program which will be applied to each instruction after the program's last texture fetch.

Determining the last texture fetch is not always straightforward; for instance, multi-phase shaders may rely on complex control structures to repeat texture fetch loops. In this case, a dependency graph is constructed at the shader's compilation, rather than the simpler approach we have taken for this work of simply noting the position of the last texture fetch and applying a different precision to subsequent instructions. If the control structures modify texture coordinates, this information would be captured in the dependency graph and used to choose a precision.

Just as a static analysis will not have access to the texture data in use, it will also not have access to the rest of the fragment's data - position, color, normal information, etc. We can handle this restriction more effectively, however, by assuming the worst-case error for each arithmetic instruction. This will cause overly conservative estimates in most cases, but the error is guaranteed to be within the local tolerance.

4.2 Dynamic Programmer-Directed Selection

Our static analysis assumes the worst-case inputs, which may cause the final chosen precision to be too conservative, leaving unclaimed energy savings. Similarly, it is impossible to determine a safe reduced operating precision for computations affecting texture coordinates with a static method, while a dynamic approach will be able to monitor errors while reducing the precision of these computations, saving more energy. So, we propose a simple scheme to allow the application's developer to control the precision of each shader effect in tandem with the effect's development. This will allow the developer to stipulate that certain shaders can tolerate large reductions in precision without noticeable degradation; here, of course, the developer is able to decide what is noticeable on a case by case basis.

Currently, most pixel shaders are developed inside a dedicated shader editor. This allows artists to tweak certain parameters and see the results in real time. With hardware support for variable-precision arithmetic, two extra parameters (precisions before and after the last texture fetch) for each shader program will be a natural addition to the artist's array of variables. Once the shader is finalized, the chosen precisions can be encoded either as constants in the program or as instructions, depending on the implementation. Alternatively, the precision for each stage of the program could be encoded with the current rendering state. This way, context switching would automatically handle loading and storing correct precisions.

In the extreme case, the programmer could have control over the precision of each instruction independently. This would allow for more savings in the arithmetic but would carry with it a higher control cost: either encoding a precision in each floating-point instruction or using many more precision setting instructions. It would be up to the programmer to evaluate the tradeoffs in their particular application, though we do not think this level of fine-tuning will be necessary.

4.3 Automatic Closed-Loop Selection

To remove any burden from the application's developer, we have developed a closed-loop method by which the actual errors can be monitored as precision is reduced at runtime. At the highest level, when the error is larger than a given threshold, the precision is increased to avoid continued errors. In this section, we describe an efficient method to monitor runtime errors and change the precisions of individual shader programs at the driver level.

4.3.1 Monitoring Errors

In order to determine that the current operating precision for a shader program is either too high or too low, the error between the shader's output (commonly in an 8-bit per channel format) at the current precision must be compared to the output at full precision. To do this, the hardware must compute both the reduced-precision result of the shader as well as the full-precision result. The difference between these two will give the error of the shader at its current precision. There are different ways of implementing this process in hardware, but we will first show that it is a viable method of energy saving. We save a discussion of one possible implementation for Section 7.1. We note here that these errors will be monitored regularly throughout each frame, and that the reaction to this monitoring does not need to wait until the next frame. If the precision is updated mid-frame, then the response time will be quite short; it is unlikely that errors will persist for more than a single frame. In all of our simulations, we did not see any multi-frame errors.

4.3.2 Sampling Generated Fragments

Clearly, this method will not save any energy if each pixel of every frame is computed twice - the overhead in this case would be 100%! Rather, the redundant execution should be predicated on some flag; this flag could be anything from a randomly-selected boolean input assigned by the rasterizer, to a value obtained from hashing the input fragment's position at the start of the shader. The method chosen will depend on many factors, but it must be able to flag a subset of fragments for error determination. Ideally, this subset will be as small as possible, leading to a very small incurred overhead. In our experiments, we have explored varying both the sampling rate and sampling pattern. What we found is very promising - sparsely sampling every nth generated fragment performs nearly as well as denser random sampling. See more on these results in Section 6.2.

