Energy-Precision Tradeoffs in the Graphics Pipeline

Jeff Pool

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2012

Approved by:

Anselmo Lastra

Montek Singh

Dinesh Manocha

Steve Molnar

John Poulton

ABSTRACT

JEFF POOL: Energy-Precision Tradeoffs in the Graphics Pipeline.
(Under the direction of Anselmo Lastra and Montek Singh.)

The energy consumption of a graphics processing unit (GPU) is an important factor in its design, whether for a server, desktop, or mobile device. Mobile products, such as smart phones, tablets, and laptop computers, rely on batteries to function; the less the demand for power is on these batteries, the longer they will last before needing to be recharged. GPUs used in servers and desktops, while not dependent on a battery for operation, are still limited by the efficiency of power supplies and heat dissipation techniques. In this dissertation, I propose to lower the energy consumption of GPUs by reducing the precision of floating-point arithmetic in the graphics pipeline and the data sent and stored on- and off-chip.

The key idea behind this work is twofold: energy can be saved through a systematic and targeted reduction in the number of bits 1) computed and 2) communicated. Reducing the number of bits computed will necessarily reduce either the precision or range of a floating point number. I focus on saving energy by way of reducing precision, which can exploit the over-provisioning of bits in many stages of the graphics pipeline. Reducing the number of bits communicated takes several forms. First, I propose enhancements to existing compression schemes for off-chip buffers to save bandwidth. I also suggest a simple extension that exploits unused bits in reduced-precision data undergoing compression. Finally, I present techniques for saving energy in on-chip communication of reduced-precision data.

By designing and simulating variable-precision arithmetic circuits with promising energy versus precision characteristics and tradeoffs, I have developed an energy model for GPUs. Using this model and my techniques, I have shown that significant savings (up to 70% in computation in the vertex and pixel shader stages) are possible by reducing the precision of the arithmetic. Further, my compression approaches have enabled improvements of 1.26x over past work, and a general-purpose compressor design has achieved bandwidth savings of 34%, 87%, and 65% for color, depth, and geometry data, respectively, which is competitive with past work. Lastly, an initial exploration in signal gating unused lines in on-chip buses has suggested savings of 13–48% for the tested applications' traffic from a multiprocessor's register file to its L1 cache.

ACKNOWLEDGMENTS

I have been incredibly blessed and have lots of people to thank for my having had a smooth and fruitful time learning and working at UNC. God has indeed been good to me!

Anselmo Lastra and Montek Singh, my advisors, have been indispensable. Without their encouragement, direction, and unflagging dedication to producing the best quality work, I'd be nowhere near the completion of anything worth completing. So, thanks to them for pushing me to meet deadlines, their help in sussing out the real issues and obstacles in my research, and their ruthless editing over the last five years. They have provided immeasurable insight into not only the problems encountered in my research, but also into how to be more effective at research in general.

My committee members, Dinesh Manocha, Steve Molnar, and John Poulton, also deserve a very large "thank you." Their feedback on my work was invaluable at all stages: my proposal, defense, and on up to the final version of this dissertation. My work has improved not insignificantly as a direct result of our discussions as I was forced to look at things from all sorts of new angles, not just ones with which I was familiar.

I also am grateful to NVIDIA for providing me with the phenomenal experience of having four summer internships working on a multitude of challenging problems. Seeing the amount of effort it takes to wind up with a functioning piece of silicon at the end of the process was eye-opening, and having the opportunity to be a part of this process was a true thrill. Those summers would have been far less productive and engaging without the help and guidance of my mentors there: Paul MacDougal, Lars Nyland, and Steve Molnar.

As an undergraduate at the University of South Carolina, I was very fortunate to have an advisor willing to make me figure out what I really wanted to do. Robert Pettus was that advisor, and he urged me to seek an internship with his past students Chris King and Michael Sechrest. They graciously allowed me to work for them at IDV,

Inc., despite my lack of experience. My time there was fantastic, and their guidance and generosity was an integral part of my starting down the road of Computer Science.

As much as the aforementioned parties have pushed me to learn and grow academically, I am enormously grateful to my family and friends for everything else. My parents, Mike and Dale, and brother, Greg, have supported me unconditionally in everything I do. I could not have asked for anything more from them, and I rarely even had to ask at all. They, along with my friends (old, new, near, and far), were incredibly understanding and tolerant of my going into hiding for weeks and even months at a time to get things done. Everyone played their own important role in keeping me going—thank you all for the continued pushes, proofreading, kind words, and help that defies categorization!

Finally, my research was supported by the National Science Foundation under grant CCF-0702712, and equipment was provided by NSF Research Infrastructure grant number 0303590.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU	arithmetic logic unit
AMD	Advanced Micro Devices
BVH	bounding volume hierarchy
CMOS	complementary metal-oxide semiconductor
CPU	central processing unit
CUDA	Compute Unified Device Architecture
DCT	discrete cosine transform
DRAM	dynamic random-access memory
DVFS	dynamic voltage and frequency scaling
fov	field of view
FPGA	field-programmable gate array
FPS	frames per second
FPU	floating-point unit
GPGPU	general-purpose computation on graphics processing units
GPS	global positioning system
GPU	graphics processing unit
GUI	graphical user interface
HDR	high dynamic range
HPC	high-performance computing
LSBs	least significant bits
LTF	last texture fetch
PC	personal computer
PCI	peripheral component interconnect
PSNR	peak signal-to-noise ratio
SDK	software development kit
SIMD	single-instruction multiple-data
SoC	system-on-a-chip
SRAM	static random-access memory

Chapter 1

Introduction

1.1 Motivation

Graphics processing units (GPUs) in desktop computers have become very powerful in recent years, capable of creating nearly photo-realistic images by processing hundreds of millions of triangles and pixels every second. Similarly, graphics hardware has been used for general-purpose computation on graphics processing units (GPGPU) in applications to accelerate the solutions to problems such as molecular simulations, modeling large-scale crowds, and weather predictions. GPUs have been integrated into mobile devices, such as smart phones and tablets, to enrich the user experience and enable high-definition video applications. In all these domains—desktop graphics, GPGPU, and mobile devices—energy is a limiting factor in the performance of the GPU. While mobile devices are ultimately limited by the total energy available after a battery charge, desktop and server hardware also has to perform within the limits of their power supplies and heat dissipation solutions. Thus, energy consumption is directly related to performance! Decreasing the energy demands of the hardware will allow for both higher performance and longer battery lifetimes.

Nearly all computer graphics are based on the interaction of different types of light sources with different surfaces, which are well-understood natural phenomena. By simulating these interactions accurately, computers can render scenes that increasingly approach reality. As with any simulation, however, the results are approximate. Real-time graphics regularly employ many tricks to hide errors in these approximations, and even images generated by offline renderers are not exact replicas of scenes in real life. At the lowest level, the hardware used to render graphics has limited precision, and monitors can only display a finite number of different colors. Hao and Varshney first

looked at variable-precision rendering for speed benefits when the vertex operations in the graphics pipeline were implemented in software and executed on the central processing unit (CPU) (Hao and Varshney, 2001). By reducing precision requirements, CPU rendering could be sped up. I take a similar approach, but my target is GPUs and my objective is saving energy consumption. My approach exposes a tradeoff between rendering precision and energy demands, which can equate to battery life improvement in mobile devices and performance increase in power-limited desktop units.

There are many approaches to reducing the energy used by hardware, from semiconductor manufacturing techniques, to reducing the voltage and/or frequency at runtime, to shutting down entire processing cores. My work is orthogonal to these techniques; the systematic reduction of precision throughout all levels of the pipeline can be used in tandem with these and other standard approaches to enable further savings. This work focuses on just the graphics pipeline, a single component of an overall system which typically has many other components using energy, such as some number of CPUs and display screens. While all these other components may use significant energy, the consumption of the GPU can often limit both the battery life and performance of the system. For example, a mobile device may deplete its battery much more quickly if the GPU is used for an extended period of time, and the performance of a GPU running a graphics or general-purpose application may be unnecessarily limited. For modern GPUs, power consumption is a significant issue that can cause performance bottlenecks and frustrate users.

1.2 Background: Graphics Pipeline

Before discussing specifics of how to save energy in the graphics pipeline, let me first briefly present a high-level view of graphics in general. Computer graphics is, at its heart, a series of similar computations performed on different data. These computations are performed in a pipeline, a simplified view of which is shown in Figure 1.1. The first stage is the transformation of input data—vertices from disparate coordinate frames—into a unified "world-space" and then into "screen-space" (often combined into a single matrix multiplication). These transformed vertices are then assembled into triangles visible on the screen, possibly sharing a transformed vertex between several triangles. These triangles are sent through the rasterization stage, which generates a list of pixels that are wholly- or partially-covered by each triangle ("fragments"). These pixels are finally "shaded," or given a final color based on lighting and texture information. It is

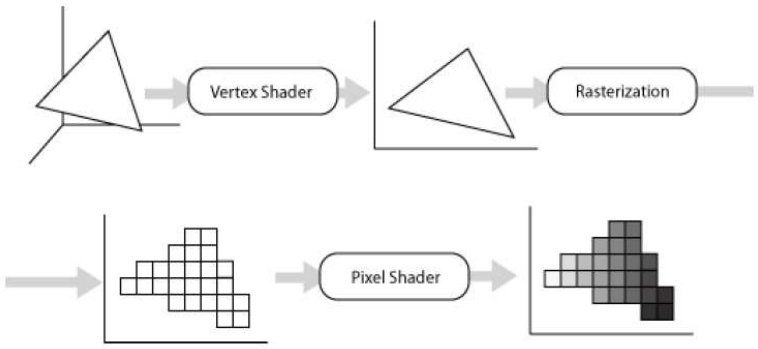


Figure 1.1: A simplified view of the traditional graphics pipeline. Vertices enter the vertex shader, where they are transformed to screen space through a series of matrix multiplications. These transformed vertices are assembled, or "set up," into triangles. Next, these triangles are rasterized, creating lists of pixels covered by the on-screen triangles. These pixels are shaded, determining their final colors, before compositing them with geometry that has already been rendered to the final frame-buffer.

possible to discard (or "cull") data at any of these stages for reasons such as triangles existing entirely off-screen, or a set of pixels being entirely occluded behind opaque geometry that has already been drawn to the screen.

1.3 The Use of Graphics Hardware

The first dedicated graphics hardware was built to satisfy the demanding performance requirements of flight simulators. Strict requirements, such as real-time frame rates and low latency from user input to response on the screen, meant that general processors of the time period were not able to take on the job. The reader is referred to a survey of the topic for more information (Mueller, 1995).

In the personal computer (PC) market, these operations were, for a time, performed on a computer's CPU, the same general-purpose processor also responsible for executing all applications and operating system functions. However, as rendered scenes became increasingly complex, dedicated hardware (the GPU) that could be added to a PC was built to handle part of this load. At first, this hardware handled only rasterization and pixel operations; later it also performed vertex transformation and lighting operations. Early graphics hardware used a fixed-function implementation, which allowed for only minimal control by exposing different "modes" to the programmer, letting them change such parameters as lighting functions, blending modes, depth cueing, and backface

culling.

In time, this fixed function hardware gave way to programmable hardware, letting the application programmer or artist dictate how to transform vertices and color pixels. This technology allowed for much more complex rendering techniques than were previously possible. At this point, scientists realized that these highly-parallel GPUs could also be exploited for general-purpose computations if they expressed these computations as graphics operations (early GPGPU). As programmability increased, the processing cores used for vertex transformations and pixel shading became progressively similar, eventually merging into a larger pool of "unified shaders" that can be allocated dynamically to adapt to varying workloads. This unification has allowed for new pipeline stages to emerge, such as geometry and tessellation shaders. Further, hardware vendors have made it easier to program the GPU as a general-purpose processor, allowing widespread use of the hardware for GPGPU and high-performance computing (HPC) applications in addition to the graphics workloads for which it was designed.

1.4 Contribution: Precision-Energy Tradeoff

Nearly all of computer graphics is an approximation, even with all the processing power available in modern GPUs. Lighting equations are simplified to run in fractions of a second. Reflections on surfaces are, at times, not updated each frame. Research into the human visual system has led to lossy compression formats that are used to save memory and bandwidth. It is these approximations that lead to the key insight behind this thesis: reducing the precision of graphics operations need not have a negative effect on the application's usability and can save significant energy.

This tradeoff between the energy efficiency of a graphics application and the precision with which it computes the results can allow the user to choose an operating mode along the continuum connecting the two extremes. At one end, the user can enjoy a faithful reproduction of the application designer's vision at the expense of higher energy consumption. In mobile devices this will mean a shorter battery life, and in desktop and server settings, this will mean more heat that must be dissipated and higher energy costs. At the other end of the continuum is very long battery life (in a mobile device) with very noticeable errors. It is my intention that the user can choose a point in the middle that saves significant energy yet does not incur any noticeable errors.

This collection of ideas creates my thesis statement:

Figure 1.2: "Crysis," a popular video game, is an example of a class of applications that can benefit from my proposed techniques. The pictured scene's depth information was compressed by a factor of 7.7x, and the geometry data was compressed by a factor of 3.3x with my unified buffer compressor (see Chapter 6).

Reducing the work done in the modern graphics pipeline through novel communication and variable-precision computation techniques can enable a tradeoff between energy savings and image fidelity, leading to significant energy savings without perceptible loss of image quality.

1.5 Results

In order to defend this thesis, I approach the larger problem of energy savings in several parts, discussed independently, below. All of my proposed techniques apply to and have been tested on large-scale real-world applications, such as "Crysis" (Figure 1.2) (Crytek, 2007).

1.5.1 Energy Model

I first develop an instruction-level energy model for a GPU (Figure 1.3) by experimentally measuring the total energy used by a reference graphics card. For each operation (memory accesses, arithmetic, and fixed-function graphics operations), I measure the energy required for a directed microbenchmark. Then, I combine these individual energy per operation values to construct a model for any given workload. Accurate to within 10–15%, it allows programmers or architects to estimate the energy consumed by a particular graphics application on a particular architecture. By using the model's

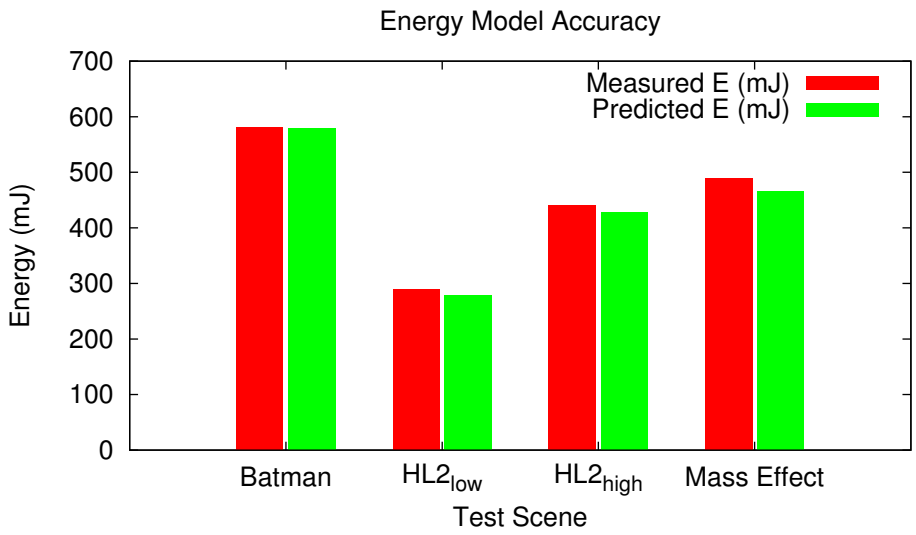


Figure 1.3: The accuracy of my energy model for GPUs. The model is accurate to within 10–15% for the tested data sets, leading to very accurate predictions of energy consumption of the system as a whole, as well as different stages of the pipeline (see Chapter 3).

predictions for discrete sections of the hardware, I am able to estimate the impact of reducing the energy in a single part of the graphics pipeline on the overall energy consumption of the entire GPU. This, in turn, leads to an estimate of overall savings possible by putting the following techniques into practice.

1.5.2 Energy Savings in Computation

Due to the inherently approximate nature of computer graphics, the precision of floating-point numbers can be reduced significantly without noticeably affecting the final result, though the degree to which the precision can be reduced depends on the data used in the computations. Figure 1.4 shows an example of this: the dragon on the top was rendered with full-precision arithmetic (24 bits of precision) in the pixel shader, while the lower image used an average of only 12.5 bits of precision. Similar reductions are possible in the vertex shader stage, leading to average energy savings of 70% in the shaders' arithmetic (details in Chapter 5). I have also categorized the different types of errors that manifest themselves at different points in these shaders. Finally, I present several techniques for choosing a successful operating precision that saves energy without incurring intolerable error.

(a) Full Precision

(b) Reduced Precision

Figure 1.4: Figure 1.4(a) is the reference frame produced by full-precision computation (24 bits) throughout the pixel shader of a screen space ambient occlusion demo. Figure 1.4(b) shows the result when using an average of 12.5 bits of precision in the pixel shader. 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.

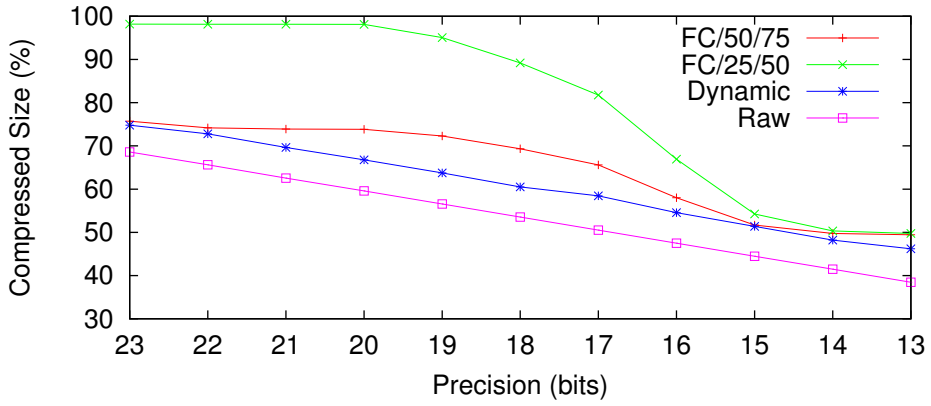


Figure 1.5: Range reduction of variable-precision prior to compression data is very effective when used with dynamic bucket selection (HDR_1 scene) (see Chapter 6). My approach ("Dynamic") leads to compressed data sizes closer to the ideally-compressed size ("Raw") than two different standard static bucket selections ("FC/50/75" and "FC/25/50").

1.5.3 Energy Savings in Communication

I also save energy by reducing the number of bits that are necessary for communication, both on- and off-chip. First, I suggest two enhancements to a state-of-the-art compression scheme: dynamic bucket selection and using a Fibonacci encoder. These two techniques lead to an average improvement of 1.26x for an existing compressor.

Next, I describe a general-purpose compressor that is able to handle data from any source and of any layout without modification, which is a limitation of past work. It is clear that the GPU is a very general-purpose device; I feel the use of different specialized compressors for color, depth, etc., is not beneficial to the GPU's utility in a broad range of applications. Using this compressor, I estimate average bandwidth reductions of 1.5x, 7.7x, and 2.9x for color, depth, and geometry data, respectively.

Lastly in off-chip communication, I suggest a straightforward method that will take advantage of unused bits in compressing reduced-precision data, called "dynamic range reduction." Essentially, this technique treats reduced precision data in a similar manner to the computation: lower bits are simply ignored. The bandwidth savings will vary, depending on the data and precisions of the applications, but are expected to be between 5% and 20%. Figure 1.5 shows dynamic bucket selection and dynamic range reduction working in tandem to lower the bandwidth of reduced-precision data.

Saving energy in on-chip communication takes a different form; since compression is seldom used for sending data relatively short distances, I explore the use of signal

gating on a bus from a processor's register file to its L1 cache. I have shown that the energy savings are nearly linear with bit width of the bus, so disabling 8 out of 32 bits will reduce the energy consumption by 25%. By simulating the data sent on this path for several applications, I have enabled savings between 13–48%, with an average energy savings of 36%. This technique requires only a minimal overhead, which is more than reclaimed for any "burst length" seen by the bus.

1.6 Outline of This Thesis

Following is the organization of this dissertation. Chapter 2 presents an overview of related work in the area of low-power graphics and hardware. Then, I present my energy model in Chapter 3. In Chapter 4, I develop and simulate several variable-precision arithmetic circuits whose energy-precision characteristics are used to estimate energy savings in later chapters. Chapters 5 and 6 detail my work in variable-precision computation and communication, respectively. Finally, Chapter 7 summarizes my findings and offers some conclusions.

Chapter 2

Background and Related Research

This chapter contains a short primer on power and energy, as well as common energy-saving techniques, both in the computation and the communication of data. Also, I review existing variable-precision applications, specifically, work in graphics and physics simulations.

2.1 Power and Energy

As discussed in Chapter 1, reducing the power and energy consumption of graphics hardware is an important task for modern architects, hardware designers, and software developers. The first step in reducing power is understanding exactly how the power is consumed on a chip. Let us start at a very low level: a simple complementary metal-oxide semiconductor (CMOS) inverter, pictured in Figure 2.1. The function of the inverter is to transform a high signal (logical '1') on the input side to a low signal (logical '0') on the output side, or a low input signal to a high output signal, thereby inverting the input. This is accomplished by a single pair of NMOS and PMOS transistors that conduct in opposing situations. When the input is low, the PMOS transistor (on top in the figure) conducts, allowing the output to be pulled high. Similarly, the NMOS pulls the output low when the input is high. This seemingly simple idea can be extended to create any logical gate necessary, such as NAND and NOR gates. These gates, arranged in a particular fashion, can compose a basic circuit: an integer adder, for example. These basic circuits, in turn, can be combined to create more complex circuits and, eventually, a processing unit.

Stepping back to the inverter, I will discuss how it consumes power to perform its simple task. First, we should look at the inverter in the context of a larger circuit: the

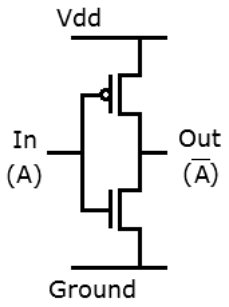


Figure 2.1: A CMOS inverter.

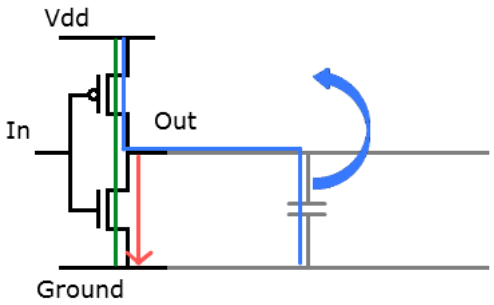


Figure 2.2: A CMOS inverter with output capacitance and main sources of power use illustrated: dynamic power (blue), short circuit power (green), and leakage power (red).

inverter's input and output will be connected to other gates. The inverter's output, in particular, is said to "drive" the next element (just as the inverter itself is being driven by whatever came before it). A capacitive load is seen by this output, the magnitude of which is determined in part by the size of the driven element and the length of connecting wire. This capacitance is illustrated in Figure 2.2.

There are three main categories of power consumed during an inverter's operation. The first, *dynamic* (or *switching*) *power*, is highly dependent on the output capacitance. It is this capacitance that causes the transistor to have to work to change the output from one signal to another; the capacitance stores charge, and it must be either charged or discharged for the signal to change. The second, *short circuit power*, is due to the inability of transistors to switch on and off instantaneously. During a transition, there will be a brief moment when both the PMOS and NMOS transistors are conducting. At this point, current will have a direct path from the power supply to ground, creating a short circuit which consumes power without doing any useful work. Finally, there is

leakage power, which is always present and not dependent on the activity of the gate. The transistors that make up the circuit are not perfect switches; some charge "leaks" through them even when they are disabled. These three sources of power consumption are illustrated in Figure 2.2.

Dynamic and leakage power are the two dominant consumers of power, so let us discuss them in more detail. Switching power is dependent on the capacitance: the larger the output capacitance, the more the gates have to work to change the output charge. Likewise, the higher the source voltage, the more the output has to change; this relationship is quadratic. Finally, the frequency with which the output changes directly affects the power consumed. This is determined by two quantities: the frequency at which the circuit is operating (or clock speed, f) and the *activity factor* (α) of the individual gates. The activity factor is an estimate of how often a given gate undergoes a transition and is a number between '0' (never) and '1' (every cycle). These various quantities are shown together in Equation 2.1.

$$P_{switching} = \frac{1}{2} C V^2 f \alpha \tag{2.1}$$

Leakage power does not depend on the frequency or activity factor of a circuit or gate. Instead, it is determined by the source voltage and the area of the gates. (This is a very high-level treatment that captures the most important aspects of leakage current; the reader is referred to a more in-depth analysis for further details (Yeap, 2002; Mukhopadhyay et al., 2003; Butzen et al., 2007; Rastogi et al., 2008).) The leakage power consumed by a circuit component is shown in Equation 2.2.

$$P_{leakage} \propto V * Area \tag{2.2}$$

Instantaneous power, in units of watts (W), is an important factor in the heat produced by high-performance hardware, so the effectiveness of the cooling (passive, fan, or even water-based) used on a circuit can often dictate how much power the circuit can handle. Further, the power supplies driving computational hardware have bounds on the amount of power they can deliver. These limits have recently become a bottleneck in GPUs; if graphics hardware used less power, it could run at a higher frequency. So, power efficiency is a concern to today's GPU designers.

Like power, energy is a consideration that designers must keep in mind. Energy, with units of joules (J) or kilowatt-hours (kWh), is instantaneous power over a period of time, shown in Equation 2.3. While energy consumption can be important for desktop

and server hardware, as energy cost directly impacts the total cost of operation, energy efficiency is even more crucial for mobile hardware. Batteries are limited by the amount of energy they can store; when that energy is depleted, the device powered by that battery is useless until the battery is replaced or recharged. While batteries do have limits on the instantaneous power they can deliver, the lifetime of the battery is often the more pressing concern to the consumer.

$$E = \int_0^T P(t)\, dt \tag{2.3}$$

2.2 Saving Energy in Computation

Saving energy in the computational phase of a program can come at all levels of the device: algorithmic, architectural, circuit, and even changes at the transistor level affect energy consumption. Though surveys of existing techniques exist (Benini et al., 2001; Hung et al., 2009), I will briefly mention and discuss common energy-saving approaches detailed in these surveys and refer the reader to them for more details.

2.2.1 Power, Clock, and Signal Gating

The most straightforward way to save energy is to attack the power term in Equation 2.3. In turn, there are two quantities in Equations 2.1 and 2.2 that can be changed at runtime: the voltage and frequency of the circuit. (The hardware's area, capacitance, and switching activity are tied to circuit- and architectural-level decisions.) Shutting off either the power or the clock that drives the circuitry will stop the circuitry from performing useful work but can drastically reduce the power, and therefore energy, consumed. Both techniques have their own benefits and caveats, however.

Power gating refers to completely turning off the supply voltage to some area of circuitry. Clearly, a circuit with no voltage will not function, so it is used on circuitry which is not currently needed, such as a floating-point unit (FPU) during an integer operation. Since a circuit's voltage plays a role in both its dynamic *and* its leakage power, power gating will reduce both of these quantities. Power gating can also apply to many levels of the hardware, from entire cores and partitions down to computational paths for individual bits. However, completely turning the voltage to a circuit on and off is not instantaneous—the hardware may take some time after it is re-enabled before it is usable again. Scheduling power events is a complicated problem, both for making

sure the hardware is available for a task (Wang et al., 2010), as well as reducing noise in the power lines driving the hardware (Jiang and Marek-Sadowska, 2008).

Clock gating does not affect the supply voltage; rather, it eliminates switching activity in a component by changing the effective frequency to zero. As this frequency term is only found in the equation for dynamic power, clock gating does not affect leakage power; current still leaks through the transistors in the path to ground. However, re-enabling a clock-gated circuit is much faster than re-enabling a power-gated circuit, and there are no issues with noise on the power and ground rails. As with power gating, clock gating can be applied to many levels of the hardware's design.

One last type of gating does not change either the voltage or the frequency, but focuses on the activity factor (α) in Equation 2.1: signal gating. If it is known beforehand that the result of an operation (or sequence of operations) will not be used, then the inputs to the hardware can be "frozen," or held at a constant value. This value can be a logical '1' or '0' (which is simple in implementation, but can require a small amount of power to force the inputs to a particular value) or it can take the existing value as the constant value (which can be slightly more complicated in implementation, but requires no power to change the values). Signal gating can be applied to entire registers, or even just to individual computational paths (Huang and Ercegovac, 2001).

In Chapter 4, I use very fine-grained power gating to shut down sections of arithmetic circuits for energy savings, and I use signal gating in Chapter 6 to save energy in on-chip communication of reduced-precision data.

2.2.2 Dynamic Voltage and Frequency Scaling

While simply gating the voltage or clock signal to a circuit can save significant energy, it can sometimes be too heavy-handed; reducing the voltage and clock speed by some factor can often save energy while still allowing the circuitry to function as intended. Changing voltage and frequency at runtime, or dynamic voltage and frequency scaling (DVFS) (Benini et al., 2001), allows for a tradeoff between power or energy and performance. A simple example showing the effect of DVFS on a circuit's dynamic power should make this tradeoff clear.

Given a simple circuit that performs work at a voltage of 2V and a frequency of 100Hz, and expressing switching activity (α) and capacitance (C) (seen in Equation 2.1) as a single constant, k, the switching power is given in Equation 2.4. Equation 2.5 shows the power consumed by the same circuitry running at half the voltage and

half the frequency of the original. (Voltage and frequency do not necessarily scale with the same ratio; this is a contrived example!) Finally, Equation 2.6 shows the ratio of dynamic power consumed by the circuit operating at original and scaled voltage and frequencies; the circuit under DVFS uses only one-eighth the power.

$$\begin{aligned} P_{base} &= k * 2^2 * 100 && (2.4)\\ &= 400 * k \\ P_{scaled} &= k * 1^2 * 50 && (2.5)\\ &= 50 * k \\ P_{scaled} &= \frac{1}{8} P_{base} && (2.6) \end{aligned}$$

There is one final step to find the energy savings. Since the frequency of the circuit was halved, the time spent in the computation was doubled, which, as we saw in Equation 2.3, will play a role in the energy. The energy consumed by the circuit (Equation 2.7) is reduced by using DVFS (Equation 2.8) to only one fourth of the original energy (Equation 2.9):

$$\begin{aligned} E_{base} &= \int_0^T P_{base}\,(t)\,dt && (2.7)\\ E_{scaled} &= \int_0^{2T} P_{scaled}\,(t)\,dt && (2.8)\\ &= \frac{1}{8}\int_0^{2T} P_{base}\,(t)\,dt \\ &= \frac{1}{4}\int_0^{T} P_{base}\,(t)\,dt \\ E_{scaled} &= \frac{1}{4} E_{base} && (2.9) \end{aligned}$$

The performance and power tradeoff should be clear. However, in some circumstances, there need not be a performance hit. With two similar units (be they simple adders or entire processors) and a sufficiently parallelizable workload, the same work can be done in the same time with much less energy by using both units at the same time. This is the approach taken by NVIDIA when motivating the use of multiple CPUs in their Tegra 3 system-on-a-chip (SoC) (NVIDIA Corporation, 2012).

2.2.3 Workload Reduction

A slightly different approach to saving energy is to simply do less work. If the work that is not performed was not necessary, or at least not noticeably important, then the larger application can save energy by not performing it. For instance, Lafruit et al. present a method for estimating the time necessary to render a frame based on input statistics and render states and reducing the workload gracefully if this time is too large (Lafruit et al., 2000). By reducing the render buffer (virtual screen) size, texture resolutions, and mesh resolutions, the authors have shown that rendering time can be (approximately) bounded with a "full" result, rather than truncating the rendering process in the middle of a frame. This allows for a quality/performance tradeoff, starting with a full-quality input scene and scaling down as desired.

Similarly, variable-precision techniques seek to do less work and arrive at approximate answers to computations, which, in many applications, are close enough to the correct answer. This is a broad topic, and I discuss it more fully in Section 2.4.

2.3 Saving Energy in Communication

Communication, not just computation, of data can also be a target for significant energy savings, and there have been studies detailing the power consumption in communication hardware (Lahiri and Raghunathan, 2004). Long-distance data communication consumes roughly an order of magnitude more energy than the computation performed on that data (Keckler et al., 2011), and this disparity is expected to increase in the future as transistors continue to shrink. There are many ways of approaching reducing the energy consumption of communication, which can target the amount of data, encoding of data, or even how the data is sent over long wires (Oh et al., 2006). Caches can reduce the amount of data that must be sent over a long distance, effectively increasing the bandwidth and energy efficiency of the hardware, but can hurt performance when handling poorly-behaved data access patterns (Bahar et al., 1998). Choosing different encodings for the data sent across a bus can reduce the transitions seen on long wires (the α term in Equation 2.1) (Zhao et al., 2007; Lindkvist and Lofvenberg, 2005).

I look closely at compression of memory traffic on graphics hardware (Ström et al., 2008; Rasmusson et al., 2009). Compression can reduce redundancy within data, expressing the information contained within it more compactly and making it more efficient to send across wires. More information can be found in Chapter 6.

2.4 Variable-Precision Applications

Reducing the precision of the variables used for computation in an application can be seen as a reduction in workload. The results may no longer be as exact, but the computational effort can be greatly lessened. This type of tradeoff has been explored in many domains, which I discuss briefly below.

2.4.1 Graphics

Variable-Precision Rendering

Hao and Varshney looked in-depth at variable-precision rendering in the geometry transform and lighting stage to accelerate 3D graphics (Hao and Varshney, 2001). It is important to note that their work focused on CPU-side rendering, so they exploited the use of MMX (a single-instruction multiple-data (SIMD) instruction set designed by Intel) instructions and operated on integer and fixed-point representations. Further, they applied their work to the fixed-function pipeline, which has fallen to the wayside with the introduction of programmable shading. However, their work provides a foundation upon which to build a modern exploration. First, they present a breakdown of sources of error in data sets and computations for inputs with n bits, listing worst-case errors.

1. Representation error. These are statistical and observational uncertainties. At worst, the representation error is one half bit: $\epsilon^{rep} \leq \frac{1}{2}$.

2. Addition error. Propagation error leads to at most one bit of lost accuracy for each addition.

3. Multiplication error. Using $2n$ bits to store the intermediate result, the worst case error occurs when both operands are close to 2^{n-1} and the representation error is $\frac{1}{2}$: one bit of accuracy can be lost during each multiplication.

