GPU ENERGY MODELING AND ANALYSIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Zain Asgar
June 2015

This dissertation is online at: http://purl.stanford.edu/qb097xt0874

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Montrym, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Over the past couple of decades GPUs have enjoyed tremendous scaling in both functionality and performance by focusing on area efficient processing. However, the slowdown in supply voltage scaling has created a new hurdle to continued scaling of GPU performance. This slowdown in voltage scaling has caused power consumption to limit the achievable GPU performance. Since GPUs currently use many of the well-known hardware techniques for reduced power consumption, GPU designers need to start looking at architectural techniques to improve energy efficiency. This work explores how construct an accurate energy model to enable this architectural exploration. We also use our model to explore potential energy reduction techniques.

For the energy model, we utilize both a detailed model that can be adjusted for a large number of parameters (for example: clock speed, pipeline depth, etc.) as well as a regression based designs blocks derived from existing designs. The model achieved high correlation when compared to actual GPU designs.

Using our model we performed several "what if" studies to explore opportunities for energy savings. One of the studies we looked as was reducing the precision of computation. We found that this optimization can yield 20-30% energy savings in current GPUs without sacrificing image quality.

We then used our model to conduct two studies related to reducing the amount of work done by the GPU. The first one is reducing thread level redundancy (scalarization), and the other is reducing overdraw (which occurs when a given pixel's value is computed more than once). Reducing thread level redundancy can potentially yield significant energy savings of 20 - 50%. Overdraw reduction yielded a much smaller benefit that we initially expected. Interestingly, we found that there is significant

amounts of overdraw in most of the frames that we studied. However, the energy savings are less than 15% even with an over draw of 2.3X (meaning that every pixel on average is drawn more than 2 times). It turns out that overdraw is easier for software developers to profile and reducing overdraw also yields significant performance benefits. So while the overdraw in our frames was high, relatively simple computation was getting overdrawn with much more complex computation.

The studies showcase the importance of holistically looking at the performance and energy models. Looking at just one alone might give misleading results, as it was evident from the scalarization and overdraw studies. In fact significant opportunities for energy reduction might exist where an in-efficiency in the design does not affect performance in any observable way.

Acknowledgement

As I complete the PhD program I have many people to be thankful for during my journey as a grad student. My time at Stanford has been full of learning and growing as a person, and there are a lot of people who helped me throughout the process.

When I started Stanford many years ago I was privileged to have professor Mark Horowitz as my advisor. I have learned so much from Mark, not only in my research area, but also in many other ways. I am not only grateful for all his guidance, support and mentorship over the past few years, but his extreme patience and trust in me during my years in graduate school. I could not have gotten a better advisor.

Next, I want to thank members for Mark's research group. Early in my PhD program I got to work on the Smart Memories project. I got to work with many people and learned a tremendous amount during this project. In particular: Alex Solomatnikov, Amin Firoozshahian, Ofer Shacham, Megan Wachs, Don Stark, and Stephen Richardson. This was probably my most memorable time at Stanford, and has forged many long-term friendships. In particular I want to thank Ofer Shacham, who has always been a great friend and is there whenever I need help with anything. I also want to thank Don Stark, who was also a great mentor to me during my time at Stanford. I am also very thankful for my other friends in Mark's group. In particular Pete Stevenson and Omid Azizi have been very good friends of mine.

There is also a number of support staff to which I am grateful. I need to thank Mark's administrative assistants for always be willing to help with setting up meetings, travel, etc. During the earlier years in Mark's group Teresa Lynn helped me setup my initial meetings with Mark, handled issues with funding, and countless other things. Mary Jane Swenson was the administrative assistant for Mark during

the latter half of my PhD. She has been absolutely helpful in setting up meetings with many busy calendars, getting my orals setup, and helping get my thesis signed. I must also thank Charlie Orgish for helping administer also the computer systems.

I am also very grateful to a number of friends I had outside of my research group. In particular: Oana Carja, Roxana Daneshjou, Siejen Yin-Stevenson and Sathish Jothikumar. They have always been there for me in times of need. I would also like to thank my frequent project partners at Stanford: Yanjing Li and Vishal Parikh.

I would also like to thank my co-advisor, John Montrym. During part of my PhD I was working at NVIDIA to learn more about GPUs and get access to real designs to help me with my research. John helped me get my research work setup at NVIDIA and provided me with lots of guidance and many interesting discussions throughout my time at NVIDIA. I cannot thank John enough for so much time with me to help me throughout my PhD.

I would also like to thank all my PhD committee members. Professor Subhasish Mitra was not only on my committee but was a mentor to me in my early days at Stanford. Professor Christos Kozyrakis was a collaborator throughout my PhD process and is in both my reading and orals committees. I am also very grateful for Professor Dawson Engler to take the time to chair my orals committee. There are many other people at NVIDIA who I wish to thank. My managers at NVIDIA: Ashish Karandikar, Narayan Kulshrestha and James Reilley. They were very helpful for me setting up my research and supporting me throughout the process. I would also like to thank Jonah Alben, who helped with the logistics of doing my research as well as some great insights. Furthermore, I need to thank the many people at NVIDIA who collaborated with me either directly or indirectly while I did my work there. In particular Colin Sprinkle, and Visu Subramanian helped me substantially throughout my work at NVIDIA. Daniel Finchelstein, also helped me a lot with getting instrumented data out of the GPU. I would also like to thank Bill Dally, Brucek Khailany, Stuart Oberman, John Edmondson, and Michael Fetterman for the many discussions we had about GPU energy. I would also like to thank other members (and friend) in my group at NVIDIA, in particular: Kaushal Gandhi, Miodrag Vujkovic, and Anish Muttreja.

While at Stanford, I was also lucky to meet my wife Chang Liu. She has made by life filled with happiness and has endlessly supported me through the process of writing my thesis. I truly love her for bringing out the best in me and for all the fun and adventures we have together.

None of this would have been possible without my Family. In the end, I truly owe everything to them. I thank my Mom (Sultana) and my Dad (Mohamed) for the endless love and support that they gave me. They raised me to always do the right thing, provided me with a good education and instilled a love for science and math. I still remember the time my dad helped me with building circuits and writing computer programs. Finally, I would like to thank my elder sister (Unbareen), who is always willing to do anything to help out.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Power constraints are quickly changing the way GPUs are designed. Historically, GPUs have achieved tremendous scaling by focusing on area efficient processing (Performance/mm^2). This focus has allowed GPUs to scale in both functionality and performance by orders of magnitude.

Since graphics is a highly parallel application, chips with massively replicated cores have been designed where the number of transistors available largely dictated the peak performance. As a result, the highest performance GPUs have historically been sized to be the largest die size possible (ie. the reticle limit of the fab). This maximized the performance at a given technology node. The continued push towards higher performance has also caused many high performance GPUs to be fabricated using half-node processes to enable larger transistor count, and hence higher performance parts.

However, the slowing of supply voltage scaling post 90nm has created a new hurdle to GPU performance. The energy density increases rapidly when core voltages remain nearly constant and feature sizes shrink. This means that if you have the same average activity/transistor the power consumption of the chip increases as transistor counts increase due to technology scaling. Over the past few years this has caused GPU performance to be more limited by Performance/Watt than Performance/mm^2. Due to physical cooling and other practical limitations high performance consumer GPUs have effectively hit the power wall. Now further performance scaling can only come

with a reduction in the energy consumed by each operations and not just area scaling offered by technology since Energy/Operation · Operations/s, which is power, must stay constant.

Over the past few years the power wall has caused GPU design to pick up hardware features such as clock gating, power gating and other improvements to the hardware design that have dramatically improved the Performance/Watt of GPUs. These hardware optimizations have allowed GPUs to continue to scale in performance. These optimizations are not sufficient to continue performance scaling in the near future since the benefits of these techniques have been mined out. Similar to CPUs, GPU designers will have start looking at architectural efficiency.

To study and improve architectural efficiency we need tools and methodologies to model the performance and energy consumption of a massively parallel but heterogeneous system like a GPU. Although GPUs contain processing cores, they also contain a large number of dedicated hardware units which are responsible for graphics acceleration. This dissertation explores how we can build tools to get both fast and early energy feedback, which can be used to inform architecture level design decision on future GPU designs.

In the first part of this dissertation we look at the breakdown in technology scaling which has caused GPUs to hit the power wall. Historical data will show that, over the past few years, GPU designers have responded by focusing on Performance/Watt improvements to their designs even though there have been no massive architectural changes. We also provide a basic overview of a contemporary high performance graphics architecture and one of the core parts of the graphics processor: the shader.

Chapter 3 explores the creation of an energy model for a GPU which can be used to do architecture level trade-offs. Furthermore, we also explore why existing energy models don't work well for modeling a GPU. Creating these models and analyzing the results requires that we gather statistics from existing graphics applications so we also explore techniques that we used to gather this information. In Chapter 4, we then apply this model to an existing GPU design and correlate its results against the existing designs. A walk through of the shader model, which is the core processing unit inside a GPU, shows how a simple energy model can be created with modest

effort, yet provide accurate energy estimates.

In Chapter 5, we explore how this model can be used to do architectural studies. This chapter first introduces the benchmark frames (from leading graphics benchmarks) that we use to do these studies. We first explore how much numerical precision we really need for graphics applications since eventually the data will be downsampled to the color depth of the display. The second study focuses on overdraw in graphics applications. Overdraw occurs when a given pixel value is overwritten by another value which implies that the original computation is wasted work. We see that modest improvements (20%) can be achieved by reducing overdraw, though this is a lot smaller than if you were to look at the overdraw counts alone. As the last study we look fine-grained redundancy within the shader processor. Since GPUs are designed to be massively parallel, the shader processor is designed to concurrently execute multiple threads in parallel across several datapaths. If multiple threads are computing the same data values then there is redundancy that we can exploit for energy savings. We show that this can yield significant energy savings of (20 - 50%).

Chapter 2

Background

This chapter provides the background necessary to understand the unique aspects of graphics applications and graphics processing units (GPUs). The first section describes how graphics applications are structured and introduces the application programming interfaces (APIs) such as DirectX and OpenGL. The next section discusses how these APIs are generally mapped to specially designed hardware. As part of this discussion we will talk about the shader processor, which is a core compute unit in a GPU. While the shader processor is important, not all tasks are run on the shader engines, some well structured tasks are abstracted into custom hardware which makes the overall machine much more efficient. Finally, we will discuss how GPUs have scaled and the impact of the power wall on current and future GPU designs.

2.1 Overview of Graphics Applications

Graphics applications take a representation of a 2-Dimensional (2-D) or 3-Dimensional (3-D) scene, which consists of a collection of objects and produces an image of the scene from a given viewpoint. Typical applications include computer aided design, movie production and computer gaming. This widespread usage has driven standardization between hardware and software; most graphics applications are now built using these standard graphics APIs. OpenGL and DirectX are the most common APIs used to write graphics applications.

While there are many methods to create and render a scene, typical graphics applications take a representation of objects defined using primitives such as triangles[1] and convert them into visible/colored regions on the screen. This process is traditionally broken up into three phases [44]. First, we take the primitives and determine their projection on the screen, along with computing orientation and vertex colors. Then we take this projection and rasterize the primitives. This process will determine which pixels are covered by the primitives and produces fragments (collection of pixels) which need to be colored. As the last step the individual pixel values in each fragment are computed. Pixel computation may consist of several sub-steps, which include texture mapping, determining visibility, and blending with other pixels and finally calculation the actual color value.

To understand this software pipeline for graphics we start by looking at a simplified graphics pipeline, as shown in Figure 2.1. The processing pipeline starts by taking vertex of the primitives and assembles/maps them based on the coordinate system[2]. As part of this mapping the graphics system will do a projection map (which will translate the 3-D coordinates to 2-D coordinates, since the final rendering surface is typically 2-D. After the primitives are mapped to screen coordinates we can perform rasterization, which is the process of taking the primitives and finding out which pixels are covered by the primitives and need to be colored. A simplified view of rasterization is shown in Figure 2.2.

After we determine which pixel values are covered we can then determine what the output color values need to be. There are several different ways to perform this process which is typically referred to as pixel shading. The simplest method is to do linear interpolation of the values supplied for the primitive vertices. More complex shading methods are typically employed by modern systems and will be discussed in

[1]Triangles are used because they are guaranteed to be planar and significantly simplify computation required to determine primitive intersection. It should also be noted that one exception to this is graphics systems which work based on ray tracing. However, contemporary GPUs don't natively do ray tracing and are out of the scope of this document.

[2]Graphics systems typically employ multiple different coordinate systems (typically the object, world, screen) to provide support for proper scaling across screen resolutions and also provide and easy way to move objects (as is common in most graphics applications). Objects are usually defined in a local coordinate system, which are mapped to world coordinates (typically the world ranges from -1 to 1 in all dimensions).

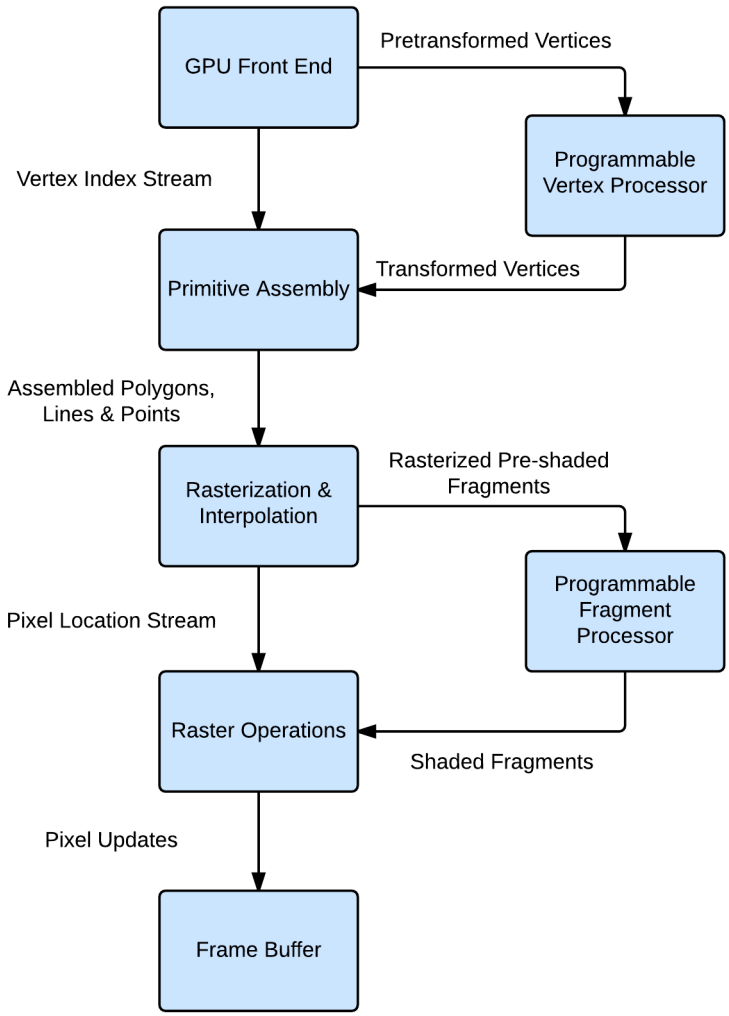


Figure 2.1: Simple graphics pipeline. A set of pre-transformed vertices are transformed by the vertex processor. These transformed vertices are then assembled into primitives and rasterized. The rasterization process produces fragments which are shaded by the fragment processor. Raster operations such as anti-aliasing and blending are performed before the resulting image is stored in the frame buffer

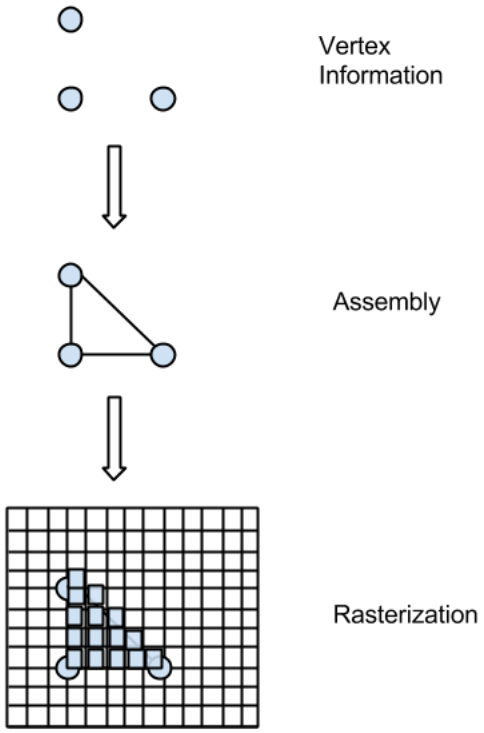


Figure 2.2: Rasterization process. The vertex values are assembled into primitives where are mapped to screen coordinates. Pixel values which are covered by the primitive are marked as being covered by the rasterization logic[65, 27]

the following sections.

The last stage of the graphics pipeline is to determine which color values should actually be output to the screen. Since we determine the color values of each of covered pixels without sorting the incoming primitives we can potentially calculate color values for regions, which are occluded by other primitives. These color values can be sorted based on their depth value to determine what the actual color value of each pixel should be. Conceptually we can think about representing each pixel on the screen with a 2-tuple, which consists of the color values and the depth value. We only overwrite the values if the depth value is closer from the viewer's perspective than the current depth value of a given pixel.

The above description provides a minimal conceptual view of the process to convert a given primitive to a pixel. Common graphics APIs such as DirectX and OpenGL typically employ a much more complex pipeline driven by the need for more realistic graphics and real-time performance requirements. The contemporary DirectX 11 (DX11) pipeline is shown in Figure 2.3. While conceptually this pipeline resembles that of the simplified pipeline and consists of similar stages (vertex processing, rasterization, pixel processing, output merging) each stage has a few more intermediate steps. In particular the vertex processing in DX11 is done using a vertex shader, which is programmable. This allows for complex mapping steps instead of simple pre-programmed projections. It also includes a tessellator stage, which can produce more primitives in the pipeline to generate even more realistic looking objects without having to transfer lots of detailed primitive locations to the graphics system. A small program that is executed by the shader processor instead generates the primitive values.

Conceptually the rasterization step is similar to the simplified pipeline description. The pixel shader stage is fully programmable allows the execution of fairly arbitrary code to generate color values of pixels. This means that the graphics system is no longer limited to shading strategies such as linear interpolation, Phong, etc [40]. Conceptually the output merge remains the same, however, since the DX11 specification supports anti-aliasing[3] the output merger is also generally responsible for merging

[3]Anti-aliasing is done to reduce artifacts which occur due to sampling very high resolution data

sample values[4].

Since graphics workloads are typically very numerically expensive they generally perform poorly on general-purpose processors. Also, the availability of popular APIs and a well structure workload leads itself well to custom designed hardware. In the next section we will take a look at the architecture of a graphics processor and how the software pipeline maps to a custom implementation.

2.2 GPU Pipeline and Architecture

A contemporary GPU architecture (resembling NVIDIA's Kepler Architecture[13]) is shown in Figure 2.4. To understand the architecture of a GPU and how it performs the necessary work we can take a look at how different pieces of hardware map to the various stages in the DirectX 11 pipeline, shown in Figure 2.3.

Looking at the DirectX 11 pipeline we can see that it consists of several shader stages, which in current GPUs, are all handled in a unified programmable shader unit [39, 13, 69, 64]. The shader unit is the core part of the GPU which is responsible for executing shader programs that work with either pixel or vertex data. The shader unit internally contains a processor optimized for graphics workloads along with some dedicated logic which helps it achieve high efficiency. Since the shader unit is a large and important part of a graphics processor it will be discussed in more detail in the following section.

The rasterization is performed by a series of fixed function units outside of the shader unit. The series of operations performed are coarse raster, fine raster and ZCull. The coarse raster unit will first coarsely determine the outline based on the vertex locations of the primitives and produces tiles for the visible regions. These tiles are a small block (for example 16 x 16) of pixels (or sub-pixel samples if anti-aliasing

to the finite resolution of the screen. Many methods exists to do anti-aliasing but a popular method is to essentially render the sub-pixels (refereed to as samples) and merge the sub-pixels into final pixel values

[4]Since the pixel shader is programmable we can in theory merge samples together using the shader and bypass the output merger. This is the basis for FXAA (Fast Approximate Anti-aliasing) and will be discussed in Chapter 5, since FXAA is used by some of the applications we use for benchmarking

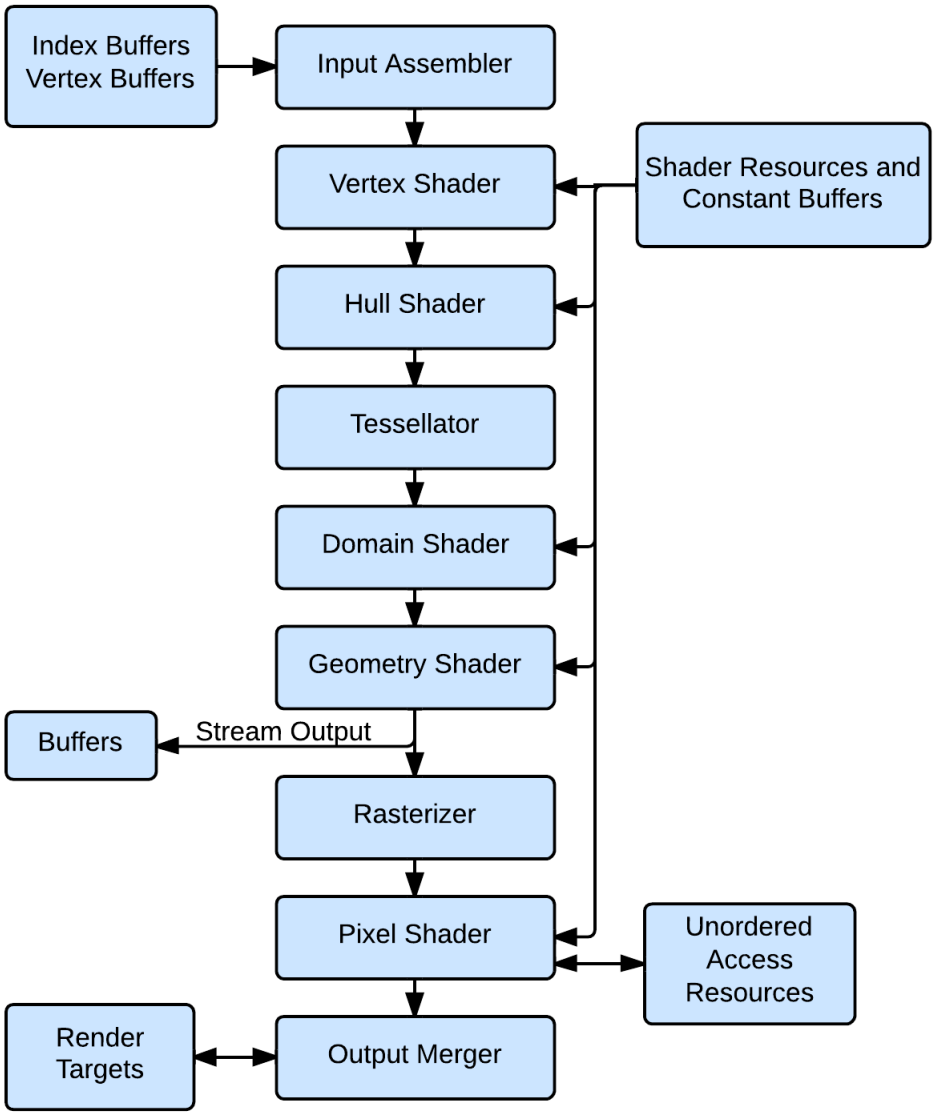


Figure 2.3: DirectX 11 (DX11) Software Pipeline ([5, 64, 58]). The DX11 is a complex pipeline consisting of many programmable shader stages. The vertex, hull, domain and geometry shaders execute a graphics program on pre-rasterized results. The pixel shader determines the pixel color values based on executing custom software. The output merger is fixed function hardware, which is responsible for taking either multiple samples or semi-transparent pixels and merging them into a final output image.