4.3.3 Precision Control

With an accurate measure of the error caused by the precision reduction of a particular pixel shader, we must now determine how to change the operating precision, if at all. We expect that, due to differences in the responses of texture fetches and regular arithmetic to reducing precision, we will see different minimum precisions for the two phases (texture fetches and color computation) within a single shader. Some texture coordinates must be computed with high precisions, yet the results of their corresponding fetches are subject to a series of operations that can be performed at lower precisions. Other texture coordinates, however, can tolerate low precisions, which is one of the advantages of our dynamic approach. So, we store two precisions for each active shader: one used prior to the last texture fetch (LTF) that will control the precision of any computed texture coordinates, and one used after the last texture fetch, which will incur predictable arithmetic errors.

One complication with this dual-precision approach is that when both are reduced, it can become difficult to correctly determine which precision is the source of an error in the final pixel value. We examine several heuristics for controlling these precisions: a "simple" approach that merely acts on an "error detected" signal, a modified "simple with delay" approach that adds a configurable latency with the goal of determining the source of the error more accurately, a "texture fetch priority" approach that acts on the magnitude of the error, and two "dual test" approaches that attempt to determine precisely the source of the error at the expense of higher overheads. For all these approaches, the precisions of each shader are initially reduced in the same manner. First, the post-LTF precision is reduced at the rate of one bit per frame until an error is seen, after which it is immediately raised by a single bit. Then, the pre-LTF precision is reduced at the same rate until an error is seen. The behavior at this point is dictated by the control system in use. We describe the five we explored below.

4.3.3.1 "Simple" Control This approach does not make any attempt at determining which precision is the source of the error. Rather, it increases the pre-LTF precision until there is no error above the tolerance threshold with the assumption that it was this precision that caused the error. Once the pre-LTF precision is at its maximum value (of 24), the post-LTF precision is increased if any errors over the tolerance are measured. The overheads for this approach are minimal, just an extra pass for the full-precision results and some control logic in the driver.

4.3.3.2 "Simple with Delay" Control As with the "simple" control, the post-LTF precision is reduced until an error is seen. However, there is then a configurable delay period of some number of frames added before decreasing the pre-LTF precision. This

will allow for more error sampling with only one source of error, so that we can be more confident that the chosen precision setting for the post-LTF instructions is sufficiently high. If another error is seen during this period, it restarts. After this point, this technique is identical to the "simple" approach, and it has similar overheads, with only slightly more storage necessary for the remaining time in the assigned delay period. In our explorations, we chose to add ten frames of delay.

4.3.3.3 "Texture Fetch Priority" Control An error's magnitude may hold a clue to its source. Since imprecise texture coordinates *can* lead to very incorrect results, if a large error is encountered during runtime, it is likely due to the pre-LTF precision being too low. A simple arithmetic error is unlikely to cause a very large error rapidly. So, when an error is seen, we take its magnitude into account and increase the pre-LTF precision if the error is high. However, we cannot assume that a low error indicates arithmetic imprecision, since low-frequency texture data will also lead to relatively small errors. In this case, we fall back to the "simple" controller. The overhead for this technique is only slightly higher than the "simple" control due to slightly more complicated control logic.

4.3.3.4 "Dual Test" Control It is possible to diagnose which precision caused an error by performing the computations again with one of the two precisions, pre- and post-LTF, set to 24 bits. Performing the computations yet again with the other precision at 24 will make it likely that the culprit will be accurately determined. Whichever instruction group's pass causes the lesser error will be the group to have its precision raised. This approach (and the next) incurs the highest overhead, at more than 3 times that of the simplest approach due to two extra passes and more control logic.

4.3.3.5 "Dual Test with Gradient Climb" Control A variant of the "dual test" control, this approach simply steps either precision up by a single bit, giving us the local gradient of the errors with respect to precision. We expect this method to perform better than plain "dual test" because it predicts the effects of performing the eventual action, rather than the effects of maximizing the precision.

4.4 Local Shader Errors vs. Final Image Errors

Both the static and automatic approaches give us reliable access to the *local* errors at each pixel shader; however, these errors do not necessarily correspond to the errors in the final image presented to the viewer. For example, in a scene with a car driving through a city, an environment map will be generated to show the reflections on the car's surface. This environment map may have slight errors from the reduced precision in its pixel shader. When the map is sampled during the car's draw call, further errors may be imparted on the same pixels. If this generated image is then used as input to a post-processing shader, more errors may be compounded upon the preexisting errors.