4. Division error. Assuming the division is in the transformation from homogeneous coordinates to 3D image-space coordinates, the loss of accuracy is:

$$\left\lceil log_2(1 + \frac{distanceoffarplanefromeye}{distanceofscenevertextoeye}) \right\rceil$$

Finding the total error incurred is a linear combination of errors for each operation. Working backwards from, for example, 10 bits of precision in x and y for a 1024x1024 rendering window, one can find the necessary bits at the input to guarantee 10 output bits of precision. Sub-pixel accuracy is computed by artificially enlarging the window size.

Small objects in the distance do not need as much precision as a big object in the foreground. They propose an octree-based bounding volume hierarchy (BVH) to keep track of the position of rendered items in space to take advantage of this technique. If the near and far vertices in a cell need the same number of bits to be represented accurately, then this number can be used for every vertex in the cell; otherwise, it must be split.

Spatial coherence can be exploited in 3D models by encoding neighboring vertex positions as offsets from previous positions. Temporal coherence can be similarly exploited by expressing a transformed vertex as the sum of the originally transformed vertex and the original vertex transformed by the difference between the previous and current transformation matrices.

There are further sources of error in lighting operations that were not present in vertex transformations.

1. Operands with different accuracy. When two operands have different precisions, results always take on the precision of the lesser-precise operand.

2. Dot products (of unit vectors). For dot products of two three-component vectors, the results will lose one to two bits of precision.

3. Square roots. When implemented with a lookup table, the result will have nearly the same precision as the input (as long as the input is bigger than 2^{2n-2}).

4. Exponentiation. A step in the calculation of the specular component which will incur a loss of precision of 6 bits.

Lighting computations can be treated just like spatially-coherent geometry, calculating one vertex's lighting as an offset from a neighboring vertex's result.

Minimum Triangle Separation

A common problem that has plagued graphics applications for years is called z-fighting, and it occurs when two triangles are (nearly) co-planar. The limited precision of the

Figure 2.3: Z-fighting in the shoreline of a frame from "Grand Theft Auto: IV."

depth buffer cannot capture the correct rendering order across the entirety of the triangles. So, one triangle is rendered in front of the other triangle in some pixels, with the opposite ordering chosen for other pixels. The effect is exacerbated as the viewpoint moves, since the ordering is not spatially coherent. An example of z-fighting in the video game "Grand Theft Auto: IV" can be seen in Figure 2.3 (Rockstar Games, 2008). Apparent even when rendering a scene at full-precision, this problem can become worse as geometric precision is reduced.

Akeley and Su analyze the minimum triangle separation in object-space for correct occlusion given a viewing environment: camera position, field of view (fov), and window coordinate precision (Akeley and Su, 2006). By beginning with a minimum triangle separation, instead, an artist can calculate a final minimum necessary buffer and geometric transform precision to use when reducing the precision of an application that utilizes their 3D models.

Their method works as follows: an uncertainty cuboid is formed for each 3D location in window coordinates, the depth of which is the numeric distance between the representable z-buffer values nearest its location, and whose width and height (identical for all cuboids in a window) are determined by b, the precision of the window coordinates. Given a traditional z-buffer, cuboids near the near plane will be shallow; those near the far plane will be deep. Conversion to eye coordinates is done by inverting the projection and viewport transformations to reverse map the cuboids, which become frusta.

Parallel triangles may swap order (fight) if and only if any of their uncertainty frusta overlap. The minimum distance, S_{min}, is the length of the frustum's longest diagonal.

A frustum in a screen corner will be highly sheared, meaning its diagonal will be longer than it would be at the center of the screen. This factor is labelled K_{fov}—the ratio of corner-screen to center-screen diagonal length for uncertainty frusta on a given z_{eye} plane. The minimum separation depends on all these factors—simulations show that discounting any one of them will lead to an under-prediction and possible punch-through.

Finite-precision projection, viewport, and rasterization (mapping) arithmetic can further increase the minimum precision. The authors modeled the error in these operations by performing them in double precision. The contribution of this mapping error to S_{min} is minor due to the spatial-related error dominating the depth-related error; 10.8 fixed-point spatial precision used in the representation of window coordinates x_{win} and y_{win} is far below that of floating-point.

Texture Mapping

Textures, or pre-computed images, are often applied to triangles to add detail that is not captured by lighting equations alone. (While texture mapping can be performed at both the vertex and pixel shader stages, I will discuss texturing at the pixel level in particular.) These textures can represent color, normal, reflectance, and many other types of information. Special fixed-function hardware is used to determine what texture element, or *texel*, is to be applied to a particular pixel based on that pixel's texture coordinates, effectively an address into the texture memory. This address, though, is often a floating-point number that selects an element a fraction of the way through the data. If this address is greater than '1,' either the address is clamped or the texture is treated like a periodic signal.

Since floating-point addresses do not often land precisely on a single texel and a single pixel may cover several texels, the texture mapping hardware must decide what value to return. The simplest approach the hardware can take is to choose the nearest texel; this is seldom used in practice because of its poor quality and aliasing artifacts. Instead, filtering (i.e. interpolation) is often performed. By examining the four nearest texels to the pixel's center and performing a weighted average on their values, the texture hardware can enabled smoother gradients across texel boundaries. This is referred to as bilinear filtering. Trilinear filtering, on the other hand, performs bilinear filtering on two mipmap levels (Williams, 1983) and linearly interpolates between these two

values to find a single result. This inclusion of mipmapping leads to gentle transitions when a texture is applied to triangles of varying sizes. Finally, anisotropic filtering is the highest-quality filtering commonly used; it addresses cases in which a texture is applied to a triangle at a high relative angle to the camera, meaning it is much larger in one dimension than the other (*not* isotropic).

Chittamuru et al. present a method of trading off energy for quality in this texture mapping hardware (Chittamuru et al., 2003). They discuss two techniques for skipping certain `MAC` operations in texture filtering: weight-based and intensity-based techniques. If texel weights in the bilinearly or trilinearly sampled texels are small enough, they can be ignored. Similarly, if two neighboring texels are roughly equal, the two `MAC` operations can be transformed into an addition and a multiplication. This technique offers a tradeoff: comparing more bits of neighboring texels leads to more accurate results, and comparing fewer bits will lead to fewer `MAC` operations. The authors also present an architecture for efficiently evaluating texel and weight similarities, so that power spent in comparisons will not outweigh the savings realized. In total, the authors save 30–50% of the power and speculated that up to 80% could be saved with the use of multiple voltage supplies.

2.4.2 Physics

Yeh et al. explored error tolerance in physically based animation (Yeh et al., 2006; Yeh et al., 2009). Physics simulations are usually performed in several steps: broad-phase and narrow-phase collision detection, island creation (grouping colliding objects together), and the simulation step (applying forces to simulated bodies). By injecting bounded random errors into these different phases, the authors determined acceptable limits. From a quantitative analysis of the errors, they choose the "knee" at which the system suffers a catastrophic failure as the last acceptable error threshold. Several tests confirmed this choice: visual inspection, comparison with a previous contrived system, comparison of the magnitude of observed errors to that seen in constraint reordering, and examining the effect of different timesteps on the errors. They observe that the overall energy in the system is a good indicator of whether or not a given simulation is well-behaved; if the total energy does not remain constant, then the simulation will likely explode into very implausible behavior. Finally, the authors present four case studies in trading off accuracy and performance.

1. Simulation timestep. It was shown that a frame rate of at least 34 frames per

second (FPS) is necessary to keep the simulation stable.

2. Iteration count. 11-30 solver iterations are result in a stable simulation at 60 FPS, but some iteration counts over 30 are *not* stable for only 30 FPS. So, iteration count can't be traded for frame rate.

3. Fast estimation with error control. Previously presented by Yeh (Yeh et al., 2006), this method creates a precise thread and an estimation thread for a given computation. The estimation thread completes first, allowing other components (rendering, AI) to begin working with this estimated result. The precise results are fed to the input of both threads for the next frame. Reducing the iteration count of the estimation thread led to stable simulations for iterations counts as low as 1, due to the precise input used for each frame.

4. Precision reduction. The precision of the computational steps is reduced, rather than range, because exponents are less tolerant of bit width reduction and the possible savings are lower. Precision thresholds derived numerically are much higher than thresholds arrived at through perceptual metrics. For the authors' tests, around 7 mantissa bits were the most ever needed by a phase of the simulation to remain stable.

Chapter 3

Energy Model

3.1 Motivation

Hardware designers and, recently, software designers, have gone to great lengths to reduce the power consumption of their hardware and applications. Hardware designers seek to minimize heat, keeping power and cooling requirements low in desktop units. In mobile GPUs, they go to great lengths to maximize battery life to make their solutions more attractive to buyers. Likewise, software designers know that their applications will likely see more use if they are not excessively power hungry. It is for this reason that mobile platforms, such as Apple's iOS, include tools for monitoring the power consumption of applications under development (Apple Inc., 2010).

A validated energy model for GPUs would be helpful in predicting the impact of modifying applications on energy efficiency. For example, one could determine how the mix of operations performed in the hardware will change when using a different architecture (such as a tiled rendering scheme, similar to the popular POWERVR graphics solutions (Imagination Technologies Ltd., 2010)). Alternatively, one could look at the energy efficiency of different algorithms used for a graphical technique called bump-mapping (NVIDIA Corporation, 2004). With the energy model I introduce in this chapter, I am able to examine both these facets of hardware and software design.

3.2 Related Research

A commonly used solution for modeling the power consumption of computer hardware at the architectural level is Wattch (Brooks et al., 2000), which uses cycle-level simulation and parameterizable hardware power models to estimate power usage of different

CPU architectures and compiler techniques to within around 10% of the actual reported value in most cases. Since then, there have been studies on the modeling of a single-core system at the architectural level, so that weeks of simulation are not necessary, with good results (Chen et al., 2001; Varma et al., 2008).

For multicore systems, perhaps the most promising work is an approach that maps the power consumption of the various cores to the power consumption on an analogous network model (Eisley et al., 2006). This model was simulated with LUNA, a high-level network power analysis tool (Eisley and Peh, 2004), to give a reported 9% error in most cases. There have also been tools developed explicitly to measure the energy in high-performance systems (Ge et al., 2010), but they have not been adapted or used for modeling or prediction.

Power modeling for the GPU, in particular, is far less advanced. There have been multiple works published that advocate the use of GPUs for general-purpose computing from an energy standpoint, observing that though they require more power, their higher speeds reduce overall energy (Rofouei et al., 2008; Huang et al., 2009). A framework called PowerRed, which was designed for exploring power efficiency in GPUs, seems very promising, but it has not been validated or run on real-world graphics applications, only short tests (Ramani et al., 2007). QSilver, a GPU simulator with power analysis capabilities, can be a very powerful tool, but requires time to build a model and simulate existing application traces (Sheaffer et al., 2004). This tool remains, to my knowledge, unvalidated.

Recently, there have been several groups looking at statistical approaches for power and performance modeling of GPUs. Nagasaka et al. examine two NVIDIA graphics cards and use benchmarks in the Compute Unified Device Architecture (CUDA) software development kit (SDK) to develop a linear regression fit for a number of exposed performance counters (Nagasaka et al., 2010). Their model achieves an average of 4.7% relative error for a set of GPGPU kernels, but does not include many graphics-specific operations, such as rasterization and texture fetches. Zhang et al. also looked at a statistical approach for finding performance and power characteristics of a GPU, though for a specific architecture made by Advanced Micro Devices (AMD), and discussed the relative importance and interdependence of various metrics (Zhang et al., 2011). They find that, for this particular architecture, making full use of special hardware for writing to the GPU's dynamic random-access memory (DRAM) is essential to achieving full performance. Further, they show that significant power can be saved by slightly under-utilizing the hardware, though the energy consumption increases due to

the performance loss.

Hong and Kim take a different approach (very similar to mine), in which they measure the energy of different operations directly via micro-benchmarks and compute the cross-product of energy costs and operation frequencies to find the overall power consumed (Hong and Kim, 2010). However, like the above statistical models, they focus on GPGPU applications and do not integrate operations unique to graphics workloads.

3.3 Approach

Developing an energy model for graphics applications on a GPU is challenging because (i) many types of operations, both arithmetic and memory accesses, occur in a typical graphics pipeline; and (ii) while some parts of the pipeline are fixed-function, other parts are user-programmable. The mix of arithmetic versus memory accesses and fixed-function versus programmable hardware make it difficult to accurately predict what processing will occur for any given frame of an application. My methodology consists of carefully applying targeted tests to find the energy of arithmetic and memory operations, and that of the fixed-function and programmable stages. In particular, I use NVIDIA's CUDA—a framework for running general-purpose programs on GPUs—to determine the energy usage in the programmable stages. For the fixed-function units, I develop targeted graphics applications that stress only the unit in question, isolating its energy usage. For each of these tests, I measure the actual energy consumed by the GPU by measuring the current drawn by the hardware. I then use these measurements to develop my energy model.

I validate my model against existing applications with different types of workloads. Two different configurations for a frame of "Half-Life 2: Lost Coast" (Valve, 2005) are used—high and low graphical quality at a high resolution (1600x900 pixels). The test frame from "Batman: Arkham Asylum" (Eidos Interactive Ltd., 2009) and "Mass Effect" (BioWare, 2007) have a large amount of input geometry ($\geq$300,000 triangles), very arithmetic-intensive vertex and pixel shaders, and a modest resolution (1024x768 pixels). "Mass Effect," though, makes use of occlusion queries to minimize shading work. These three test applications are shown in Figure 3.1.

(a) Batman: Arkham Asylum

(b) Half-Life 2: Lost Coast

(c) Mass Effect

Figure 3.1: Test applications used to validate my energy model: "Batman: Arkham Asylum" (a), "Half-Life 2: Lost Coast" (b), and "Mass Effect" (c).

3.3.1 Instruction-Level Energy Measurements

The key idea of my model is that the total energy consumed for a frame of a graphical application can be estimated by the sum of the energy used in each of the operations—arithmetic and memory—performed in rendering that frame. This is similar to Tiwari et al.'s approach to software power estimation (Tiwari et al., 1996). So, my goal is to find a representative value of energy for each operation performed in a GPU. There are two types of components where energy is consumed: in programmable units, such as the vertex and pixel shaders, and in fixed-function units, such as texture filtering and rasterization. In the programmable units, I focus on floating-point operations, as they are the most expensive and common arithmetic in practice. However, I also categorize different types of memory transactions: loads and stores to both local and global (on- and off-chip) memories. Below, I describe the process used to measure the energy in programmable floating-point operations, memory operations, and fixed-function operations.

To measure the energy used by operations on the GPU accurately, I must first decouple the computer's power supply from that of the GPU. I use a peripheral component interconnect (PCI) riser, the PEX16LX made by Adex Electronics, Inc., to lift the GPU from the motherboard to accomplish this (Adex Electronics, Inc., 2008). This allows me to interrupt the power lines and supply my own metered 12V and 3.3V rails to the GPU. I also supply the GPU fan with its own 12V supply, which is not counted towards the energy measurements, in case dynamic fan control changes the current drawn during a running experiment. So, I have isolated the energy consumed by the GPU (NVIDIA's 8300GS in this case).

A similar procedure is used for each of the instructions I wish to examine. I first design a CUDA kernel, or program to be run on the GPU, that will stress the operation in question. The key features of these kernels are that they include a minimum of overhead operations (loop counters, initialization, etc.), do a large amount of work to allow for an experiment of significant length for accurate timing and current measurement, and exhibit high utilization of the CUDA cores with as few data hazards as possible. As an example, the kernel I developed and used to measure the energy costs of addition is shown in Listing 3.1. At execution, the data (an array of 1,536 random floating-point values) is first transferred to the GPU's memory before any timing or measurement begins. Timing begins at the first execution of the kernel and continues for up to a minute, depending on the operation under investigation. During this time,

Table 3.1: Energy used by various floating-point operations in programmable units of the GPU.

Operation	Energy (nJ)
ADD	0.443
MUL	0.357
MAD	0.455
RCP	2.440
EXP	1.512
LOG	5.177
SIN/COS	22.997
POW	16.366

the current drawn remains steady, as the workload is regular and constant. At the conclusion of execution, the timing stops before transferring the data back to the host memory, giving me just the time taken for computation. This allows me to compute the energy taken for all the computations, which then leads to an average energy for a single operation.

Programmable Floating-Point Operations

For the programmable portions of the graphics pipeline, such as the vertex and pixel shaders, I do not actually run any graphics applications. Instead, I take advantage of NVIDIA's CUDA (Lindholm et al., 2008), which allows me to map an operation to execution on the device more directly. Since programmable stages of the graphics pipeline execute on the same processors that are responsible for CUDA's computations, I can measure the energy of the necessary operations with less uncertainty about what is actually taking place. In the context of a graphics processor, there are many often unseen optimizations and other operations that take place without the programmer's knowledge. CUDA, however, allows for execution of the bare arithmetic operations.

The floating-point operations used most often by the programmable graphics pipeline are as follows: `ADD`, `MUL`, `MAD`, `RCP`, `EXP`, `SIN/COS`, and `LOG`. Other instructions are available in the programming units, such as `DP3`, `DP4`, `NORM`, and `LRP`, but their energy requirements can be approximated by their constituent operations. For example, a `DP4` is made up of 1 `MUL` and 3 `MAD` instructions, in both implementation and my energy model. Table 3.1 gives the energy usage for each of the measured operations.

Listing 3.1: An example CUDA kernel used to measure the energy per floating-point addition performed on a GPU

```
__global__ void DoAdditionOnDevice (float *data)
{
    // set up number of iterations, index of data,
    // and temporary local storage (registers)
    const int iters = 392;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float temp[4];

    // populate the temporary storage
    #pragma unroll
    for (int j = 0; j < 4; ++j)
        temp[j] = data[i] + j;

    // perform repeated additions
    #pragma unroll
    for (int j = 0; j < iters; ++j)
    {
        temp[0] += temp[2];
        temp[1] += temp[3];
        temp[2] += temp[0];
        temp[3] += temp[1];
    }

    // prevent optimization
    #pragma unroll
    for (int j = 1; j < 4; ++j)
        temp[0] += temp[j];

    // store the final value (prevent optimization)
    data[i] = temp[0];
}
```

Table 3.2: Energy per operation (4-byte word) for different types of memory accesses performed on the GPU.

Operation	Energy (nJ)
Local load	1.49
Local store	1.49
Global load (coalesced)	8.39
Global store (coalesced)	5.19
Global load (uncoalesced)	67.4
Global store (uncoalesced)	42.7

Memory Operations

Typical graphics cards have both a large off-chip DRAM for storing geometry, texture information, and other persistent data, as well as a pool of smaller on-chip static random-access memory (SRAM) for storing intermediate data and caching accesses to main memory. In addition, for the architecture I examine, there are hardware optimizations that take advantage of reading or writing 32 consecutive bytes of data stored in global memory (DRAM), such as reading a cache line (or portion thereof) from global memory into the cache. In this case, the global load is called *coalesced*. If, however, the data is accessed in a more random manner, the load will be *uncoalesced*. Coalesced accesses (both reads and writes) are much faster than uncoalesced accesses. So, it is likely that they have different energy characteristics, as well, requiring separate treatment.

I explored each of the six types of memory accesses in a manner very similar to that described for floating-point operations. I developed a kernel to stress only the particular operation in question, executed a set number of iterations, and measured for both timing and current information, leading to an average energy per operation. (An operation is reading or writing a 4-byte word, which is typical for floating-point and integer values.) These values are shown in Table 3.2.

There are two trends to note about these results: first, when compared with the simple arithmetic operations, memory operations require around an order of magnitude more energy, on average. Second, uncoalesced accesses are 8 times as expensive as coalesced accesses, reflecting the ability of coalesced accesses to read 32, rather than 4, bytes at a time.

Fixed-Function Operations

Some parts of the graphics pipeline are not user-programmable, but they still play a role in the overall energy consumption of the device. In particular, the two most often-used fixed-function units in contemporary graphics are the rasterizer and texture filtering units. Since the behavior of these units can not be captured by CUDA's hardware and usage patterns, I make use of the graphics drivers to exercise them and measure their characteristics.

Rasterization To determine the energy used in rasterization, I designed a pair of experiments that would perform the same work, with the exception that one exper-

Table 3.3: The energy cost of fixed-function hardware.

Operation		Energy/Pixel (nJ/P)
Rasterization		0.2384
Texture Mapping		
Filtering	Mipmapping	
Nearest	-	13.3
Bilinear	-	13.8
Nearest	Nearest	7.07
Bilinear	Nearest	7.76
Bilinear	Linear	10.6

iment would not include any rasterization. The first experiment sent a single unlit, untextured, screen-sized rectangle through the card: the vertices were transformed and assembled into triangles which were then sent through the rasterizer. However, depth-testing was used to keep any generated fragments from undergoing further processing—I set it to reject all fragments, regardless of depth. In the second experiment, I made use of triangle backface culling. If the vertices of the triangle are sent in reverse order, the hardware can treat it as a triangle that faces away from the virtual camera and not process it any further, a common optimization in real-time graphics. The only difference between the two experiments was that rasterization was performed in one and not the other.

I conduct a variety of rasterization tests with varying window sizes and frame rates (this can be controlled via vertical-syncing, or timing the scene redraw with the refresh rate of the monitor) and perform a linear regression on the results, fitting a line to a plot of frames per second versus power. The slope is power/frame/second, or energy per frame, and the difference between the two slopes is the energy required for just rasterization. The energy results are given in Table 3.3.

Texture filtering When a triangle is textured, there is seldom a 1-to-1 pixel to texel ratio; the texture is usually filtered in some way so a pixel can be assigned a texel value. This filtering is performed at run time, although many textures are also pre-filtered for use when texturing a smaller triangle. A texture that has had smaller versions of itself created is said to be *mipmapped* (Williams, 1983); each smaller version is known as a *mipmap level*. This allows for the hardware to load smaller textures into the texture cache, yielding greater performance. When using mipmaps, bilinear filtering

can either pertain to just one mipmap level, or the texture can be filtered between two levels, yielding a trilinearly interpolated value for a single pixel. If a texture is not filtered, it is said to use *nearest* sampling, because texels are assigned to pixels based on simple proximity.

I used a very similar approach to measuring the energy of texture filtering as my approach to rasterization. I sent a single, unlit, screen-sized rectangle through the pipeline and varied the texturing applied to it. However, I disabled color writes, depth testing, depth writes, and stencil testing so that no unnecessary work was done and all work stops after the shading of the fragments. In this case, the only shading work done was texturing. The average results for various types of filtering to map large texture onto different-sized triangles are shown in Table 3.3.

Why is the most complex filtering less expensive than not doing any at all? The two steps in texture mapping are (i) fetching and (ii) filtering the data. Mipmaps allow more of the texture to fit into the cache, greatly reducing the amount of DRAM traffic. The hardware that performs the actual filtering is very specialized and efficient, so performing more complex filtering within any mipmapping *is* more expensive, but the energy is more than reclaimed by the lessened memory traffic.

Finally, I have omitted incoherent texture mapping, in which the texture-space texels do not align on corresponding screen space pixels. In my model, though, I do allow for adding a penalty if a particular texture fetching operation is known to be incoherent (see Section 3.5.2 for an example). However, most texture caches, both by design and due to their usage patterns, have very high hit rates: usually 97% or higher (Al Maashri et al., 2009).

3.3.2 Frame-Level Energy Prediction

With the energy for all of the operations performed on a GPU measured, it remains for me to determine a methodology for modeling the frequency of each operation for an arbitrary program. I adopt a strategy similar to that used in the past (Molnar et al., 1994) and assign values to various input parameters (primitive count, primitive size, resolution, etc.) and follow them through the graphics pipeline (abstracted in Figure 1.1) to see how workloads change. I also need a count of the operations performed in the different programmable units, which can either be known (in the case of modeling an application in development) or approximated (if the source code is unavailable). Furthermore, I must make some assumptions about values that are not readily available,

such as cache sizes and performance, compression ratios, and depth test efficiency. Below, I explain the model at each stage of the pipeline.

Input Vertices

The rendering pipeline is fed by primitives, usually triangles which contain various data, or attributes, at each vertex. (Rarely, general polygons are used as input, in which case the first step performed by the hardware is to decompose these polygons into triangles. Triangles are preferred and the most common input type, however, since polygons with more vertices do not have a unique triangulation, which can lead to ambiguous plane formation.) Typically, these attributes are in a floating-point format, either 32 or 16 bits, and contain information such as position, normals, one or more texture coordinates, color, and other values. Each primitives is made up of vertices, but these vertices may be shared between primitives; therefore, the vertex to triangle ratio approaches 1 in the ideal case, but is usually closer to 2 in my test applications. The energy in this stage is directly related to the number of input vertices, v_{in}, and the data per vertex, d_v, as seen in Equation 3.1 (where E_{cgr} is the energy required for a coalesced global read. Uncoalesced, local, or write operations are subscripted similarly in later equations).

$$E_{VI} = v_{in} * d_v * E_{cgr} \tag{3.1}$$

Vertex Shading

After the vertex data has been transferred from main to local memory, the vertices enter a programmable unit known as the vertex shader, where they are transformed from a local to a screen space coordinate system through matrix multiplications. Optionally, the vertices are lit or textured, though these operations are more commonly performed later, in the pixel shader, for higher quality results. The energy spent in this stage is the product of the number of vertices which undergo shading and the energy of the operations contained in the vertex shading program, collectively shown in Equation 3.2:

$$E_{VS} = v_{in} * \sum_{op}^{numOps} E_{op} \tag{3.2}$$

Rasterization

After the vertices are processed, they are assembled into screen space triangles and sent to the rasterization stage, where a fragment is generated for each pixel the triangle covers. The energy spent in rasterization is a function of the number of generated fragments, which is roughly the product of the display triangle size in pixels and the number of display triangles, as shown in Equation 3.3:

$$E_R = t_{disp} * size_{t_{disp}} * E_{raster} \tag{3.3}$$

Depth Testing

The generated fragments streaming from the rasterizer are ultimately bound for fragment shading. However, by testing the depth of a generated fragment against the current minimum depth of fragments at its position, the hardware may be able to discard some or all of the fragments for a triangle before shading them. Another common optimization is the use of a hierarchical depth testing acceleration structure: if the depth of a fragment is greater than the greatest depth for a large area of the screen, then the fragment does not have to be tested at a finer granularity (which can be as fine as a per-pixel test). Both these factors contribute to the efficacy of the depth test, eff_{z-test}. Additionally, depth values are often compressed to cut down on reading and writing bandwidth, represented by a z_{comp} factor. The energy in this stage is spent reading and writing depth values from main memory through a dedicated cache; thus, it will depend on the cache's performance (zc_{hit}) and line size (zc_{ls}). The model is shown in Equations 3.4-3.6:

$$E_{D_{read}} = (f * eff_{z-test}) * (zc_{hit} * E_{clr} + (1 - zc_{hit}) * zc_{ls} * E_{cgr}) \tag{3.4}$$

$$E_{D_{write}} = (f * eff_{z-test}) * (1 - zc_{hit}) * zc_{ls} * E_{cgw} \tag{3.5}$$

$$E_{D_{total}} = (E_{D_{read}} + E_{D_{write}}) * z_{comp} \tag{3.6}$$

Fragment Shading

All fragments that pass the early depth test successfully are sent through the fragment shader, which will produce the final color for the fragment. Operations here often

include lighting and texture mapping. Much like the vertex shader stage, the energy spent in fragment shading is the product of the number of fragments and the energy of the operations that are applied to them, shown in Equation 3.7:

$$E_{FS} = f_{in} * \sum_{op}^{numOps} E_{op} \tag{3.7}$$

Framebuffer Operations

Once the fragments' color values have been generated, they need to be sent to the framebuffer so they can be sent out to the display during "scan out." They are first sent to a color cache, so that different processors working on neighboring pixels can combine their writes. Also, framebuffer data is often compressed in order to cut down on high bandwidth costs. So, the energy spent in sending data to the framebuffer is related to the number of fragments generated, data per fragment (usually 4 channels of 8-bit color), d_f, color cache hit rate, cc_{hit}, color cache line size, cc_{ls}, and framebuffer compression ratio, fb_{comp}. My model for writing to the framebuffer is given in Equation 3.8:

$$E_{F_{write}} = f * fb_{comp} * (d_f * cc_{hit} * E_{clw} + (1 - cc_{hit}) * cc_{ls} * E_{cgw}) \tag{3.8}$$

Reading from the framebuffer occurs when pixels are to be blended, most commonly when rendering a translucent surface. Since this is not the common case, an extra parameter is necessary for my model—the alpha blending ratio, a_{ratio}. My model for reading from the framebuffer is shown in Equation 3.9:

$$E_{F_{read}} = f * fb_{comp} * a_{ratio} * (1 - cc_{hit}) * E_{cgr} * cc_{ls} \tag{3.9}$$

3.4 Validation

With my energy model fully developed, I now test it against actual applications to see how well the composite equations perform. The applications are a selection of video games with different characteristics, to represent varied real-world workloads. "Batman: Arkham Asylum" has a medium geometric complexity but very arithmetic-intensive shaders, and it was running with a window size of 1024x768 pixels (a relatively low resolution). "Half-Life 2: Lost Coast" has less input geometry and simpler shaders, but

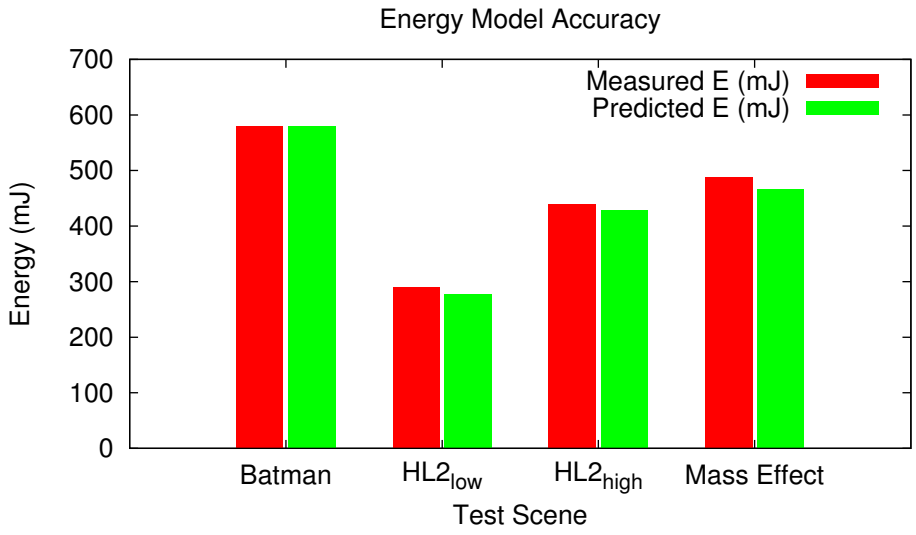


Figure 3.2: Accuracy of the developed energy model. The relative errors ranged from 0.14% to 4.6% with an average error of only 2.9% when the framerate was assumed to be in the middle of the two bounding integers reported at runtime.

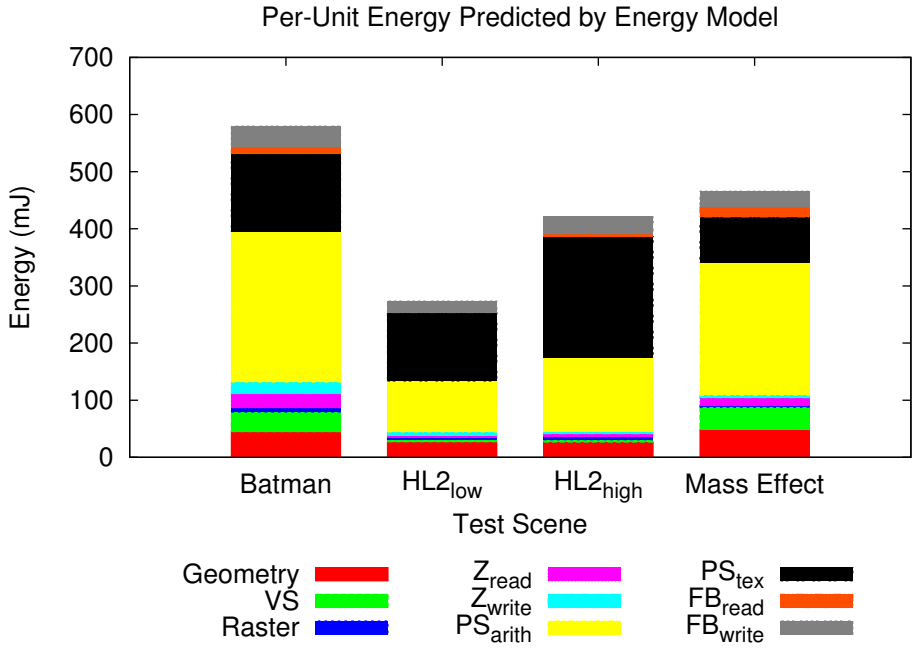


Figure 3.3: Energy used per stage of the graphics pipeline.

a much higher resolution: 1600x900 pixels. "Mass Effect" is quite similar to "Batman," but performs occlusion queries to minimize unnecessary shading. These test frames can be seen in Figure 3.1.

To ensure accurate and repeatable results, the graphics loads were kept constant, meaning the frame rate, work performed, and current drawn were measured at steady, specific values. I then subtract the idle values to find the energy of the graphics work. While these values were steady, a single frame was captured with Microsoft's PIX for Windows, a tool included in the Microsoft DirectX SDK used for debugging graphical applications developed with DirectX.

From the data gathered with this tool, I was able to extract, for each individual draw call within the frame, the characteristics (triangle count, included attributes) of the input geometry, render states (alpha blending, depth testing), vertex and pixel shader code, and the contents of the framebuffer. From this last data, I have approximated the number of shaded and depth-tested fragments for each draw call. Essentially, I gathered all the input parameters that are not specific to the hardware that I need to populate my model. I briefly give and justify the values I chose for parameters that are not able to be derived from the experiments.

1. Cache performance. There have been no publicized cache performance figures from existing hardware, so I look to other research. An exploration in 3D chip stacking simulates several graphics applications and reports values of close to 100% and 95% for the texture and depth caches, respectively; I assume the color cache behaves likewise (Al Maashri et al., 2009).

2. Cache line size In the same vein as cache performance, I look to other sources for cache line sizes. There is no validated work, so I chose a line size of 128B, which is in line with other used values (Al Maashri et al., 2009).