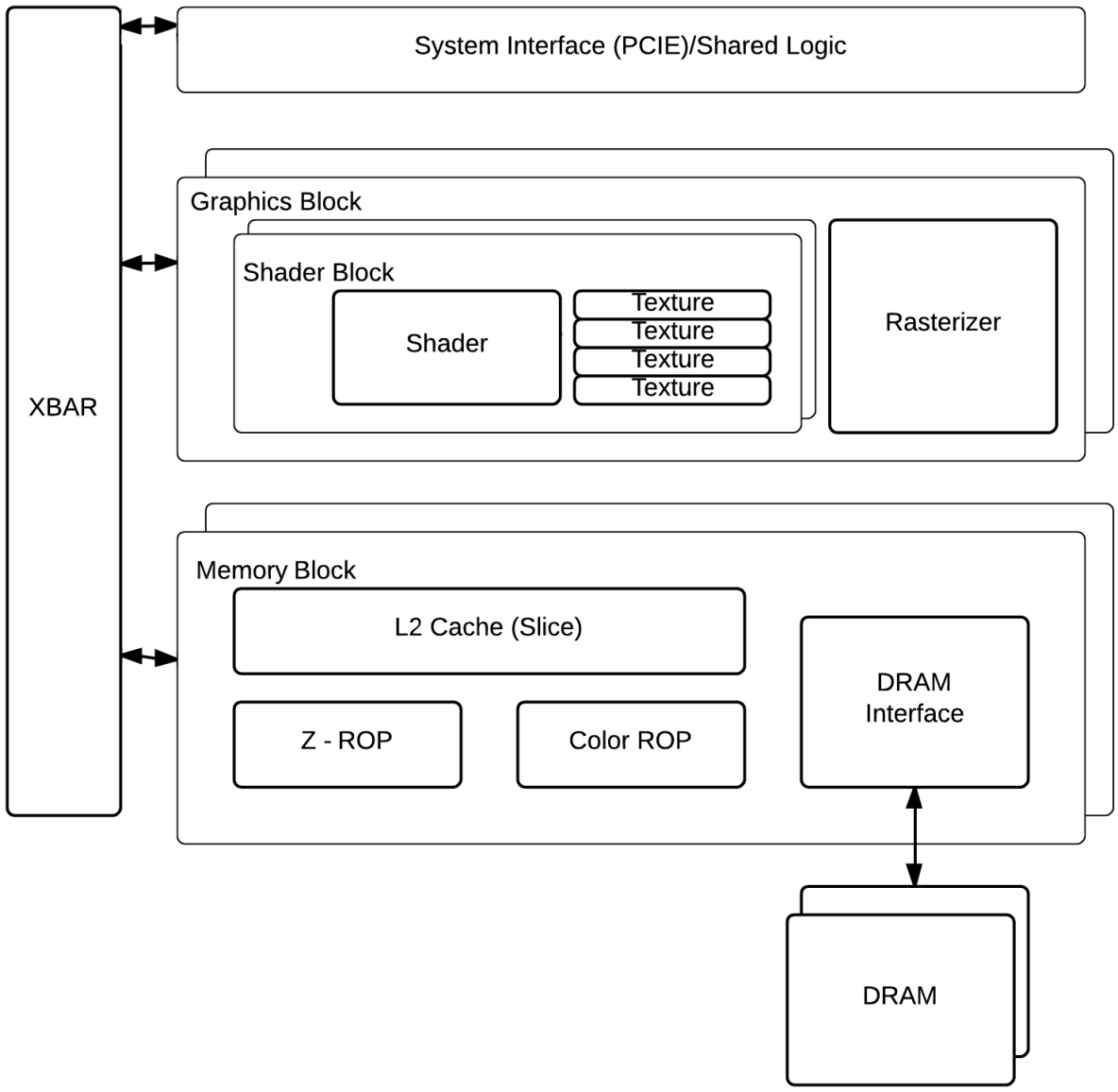


Figure 2.4: Contemporary GPU Architecture. The GPU is a complex system consisting of many different functional units, which are generally replicated. The design shown above resembles NVIDIA's GPU architecture and consists of graphics blocks, shader blocks and memory blocks which are interconnected using a crossbar ([13, 69]). GPUs have lots of replication, for example there are many graphics blocks, which internally consist of many shader blocks and supporting logic such as rasterizers. There are also many memory blocks that provide parallel and high performance access to on chip caches and external memory.

is enabled). The tiles are then filled in by the fine-raster logic. The ZCull logic will then buffer up these tiles and try to merge tiles together to reduce work downstream [38]. Since the ZCull unit has limited storage, its purpose is to eliminate work that is obviously redundant. Since ZCull has some internal storage it can buffer up primitive to remove obviously occluded cases. It can also remove primitives which are facing away from the camera viewpoint, which are referred to as back-facing primitives.

The raster operations pipeline (ROP, output merger in DX11 API) is fixed function logic that is further broken down into two ROP (Raster Operations) sub-units the Color-ROP and the Z-ROP. The Color-ROP unit is responsible for computing the final output values of each pixel. For example it may merge multiple samples to perform anti-aliasing or perform blend operations on fragments, which are semi-transparent. The Z-ROP unit does depth tests to show only fragments, which are not occluded by other fragments. In other words its responsible for sorting the depth for each of the pixels. Raster operations require a lot of memory bandwidth since they operate on actual pixel values and might need to access a given pixel multiple times. Due to the high memory bandwidth requirements the ROP unit is co-located with the L2 cache and memory controller which helps localize the high memory traffic.

One thing to note is that GPUs can perform early-Z which means that they perform the Z-test on pixels before shading them. This reduces the shader workload assuming that we draw all the primitives from front to back in depth order. The draw order is important since the GPU does not actually buffer up primitives but simply performs the depth tests after rasterization against all the primitives which have already been seen.

To summarize we will start with a primitive triangle and see how it is processed by the GPU pipeline. The primitive will enter the various shader stages (performed by the Shader) which will perform vertex level manipulation of the object writing intermediate results to memory. After the primitive is mapped to screen coordinates the rasterizer will then convert these to fragments that can be colored. If early-Z is enabled (typical use case), the fragments will flow through the XBAR into the Z-ROP, which will perform the Z-test; if it passes the Z-test the primitive will re-enter the Shader to perform pixel shading. After pixel shading the fragments will go

through the XBAR into the Color-ROP that will perform operations such as blending or merging of multiple fragment samples (in the case of anti-aliasing) and produce the final pixel values. These values are finally written to the frame-buffer, which is a dedicated section of memory located on the DRAM.

Through this entire process each shader program was executed independently on each primitive or a fragment[5]. This implies that there is a lot of available parallelism. Graphics hardware exploits this parallelism to improve performance.

2.3 Parallelism and the Shader Design

The key to understanding GPU architecture is to realize that a small set of programs are executed across tens of thousands, if not more, primitives and even more fragments. Since computations across both primitives and fragments are independent of each other they can be computed in parallel. Thus the number of resources available to execute these programs and not data dependencies limits the performance. Overall GPU designs are bound by the number of functional units, the balance of different fix function units, and getting the data into the functional units.

As a result, GPUs are designed to have highly replicated functional units. These replications are shown in Figure 2.4, which shows a GPU similar to NVIDIA's Kepler [13]. A chip can contain multiple graphics blocks and multiple memory blocks. A graphics block can contain multiple shader blocks, etc. The number of each type of functional unit is determined by performance constraints as well as expected use cases.

On graphics processing units (GPU) the parallelism is exposed as operations on individual primitives and pixel values. This explicit parallelism allows GPUs to focus hardware resources on compute area. In contrast, a CPU is dominated by hardware

[5]Programs running on pixel shaders are very restrictive since they only have values of the fragment they are executing. Any effect that requires access to a region larger than the neighboring pixel quad (2 x 2 pixels) requires a multi-pass solution. The first pass will render the values to the buffer and the second pass will then map this buffer to a texture. The shader program can then access the texture to get the values of neighboring fragments

needed to enable out-of-order execution to extract implicit instruction level parallelism [44, 69]. As discussed earlier, the GPU architecture also contains fixed function hardware for performance and energy efficiency.

The shader, which ends up being one of the most replicated blocks in the system, is responsible for performing all the shader stages of the graphics API as well as any general purpose computation[39][6]. The high level architecture of a shader processor similar to NVIDIA SMX[31] is shown in Figure 2.5. Internally a shader is organized as yet another replication of building blocks. Each shader consists of four replicas of one "quadrant" which contains a large datapath. Each quadrant of the shader consists of a datapath, which can process 32 floating points operations in parallel, along with a dedicated register file. To supply the needed data, these quadrants share the load store unit, along with special function units (for example, log, exp, sine) that have lower throughput requirements. While these shared datapaths are also 32 wide internally they are shared with the 4 quadrants and hence, on average, have a quarter of the throughput.

The programming abstraction for the shader is multiple parallel SIMD shader processors. Each of the SIMD lanes in the processor is abstracted as a thread (instead of a vector as is done in conventional CPUs). Groups of 32-threads are referred to as warps or thread blocks. This greatly simplifies the programming model since we can effectively write functional programs which operate on pixel values and in most cases these can be grouped together and performed in parallel across the SIMD lanes since all the threads are executing the same shader program at any given time. However, since shader programs can contain branches this can lead to divergence in the program execution flow. The divergence is handled using one of two techniques. The first option is predicate execution: code from both sides of the branch is issued and execution of these instructions depends on the branch predicate. The other method is replay, where we execute the branch code but predicate based on the branch condition. After the first side of the branch has finished executing we replay the other side of the branch until all the divergence is canceled. The former is generally used

[6]Since graphics processors support programmable shading we can exploit this for general purpose computation as well. This is commonly referred to as General Purpose Graphics Processing Unit (GPGPU)

for small branches while the latter is used for much larger (or less common) branches. The decision to insert predication vs replay is left to the compiler.

The shader processors also use abundant parallelism to hide the effects of latency on program performance. For example, the shader itself has no branch prediction and also has minimal forwarding logic to deal with data path and register file latencies. Instead, it has a very large register file (on the order of 32 - 64 KB), which contains the state of many active threads with a set of registers used for each thread. When an instruction with high latency is executed, the shader simply moves to another thread, which is ready to be executed. Furthermore, the latency of most operations is known at compile time and can be statically scheduled by the compiler, removing overhead for complex logic needed to track thread states. For instructions with variable latency (for example, memory or texture operations), a barrier instruction is used to block thread execution.

The shader also contains its own internal L1 cache, which can also be used as shared memory (software managed cache). The shared memory is an important aspect of the shader as it allows local scratch memory to be software managed and allows for efficient implementations of certain operations such as transpose which are expensive with traditional memory systems. The caches and shared memory are interconnected via a XBAR to the load/store units.

Unlike conventional high performance CPU designs, the majority of the shader code can be software scheduled by the compiler by inserting hints into compiled shader program. These static hints are always the first instruction in each instruction cache line and have scheduling hints for all the other instructions in the cache line. The hints contain information about both latency and pairing. For example, the hint can say the instruction requires 4 cycles for the results to be available. Another common hint used is pairing, which informs the shader that the two instructions referred to can be issued together. This method can be used to efficiently pipeline a multi-issue processor with datapath latencies without incurring hardware necessary to schedule these instructions. Since single threaded performance is not very important dynamic run-time optimization is not necessary.

While not immediately apparent even the shader contains a lot of fixed function

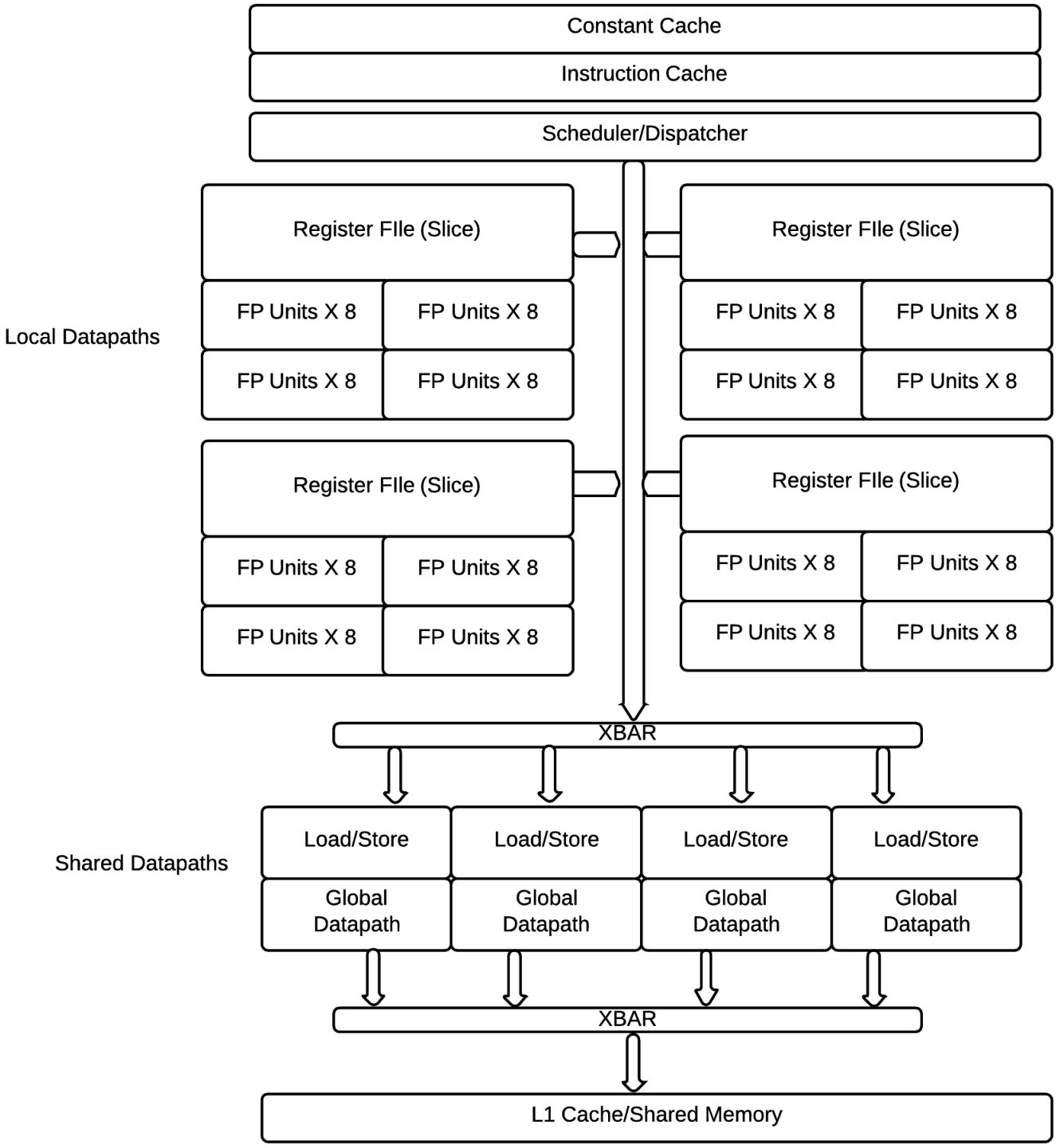


Figure 2.5: Shader Architecture (SMX Style). This shader resembles the shader unit inside of NVIDIA Kepler GPU. It consists of four sets of function units with dedicated register files, which are interconnected using a crossbar. There is a set of shared functional units, which are responsible for operations such as load/store and less common math operations

logic not usually found on general purpose CPUs. For example, a common graphics operation is attribute interpolation based on vertex values. The shader contains special instructions to perform lower precision floating operations to do attribute interpolation for primitives based on coordinates[7]. In addition a cache is used to hold these primitive values since many attribute calculations are usually performed hence this operation has high locality. The shader also contains dedicated logic to do transcendental operations, which is not common on most CPU implementations.

There is also fixed function logic to perform texture filtering which is handled by the texture unit and is closely coupled to the shader. Texture filtering, which is the process of sampling a texture at different resolutions, can be very efficiently performed using fixed function hardware [56, 44]. While the texture unit itself is not part of the shader processor it is tightly coupled by the shader processor. In fact, most of the memory accesses done by shader programs are texture operations performed by the texture unit.

2.4 GPU Scaling and the Power Problem

Over the past decade GPUs have had massive scaling both in terms of absolute performance (as shown in Figure 2.6) and in terms of performance/mm. This scaling occurred both due to architectural improvements and process scaling due to Moore's law. Although an empirical observation, Moore's law has held true over the past five decades as the number of transistors per chip has continued to exponentially increase. While Moore forecasted the scaling of transistor densities, Dennard[16] in 1974 showed how MOS transistors would change with scaling.

Dennard's scaling theory showed how all chips can achieve a triple benefit from scaling. First, device sizes scale by α in both the x and y dimensions, allowing for $\frac{1}{\alpha^2}$ more transistors in the same area. Second, capacitance is scaled down by α (because $C \propto \frac{L \cdot W}{t_{ox}}$, where C is the capacitance, L and W are the channel length and width, respectively, and t_{ox} is the gate oxide thickness[8]. But, if we assume that the electric

[7]Since the attribute interpolation is somewhat of a preprocessing step for attributes, some GPU architecture perform this step in dedicated logic outside of the shader processor

[8]We should note that wire capacitance (and resistance) is relevant to both the energy and delay.

field V/L is constant, implying that V scales by a factor of α, the charge Q that must be removed to change a particular node's states scales down by α^2 (because $Q = CV$). The current is also scaled down by α, so gate delay D which is Q/i also decreases by α. Finally, because energy is equal to CV^2, energy decreased by α^3. Thus, following constant field scaling, each generation supplied more gates per mm^2, gate delay decreased, and energy per gate switch decreased. Most important, following Dennard scaling maintained constant power density: logic area scaled down by α^2, but so did power: energy per transition scaled down by α^3, but frequency scaled up by $\frac{1}{\alpha}$, resulting in an α^2 decrease in power per gate. In other words, with the same power and area budgets, $\frac{1}{\alpha^3}$ more gate switches per second were possible. Thus, scaling alone was able to bring about significant growth in computing performance at constant power profiles.

However, if we observe GPU power density data as shown in Figure 2.7, we can clearly see that the power density is increasing. The reason this happened is two fold. First, GPU designers scaled up GPU performance faster than pure technology scaling allows and second, V_{DD} did not scale linearly with feature size. Recently the VDD problem has gotten worse: it has nearly stopped scaling. This happened because V_{th} scaling drastically slowed down due to transistor leakage. In order to maintain performance V_{DD} had to be maintained constant in relation to the threshold voltage. With constant voltages, energy now scales as α rather than α^3, and as we continue to put $\frac{1}{\alpha^2}$ more transistors on a die, we are facing potentially dramatic increases in power densities unless we decrease the average number of gate switches per second. Although decreasing frequencies would accomplish this goal, it isn't necessarily a good solution, because it sacrifices performance.

The need for higher energy efficiency is urgent. From Figure 2.8, it is apparent that high end consumer GPUs have hit a power wall at around 250W. It turns out that past 250W the cost of cooling the GPU quickly becomes infeasible. The result, as Figure 2.9 shows, that the power efficiency of GPUs has been drastically improving after 2011 (with a slight slowdown in 2010/2011)[9].

As technology scales and more complex designs are built the relative wire lengths tend to increase which makes the wire capacitance an increasing fraction of the gate capacitance

[9]Interestingly, this slow down in scaling happened when GPUs hit the power wall after which we

Initial energy efficiency improvements in GPUs came using the same techniques as those pioneered by the CPU community. For example, both clock and power gating were introduced into GPUs during this time frame. Other improvements have included better datapath architectures along with moving to unified shaders (which appear as increased FMA throughput since some of the fixed function logic was moved into programmed units).

While historically GPU architects have focused on improving area efficiency, it is clear that current designer need to focus on improving energy efficiency if GPU performance is to continue scaling. Energy efficiency improvements can come in the form of better hardware designs (improved clock gating, dynamic voltage control, etc), architecture and algorithmic improvements. To aid this process, this thesis focuses on the development of tools and methodologies to improve architectural and algorithmic energy efficiency.