We find that limiting the tolerance of the local errors is sufficient to limit the noticeability of differences in the final image, despite two discouraging observations. The first is that it is impossible to relate the local errors in each shader stage to the errors in the final image. The second is that it is impossible to sparsely sample errors in the final image effectively. This is due to a final pixel of position (x,y) being composed of several other pixels with varying positions, not necessarily (x,y), in their own render targets and textures; predicting these positions to sample during program execution is infeasible. However, these shortcomings do not make these local error limiting approaches ineffective, as we will show in Section 6.3.

Only our programmer-directed dynamic approach allows for consideration of the final image errors. So, this approach will better bound the final errors, which may be necessary in some circumstances when the local errors do not predict the final errors well. However, this benefit comes at the cost of extra work on the part of the programmer or artist.

5 Experimental Setup

Our static analysis requires no simulation or user-intervention, we simply require data sets to analyze. To examine our programmer-directed approach, we modify several simple applications meant to demonstrate a single shader effect. This allows us to demonstrate the ease of use of this approach as well as determine the tolerance to precision reduction of cutting-edge pixel shaders. Our automatic approach will require a simulation environment to test our control schemes.

5.1 Programmer-Directed

We adapted several demo applications developed by NVIDIA and AMD (see Section 5.3.1) to show how they may appear to an artist developing an application for use on variable-precision hardware. Ideally, the artist or programmer will have control over two precisions per shader, as discussed in Section 3. However, modern applications are written in a higher-level language, and we do not have access to the compiled assembly program. This makes it difficult to divide the operations into two groups representing instructions before and after the last texture fetch at runtime of a live application (rather than a simulation). So, our programmer-directed approach will only make use of one precision per shader. This is still enough to prove our concept, though, and we still see significant savings, even with this limitation (see Section 6.4). Lastly, this simplification precludes us from determining the errors introduced by the precisions chosen by a static analysis; we are still able to present energy estimates for our static analysis, however.

5.2 Simulator

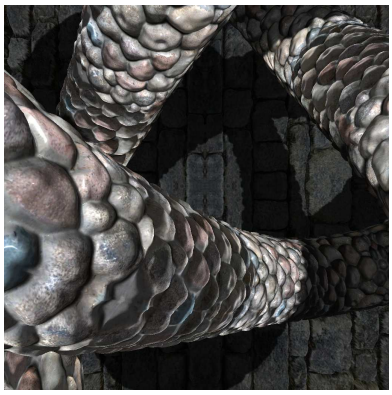
We chose to use the ATTILA simulator [del Barrio et al. 2006] to test our reduced-precision pixel shader approaches. It has been used successfully in the past to explore similar techniques in the vertex shader [Pool et al. 2008], and its designers have recently released a version that can use traces of DirectX 9 applications captured by Microsoft's PIX tool [Microsoft 2011]. This allows us to experiment on recent applications with modern pixel shaders.

We modified ATTILA in several ways. First, we added support for variable-precision arithmetic to the GPU's emulated arithmetic hardware. This allows us to specify a single precision for the entire simulation. Next, we implemented independent precisions per shader, as well as dual-precisions for before and after the last texture fetch of each shader. Finally, we added support for our various precision control techniques in order to see how each behaved.

5.3 Data Sets

Our static approach uses the data sets from both the other approaches, since the shader programs require, at most, a simple translation into a common format for analysis. The dynamic approaches will require their own data sets that are compatible with their associated simulation environments. We list these in detail below.

(a) Depth of Field



(b) Parallax Mapping

(c) Screen-Space Ambient Occlusion

Figure 3: *Data sets used to test our developer-driven dynamic precision control techniques.*

(a) Half-Life 2: Lost Coast (1), 80 frames

(b) Half-Life 2: Lost Coast (2), 80 frames

(c) Doom 3, 250 frames

(d) Need for Speed: Undercover, 63 frames

(e) Metaballs, 2000 frames

Figure 4: *Data sets used to test our closed-loop precision control techniques.*

5.3.1 Programmer-Directed Approach

We examine three recent pixel shader effects in the context of a shader editor that an artist might use. These effects are shown in Figure 3: depth of field [AMD 2008], parallax mapping [NVIDIA Corporation 2010], and screen-space ambient occlusion [NVIDIA Corporation 2010].