3. Compression rates Color and depth buffers are often compressed as rendering is performed, and texture data is nearly always compressed before being released with the game. Both of these compression areas will decrease the amount of global memory traffic. For texture data, I used a common compression rate of 4:1 (Microsoft Corporation, 2012b). Data going to and from the frame buffer was assigned an average rate of 2:1.

There are two parameters that are excluded from this harvesting process: intra-draw

call overdraw and local data traffic. There can be pixels rendered to the scene that are immediately overdrawn in the same draw call, but this information is impossible to find with the infrastructure I have available. Further, local data traffic is heavily dependent on the architecture and specific techniques used on the hardware, such as redundancy elimination (Wittenbrink and Ordentlich, 2005). These two missing parameters will likely lead to energy estimates that are lower than the actual consumption for the same scenes.

The results of my model's validation are given in Figure 3.2. The relative errors are very small—between 0.14% and 4.6% with an average of only 2.9%. These estimates, however, could be off by as much as ±5–10%, depending on the actual frame rate of the application due to inaccuracy in the measuring of the frame rate. I then separate the energy used by the whole GPU into discrete energy consumption values stage by stage, shown in Figure 3.3. I see that the pixel shader is by far the most energy-hungry stage, both when considering arithmetic as well as texture fetches, followed by memory transactions (reading input geometry, reading and writing the depth buffer and framebuffer) and vertex shading.

3.5 Case Studies

I now explore two potential uses for the energy model I have developed: exploring the impact on energy of (i) architectural and (ii) algorithmic modifications.

3.5.1 Architectural Study

The most popular high-performance graphics architectures are currently all very similar and fall under the sort-last-sparse classification (Molnar et al., 1994). (It is into this category that the NVIDIA 8300GS used in validating the model falls (Lindholm et al., 2008)). In the mobile market, however, the leading architecture is known as a tiled renderer (Imagination Technologies Ltd., 2010), a type of sort-middle architecture (Molnar et al., 1994). Briefly, a tiled renderer will process the scene one screen space tile at a time; all geometry is sorted into the appropriate tiles, the tile is rendered to a local rather than global framebuffer memory, and only the finished tile is sent to global memory for scan out.

When designing for the mobile realm, battery life is of the utmost importance, and proponents of tiled renderers claim that the very coherent writes to local framebuffer

memory will offset any overhead in reading and transforming some fraction of the input geometry more than once. However, there has been no published verification of this claim. By adapting my energy model's underlying architecture to that of a tiled renderer, I can gain insight into the veracity of this assertion. Since validating this new model would be impossible without fabricating an entirely new architecture, I perform several sanity checks before applying the model to my test scenes.

The possible search space for this question is enormous, so I simplify the problem somewhat. First, I will only explore three different parameters to check the validity of the new model: (i) input geometry count, (ii) screen size, and (iii) the depth complexity of the finished scene. (Depth complexity is a measure of how much work is performed shading pixels that do not appear in the final image. The three test applications have a depth complexity of between 4 and 5. Very complex scenes can have a depth complexity as high as 30 in some limited testing I performed.) The baseline scene will have 100,000 triangles, a screen size of 1280x1024 pixels, and a depth complexity of 3. Other scene assumptions are: 48B of data per input vertex, 16B per fragment, 1.8 vertices per primitive, a depth fail rate of 0.5, an alpha blending rate of 0.25, framebuffer and depth compression ratios of 1.5, and vertex and pixel shaders with equal complexity. Additionally, I make the following assumptions in my tiled renderer's energy model:

1. The added cost of pre-sorting the geometry will be one read of the input geometry, a pass through the vertex shader, and a write of a batch ID for every 32 input primitives,

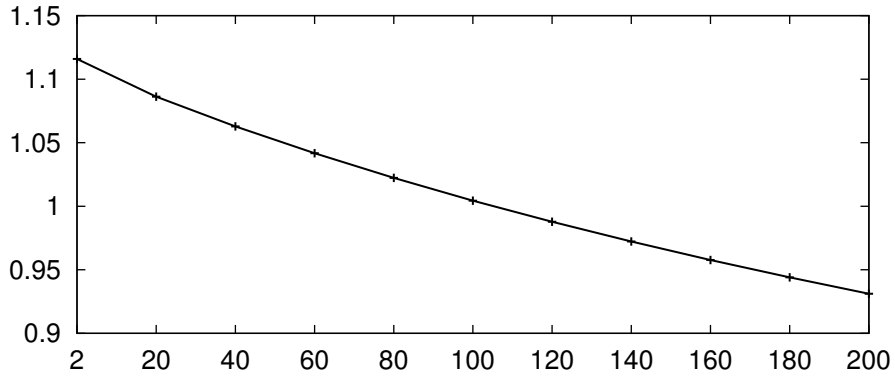
2. There is a local framebuffer storage of 2MB,

3. The tile size will be fixed at 128x128 pixels, and

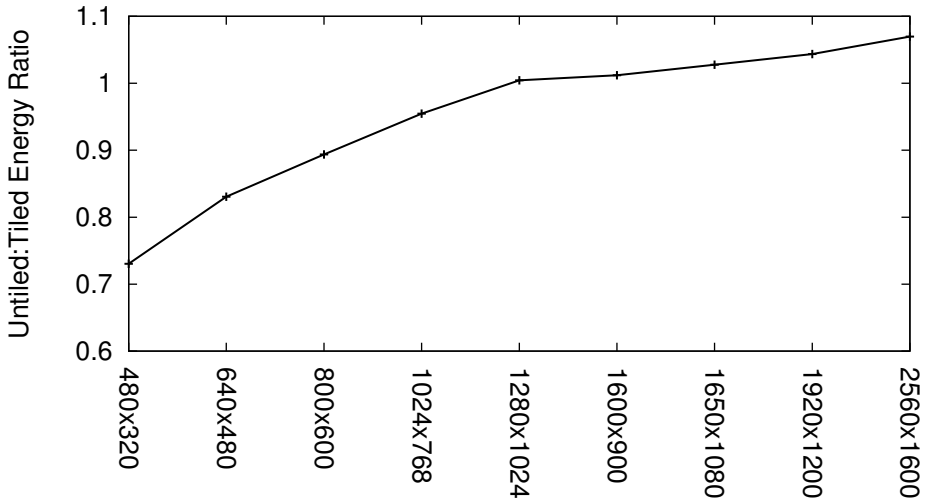
4. The depth buffer is *not* stored to global memory after a tile is finished processing.

The results of my three experiments are shown in Figure 3.4.

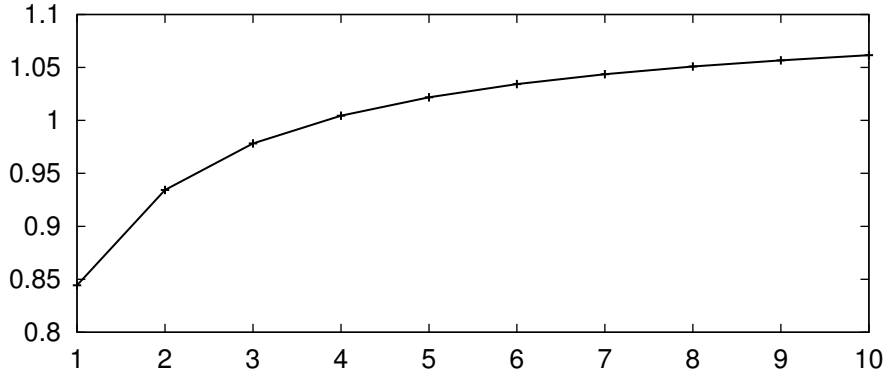
1. Input geometry count. Increasing the input geometry directly increases the overheads seen by a tiled renderer. Thus, the tiled renderer becomes relatively less energy-efficient as the geometry count increases. (I assume that when increasing the triangle count of a scene, the extra geometry will be put towards refining meshes, decreasing the size of the average triangle, therefore keeping the generated fragments and depth complexity the same.)



(a) Triangle Count (thousands)



(b) Screen Size



(c) Depth Complexity

Figure 3.4: Energy efficiency of tiled versus untiled renderers for different scene parameters. A ratio greater than one indicates that the tiled renderer is more efficient.

2. Resolution. The strength of the tiled renderer comes from its ability to write to local memory during framebuffer operations. As resolution increases, the locality of these writes and the disparate nature of the full-screen renderer's cause the relative efficiency of the tiled renderer to increase.

3. Depth complexity. Closely related to resolution is depth complexity. I note a similar trend: as more pixel processing and framebuffer writing is performed, the relative efficiency of the tiled renderer increases.

When applying the new tiled renderer model to my existing test applications, I see that they are all less efficient with a tiled renderer (13% on average). While this would seem to suggest that this architecture is less efficient, there are several things to consider before accepting this outcome at face value. Firstly, my naïve approach to the tiled renderer's architecture certainly lacks optimizations employed by implemented hardware. For example, it is doubtful that the price paid to presort the geometry is a doubling of the initial effort, and the sorted geometry may even fit in on-chip memory. Secondly, the test scenes I examined all had complex input geometry; they were not meant to be run on mobile hardware! Developers would likely optimize their geometry and applications for such an environment.

To test how these applications would consume energy on an existing mobile platform, I adapted them to have a workload more characteristic of mobile applications. So, I first scaled the applications to be the resolution of a current mobile device, the iPhone 4: 640x960 pixels. Next, I treated the amount of input geometry as on par with the peak triangle rate of this device at 30 frames per second. The results after these modifications are shown in Figure 3.5. The main difference is that reading geometry and vertex shading steal some energy away from pixel shading due to the relatively small screen size; this is an expected result.

3.5.2 Algorithmic Study

When designing a graphics application, the developer often has to choose between many algorithms to achieve a certain effect. For instance, dynamic shadows can be implemented with a number of techniques, such as shadow maps or shadow volumes. Usually, developers will look at such metrics as rendering time, programming cost, or memory overheads when deciding which algorithm to choose. However, it may be advantageous to consider the energy efficiency of different algorithms, as well, which

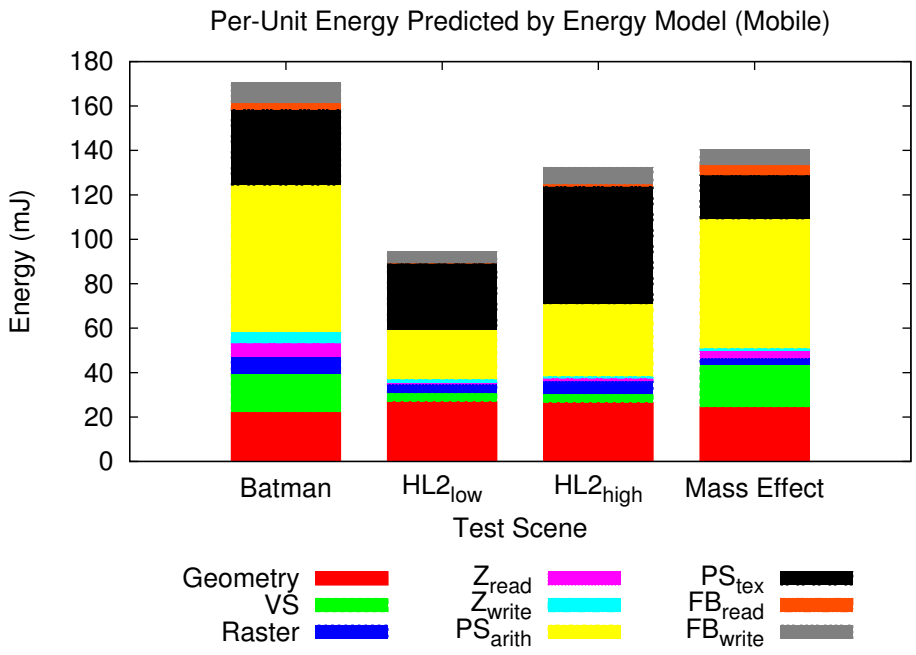
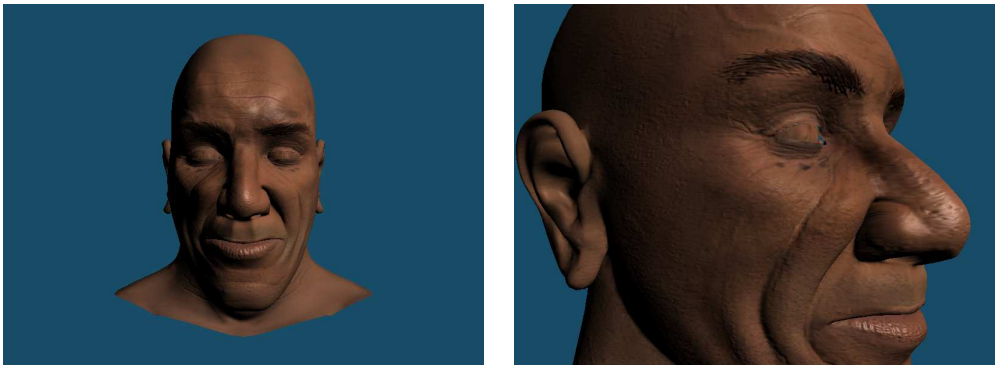


Figure 3.5: Energy used by modified applications per stage of a hypothetical mobile graphics pipeline.

my model allows.

As an example, a typical cost/benefit analysis of different algorithms for a graphical technique looks at the use of a cube map texture to normalize a number of vectors used in a common pixel shader effect known as bump-mapping (NVIDIA Corporation, 2004). These vectors are the eye-space position of the fragment (V), the eye-space light vector (L), the computed half-angle vector (H), and the computed normal (N). In some instances, the texture lookup can be faster than normalizing a vector arithmetically. I use this demo and my model in a similar experiment to see which approach is more energy efficient. I analyzed two frames of this demo, one where the model took up a small fraction of the screen and a second frame where the model covered nearly every pixel, both in Figure 3.6. The predicted and measured energies are shown in Figure 3.7.

In the first test case, it is important to note that the amount of work being done is relatively small, since there is only one model which does not take up much of the screen. Therefore, any overheads (absolute error) not accounted for in my model will play a much larger role in the overall energy. In the second experiment, however, the error between the measured and predicted energies is much less—around 11% on average, down from 35% in the first test case. This indicates that as more work is



(a) Bump-mapping test 1 (b) Bump-mapping test 2

Figure 3.6: Test scenes from the bump-mapping application.

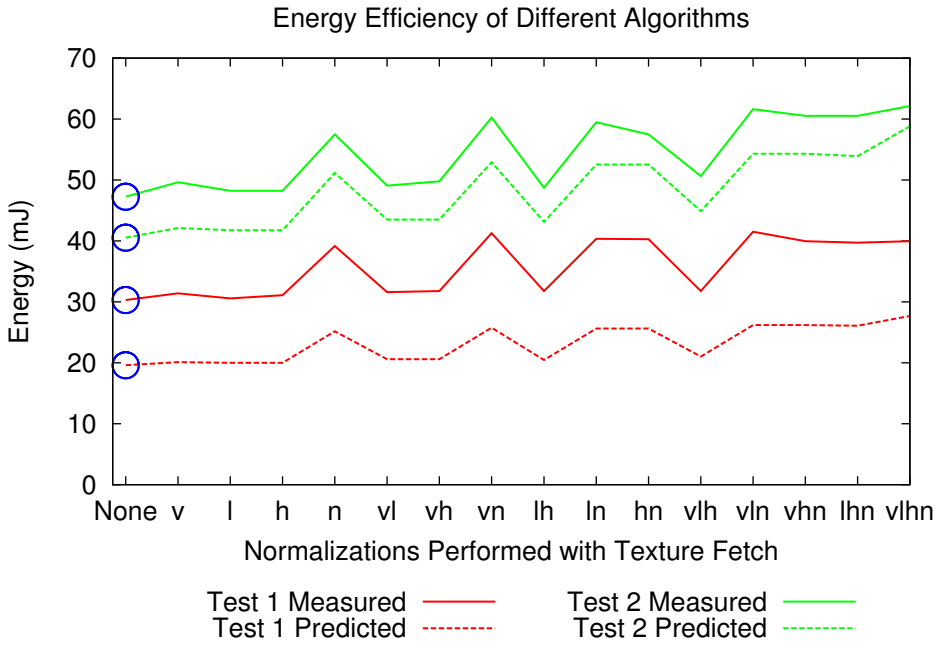


Figure 3.7: Results of the algorithmic change experiment. My model correctly predicted the best performing algorithm (circled) for both test cases.

performed, the model will become more and more accurate. Indeed, in the second case, the model accurately predicted the best and worst performing algorithms from an energy standpoint. Even with its high relative error, the model correctly predicted the best algorithm for the first test case.

This experiment demonstrates both the flexibility and necessity of my model. The original study into the rendering efficiency of the different algorithms noted that using a texture to normalize the N vector was always slower than the corresponding case that normalized N arithmetically. Since that particular vector varies rapidly in screen space, leading to incoherent accesses, it is not able to make use of the texture cache as well as the others. So, there is a penalty included for only that fetch which is known a priori and was included in my model to give more accurate results. It should also be noted that speed is no indication of energy-efficiency. While the setup labeled "lh" is the fastest in both experiments, it was not the most efficient, indicating that a model such as mine is necessary for developers looking to minimize the energy their application uses without resorting to cumbersome monitoring—it is unlikely that the average graphics developer will have access to or the time to set up power monitoring hardware.

3.6 Conclusions

I have presented an accurate energy model for GPUs that allows architectural, algorithmic, and other experimental changes to be explored without the implementation of new hardware, complicated simulations, or instrumentation of existing hardware. I also validate my model against a variety of existing graphical applications to prove its accuracy in practice, with less than 10–15% error in these tests. Different games or drastically different scenes may require new parameters to be introduced to the model to make it more accurate. With this model, I now know which parts of the pipeline consume the most energy, so I can target reducing their consumption for the largest effect on the total energy. Also, as I make stages more energy efficient through some means, I will have an accurate estimate of the overall savings gained.

Chapter 4

Variable-Precision Arithmetic Circuit Implementation

4.1 Motivation

As I showed in Chapter 3, the vertex and pixel shaders consume substantial amounts of energy in the graphics pipeline. Since these two shaders actually run on a single pool of general-purpose processors in modern hardware, developing an energy-saving strategy for one stage will likely lead to energy savings in the other, too. I update Hao and Varshney's variable-precision rendering techniques to today's GPUs, so a natural step is to find or build hardware capable of trading off precision for energy savings. Since 32-bit floating-point numbers with full precision are not necessary to perform many rendering tasks, the goal will be to reduce the amount of computation that is done in order to save energy.

This variable-precision arithmetic hardware must be able to limit both its dynamic *and* leakage power (see Chapter 2.1), so clock gating by itself is not sufficient; some form of power gating will be necessary. Furthermore, initial experiments revealed that the precision of the arithmetic must be variable at a very fine level, possibly down to a per-bit granularity, but also must be able to operate at full-precision for some graphical and scientific applications. (Not every program can tolerate having its precision reduced.) Clearly, a high-performance environment like a GPU cannot afford to have its throughput decreased by the inclusion of variable-precision hardware; this hardware must not negatively impact performance. The necessary circuits are the building blocks of a full FPU: integer adders and multipliers. These basic arithmetic circuits constitute more complex units used in graphics hardware and are responsible for the precision of

a given floating-point operation.

To put a fine point on the requirements of the variable-precision arithmetic hardware needed for saving energy in a GPU, let me enumerate them here. The circuits need to be:

1. integer adders and multipliers (that will be used in a full FPU),
2. power gated, so that leakage power will be reduced, as well as dynamic power,
3. variable-precision at a fine granularity,
4. not significantly slower than the original hardware, and
5. dynamically reconfigurable.

I will go over many past techniques and approaches for tackling this problem in the next section and will show that no existing work addresses each of the requirements listed above. So, the rest of this chapter will describe new circuits to enable precision-energy tradeoffs by not computing successive least significant bits (LSBs).

4.2 Related Research

Many methods of power gating have been presented, from simple header and footer transistors to more complex techniques. For instance, if there is a need to save the current state and data stored within a circuit while it is power-gated, Liao et al. and Kim et al. have both proposed structures allowing for this capability (Liao et al., 2002; Kim et al., 2004). However, this is far beyond what is needed for my approach to variable-precision arithmetic; there is no need to store intermediate results in the lower, power-gated bits. So, I chose simpler techniques with lesser overheads that can be applied to each bit of an arithmetic circuit, rather than the circuit as a whole.

There has also been research directed towards low power arithmetic circuit design. Sheikh and Manohar thoroughly examined a floating-point adder and designed a new one piece by piece with aggressive optimizations for energy savings (Sheikh and Manohar, 2010). Liu and Furber presented a low power multiplier (Liu and Furber, 2004), while Callaway and Swartzlander detailed the relationship between power and operand size in CMOS multipliers (Callaway and Swartzlander, 1997). Tong et al. suggested a digit-serial multiplier design with three discrete bit-widths, resulting in a linear power savings (Tong et al., 2000). Lee et al. proposed a variable-precision constant

multiplier that uses shifting in the place of multiplication by powers of 2, realizing an energy savings of between 16% and 56% (Lee et al., 2007). Most similar to my work is that of Huang and Ercegovac, who developed two-dimensional signal gating for variable bit-width multipliers, realizing up to 66% power savings over a baseline implementation (Huang and Ercegovac, 2002; Huang and Ercegovac, 2003). However, their work does not address leakage power, which is a large component of nanometer-scale CMOS hardware. They also look at the layout of the parts of an array multiplier from an energy standpoint, but do not perform any power gating (Huang and Ercegovac, 2005).

Phatak et al. presented a low power adder and included a treatment of the adder's power usage dependent on the number of significant bits (Phatak et al., 1998). Kwong filed a patent for a variable-precision digital differential adder for use in graphics rendering, but has not reported any power results (Kwong, 2005). Park et al. have proposed a scheme in which energy can be traded for quality (similar to this dissertation) in a discrete cosine transform (DCT) algorithm using only three "tradeoff levels" (Park et al., 2010). Other research by Usami et al. and Sjalander et al. has led to variable-precision power-gated multipliers, which *will* save leakage current in smaller processes (Usami et al., 2009a; Sjalander et al., 2005). However, both of these papers only allow for two different operating precisions, while the ability to operate at a full range of precisions is necessary for rendering. (In experiments for Chapter 5, there were many shader programs that could be reduced to, say, 17 or 18 bits of precision, which would not see any savings with hardware that accommodates only 2 or 3 precisions.)

Kulkarni et al. use building blocks that are slightly numerically inaccurate to create a multiplier with bounded error characteristics that saves power over a traditionally precise multiplier (Kulkarni et al., 2011). What's more, they offer a method for trading off error for power, allowing the designer to choose a point along the error-power curve that their application can tolerate, and they allow for exact computations with the use of a residual adder. This is very promising! However, these design choices must be made as the hardware is being built, which precludes the use of this approach for general-purpose hardware. The precision necessary for a GPU's applications can vary wildly from frame to frame, even from one stage of the pipeline to the next, and cannot be fixed in the hardware.

None of these approaches have all design characteristics mandated at the beginning of this chapter. My targeted applications need very fine-grained control over the operating precision; thus, coarse-grained designs which allow for, for example, 8, 16, and 24 bits of precision simply do not offer the necessary degree of control. The use

of power-gating will offer significant returns when also considering the savings in decreased leakage current (Kim and Shin, 2006). Finally, the ability to reconfigure the hardware for different precisions at runtime is imperative for use in a GPU.

The VFloat library is meant to address some of these problems - application-specific precisions, reduced leakage current - but has only been implemented for field-programmable gate array (FPGA) devices (Wang and Leeser, 2010). So, these problems are only solved by actually reprogramming the hardware, which is not possible at runtime.

Specialized hardware for other domains has also been developed to reduce leakage current by power gating the arithmetic hardware in certain ways, such as Ngo and Asari's video processing convolution hardware (Ngo and Asari, 2009). There are key differences between our approaches, though; the convolution of image data lets Ngo and Asari use a priori knowledge, such as the magnitude of common filter coefficients, that I cannot count on in my design. They use this knowledge to optimize circuit paths such as one and zero detection. Also, they can count on the dynamic range of neighboring pixels to be relatively small, leading to optimizations taking advantage of transforming this spatial coherence to temporal coherence from the point of view of the arithmetic logic unit (ALU). However, in a massively parallel GPU, it is not guaranteed that neighboring pixels will be processed on the same ALU, rendering this approach infeasible for my designs.

Other low-power techniques, such as DVFS (Mao et al., 2004) and unit-level power gating (Chowdhury et al., 2008), can be used for energy-efficient graphics hardware. These techniques are orthogonal to this work on fine-grained power gating for variable-precision arithmetic.

4.3 Hardware Implementation

To create new hardware that meets the criteria detailed in Section 4.1, I modified existing arithmetic circuits. I chose three common integer adder designs and looked into different ways of adapting a standard array multiplier. The adders used are a ripple carry, uniform carry-select, and Brent-Kung adders (Brent and Kung, 1982), each with their own strengths and weaknesses. The ripple carry adder is a simple design that uses very little hardware, but has the longest critical path and therefore the longest propagation delay. The carry-select adder is faster but, depending on the implementation, can use nearly twice as much area. The Brent-Kung adder, although

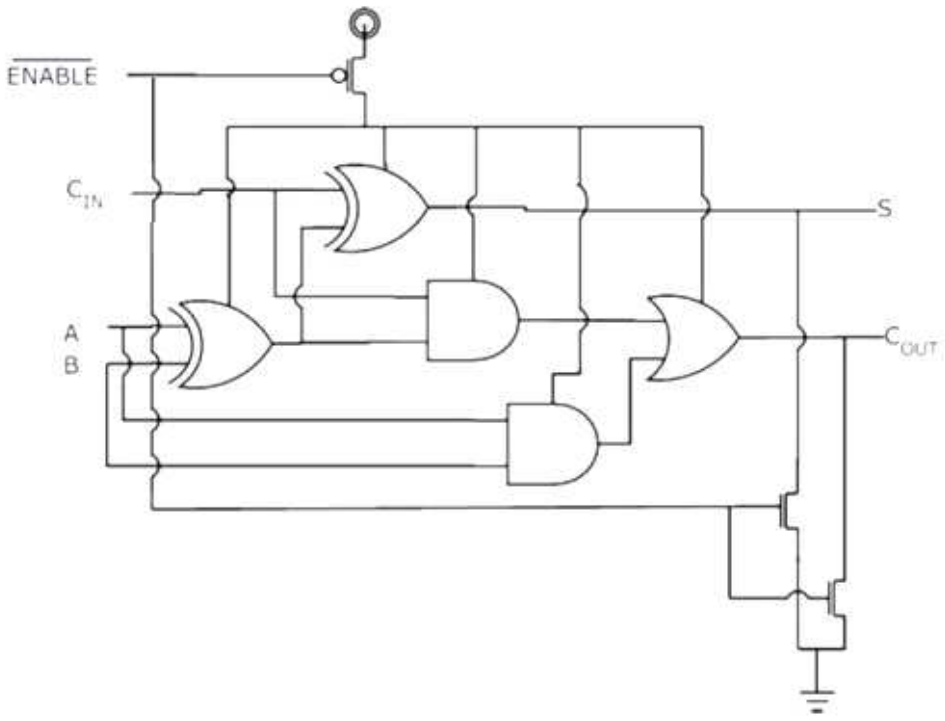


Figure 4.1: A standard full adder modified for use in a power-gated variable-precision arithmetic circuit. Depending on the value supplied on the "Enable" line, the transistors in the gates either receive an actual power source (V_{dd}) or just a floating input, which does not provide a path for current to follow. The transistors connected to the outputs only pull the values low if the block is disabled, providing components downstream from the adder with a constant value.

it has the highest area requirements, is the fastest of the three and is easily pipelined, making it a popular and commonly-used design. It is one of many parallel-prefix adders (Harris, 2003).

Three key modifications were applied to any single component subject to power gating. First, the arithmetic logic transistors were supplied with either a virtual power (header switch) or ground (footer switch) signal controlled by sleep transistors driven by an enable signal, rather than actual power or ground rails. This modification allows the power to the element to be cut off, thereby practically eliminating the dynamic power consumption and potentially reducing leakage power loss through the circuit. When deciding whether to use either a header or footer switch, I consider the power and timing implications of each (Shi and Howard, 2006), as well as the desired output in the disabled state. In the second modification, the outputs were either pulled down (for a header) or pulled up (for a footer switch), depending on the larger context of the element, so that any downstream hardware will see a consistent signal. This both reduces downstream switching and allows for transparent integration with existing hardware; no special treatment of floating signals needs to be considered because the outputs of disabled gates are not floating. Since the state of the output does not need to be preserved when disabled, no extra registers are necessary. Lastly, the logic and gating transistors in the circuit were manually resized in order to minimize the power or timing overheads of the modified designs (Mao et al., 2004; Shi and Howard, 2006; Sathanur et al., 2008). Figure 4.1 shows these changes applied to a standard full adder.

Fine-grained power gating, such as I propose, is subject to problems with ground bounce if large portions of the circuit are switched at the same time. Rush-current suppression can be implemented by skewing the switching of portions of the circuit (Usami et al., 2009b). For my design, I can skew the switching by disallowing very large changes in precision at one time. A possible approach is to have the software driver monitor precision changes and sequence overly large ones as a series of smaller changes.

The operating precision is chosen by setting enable lines to each gated unit. Several approaches are available for correctly setting these enable signals. The most straightforward is to drive each gated element based on a symbolic constant in a register. Alternatively, any manner of decoding circuitry can be used to translate a shorter enable code bundled with operands into individual enable/disable signals. The specific technique used will depend heavily on the application and the usage patterns of the unit. It is highly likely, however, that whatever area overheads are incurred by the

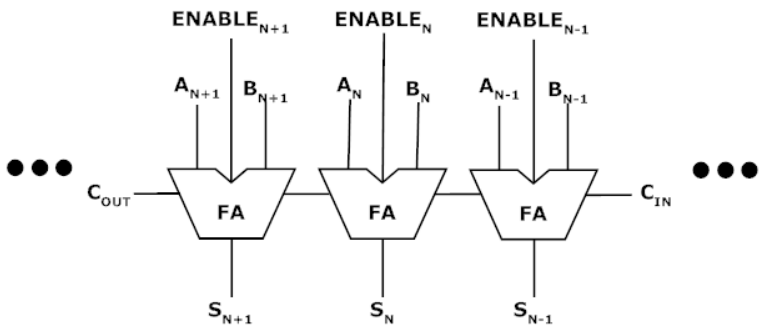


Figure 4.2: A section of a modified ripple carry adder. Each full adder has its own "Enable" signal in order to gate the power used by the unit. It is assumed that if $Enable_N$ is low, then $Enable_i$ is also low for all $i < N$.

control circuitry will be shared over several functional units, over an entire ALU, or even over multiple ALUs.

4.3.1 Modified Adder Designs

Differences in each of the three adders targeted led to distinct approaches to power gating for each. I explore designs of 24-bit integer adders, which are used in single-precision floating-point addition, a common operation in many applications. Past research has shown that, for some target applications, the most readily available savings appear in the first twelve least significant bits of a 24-bit adder, where reduced precision will not have an overly negative impact on application utility (Yoshizawa and Miyanaga, 2006; Chittamuru et al., 2003). I therefore limit the precision control of my proposed designs to the least significant sixteen bits. I note here that though two of the adder designs I explore are rudimentary and not often used in high-performance systems, I show later that they can be more energy-efficient than faster designs. Furthermore, their relatively high latency does not render them useless in a GPU; performance in a GPU is a function of throughput, which can be achieved by many pipelined ALUs with any given latency (within reason).

Ripple Carry Adder

First, let's examine a ripple carry adder. This is a very basic adder whose functionality is immediately discernible, and it will serve as a baseline implementation. A ripple carry adder simply uses one full adder per bit of precision needed by the operands. I modify each full adder as previously described and shown in Figure 4.1. Disabling

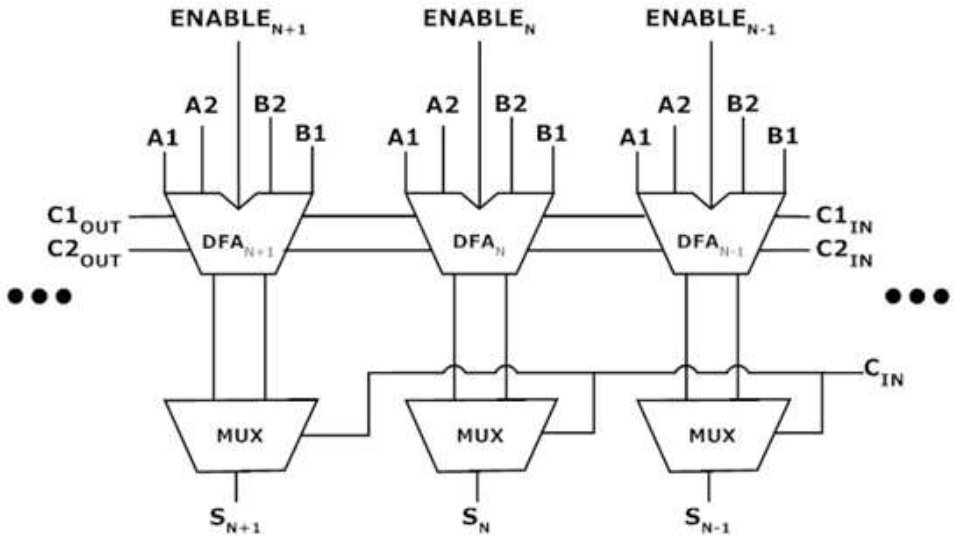


Figure 4.3: A portion of the double full adder chain of a carry-select adder block. Each gated unit is two modified full adders which share the same gating transistor, saving area and timing penalties. The final sum is chosen with a multiplexer driven by the carry-in of the previous block.

each successive full adder has the effect of reducing the precision by a single bit. The modified design is shown in Figure 4.2.

The interested reader may continue in this section for details of the other adder and multiplier designs; otherwise, results are presented in Section 4.5.