2.5 Energy Consumption and Modeling

Since improving energy efficiency is required to improve performance we need to understand how energy is consumed in the design and how it can be modeled. All GPU designs today are built using CMOS technology. Fundamentally, the energy in CMOS circuits can be broken down into three-categories: Dynamic, Short-circuit and Leakage:

$$P_{Total} = P_{Dyanamic} + P_{Short-Circuit} + P_{Leakage} \tag{2.1}$$

Dynamic power is related to a the activity of the device as given by the following equation:

$$P = \frac{\alpha f C V^2}{2} \tag{2.2}$$

Where α is the activity factor (the average number of times the capacitance changes state each cycle), f is the frequency of the design, C is the capacitance

see significant improvements in energy efficiency

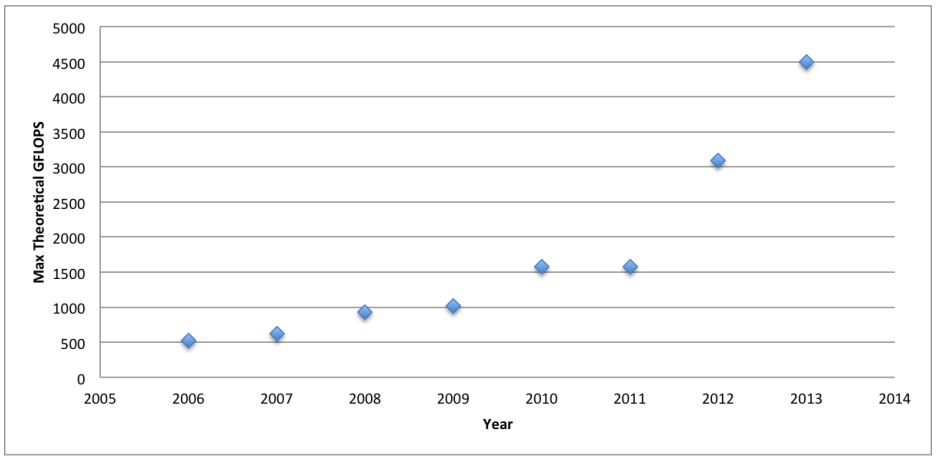


Figure 2.6: Performance scaling as shown by scaling of GFLOPS over time

and V is the operating voltage. Short-circuit power is consumed whenever individual cells switch and the PMOS and NMOS transistors are both temporarily on. This fraction of energy is usually small compared to the dynamic power and is typically in the 10% range, though it can sometime be much higher[51]. Leakage power is caused by current flow through nominally off devices and is related to transistor characteristics. It is a strong function of voltage, temperature and threshold voltage.

Using high threshold voltage transistors or power gating, a technique that turns off the power to unused blocks in the system usually reduces leakage power in designs. Decreasing the Vdd, the capacitance of the nodes in the system, the activity for a given operation or all three reduces dynamic power. Reductions can be accomplished by hardware techniques such as clock gating, or architecture/algorithmic optimizations, which reduce the amount of work necessary to produce a given result.

Since it is generally prohibitive to create a large number of complete designs and evaluate them individually, simulations and modeling tools are used to evaluate different early stage designs. Power models attempt to predict the power of these designs, usually based on activity of higher-level events.

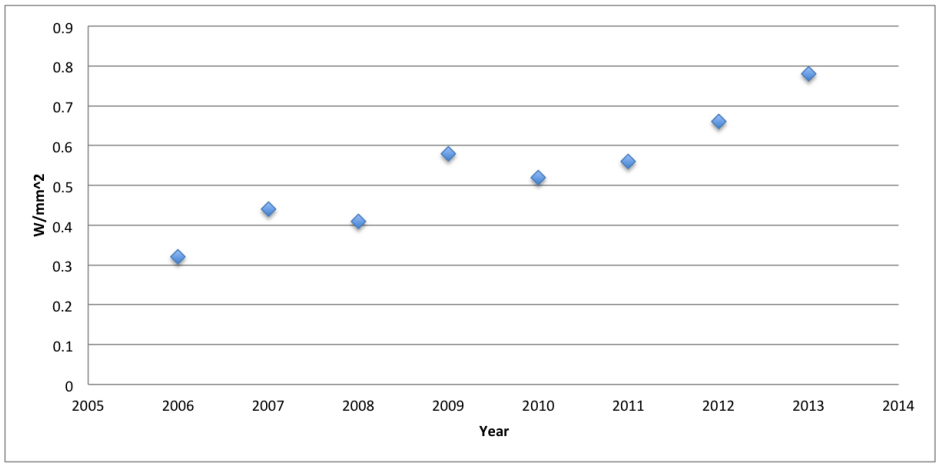


Figure 2.7: Power density of GPUs since 2006 in $Watts/mm^2$

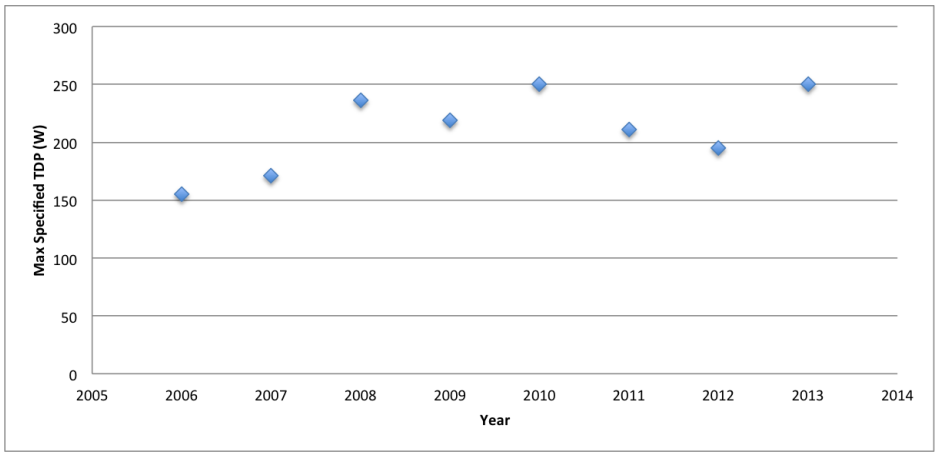


Figure 2.8: Scaling of max power (specification) of GPUs since 2006 by year of release

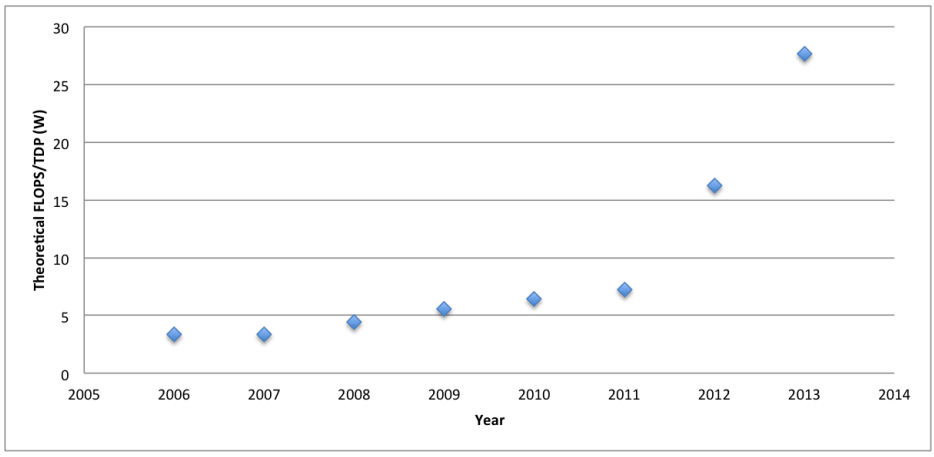


Figure 2.9: Power efficiency as defined by max GFLOPS/TDP (Thermal Design Point) since 2006

2.6 Energy Modeling Tools

Given that CPU designs hit the power wall much earlier than GPUs (around 2001), as shown in Figure 2.10, we first look whether their tools can be directly applied to GPUs. There have been many CPU oriented power modeling tools developed over the last decade. These tools can be divided into two major classes: simulation based models which use either actual designs or representations of the design to calculate power estimates and regression/statistical models which are based on correlating some events with power and using that as a proxy for power. These models are usually derived from silicon measurements.

The most straight forward method to estimate power for a given design is to do either SPICE simulations or use digital power analysis tools such as Primetime-PX [72]. While these approaches offer high accuracy, the results are usually only available after the design is mostly complete making it difficult to make meaningful architecture or algorithmic changes.

In order to get early architecture feedback there have been several different energy modeling frameworks proposed for CPU's. Cacti[62], which was one of the first energy modeling tools proposed has become the de-facto standard for creating energy models for caches in both academia and industry. Wattch[6] was one of the first frameworks

that proposed using high level architecture building blocks for energy/power estimations instead of lower level circuit and RTL based tools. In Wattch, activity from architecture simulations tools was applied to building block based models in order to arrive at power estimates. Wattch proposed the following classes of building blocks:

- Arrays: RAMs, Caches, registers
- CAMs: Content Addressable Memories
- Combination Logic: All the functional units, including wires
- Clocking: All gates associated with the clock distribution network

In Wattch, these models are based on measurement of actual designs along with theoretical calculations based on size and capacitance (for example, Array power can be determined using a mathematical model). McPAT[37] augmented these energy models with both area and timing information. McPAT also improved the underlying modeling by adding both leakage and short-circuit modeling along with making the models hierarchical. The hierarchical model can be easily configured to target different multi-processor based systems in order to rapidly create new architectural models.

Statistical modeling was is one of the older techniques to model power consumption of CPUs. The models are created by running either application or directed test on silicon and measuring power. This measured power is then correlated (using linear regression or other machine learning techniques) to either the instruction trace to obtain an instruction level power model[63, 55, 53] or performance signals to create a model based on performance counts.

Directly applying these CPU tools to GPUs is difficult. While GPUs are multi-processor systems with some resemblance to modern multi-processor CPUs they have some characteristics which make them very different than contemporary CPU designs. GPUs solve some very well defined and structured problems (ie. Texture Mapping, Anti-Aliasing, Rasterization, etc) for which they have many fixed function units. Even though GPUs have been moving towards a more general-purpose model (mostly to

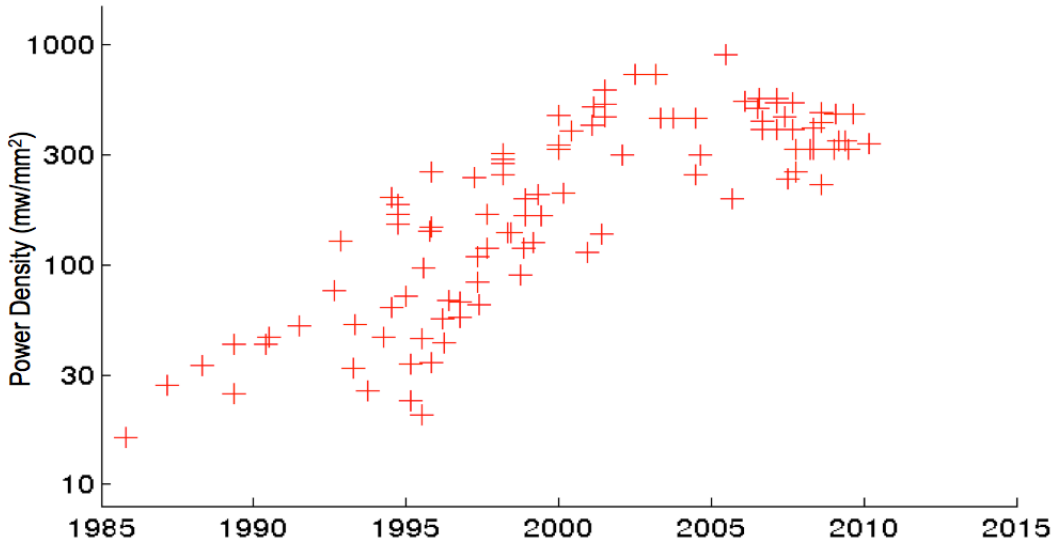


Figure 2.10: Historical TDP CPU power density showing the power wall. A power density of around $1W/mm^2$ is the upper limit.

push into HPC), they still maintain dedicated logic that helps them maintain high performance. Also, internal data structure (for example register files, caches, shared memory) have very different structures compared to CPU designs.

Most of the prior work on GPU architecture power modeling has focused on using a statistical modeling approach. This likely resulted from the fact that there was no open implementation of a GPU, and little published work on the inner workings of a GPU. One such model by Ma. et. al.[41], used a set of 5 performance signals obtained from an NVIDIA GPU to predict the power consumption of the device. The training in this case was done using a support vector machine against silicon power measurements. Several other GPU power models rely on related statistical modeling approaches to estimate power of GPUs [28, 73]. With the exception of [41] the rest of the GPU related power modeling work focuses on GPGPU workloads.

While statistical models are good for creating models for existing GPUs or GPU design with incremental changes, it hard to use such models for studying big architectural or algorithmic changes. Yet building complete bottom up model for a GPU would be an overwhelming exercise due to the sheer number of dedicated units. The

resulting model would also be less flexible and less useful for studying architectural trade-offs. To overcome difficulties in modeling a complex architecture that consists of many-fixed function operations we created a hybrid methodology for modeling GPUs.

Since we started this work in 2010, there have been other GPU models that have been crated. One model that is similar to this work and Wattch is GPUWattch[36]. GPUWattch is a model that predicts energy consumption of NVIDIA GPUs while executing GPGPU workloads. GPUWattch is driven by a custom CUDA simulator that predicts both the performance and energy consumption. Both energy and performance estimates are validated using actual hardware. While both this work and GPUWattch use a methodology similar to Wattch our model is constructed from access to detailed implementations of the design. This allows us to not only model very coarse functional units but also get very detailed and high fidelity energy estimates. Our model also consists of many validated variations of GPU components, which allows for rapid construction of different variants so that we can quickly perform architectural "what if" studies. We will provide a much more detailed comparison between our model and GPUWattch in Chapter 4.

The following chapter discusses both this methodology and other tools developed to analyze GPU architecture and energy efficiency.

Chapter 3

Methodology

In this chapter we take a look at how to model energy consumption of a given design for a given workload. This understanding is critical to both reducing the energy consumption as well as modeling actual energy improvements. We start by creating a very detailed model of the design. While accurate, this high fidelity model is difficult to modify making it not useful for our objective. The rest of the chapter explores the construction of an alternative model and methodologies to extract stimulus, which is critical for this model to function.

3.1 Simulating power for existing designs

Since we had access to several existing GPU designs we could simply measure power/energy consumption by simulating the designs. We used a single methodology to run power simulations regardless of the design size (from a single datapath to the entire shader unit)[1], which allowed us to get consistent data. Figure 3.1 illustrates the method that we used.

To begin the simulation process we first obtain the Verilog code for the design. In some cases the Verilog code was generated by a pre-processor, which allowed us to obtain different versions of a single design. We also needed test vectors, which in our case can either be a directed test or given benchmark that we want to execute.

[1]For larger designs, we had to use a divide and conquer approach.

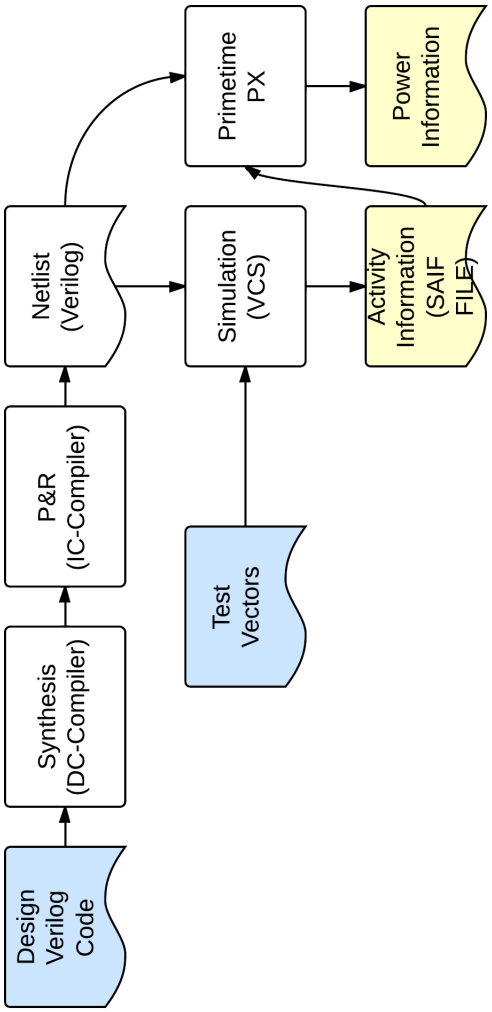


Figure 3.1: Getting energy information for a Verilog design. A simulation-based methodology was used where we generated activity information from simulation and then used Primetime to generate power/energy values

In most cases we were able to re-use the functional test environments and used test data that is more relevant for energy/power studies. The Verilog for the design is synthesized using Synopsys Design Compiler[4] and place and routed (P&R) using Synopsys IC Compiler[2,3]. The P&R step allows us to get accurate estimation of energy with wire loads. We should also note that P&R step allows us to get accurate timing information that we can use to determine the operating frequency of the design. This is useful for obtaining different design points, which is done in Chapter 4.

We then performed a Verilog simulation of this design using Synopsys VCS [60] using the post P&R netlist. The output of this process was a Switching Activity Interchange Format (SAIF) file which contains information about the toggle counts, the state of control signals during toggles and the exercised paths of individual cells[4]. The resulting SAIF file and the post P&R netlist was used to estimate power using Synopsys Primetime PX [72].

This methodology allows us to collect power information of existing designs with very high accuracy. However a limitation of this technique is that it requires the actual hardware to be designed at the Verilog level. This implies that quick exploration of the design space is not easy/possible. Instead we need to have a better way to build up a model that can be both flexible yet accurate while varying design parameters.

Energy consumption is fundamentally related to the amount of capacitance switched so it can be modeled by activity of the design and a constant, which is the energy, consumed every time that region of the design is activated. Borrowing techniques from Wattch/McPAT we can model the energy as product of block activations and building block energy cost per activation. The next section explores how we can build up such a model and what implications such a model has on accuracy.

[2]We only performed global route to get parasitic estimation. In some cases, when we had good correlation, we used back-propagated wire loads models from existing designs or DC-Topographical instead of IC-Compiler.

[3]For designs with a lot of memory elements, hints were provided to the place and route tool to obtain good placement

[4]This is known as State Dependent Path Dependent (SDPD), power estimation. The methodology yields more accurate results b/c it captures the state along with toggle. For examples, the number of times the clock toggled when the write enable of a given ram was enabled.

3.2 Our Power/Energy Modeling Approach

We used a combination of existing techniques to create an energy model for a commercially available GPU. Creating an energy model for the GPU requires modeling both the shader processor (SM) and also other fixed function logic. Modeling the fixed function logic is critical, since for a high end GPU about 40% of the GPU area is not in the cores/caches. This percentage is larger in smaller GPUs since they need to have at least one copy of all the fixed function logic.

We created a model that was a compromise between a pure statistical model and a building block level model such as McPAT. This model can be rapidly constructed but also be used for fairly detailed architectural studies as we show in Chapters 4 and 5. Our model is based on the premise that performance monitor (PM) counts can be used to predict the energy usage of the various GPU components (for example, Texture, SM, Raster, Primitive Engine). PM counts are metrics that are produced by either the simulation or the design itself that counts how many times a given operation was on a given resource in the design. Given that energy is consumed when a resource is used, we expressed the energy consumption of a design as follows:

$$Energy = \sum_{i=0}^{n} PM_i \cdot E_{PM_i} \tag{3.1}$$

Where PM_i represents the ith PM count and E_{PM_I} is the energy associated with this particular PM count. The implicit assumption here is that the energy is related linearly to the performance counts. We further assumed that the energy associated with a particular PM is always constant. This implies that factors such as data dependent power cannot be taken into account using this model[5].

The fastest and easiest way to obtain a model for an existing design is to simply fit the above model to power simulation results from existing designs as shown in Section 3.1. We used a simple linear regression model to fit the data while minimizing the error function, which is given by:

[5]Using pathological data where, for example, the data is always 0 or always switching will produce different power data. In all our experiments we used data that was expected from real-world applications. This data independence hasn't caused any large errors, perhaps because of the low correlation in the low order bits of that data

$$Err = \sum_{test=0}^{n} \left(Energy_{Actual} - Energy_{Estimated}\right)^2 \tag{3.2}$$

We used a solver called CVXOPT [14], to perform this optimization. By using this approach we came up with a model that was constructed using hundreds of small tests that can be used to predict power of various applications (We will see some of the correlation results in Chapter 4), which would not be feasible to run using RTL or gate simulations. At this point we have a regression-based model. However, the issue with this model is that it is hard to do "what if" studies since these models are fundamentally rigid. Since the model is based on a regression, the energy associated with each PM may not represent the actual physical design if the PM counts are not linearly independent. Furthermore, it may be hard to scale the values for a given PM unless the initial PM signals were carefully selected (for example data path active).

While the detailed Verilog level model was accurate, but hard to construct and not flexible, this model is accurate, easy to construct but still not flexible. In order to improve the flexibility of the model we decided to model some regions and functional units in the design using flexible building blocks. These flexible building blocks were actual functional units such as the floating-point unit, which were simulated for many different parameters and substituted into the model instead of doing a regression fit. Since these models come from synthesis and not regression, we have high confidence that they track parametric energy costs. This allowed us to create a model, which is both, relatively easy to create as well as flexible. Composing the model out of regression based models as well as building blocks allow us to do architecture exploration. Based on the level of exploration we want to do, we can change the level of detail for the energy model.