5.3.2 Automatic Approach

The ATTILA designers have released a number of traces for use with their simulator [ATTILA 2011]. We use some of these traces, as well as some that we have captured, to evaluate our techniques. The specific applications we used are, as seen in Figure 4, two scenes from "Half-Life 2: Lost Coast" [Valve 2005], "Doom 3" [id 2005], "Need for Speed: Undercover" [EA Black Box 2008], and a Metaball viewer [Baker 2011].

6 Results

In this section, we present the results of our experiments. First, we show the errors and energy savings that result from our static analysis technique. We then compare these results with our two dynamic approaches. Note: all errors reported are per-component (R,G,B) per pixel, for both average and PSNR results.

6.1 Static Analysis Results

Our static approach assumes a full precision of 24 bits of mantissa for every instruction before the last texture fetch in each shader, but determines the lowest safe precision for an error tolerance of 1

out of 255 per channel in the final output for the remaining instructions. Table 2 shows the average precision for each of our sample applications.

Table 2: *Statically Determined Precisions*

Scene	Precision
HL2LC(1)	19.2
HL2LC(2)	19.0
Doom 3	19.7
NFS	21.8
Metaballs	9.7
SSAO	20.1
Parallax	23.3
DoF	18.5

6.2 Dynamic Analysis Results

There are several dimensions in which we can vary our dynamic approach: sampling frequency, sampling pattern, local error threshold, and control method. In this section, we discuss how each of these will change the final output and finally choose an optimum set of parameters that maximize energy savings and minimize errors. We present our results of exploring the first three parameters for the HL2LC(2) scene; other data sets gave similar results. In the sampling rate and type explorations, dynamic precision control is in use; so, the precision will be changed as required by the control algorithm which will lead to slightly different precision streams per curve in the graphs. However, we still see strong trends in the results for each of the data sets.

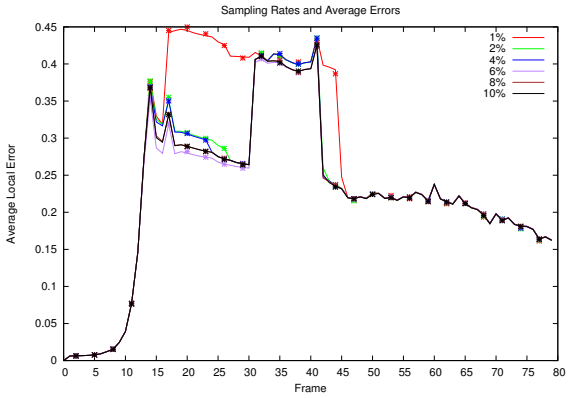


Figure 5: *Various sampling rates give different approximations of the global error trends. Sampling more shaded pixels will respond to errors more quickly but carry the cost of expending more energy and time. Sampling fewer pixels will respond more slowly with less overhead. However, regardless of the sampling frequencies we use (between 1 and 10%; any more would carry too high an overhead), the average of the sampled errors (overlaid points) agrees closely with the global averages (lines).*

A dynamic analysis requires that errors be monitored at runtime. This will incur some overhead when pixel shaders are executed twice, at both full and reduced precisions. However, this overhead need not be prohibitively high; we show that sampling a small subset of the shaded pixels can give an accurate approximation of the global error. The overhead, then, will be roughly equal to the fraction of the total pixels that are sampled. Figure 5 shows how different sampling rates lead to global error approximations. Further, Figure 6 shows that simply sampling every *n*th generated fragment

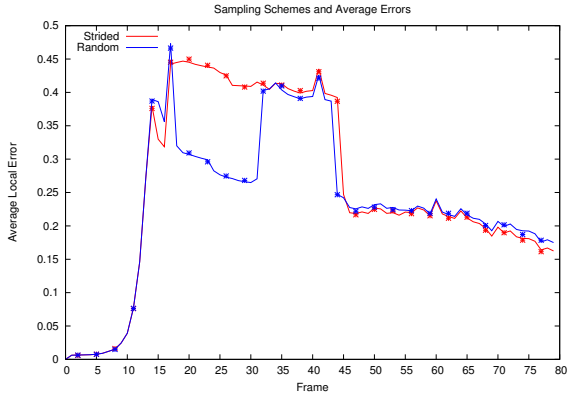


Figure 6: *Sampling every nth fragment ("strided" sampling) performs nearly as well as to random sampling (both at rate of 1%). Therefore, we use the simpler sampling scheme in our final automatic system.*