Carry-Select Adder

Carry-select adders are slightly more complicated than simple ripple carry adders. They employ several banks of smaller ripple carry adders to make up one full-width adder; each bank computes two carry paths in parallel. When the carry out signal from one block enters the next, multiplexers select the correct carry path to output to the next stage, and so on. The first ripple carry block does not have the second carry path, since its carry-in signal is always '0.' It is treated like the modified ripple carry adder above. The other type of block is made up of two ripple carry chains in parallel. Applying my technique to these blocks involves power gating each parallel pair of full adders as one unit, leading to less power and area overhead than simply using the single full adder approach. Specifically, the tested design was a uniform carry-select adder which uses four blocks of six full adders, with all but the least significant block performing additions in parallel chains. Figure 4.3 shows the details of a carry-select block with

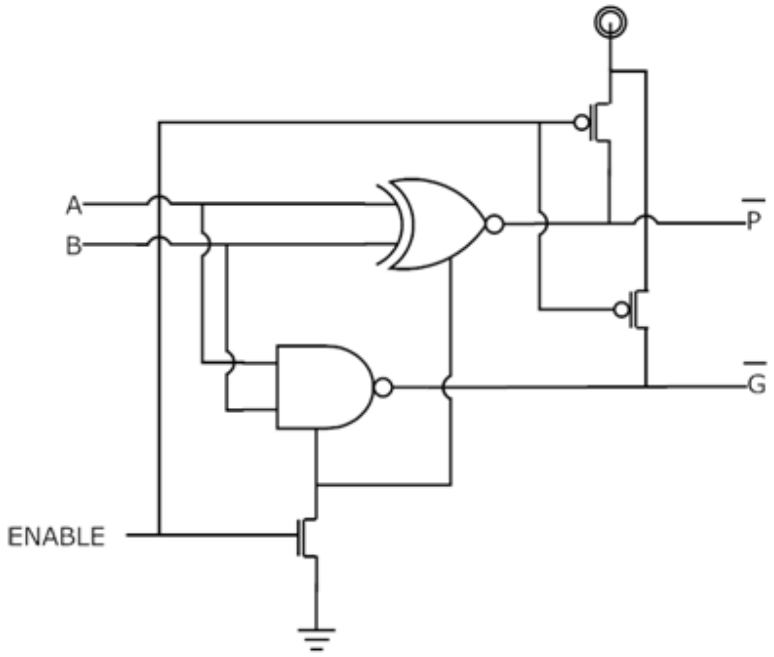


Figure 4.4: Power gating applied to the first stage of a Brent-Kung adder, the carry generation and propagation signal generation stage. Note the use of the NMOS to supply a virtual ground to the logic gates, and the PMOS to tie the output signals to a logical '1,' characteristics of a footer switch. The outputs are sent further down the computation chain of the current bit, as well as to the next stage of the next significant bit, as complementary (inverted) signals.

two layers of full adders gated as a single unit.

Brent-Kung Adder

Last, I modify a 24-bit Brent-Kung adder, one of several parallel adder designs. In contrast to the first two adder designs I explored, which generate a single bit's sum in one functional unit (a full adder), Brent-Kung adders perform addition on a single bit in several stages (Brent and Kung, 1982). Intermediate stages' outputs are used as inputs to later stages of the same bit, as well as later stages of more significant bits. So, in order to freeze the switching activity in the computation of a single bit, it is only necessary to gate the power of the first stage of that specific bit. I used a footer switch to gate this computation in order to tie the outputs high, as they are treated as complementary (inverted) signals by other signal paths. So, the eventual sums generated will be '0' in the disabled bits, which results in the same behavior as my other adder designs. While it is possible to explicitly power gate the subsequent

stages along a bit's computation path, I found that the extra power savings obtained are minimal and do not justify the additional area and speed overheads incurred. The details of these modifications to the first stage can be seen in Figure 4.4 and are the only modifications necessary for applying my technique to this adder.

4.3.2 Modified Multiplier Designs

Integer multipliers are used in many different application domains with similarly varied usage patterns. So, I explored several approaches to modifying a 24x24-bit array multiplier for variable-precision operation. A carry-save array multiplier, abstracted in Figure 4.5, is constructed with a matrix of cells (blue squares) composed of an AND gate, to generate the partial products, and a full adder. The final summation step (dark blue rectangle) of the design is performed with a ripple carry adder for simplicity. This adder is not variable-precision, in order to fully separate the two designs (adder and multiplier), though it would certainly make sense to combine my designs in practice. An nxn multiplier produces $2n$ product bits, but, in the larger context of a floating-point multiplier, only the high n bits (green squares) are used, while the low n bits (red squares) are ignored.

The full adder of each of these cells is gated in a fashion similar to that shown in Figure 4.1, but I also designed versions that have separate gating controls for the signals that propagate downwards and those that propagate to higher bits. First, I tested simply suppressing the low order bits in the operands. Next, I gated the power to just one operand's lower bits, then the lower bits of both operands. Finally, I adapted a truncation multiplier with correction constant and extended the column truncation to provide variable-precision operation with power gating. Each of the accompanying illustrations represents the gating applied to an 8x8 adder operating at 5 bits of precision.

Operand Bit Suppression

Suppressing the data entering the arithmetic units can be done in different ways. In my tests, I assumed bit suppression at the source registers or before; I do not include specialized circuitry for this purpose. My results, then, will simply show the dynamic power saved. Since there is no power gating performed, the leakage power will not be reduced.

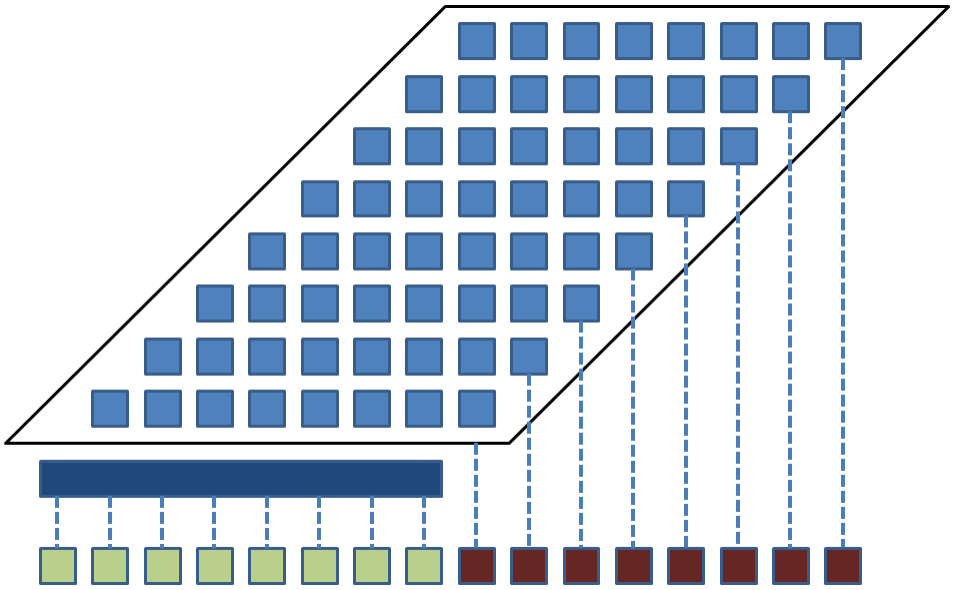


Figure 4.5: An abstracted representation of an 8x8 carry-select array multiplier, showing partial product generation (blue squares), final adder (dark blue rectangle), used product bits (light green squares), and ignored product bits (dark red squares).

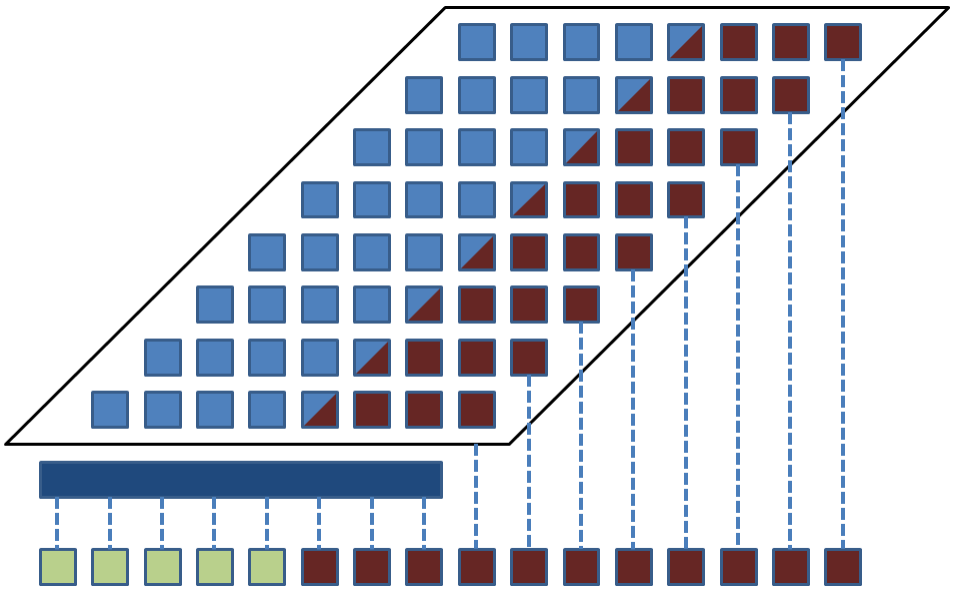


Figure 4.6: When gating only one operand, the multiplicand, diagonal slices of the partial product matrix are disabled. This allows for more precise rounding if required.

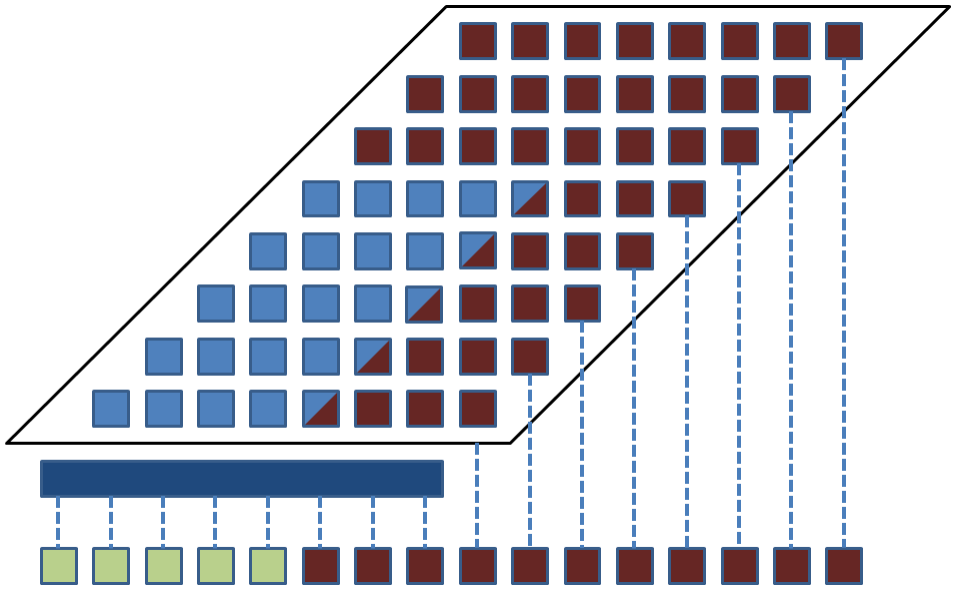


Figure 4.7: When gating both operands, entire rows of the multiplier's partial product matrix are disabled in addition to the diagonal slices of the multiplicand.

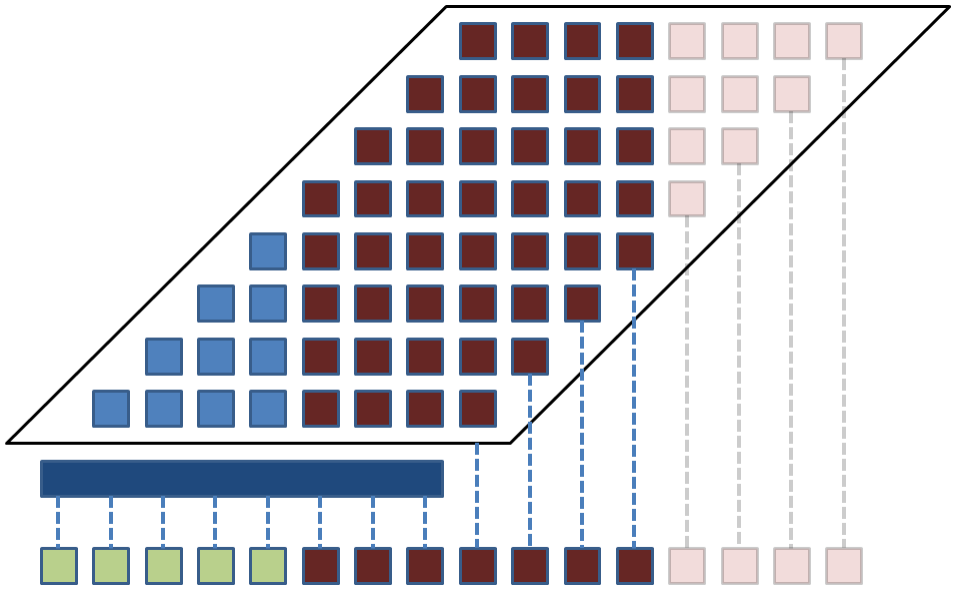


Figure 4.8: Column truncation extends the premise of a truncation multiplier by applying power gating to entire columns at a time. In addition, not every column needs to be implemented in hardware, saving significant circuit area, though this will make full-precision operation impossible.

Single Operand Power Gating

Only varying the precision of one operand (the multiplicand) shows that my design allows for handling operands of different precisions. This yields more precise rounding, if necessary, while still achieving significant power savings. For each bit of reduced precision, another diagonal slice of the multiplication matrix can be gated, as shown in Figure 4.6. Each diagonal slice consists of half of a full adder from the lower bit and half a full adder from the higher bit of the slice, so that the signals that would propagate further left are not affected. This mode will also have the lower bound for energy savings in handling operands of different precisions (one operand at full precision).

Double Operand Power Gating

By gating the low-order bits of both operands, even more circuitry is shut down with each bit of reduced precision. As in single operand power gating, a diagonal slice of the partial product matrix is gated for each bit of the multiplicand. Additionally, an entire row is gated for each reduced bit of the multiplier. This gating scheme is shown in Figure 4.7.

Column Truncation

A truncation multiplier saves area and power by simply not implementing low-order columns of the partial product generation stage. A correction constant which reasonably handles the average case is added to the carry-in stage of the existing circuitry to correct for the incurred error, but errors can still be large when the generated partial product in a column would all be '0' or '1.' I extended the idea of a truncation multiplier (Ercegovac et al., 2000; Walters and Schulte, 2005) by applying power gating to entire columns in order to reduce the operating precision (Figure 4.8). As more columns are gated, the correction constant (supplied in a similar manner to the precision selection) is changed by software to minimize the average error. Since this scheme has an immediate loss of precision, it is not likely a reasonable choice for hardware that may need to operate at full-precision, but I have included it as another example of a design to which fine-grained power gating can be applied.

4.4 Simulation Setup

I used LT Spice IV (Linear Technology, 2010), built on the well-known Spice III simulator (The University of California at Berkeley, 2010), to simulate the netlists generated by Electric (used for rapid prototyping of smaller circuits) for power and timing figures for a 0.13μm TSMC library with a V_{dd} of 1.3V, frequency of 100MHz, and load capacitances of 0.01pF. The Spice models were at the TT corner and simulated at a standard 25C. (A higher temperature and voltage would exacerbate leakage effects.) First, I tested a smaller 8-bit version of each adder exhaustively for correctness, and then I compared the results of adding 200 random operands to a baseline 24-bit ripple carry adder and visually compared the results to waveforms produced by the operations in software. I repeated these steps for the multipliers. In this way, I verified the functionality of my designs. The same set of random 24-bit operands was used for the power usage simulations of each modified unit at each operating precision. The current drain through the supply voltage source was tracked to determine the power consumed and energy used over these operations. Next, a set of worst-case operands was used to find the longest propagation delay of each adder, measured from the 50% level of the input's voltage swing to the 50% level of the slowest output's voltage swing. Leakage power was found by first performing an operation on random 24-bit operands to approximate the average case current draw. Then, power was measured 500ms after the operation to allow for the dynamic current draw to fade away, leaving only quiescent current. I also devised an experiment to time the worst case delay in enabling/disabling all 16 controllable bits at a time. This will be, in effect, the timing penalty incurred for dynamically changing precisions. It may be necessary to slow this down in order to avoid ground bounce, as described above, but it will serve as a worst-case penalty.

4.5 Results

I now present the power savings and area/timing overheads of my designed circuits from simulation. These results are from simulations of pre-layout circuit designs with realistic load capacitances and transistor sizes. While a more detailed, post-layout simulation would also include the effects of wire capacitances, the results presented are strong indicators of the trends of energy savings realizable as arithmetic precision is reduced. Area and timing overheads are difficult to classify as either acceptable or unacceptable (Sathanur et al., 2008), so I compare my overheads with those in other

techniques. Finally, I compare my power savings with other approaches.

4.5.1 Energy and Power Savings

The overall energy consumption for my adder designs as a function of precision is shown in Figure 4.9(a). To demonstrate that these designs help suppress leakage power, which is likely to become increasingly significant as transistor technologies continue to shrink (Roy et al., 2003), Figure 4.9(b) shows the leakage power for each adder circuit as a function of the operating precision. Similar graphs are shown for the results of the modified multiplier power savings in Figures 4.10(a) and 4.10(b). For reference, single full-precision ripple carry, carry-select, and Brent-Kung additions require 3.5, 6.7, and 8.2 pJ, respectively, and a single full-precision multiplication requires 196.1 pJ.

Adders

The desired linear power savings are very apparent and significant in my proposed adder designs. When using a Brent-Kung adder, for example, reducing the precision by just four bits will cause each operation to use roughly 80% of the energy used by full precision operations. In many applications, the precision can often be reduced by more than just four bits without sacrificing fidelity. I will show in Chapter 5 that up to 12 bits can be lost without causing several graphics applications to become unusable. This would give energy savings of close to 50% for additions. Also, though there were energy overheads caused by the circuits becoming slightly slower (see Section 4.5.3), these were overcome after reducing the precision by just 3 bits in the worst case, and only 1 bit in the case of the Brent-Kung adder.

There are some expected characteristics of the energy per operation versus precision trends worth noting. Firstly, the ripple carry adder has an almost perfectly linear slope. This is exactly what one would expect, since precisely one full adder per bit is gated. Second, the carry-select adder has two different regions of savings, due to the structure of its design. The first is seen in precisions 24 through 18, which corresponds to the single layer of full adders being gated in succession. After bit 18, at a precision of 17 and below, the savings are more aggressive as two full adders per bit are gated and consume minimal power.

Leakage power consumption (Figure 4.9(b)) shows analogous trends. Firstly, all the adders show linear savings, as expected. Also, the carry-select adder displays the same dual-slope that was seen in the total power results. Furthermore, while there are

some overheads, due to the added transistors, they are overcome with a reduction in precision by only 4-6 bits.

Multipliers

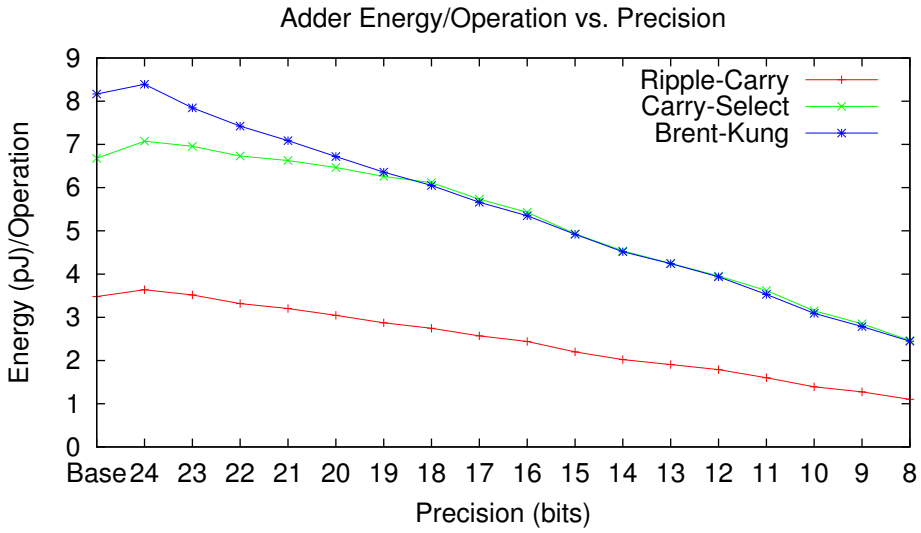
The power savings for the multiplier designs (Figure 4.10) are even more promising than those of the adders, due to the quadratic complexity of the multiplier's hardware.

Just as the adders displayed interesting behavior, the multipliers show trends that warrant remark. The design with the lowest energy savings is that with only one gated operand ("X Gating"), which naturally results in linear energy savings. Simple operand suppression is more useful, but, as previously noted, does not stop leakage current (see Figure 4.10(b)), which will be more of a problem when using a smaller technology. Gating both operands ("XY Gating") performs better than suppression with a similar inverse quadratic decay, expected from the gating pattern. Using this approach, one must only reduce the precision by 5 bits in order to see a 50% decrease in power consumption. Column gating exhibited even more dramatic power savings, which is to be expected, as roughly half of the multiplier was disabled (or not implemented) from the start. However, it must be noted that the precision is not guaranteed to be exactly the number specified, since the correction constant does not change with operands, only with precision. Errors of one to a few low-order bits must be acceptable when using this scheme, which limits its utility somewhat but gives it the greatest power savings.

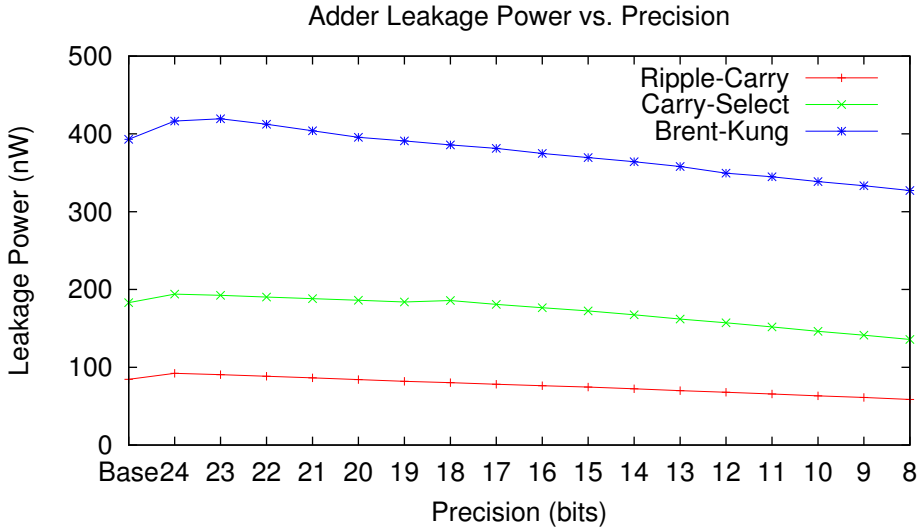
The leakage power versus precision curves, in Figure 4.10(b), resemble those of the full energy per operation versus precision curves. While operand suppression does not reduce leakage power, as was expected, the other designs save significant power and overcome very small power overheads after only one bit of precision reduction. So, the power savings will be immediately realized.

4.5.2 Area Overheads

The extra area incurred by the gating and control circuitry must not overshadow the power savings they enable. Table 4.1 shows the overheads, as extra transistor area, for each adder type, and Table 4.2 shows the same figures for the multiplier designs. I have not included the area penalty for precision control circuitry, as it is dependent on the implementation chosen. Also, any overhead of the control hardware would likely be shared among several units; the amortized impact on a single unit, such as an adder, would likely be acceptably small.

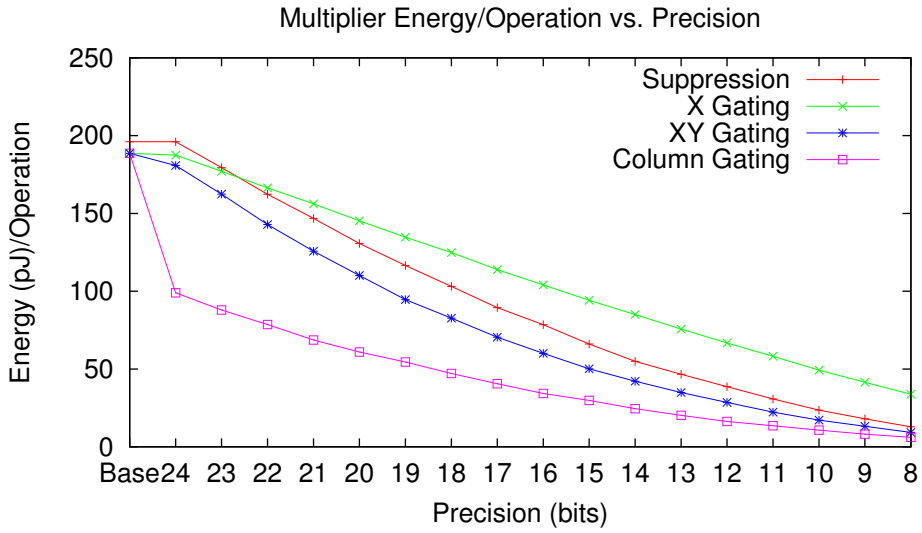


(a) Energy per Operation

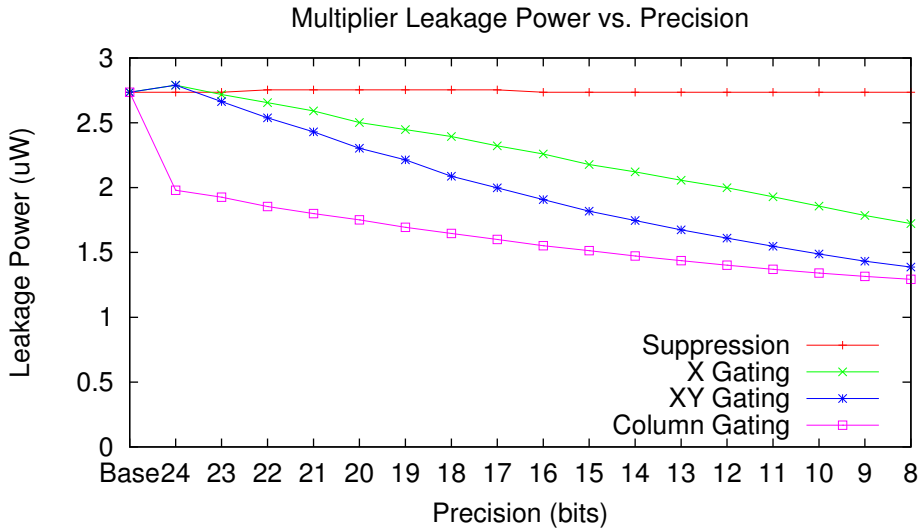


(b) Leakage Power

Figure 4.9: Energy per operation and leakage power versus precision of the different adder designs. The ripple carry adder uses very little energy per operation, while the carry-select and Brent-Kung adders use nearly double this amount. These two, however, are significantly faster. Like the energy per operation, leakage power declines roughly linearly with precision.



(a) Energy per Operation



(b) Leakage Power

Figure 4.10: Energy per operation and leakage power versus precision of the different multiplier designs. Simply gating one operand ("X Gating") leads to a linear savings, while gating both operands ("XY Gating") and taking advantage of the multiplier's quadratic complexity yields more aggressive savings with minimally reduced precision. Suppressing operand data does not reduce leakage power at all, but the other curves show trends similar to those seen in the energy per operation savings.

Table 4.1: Extra area needed for modified adders.

	Transistor area (μm^2)		
Adder type	Unmodified	Modified	Increase (%)
Ripple Carry	4.606	5.383	16.9
Carry-Select	9.165	10.319	12.6
Brent-Kung	13.487	14.735	9.3

Table 4.2: Extra area needed for modified multipliers.

	Transistor area (μm^2)		
Gating type	Unmodified	Modified	Increase (%)
X	128.65	172.78	34
XY	128.65	172.78	34
Column	69.10	80.10	16

Overheads in the on-chip area are not of a degree to prohibit my designs from being used. To control 16 bits of a 24-bit unit, the areas of ripple carry, carry-select, and Brent-Kung adders increase by 16.9, 12.6, and 9.3%, respectively, and the multiplier's area increases by 16 or 34%, depending on configuration. 16 bits is likely at the upper threshold of bits of precision that can be safely lost without adversely affecting the function of an application that normally operates at 24 bits of precision. Choosing a design that controls fewer than 16 bits will use even less extra hardware, both by reducing the number of gating network transistors needed and also by simplifying the control logic. For comparison, simpler signal-gating approaches have incurred overheads of 5-16% (Huang and Ercegovac, 2002) (measured by counting the number of inverters with the simple assumption that each sequential unit has five inverters, while offering only a fixed reduced precision). Only the circuitry to gate at a certain bit (22 in the X dimension and 16 in the Y dimension) was included in the cited work. Overheads would be much higher were their circuits to allow a full range of operating precisions, as mine do.

4.5.3 Timing Overheads

The proposed variable-precision units incur two types of delay penalties. The first is the extra time needed for the input signals to propagate through the resized gates to the output. The second is the time taken to change operating precisions, or the turn-on

Table 4.3: Time overheads of the modified adders.

	Critical path delay (ns)			
Adder type	Unmodified	Modified	Increase (%)	Turn-on time (ns)
Ripple Carry	5.6	5.9	6.9	2.1
Carry-Select	2.4	2.5	6.9	1.4
Brent-Kung	1.066	1.069	0.4	1.069

Table 4.4: Time overheads of the modified multipliers.

	Critical path delay (ns)			
Gating type	Unmodified	Modified	Increase (%)	Turn-on time (ns)
X	6.99	7.26	3.8	7.15
XY	6.99	7.26	3.8	7.15
Column	6.99	7.26	3.8	7.15

time. Table 4.3 lists these figures and compares the propagation delays of the modified and original designs for the new adders, and Table 4.4 reports my findings for the new multiplier designs.

These timing overheads are also acceptable. Firstly, the worst-case turn-on time due to precision changing is a cycle or less for each of the modified designs; allowing that my simulations are pre-layout, this is reasonable. The propagation delay penalty is also quite acceptable, less than 7% at maximum for the adders and less than 4% at maximum for the multipliers. While this overhead is already quite low, in low-power devices, a high clock speed is usually not the primary concern. In fact, the clock may be dynamically slowed to take advantage of lighter workloads. My techniques are orthogonal to DVFS; both can be used on the same circuitry to gain energy savings. As before, my designs are competitive compared with a signal-gated approach that shows delay overheads of 7-11% (Huang and Ercegovac, 2002).

4.5.4 Comparison with Other Techniques

Here, I compare the energy savings of my proposed circuits with the savings of other variable-precision techniques. This is a difficult task, as other reported findings differ in technology sizes and other factors. I offer comparisons of my approach versus both coarse-grain power gating and signal gating.

I first look at one representative coarse-grain power gating technique, a twin-precision

multiplier, which is nearly directly comparable with my results, thanks to the same size process (130nm) and similar driving voltages (my 1.3V versus their 1.2V) (Sjalander et al., 2005). There are several differences between our two approaches: Sjalander et al.'s circuit is based on a tree multiplier, while mine is a simpler array multiplier. Also, their approach allows for only two different precisions to be used, whereas my design offers a continuum of operating precisions. While they do not report all the necessary results, such as power consumption of the multiplier in 16-bit mode, one metric that I can compare is the power consumption of a standard 16-bit multiplier operating on 8 bit operands compared to their twin-precision cutoff multiplier operating on 8-bit operands. The ratio between these two is 3.2, whereas the ratio between my multiplier operating at full and half precisions is 6.8, indicating that I see more savings for the same reduction in precision. However, this comparison is unfair, as I do not implement power gating below 8 bits. So, if I treat 8 bits as '0' and find the ratio between the new full and half precisions (24 and 16, respectively), I arrive at a ratio of 3.4. This is slightly better than the twin-precision multiplier. Lastly, even though my unpipelined multiplier has a delay of 4 to 5 times that of Sjalander et al.'s, depending on configuration and despite my 50% larger bit width, my design is more flexible and has an energy efficiency 1.7 times higher than their design.

I now compare my results against a signal-gated approach by Huang and Ercegovac (Huang and Ercegovac, 2002). In this compared work, a 32-bit multiplier is signal-gated in both the X and Y dimensions, and is the technique on which I have based my "XY" power gating approach. However, they hardwire gating lines at the 22^{nd} bit of one dimension and then 16^{th} bit of the other. I have only reported results for symmetric power gating, though my circuit could be driven with two different precisions. So, to choose a comparison, I first observe that they report results when gating, on average, 40% of each operand. This equates, in my design, to an operating precision of 14.4 bits. So, I will compare their reported results with my results linearly interpolated between 14 and 15 bits. They report energy savings of 67% when using their most low-power design, and I show savings of 76% for my analogous "XY" gating technique. (Column gating would yield better savings, but incurs computational errors not seen in their approach.) As expected, my own "Suppression" technique, which mimics their coarse-grain signal-gating approach, has an energy savings of 69%, which agrees closely with their results.

4.6 Conclusion

I have applied power-gating techniques to several standard integer adders and an array multiplier, converting them to be dynamic, fine-grained variable-precision circuits. My designs show significant savings when reducing the precision of integer adders and multipliers in order to save dynamic and static power consumption. I have shown that the overheads caused by this power gating are modest, and that the precision only needs to be reduced by 2 or 3 bits in order to start seeing energy savings. I will use the energy versus precision characteristics of these circuits in Chapter 5 to build an energy model of the vertex and pixel shader stages of a GPU that can trade precision for energy savings.

There is significant remaining work in the area of variable-precision arithmetic circuits. First, none of my designs are pipelined, which is a common optimization in throughput-oriented devices like GPUs. Second, my designs are only the foundation for an FPU; they will need to be assembled into a variable-precision ALU with floating-point specific hardware to handle exponents, rounding, etc. Lastly, while I have presented several adder designs, I am confident my approach will apply to other adders, as well, including carry-save adders or Kogge-Stone and other parallel adders (Harris, 2003). Likewise, the application of my techniques to different multiplier designs, such as Wallace or Dadda trees, may reveal an even more useful design.

Chapter 5

Energy Savings in Computation

5.1 Motivation

In this chapter, I look at reducing precision in the vertex and pixel shader stages. Shaders currently perform all their operations with 32-bit floating point numbers by default, 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, and vertex positions do not need 24 bits of precision in order to map to the correct pixel, even in a large (1920x1200 pixel) render target. I show that it is possible to reduce the precision of shader operations without incurring noticeable differences in the final image, allowing me to use variable-precision hardware (developed in Chapter 4) to save energy.