Constructing this model requires us to separate out the regions that are regression based and parts that we are going to create using building blocks. In general we opted to have control logic be regression based while using building blocks for memories and datapaths. We can also pick which regions are building blocks based on the study, which needs to be conducted. To understand how to create a basic model we look at the simple design shown in Figure 3.2. In this design there are 4 PM counts:

Active, Instruction Fetch, Register File Accessed and Datapath Active. The design consists of three major building blocks: a datapath, a register file and miscellaneous control logic. If we are interested in doing a study related to the organization of the datapath and register file but not in exploring changes to control logic, we will simply associate the energy of the datapath and the register file to their respective PMs and use the linear regression shown above in order to solve for the remaining energy. This allows for the energy associated with some of the PM signals to now become a simple function of the building block actually used instead of a fixed number determined by linear regression. Similarly to Wattch and McPAT we have several different kinds of building blocks:

- Datapaths and larger functional units
- RAM's
- Clock Distribution/Gating
- XBAR's
- Wires/Interconnects

Unlike Wattch [6], we associate local clock distribution and flip-flop overhead in the building blocks themselves. This means that there are several different datapath models targeting different frequencies and pipeline depth that can be selected depending on the design being studied. We found that this leads to better correlation when aggressive re-timing is used in the design. We also did not create models for large caches and CAM's since they are not used in contemporary GPU architectures. Furthermore, we found that Cacti is a poor proxy for predicting the power consumption of GPU caches which is most likely due to the large amount of fixed function logic present in a GPU's cache (for compression, decompression, format conversion, etc). Instead we opted to use a custom energy model for modeling the GPU cache.

Using the above modeling approach we were able to quickly develop models for the GPU while only focusing on having details on an as-needed basis given a particular

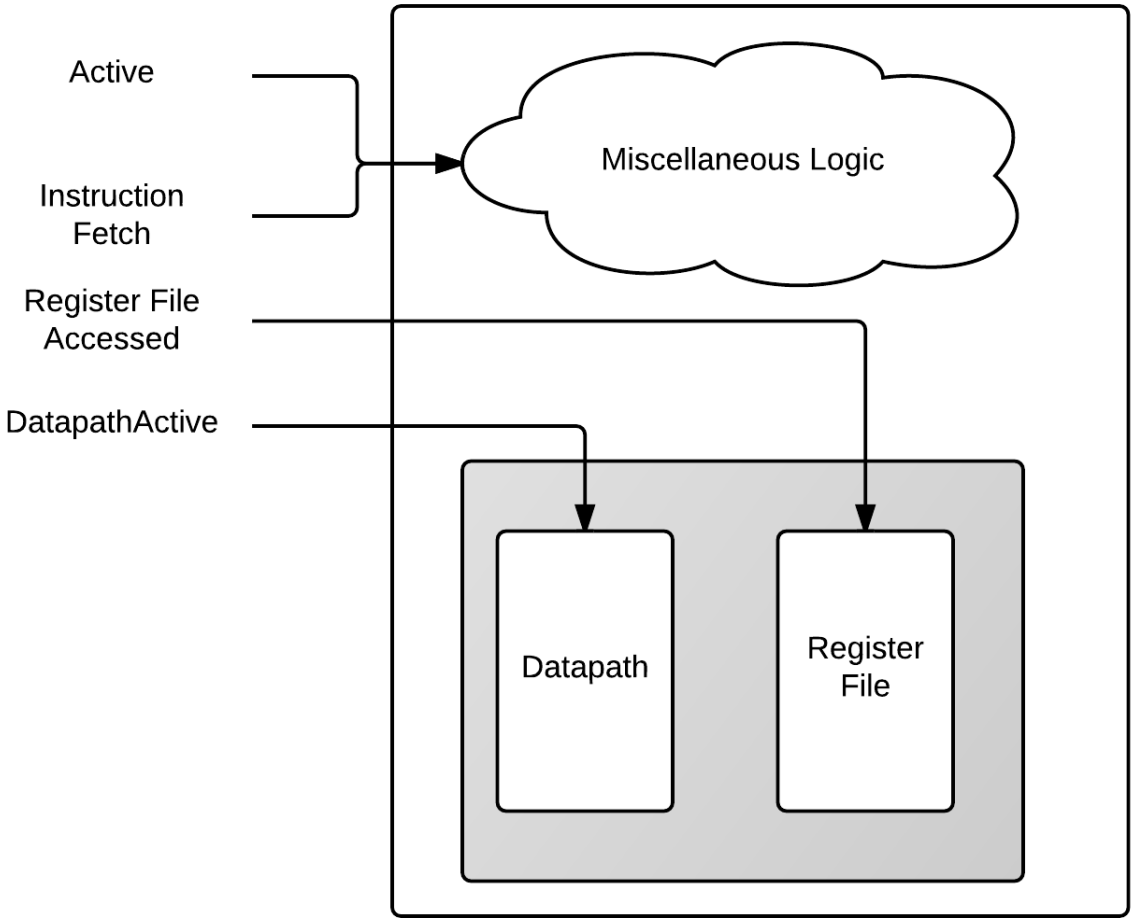


Figure 3.2: High level representation of energy modeling approach

study that needs to be performed. The actual models and correlation results are shown in more detail in Chapter 4.

The energy per event of the building blocks was obtained by simply simulating existing designs using the flow shown in Section 3.1. This direct simulation approach gives a high fidelity model we use to calibrate the architecture level models as shown above. A few assumptions are made during the characterization of these models:

- Data activity is not a factor in the creating of these models. We simply assumed a random distribution of data that is similar to what we might expect in most real world application. As such, contrived workload with either low or excessive data activity will exhibit large error.

- We also did not model the cost of switching instructions. For example if a datapath supported two operations multiply and add, we characterized the two energies separately and did not take the energy associated with switching between add/multiply into account.

- RAM models were generated from commercially available RAMs and energy data was only available for RAMs running at one particular frequency. As such the energy/access is constant regardless of the frequency of operation.

By putting together both the building block and regression based model we were able to create an energy model such as the one shown in Figure 3.3. This model basically consists of various hierarchies of the design with each design consisting of a regression based model for associating PM counts with energy and possibly building blocks from a library which are associated with a particular PM count.

As discussed we need PM counters to derive the power information. These PM counters can be derived by performing either an instrumented simulation or using a performance specific simulator.

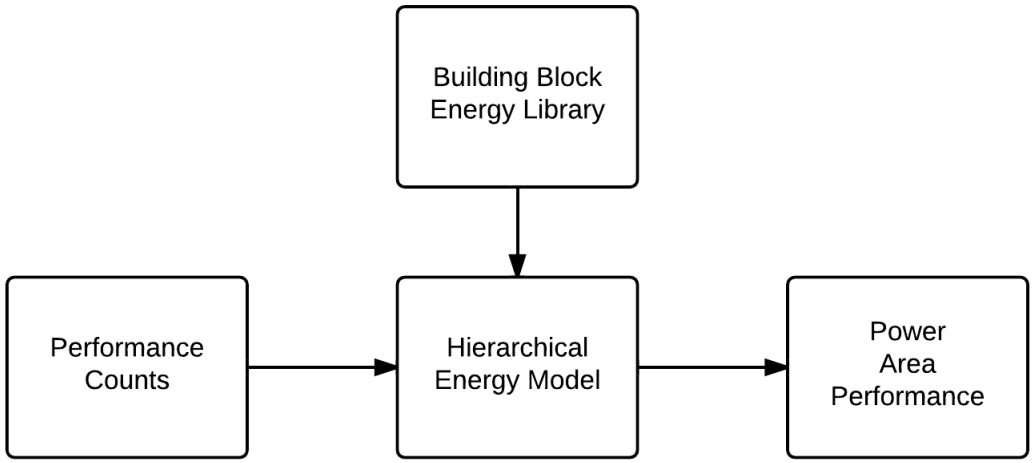


Figure 3.3: Energy Model : The big picture

3.3 Obtaining Performance/Application Level Data

Performance data is extremely critical to the construction of a high-level energy model and many techniques exist to obtain this data. We can capture the performance counters from existing silicon, simulate the performance counters in a performance simulator or simply monitor the performance counters in the Verilog simulation. We actually used all of these methods at different stages of this model development, and for performing the various studies.

Using silicon to get performance data has the advantage of being very fast. We can obtain performance data for a given frame in seconds rather than hours as required by simulation. The GPU has several performance counters, which are accessible during or after application run-time. Some of these performance counters can be accessed with publicly available tools such as NVIDIA NSIGHT[11], but we also used internal tools to access more detailed and larger number of performance counters. This information was immensely helpful in finding interesting workloads and even to determine if trimmed application traces exhibit the same behavior (allowing for their use in the detailed performance simulator).

Another method we used is the performance simulator. Other researchers constructed these tools, so the functionality of these tools is only briefly reviewed next. We broadly used three different types of simulators for doing the various studies. These are the architecture simulator, performance simulator and SM performance simulator.

The architectural simulator is a bit-approximate functional simulator with no notion of timing. This particular simulator is useful to gather quick statistics about how much work needs to be done (for example, primitives processed, shader instructions, etc.). However, this model is only SW architecturally correct and does not model any work reduction optimizations done by the actual design limiting its usefulness for direct power analysis.

The performance simulator is an extension of the architecture simulator described above. The performance simulator adds the work reduction features of the real design along with actual timing information. It is a fully execution driven simulation which is bit-accurate and cycle-approximate. We attempted to make sure that knobs in this simulation framework had equivalents in the power models.

The SM simulator (which is a part of the performance simulator) is a stand-alone trace driven performance simulator of the SM. This simulator simply models the ordering and timing behavior of the SM without modeling the actual functionality of the SM. Since this model is much faster than the full performance simulator, we used it for exploring and modeling the effects of many architectural changes for a number of applications traces.

For simple studies and to obtain data for fitting the model we just used direct simulation of the design using both internal and commercial tools. All power analysis was performed on the gate level netlist as described in Section 3.1. Using netlist simulations we were severely constrained on data collection and were only able to run really short application traces or directed tests. To alleviate some of these issues and obtain more data we were able to capture performance and power data from emulators developed by Cadence.

We used Cadence Palladium emulation[8], which allowed us to collect detailed power and performance information on actual application frames or frame segments

which were obtained by scissoring. Using this emulation process we can capture detailed design activity and performance signals during run-time. The process of using emulation and converting it to power information is shown in Figure. 3.4. Using emulation we were able to obtain detailed data for several frames a day, something, which could take weeks, if not months on RTL simulation.

The combination of the above performance measurements tools gave us the ability to both train our model and use it to generate power/energy information. However, we also wanted to understand application level characteristics of our app, such as data patterns and ordering. To capture this information we instrumented the shader code.

3.4 Shader Instrumentation

Perhaps the most useful silicon instrumentation was the ability to modify the shader code executed by the GPU. This is in practice similar to other binary instrumentation tools such as Valgrind and Cachegrind[48], which are used to examine execution of software on CPUs. Most of the stimulus data in Chapter 5 was obtained by using shader instrumentation. We used two major types of shader instrumentation:

- Operand Profiling : Log operands of individual instructions
- Pixel Profiling : Log the pixels that are affected by individual shader threads

The shader code is profiled by modifying the code as shown in Figure 3.5. Essentially for every instruction we append instructions that store the value of the operands to the GPU global memory. Right before the EXIT instruction we also add information about the x,y,z coordinate of this particular thread and write that information to the global memory as well. This information can then be scanned out at the end of the frame, which will allow us to capture instruction level details of the shader program while being able to run orders of magnitude faster than simulation.

Using the methodology described in this chapter we created a flexible energy model for a GPU. In the next chapter we will discuss both the details of how this model was constructed and details about the underlying building blocks used.

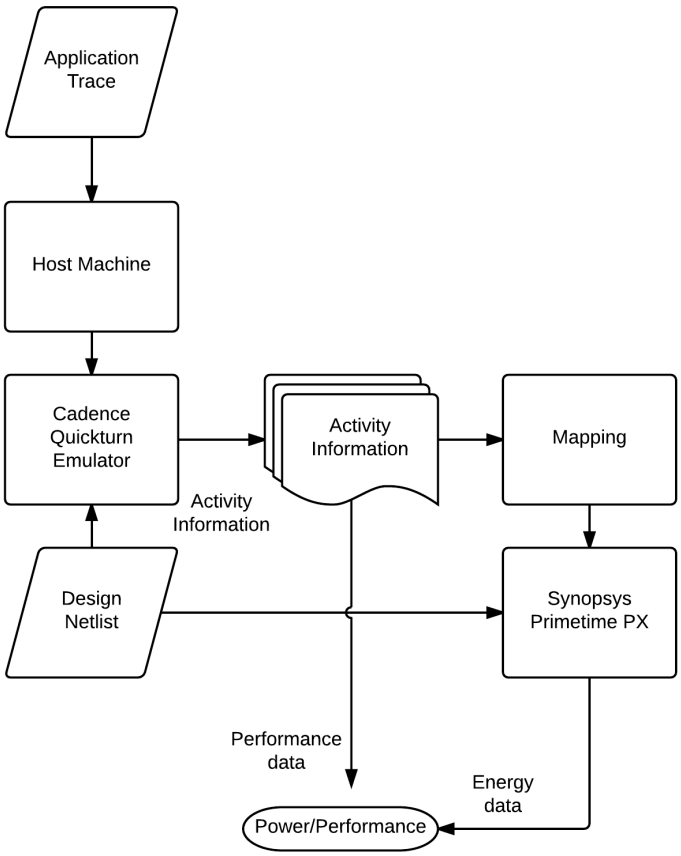


Figure 3.4: Getting Power/Performance/Energy Data From Emulation. The activity information which represents performance data is extracted from emulation and is combined with energy/activation information from synthesis to extract both power and performance data.

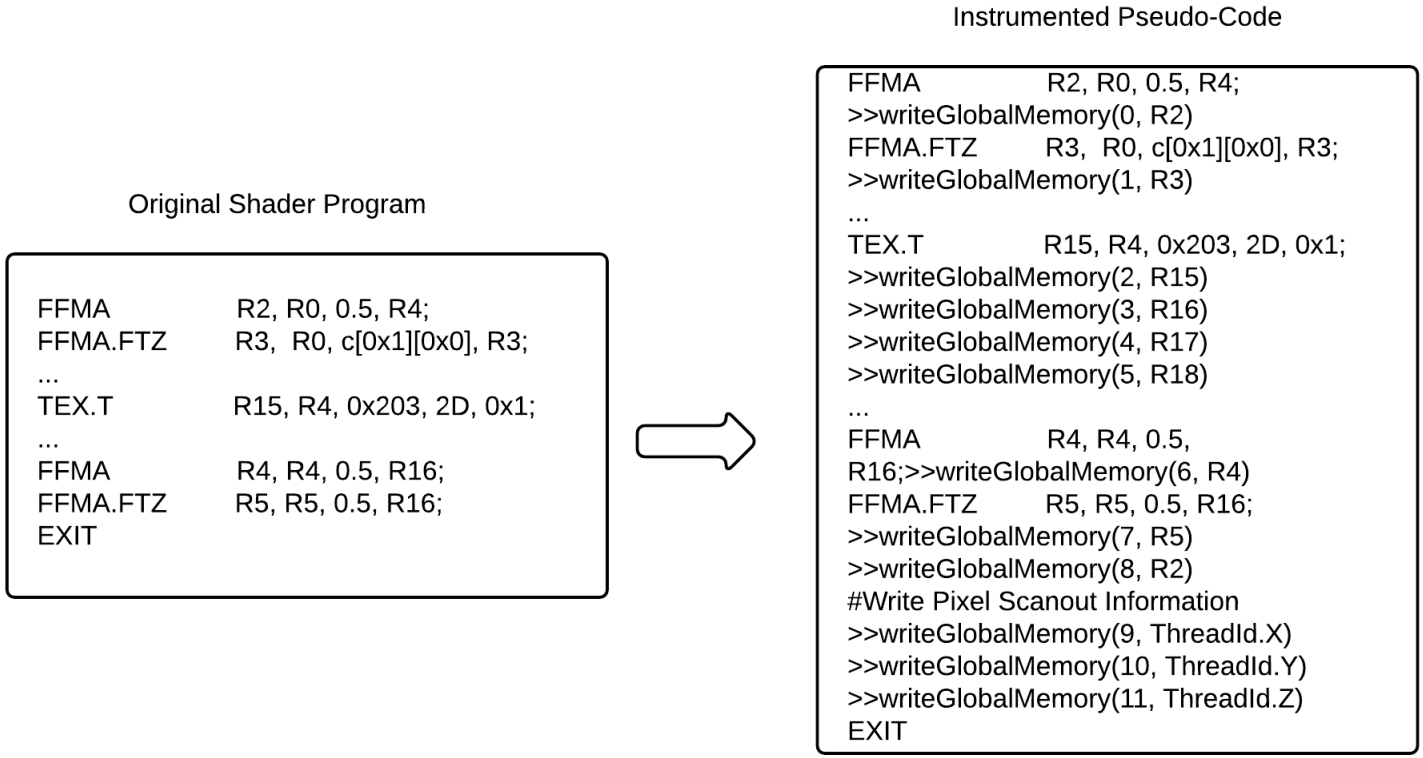


Figure 3.5: Instrumenting the shader to mine data and statistics. Instructions are added to the shader program, which write program coordinates to the GPU global memory. This memory can be read after program execution to gather information about the pixel coordinates processed, operand information, etc.

Chapter 4

GPU Energy Modeling

In Chapter 3, we described the methodology we used for energy modeling. This chapter discusses how we created the models. One of the guiding principles of creating our energy model was to create energy models based on only the level of detail necessary to capture the energy accurately and to model the most interesting (i.e. high energy) components first. To follow this principle we first looked at the energy breakdown of GPUs to understand where the energy is spent and what information is needed to model this energy.

Following this discussion, the next section describes how we constructed the detailed models for the building blocks. We will then use these models along with linear regression to create models of larger function like the shader and texture units. Correlation studies between our model and measured results show that we can predict the power/energy consumption of a GPU with good fidelity.

4.1 Energy Distribution

Understanding the energy distribution in a GPU is important: it focuses our modeling attention to the blocks that matter. Looking at the design of the GPU itself we can surmise that the shader is one of the heavy hitters but we wanted to know how much of the energy is actually used by the shader and how relevant the energy consumption of the other components in the GPU system are.

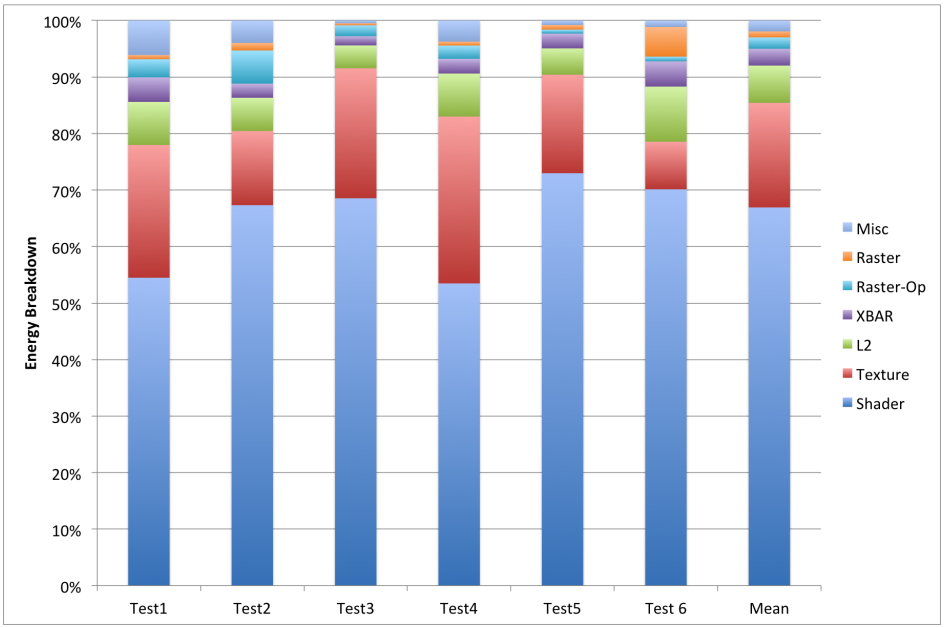


Figure 4.1: GPU energy breakdown across several tests.

Parameter	**Value**
Shaders	3
Textures	4 per Shader
L1 Cache + Shared Memory	3 X 64 KB
Register File	64 KB
L2 Cache	192 KB
Frame Buffer Partitions	1
Clock Frequency	1 GHz
Process	TSMC 28HP Typical
Nominal Voltage	1.0V

Table 4.1: GPU configuration used in performance/power simulation. Scaled version of high-end GPU.

To obtain this information we measure the dynamic energy distribution of a GPU using high speed simulation as described in Section 3.3. We did not measure the leakage component of the power consumption, as we were not looking at optimizing low power and sleep states of the GPU[1] The results of this are shown in Figure 4.1. The specific tests that we used were chosen from leading applications and benchmark suites and represent high-power use cases for the GPU. For this study we primarily focused on state of the art game frames, which stress various parts of the GPU. The GPU power is broken down broadly in the following categories:

- Shader : Unified shaders + L1
- Texture : Texture units
- XBAR : Fully connected XBAR interconnecting the Shaders to L2
- L2 : L2 Cache, Compression and Decompression
- Raster : Raster (setup, raster, zcull, tiler)
- Raster Operations Pipeline (ROP) : Color, Z raster units (anti-aliasing, depth)
- Misc : Top level control logic, Framebuffer compression, Framebuffer decompression, Framebuffer control

As shown in Figure 4.1, the shader in the GPU dominates the energy consumption followed by the Texture and L2 units. The energy breakdown is only of the GPU chip itself and does not include the energy used by the DRAM and PCI express host interface[2]. Given this breakdown, the vast majority of effort in modeling was spent on creating accurate models for the shader followed by accurate models for Texture and other functional units on the GPU.

[1]Historically, at peak dynamic power consumption the leakage power tends to be approximately 30% of the total power.