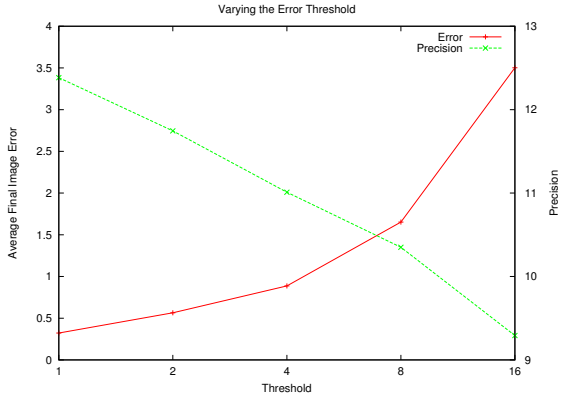


Figure 7: *Error thresholds greater than 1 out of 255 do not give significant precision savings to warrant their higher errors.*

is just as effective as sampling errors randomly. Neither of these sampling approaches have the potential drawbacks that a screen-space based pattern might, such as needing to reconfigure the sampling pattern every frame. Finally, Figure 7 (generated from the statistics of the final frame of the HL2LC(2) scene) shows that increasing the local error threshold significantly increases the final image error, but does not yield equivalent precision savings.

Figure 8 shows how the precisions change for each application as the traces progress. The average precision used in each application decreases initially as the precision is lowered without any above-threshold errors. Next, the precision curve will level out as the precisions are held constant due to errors seen in the data stream. After this, more errors may be seen, causing the precisions to rise slightly. Finally, each curve may fluctuate due to changes in precision distributions; different workloads will lead to different shaders (and their respective precision selections) performing different fractions of the overall work. At this point, our control algorithms do not consider lowering the precision after it has been raised due to an unnacceptable error, so the final decrease in the precision of the HL2LC(2) scene is due to a different workload. However, this type of extension to our control schemes is natural and straightforward, since a new workload may be tolerant to lower precisions than those seen previously.

We see in Figure 9 that the precision reductions are greater with

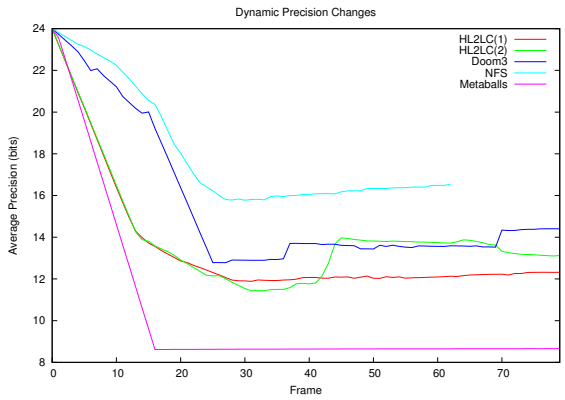


Figure 8: *As each program progresses, dynamic precision selection will change the average precision used by the program.*

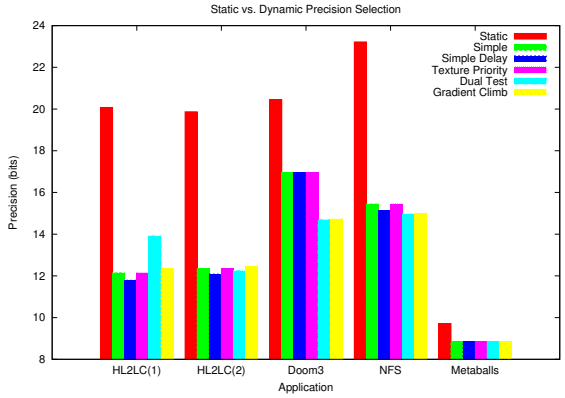


Figure 9: *A simple precision control scheme performs, on average, as well as a more complex error-tracking scheme, while using significantly less control logic and execution overhead. The "Metaballs" data set's shaders did not have any instructions before the last texture fetch, so the results for all control schemes are identical.*

our dynamic approach than with our static approach. This is for two reasons: first, the actual data are used, rather than worst-case operands; and second, we are able to vary the precisions of operations both before and after the last texture fetch. Another important observation about our automatic approach is that the simpler control methods perform very competitively with the more complex methods. As expected, our "simple with delay" approach performs better than our "simple" approach in most cases, but never worse. Our "texture priority" approach seems to perform no better - this is likely because when it reacts to a large error by raising the pre-LTF precision, the "simple" algorithm makes the same decision for different reasons; it was simply the pre-LTF precision that was being increased at that point.