The precision of computations in GPUs is often dictated by graphics APIs, such as Microsoft's DirectX (Microsoft Corporation, 2012a). So, the underlying hardware must be capable of performing these full-precision computations. In order to use less precision, the API must be changed or amended for specific cases, or a different graphics API should be used.

Since choosing a single precision for all applications would result in sub-optimal energy savings for some and intolerable errors for others, per-program precision selection must be made available. I explore several approaches to choosing the final operating precision of the hardware for maximizing energy savings. I develop these approaches and present the findings in the context of pixel shaders of several existing applications.

I briefly introduced the efficacy of my techniques in Section 1.5 with Figure 1.4, which showed savings possible in the pixel shader stage. Similar savings are possible in the vertex shader stage, shown here in Figure 5.1.

(a) Full Precision (24 bits)

(b) Reduced Precision (19 bits)

(c) Reduced Precision (16 bits)

Figure 5.1: Figure 5.1(a) is the reference frame produced by full-precision computation (24 bits) throughout the vertex shader of the video game "Doom 3." Figure 5.1(b) shows the result when using 19 bits in the vertex shaders. There are no perceptible differences between the two images, yet the reduced-precision image saved 62% of the energy in the vertex shader stage's arithmetic. Figure 5.1(c) shows the same frame computed with 16 bits of precision, leading to visible errors commonly referred to as "z-fighting," though it did save 76% of the energy.

5.2 Related Research

Sections 2.2 and 2.4 covered standard techniques for saving energy in computation and variable-precision computation, respectively.

5.3 Reduced-Precision Shading

Vertex and pixel shaders have very different characteristics that necessitate individual exploration of the effects of reducing their precisions, discussed below.

5.3.1 Vertex Shaders

Vertex shaders are primarily responsible for the transformation of input vertices in a local 3D object space to output vertices in a common 2D screen space. At this stage, vertices can also be "lit," or have lighting equations evaluated at the vertices' positions, but this operation is more commonly done in the pixel shader for higher-quality results. Transformations, then, are the main operation and the focus of my experiments.

There are two types of errors that can occur when vertices are transformed incorrectly. The first and most commonly expected error is that a vertex will shift its on-screen position in the X and/or Y directions, so that it will end up at a different pixel than the same vertex transformed with full precision operations would. I refer to this type of error as an "XY" error. The second is an error in the depth of the vertex. If a vertex is assigned an incorrect depth, then there may be very subtle changes in the eventual lighting due to an incorrect plane equation. Much more drastic, however, is an error called "z-fighting," which happens when two nearly coplanar faces intersect. The limited precision of the depth buffer can not distinguish between the depths of the two surfaces and does not choose just one surface to be "in front" consistently across the intersecting region. As a result, there may be very distracting spatial and temporal aliasing at this location on the screen as the two surfaces fight for dominance (see Figure 2.3 for an example of this in a full-precision commercial application). This is a common problem that happens even without reducing the precision of operations in the vertex shader that has been researched in the past (Akeley and Su, 2006).

5.3.2 Pixel Shaders

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 immediately apparent. Figure 5.2 demonstrates these two types of errors. Here, I discuss these characteristics so that I can take them into account in my algorithmic and experimental sections.

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

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. I assume that these operations have an error bound of no greater than one unit in the lowest place. With these error characteristics, I am able to apply my 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.

(a) Full Precision

(b) Reduced Precision Texture Fetches

(c) Reduced Precision Color Computations

Figure 5.2: Figure 5.2(a) is the reference frame produced by full-precision computation (24 bits) throughout the pixel shader. Figure 5.2(b) shows an exaggerated result due to reducing the precision of texture coordinates to 8 bits, and Figure 5.2(c) shows similarly exaggerated results of reducing the precision of color computations to 4 bits. Errors of this magnitude are never seen in my test applications when using my techniques to select precisions; these images are shown only to demonstrate the types of errors that are possible.

5.4 Precision Selection

Simply knowing that applications can handle a reduction in the precision of their computations is not enough to enable energy savings; the applications must also know *how far* their precisions can be reduced before errors become intolerable. As seen in Figure 5.3, applications can have very different errors for the same precision. This operating precision can be found in many ways; I propose and discuss several static and dynamic precision selection techniques. I examine these techniques in the context of the pixel shader, as it will have two different precisions (in order to accommodate pre- and post-last texture fetch (LTF) instructions) and is therefore more complicated. Applying any of the proposed approaches to the vertex shader will only require simplification, not

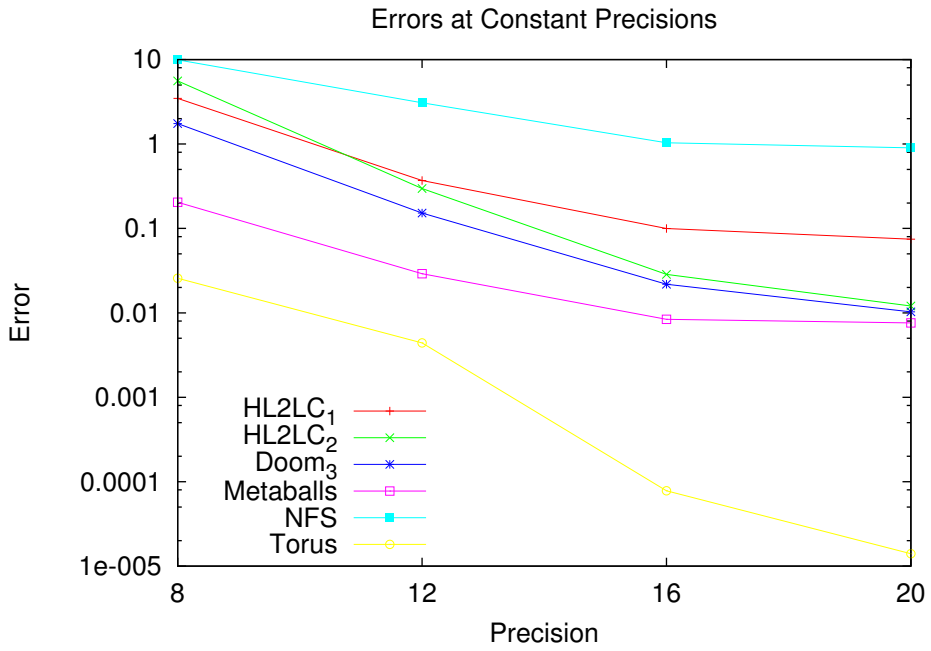


Figure 5.3: Simulating several fragment shaders at various precisions shows that the error is not the same for each shader.

more work.

5.4.1 Static Program Analysis

A static analysis of the shader programs used by an application will determine reduced precisions with guaranteed error bounds. My 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 I 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, I am not able to safely change the precision of instructions that modify texture coordinates. The output of my 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 I 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. I 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.

5.4.2 Dynamic Programmer-Directed Selection

My 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, I 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, existing context switching procedures 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 I do not think this level of fine-tuning will be necessary.

5.4.3 Automatic Closed-Loop Selection

To remove any burden from the application's developer, I have also 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, I describe an efficient method to monitor runtime errors and change the precisions of individual shader programs at the driver level.

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 reduced 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 caused by the precision reduction. There are different ways of implementing this process in hardware, but I will first show that it is a viable method of energy saving. I save a discussion of one possible implementation for Section 5.8. I 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 my simulations, I did not see any multi-frame errors.

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 select a subset of fragments for error determination. Ideally, this subset will be as small as possible, leading to a very small incurred overhead. In my experiments, I have explored varying both the sampling rate and sampling pattern. What I 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 5.7.3.

Precision Control

With an accurate measure of the error caused by the precision reduction of a particular pixel shader, I must now determine how to change the operating precision, if at all. I expect that, due to differences in the responses of texture fetches and regular arithmetic operations to reducing precision, I 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 my dynamic approach. So, I store two precisions for each active shader: one used prior to the 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. I 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. I describe the five I explored below.

"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.

"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 I 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 my explorations, I chose to add ten frames of delay.

"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. A large error encountered during runtime 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, I take its magnitude into account and increase the pre-LTF precision if the error is high. However, I cannot assume that a low error indicates arithmetic imprecision, since low-frequency texture data will also lead to relatively small errors. In this case, I 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.

"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.

"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 the local gradient of the errors with respect to precision. This gradient is then used to predict which phase of the shader is the source of the error. I 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.

5.4.4 Local Shader Errors vs. Final Image Errors

Both the static and automatic approaches give 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.

I 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 I will show in Section 5.7.3.

Only my 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.

5.5 Precision to Energy Model

In this section, I present the energy model I use for estimating the energy spent in the computations necessary to render a frame at a certain precision. I justify the use of the energy characteristics of the circuits developed in Chapter 4 as the models for addition and multiplication. I then use the energies for these operations to model those of more complex operations. The energy spent in a dot product, for example, is the sum of its constituent addition and multiplication operations, and likewise for multiply-add and `MIN/MAX` operations. I also show the model used for reciprocal and reciprocal/square root operations. I assume that these composite operations are performed sequentially.

5.5.1 Addition

Floating-point additions are computationally expensive operations consisting of several steps. First, the operands' exponents must be made equal, which requires shifting a mantissa. Then, mantissas are added. Another normalization step is needed to align the sum's mantissa, followed by a configurable rounding step. I focus on only the addition of mantissas when modeling the floating-point energy. The energy in rounding is significant in a traditional FPU (Jain, 2003; Sheikh and Manohar, 2010), but when doing reduced precision calculations, I assume a simple round toward zero (truncation) scheme which does not need any intricate rounding support. Shifting, too, is significant, but of a lesser magnitude than the addition itself once a simple shifter that need not perform any necessary rounding operations is implemented (Sheikh and Manohar, 2010). Furthermore, the energy in shifting will scale linearly with the bit width of the operands, just like the addition itself. So, the energy spent in a reduced-precision floating point adder will consist of the integer addition (such as my Brent-Kung design) and the shifter energy, both of which will scale with precision. So, I model the energy used by a floating point adder as the energy of an integer adder with the understanding that my estimated energy will be less than the real energy by an amount that will decrease along with the operating precision. I use the results of my modified Brent-Kung adder design as the energy model for addition, E_{ADD}, as a function of precision, p:

$$E_{ADD}(p) = BKEnergy[p] \tag{5.1}$$

5.5.2 Multiplication

Multiplication is modeled as integer multiplication at a given precision. Tong et al. found that 81.2% of the energy consumed in floating point multiplication is spent in the mantissa multiplication or over 98% when the rounding unit is disregarded, which is the case for simple truncation (Tong et al., 2000). Therefore, I focus on the mantissa multiplication, and use the results of a standard array multiplier, modified for variable-precision operation with XY operand gating, as presented in Chapter 4, as my energy model for multiplication:

$$E_{MUL}(p) = ArrayXYEnergy[p] \tag{5.2}$$

5.5.3 Reciprocal/Reciprocal Square Root

Historically, several types of iterative reciprocal (`RCP`) and reciprocal square root (`RSQ`) calculations have been used in hardware. SRT division converges upon the correct result linearly, while Newton-Raphson (and others based on Newton's method) (Chen et al., 2005) and Goldschmidt (and other power series expansions) (Foskett et al., 2006) converge quadratically to the result. In order to make use of my variable-precision designs, I chose to model reciprocal and reciprocal square root computations with narrow bit-width multiplications introduced by Ercegovac et al. (Ercegovac et al., 2000), based on Taylor series expansion. This method consists of a reduction step, evaluation step, and post-processing. Several iterations of the evaluation step are needed, for which some operations require only $\frac{p}{4}$ bits of precision. (For my circuits, low precision is bounded at 8, so this term is constant, though it could be variable if the application called for such low precisions that control of the lower bits were implemented.) When the energies for all stages are summed, the total consumptions are as follows for a reciprocal (5.3) and a reciprocal square root (5.4) operation:

$$E_{RCP}(p) = log_2(p) * \left[5 * E_{MUL}\left(\frac{p}{4}\right) + E_{ADD}\left(\frac{p}{4}\right)\right] + E_{MUL}(p) \tag{5.3}$$

$$E_{RSQ}(p) = log_2(p) * \left[4 * E_{MUL}\left(\frac{p}{4}\right) + E_{ADD}\left(\frac{p}{4}\right) + E_{MUL}(p)\right] + E_{MUL}(p) \tag{5.4}$$

5.5.4 Dot Product

I modeled the energy consumed in 3- and 4-component dot product operations as the sum of the energy in the constituent additions and multiplications:

$$E_{DP3}(p) = 3 * E_{MUL}(p) + 2 * E_{ADD}(p) \tag{5.5}$$

$$E_{DP4}(p) = 4 * E_{MUL}(p) + 3 * E_{ADD}(p) \tag{5.6}$$

5.5.5 Multiply-Add

Like dot products, a multiply-add operation can be modeled as a combination of a multiplication and an addition:

$$E_{MAD}(p) = E_{MUL}(p) + E_{ADD}(p) \tag{5.7}$$

5.5.6 `MIN/MAX`

Comparisons are typically implemented as a subtraction operation followed by checking the sign bit of the result. Therefore, the energy of a `MIN/MAX` operation is simply modeled as an addition:

$$E_{MIN/MAX}(p) = E_{ADD}(p) \tag{5.8}$$

5.5.7 Summary

Most arithmetic operations are built upon the addition and multiplication units. Energy consumed in addition is purely linear with respect to precision, while multiplication's energy trend is quadratic. Thus, I expect my experimental results to have a second-degree polynomial curve, somewhere between purely linear and purely quadratic, depending on the relative frequencies of use of these operations.

5.6 Experimental Setup

My static analysis requires no simulation or user-intervention, I simply require data sets to analyze. To examine my programmer-directed approach, I modified several

simple applications meant to demonstrate a single shader effect. This allows me to demonstrate the ease of use of this approach as well as determine the tolerance to precision reduction of cutting-edge pixel shaders. My automatic approach requires a simulation environment to test my control schemes.

5.6.1 Programmer-Directed Precision Selection

I adapted several demo applications developed by NVIDIA and AMD (see Section 5.6.3) 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 5.3.2. However, modern applications are written in a higher-level language, and I 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, my programmer-directed approach will only make use of one precision per shader. This is still enough to prove my concept, though, and I still see significant savings, even with this limitation (see Section 5.7.3). Lastly, this simplification precludes me from determining the errors introduced by the precisions chosen by a static analysis; I am still able to present energy estimates for my static analysis, however.

5.6.2 Simulator

I chose to use the ATTILA simulator (del Barrio et al., 2006) to test my reduced-precision vertex and pixel shader approaches. Its designers have recently released a version that can use traces of DirectX 9 applications captured by Microsoft's PIX tool (Microsoft, 2011a). This allows me to experiment on recent applications with modern shaders.

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

I also added extra logging functionality. The first type is activated when an operation is executed in a shader: the shader itself calculates the operation's energy based

on its current precision and logs this information for further analysis. The second type is the ability to save transformed vertices so that errors in vertex positions can be analyzed. (Color buffer logging was built-in to ATTILA already.)

5.6.3 Data Sets

The data sets used are different for each experiment: vertex shading, static analysis, dynamic, and programmer-driven precision selection. My static approach uses the data sets from both the other two selection 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. I list these in detail below.

Vertex Shading

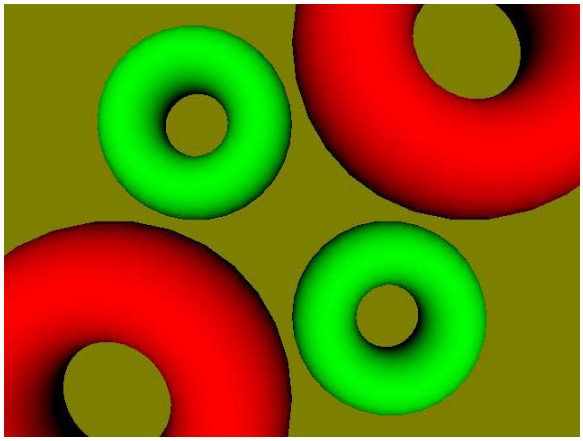
The applications that I simulated and analyzed at the vertex shader stage were "Doom 3," "Prey," "Quake 4," and a simple torus viewer, all traces released by the ATTILA group specifically for use with the simulator and seen in Figure 5.4. Several hundred frames (to create useful videos) of the first two applications were simulated, and several sample frames, used for energy and error analysis, were logged for all four applications. The sample frames for the three games were chosen arbitrarily, and each included a large amount of texture-mapped and lit 3D geometry as well as 2D user interface components. Only a single frame was traced for the simple torus viewer. I simulated these applications at a resolution of 640x480 pixels, which is a higher resolution than all but the newest mobile devices. I also note that relative error is independent of screen size, so the visual errors will not be any more noticeable at higher resolutions; my approach will still apply to a range of devices.

Programmer-Directed Approach

I examine three recent pixel shader effects in the context of a shader editor that an artist might use. These effects are shown in Figure 5.5: depth of field (AMD, 2008), parallax mapping (NVIDIA Corporation, 2010), and screen space ambient occlusion (NVIDIA Corporation, 2010).

(a) Quake 4

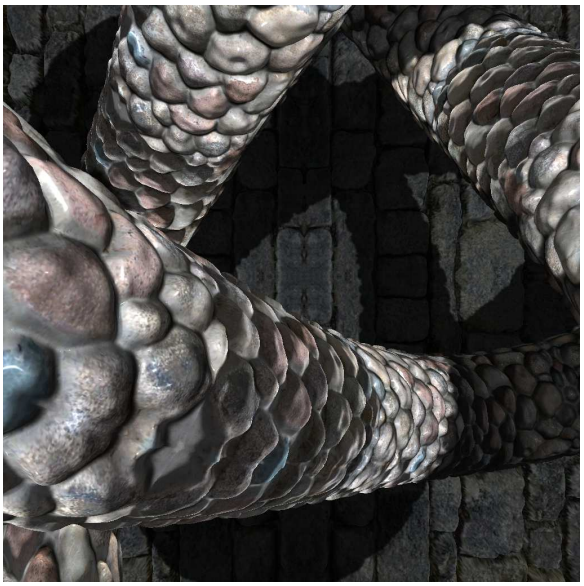
(b) Prey



(c) Torus Viewer

Figure 5.4: Single frames simulated for error/energy purposes ("Doom 3" was shown in Figure 5.1(a)). Three applications are commercial video games, but the torus viewer has much simpler and more compact geometry.

(a) Depth of Field



(b) Parallax Mapping

(c) Screen Space Ambient Occlusion

Figure 5.5: Data sets used to test my 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 5.6: Data sets used to test my closed-loop precision control techniques.

Automatic Approach

The ATTILA designers have released a number of traces for use with their simulator (ATTILA, 2011). I use some of these traces, as well as some that I have captured, to evaluate my techniques. The specific applications I used are, as seen in Figure 5.6, 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).

5.7 Results

5.7.1 Vertex Shaders

As a frame of an application was simulated with ATTILA, information regarding its transformed vertices was logged, and only those vertices that were within the view frustum were analyzed. For those vertices that were on screen at full precision, the error was found by taking the root of the squares of the x and y distances to the full precision position. As in past work (Akeley and Su, 2006), clipped vertices were not taken into account. The errors seen at precisions less than 8 bits were, for the most part, far too high to make the applications usable, so only precisions greater than or equal to 8 bits were simulated and analyzed for these examples. This correlates to the minimum precision allowed for in the circuits I designed in Chapter 4.

Table 5.1: Summary of average error per vertex (in pixels) in a single frame of applications simulated at various precisions. It is seen that each application's average error increases with a decrease in precision, as expected. Also, resolution plays only a minimal role on relative screen space error. That is, doubling the resolution effectively doubles the error of a transformed vertex so the relative error is not affected. In the case of "Prey," though, the relative error actually lessens to a minor degree with an increase in resolution. Thus, increases in display resolutions will not pose any problem to the efficacy of reduced precision transformations.

Application	Doom 3		Quake 4		Prey		Torus
Resolution / Precision	320x240	640x480	320x240	640x480	320x240	640x480	640x480
8	10.54	21.02	7.33	14.62	7.72	14.17	1.169
10	5.34	10.68	1.62	3.20	1.26	2.25	0.355
12	1.32	2.60	0.49	0.93	0.46	0.80	0.089
14	0.33	0.64	0.11	0.21	0.14	0.23	0.020
16	0.10	0.14	0.02	0.04	0.03	0.05	0.006
18	0.02	0.04	0.01	0.01	0.01	0.01	0.002
20	0.01	0.01	~0.00	0.01	~0.00	~0.00	0.001
22	~0.00	~0.00	~0.00	~0.00	~0.00	~0.00	0
24	0	0	0	0	0	0	0

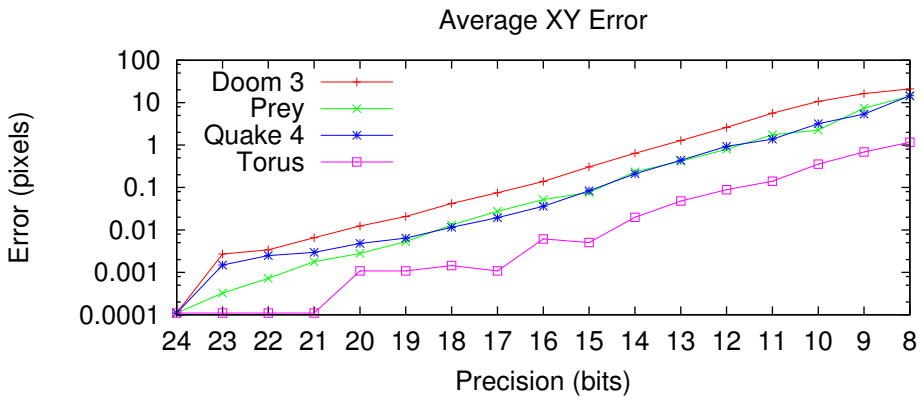


Figure 5.7: Average screen space vertex position error for a single frame of several applications rendered at 640x480 pixels. Note the log scale in the y-axis used to show the minuscule errors at high precisions.

Comparing error versus precision across multiple applications reveals that the pattern is consistent and is not limited to just one application. Table 5.1 lists data gathered from each of the applications. Figure 5.7 shows that each application follows a similar trend: very low errors which increase logarithmically as precisions are reduced. Though the trends in each application are very similar, they are not identical or on the same scale. This suggests that the errors seen in reduced-precision vertex shading is content-dependent and predicting errors without knowing the data itself is impossible.

The low error of the torus model may be due to its relative simplicity. Its geometry is compact and regular, similar to that of other mobile 3D applications, such as a global positioning system (GPS) display or graphical user interface (GUI), while the other applications have vertices both very close as well as very far away in all regions of the screen that are subject to disparate transform matrices. The XY errors are quite small and even less of a factor in application usability than when first considered. A user may not even see a screen space error of several pixels as an artifact, since this error would be shared by all triangles that share that vertex. So, all related geometry would be moved, not just a vertex or triangle here and there, making the error less visible. Partly for this reason, the XY error is not the limiting factor when choosing a reduced precision.

Z Errors

Significant screen space error did manifest itself at very low precisions, but the limiting factor in usability was due to Z errors. Videos made from frames rendered at different

precisions reveal that Z errors are visible long before XY errors in the transformed geometry.

As mentioned above, XY errors will be shared among triangles sharing a particular vertex, so they will not be seen as actual errors until progressive viewpoints cause the error to be manifested differently. Much more apparent are errors in the depth of a vertex. As precision drops, the depths of transformed vertices begin to converge to a more and more limited set of values. This results in z-fighting, seen in the vending machines in Figure 5.1(c), which, since it flickers from frame to frame, is much more immediately apparent to a user than slight vertex displacement.

This trend has an important implication: the precision at which XY errors become unreasonable is much lower than the precision at which depth errors become unreasonable. There are several ways that developers can address this issue. These methods are well known to developers and artists, as z-fighting is a problem even at full precisions. Reduced precisions require more aggressive use of these techniques. If hardware depth testing is disabled in areas prone to z-fighting and the correct draw order is observed, there will be no ambiguity as to which of two coplanar polygons should be drawn. This will eliminate the z-fighting artifact, but could add an extra step to the graphics programmer's pipeline. Also, when an artist designs a model for an application, such as the vending machines, designing them so that there are not two near-coplanar faces can greatly delay the onset of z-fighting artifacts as precision is reduced.

Energy Savings

The energy characteristics of the applications were as generally expected, given my energy model: the energy usage was higher at higher precisions, and decayed quadratically (due to the multiplication unit's savings) towards lower precisions. The energy savings compared to the unmodified circuitry (far-left data point of each curve: "Base") is significant, even for the variable-precision circuitry running at full precision ("24"), due to the ability to perform intermediate computations of `RCP/RSQ` operations with less precision. Full-precision hardware does not have this immediate savings. Furthermore, work involved in transforming vertices is not dependent on screen size, so the results were identical for the same frame of a given application at different sizes. Figure 5.8 shows the graph of simulated power versus precision for the sample frames of each application.

I also present a method of characterizing the tradeoffs in energy and image quality in reduced precision rendering. I allow the designer and user of an application to choose

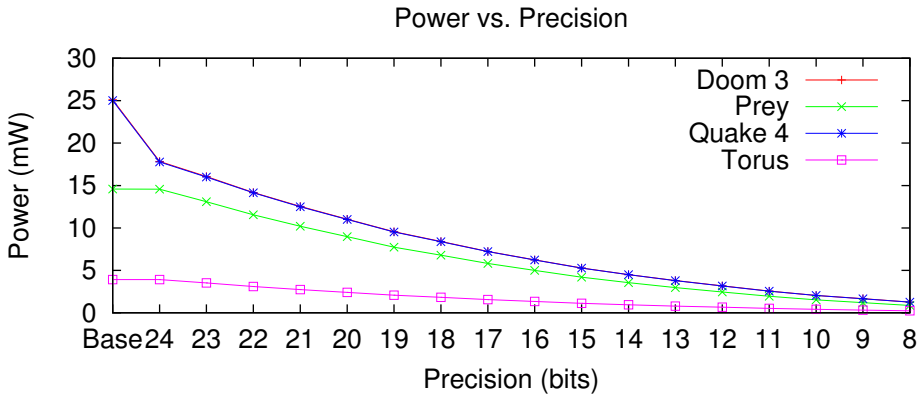


Figure 5.8: Power consumption of vertex shaders as a function of precision, which shows the expected convergence towards zero. "Base" precision is the consumption for the unmodified, full-precision circuitry. Variable-precision hardware allows for reduced-precision intermediate calculations in `RCP`/`RSQ` operations leading to immediate savings in the case of "Doom 3" and "Quake 4."

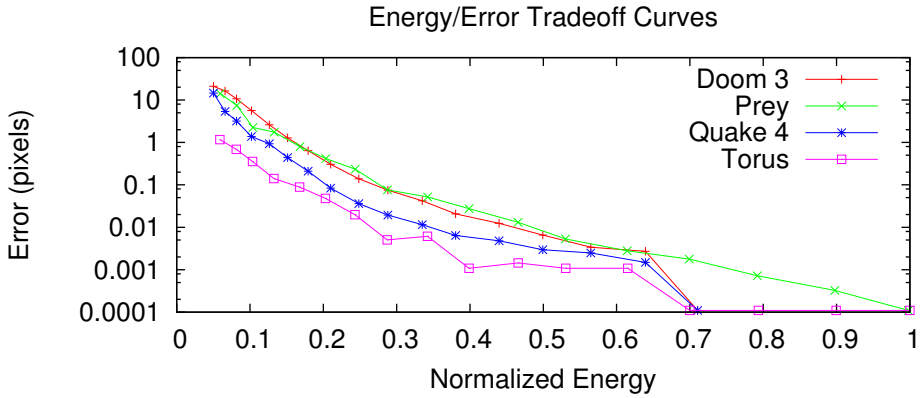


Figure 5.9: Energy-error tradeoff curves for all simulated vertex shaders at 640x480 pixels (note the log scale on the Error axis). At the far left of each data set is the data point for 8 bits of precision, increasing by one to the right, with 24 bits of precision (0 error) represented on the far right of each set with an error of 0.0001 pixels due to the logarithmic scaling. Error is the screen space distance between full- and reduced-precision vertices.

a balance between energy savings and image quality appropriate to their needs. Figure 5.9 shows the energy-error tradeoff curves for all simulated applications. Energy usage is normalized for each application so savings are readily apparent as a percentage of the total energy consumed. I found that XY errors did not cause any perceptible quality degradation when these errors were less than a tenth of a pixel on average. Furthermore, applications did not become unusable until the errors in x and y exceeded, on average,

a pixel. At these errors, energy saved was roughly 75% and 85%, respectively. However, actual savings were not quite this pronounced, since z-fighting limited the utility of the applications before XY errors grew to an unacceptable level.

5.7.2 Pixel Shaders

Now, let us turn from the errors seen in vertex transformations and focus on the results of reducing the precision of pixel shaders. I first show the errors and energy savings that result from my static analysis technique. I then compare these results with my two dynamic approaches. Note: all errors reported are per-component (R,G,B) per pixel, for both average and peak signal-to-noise ratio (PSNR) results.

5.7.3 Precision Selection

Static Analysis

My static approach assumes a full 24 bits of precision 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 5.2 shows the average precision over all instructions for each of my sample applications. The reductions are not high, except in the case of the metaballs application, since it had no instructions before the last texture fetch, and the instructions after the last texture fetch are very simple.

Table 5.2: 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

Dynamic Selection

There are several dimensions in which I can vary my dynamic approach: sampling frequency, sampling pattern, local error threshold, and control method. In this section, I 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. I present the 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, I still see strong trends in the results for each of the data sets.

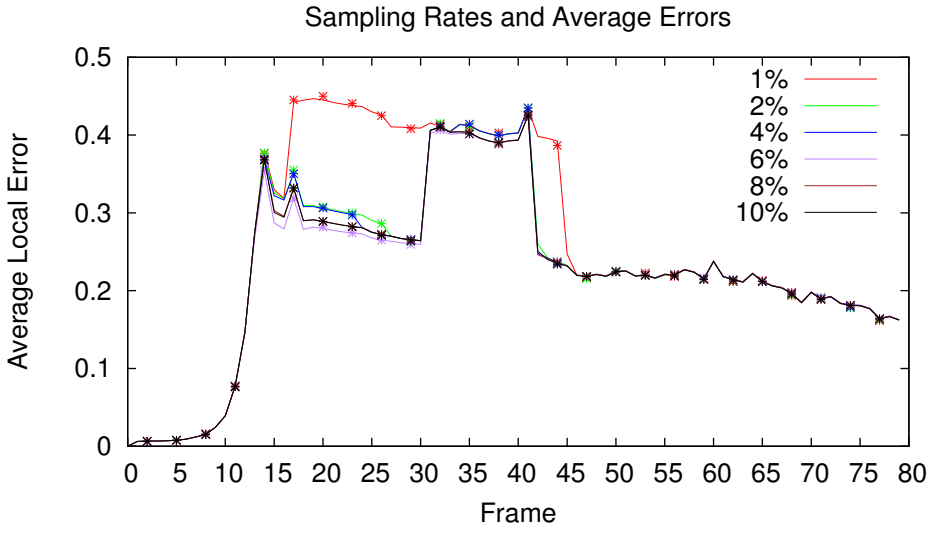


Figure 5.10: Various sampling rates give different approximations of the global error trends in pixel shaders. Sampling more shaded pixels will allow the application to respond to errors more quickly but carry the cost of expending more energy and time. Sampling fewer pixels will lead to a slower response with less overhead. However, regardless of the sampling frequencies I used (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; I 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 5.10 shows how different sampling rates lead to global error

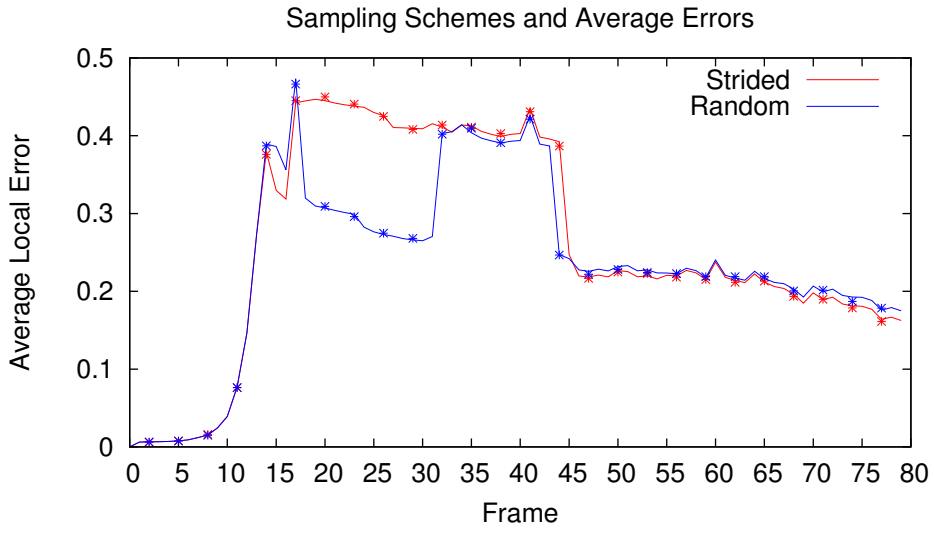


Figure 5.11: Sampling every nth fragment ("strided" sampling) performs nearly as well as random sampling (both at a rate of 1%). Therefore, I use the simpler sampling scheme in my final automatic system.

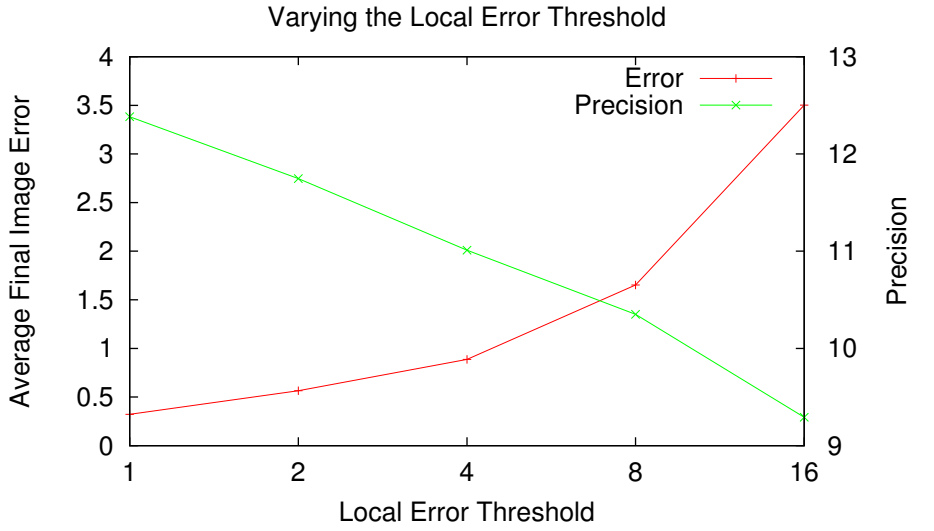


Figure 5.12: Local error thresholds greater than 1 out of 255 do not give significant precision savings to warrant their higher errors.

approximations and indicates that infrequent sampling leads to the same result as more frequent sampling rates. Further, Figure 5.11 shows that simply sampling every nth generated fragment 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 5.12 (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.