[2]The work in this thesis did not target the energy consumption of the DRAM or PCI express host interface so not much time was spent obtaining this information. Furthermore, the high-speed simulation methodology that we used does not work for obtaining these results. However, empirically, we know that the DRAM power is about 20-30% of the overall GPU power consumption when the GPU is operating in a high performance state. The PCI express interface is a much smaller part of the power consumption accounting for less than 5% of the total power

4.2 Energy Models

Our hybrid model must capture most of the shader, texture, L2 and XBAR energy as building blocks to provide the desired flexible, accurate energy model. This section discusses how the building block energy models were constructed, while the next section goes into details of actually constructing a regression based model which use these building blocks. Since the shader is a type of processor, and it's a large fraction of the GPU power, we used categories similar to the ones used in CPU power models such as Wattch [6]. We broke down the energy models into the following categories:

- Datapaths : The actual compute units such as floating point units, arithmetic units, etc.
- Storage/Memory : Register File, Caches, SRAMs, TLB
- Interconnect : Wiring, XBARs
- Clock Distribution : Clock trees, clock buffering

Unlike existing energy models we had access to a wealth implementations of actual functional units used in GPUs. This allowed us to create a both a high fidelity and flexible model. Furthermore, all of our functional unit models were fully self-contained including the overhead for physical placement, clocks trees and any sequencing overhead introduced by pipelining. Having the models fully self-contained allowed us to compose different sets of models together based on the configuration of the performance simulator. For example, if the performance simulator was setup to have 4-cycle latency in the datapath, the energy model of the data path would account for this design choice.

The datapath was hierarchically broken down into smaller units, until reaching a "leaf" unit. An example of a leaf building block is a simple floating-point unit. These leaf units are parameterized, which increases the flexibility of the higher-level models. For example a leaf floating point unit might have parameterized pipeline depth, frequency and data width. The models contain information about the resulting energy/operation, and its area and clock loading. To provide a feeling for the

complexity of the models we created, Figure 4.2 displays the energy for a fused multiply/add operation in a leaf floating point unit. The figure only provides data that is on the optimal frontier for the chosen pipeline depth.

Generating these models required extensive simulations that attempted to find an optimal floating-point unit by varying the pipeline depth and frequencies then using synthesis and place and route to create the actual design. We obtained these models by using a parameterized/generated Verilog design for a FMA floating-point unit, which allowed for different topologies and pipeline depths. For each of the parametrized design points we performed logic synthesis using Design Compiler [4], while sweeping the clock period to obtain different design points. We used the power estimation methodology as discussed in Section 3.1. This process of obtaining this model is similar to the methodology used by Sameh et. al. [20], and this work was conducted concurrently.

Performing this parametrized synthesis allowed us to obtain the energy, area and throughput information with various frequencies and architectures. This allowed us to construct a Pareto curve for this unit as shown in Figure 4.2. In this curve we plotted the area efficiency, which is the area/throughput ($mm^2/Gops$), vs. the energy efficiency, which is energy/operation (pJ/Op). Designs not on the Pareto frontier (following the edge of the curve towards the lower left) are not optimal. If a design does not appear on the Pareto frontier we can always select another design that will either have better area or energy efficiency. The Pareto curve allows us to quickly see the trade-offs with picking different design points. For example, if we want a more area efficient design we would target the left side of the graph, which achieved the most $Flops/mm^2$. These are more deeply pipelined machines, and were the best choice when designs were transistor limited. But they are also energy inefficient. Because of the law of diminishing returns, designs at either end of the graph are rarely a good choice. Looking at this graph we can quickly see that a 4-stage pipeline design occurs on the Pareto frontier (and the knee) of this graph. This also happens to be the pipeline depth of the datapath used in the shader units that we studied.

We used a similar methodology to construct models for other datapath building blocks. This included integer units, units at different bit-width and even custom

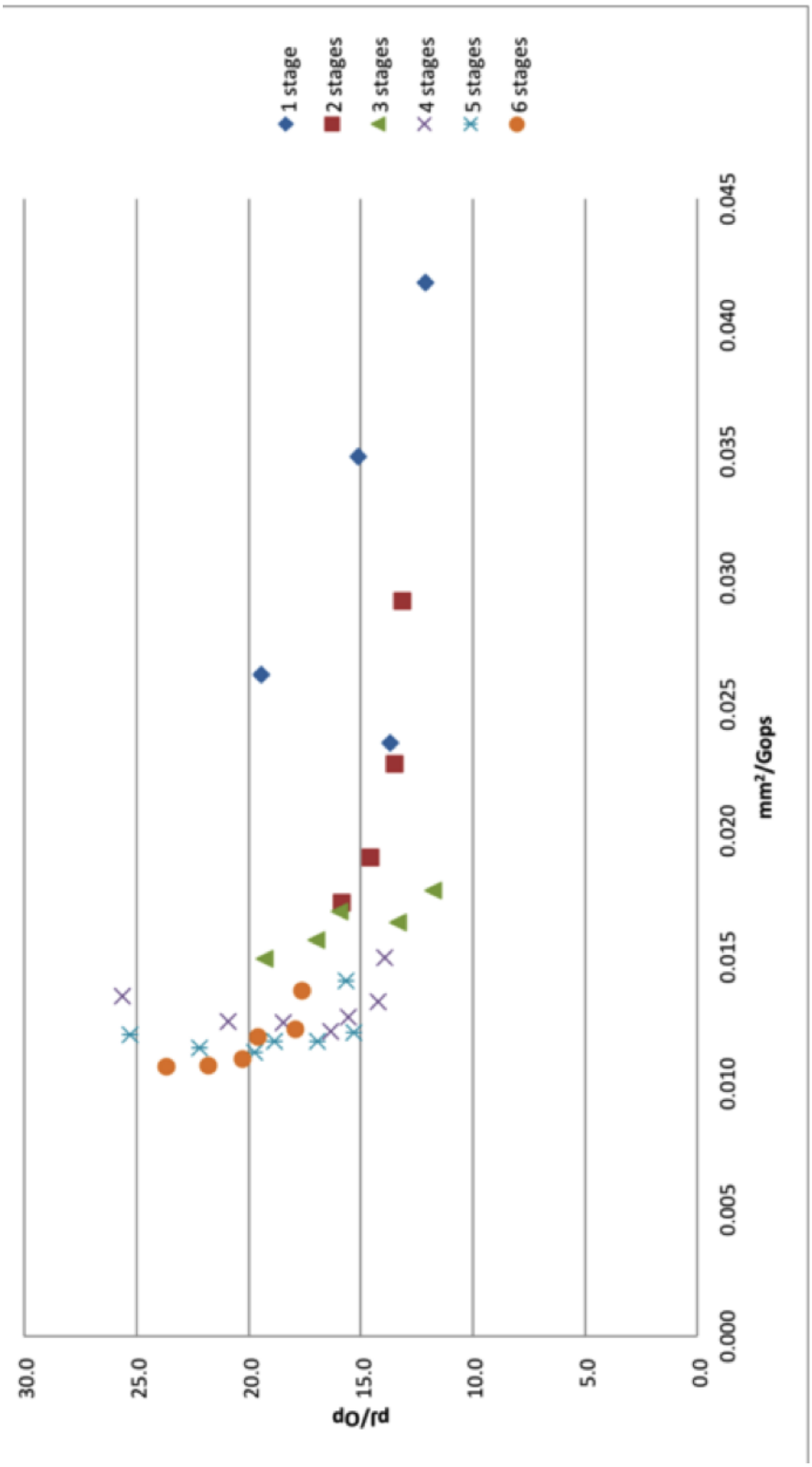


Figure 4.2: Energy, area efficiency for various datapath designs executing the single precision fused multiply add instruction in 28nm HP technology. Only the result with optimal frequency for each pipeline depth is shown

logic blocks. We also created detailed models for clock distribution networks, communication crossbars, memory structures and various specialized stages of the texture pipeline. This allowed us to have various designs with parameterization that we can use to compose more complex models.

The energy models for all storage elements were based on using memory compiler/generators that were used for creating memory by tiling custom designed memory cells. Since all the memory cells were custom designs we did not have energy values for these designs optimized for different frequencies. Hence all the energy numbers used for the storage elements were done assuming a 1GHz operating frequency with 28nm typical process.

The interconnect models were fairly straight forward based on expected capacitance/mm from the process specification and the overhead introduced by repeaters and flip-flops for longer wires. The energy models for interconnects are only accurate for fairly long wires (>300um) since smaller wires tend to be dominated by the input capacitance of the sink and our model was only accurate when at least one repeater was used. This was not a problem since we only used these models for longer global interconnects and not for shorter local wires, which were estimated using the load capacitance of the sink model.

The clock tree model was constructed to take clock gating into account. There are two main forms of clock gating: fine-grained clock gating, and block level clock gating. Fine-grained clock gating is used to gate a small group of flip-flops (usually between 4 and 32) while block level clock gating is used to turn off entire functional units. Specialized cells called ICG's (Integrated Clock Gates) are used to turn off the clock. These cells contain a latch to prevent glitching and provide support to turn on clocks during scan testing. The structure of the actual clock model is shown in Figure 4.3. As in most well designed clock trees, energy tends to be dominated by the leaf nodes. The clock model breaks down the reported power into various categories:

- Trunk : These are the main clock distribution buffers and their energy is mostly dominated by wire distribution. (Chip Level)
- Main Tree : The tree after any block level clock gating (wiring dominated)

- FGCG : Fine-grained clock gates. Responsible for turning off the clock for a small group of flip-flops
- Gated/Ungated Subtree : The part of the tree that actually drives the leaves.

The base model for the clock tree is still the interconnect model as most of the energy in the upper part of the tree is due to running long distances. The main difference here is that unlike signal wires clock tree wires do not pass through flip-flops and need to maintain sharper edges. The need to maintain sharper edges implies smaller spacing between repeaters and hence more gate capacitance. As a matter of fact, the energy/mm is approximately 3 times higher than signal wires at approximately 300fJ/mm in 28nm process at 1.0V.[3]

This comprehensive clock energy model keyed on number of leafs, leaf capacitance, design area and clock gating percentage was useful to quickly estimate the clocking overhead of various designs and floorplans without doing a full place and route.

4.3 Constructing Regression Based Models

Now that we have these building block models, we need to derive the hybrid regression building block model that we discussed earlier. Our strategy will be to first subtract the energy use estimates generated from the building blocks and the performance counters, and run the regressions on the resulting residual energy. This approach not only merges the two models, it also captures how well the building blocks model the energy of the underlying units. If the building block models are accurate and capture a large part of the design, the resulting residual energy correlated with the block activation should be small. Even if the residual is not zero, since the regression-based model applies to a small portion of the design (10-15 %), bad scaling heuristics are unlikely to cause a significant error in energy estimation.

[3]An interesting data-point is that the amortized costs of a flip-flop was 7fJ with about 60 percent of the energy consumed in the leafs (clock buffers/logic in the flip-flops). This information turns out to be useful for making quick estimates about the energy cost of sequential logic. As a side effect of this work this also led to some further hardware optimization of the actual flip-flop elements used in the design.

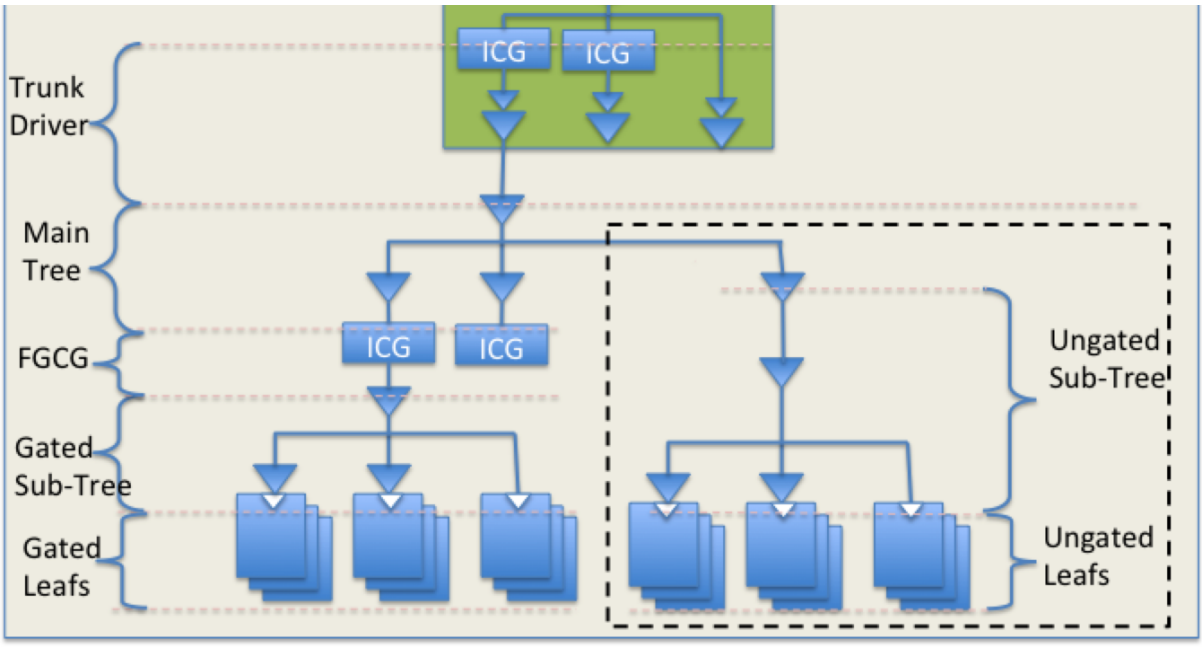


Figure 4.3: Clock tree model, clock is broken into several sections trunk, gated, ungated

To show how this method works, this section walks through the modeling of the shader processor, one of the most complex, replicated, and energy-consuming units on the GPU. Figure 4.4 shows how we decided to partition the shader model between building blocks and regression fits.

As explained in Chapter 3, we started with the building blocks and assigning performance signals that should correlate well with the activity of that block. Some of the performance signals used are shown in Table 4.3, along with the blocks that they influence. Since the model is hierarchical we can assign the signals we use to a general area of the design. The energy of the major building blocks for 1GHz operation in a 1V 28nm technology for the config shown in Table 4.1 is shown in Table 4.2. In Chapter 5, when we perform studies on the shader, we show how energy of these building blocks scale.

After we assign performance signals to the various parts of the design we can then take the performance signals that correspond to specific building blocks and subtract the energy used by the building block from the total, whenever the given building

Building Block name	**Energy Consumption**
Local Datapath (FMA)	12 pJ/op
Local Datapath (FMUL)	10 pJ/op
Local Datapath (FADD)	7 pJ/op
Load/Store (Load/store unit + XBAR)	34 pJ/ 4-byte word
L1 Read	8 pJ/ 4-byte word
L1 Write	10 pJ/ 4-byte word
RF Read	5 pJ/ 4-byte word
RF Write	7 pJ/ 4-byte word

Table 4.2: Approximate energy values used for the major building blocks

block is active. We perform this for all the building blocks used in the design and are left with the residual energy, which is simply the total energy of the design minus the energy of the active building blocks as predicted using the performance signals. For example, if the datapath is active and a FMA instruction is being performed we will subtract the FMA energy from the total energy. The residual energy is then put into linear regression to derive weights. If no building blocks are used then the residual is simply the entire energy and the generated model will be purely regression based and not hybrid.

To build the model we obtained training data by running unit level simulations of the shader under various circumstances including the use of directed tests. These unit level simulations yielded two useful artifacts: performance counter values and the energy used by the design. The energy was obtained by simulating post synthesis, topological wire routed designs at the gate level. For the most part we used functional tests that were used for verification of the design but sometimes instrumented them with additional stalls in order to capture the stall behavior of the design. Overall, approximately 30 unique tests were used, however, we had over 1000 training points since we broke down different regions of a given test and used each as a unique observation. We also used cross validation in our regression to reduce over-fitting of the model.

In order to validate the model we obtained more data by running parts of various industry leading benchmarks (3DMark, Citadel, etc) on high-speed simulation (as discussed in Chapter 3). While running these tests we captured the energy used by the design at approximately 5μs intervals. Once again, we replayed the simulation results

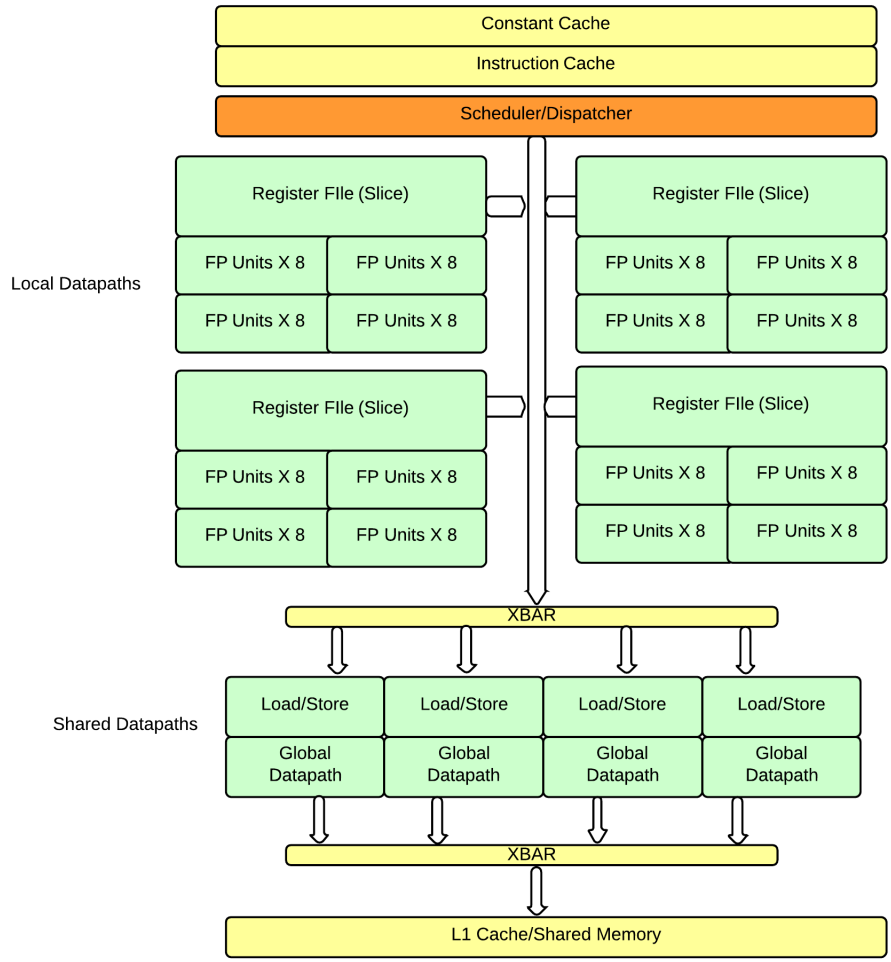


Figure 4.4: High level SM architecture showing modeling methodology used. Green means the unit was modeled using energy from building blocks. Yellow means a mixed model was used where the regular structures were modeled from building blocks but control was modeled using regression fits. Orange blocks were modeled purely by using a regression-based model. Other logic that is not shown (for example, texture control) was modeled using a regression fit.

Performance Parameters Used	**Comments**	**Used for modeling**
register_read_bank[0..3]	Register read by bank	RF, XBAR
register_write_bank[0..3]	Register write by bank	RF, XBAR
inst_issued_pipe[pipe_name]	Number of instruction executed by a given datapath in the shader	Datapaths, XBAR
inst_issued	Number of instructions being issued	Control
tex_fetches	Pixel quads from texture unit	Control, Tex Interface
inst_executed	Number of instructions being executed	Control, Misc
store_bus_active	Store bus is active	Control, Store
imc_hit	Immediate Cache hit	Immediate Cache
imc_access	Immediate Cache Access	Immediate Cache
l1c_hit	L1 Cache Hit	L1 Cache
l1c_miss	L1 Cache Misss	L1 Cache

Table 4.3: Sample of performance signals used in the modeling of the shader. This is not a comprehensive list, however, these are the most important signals overall

on post synthesis, post topological wire routed designs to obtain the simulated actual energy. This actual energy normalized to the maximum energy is plotted against the model energy in Figure 4.5.

The resulting model has very good correlation with this validation set, the $r^2 = 0.98$. The validation set was not used during the training and model selection phase. The model, however, tends to slightly under predict the power on average[4] There is also a significant spread in the middle power ranges. While, some tests are perfectly on the 1:1 slope there are certain characteristics of the design that are not extremely well modeled. For example, units in the model that are not active do not consume any power while in the actual design might not be perfectly gated. One scenario where this will happen is if there is a bubble in the pipeline. This happens when we execute a floating point operation followed by several no-ops. In the model the data path will consume all the energy when the operation is executed (single cycle). In the actual hardware the unit will remain active for multiple cycles because an instruction is in flight in the pipeline. Though the total energy used for the operation will be the same across the actual implementation and the model, the actual implementation has some wasted power while it's kept active. We accounted for this inaccuracy in areas of the design that were modeled using regressions. In areas of the design that were completely modeled using building blocks this inaccuracy remains. Some of this inaccuracy might have been alleviated if we fit residual energy for the large areas of the design that were completely modeled by building blocks.

This model can be improved by better taking into account some of the inefficiencies of the design or perhaps by increasing the size of the training set to account for more scenarios. In general the correlation of the model improves as we increase the interval over which the results are averaged. Overall our results are quite good, the model is within 15% of the expected power results while remaining flexible enough to explore interesting trade-offs.

Another case study we looked at is applying the energy model to the texture unit. In this case we followed a similar test methodology as the shader. The texture unit

[4]This is not normally expected from a linear regression based fit. We mostly attribute this to inaccuracy in the power estimation methodologies, especially because the high speed simulation does not account for state dependent, path dependent power in the SRAMs as discussed in in Chapter 3.

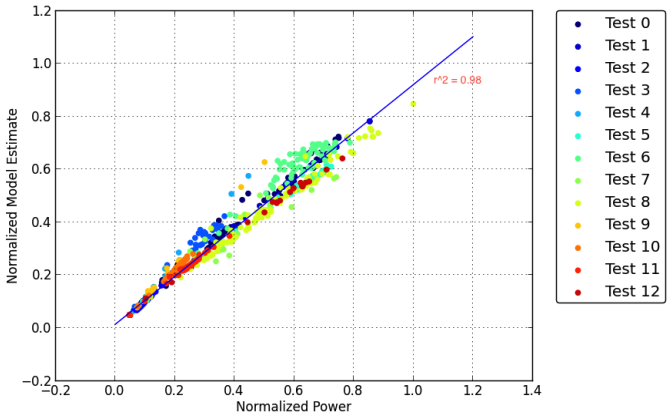


Figure 4.5: Power correlation for shader. Normalized to max power obtained from simulation

was modeled by breaking it down into 9 major segments. Each of these segments performed a major operation in the texture unit for which we had corresponding performance signals available. The texture unit had building blocks for various internal datapaths as well as internal memory structures. We manually assigned performance signals to various parts of the texture unit then fit the left over residual energy using linear regression as discussed in Chapter 3.