The relative performance of our "dual-test" heuristics is not consistent; in the HL2LC(1) scene, they both perform worse than the simpler algorithms. However, in Doom 3, they perform much better. This is also the one test scene in which our "simple with delay" approach does no better than the plain "simple" approach. Both these facts can be attributed to the following observation: errors seen late in the application were due to arithmetic imprecision, not texture fetches. This indicates that the delay added in the "simple with delay" approach was not of sufficient length to allow for these errors to manifest themselves before the pre-LTF precision was decreased. So, the complex approaches, the "dual-test" methods, were able to take advantage of this inherent shortcoming in the "simple" approaches. The straightforward way to combat this is to enforce a longer delay and allow for sampling more data. While this would mean a longer settling time, it could still be on the order of a few seconds. This period of time's slightly higher energy (due to the pre-LTF precision not yet having been reduced) would likely be dwarfed by the savings seen in the lifetime of the entire scene.

So, we recommend the following for use in a dynamic error-monitoring system: sampling every 100th generated fragment (both sparse and regular), a minimal error threshold (1 out of 255), and a simple control method. This combination of settings gives low final image errors with minimal overheads and acceptable response time. It is this combination that we use to generate our error and energy results in Sections 6.3 and 6.4.

6.3 Overall Errors

In Section 4.4, we observed that the measured local errors in each shader do not correspond to the errors seen in the final image. Here, we offer evidence that supports the feasibility of our static and automatic approaches despite this shortcoming. First, we simulated over sixty frames of each data set and were not able to tell any difference between the reduced- and full-precision frames. Furthermore, there were no temporal effects observed from the gradual reduction in precision in our automatic selection method. Finally, we quantified the errors by measuring their peak signal-to-noise ratios, presented in Figure 10. The steady-state values are all above 40dB. Similarly, we quantify the errors seen in our programmer-directed approach in Table 3, which all have similarly high values.

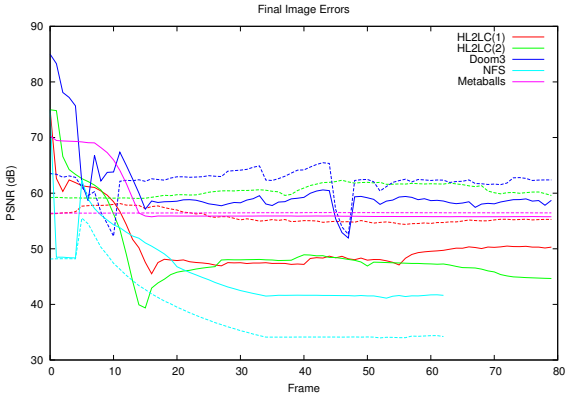


Figure 10: *With both a 1% strided sampling scheme and a local error threshold of 1 out of 255 for the closed-loop system (solid lines) and our static technique (dashed lines), the errors for each of the data sets are not noticeable. This indicates that a low local error threshold is sufficient to limit final image errors to unnoticeable levels in modern applications.*

6.4 Energy Savings

We now present the predicted energy savings in the arithmetic of the pixel shader and its contribution to the GPU's savings as a whole. As indicated in Section 2.2, we use the energy saving characteristics of our variable-precision circuits [Pool et al. 2011] to estimate the energy saved by the pixel shader's arithmetic. For our work, we consider the "pixel shader's arithmetic" to be only the actual computation performed in the ALU; instruction and data fetching, as well as control logic, are not counted in this number. In order to translate this local savings into the context of the GPU as a