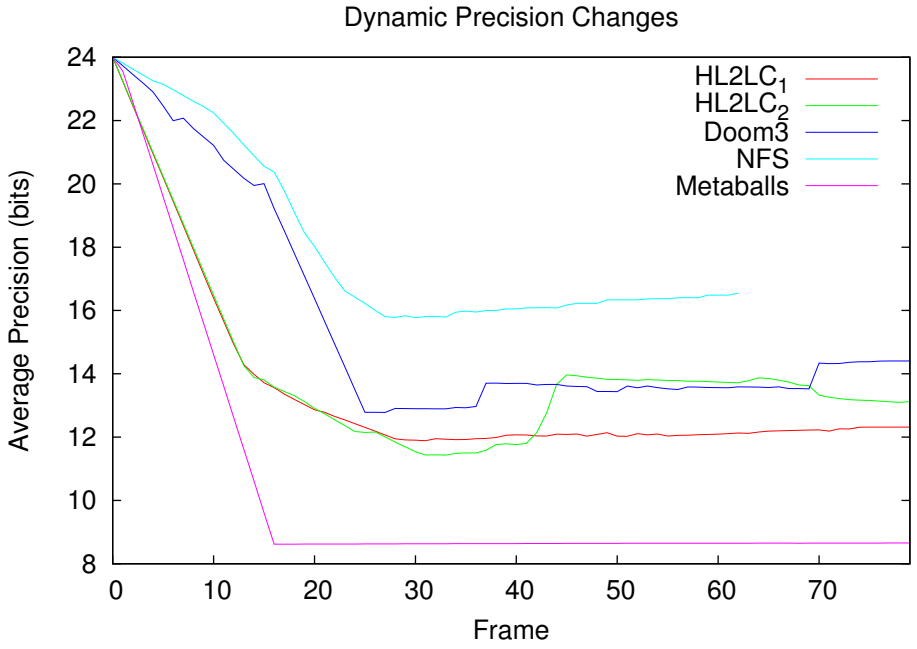


Figure 5.13: As each program progresses, dynamic precision selection will change the average precision used by the program.

Figure 5.13 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 levels 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. Currently, my control algorithms do not consider lowering the precision after it has been raised due to an unacceptable error, so the final decrease in the precision of the $HL2LC_2$ scene is due to a different workload. However, this

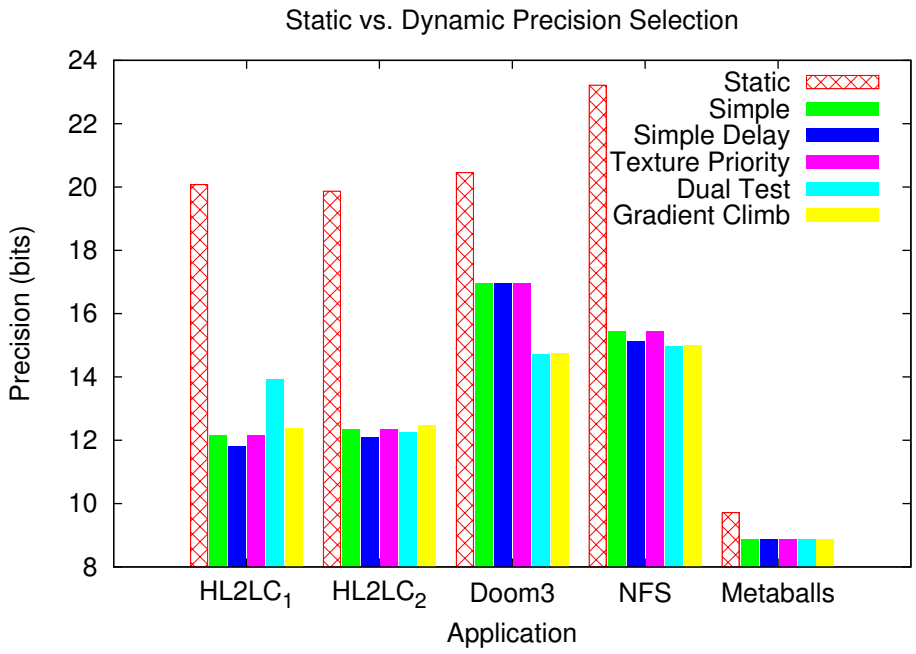


Figure 5.14: A simple precision control scheme performs, on average, as well as a more complex error-tracking scheme, while using significantly less control logic and execution overhead. The "Metaballs" data set's shaders did not have any instructions before the last texture fetch, so the results for all control schemes are identical.

type of extension to my control schemes is natural and straightforward, since a new workload may be tolerant to lower precisions than those seen previously.

I see in Figure 5.14 that the precision reductions are greater with my dynamic approach than with my static approach. This is for two reasons: first, the actual data is used, rather than worst-case operands; and second, I am able to vary the precisions of operations both before and after the last texture fetch. Another important observation about my automatic approach is that the simpler control methods perform very competitively with the more complex methods. As expected, my "simple with delay" approach performs better than my "simple" approach in most cases, but never worse. My "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 my "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 my "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 application.

So, I 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 times. It is this combination that I use to generate my error and energy results in Section 5.7.3.

Overall Errors

In Section 5.4.4, I observed that the measured local errors in each shader do not correspond to the errors seen in the final image. Here, I offer evidence that supports the feasibility of my static and automatic approaches despite this shortcoming. First, I simulated over sixty frames of each data set and was 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 my automatic selection method. Finally, I quantified the errors by measuring their PSNR, presented in Figure 5.15. The steady-state values are all above 40dB. Similarly, I quantify the errors seen in my programmer-directed approach in Table 5.3, which all have similarly high PSNR values.

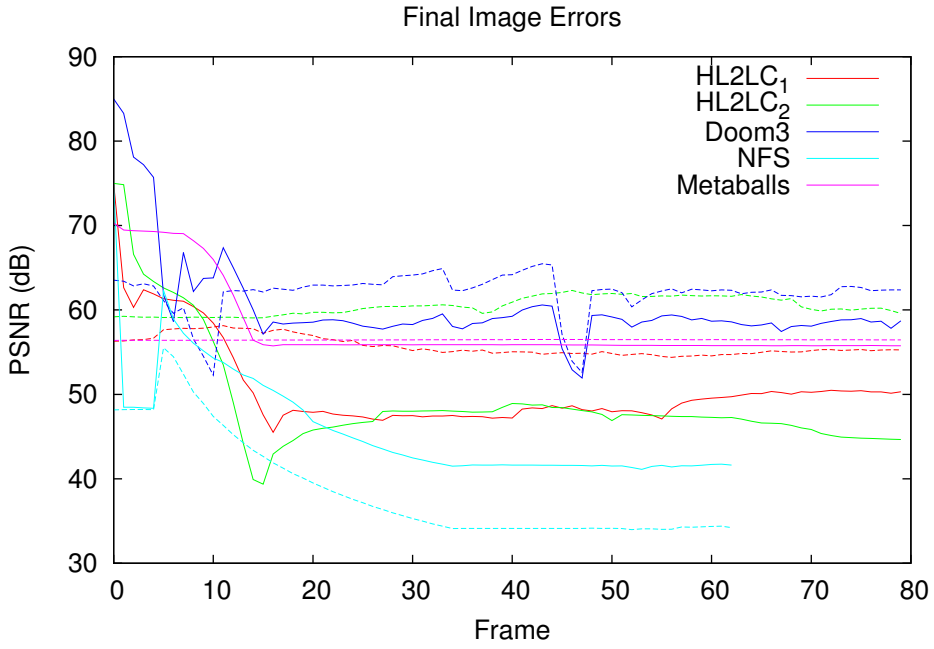


Figure 5.15: With both a 1% strided sampling scheme and a local error threshold of 1 out of 255 for the closed-loop system (solid lines) and my static technique (dashed lines), the errors for each of the data sets are not noticeable. This indicates that a low local error threshold is sufficient to limit final image errors to unnoticeable levels in modern applications.

Energy Savings

I now present the predicted energy savings in the arithmetic of the pixel shader and its contribution to the GPU's savings as a whole. I use the energy saving characteristics

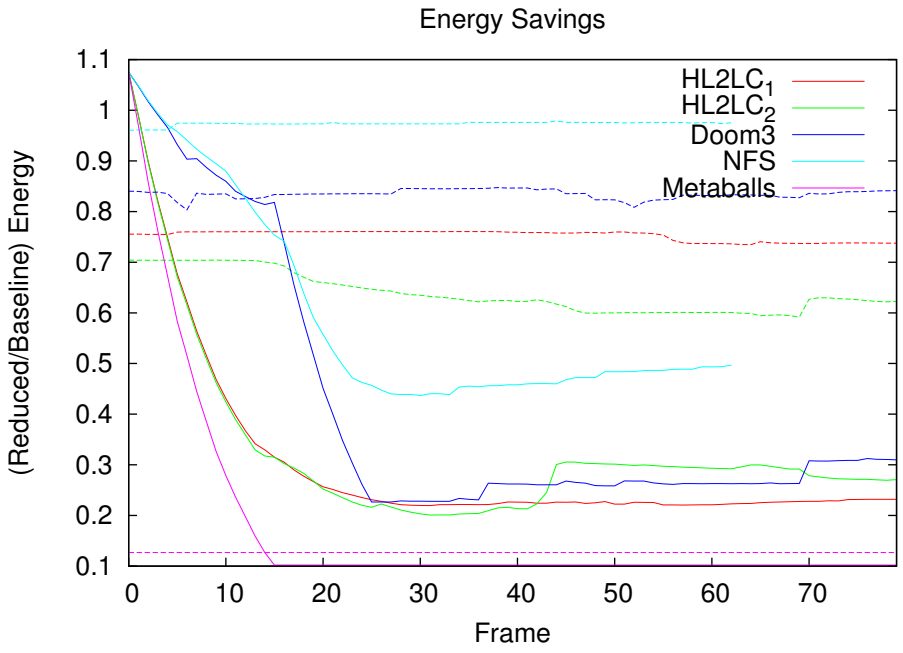


Figure 5.16: Energy savings achievable in fragment shaders with variable-precision hardware. The solid line represents the savings seen by my automatic approach, while the corresponding dashed lines are the savings seen by my static approach. In each case, the dynamic approach saves more energy.

of my variable-precision circuits (Chapter 4) to estimate the energy saved by the pixel shader's arithmetic. For my work, I 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 whole, I use my GPU energy model from Chapter 3 that shows that an average of 33% of the energy in the GPU is spent in the pixel shader's arithmetic.

To be clear, my savings only apply to the shader's arithmetic circuitry; I 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 I have shown that such data is usable in modern applications. However, my results must be seen in this context. Since my baseline energies came from actual measurements of hardware, I can be confident in their relative magnitudes. I was not able to differentiate between the energy of performing arithmetic and reading and writing operands, though, so I have presented my findings at the finest granularity

possible. In the future, the relative costs of arithmetic and register file accesses will lead to a more accurate estimate.

Table 5.3: 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%

In each case, my dynamic approaches outperform a static analysis in terms of energy savings. In Figure 5.16, I see that my dynamic approaches save, after settling on final operating precisions, roughly 71% of the energy in the pixel shader's arithmetic. My static approach, on the other hand, has very limited energy savings, 31% on average, except in the simplest of cases - the Metaballs application. I see similar results for my programmer-directed approach, shown in Table 5.3, 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 5.4: Strengths and weaknesses of precision selection techniques.

Approach	Savings	HW Cost	Developer Effort
Static	Low	Low	Low
Directed	High	Low	Medium
Runtime Automatic	High	High	Low

5.8 Conclusion

I have shown that there is a tradeoff inherent in variable-precision vertex and pixel shaders between the error and energy savings for an application. After developing an energy model for the shader stage of a GPU, I simulated several applications at different precisions. Further, I have explored the relationship between rendering error and energy savings in order to help developers and users of mobile applications choose an operating range to find an acceptable mix of error and energy savings to prolong battery life.

I have also presented several methods for choosing a safe operating precision for modern pixel shaders. I first statically analyzed the shaders used by several graphics applications to determine a safe reduced precision. Then, I developed 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 5.4, I summarized the strengths and weaknesses of each. However, when I also considered that my static and automatic approaches cannot take the error in the final image into account, the programmer-directed approach began to pull ahead. It was 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.

Hardware Feedback Control

One possible hardware implementation for my closed-loop feedback controllers uses 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 my "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.

5.8.1 Future Work

Having shown that the pixel shader stage can save up to 20% and that the vertex shader stage can save up to 10-15% of the GPU's energy by using my proposed variable-precision circuits and control methods, I can now turn to producing even more savings

in other areas of the pipeline. With shader outputs needing fewer bits to represent equivalent fragment and vertex data, there is no need to transmit and store these bits, either on- or off-chip. So, I will look into static and dynamic RAMs and data buses (each a major part of a GPU's data path) to determine the possible energy savings.

There is still work to be done in the area of variable-precision computation, though. Currently, I assume that control overhead and local data storage costs are negligible; a more complete estimate or measure of energy savings would include these quantities.

Chapter 6

Energy Savings in Communication

6.1 Motivation

As I have mentioned before, communication can be more expensive from an energy standpoint than computation of data (Keckler et al., 2011). Saving energy in off-chip communication usually involves compression, while on-chip communication relies on bus encoding, caching, and other techniques. So, I explore compression of full- and reduced-precision data in this chapter, as well as improvements to the compression used on GPUs for full-precision data. Further, I look into an approach to saving energy in reading and writing reduced-precision data to on-chip memories, another significant source of power consumption.

6.2 Related Research

Compression of numerical data has been well-studied. Common to nearly every approach is the encoding of errors of predicted values rather than the values themselves. This has been used in several approaches to compressing floating-point scientific data (Lindstrom and Isenburg, 2006; Ratanaworabhan et al., 2006; Burtscher and Ratanaworabhan, 2009; Hidetoshi and Yokoo, 1994). All the viable methods for compressing geometry, color, and depth buffers in graphics applications known to the authors use this basic technique with specialized schemes for particular applications.

6.2.1 Geometry Buffer Compression

Compressing geometry data can be more complicated than just compressing coherent position values due to associated connectivity and property data. Connectivity data

indicates which vertices are grouped together to form faces in a three-dimensional mesh, and property data determines the colors, normals, binormals, any number of texture coordinates, etc. for these vertices and faces. Deering presented the first major work on geometry compression which approached all of these facets of geometric data in a lossy manner, achieving compression rates of 6-10x (Deering, 1995). More recently, there has been work on the lossless compression of geometry (Isenburg et al., 2005), which led to compression rates of 30-50%.

The details of geometry compression used in commercial hardware are not readily available. Examining patents shows that most work in the area deals heavily with connectivity data (Gruetzmacher, 2010), an approach which is too complicated to map well to the streaming nature of graphics hardware. I found few patents on geometry compression issued directly to makers of commercial GPUs. Wittenbrink and Ordentlich take advantage of data that remains uniform for a series of vertices or faces in order to reduce transmitted data (Wittenbrink and Ordentlich, 2005). Other publicly-available information pertains to geometry and tessellation shaders (Goel and Martin, 2009; Ramey et al., 2008; Dmitriev and Moreton, 2011). (These techniques are two types of "geometry amplification," which generate new vertex data from existing vertex data. Since geometry amplification is orthogonal to numerical compression, they can be used together for even higher bandwidth savings.) Danilak cites pixel compression specifically, but makes no mention of compressing vertex data, despite its storage and transmission on and to multiple GPUs (Danilak, 2009).

6.2.2 Color/Depth Buffer Compression

Historically, color and depth buffers in GPUs have had primarily integer formats. Rasmusson et al. provide a thorough summary of the state-of-the-art in color buffer compression, with an emphasis on integer formats (RGBA8) (Rasmusson et al., 2007). A similar summary exists for depth buffer compression (Hasselgren and Akenine-Möller, 2006). Hardware-accelerated decompression of compressed formats are mostly based upon S3's Texture Compression (S3TC, or DXTC) (Iourcha et al., 1999). Many formats used for different purposes have sprung from this format (Microsoft, 2011b), as well as incremental improvements (Yifei and Dandan, 2010). Most of these texture compression schemes are asymmetric; the data is compressed just once when authored on a single host CPU but decompressed many times on the GPU. So, compression can take an arbitrary amount of time, but decompression must be fast and simple.

Compression of floating-point buffers in hardware is a recent development with the advent of floating-point buffer formats. Commercial techniques have not been made public. Ström et al. present a method for compressing 16-bit floating-point color and depth buffers in a unified manner, with several limitations (Ström et al., 2008). That scheme does not allow negative values and assumes the alpha channel is 1.0f. Later work (Wennersten and Ström, 2009) specifically addressed compression of the alpha channel in color data, but using these two separate compressors for color data introduces complexity. Further work has been performed to allow for lossy compression of color buffers (Rasmusson et al., 2009), which further decouples color and depth buffer compression.

6.3 Improving Compression of Off-Chip Data

A GPU's performance is ultimately limited by many factors. Historically, memory bandwidth and computational power have been limiting factors for different workloads. Recently, power consumption became as important. Though chips have gotten more capable in terms of number of processors and computing power, available memory bandwidth has not increased at the same rate, a trend that is expected to continue (Keckler et al., 2011). Furthermore, there is a limit to the power that can be used to drive a chip; there are practical considerations such as heat, fan noise, and power supply limitations to take into account. In this section, I focus on a method to reduce memory traffic by compressing the data transferred to and from off-chip buffers. While my approach (and the state-of-the-art in lossless buffer compression (Ström et al., 2008)) does not reduce the amount of memory used to store the data, it directly impacts the amount of data that is transferred. As a result, both memory bandwidth requirements *and* energy consumption are reduced, relieving pressure on two major bottlenecks.

Data compression in GPUs has become commonplace in two areas: texture and buffer compression. Texture compression is a particular type of general buffer compression that is performed once off-chip during asset authoring; these compressed textures are then decompressed on-chip many times during execution of the graphics program. So, texture compression is usually asymmetric, and can often be lossy, since a human is in the loop during compression to verify that results are acceptable. As buffer compression is performed on-chip, it is often much simpler than texture compression. While there has been much prior work on buffer compression, most of it is targeted to integer formats. This work instead focuses on the relatively new floating-point buffer formats,

for which very little prior work exists. These formats are important for GPGPU applications and complicated multi-pass graphics techniques.

I develop a general-purpose lossless compression scheme that is able to handle data from any source: color, depth, geometry, and GPGPU buffers. This is a departure from most past techniques, which go to great lengths to exploit knowledge of the buffer's contents. As GPUs become more general-purpose, I believe that such codec specialization hinders generality. My goal is to remain buffer-agnostic so that I can reasonably compress any set of data. I target lossless compression in order to serve general data producers and consumers; I cannot assume that a general application can handle lossy compression without destroying its functionality. As this compressor is expected to be used heavily in rendering color and depth data, I retain random-access read- and write-ability to the same degree as in past work: 8x8 tiles are stored together, and geometry is exposed at a similar granularity.

The state-of-the-art in lossless compression of GPU floating-point buffers targets 16-bit floating-point color and depth data (Ström et al., 2008). I examine this work and its performance on 32-bit floating-point buffers to serve as a comparison to my general-purpose codec architecture. I also suggest two enhancements, applicable to both existing work and my proposed compressor, that lead to higher compression ratios. The specific suggestions I make in this section are as follows:

- a unified codec architecture capable of handling any type of buffer without regard for its contents,
- dynamic selection of compression buckets,
- examination of an alternate encoder for compressing residuals, and
- range reduction for variable-precision data.

6.3.1 Description of the Current State-of-the-Art

Before presenting my novel approaches, I first discuss the design and operation of an existing lossless floating-point color/depth buffer compression/decompression scheme (Ström et al., 2008).

The input to the compressor block is an 8x8 tile of RGB(A) pixels or depth values represented as 16-bit floating-point numbers, and the output is a stream of bits that will be tagged with how the input was compressed: uncompressed, "fast-cleared" (consisting

of a single value), or compressed to 25% or 50% of its original size. Each tile's tag will depend on the compressability of its contents and is stored in a "tile map" that maps tiles to their compression rates to enable random accesses (mandated by graphics APIs). These floating-point numbers are interpreted as integers so any arithmetic performed is exact and not subject to rounding errors, as might be the case with floating-point arithmetic. The 8x8 tile is divided into four 4x4 tiles for further processing. For color data, the red channel is encoded, then the difference between the green and the red, and finally the difference between the blue and the green channels, to take advantage of any correlation between color channels. Tiles that include negative numbers or alpha values less than one are ignored and stored uncompressed.

Each 4x4 tile is handled similarly. Starting at the top-left value, the difference between one value and the next is computed along the top row ($D_{1,1} = V_{1,1} - V_{2,1}, D_{2,1} = V_{2,1} - V_{3,1}, etc.$) and left column ($D_{1,2} = V_{1,2} - V_{2,2}, D_{2,2} = V_{2,2} - V_{3,2}, etc.$), implicitly predicting each value from the preceding one. These integer differences are encoded (as described in the next paragraph) with the hope that the difference between values will be small and therefore more compactly represented. A more complicated prediction scheme intended to minimize these differences is used for the remaining 3x3 values, choosing either the value above, to the left, or an average of these two as the predicted value. This is intended to handle cases where there is a discontinuity within the tile that would lead to poorly predicted values. A guide bit for values along this discontinuity indicates how the values are to be predicted (and reconstructed). Further, there may be a "restart value" (requiring a 4-bit position and 15-bit value) to indicate a more fitting starting point for values on the other side of the discontinuity. To better handle discontinuities, the entire tile may be rotated 90 degrees counter-clockwise (indicated with a single bit). The beginning of the encoded data stream, then, is a single restart bit, optional restart position and value, a rotate bit, and the top-left value.

Difference values are encoded with a Golomb-Rice encoder. Since the input values were all positive, the differences can be represented with 16 bits, which are then mapped to the positive domain with a simple reversible transformation which ensures that numbers with similar magnitudes will appear sequentially. Negative values are multiplied by $2n - 1$ while positive values are multiplied by $2n$. Thus, values of -1 and +1 will both have small representations. To encode a value, it is divided by some power of two, 2^k, to yield a quotient and a remainder. Unary encoding is used to encode the quotient, q: a series of q ones with a terminal 0. The k bits of the remainder are stored in binary. The best k value for each 2x2 block in a 4x4 sub-tile is found through

exhaustive search and then shared among the four values, stored before the quotient and remainder data. The rest of the encoded stream is as follows: k values (16 bits), a variable number of guide bits, and Golomb-Rice bits (quotient and remainder data).

6.3.2 Proposed General-Purpose Compressor Design

My general-purpose compressor is based on the approach described in Section 6.3.1. At a high level, my design is meant to handle any type of data: not only color and depth, as in past work, but also geometry or general-purpose data. As GPGPU applications become more prevalent, I believe it would be of great benefit to the simplification of hardware if a single compression/decompression block could serve all clients with good compression rates. To support this, I must accept negative numbers to my compressor, be able to handle data of various layouts (not just square tiles), and still allow for random access to the data. I describe the modifications necessary for each of these capabilities in following sections.

As shown below, due to the general nature of my design, I do not need the guide bits, restart bits (flag/value/position), or rotate bit. This simplifies both the hardware necessary for my design and the encoding/decoding effort. Further, I do not need to store these bits, which saves space; though, of course, I cannot use them for their intended purpose if they would be helpful.

Handling negative values

Handling negative values is not straightforward. When input floating-point values are negative, they will have information in the sign bit. When operating under the stipulation that all input values to be compressed are positive, this sign bit is always zero, and so the difference between two floating-point numbers interpreted as integers will never overflow. With varying sign bits, this is not always the case, and overflow can occur when subtracting or re-mapping the difference values, a necessary step in many encoding schemes, including Golomb-Rice encoding. To avoid overflow, I specify that subtraction and remapping take place immediately prior to encoding. Integer hardware that can handle numbers one bit larger than the values themselves can be used to keep track of this overflow without needing to handle storage and transmission of a non-power-of-two number of bits, which would be necessary if further processing is to be performed on residual values. Since I perform direct encoding and do not need to compute guide bits and restart values, I can make this simplification that is unavailable

to past work. While the discontinuity around 0.0f when floats are interpreted as integers will lead to disproportionately large residuals and hurt compression rates, functionality is not affected.

Arbitrary numbers of attributes

Rather than hard-wiring my compressor to deal with some fixed tile size, I must allow it to handle buffers of any layout. To this end, I let the compressor work on vectors of data instead of tiles. To construct a vector of values, the stride of the data is necessary. For geometry data, this stride is readily available from the vertex buffer descriptor. An example will make this approach clear. Consider a vertex attribute buffer containing (x,y,z) positions, (xn, yn, zn) normals, and (u,v) texture coordinates for each of N vertices. The stride of this data layout is (3+3+2) values * 4B = 32B per vertex. Using this stride in my scheme, then, the x value of the positions would first be compressed. The first value, *x0*, is stored uncompressed, followed by the *encoded* difference between the first and second values, *x1-x0*, and so on. Incrementing the offset by 4B and repeating the stride allows me to encode the remaining attribute values.

This approach has two major benefits. First, it does not assume any particular shape of data; it is simply repeated for each vector. Second, it exploits much of the available coherence in the data. It is much more likely that neighboring x values will be related, rather than the x and y values of a particular vertex. In the case of color data (r, g, b, a), a tile is stored one-dimensionally in memory, not in two dimensions. This means that it will be input to the compressor as a series of pixels, and each vector of data will be comprised of a single color component if the correct stride is observed. This addresses an issue noted in past work (Wennersten and Ström, 2009)—existing compression formats assume coherence between color channels where there may not be any. However, if there *is* coherence between color channels, I will not exploit it without further effort, as past work (Ström et al., 2008) has done.

Enabling random access

Random access of input data is key in many common uses of graphics hardware. For color and depth data, hardware access is at the tile level. Geometry has no such predefined subdivision finer than an addressable buffer. The user, though, is free to start rendering at any point in the buffer or update only certain parts of it between rendering commands, such as when animating a subset of particles that are still in

a "live" state in a larger particle system. To allow this, I simply compress a subset of the buffer at a time. For instance, in the above example with N vertices, I will not compress all N vertices at once. Instead, the first C vertices will be compressed, then the second C vertices, and so on. There is a tradeoff inherent in the size of the subset. The fewer vertices I define as a subset, the finer the addressing granularity is. Also, as a Golomb-Rice encoder encodes difference values, it shares some parameters for the whole compressed buffer. The smaller C is, the better values shared among the subset will fit the data, leading to a smaller representation. However, as C shrinks, these values will be replicated more often and may not be different enough to warrant a unique value. I experimented with different subset sizes and found that a size of 64 gives good results for all my data sets.

If random access were not a requirement, such as in streaming general-purpose computations, several simplifications would become available. Most importantly, compressed blocks would not be constrained by a quantized compression rate; a continuum of compression rates could be supported rather than just a subset. With this relaxation, the stored data could be compacted, saving space in memory as well as bandwidth. Therefore, there would also be no need for the buffer map, since reading and writing data would happen sequentially from a start address. If a buffer can be declared as read-only, such as most geometry buffers, I could also compact the stored data, since there would be no chance of having to write more data than could fit into the alloted space.

6.3.3 Proposed Techniques

Here, I discuss two proposed techniques for increasing the efficiency of any given hardware compression scheme: (i) an algorithm for the dynamic selection of compression buckets for buffer maps and (ii) an encoder that can be more efficient for encoding residual values than the unary encoding used in standard Golomb-Rice compression. These novel techniques target two different areas that play a major role in determining the amount of data transmitted: the assignment of an overall compression ratio (which I call a compression "bucket") and the encoding of residual values. These techniques can be used independently if one or the other fits a particular compression scheme.

Bucket Selection

In past work, buckets have been chosen by the hardware designer and set statically in the hardware itself. Typical bucket values are "Fast Clear (FC), 25%, 50%, and Uncompressed" (Ström et al., 2008). However, this poses two problems. First, the buckets that best capture one buffer may not serve another buffer as well. Second, in my work with variable-precision data, the buckets that best fit a particular buffer at a high precision may limit the savings possible when the precision is reduced. There are two simple approaches one might take. First, seeing that these buckets are too optimistic for some buffers (see Section 6.3.6), one could choose higher buckets, such as "FC/50%/75%." However, this will limit the compression rates achievable by highly compressible data sets. Another straightforward approach is to increase the number of buckets, say from four to eight. However, this would increase the storage needed by the buffer map. (In past work, this was called the "tile map;" I feel that "buffer map" better describes its use in a general compressor, which may or may not be tile-based.) The buffer map's presence in an on-chip cache is very important, as every access to memory depends on its contents. So, increasing its size is not a viable option.

Listing 6.1: Dynamic Bucket Selection

```
Bucket assignBucket(uint inSize, uint outSize) {
  Bucket smallest = chooseSmallest(inSize, outSize);
  if (smallest in chosenBuckets)
    return smallest;
  if (bucketCount < maxBuckets)
    chosenBuckets[bucketCount++] = smallest;
  else
    smallest = nextLargest(chosenBuckets, smallest);
  return smallest;
}
```

I seek to assign buckets dynamically for every unique buffer, be it a render buffer, depth map, final frame buffer, or input geometry buffer. Each buffer will store its currently selected buckets in its descriptor. I constrain my algorithm in four ways. First, I do not allow more than four buckets per buffer, which keeps the size of the buffer map the same as in past work—2 bits per buffer. Second, by necessity, the "uncompressed" bucket is non-negotiable; I must assume that there will be input data that will not be able to be compressed at all. This leaves three available buckets that can be chosen dynamically. Third, I am not allowed a pre-pass to examine the

buffer; it must be compressed on-the-fly. Lastly, I allow a bucket granularity of eighths. While having even more bucket options with a smaller size *may* perform better, it is unreasonable to expect finer granularities when reading from memory. My $1/8^{th}$ granularity buckets for 32-bit data are the same size as the $1/4^{th}$ buckets used for 16-bit data in past research.

My dynamic bucket selection algorithm is a three step process. First, the smallest bucket (again, in one eighth increments) that will fit the output data is chosen. If this bucket is already in use, then the algorithm is complete and that bucket is used in the buffer map. If this bucket it not in use and there is still an "open" bucket, this open bucket is set to be the chosen bucket and is written to the buffer map. In the worst case, the smallest bucket cannot be used, and the next largest bucket must be used from the already-chosen list. This process is illustrated in Listing 6.1. The extra data structure, the "bucket list," will take up 9 bits—3 bits (8 choices) for each of 3 buckets (since the first bucket is always "uncompressed")—and will be stored alongside the buffer map for the data since they are accessed in tandem. As a block is decompressed, the buffer map indicates which entry (0–3) in the bucket list contains the compression rate, which then dictates how much data to request from memory.

Fibonacci Encoder

I saw larger residuals (compared to the size of the input value) than seen in past work when compressing color and depth data with my general compressor (Ström et al., 2008; Rasmusson et al., 2007). There are three reasons for this: negative values, frequent unclamped values, and a larger mantissa to total representation ratio. Allowing negative values can cause differing sign bits, leading to differences with maximum magnitudes in common cases. Further, since general data is not expected to be commonly in the range of [0.0..1.0], like color, normal, and many depth formats, differing exponents for neighboring data values will lead to larger residuals, even in same-sign values. Lastly, 32-bit floating-point numbers will be left with relatively more information after subtraction than 16-bit floating-point numbers, as used in past work. This is due to the ratio of mantissa bits to overall bits in the two representations—23:32 (~3:4) for 32-bit data, and 10:16 (5:8) for 16-bit data. Taking all this into account, I explored using an encoder other than unary encoding to store quotient values generated by a standard Golomb-Rice encoder.

I used a Fibonacci encoder (Fraenkel and Klein, 1996) to alleviate pressure caused by larger residual values. Fibonacci code words share several important properties with

unary code words; they satisfy the prefix condition, are instantaneous, and map smaller values to smaller code words. This last point is important when the expected values cluster around zero, as they do in numerical data. The main difference is that the size of unary code words grows linearly with value encoded, whereas Fibonacci code words grow sub-linearly. This allows for larger values to be encoded before the code word reaches a prohibitive length. While unary codes are shorter for small values (<4), Fibonacci codes will be more efficient for these expected larger residual values.

The key insight to the Fibonacci encoding is that any positive integer can be represented as a sum of non-consecutive Fibonacci numbers in the series '(1,)1,2,3,5...'. To encode a value v, find the largest Fibonacci number less than or equal to v. Decrease v by this number, and seed the encoded value with a '1.' While there are more Fibonacci numbers, repeat the following steps: 1. Move to the next lesser Fibonacci number. 2. If this number is less than v, decrease v by this number and prepend a '1' to the encoded value; otherwise, prepend a '0.' At the end of this process, append a final '1,' causing '11' to finish the stream. Since this encoding takes a greedy approach to computing the sum, no other consecutive '1's will be found in the stream. For instance, the integer value '12' is encoded as shown in Table 1. This code requires only 6 bits, while encoding '12' with a unary code would require 13 bits ('1111111111110').

Table 6.1: Encoding the value '12' as a Fibonacci code gives '101011.'

1	2	3	5	8	13	...	'1'
1	0	1	0	1	_	_	1

Hardware Implementation

My proposals are able to be implemented in hardware without any major changes in the architecture. Fibonacci coding, while nontrivial, is not particularly difficult. The necessary Fibonacci numbers could be stored in a look-up table; to encode a maximum difference of 33 bits, less than 50 Fibonacci numbers are needed. To encode a value, the algorithm simply marches backwards through the numbers, logging whether or not a number is in the sum and subtracting any included Fibonacci numbers from the encoded value.