Even though the texture unit performs some complex tasks, it internally resembles a more fixed function pipeline compared to the shader. This allowed the model to be extremely accurate. Using the same tests as were used for the shader we obtain correlation plot as shown in Figure 4.6. The texture model achieves a r^2 value of 0.987. Once again while the fit is good there is still some variation and the model still tends to under-predict the energy. We expect that this error is observation error in our validation data as discussed earlier.

Using a similar methodology as the texture unit we built up a model for the other major units on the GPU. The correlation of this versus the power obtained from high speed simulation is shown in Figure 4.7. Overall, the fit is very good however there is still some under-prediction. Some of this under-prediction occurs for reasons similar to the SM and texture. Some under-prediction also results because we did not model all the small units of the GPU and they add up to about 5% of the power as shown

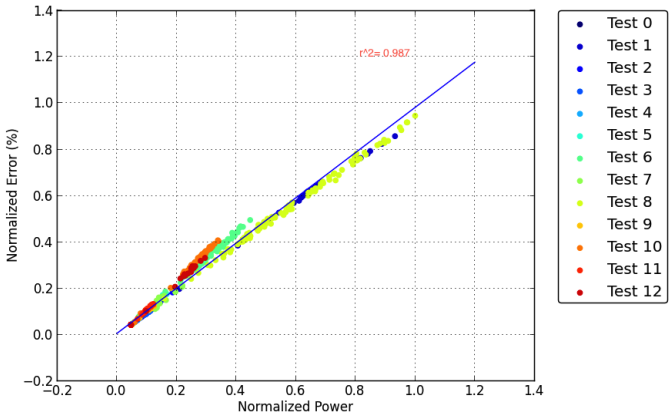


Figure 4.6: Power correlation for texture unit. Normalized to max power obtained from simulation

in Figure 4.1.

4.4 Comparison to GPUWattch

While this work was conducted around the same time as GPUWattch, it's worth noting similarities and differences. The overall methodology of GPUWattch is very similar to this work. Their model is constructed from a set of building blocks, which are compared to measured power using microbenchmarks that stress a specific component. They then refine this model by scaling the numbers, or by obtaining more accurate microbenchmarks, until they get an acceptable fit.

When constructing our model we had access to the details of the micro-architecture as well as the ability to execute simulations and measure power on a given part of the design. This allowed us to get a better fit, which gives us more accurate unit level breakdowns and overall prediction of total power. For example, comparing Figure 9 from [36], it is clear that our model has better overall prediction of the power as shown in Figure 4.7. Another interesting thing to note is the difference between energy breakdowns between different units of the GPU. For example, in GPUWattch, on average the NOC uses 9.5%, which in our model is called the XBAR. The XBAR in our model uses substantially less energy, while the L2/shader uses a lot more.

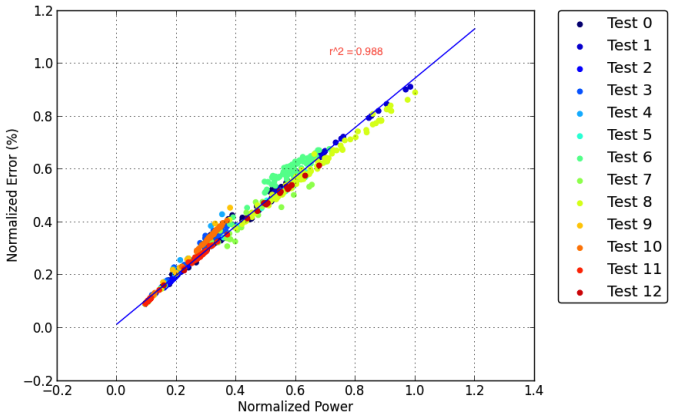


Figure 4.7: Power correlation for the GPU. The shader is model is constructed using a hybrid model. All other models are purely regression based. Normalized to max power obtained from simulation

Since our model was constructed with detailed knowledge and simulation of various components we know that our component level breakdowns are accurate. While it's impossible to a 1:1 comparison of GPUWattch with his work, since they use different set of benchmarks (compute focused vs. graphics focused), and a different generation of GPUs (Fermi vs. Kepler), it is clear that there is some level of systematic variation.

Having a different breakdown than the actual design can cause one to make different design tradeoffs. For this reason it's important to have both an accurate model as well as accurate breakdowns of where energy is spent in the design. There are likely several factors that led to GPUWattch having different results than us. GPUWattch is based on using external measurements of the GPU, which can be inaccurate due to RLC effects as discussed in [36]. While they compensate for that, it's hard to get a microbenchmark that can isolate a region of the design with complete accuracy. One of the things that we learned throughout this process is that it is very difficult, if not impossible, to create an accurate energy model without a detailed understanding of the design. Furthermore, having detailed energy breakdowns from existing designs helps guide model construction.

In this chapter, we walked through how both building block and actual energy models are created. We dived deeply into creating a model for the SM by using various performance counters, and then followed that with a model for texture unit

and the other parts of the GPU. Overall, we have an accurate yet flexible model for the GPU, which is suitable for conducting architecture exploration and studies. In the next chapter we will explore using these models in studies, which might yield insights that can lead to significant reductions in energy consumption for GPUs.

Chapter 5

Energy Limit Studies

This section uses the energy models as presented in Chapter 4 to understand some inefficiencies and redundant work done by the shaders in the GPU. The first section will describe energy breakdowns of a typical GPU SM (shader processors). It can be seen in these breakdowns that the shader energy is dominated by the datapaths: control/instruction overhead is small since it is amortized by having a large number of floating point operations that are processed in parallel.

While a typical GPU shader is meant to be an efficient processor with high computation density we found several cases where inefficiencies in the pipeline result in extra work being done by the shader. This extra work translates into lost energy and overall higher power consumption of the shader. The next few sections describe limit studies, which try to look at inefficiencies caused by extra work. These include:

- **High Precision Math:** In modern GPU shaders the majority of math operations are done using either 32-bit or 64-bit IEEE compliant floating point operations[69, 68, 50]. This makes the SW development much easier since one does not need to worry about overflow or loss of precision during scaling. After looking at several frame level profiles we found that in most cases the high precision of these floating point units is never used. We wanted to quantify the energy savings of going to lower precision floating point or even integer operations. This information can then be used to drive algorithmic/compiler improvements to reduce computation precision.

- **Overdraw:** Overdraw occurs when the shader processes a single pixel value more than once. This can happen when the primitives are not rendered in front to back order (reverse painter's algorithm)[22, 67] which can cause the shader to run on the hidden pixels causing extra computation which will have no impact on the image output. We take a look at the overdraw across several frames and quantify the idealized energy savings assuming everything was rendering in front to back order.

- **Redundancy Across Threads:** The shader also operates on 32-wide thread groups, which reduces the control overhead of the design. We refer to this grouping with the NVIDIA terminology of warps. This is the equivalent of the SIMD width on the CPU. Profiling several frames, we found that there is significant thread level redundancy where more than one thread in the warp is using the same input operands. We take a look at these frame level profiles and quantify the energy savings of shutting down redundant threads.

5.1 Benchmarks and Methodology

To perform the shader studies, several application/benchmark frames were used as shown in Table 5.1 and Figure 5.1. These frames were specifically chosen as they are considered important for the next generation of GPU designs and come from applications commonly used to showcase GPU performance. The frames were also chosen since they have fairly diverse workloads (but all were pixel shader heavy as are most frames) as characterized by GPU performance counters. This choice of benchmarks was made to avoid biasing the results to a particular workload.

Shader programs were dynamically instrumented by the driver as described in Section 3.4. This instrumentation was used to capture operand data (for shader precision and redundancy study) and the frame buffer writes (for the overdraw study). All pixel shader programs for a given frame were instrumented to capture operands while only pixel shader programs that write to the final frame buffer and had blending disabled were used for the overdraw study. If blending is enabled, then the overdraw

Table 5.1: These are the benchmarks that were selected. We used specific frames from these benchmarks that are known to have relatively high energy consumption

Frame Number	Application
1	3DMarkVantage
2	3DMarkVantage
3	3DMarkVantage
4	Batman Arkham City
5	Battlefield 3
6	Dirt3
7	Samaritan Demo (FXAA)
8	Metro 2033 Benchmark

changes the pixel values and is not redundant computation. The accuracy of this data collection method is also discussed in this section.

Since capturing all operands for every instruction executed in the shader for a particular frame is not very practical (>30 Billion operands per frame) sampling was used to reduce the data captured while trying to maintain the overall statistics of the data. For most cases, capturing 5% of the frame was sufficient and these were selected by randomly scissoring the output frame into 64 x 64 pixel non-overlapping regions. The size of 64 x 64 was chosen since it was the most optimal unit for capturing the shader data for the particular GPU used.

Energy estimation in this chapter was done using performance counters and energy models as described in Chapter 3. While a lot of silicon/netlist level correlation was done on the energy models described earlier, obviously the results in this chapter are based purely on the performance models since these designs have not yet been implemented.

5.2 Shader Energy Breakdowns

The frames (Figure 5.1) were taken through the energy estimation flow as described in Chapter 3 on the original shader implementation. The results of the energy estimation were broken down broadly into the following categories:

(a) 3MarkVantage (Frame 1)

(b) 3MarkVantage (Frame 2)

(c) 3MarkVantage (Frame 3)

(d) Batman Arkham City (Frame 4)

(e) Battlefield 3 (Frame 5)

(f) Dirt3 (Frame 6)

(g) Samaritan (Frame 7)

(h) Metro 2033 (Frame 8)

Figure 5.1: Frames used for benchmarking

- **Idle:** The amount of energy that is used when the shader is idle but the clock is ticking. This refers specifically to dynamic power not static leakage. Most of this energy is waste since this can be mostly eliminated with perfect clock gating (other than clock gating overhead)

- **Control:** This is the energy spent in instruction decode, scheduling, ordering and managing texture transactions.

- **RF:** This is the energy spent in the register file of the GPU. Keep in mind that the register files on GPU are substantially larger than those on conventional CPUs.

- **Local DP:** The data path that is local to a single slice of the shader and is physically co-located to a slice of the register file. This local datapath is responsible for performing common operations such as FFMA, FMUL, FADD, etc.

- **Global DP:** This is the datapath that is shared across several shader slices. While the throughput of the global data path is lower it is used to perform either rare or complicated instructions. For example: SIN, EXP, DFMA, etc. The texture unit is also in the global datapath of the shader but its energy is not included since it is physically not part of the shader and can be shared with several shaders (or have multiple instances per shader).

- **Cache:** This refers to the energy used by the shader L1 cache or shared memory. In most pure graphics workloads if the application does not contain CUDA or DX Compute code the shader cache is used as either a shared memory or as buffers for the primitive engine.

The estimated energy broken down by functional units is shown in Figure 5.2. The energy has been normalized so that the breakdown can be shown without regard for the actual energy consumed by any given benchmark.

It is clear that the shader spends most the energy in the datapaths and the register file. These units account for about 50% of the energy. This is in contrast to general

purpose CPUs where the majority of energy is spent getting the instructions and data needed for computation [25, 2]. These results imply that improving shader design can at most yield a 40 percent reduction in energy if nothing was done to the datapaths or register files. There is room available for reducing energy in the control logic of GPUs by switching to doing static scheduling or reducing waste, however, we can't get large reductions in energy without either improving the datapaths or figuring out how to do less data processing.[1] The next few sections touch on ideas to both reduce data processing cost and also reduce the amount of data processing required to render a frame.

5.3 Understanding Floating Point Precision

The majority of datapath operations performed on the GPU are done using 32-bit IEEE compliant floating point, which consists of a 8-bit exponent, 23-bits of mantissa and 1-bit sign. The final render target for a GPU is reduced to display precision, which is at usually 8 to 10 bits (integer) per color channel.

GPUs do computations in higher precision and truncate later for several reasons. In vertex shader workloads world coordinates can have a large dynamic range. Supporting a large dynamic range without floating point numbers is difficult since a lot of rescaling might be necessary. For pixel shaders, use of floating point numbers allows intermediate computations to be done without worry of saturation or overflow. This greatly simplifies the programming model at the cost of more complex and less efficient hardware. This section will focus mostly on pixel shader workloads since that is where the majority of energy is spent.

There are two parts to understanding if we can/should do cheaper arithmetic operations. One is to understand the impact of reducing the precision on the energy consumed by the shader. The other is to understand the workloads and determine whether such a change would lead to unacceptable image quality reduction.

One way to get a quick estimate of the potential energy savings is to scale the

[1] There has been ongoing work/research into creating more efficient floating point data paths. We take into account the existence of these datapaths in our results [20]

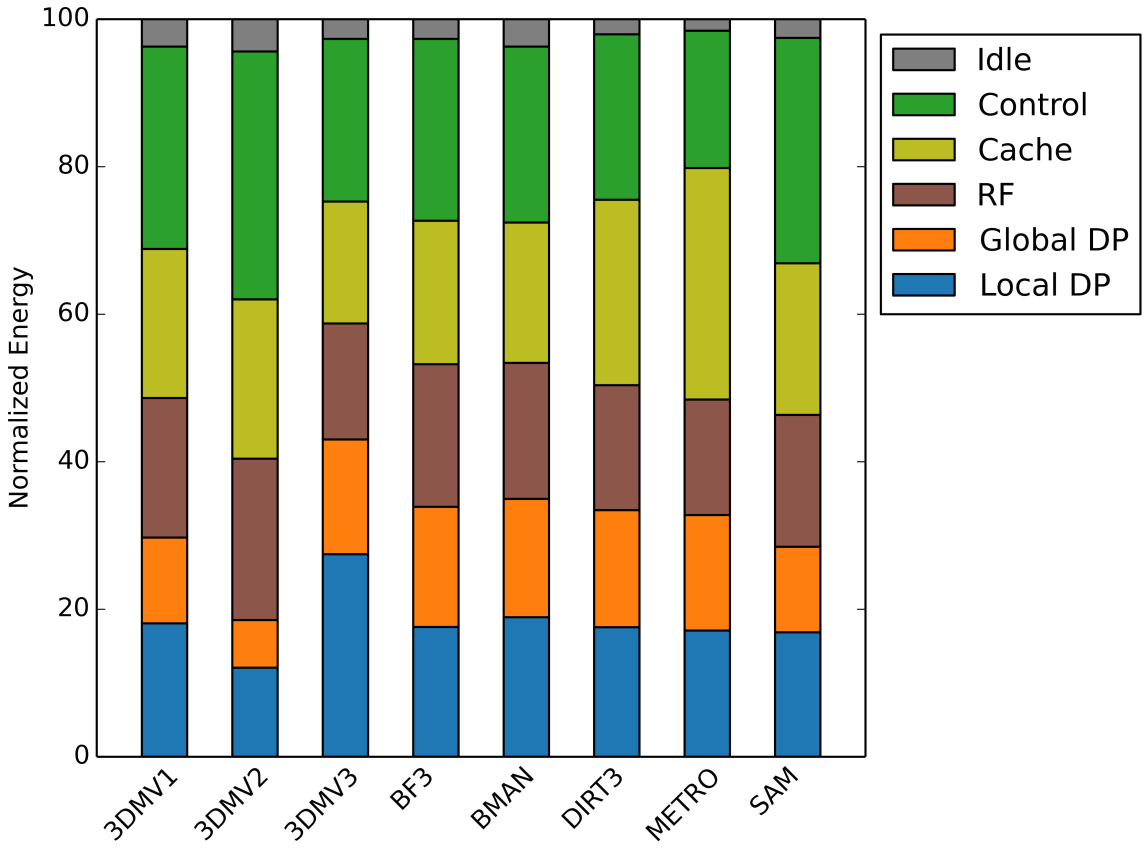


Figure 5.2: Shader energy breakdown across benchmarks

arithmetic and register file energy based on a lower precision design. Since, the shader relies heavily on compiler driven software pipelining we wanted to keep design changes to a minimum and decided to evaluate datapaths designs that could complete the given operation in 4 cycles at 1GHz in a nominal 28nm process.

The energy per arithmetic operation is shown in Figure 5.3. These energy numbers were obtained on a nominal 28nm process for a datapath running at 1GHz with a latency of 4 cycles. From the graph we can see that there is a large energy difference between 32-bit FP and 64-bit FP. Part of this is because double precision floating point is more complex and the other is because of the cycle time pressure causing gates to be up-sized. Similarly we see that savings decrease between 12/16 bit integer data paths as the design is starting to get dominated by overhead imposed by flip-flops and clocking structures. Some of this overhead may be artificially introduced in our study since we decided to keep the datapath latency the same (4-cycles) to prevent impacting the control/scheduling logic. Changes to scheduler/compiler might yield more further improvements in energy efficiency. Similarly, the energy/access/byte (assuming 32 word wide access) for the register access is shown in Figure 5.4.

Using this information and performance counters we can come up with performance estimates for the design assuming that we can change all single-precision floating point operations (evaluation of which will be discussed later); Figure 5.5 shows these results. Moving to 16-bit FP saves about 20 percent of the overall energy across the frames benchmarked while moving to 16-bit integer saves about 30 percent of the overall energy. As we move to lower precision the design becomes more dominated by control and idle energy. While the shader design is well balanced for a 32-bit FP arithmetic it might need significant changes to optimally use a 16-bit/12-bit design, including changes in the datapath cycle time which can have a significant impact on energy.

Given the significant potential energy savings, we now need to look at the workloads to see if this might be possible. In order to do this we extracted operands from shader programs as discussed in Section 3.4. In particular we captured the output values of every single math operation in the sampled region of the frame. Each operation was broken down into the exponent and mantissa and analyzed. The exponent

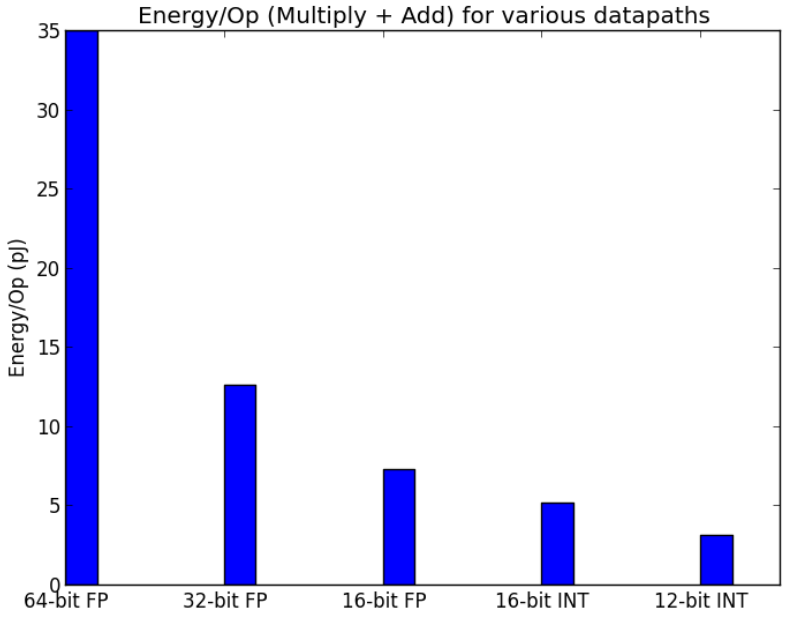


Figure 5.3: Energy/Op for performing a fused multiply-add using datapaths of different precision. All the datapaths used have a 4-cycle latency and include the corresponding clock and pipelining overhead.

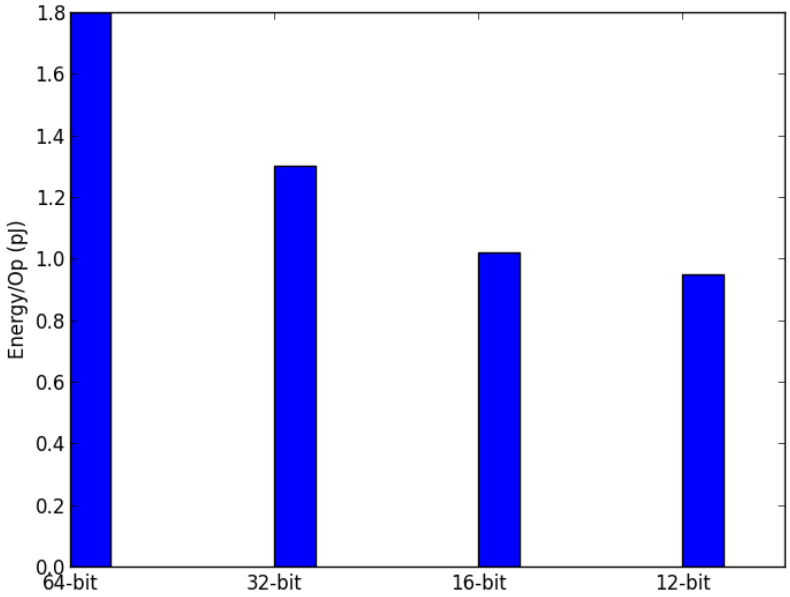


Figure 5.4: Energy/Access/Byte for 16KWords register file vs word width

is plotted in Figure 5.6 and is expected to be in the range of -126 to 127 if the entire range is being used since it is a 8-bit number. However, we see that the 99th percentile does not exceed half the range afforded by the 8-bit number. As expected the mantissa values were uniformly distributed.

While it is obvious that the full exponent range is not really utilized reducing it has only a marginal savings in energy, since the extra exponent range accounts for just one extra bit and an extra shift in the data path. The majority of energy for an FMA operation is consumed by the mantissa computation. At initial glance it might not seem like this data really supports moving to a lower precision design since the entire mantissa range is used and only a single bit can be eliminated from the exponent.