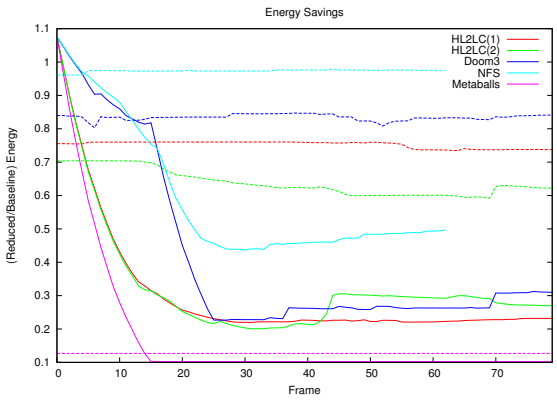


Figure 11: *Energy savings achievable with variable-precision hardware. The solid line represents the savings seen by our automatic approach, while the corresponding dashed lines are the savings seen by our static approach. In each case, the dynamic approach saves more energy.*

whole, we use work deriving from our GPU energy model [Pool et al. 2010] which shows that an average of 33% of the energy in the GPU is spent in the pixel shader's arithmetic.

To be clear, our savings only apply to the shader's arithmetic circuitry; we do not yet see savings in transmitting or storing data in memories, register files, etc., or in control logic. Control logic, though, will be amortized over some number of ALUs. Savings due to reading and writing reduced-precision data is now an important topic since we have shown that such data is usable in modern applications. However, our results must be seen in this context. Since our baseline energies came from actual measurements of hardware [Pool et al. 2010], we can be confident in their relative magnitudes. We were not able to differentiate between the energy of performing arithmetic and reading and writing operands, though, so we have presented our findings at the finest granularity possible. In the future, we hope to determine the relative costs of arithmetic and register file accesses.

In each case, our dynamic approaches outperform a static analysis in terms of energy savings. In Figure 11, we see that our dynamic approaches save, after settling on final operating precisions, roughly 71% of the energy in the pixel shader's arithmetic. Our static approach, on the other hand, has very limited energy savings, 31% on average, except in the simplest of cases - the Metaballs application. We see similar results for our programmer-directed approach, shown in Table 3, including one case in which the static approach led to higher energy consumption than the baseline, due to the slightly less efficient variable-precision hardware having to operate at full precision for a majority of the operations.

Table 3: *Programmer-Directed Errors and Energy Savings*

	Directed			Static	
Scene	Precision	PSNR	Savings	Precision	Savings
SSAO	13.0	53.4	71%	20.1	49%
Parallax	15.2	39.7	61%	23.3	-2%
DoF	12.0	45.6	79%	18.5	33%

7 Conclusion

We have presented a method of saving energy in the pixel shader stage of modern GPUs through the use of variable-precision arithmetic. We first analyze the shaders used by several graphics applications to determine statically a safe reduced precision. Then, we develop two dynamic precision determination schemes - the first directed by the application programmer, the second an automatic error-monitoring approach. When considering the metrics of energy savings, ease of implementation, and ease of use, there is no clear winner. In Table 4, we summarize the strengths and weaknesses of each. However, when we also consider that our static and automatic approaches cannot take the error in the final image into account, the programmer-directed approach begins to pull ahead. It is able to save up to 79% of the energy in the pixel shader stage's arithmetic, or up to 20% of the overall GPU energy without incurring any errors that the application's programmer or artist deemed unacceptable. The hardware overhead for this method is only that involved in the variable-precision hardware itself, and no runtime monitoring or control is necessary.

7.1 Hardware Implementation of Feedback Control

We briefly discuss a possible hardware implementation for our closed-loop feedback controllers. First, we recommend redundant hardware to sample and calculate the full-precision results of each shader. This will most easily fit into some level of the grouping of ALUs on a GPU; an extra full-precision ALU per group of 32, 64, or 128 (depending on the architecture) that mirrors the operation of the "last" ALU in the group will provide sufficient sampling. This ALU need not be modified for variable-precision operation. It will fetch the same data as the ALU it mirrors, so it will not see significant energy or latency overheads due to needing different data.

The difference between the full- and reduced-precision ALU results could be calculated by dedicated hardware, which will then store the necessary information (just a bit flag indicating an error was detected in the case of our "simple" controllers) in a specific memory location for the driver to query. The driver will decide how to handle this flag based on the current state of the control system.