Dynamic bucket selection is able also to be implemented in hardware. Each buffer already carries with it a descriptor of some type. For color buffers, this holds component and data formats; geometry data has a data layout header. This is where the

chosen buckets for this buffer could be located at a cost of 12 extra bits (to encode 10 possible states for three different buckets). Decompression of data is only slightly more complicated with the addition of the bucket list. Since the descriptor information must be available to read and write the correct data in any case, the chosen buckets will also be available before consulting the buffer map. The combination of the chosen buckets and the buffer map will allow the memory controller to request only as many lines as necessary. Compressing data requires only a slight modification to Ström et al.'s approach. Rather than just assigning one of three buckets based on the input and output size, the buffer is first assigned to one of eight preliminary buckets; this is no more complicated than before. After this, the chosen buckets for the buffer are consulted and possibly updated. Since, as when reading, the chosen buckets and buffer map are available in local memory, accessing these lists is fast and cheap.

6.3.4 Compressing Reduced-Precision Data

In Chapter 5, I performed variable-precision arithmetic on the GPU by only using the most significant p bits of the mantissa in computations. Since this leaves 23-p bits unused, it is unnecessary to move these bits off-chip. Taking advantage of this, I modify the range of the values input to the compressor. As values are input to the compressor, a standard step is to reinterpret the floating-point values as integers. By dropping bits on the right that have been ignored by variable-precision arithmetic, I can lessen the magnitude of the values, and therefore the magnitude of the difference between them, which determines the size of the encoded stream.

The precision of the data undergoing compression or decompression is constant per buffer and assumed to be stored in the data descriptor. For color buffers, this header commonly contains information such as width, height, number of components, and compression used (in hardware which supports multiple compression schemes). For input geometry, this would be the vertex buffer declaration, which holds information about each stream, such as data type, number of components, and, in some cases, intended use. By storing the buffer's precision with this standard collection of data, I do not have to store it with the buffer data itself, avoiding overheads. It would be possible to store the precision of each compressed buffer and opportunistically perform variable-precision compression for data that happens to have a number of trailing zeros in the same way that standard compression takes advantage of leading zeros. However, I found that trailing zeros in full-precision data are not common enough to overcome

the overheads of storing precision with the buffer data.

This approach is different from past techniques in several ways. First, though Rasmusson et al. describe a similar system of quantization, theirs is intended to perform lossy compression (Rasmusson et al., 2009). Therefore, they also monitor errors at runtime and scale back quantization when it could lead to larger errors. My approach is lossless, though it operates on data that has been quantized. The difference is that the quantization step for variable-precision data has been performed by the artist or programmer (see Chapter 5) and has been judged acceptable; I do not risk incurring more errors. Further, Rasmusson et al. performed quantization on the residuals of the predicted values (Rasmusson et al., 2009). Over the span of a tile, this leads to errors as values are reconstructed erroneously from previous values. Since I quantize the input values themselves, errors do not compound; the input value can be reconstructed exactly.

6.3.5 Experimental Setup

I seek to report compression rates seen by real applications during use. Therefore, I do not look merely at the final color or depth data for a frame; rather, I treat each intermediate draw stage when possible for a realistic estimate of saved bandwidth. (A more complete simulator would perform compression on a cache eviction, leading to many more partially-covered tiles. However, I do not have access to this level of detail in the rendering pipeline. As a result, the presented results will be optimistic.) To do this, I log intermediate buffers from running applications as vectors of floating-point values. I can replay these buffers in a custom simulator and infer which tiles were changed as the result of any given step. These tiles are then re-compressed with the new data, and the bits spent in transferring old and new tile data is counted towards a running total for the experiment. Similarly, I examine the actual vertex data fetched during execution and count only it. During simulation of a buffer, dynamic bucket selection is coherent; that is, the buckets chosen during the first pass on the buffer do not change for successive passes.

I do not explicitly model a cache hierarchy. While such a model would change my absolute results for amount of data transferred, it will not impact my compression rates. Further, any model I devised would be an approximation and lacking the context of a full graphics system; such a full model is out of the scope of this work. Thus, my work should be viewed as presenting novel compression techniques for general data with

resulting compression rates, not as a prediction of ground-truth bandwidth savings.

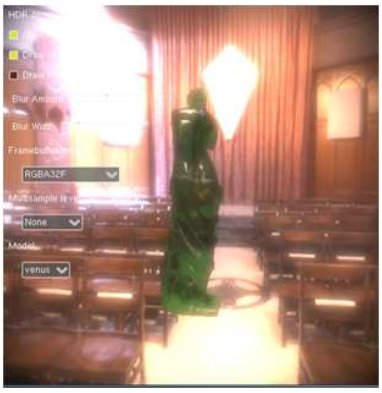
Several implementation details must be noted for adapting the baseline technique (Ström et al., 2008) to handle 32-bit variable-precision data. First, the constant error threshold used to select guide bits needs to be much larger; I found 2^{28} to work well for all data sets. Second, to support variable-precision values, this error threshold must be updated when the range of input values changes. This is as simple as shifting this error value at the same time that the values themselves are shifted, and by the same amount.

Data Sets

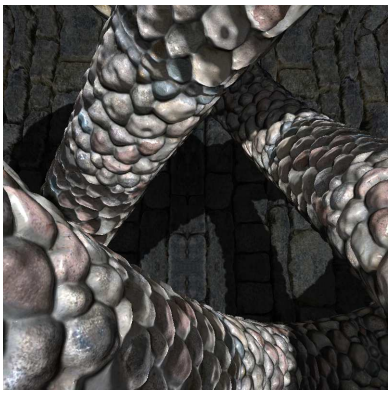
To test my algorithms, I have used different buffers from several applications seen in Figures 6.1–6.3. For my test color buffers, I used scenes from a high dynamic range (HDR) rendering demo, a depth of field demo (AMD, 2008), a smoke simulation visualization, a parallax mapping demo (NVIDIA Corporation, 2010), and a demo that generates terrain data on the GPU to use for rendering ("Map") (Persson, 2006). For testing depth buffer compression, I used depth maps generated and manipulated by an application which demonstrates different shadow mapping techniques (Lauritzen, 2007), as well as from the video games "Need for Speed: Undercover" ("NFS:U") (EA Black Box, 2008) and "Crysis" (Crytek, 2007). I also used input geometry from these two games to test my compression techniques on vertex positions and attributes, as well as geometry from "Crysis: Warhead" (Crytek Budapest, 2008) and several scenes of "Half-Life 2: Lost Coast" (Valve, 2005).

It is worth noting that the "Depth of Field" and "Map" examples encode extra information in the alpha channel of the RGBA render target. "Depth of Field" encodes the depth of the scene to use in further processing, while "Map" uses the RGB channels to encode normals of procedurally-generated terrain and the alpha channel to encode this terrain's height. These data sets are not compressible by past work without modification. However, my unified system allows for compression of these nonstandard uses. In the interest of testing more data sets, I will disregard the fourth channel in these two data sets when presenting results of the existing compressor modified with my proposed techniques (dynamic bucket selection, Fibonacci encoding). When comparing my unified compressor with the past approach, however, I do include the fourth channel to show the benefit of a general compression scheme.

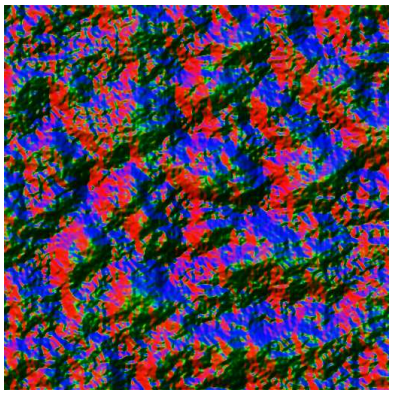
(a) DoF



(b) HDR



(c) Parallax



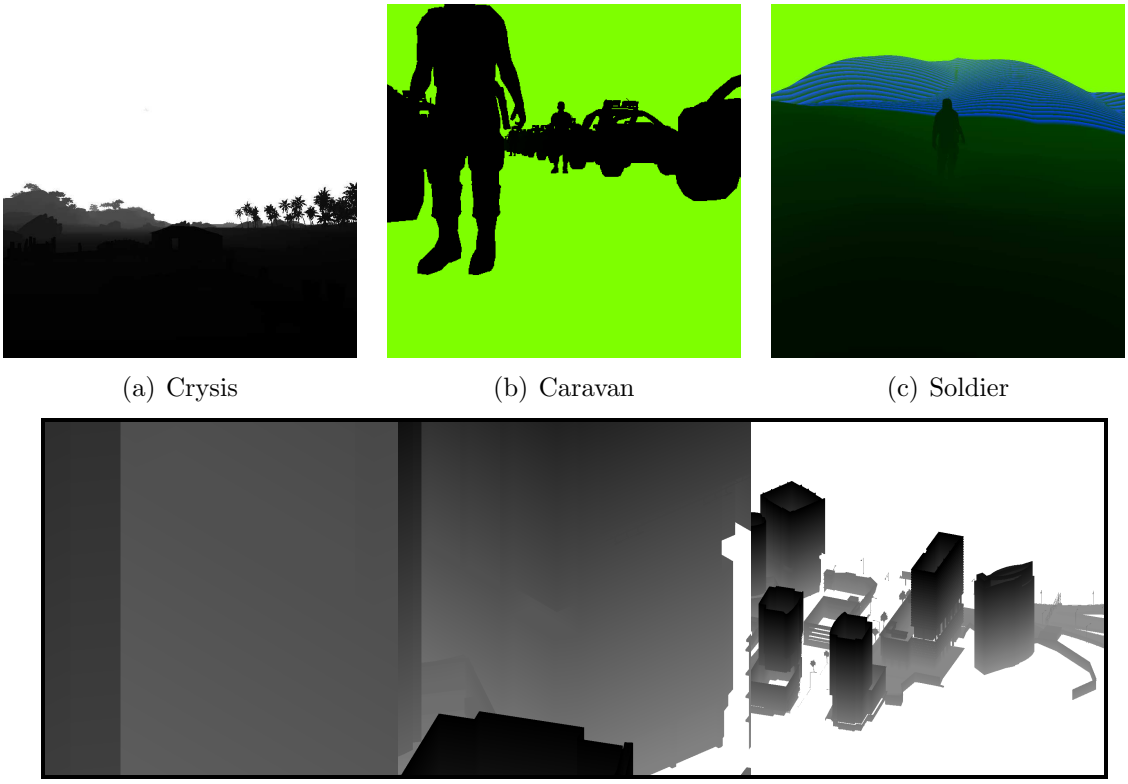
(d) Map

(e) Smoke

Figure 6.1: Color buffers used to test my compression techniques.

6.3.6 Results and Discussion

In this section, I present the compression rates achieved by the state-of-the-art lossless color and depth buffer compressor (Rasmusson et al., 2007) as well as my general-purpose compressor. I examine the impact of my three proposed techniques (dynamic



(a) Crysis (b) Caravan (c) Soldier

(d) NFS:U

Figure 6.2: The depth buffer data sets used to test my compression schemes. (I have removed the ground plane from the "Caravan" scene for ease of viewing; my simulations included it.)

bucket selection, Fibonacci encoding, and variable-precision data compression) on these two compressors. To limit the complexity of my findings, I will present each new section having implemented the previous sections' proposals. Fibonacci encoding is compared to unary encoding with dynamic bucket selection enabled, and so on.

Dynamic Bucket Selection

I first examine my dynamic bucket selection algorithm and results, shown in Figure 6.4 for my general purpose ("Unified") and modified existing ("Tiled") approaches. The first three columns of each data set show buckets of "FC/25%/50%," as in past work (Ström et al., 2008), "FC/50%/75%," a simple approach to correcting for optimistic buckets, and dynamic bucket selection. The final column of each data set ("Raw") shows the compression achieved if buckets were not imposed; the data could no longer be accessed randomly, but higher compression could be achieved. I view this as a goal,

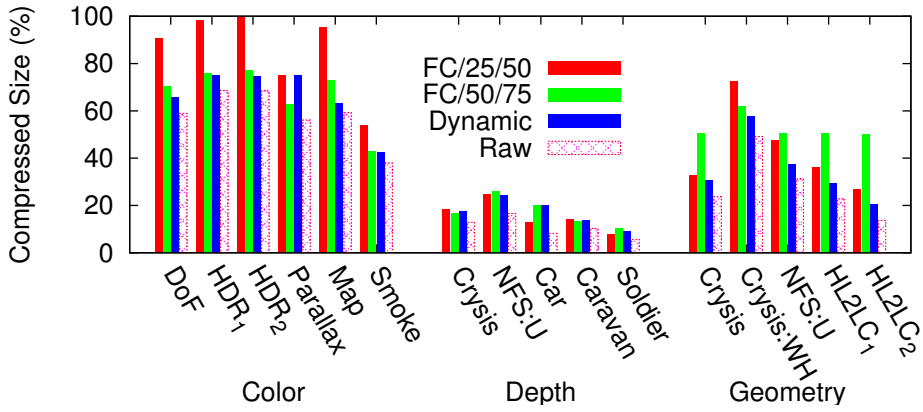
(a) NFS:U (b) HL2:LC

(c) Crysis (d) Crysis: Warhead

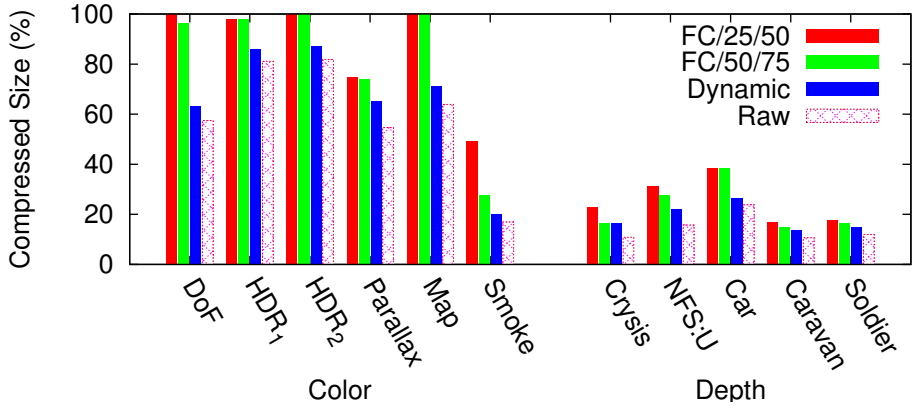
Figure 6.3: Applications from which I extracted geometry buffers for compression.

though it is unachievable in all but contrived cases in which compression rates align perfectly with bucket values.

Dynamic bucket selection outperformed static bucket selections in 22 out of 27 test cases. For my general purpose compressor, dynamic bucket selection was generally beneficial. It outperformed the two static bucket selections in 11 out of 16 cases, and by an average of 1.26x for color and geometry data. It was roughly comparable for depth data, which was already significantly compressed. Dynamic bucket selection outperforms the unmodified tile compressor in each case (by an average of 1.34x). Some test sets were simply not compressible by the "FC/25%/50%" or "FC/50%/75%" static bucket choices. The finer granularity coupled with the data-dependence of my algorithm leads to better-fitting buckets and better compression rates by an average of 1.25x over all tests.



(a) Unified

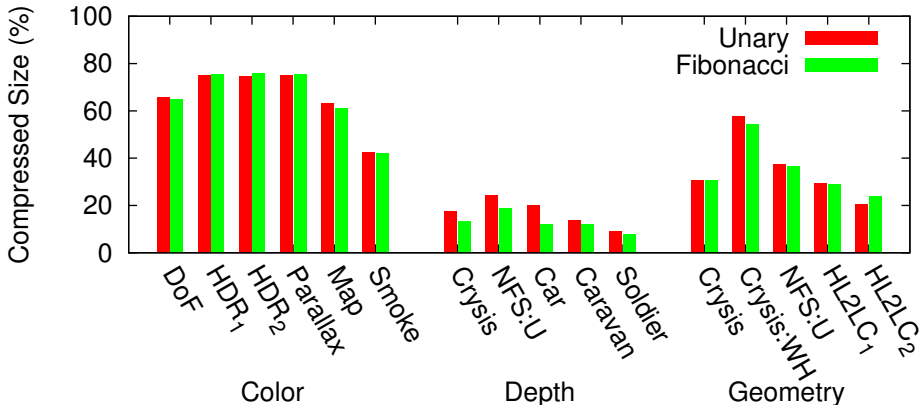


(b) Tiled

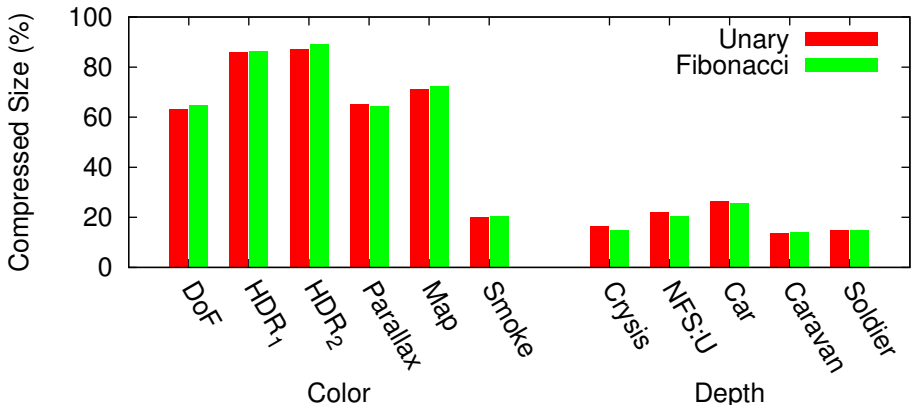
Figure 6.4: My dynamic bucket selection algorithm's performance on my proposed ("Unified") and state-of-the-art ("Tiled") compressors. In general, dynamic selection outperforms static selection by an average of 1.2x for my unified compressor and 1.3x for the tiled compressor. Ideal performance is seen in the "Raw" column.

Fibonacci Encoding

I enable dynamic bucketing when investigating the use of a Fibonacci encoder (Figure 6.5). I expected this encoder to outperform the unary encoder in most cases, since large values can map to smaller code words. The modified "tiled" compressor does not show much difference; neither encoder can be said to be definitively better. However, 12 out of 16 test cases in my unified compressor benefit from the replacement of the unary encoder with a Fibonacci encoder, by an average of 1.12x. The difference was especially positive in depth buffers, where the average increase was 1.33x. When unary encoding proved to be better (only 4 cases), the difference was no more than 1.04x. The average



(a) Unified



(b) Tiled

Figure 6.5: Performance of a Golomb-Rice encoder with standard unary and proposed Fibonacci encoders. Though the overall improvement through the Fibonacci encoder was only 1.06x, my unified compressor saw an average improvement of 1.12x, and one test case improved by 1.7x when using the Fibonacci encoder.

for both compressors was a 1.06x increase, and replacing the unary encoder with a Fibonacci encoder significantly improved compression rates in several cases. While one test case did show a 1.7x improvement over Golomb-Rice encoding, the Fibonacci encoder was not a clear improvement.

General-Purpose Data Compression

Figure 6.6 shows how my proposed unified compressor, with dynamic bucket selection and Fibonacci encoding, compares with the original and similarly-modified version of the state-of-the-art tile-based color and depth compressor (Ström et al., 2008). I

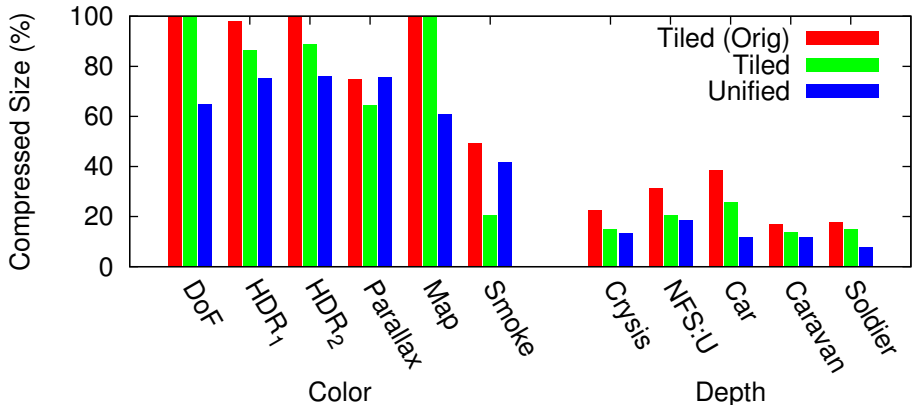


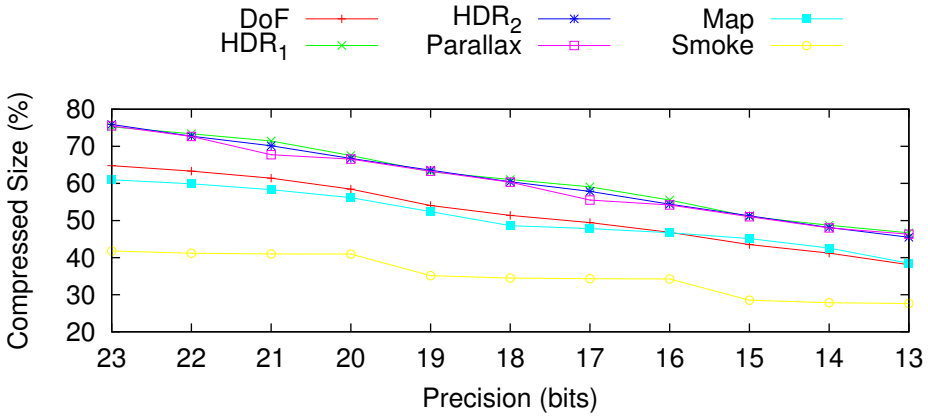
Figure 6.6: Performance of an existing compressor, that compressor augmented with dynamic bucket selection and Fibonacci encoding, and my proposed compressor with these enhancements. My unified compressor outperformed the baseline and enhanced baseline compressors by averages of 1.4x and 1.2x, respectively.

Table 6.2: Compression rates of geometric data sets.

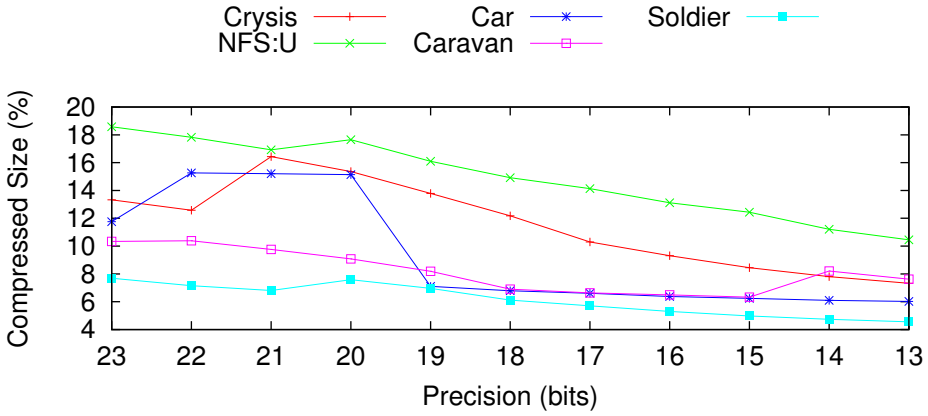
Data Set	Compressed Bandwidth (%)
Crysis	30.3
Crysis: Warhead	55.6
NFS:U	37.0
$HL2LC_1$	28.6
$HL2LC_2$	23.8

have omitted geometric data sets from this comparison, as their compression in on-chip hardware is novel to my work. Instead, Table 6.2 presents the compression rates achieved by my general purpose compressor on these data sets.

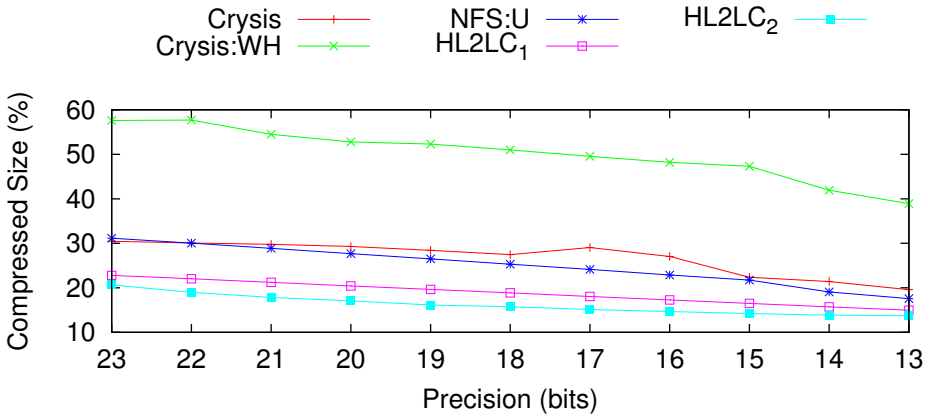
I see that in many cases, my proposed general-purpose compressor achieves better bandwidth savings. Clearly, this should be investigated further, but I propose a possible explanation: incoherent data and uncorrelated channels. The benefit of the specialized tile compressor stems from its guide bits, restart values, and rotation bit. These extra flags are intended to exploit 2D coherence in a single channel of data, say the red channel of a color buffer. When no such coherence exists, then these bits are overhead with no benefit. This tile-based approach also assumes that color channels are correlated, which, as noted in follow-up work (Wennersten and Ström, 2009), is not necessarily the case. My general-purpose compressor is able to exploit much of the same coherence without the overheads of unnecessary guide bits. There are times when these extra bits and channel correlation can make a difference, though, such as in color



(a) Color



(b) Depth



(c) Geometry

Figure 6.7: Compression rates achieved by my general-purpose compressor on color, depth, and geometry data as the precision of the data is reduced.

buffers with smoothly-changing or blocky colors ("smoke" and "parallax"). My proposed general-purpose compressor outperforms the state-of-the-art for 16-bit floating

point data adapted for 32-bit data and is also able to handle many more types of data.

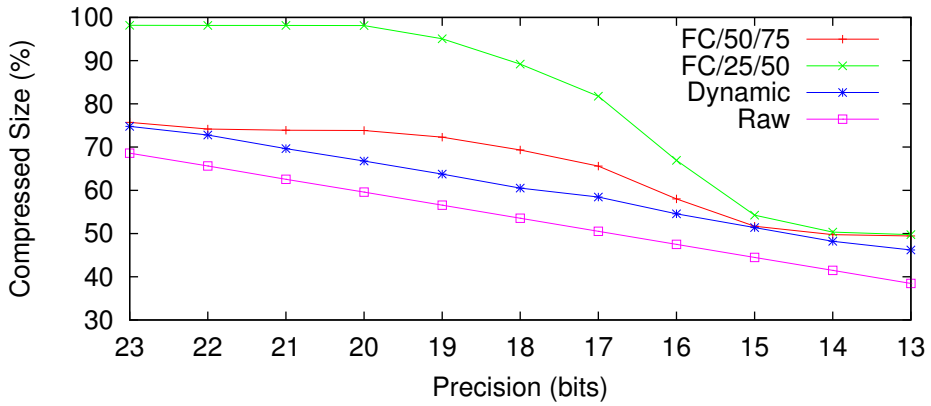


Figure 6.8: Range reduction of variable-precision data is much more effective when used with dynamic bucket selection (HDR_1 scene).

Variable-Precision Compression

As the precision of the input data is reduced, I can likewise reduce the range of that data, leading to smaller residuals. The effect of this range reduction on compression performance of color, depth, and geometry is shown in Figure 6.7. In general, I see a very promising trend: reducing the precision of the input data allows for significantly better compression rates. One minor departure from this trend occurs in some of the depth buffers. There is a discontinuity where the compressed size increases as precision decreases; this is an artifact of the dynamic bucketing. As the precision drops, new buckets are chosen, which do not fit *all* of the data well until precision drops further. However, this behavior still allows for savings and is better than having static buckets. Figure 6.8 shows the necessity of using dynamic bucketing with range reduction.

The compression rate gains expected by the range reduction of variable-precision data is hard to predict without knowing the behavior of a particular application. Chapter 5 has shown that the precision in some applications can be reduced by up to 12 bits with no noticeable errors. This type of reduction could lead to significant extra bandwidth savings.

6.4 Signal Gating of On-Chip Data

There are many areas in which data is stored on-chip, such as L1 and L2 caches, register files, and dedicated caches for texture and constant data. This data is accessed at some level for every operation performed on each vertex and pixel. In this section, I present an approach for saving energy when transmitting reduced-precision from a processor‘s register file to its L1 cache. I leave transmission across a crossbar to the L2 and beyond for future work. However, it is highly likely that a similar approach could apply to these levels of the hierarchy, as driving data buses from a cache can consume significant energy (Rodriguez and Jacob, 2006). I discuss possible approaches to these remaining levels in Section 6.4.4.

6.4.1 Approach

Reduced-precision data will have unused bits in the lower positions, so any bus activity spent on these bits is wasted effort since future computations will ignore these bits. My approach, then, is to disable any switching activity on the lines that carry these unused bits. The program being executed on each processor could change very frequently (NVIDIA Corporation, 2009), which means that the precision could also change. In turn, this means that it is not enough to count on the data stream to have back-to-back zeros in the lower bits; some active way to suppress transitions is necessary to save the maximum energy.

I propose the use of a latch at the sending side of a data bus, if such a latch is not already in use. When the data line contains valid data, the latch will be enabled; when the line contains unused bits, the latch will remain disabled, holding the current value on the line, regardless of the input data. This will suppress unnecessary transitions on the bus lines.

The latches that regulate data at the sending end of the bus will be enabled or disabled, depending on the precision of the data being transmitted. Changing state will not be a free operation; there will be some energy associated with both enabling and disabling the latch. This energy will be part of the penalty for changing precision. The other main penalty will be sending the precision information (4 bits for controlling up to 16 bits of precision) to the receiving end so it can decide how to deal with the gated bits.

6.4.2 Experimental Setup

I use LTSpice (Linear Technology, 2010) to simulate three simplex lines over two distances: 100μm and 1mm. The shorter length is driven by a single buffer stage after the latches, while three repeaters drive each of the longer lines, with neighboring lines' repeaters staggered. An RC ladder with 20 divisions approximates the resistance on the line and capacitance of the lines to each other and ground. The capacitance used for each line is 200fF/mm, and the resistance is 4.5kΩ/mm (Nguyen et al., 2005). I used a 45nm HP process from Arizona State University's Predictive Technology Model (Nanoscale Integration and Modeling Group, Arizona State University, 2012; Cao et al., 2000; Zhao and Cao, 2006; Balijepalli et al., 2007) to build the latches and drivers, which were powered with a V_{dd} of 1V. I drove each line with random signals for 500 clock cycles at 666MHz, while logging the energy spent in driving just the middle line. To find the energy savings possible, I ran the same simulation several times with different configurations of enabled latches. This led to a model to estimate the energy used by a single line based on the latch states of it and its neighbors. I performed similar experiments to find the energy required for enabling and disabling a latch. Thus, my model also includes the penalty for turning bus lines on and off.

It is important to note that a more complete study of bus energy would consider the different combinations of transitions seen by the three wires, rather than a random model (Zhang et al., 2008; Satyanarayana et al., 2009; Fan and Fang, 2011). This type of experiment would likely reveal more pronounced crosstalk characteristics that are not seen in my results. However, the data seen at this level of a GPU will likely have more coherence in the upper bits (especially if color and depth values are clamped to 1.0f), while the lower bits will likely change more from value to value. Since it is these lower bits that will be disabled, they will lead to greater energy savings compared to gating the higher bits. So, this method is a conservative estimate; a more accurate simulation might lead to even more promising results.

I exercised this model with data from real applications. For several games, I logged 100,000 output pixels from a single frame simulated by ATTILA (del Barrio et al., 2006). Each shader was sampled at the same frequency, so the mix of precisions in the final pixel selections is the same as the full frame. I built a simple simulator to send these pixel values across a bus and count transitions. Since a single processor may work on a different program at any given clock cycle, I allowed the "burst length" of the simulator to vary. This is simply how many 32-bit floating-point values from a given

shader are sent in a row before another shader program is chosen to send data, which may affect the energy consumption of the bus due to the line enable/disable penalty.

Table 6.3: Energy used in one bit line over a distance of 100μm.

Disabled Lines	Energy per Transition (fJ)	Savings (%)
None	80.532	0.0
1 Neighbor	80.868	−0.42
Self and 1 Neighbor	0.478	99.41
Self and Both Neighbors	0.545	99.32

Table 6.4: Energy used in one bit line over a distance of 1mm.

Disabled Lines	Energy per Transition (fJ)	Savings (%)
None	256.948	0.0
1 Neighbor	255.592	0.5
Self and 1 Neighbor	0.93176	99.64
Self and Both Neighbors	0.87848	99.66

6.4.3 Results

Tables 6.3 and 6.4 show the results of the energy simulations for the three simplex lines over two distances. These tables indicate that the influence of neighboring wires is minimal when tested with random data patterns. More importantly, the greater than 99% savings when disabling a latched line shows that the savings will be nearly proportional to the number of gated lines. This is a very promising result, though the overall savings will depend on the latch penalties (as discussed above). Finally, the savings are roughly the same at both wire lengths, implying that this approach is applicable to simplex buses of arbitrary distances.

Table 6.5 shows the penalties for enabling and disabling a latch. These penalties are very small when compared to the energy necessary to drive a line, especially when the data must travel a long distance. These latch penalties, coupled with the penalties for sending the encoded precision values, will lessen the benefit of disabling lines; the degree to which these penalties will detract from the overall savings will be revealed by simulating different applications' mix of precisions and data patterns.

When simulating various applications with these energy values, significant energy can be saved, regardless of the burst length, as shown in Figure 6.9. Sending 8 values (or

two RGBA pixels) at a given precision is enough to achieve most of the energy savings possible, as the extra savings at burst lengths over 8 are modest for all applications tested. However, it is highly likely that more than 8 values will be sent in a row in practice, as each multiprocessor computes values for 16 or 32 pixels at a time. The maximum savings possible will depend on the mix of precisions used in the application, but is between 13–48% for the applications I tested (see Table 6.6).

Table 6.5: Latch enable/disable penalties.