This is where we need to consider the final format and resulting values of the pixel shader program. The results of the pixel shader are truncated to screen precision which is 8 to 10 bits per color channel. This means that most of the mantissa range is unused unless there are large mantissa over/underflows. Since the exponent is not

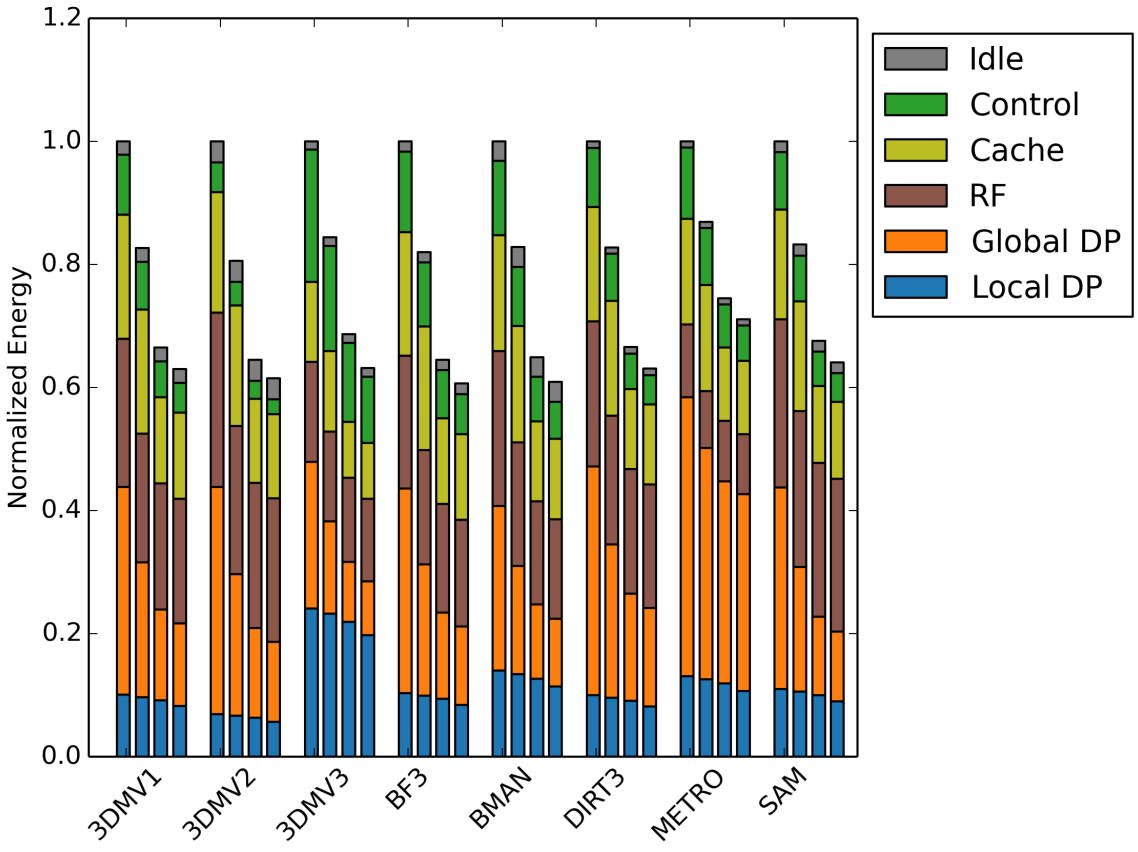


Figure 5.5: Estimated datapath energy for the various benchmarks. Each group consists of 4 bars that correspond to 32-bit FP, 16-bit FP, 16-bit Integer and 12-bit Integer from left to right.

fully utilized we can see that a large part of the resulting mantissa is truncated and lost when converting to screen resolution, which means that we could have skipped computing it in the first place.

Since we see that the entire numeric range is not utilized we can exploit this to save energy in several ways. One is to replace the FP hardware with lower precision hardware. This option might not be entirely practical since high precision is still needed in some cases. Also, 32-bit compliant FP math is commonly used in the ever increasing GPGPU programs [50]. Another option is to analyze the required precision by applying techniques in [23] and use multi-precision hardware [29] which will execute the programs with precision necessary based on input dynamic range and required output range.

To understand how much error would be introduced by computing pixel shader values using lower precision math we compared the operand values for all the math operations performed. If the math operation is a multiply or divide then the dynamic range of the inputs has a significant impact on the results. If the operations performed are an add/subtract the higher value will dominate with a large difference in dynamic range. As shown in Figure 5.7 we aggregated this data across our benchmark applications and recorded the percentage of threads where there was at-least one computation that could not be performed with lower precision and the value was utilized. Most of the applications had less than a tenth of a percent of threads that were incorrect, with Samaritan being the largest at about 0.4% for computing with FP16 math. While there were errors in the results they were mostly isolated to a few threads, which could perhaps be executed at higher precision.

To further understand where in a given thread we are having a loss in precision we collected the min/max exponent range of individual shader programs for every math operation executed. This information is shown in Figure 5.8. Two particular pixel shader programs are shown but they are representative of the many different pixel shader programs. In the first pixel shader program we can see that there is a large region of the program where even 4-bits of the exponent are not used. As a matter of fact there are only a few instructions, which use almost the entire range. The second program shown is similar except that there are more programs utilizing

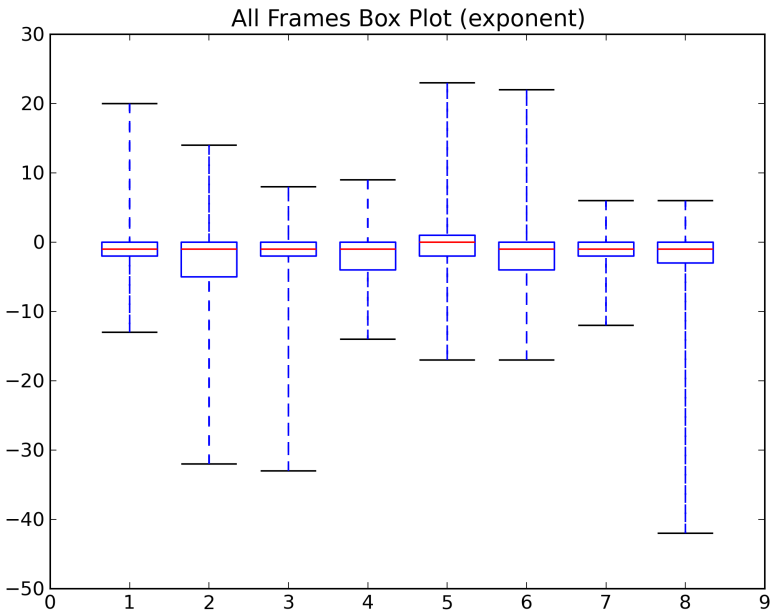


Figure 5.6: Exponent distribution for all benchmarks. The box represents the 25-75 percentile range. The red line is the median and the whiskers represent the 1-99 percentile range.

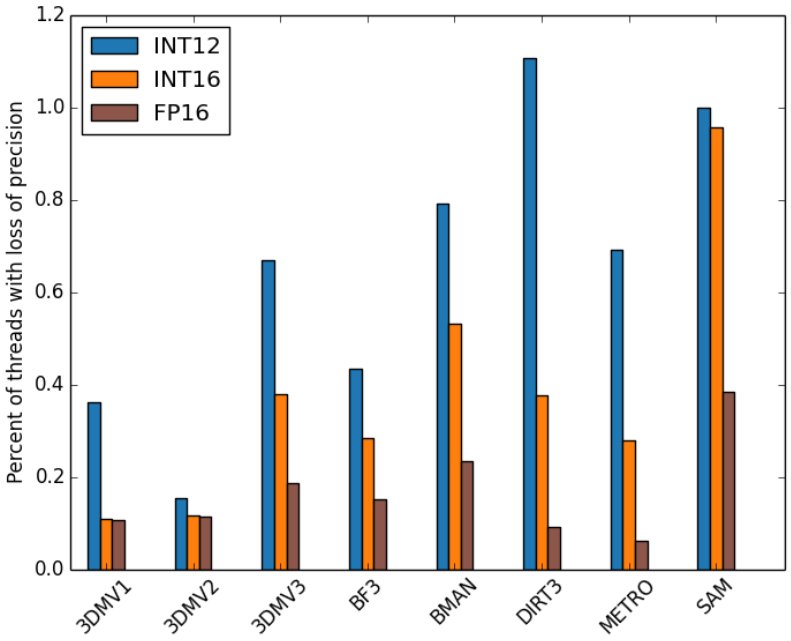


Figure 5.7: Shows the percentage of threads in which at-least one operation had an incorrect result because the dynamic range between the inputs was too large to be represented accurately with the specified precision

the entire range.

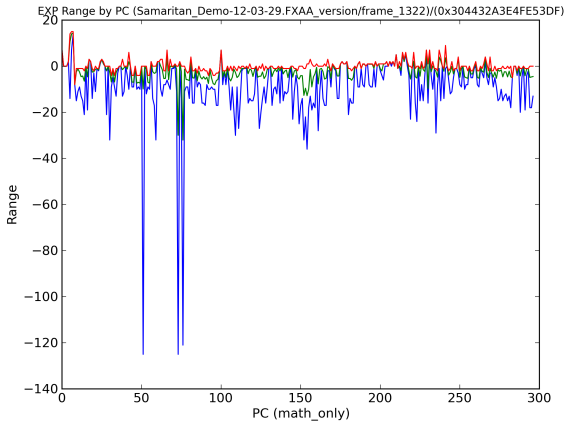
Overall, we saw that even though GPUs use 32-bit FPU arithmetic they use only a small part of that range in real pixel shader application. This can be exploited to yield significant power savings of 20-30 percent. These savings might be even larger if we can use a scheduler/control design specifically for lower precision math.

5.4 Overdraw in GPUs

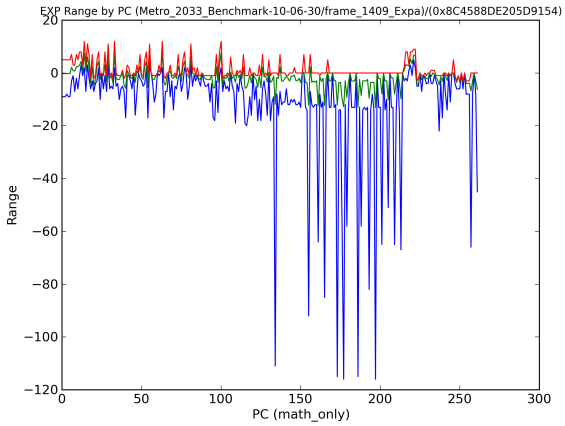
The next inefficiency we are interested in looking at is overdraw. Overdraw occurs when the GPU over writes a particular values in the frame buffer. This occurs for instances when you have an object that appears in front of another occluding a part of the background object. This problem is traditionally dealt with by using a technique called deferred rendering where all primitives are sorted to make sure they can be perfectly culled [33, 18, 30].

In most high performance GPUs, primitives are not sorted because this introduces a serialization point in the pipeline which means all generated objects need to be stored in intermediate storage before rendering. This is hard to do with the high primitive counts, which become larger with tessellation. Instead they rely on software rendering objects in front to back order for optimal performance. If objects are rendering in front to back order they can be culled based on primitives seen in the past without having to buffer all primitives and sort (a process known as Early-Z[32, 54, 45]). The GPU processes primitives in order to make sure that primitives that appear in the foreground correctly overwrite the frame buffer.

Overdraw is pure waste and can be reduced either in hardware or in software. Hardware solutions are complex and require primitive storage and sorting while software solutions pose their own challenges by requiring that applications keep track of object ordering. In order to understand how much effort to spend reducing overdraw we would like to quantify the potential energy savings of the instrumented shader programs. The instrumented shader programs wrote a side buffer everytime the frame buffer was touched for a given pixel. This allowed us to capture the hit mask for every pixel, for every pixel shader that wrote to the final frame buffer. Shader programs



(a) Temporal Exponent Range. (Samaritan Demo shader)



(b) Temporal Exponent Range (Metro 2033 shader)

Figure 5.8: Temporal Exponent Range. Blue shows the minimum, green the median and red the maximum. The x-axis shows math operations in the order they were performed by the application.

that had blending enabled in the ROP were not profiled and included in the hit mask. By definition blended programs can't have overdraw since the previous value of the frame buffer can have an impact on the results.

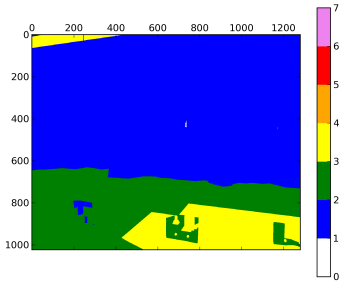
The results of capturing the shader program invocations for each pixel are shown in Figure 5.9. Metro 2033 hung the pixel profiler and is not included here. What we see here are really interesting results where the majority of frames overdraw pixels more than 2 times. Actually, it is apparent from the images that some complex parts of the screen are drawn several times while more background regions are drawn only once averaging to approximately 2.3X.

These initial results indicate a potential for large energy savings. To evaluate the energy savings we used detailed performance counters as well as information about which shader values were finally used to estimate energy. For example if shader program 1 wrote to the frame buffer and then subsequently shader program 2 writes to the same pixel on the frame buffer, we assume the work done by program 1 was waste. Also we took care to make sure an entire 2x2 pixel quad of work was eliminated before counting it as waste since the values would have to be used for texture (level of detail) LOD calculations[69] otherwise and would need to be rendered anyways. Looking closely at Figure 5.9 you can see this square blocking pattern for the quads.[2]

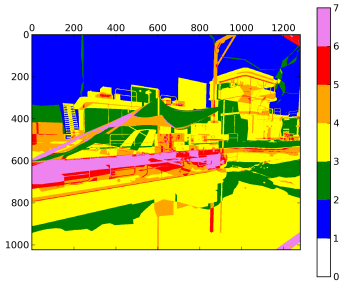
The computed energy savings are shown in Figure 5.10. Interestingly, the energy savings while significant are not as large as we expected them to be. The average energy reduction is just 15 percent while the average overdraw is 2.3 times. Looking through the actual shader programs it was pretty apparent that pixel shaders that were overdrawn generally were very short programs that consumed very little energy while the final program was generally much longer and more complicated.

While this may initially seem counterintuitive it actually makes sense. Overdraw is an inefficiency that can cause significant loss of performance because of all the extra work that it causes the GPU to do. In other words it is actually an important optimization criteria for game developers who work on these complex games. As a matter of fact there are tools such as NVIDIA PerfHUD [12] which help developers visualize

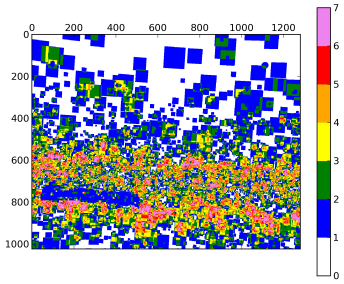
[2]Quads are a group of 2x2 pixels which are processed together. This grouping allows the graphics processor to calculate finite-difference derivatives, which are used to calculate texture level of detail (LOD)



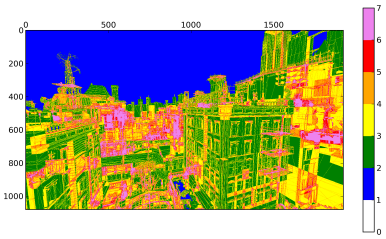
(a) 3MarkVantage Frame 1



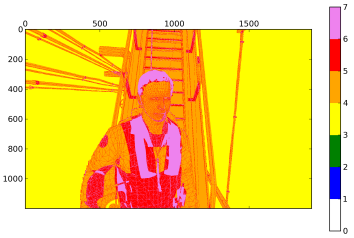
(b) 3MarkVantage Frame 2



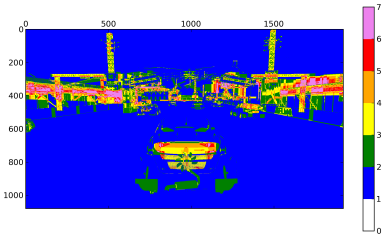
(c) 3MarkVantage Frame 3



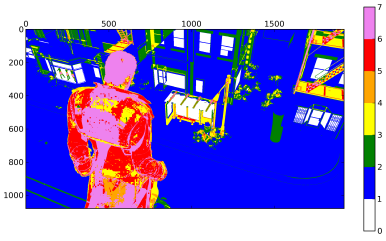
(d) Batman Arkham City Frame 1



(e) Battlefield 3 Frame 1



(f) Dirt3 Frame 1



(g) Samaritan Frame 1

Image Not Captured

(h) Metro 2033 Frame 1

Figure 5.9: Results showing overdraw for benchmark frames. The color shows the number of times a given pixel was drawn.

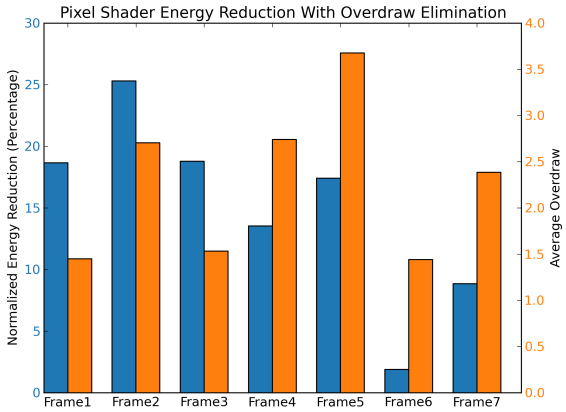


Figure 5.10: Overdraw Ratios and Energy Reduction

and reduce overdraw. Our guess here is that the developers optimized overdraw until they picked out all the low hanging fruit to get the maximum performance. This in turn means that they indirectly reduce the amount of wasted computation and hence energy.

Overall, while there is about 15 percent energy savings to be had, the case for overdraw elimination is not as compelling as we initially thought. Even if the savings are sligly higher when we consider other parts of the system such as the caches and DRAM, the most significant energy reductions can be made with just application or driver level shader program optimization that reduce the amount of overdraw done by more complex shader programs. So the best approach to achieve this scaling is through software and not hardware.

5.5 Understanding Thread Level Redundancy

The final shader optimization explores idling threads that share computation with other threads. In contemporary shader architecture, operations are performed over several pixels in parallel with each instruction. This amortizes the control overhead in a system where there is a lot of data parallelism, and creates a basic SIMD[26] execution engine. This engine is similar to that used to execute SIMD instructions in general purpose CPUs. In GPUs, however, there are several such SIMD cores with each SIMD core executing independent instructions streams. This is referred to as Multiple Instruction Multiple Data (MIMD) [17, 43]. Also, GPUs generally have much more relaxed constraints about memory access patterns in SIMD and generally allow words to be swizzled between operations.

This architecture allows GPU shaders to achieve a high compute density of 32-math operations/instruction. However, we found that in many cases the shader performs more work than necessary to produce the correct result. In these cases threads within a given warp were operating on the exact same data and producing identical results. If there was some way to identify that similar operands were present we could turn off some of the datapaths and save energy while still computing all the values that we need.

To quantify the energy saving we profiled the output results of all the math operations performed by the shader using the methodology shown in Section 3.4 and used our energy model to compute the potential energy savings. From this profile we computed the ratio of the unique operations performed in each step of the warp execution and the total operations performed. The total number of operations performed by the warp in each cycle might not be 32 since some lanes in the datapath may be shut off due to predication.

To calculate the energy we applied the energy model and scaled the register file and datapath energy based on the operations being scalarized. Figure 5.11, shows an example of what the energy would look like for a shader performing FMA operations. The Y-intercept is roughly the control energy for each instruction. Thus one should note that while the energy decreases when there are fewer active threads in a given

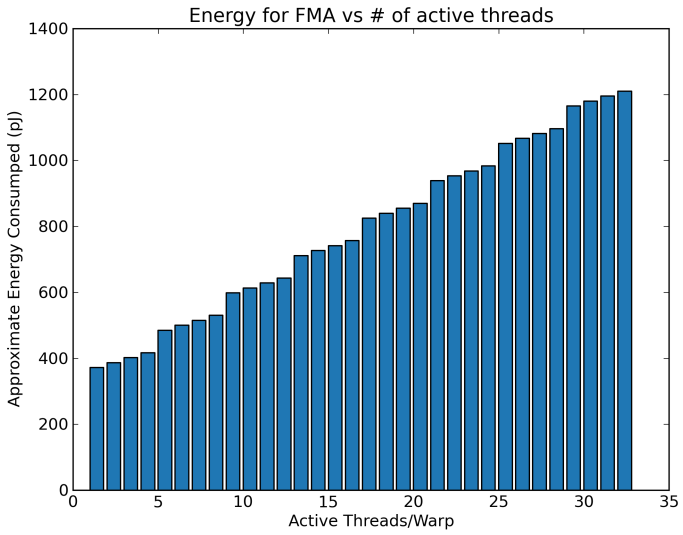
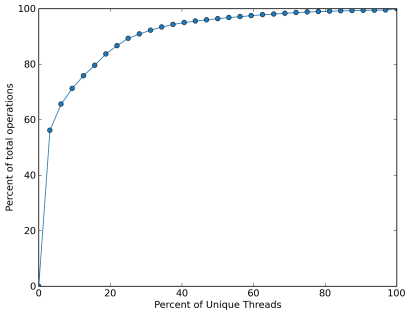


Figure 5.11: Theoretical energy scaling for performing FMA operations across a shader unit.

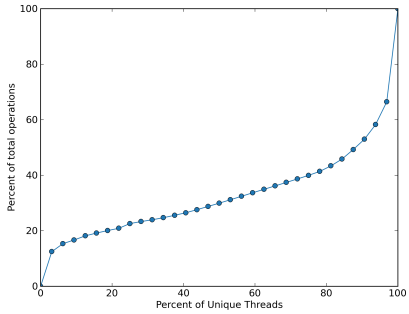
warp the efficiency also reduces. This is since every math operations performed helps amortize the control overhead.

Some sample results generated by mining the datapath values are shown in Figure 5.12. Each cumulative histogram shows percentage operations that required less than a certain percentage of the available threads in a given warp. This plot is always 100 percent at 100 percent of the threads. In general most warps exhibit a couple of patterns. One pattern as can be seen in (b, c, d) is where we have a bi-modal distribution, threads are either totally unique or not unique at all, with very few cases in between. The other common pattern is where we have almost every thread doing redundant computation, as shown in (a). One reason this might be happening is due to a lack of a pure scalar unit in the GPUs we used. The DX11 specification provides a scalar shader abstraction and some GPUs provide scalar shader hardware [64].