7.2 Future Work

Having shown that the pixel shader stage can save up to 20% in this work and that the vertex shader stage can save up to 10-15% of the GPU's energy by using our proposed variable-precision circuits [Pool et al. 2011] and control methods, we can now turn to producing even more savings in other areas of the pipeline. With pixel shader outputs needing fewer bits to represent equivalent colors, there is no need to transmit and store these bits, either on- or off-chip. So, we will look into static and dynamic RAMs, data buses, and register files (each a major part of a GPU's data path) to determine the possible energy savings.

Acknowledgements

Financial support was provided by the National Science Foundation under grant CCF-0702712. Equipment was provided by NSF Research Infrastructure grant number 0303590.

Table 4: *Strengths and Weaknesses*

Approach	Savings	HW Cost	User Effort
Static	Low	Low	Low
Directed	High	Low	Medium
Automatic	High	High	Low

References

AKELEY, K., AND SU, J. 2006. Minimum triangle separation for correct z-buffer occlusion. In *21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ACM, New York, NY, USA, GH '06, 27–30.

AMD, 2008. Rendermonkey 1.82. `http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx`, Dec.

ATTILA, 2011. Traces - AttilaWiki. `http://attila.ac.upc.edu/traceList/`, Accessed: April 2011.

BAKER, P., 2011. Metaballs II. `http://www.paulsprojects.net/metaballs2/metaballs2.html`, Apr.

CALLAWAY, T., AND SWARTZLANDER, E.E., J. 1997. Power-delay characteristics of CMOS multipliers. In *13th IEEE Symposium on Computer Arithmetic, 1997*, 26 –32.

CHITTAMURU, J., BURLESON, W., AND EUH, J. 2003. Dynamic wordlength variation for low-power 3D graphics texture mapping. In *2003 IEEE Workshop on Signal Processing Systems*, SIPS '03, 251 – 256.

DEL BARRIO, V., GONZALEZ, C., ROCA, J., FERNANDEZ, A., AND ESPASA, R. 2006. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '06, 231 – 241.

EA BLACK BOX, 2008. Need for Speed: Undercover. `http://undercover.needforspeed.com/home.action`, Nov.

HAO, X., AND VARSHNEY, A. 2001. Variable-precision rendering. In *2001 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, I3D '01, 149–158.

HUANG, Z., AND ERCEGOVAC, M. 2002. Two-dimensional signal gating for low-power array multiplier design. In *IEEE International Symposium on Circuits and Systems*, ISCAS '02, 489 – 492.

ID, 2005. Doom 3. `http://idsoftware.com/games/doom/doom3/index.php`, Apr.

LEE, Y., PARK, J., AND CHUNG, K. 2007. Design of low power MAC operator with dual precision mode. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, 309 –318.

LEE, Y., JUNG, H., AND CHUNG, K. 2009. Low power MAC design with variable precision support. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science 92-A*, 7, 1623–1632.

LIU, Y., AND FURBER, S. 2004. The design of a low power asynchronous multiplier. In *2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, 301 – 306.

MICROSOFT, 2011. PIX. `http://msdn.microsoft.com/en-us/library/ee417062(v=VS.85).aspx`, Apr.

NVIDIA CORPORATION, 2010. NVIDIA Direct3D SDK 10 Code Samples. `http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html`, Jun.

POOL, J., LASTRA, A., AND SINGH, M. 2008. Energy-precision tradeoffs in mobile graphics processing units. In *2008 IEEE International Conference on Computer Design*, ICCD '08, 60 –67.

POOL, J., LASTRA, A., AND SINGH, M. 2010. An energy model for graphics processing units. In *2010 IEEE International Conference on Computer Design*, ICCD '10, 409 –416.

POOL, J., LASTRA, A., AND SINGH, M. 2011. Power-gated arithmetic circuits for energy-precision tradeoffs in mobile graphics processing units. *Journal of Low Power Electronics 7*, 2 (April), 148–162.

TONG, J., NAGLE, D., AND RUTENBAR, R. 2000. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8*, 3 (jun), 273 –286.

VALVE, 2005. Half-Life 2: Lost Coast. `http://store.steampowered.com/app/340`, Oct.

WILKINSON, J. H. 1959. Rounding errors in algebraic processes. In *1959 International Conference on Information Processing*, 44–53.