Action	Energy Penalty (fJ)
Enable	0.479
Disable	10.049

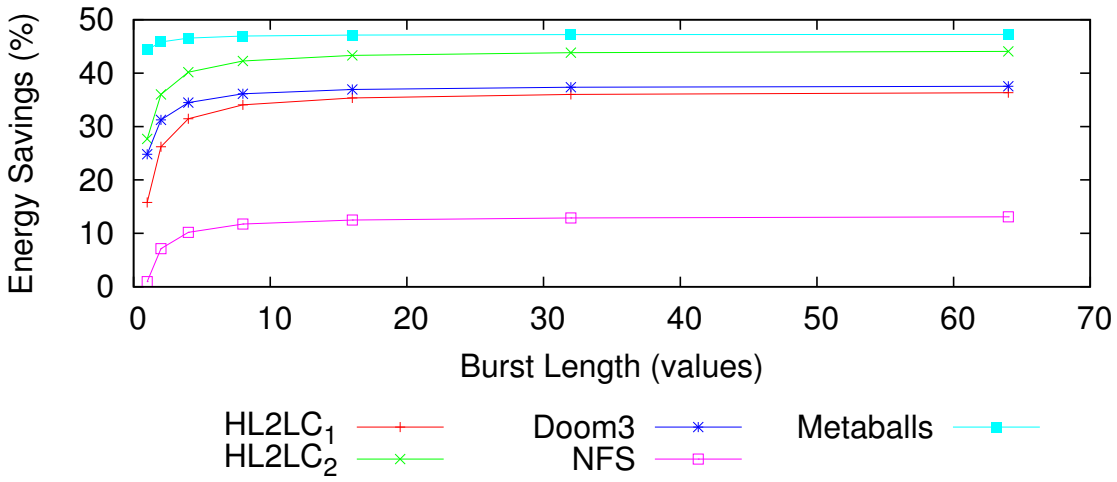


Figure 6.9: Energy savings in the bus between a processor's register file and L1 cache for several applications. Sending any number of values at the same reduced precision saves energy, but the savings increases as more values with the same precision are sent in a row.

6.4.4 Other Levels of the Memory Hierarchy

I have explored a possible data path from a multiprocessors' register file to its L1, though there are other levels of the memory hierarchy that consume significant energy. The L1 to L2 data path likely involves some sort of global interconnect, such as a crossbar. This type of bus is much more complicated than the simplex lines I explored above; switched duplex lines will connect various elements. These duplex lines will

Table 6.6: Application precisions and maximum savings (for a burst length of 64 over a 1mm simplex bus).

Application	Average Precision	Maximum Savings (%)
$HL2LC_1$	10.94	36.34
$HL2LC_2$	10.17	44.08
Doom 3	9.75	37.54
NFS	19.39	13.09
Metaballs	8.20	47.25

likely be driven by tri-state buffers to switch between reading and writing. Signal gating at this level may take a different form from the latches used above. It may be possible to use the Hi-Z state of the tri-state buffers on *both* ends of the data path to allow the lines to float and a pull-down transistor at the destination to ensure a constant signal. If this approach is found to cause signal integrity problems or spurious activity in the repeaters along the longer wires, though, latches may be used to disable transitions as above.

Another complication is the propagation of precisions from level to level of the hierarchy. At the register file and L1 level, the precision information is local to the multiprocessor, so will be available at one or both locations. Higher in the hierarchy, at the L2 and global memory levels, it will likely be necessary to consult the current rendering or computational state to determine the precision to use when reading or writing data. The necessary additional complexity will depend on the existing steps in address translation on a particular architecture, so this approach may be feasible, but a different technique may be necessary to keep the overheads to an acceptable limit.

6.5 Conclusion

I have designed a general-purpose compression and decompression scheme for 32-bit floating-point data on graphics hardware. It both outperforms an existing 16-bit compressor (Ström et al., 2008) adapted to handle 32-bit data *and* is able to compress general data. I have shown this capability by presenting promising compression rates for geometry data (vertex positions, normals, texture coordinates, etc.) for real-world applications. Average rates for color, depth, and geometry data are 1.5x, 7.9x, and 2.9x, respectively.

Furthermore, I have proposed two novel techniques applicable to any hardware

compression scheme: dynamic bucket selection and the use of a Fibonacci encoder. These proposals increased compression ratios by averages of 1.25x and 1.06x, with maximum improvements of 2.4x and 1.7x, respectively. Note that these are not just compression rates, this also takes quantized storage into account. So, these results should not be viewed as a single tile seeing an improvement of 1.25x (for example) but as several tiles remaining unchanged, and several others improving by 2x. I believe that these techniques are suitable for a hardware implementation and discussed my justification. Lastly, I have shown that extra savings are available by using range reduction on variable-precision data. The additional savings will depend on the specific application but are expected to be between 5% and 20%, for overall color, depth, and geometry compression rates of 1.9x, 10.7x, and 3.6x, respectively.

I have also conducted a preliminary exploration of the savings possible in on-chip communication by performing Spice simulations of 3 simplex lines and expanding the results into a larger on-chip bus model. By applying a form of signal gating on bus lines from a multiprocessor's register file to its L1 cache, an average savings of 36% (between 13–48%) is possible when transmitting reduced-precision data from five test applications.

6.5.1 Future Work

My dynamic bucket selection algorithm has been shown to work well in practice. However, its performance *is* dependent on the order in which the data is seen; a flipped or rotated buffer could drastically change the results. Extensions to this algorithm may be possible, such as delaying the selection of a bucket until some minimum number of chunks fall into it. Another approach may be to dynamically re-select buckets with the realization that moving a selected bucket from a smaller value to a higher value does not pose any functional problems; any buffers mapped to the selected bucket will still fit in its new value.

Compression of 32-bit data is important, but many scientific applications also make use of 64-bit floating-point representations. This is an obvious extension of my 32-bit unified compressor. Handling 32- and 16-bit data with a 64-bit compressor need not be complicated, however, if dynamic range reduction is also used. For example, 32-bit values can be promoted to 64-bit values and padded with zeros on the right. Since the comparison constant is chosen for 64-bit values, dynamic range reduction will handle shifting this value at the same time that the data values are shifted by

32 bits. The efficiency of the compressor will then be identical to the efficiency of a 32-bit compressor, though far more general. This idea can also be used to handle 16-bit data in a 64-bit compressor, or even 16-bit data in my 32-bit compressor. This will also allow for very common 8-bit formats to be compressed with my scheme. While techniques targeted at 8-bit data will not be used in my general-purpose compressor, these often-used formats can still be handled. I will have to test this technique before reporting any performance results, though the extra hardware and control will likely mean that a dedicated 8-bit compressor will be more energy-efficient than my suggested approach for 8-bit data sets.

Compression of geometry buffers is often able to be asymmetric; many game applications have geometry that is authored once and read many times. Thus, it is reasonable to expect that a two-pass algorithm could be used on the data after authoring in order to choose the best buckets for a particular buffer. My dynamic selection decompression scheme would still be necessary to make use of these buffers, as used buckets are still a subset of the available buckets.

There are still many areas to explore in on-chip communication energy efficiency. I have only addressed possible savings in simple buses, such as from a register file to an L1 cache. More complex interconnects remain, such as those from the L1 to a shared L2, which may involve a crossbar or other switched network. Moreover, savings in data storage, such as in register files, caches, and SRAMs, may be available. Further work will examine each of these areas.

Chapter 7

Summary and Conclusion

7.1 Summary

GPUs are used across the continuum of computers—mobile devices, personal computers, and high-performance computing (HPC) servers. In each of these domains, energy consumption is increasingly becoming a primary concern to hardware and software designers. Mobile devices, such as cellular phones and tablets, are constrained by their batteries; the more energy-efficient the device, the longer it will last before requiring user intervention to recharge or replace the battery. The performance of personal computers and HPC servers is limited by the power that can be drawn from the power supply and then dissipated on the chip. Were these computers to be more efficient, they could operate at higher frequencies on the same power budget. In Chapter 1, I proposed a method to trade off precision for energy in the operation of the graphics pipeline, allowing for longer battery lifetimes or higher performance (depending on the larger context of the GPU). Now, let me recap how the pieces of my work—an energy model, variable-precision arithmetic hardware, and explorations in saving energy in both the computation and communication of data—fit together to enable these tradeoffs.

7.1.1 Energy Model

I presented an energy model for graphics applications on GPUs in Chapter 3. My model works by aggregating the energy used in each operation in the pipeline (both programmable and fixed-function stages, and both memory and arithmetic operations) to estimate the overall energy used by the hardware. Thus, no time-consuming simulations or repeated run-time monitoring of existing hardware is needed; hypothetical

architectures can be explored without having to build or simulate test chips. I validated my model with different scenes from three commercial games and found it to be accurate to within 10–15% for each test. I then used the model in two case studies: an algorithmic study to find the most energy-efficient method to perform vector normalization and an architectural study to determine how efficient a tiled renderer might be when rendering the test scenes.

This model, then, allows for rapid and accurate estimates of the energy efficiency of both existing and in-design hardware for any workload. Further, it estimates the energy used by each section of the graphics pipeline, giving insight into the most fruitful avenues to explore for energy savings. This also allows me to translate savings in any one section to savings in the context of the entire GPU.

7.1.2 Variable-Precision Hardware

Hao and Varshney were the first to look at variable-precision arithmetic in the graphics pipeline (Hao and Varshney, 2001), though it was for performance reasons. Since I seek to save energy, I developed variable-precision arithmetic circuits that use less energy as fewer bits are computed; this is the subject of Chapter 4. These are standard adders and multipliers that have been modified to accommodate fine-grain power gating to save both dynamic and leakage power. The energy versus precision curves are very promising; the adders have linear savings with each bit of reduced precision, and the multipliers have quadratic savings. I use the energy characteristics of these arithmetic circuits in later sections to estimate energy savings in graphics applications. Lastly, the timing and area overheads are acceptable and small, and they do not preclude mobile or desktop hardware from using these circuits.

7.1.3 Energy Savings in Computation

With an energy model and variable-precision circuits in hand, in Chapter 5, I explored the energy savings possible in the computation of data in vertex and pixel shaders. As precision is reduced in either of these places, different types of errors are manifested, so I categorized and, where possible, quantified these errors. Using a GPU simulator, I developed and presented several techniques for choosing an operating precision that will yield significant energy savings without incurring intolerable errors. Using one such technique in which the artist is responsible for choosing the precision for each effect used in an application, up to 79% of the energy in the pixel shader's arithmetic could be

saved (with an average savings of 70% for the applications tested with this technique). Similar savings are possible for the computation in vertex shaders.

7.1.4 Energy Savings in Communication

Having shown that energy can be saved by reducing the precision of data's computation, the hardware can take advantage of unused bits in the data's representation when moving data around and off of the chip. So, in Chapter 6, I look at approaches for saving energy in the communication of data. Data is usually compressed before it is read from or written to an off-chip memory, such as the GPU's global DRAM. I propose two enhancements (dynamic bucket selection and Fibonacci encoding) for existing compression schemes that increase their compression rates by an average of 1.25x and 1.06x, with maximum improvements of 2.4x and 1.7x, respectively. I also describe a new unified compressor that is able to operate on any type of data, not just a subset of color and depth buffers, thereby overcoming a major limitation of past work. This new compressor achieves compression rates of 1.5x, 7.9x, and 2.9x for color, depth, and geometry buffers, respectively. As the last piece in data compression, I propose a method for compressing reduced-precision data, which allows additional energy and bandwidth savings of from 5–20%, depending on the application.

I also conducted a preliminary exploration into saving energy in on-chip communication. By using signal gating on simplex buses, such as from a multiprocessor's register file to its L1 cache, linear energy savings can be realized as the number of bits gated increases. The aggregate savings will depend on the usage pattern of this bus, since switching precisions will carry with it a penalty, so I developed a simple simulator that will estimate this overall savings given an application's mix of the data and precisions used by each shader. For several test applications, savings were between 13–48%, with an average of 36%.

7.2 Future Work

In performing this research, I explored many avenues for energy savings as thoroughly as I could. However, there are still many relatively untouched areas or topics that warrant further investigation, and I discuss them in this section.

7.2.1 Energy Model

Though my model takes into account both fixed-function and programmable units, I did not characterize the energy necessary to perform triangle setup. I expect this cost will be relatively low in relation to the other operations but cannot be certain without performing directed tests and measuring the energy requirements. Further, the GPU (NVIDIA's 8300GS) for which I developed my model is not the most up-to-date architecture or transistor process available. The 80nm transistors used in this card have very different characteristics than the newest 28nm (and smaller) technologies, and there have been architectural changes in recent years that will affect the model's accuracy on newer hardware. These changes may require more directed tests to find, for example, the relative energy costs of L1 and L2 cache accesses. Lastly, when applying my model to different graphical applications, I was forced to estimate many aspects of the workload seen on the hardware; results would likely be much more reliable and accurate if a more direct way of finding values for model parameters were available. Applications with vastly different characteristics may require additional model parameters.

7.2.2 Variable-Precision Hardware

Though my designs show that trading off energy for precision is a viable approach to saving energy, they are not sufficient for a completely accurate estimation of absolute savings. First, I have designed integer arithmetic circuits, while any implementation in a GPU would need floating-point support. So, extra hardware to support the addition and multiplication of floating-point exponents and sign bits, as well as rounding, is needed. Also, these operations may be pipelined, so intermediate latches may be necessary to store the data after each stage. Second, these various circuits need to be assembled into an FPU to be functional in larger tests; there are control overheads that are not accounted for in my designs. Finally, these circuits were designed and simulated for a technology size of 130nm; this is quite large by today's standards and should be updated before proceeding further. This updating process may lead to different energy savings characteristics that lead to different types of power gating designed to amortize the cost of a header or footer switch over more gated transistors, as power gating is currently applied to larger groups of transistors than those used in my designs.

In my designs, I forced a granularity of per-bit power gating; it may be interesting to explore the savings possible with power gating at a coarser granularity, e.g. at every

other bit. In this scenario, the timing and area penalties may be lower, control circuitry more compact, and control algorithms not as complicated, all while still allowing for significant savings. The realizable energy reductions will depend on the precisions tolerated by the final applications running on the hardware.

7.2.3 Energy Savings in Computation

I focused on the vertex and pixel shader stages of the graphics pipeline, but there are currently other stages that warrant their own investigations: the geometry, tessellation, and compute shaders. This last stage is meant to enhance interoperability between graphics and general-purpose computations, and so its execution on variable-precision hardware could be approximated with GPGPU applications running as CUDA programs. In any case, I have not performed any tests on GPGPU applications, despite promising results found in reduced-precision physics simulations (Yeh et al., 2006; Yeh et al., 2009). In the HPC domain, researchers are accustomed to using the highest precision available, and will likely not want to risk reducing the precision of their computations for energy savings. It will likely be necessary to perform automatic algorithm restructuring and static precision analysis to guarantee maximum errors and achieve significant energy savings.

7.2.4 Energy Savings in Communication

In the compression of data for off-chip transmission, I explored simple approaches to handling the data. For example, I proposed a simple scheme for dynamic bucket selection. While, in many of my test cases, this works well, it is dependent on the order in which the data appears. There may be a different (but still simple and easy to implement in hardware!) heuristic that leads to better-fitting buckets for the whole data set. For instance, this could take the form of changing the 62.5% entry in a bucket list also containing 37.5% and 50% to 75%; this would not affect the correctness of the already compressed data's decompression, but it may allow for more buffer chunks to be compressed. I also made the assumption that all data buffers are writeable; in reality, many geometry buffers are read-only. This fact may open new opportunities for asymmetric decompression, perhaps by using a two-pass algorithm on the data as it is authored, similar to texture compression.

BIBLIOGRAPHY

Adex Electronics, Inc. (2008). PEX16LX: PCI Express X16 Bus Tall Extender. `http://www.adexelec.com/pciexp.html#PEX16LX`.

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

Al Maashri, A., Sun, G., Dong, X., Narayanan, V., and Xie, Y. (2009). 3D GPU architecture using cache stacking: performance, cost, power and thermal analysis. In *Proceedings of the 2009 IEEE international conference on Computer design*, ICCD'09, pages 254–259, Piscataway, NJ, USA. IEEE Press.

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

Apple Inc. (2010). What's new in iPhone OS 4. `http://developer.apple.com/technologies/iphone/whats-new.html`.

ATTILA (2011). Traces - AttilaWiki. `http://attila.ac.upc.edu/traceList/`.

Bahar, R., Albera, G., and Manne, S. (1998). Power and performance tradeoffs using various caching strategies. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 64 –69.

Baker, P. (2011). Metaballs II. `http://www.paulsprojects.net/metaballs2/metaballs2.html`.

Balijepalli, A., Sinha, S., and Cao, Y. (2007). Compact modeling of carbon nanotube transistor for early stage process-design exploration. In *Proceedings of the 2007 international symposium on Low power electronics and design*, ISLPED '07, pages 2–7, New York, NY, USA. ACM.

Benini, L., De Micheli, G., and Macii, E. (2001). Designing low-power circuits: practical recipes. *Circuits and Systems Magazine, IEEE*, 1(1):6 –25.

BioWare (2007). Mass Effect. `http://masseffect.bioware.com/me1/`.

Brent, R. and Kung, H. (1982). A regular layout for parallel adders. *Computers, IEEE Transactions on*, C-31(3):260 –264.

Brooks, D., Tiwari, V., and Martonosi, M. (2000). Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 83–94, New York, NY, USA. ACM.

Burtscher, M. and Ratanaworabhan, P. (2009). FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.*, 58:18–31.

Butzen, P. F., Reis, A. I., Kim, C. H., and Ribas, R. P. (2007). Modeling and estimating leakage current in series-parallel cmos networks. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, GLSVLSI '07, pages 269–274, New York, NY, USA. ACM.

Callaway, T. and Swartzlander, E.E., J. (1997). Power-delay characteristics of CMOS multipliers. In *13th IEEE Symposium on Computer Arithmetic, 1997*, pages 26 –32.

Cao, Y., Sato, T., Orshansky, M., Sylvester, D., and Hu, C. (2000). New paradigm of predictive mosfet and interconnect modeling for early circuit simulation. *Proceedings of the IEEE 2000 Custom Integrated Circuits Conference Cat No00CH37044*, pages 201–204.

Chen, D., Zhou, B., Guo, Z., and Nilsson, P. (2005). Design and implementation of reciprocal unit. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 1318 –1321 Vol. 2.

Chen, R. Y., Irwin, M. J., and Bajwa, R. S. (2001). Architecture-level power estimation and design experiments. *ACM Transactions on Design Automation of Electronic Systems*, 6:50–66.

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, pages 251 – 256.

Chowdhury, M., Gjanci, J., and Khaled, P. (2008). Innovative power gating for leakage reduction. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1568 –1571.

Crytek (2007). Crysis. `http://www.ea.com/crysis-1`.

Crytek Budapest (2008). Crysis Warhead. `http://www.ea.com/crysis-warhead`.

Danilak, R. (2009). Efficient multi-chip GPU. *US Patent US 7,616,206 B1.*

Deering, M. (1995). Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 13–20, New York, NY, USA. ACM.

del Barrio, V., Gonzalez, C., Roca, J., Fernandez, A., and E, E. (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, pages 231 – 241.

Dmitriev, K. and Moreton, H. P. (2011). Method for watertight evaluation of an approximate Catmull-Clark surface. *US Patent US2011/0085736 A1*.

EA Black Box (2008). Need for Speed: Undercover. `http://undercover.needforspeed.com/home.action`.

Eidos Interactive Ltd. (2009). Batman: Arkham Asylum. `http://www.batmanarkhamasylum.com`.

Eisley, N. and Peh, L.-S. (2004). High-level power analysis for on-chip networks. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 104–115, New York, NY, USA. ACM.

Eisley, N., Soteriou, V., and Peh, L.-S. (2006). High-level power analysis for multi-core chips. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 389–400, New York, NY, USA. ACM.

Ercegovac, M., Lang, T., Muller, J.-M., and Tisserand, A. (2000). Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *Computers, IEEE Transactions on*, 49(7):628 –637.

Fan, C.-P. and Fang, C.-H. (2011). Efficient rc low-power bus encoding methods for crosstalk reduction. *Integr. VLSI J.*, 44:75–86.

Foskett, N., Prevett, R., and Treichler, S. (2006). Method and system for performing pipelined reciprocal and reciprocal square root operations. *US Patent 7117238*.

Fraenkel, A. S. and Klein, S. T. (1996). Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64:31–55.

Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., and Cameron, K. W. (2010). PowerPack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21:658–671.

Goel, V. and Martin, T. (2009). Merged shader for primitive amplification. *US Patent US2009/0295804 A1*.

Gruetzmacher, G. P. (2010). Method and apparatus for model compression. *US Patent Application 2010/0008593 A1*.

Hao, X. and Varshney, A. (2001). Variable-precision rendering. In *2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 149–158, New York, NY, USA. ACM.

Harris, D. (2003). A taxonomy of parallel prefix networks. In *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 2, pages 2213–2217.

Hasselgren, J. and Akenine-Möller, T. (2006). Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 103–110, New York, NY, USA. ACM.

Hidetoshi and Yokoo (1994). Adaptive encoding for numerical data compression. *Information Processing & Management*, 30(6):863 – 873.

Hong, S. and Kim, H. (2010). An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 280–289, New York, NY, USA. ACM.

Huang, S., Xiao, S., and Feng, W. (2009). On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA. IEEE Computer Society.

Huang, Z. and Ercegovac, M. (2001). On signal-gating schemes for low-power adders. In *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, volume 1, pages 867–871.

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, pages 489 – 492.

Huang, Z. and Ercegovac, M. (2005). High-performance low-power left-to-right array multiplier design. *IEEE Transactions on Computers*, 54(3):272 – 283.

Huang, Z. and Ercegovac, M. D. (2003). Two-dimensional signal gating for low power in high-performance multipliers. *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, 5205(1):499–509.

Hung, Y.-C., Chen, J.-C., Shieh, S.-H., and Tung, C.-K. (2009). A survey of low-voltage low-power technique and challenge for CMOS signal processing circuits. In *Integrated Circuits, ISIC '09. Proceedings of the 2009 12th International Symposium on*, pages 554 –557.

id (2005). Doom 3. `http://idsoftware.com/games/doom/doom3/index.php`.

Imagination Technologies Ltd. (2010). POWERVR Graphics. `http://www.imgtec.com/powervr/powervr-graphics.asp`.

Iourcha, K. I., Nayak, K. S., and Hong, Z. (1999). System and method for fixed-rate block-based image compression with inferred pixel values. *US Patent 5956431*.

Isenburg, M., Lindstrom, P., and Snoeyink, J. (2005). Lossless compression of predicted floating-point geometry. *Comput. Aided Des.*, 37:869–877.

Jain, S. A. (2003). Low-power single-precision IEEE floating-point unit. Master's thesis, Massachusetts Institute of Technology.

Jiang, H. and Marek-Sadowska, M. (2008). Power gating scheduling for power/ground noise reduction. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 980–985, New York, NY, USA. ACM.

Keckler, S., Dally, W., Khailany, B., Garland, M., and Glasco, D. (2011). GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7 –17.

Kim, H.-O. and Shin, Y. (2006). Analysis and optimization of gate leakage current of power gating circuits. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 565–569, Piscataway, NJ, USA. IEEE Press.

Kim, S., Kosonocky, S. V., Knebel, D. R., and Stawiasz, K. (2004). Experimental measurement of a novel power gating structure with intermediate power saving mode. In *Proceedings of the 2004 international symposium on Low power electronics and design*, ISLPED '04, pages 20–25, New York, NY, USA. ACM.

Kulkarni, P., Gupta, P., and Ercegovac, M. (2011). Trading accuracy for power with an underdesigned multiplier architecture. In *Proceedings of the 2011 24th International Conference on VLSI Design*, VLSID '11, pages 346–351, Washington, DC, USA. IEEE Computer Society.

Kwong, M. (2005). Low power, variable precision DDA for 3D graphics applications. *US Patent 6947056*.

Lafruit, G., Nachtergaele, L., Denolf, K., and Bormans, J. (2000). 3D computational graceful degradation. In *Circuits and Systems. Proceedings of the 2000 IEEE International Symposium on*, volume 3 of *ISCAS 2000*, pages 547 –550 vol.3.

Lahiri, K. and Raghunathan, A. (2004). Power analysis of system-level on-chip communication architectures. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 236–241, New York, NY, USA. ACM.

Lauritzen, A. (2007). Variance shadow maps demo (D3D10). `http://www.punkuser.net/savsm/variance_shadow_map_d3d10_2007-04-26.zip`.

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, pages 309 –318.

Liao, W., Basile, J. M., and He, L. (2002). Leakage power modeling and reduction with data retention. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 714–719, New York, NY, USA. ACM.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55.

Lindkvist, T. and Lofvenberg, J. (2005). Minimal redundancy, low power bus coding. In *NORCHIP Conference, 2005. 23rd*, pages 277 – 280.

Lindstrom, P. and Isenburg, M. (2006). Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12:1245–1250.

Linear Technology (2010). LTSpice. `http://www.linear.com/designtools/software/#Spice`.

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, pages 301 – 306.

Mao, J., Zhao, Q., and Cassandras, C. (2004). Optimal dynamic voltage scaling in power-limited systems with real-time constraints. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 2, pages 1472 – 1477 Vol.2.

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

Microsoft (2011b). Texture block compression in Direct3D 11. `http://msdn.microsoft.com/en-us/library/windows/desktop/hh308955(v=vs.85).aspx`.

Microsoft Corporation (2012a). Direct3D Reference: Shader Model 3. `http://msdn.microsoft.com/en-us/library/bb147365`.

Microsoft Corporation (2012b). Programming Guide for Direct3D 10: Block Compression. `http://msdn.microsoft.com/en-us/library/bb694531`.

Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. (1994). A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32.

Mueller, C. (1995). Architectures of image generators for flight simulators. Technical Report 95-015, University of North Carolina at Chapel Hill.

Mukhopadhyay, S., Raychowdhury, A., and Roy, K. (2003). Accurate estimation of total leakage current in scaled cmos logic circuits based on compact current modeling. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 169–174, New York, NY, USA. ACM.

Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., and Matsuoka, S. (2010). Statistical power modeling of GPU kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115 –122.

Nanoscale Integration and Modeling Group, Arizona State University (2012). Predictive Technology Model. http://ptm.asu.edu/.

Ngo, H. T. and Asari, V. K. (2009). Partitioning and gating technique for low-power multiplication in video processing applications. *Microelectron. J.*, 40:1582–1589.

Nguyen, V., Christie, P., Heringa, A., Kumar, A., and Ng, R. (2005). An analysis of the effect of wire resistance on circuit level performance at the 45-nm technology node. In *Interconnect Technology Conference, 2005. Proceedings of the IEEE 2005 International*, pages 191–193.

NVIDIA Corporation (2004). Normalization heuristic. http://developer.nvidia.com/object/normalization_heuristics.html.

NVIDIA Corporation (2009). Fermi compute architecture white paper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

NVIDIA Corporation (2010). NVIDIA Direct3D SDK 10 code samples. http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html.

NVIDIA Corporation (2012). The benefits of multiple CPU cores in mobile devices. http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf.

Oh, K.-I., Cho, S., and Kim, L.-S. (2006). A low power soc bus with low-leakage and low-swing technique. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, page 4 pp.

Park, J., Choi, J. H., and Roy, K. (2010). Dynamic bit-width adaptation in DCT: an approach to trade off image quality and computation energy. *IEEE Trans. Very Large Scale Integr. Syst.*, 18:787–793.

Persson, E. (2006). Infinite terrain II. http://www.humus.name/index.php?page=3D&ID=65.

Phatak, D., Kahle, S., Kim, H., and Lue, J. (1998). Hybrid signed-digit representation for low power arithmetic circuits. In *Proceedings of the Low Power Workshop in Conjunction with ISCA*.

Ramani, K., Ibrahim, A., and Shimizu, D. (2007). PowerRed: A flexible simulation framework for graphics architectures. In *Procceedings of the First Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU '07.

Ramey, W. O., Moreton, H. P., and Rogers, D. H. (2008). Decompression of vertex data using a geometry shader. *US Patent US 2008/0266287 A1*.

Rasmusson, J., Hasselgren, J., and Akenine-Möller, T. (2007). Exact and error-bounded approximate color buffer compression and decompression. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 41–48, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Rasmusson, J., Ström, J., and Akenine-Möller, T. (2009). Error-bounded lossy compression of floating-point color buffers using quadtree decomposition. *Vis. Comput.*, 26:17–30.

Rastogi, A., Ganeshpure, K. P., Sanyal, A., and Kundu, S. (2008). On composite leakage current maximization. *J. Electron. Test.*, 24:405–420.

Ratanaworabhan, P., Ke, J., and Burtscher, M. (2006). Fast lossless compression of scientific floating-point data. In *Proceedings of the Data Compression Conference*, pages 133–142, Washington, DC, USA. IEEE Computer Society.

Rockstar Games (2008). Grand Theft Auto: IV. `http://www.rockstargames.com/IV/`.

Rodriguez, S. and Jacob, B. (2006). Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm). In *Proceedings of the 2006 international symposium on Low power electronics and design*, ISLPED '06, pages 25–30, New York, NY, USA. ACM.

Rofouei, M., Stathopoulos, T., Ryffel, S., Kaiser, W., and Sarrafzadeh, M. (2008). Energy-aware high performance computing with graphic processing units. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 11–11, Berkeley, CA, USA. USENIX Association.

Roy, K., Mukhopadhyay, S., and Mahmoodi-Meimand, H. (2003). Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305 – 327.

Sathanur, A., Calimera, A., Pullini, A., Benini, L., Macii, A., Macii, E., and Poncino, M. (2008). On quantifying the figures of merit of power-gating for leakage power minimization in nanometer CMOS circuits. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 2761 –2764.

Satyanarayana, N., Babu, A. V., and Mutyam, M. (2009). Delay-efficient bus encoding techniques. *Microprocessors and Microsystems*, 33(56):365 – 373.

Sheaffer, J. W., Luebke, D., and Skadron, K. (2004). A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, pages 85–94, New York, NY, USA. ACM.

Sheikh, B. and Manohar, R. (2010). An operand-optimized asynchronous IEEE 754 double-precision floating-point adder. In *Asynchronous Circuits and Systems, 2010 IEEE Symposium on*, ASYNC 2010, pages 151 –162.

Shi, K. and Howard, D. (2006). Sleep transistor design and implementation - simple concepts yet challenges to be optimum. In *VLSI Design, Automation and Test, 2006 International Symposium on*, pages 1 –4.

Sjalander, M., Drazdziulis, M., Larsson-Edefors, P., and Eriksson, H. (2005). A low-leakage twin-precision multiplier using reconfigurable power gating. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 1654 – 1657 Vol. 2.

Ström, J., Wennersten, P., Rasmusson, J., Hasselgren, J., Munkberg, J., Clarberg, P., and Akenine-Möller, T. (2008). Floating-point buffer compression in a unified codec architecture. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 75–84, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

The University of California at Berkeley (2010). Spice Simulator. `http://bwrc.eecs.berkeley.edu/classes/icbook/spice/`.

Tiwari, V., Malik, S., Wolfe, A., and Lee, M. T.-C. (1996). Instruction level power analysis and optimization of software. In *Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication*, VLSID '96, pages 326–, Washington, DC, USA. IEEE Computer Society.

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):273 –286.

Usami, K., Nakata, M., Shirai, T., Takeda, S., Seki, N., Amano, H., and Nakamura, H. (2009a). Implementation and evaluation of fine-grain run-time power gating for a multiplier. In *IC Design and Technology, 2009. ICICDT '09. IEEE International Conference on*, pages 7 –10.

Usami, K., Shirai, T., Hashida, T., Masuda, H., Takeda, S., Nakata, M., Seki, N., Amano, H., Namiki, M., Imai, M., Kondo, M., and Nakamura, H. (2009b). Design and implementation of fine-grain power gating with ground bounce suppression. In *VLSI Design, 2009 22nd International Conference on*, pages 381 –386.

Valve (2005). Half-Life 2: Lost Coast. `http://store.steampowered.com/app/340`.

Varma, A., Debes, E., Kozintsev, I., Klein, P., and Jacob, B. (2008). Accurate and fast system-level power modeling: An XScale-based case study. *ACM Transactions on Embedded Computing Systems*, 7:25:1–25:20.

Walters, E.G., I. and Schulte, M. (2005). Efficient function approximation using truncated multipliers and squarers. In *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 232 – 239.

Wang, X. and Leeser, M. (2010). VFloat: A variable precision fixed- and floating-point library for reconfigurable hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 3:16:1–16:34.

Wang, Y., Xu, J., Xu, Y., Liu, W., and Yang, H. (2010). Power gating aware task scheduling in MPSoC. *Ieee Transactions On Very Large Scale Integration Vlsi Systems*, 19(10):1801–1812.

Wennersten, P. and Ström, J. (2009). Table-based alpha compression. *Computer Graphics Forum*, 28(2):687–695.

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

Williams, L. (1983). Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '83, pages 1–11, New York, NY, USA. ACM.

Wittenbrink, C. and Ordentlich, E. (2005). Sort middle, screen space, graphics geometry compression through redundancy elimination. *US Patent 6961469 B2*.

Yeap, G. C.-F. (2002). Leakage current in low standby power and high performance devices: trends and challenges. In *Proceedings of the 2002 international symposium on Physical design*, ISPD '02, pages 22–27, New York, NY, USA. ACM.

Yeh, T. Y., Faloutsos, P., and Reinman, G. (2006). Enabling real-time physics simulation in future interactive entertainment. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, pages 71–81, New York, NY, USA. ACM.

Yeh, T. Y., Reinman, G., Patel, S. J., and Faloutsos, P. (2009). Fool me twice: Exploring and exploiting error tolerance in physics-based animation. *ACM Trans. Graph.*, 29:5:1–5:11.

Yifei, J. and Dandan, H. (2010). Improved texture compression for S3TC. In *Picture Coding Symposium (PCS), 2010*, pages 386 –389.

Yoshizawa, S. and Miyanaga, Y. (2006). Tunable wordlength architecture for a low power wireless OFDM demodulator. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences*, E89-A:2866–2873.

Zhang, J., Wu, Q., and Qiu, Q. (2008). Bus encoding for simultaneous delay and energy optimization. In *Proceedings of the 13th international symposium on Low power electronics and design*, ISLPED '08, pages 209–212, New York, NY, USA. ACM.

Zhang, Y., Hu, Y., Li, B., and Peng, L. (2011). Performance and power analysis of ATI GPU: A statistical approach. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 149 –158.

Zhao, W. and Cao, Y. (2006). New generation of predictive technology model for sub-45nm design exploration. In *Proceedings of the 7th International Symposium on Quality Electronic Design*, ISQED '06, pages 585–590, Washington, DC, USA. IEEE Computer Society.

Zhao, X., Tian, X., Yan, S., and Guan, Y. (2007). A novel low power bus coding technique for nanometer technology. In *ASIC, 2007. ASICON '07. 7th International Conference on*, pages 1066 –1069.