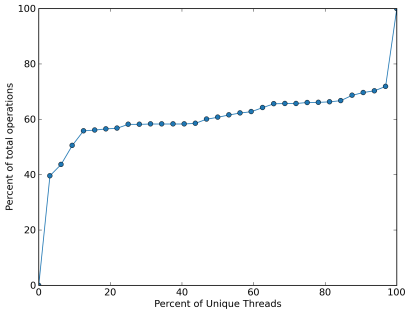
Using the energy models and thread scalarization information (for example: Figure 5.12), we can compute the energy savings and breakdowns. These results are shown in Figure 5.14. As expected, we only see changes in energy of the functional units



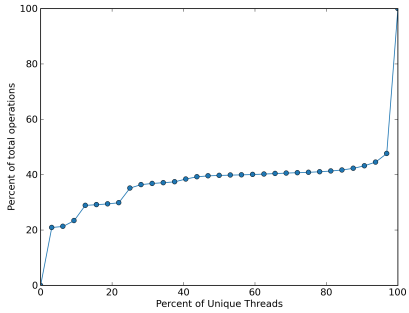
(a) Battlefield 3 (select shader program)



(b) Metro 2033 (select shader program)

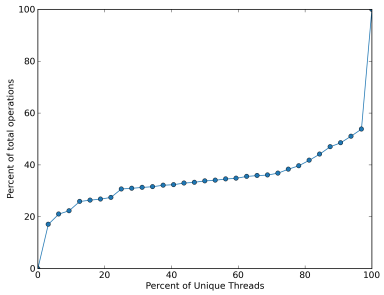


(c) Metro 2033 (select shader program)

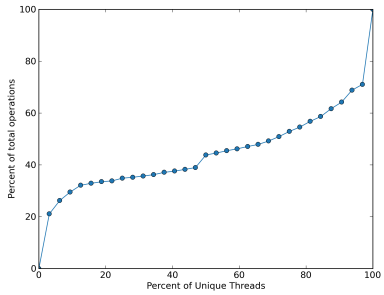


(d) Samaritan Demo (select shader program)

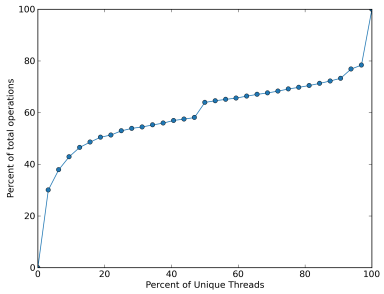
Figure 5.12: Cumulative distribution of uniqueness in shader programs. The x-axis lists the percent of total threads that are unique in a given 32-wide thread group (warp). The y-axis is the percent of total warps that have uniqueness below the value specified by the x-axis. These select shader programs were selected to show the different distributions among various different use cases.



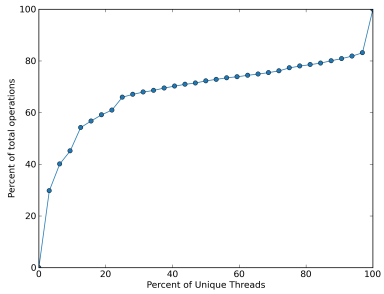
(a) 3MarkVantage (Frame 1)



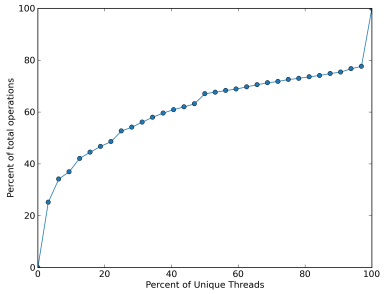
(b) 3MarkVantage (Frame 2)



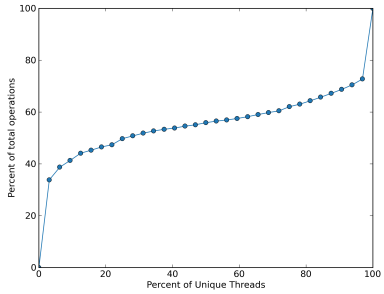
(c) 3MarkVantage (Frame 3)



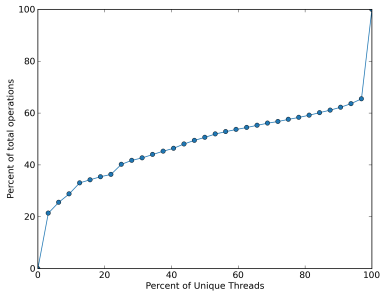
(d) Batman Arkham City (Frame 4)



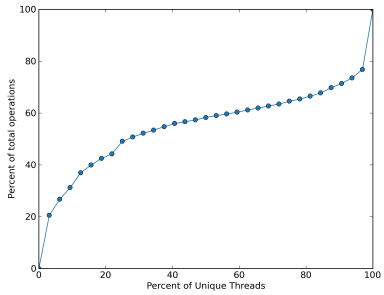
(e) Battlefield 3 (Frame 5)



(f) Dirt3 (Frame 6)



(g) Samaritan (Frame 8)



(h) Metro 2033 (Frame 7)

Figure 5.13: Application level aggregates showing the cumulative distribution of the number of percent of unique threads in a warp (32-wide thread grouping). We can see that most of the applications have only about 50% unique threads when looking at 50% of the total warps executed.

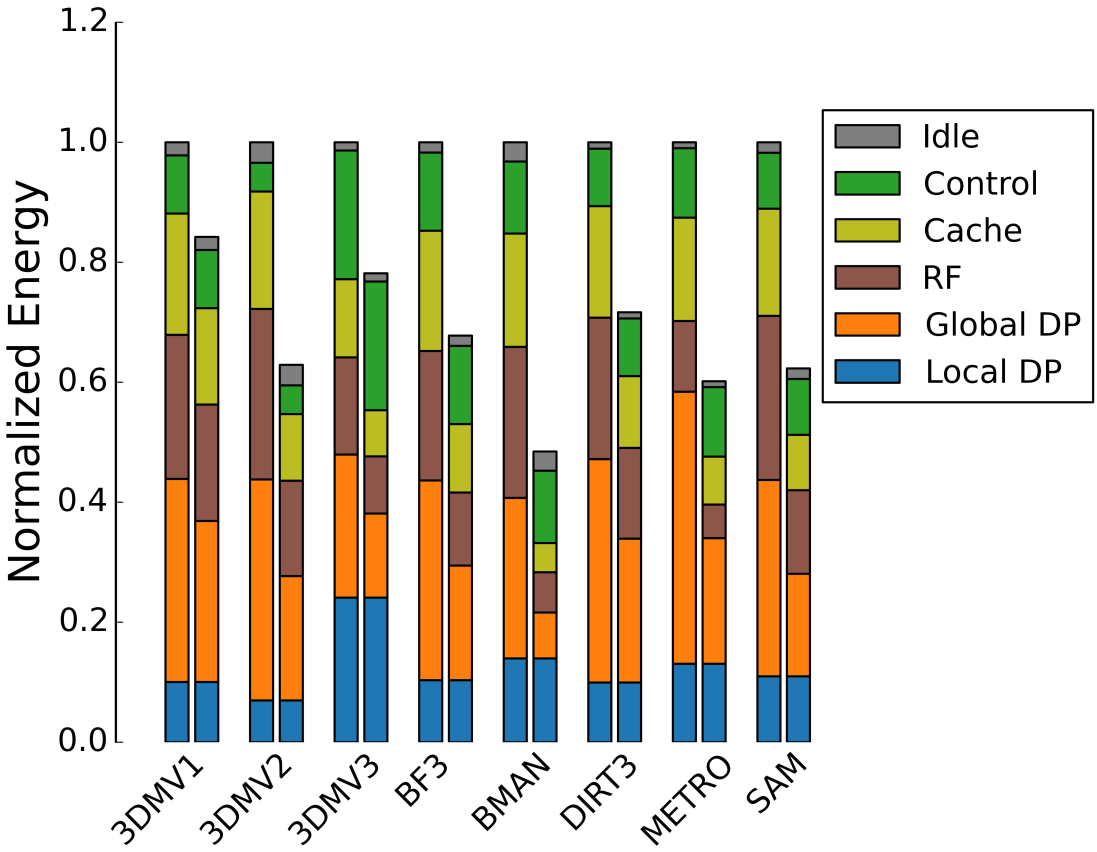


Figure 5.14: Potential energy reduction with scalarizing the shader. Left bar is original and right bar is the energy with all thread level redundancy eliminated

and register file and not the control logic. Interestingly, there is a fair amount of energy spent on duplicate data. The energy savings range from about 15% all the way to almost 50%. The large amount of thread level redundancy and energy savings initially was a surprise, since we have these 3D images with complex lighting. Also, if the energy savings are so large why haven't developers optimized their applications?

Unlike the previous study on overdraw, redundancy in operations does not affect performance. As a result there are fewer developer tools to identify thread level redundancy and due to the lack of any performance gains, little incentive to make improvements. Yet this still leaves the question of why this occurs at all.

Upon closer examination we found that a lot of energy savings result from the computation of intermediate draw calls, for example: shadow mask and stencil computations. This means that while the final image may have different values for adjacent pixels there could have been a lot of shared intermediate computation.

Now that we have seen that there is a lot of opportunity from scalarization we can explore a few possible solutions that we might be able to use to save energy. The first thing we looked at was what percent of the datapaths produced a constant value such as 0.

As shown in Figure 5.15 a significant percent of the data being operated on (5-15%) is zero (or one of the degenerate zero cases in floating point). This turns out to be fairly common and it is probably at least partially related to a lot of GPU operations having a saturate at zero attribute. Knowing this information we can think of a simple architectural changes to exploit this redundancy. For example, we can tag the register file with a bit that indicates that the value is zero. When we read the register file we can use this attribute to gate all the downstream logic and prevent energy waste.

Another area that we can consider looking at is possible compiler optimization that might help find regions of code with thread level redundancy. To understand the program behavior we plotted the uniqueness vs the sequence of program execution. This is shown for a few sample program in Fig 5.16. The figure shows some sample shader programs that exhibit different classes of behaviors. A set that has unique operations scattered throughout the entire program and another set that has

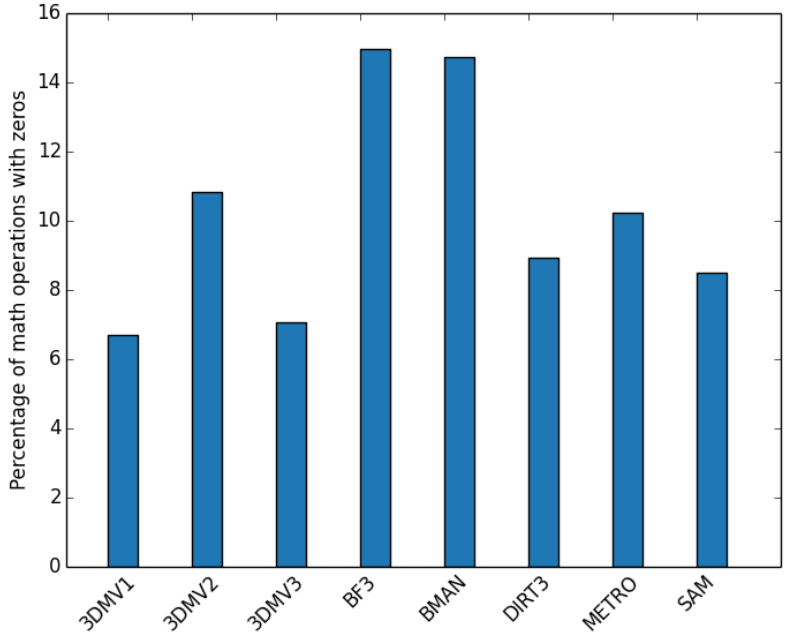
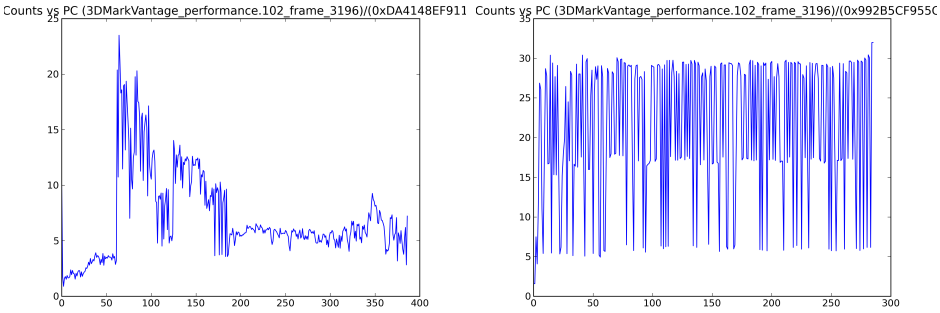


Figure 5.15: Shows the percentage of math operations where the result was a zero output.



(a) 3DMark (Frame 2, select shader program)

(b) 3DMark (Frame 2, select shader program)

Figure 5.16: Uniqueness vs sequence of program execution for select shader programs. The shader program in (a) has unique operations in a small region of the program. The shader program (b) has unique values throughout the program

a small part of the shader program containing a large percentage of unique values. This information can potentially be used to do profile driven optimization where the shader can tag relevant register file entries by performing comparisons during fairly redundant parts of the program and performing all computations when values are likely to be unique.

Looking at the holistic view of both performance and energy data we see that there is a significant opportunity to save energy by scalarizing certain parts of the shader program. While the actual implementation of an energy reduction technique is out of scope of this discussion there is an opportunity of between 15-40% depending on the application.

5.6 Conclusions

In this chapter we showed how several studies can be performed by having a flexible energy model along with relevant performance data. The shader precision studied showed that a significant reduction in energy is possible by reducing the precision at which operations need to be performed. Application traces showed that the majority of shader operations don't require the level of precision provided by a single precision

floating point unit. While further study is needed regarding what artifacts will be created or if the precision can be selected by the compiler, initial results are promising.

When studying overdraw reduction we saw that there was significant overdraw which yielded only 15% energy saving. We suspect that this is due to the fact that overdraw affects performance and software level performance optimization improved energy efficiency too. Further software optimization should be able to recover more energy savings without hardware changes. The energy issues that don't affect performance, like scalarization, have more potential for energy savings. Here we found 15-50% energy saving is possible. Taken together there is significant performance improvement possible at a constant power by implementing the energy savings discussed in this chapter.

Chapter 6

Conclusions

Since the power wall limits GPUs performance scaling, further increases in GPU performance will require improvements in energy efficiency. In this dissertation we extend current energy and performance modeling tools and optimization methodologies that allowed us to explore both the GPU architecture design space and graphics specific optimizations. As part of this framework we created a hierarchical energy model that contains both regressions and building block based models for units of the GPU. We also created tools and explored methodologies for understanding application behavior and it's impact on energy consumption. These models were then used to estimate the performance and energy of a contemporary GPU architecture.

In the process of creating these models we gained valuable insights into what is necessary to create performance/energy models for a GPU. GPUs are complex designs with many different functional units and a fairly large and complex software stack. We explored building the models based on performance counters, which are externally visible/accessible by the software. While we can obtain a fair amount of modeling accuracy using just externally visible performance counters, the flexibility of this model is limited. We found that using internal data about the architecture and circuits used in the actual design were instrumental in creating both an accurate and flexible model. We also gained a lot of insight when examining the workloads of the GPU. Initially when we wanted to see internal values in the datapaths we were inclined to use the architectural simulator. This turned out to be fairly slow compared

to just using the actual silicon and instrumenting the shader programs to write the values to a section of the memory, which could later be downloaded. Furthermore, some aggregated frame level data could also be computed without having to deal with the massive amount of data that is processed for each frame. For experiments where aggregated values were not enough we used sampling to make data volume manageable.

While creating these models we wanted to both focus on interesting regions of the design from an energy efficiency perspective and also get detailed breakdowns about where energy is spent. We opted to use a hierarchical energy models. We also made some models regression based and others based almost entirely on building blocks (such as floating point unit, register file, etc). This distinction allowed us to focus our time on modeling high impact regions of the design and regions of the design that we were interested in exploring.

Accuracy of the model is extremely important when using it to make design tradeoffs so we compared our model to both existing silicon as well as detailed gate level simulations. We showed that the generated models are good predictors of GPU energy consumptions by comparing them across various graphics benchmarks. Having a good model we conducted several potential energy optimizations.

The first study that we performed examined the data values in the shader data path for pixel shader programs. We determined that the vast majority of operations don't require the precision offered by the single precision floating point units that are used by current GPUs. We did a "what if" study where we examined the potential energy savings if we were to perform some operations with lower precision. We found that this optimization can yield 20-30% energy savings in current GPUs.

We then studied the impact of overdraw on the energy efficiency of GPUs. Overdraw occurs when a given pixel is drawn more than once because a later processed primitive overwrites the previous value at a given location. Interestingly, we found that there is significant amounts of overdraw in most of the frames that we studied. However, the energy savings are less than 15% even with an over draw of 2.3X (meaning that every pixel on average is drawn more than 2 times). While, initially surprising, this resulting low energy savings makes sense. Overdraw has a significant

impact on performance and software developers have tools that help with overdraw elimination. Even with the significant overdraw only small shader programs are getting rerun while the larger shader programs have been optimized to run only once for a given pixel.

The last study we looked at tried to understand if we can share computation across thread groups (warps) in a shader processor. It turns out this actually yields significant energy savings of as much as 50%. Relatively few tools exists for software developers to see what the thread level redundancy is, hence, few applications are actually optimized to reduce it.

The studies showcase the importance of holistically looking at the performance and energy models. Looking at just one alone might give misleading results, as it was evident from the scalarization and overdraw studies. Overall, our methodology allows for exploration of GPU design space, which can be used to improve the energy efficiency, which in turn will yield higher performance GPUs.

Bibliography

[1] Tomas Akenine-Möller and Jacob Ström. Graphics for the masses: a hardware rasterization architecture for mobile phones. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 801–808, New York, NY, USA, 2003. ACM.

[2] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. *SIGARCH Comput. Archit. News*, 38(3):26–36, June 2010.

[3] D. H. Bailey and P. N. Swarztrauber. The fractional Fourier transform and applications. *SIAM Rev.*, 33(3):389–404, 1991.

[4] Himanshu Bhatnagar. *Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler® Physical Compiler® and PrimeTime®*. Springer, 2002.

[5] Chas Boyd. The directx 11 compute shader. *ACM SIGGRAPH 2008 classes*, 2008.

[6] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, May 2000.

[7] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *Micro, IEEE*, 20(6):26–44, 2000.

[8] Cadence. Cadence palladium technical brief, 2010.

[9] Zhongliang Chen, David Kaeli, and Norman Rubin. Characterizing scalar opportunities in gpgpu applications. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 225–234. IEEE, 2013.

[10] Brett W Coon, John Erik Lindholm, Samuel Liu, Stuart F Oberman, and Ming Y Siu. Operand collector architecture, November 16 2010. US Patent 7,834,881.

[11] NVIDIA Corporation. Nvidia parallel nsight.

[12] NVIDIA Corporation. Nvidia: Nvidia perfhud version 5.1, 2007.

[13] NVIDIA Corporation. Nvidias next generation cuda compute architecture: Kepler gk110, 2012.

[14] J Dahl and L Vandenberghe. Cvxopt, 2007.

[15] V.M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and Espasa E. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 231 – 241, march 2006.

[16] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[17] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. 2009.

[18] Jerome F Duluk Jr, Richard E Hessel, Vaughn T Arnold, Jack Benkual, Joseph P Bratt, George Cuan, Stephen L Dodgen, Emerson S Fang, Zhaoyu Gong, Y Yo Thomas, et al. Deferred shading graphics pipeline processor having advanced features, October 5 2010. US Patent 7,808,503.

[19] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *Computers, IEEE Transactions on*, 60(7):913–922, 2011.

[20] Sameh Galal and Mark Horowitz. Energy-efficient floating-point unit design. *IEEE Trans. Comput.*, 60(7):913–922, July 2011.

[21] M. Gebhart, D.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. *ACM SIGARCH Computer Architecture News*, 39(3):235–246, 2011.

[22] Andrew Glassner and Henry Fuchs. Hardware enhancements for raster graphics. In *Fundamental Algorithms for Computer Graphics*, pages 631–658. Springer, 1991.

[23] Eric Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis*, pages 234–259. Springer, 2001.

[24] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 108–120, New York, NY, USA, 1997. ACM.

[25] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010.

[26] Mark Harris. Mapping computational concepts to gpus. In *ACM SIGGRAPH 2005 Courses*, page 50. ACM, 2005.

[27] F Hill and S Kelley. *Computer Graphics Using OpenGL, 3/E*. Pearson, 2007.

[28] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. *SIGARCH Comput. Archit. News*, 38(3):280–289, June 2010.

[29] Libo Huang, Li Shen, Kui Dai, and Zhiying Wang. A new architecture for multiple-precision floating-point multiply-add fused unit design. In *Computer*

Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on, pages 69–76. IEEE, 2007.

[30] Josh Klint. Deferred rendering in leadwerks engine. *Copyright Leadwerks Corporation*, 2008.

[31] Scott J Krieder. An overview of current and future accelerator architectures.

[32] Jens Kruger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society, 2003.

[33] Andrew Lauritzen. Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading*, 2010.

[34] B.C. Lee and D.M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 185–194. ACM, 2006.

[35] S. Lee, A. Ermedahl, S.L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded risc processors. In *ACM SIGPLAN Notices*, volume 36, pages 1–10. ACM, 2001.

[36] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 487–498. ACM, 2013.

[37] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[38] E Lindholm, H Moreton, J Montrym, and S Whitman. Apparatus and method for raster tile coalescing, 06 2009.

[39] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 –55, march-april 2008.

[40] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96 –100, feb. 2007.

[41] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[42] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1061–1079, 1998.

[43] Ogier Maitre. Understanding nvidia gpgpu hardware. In *Massively Parallel Evolutionary Computation on GPGPUs*, pages 15–34. Springer, 2013.

[44] J. Montrym and H. Moreton. The geforce 6800. *Micro, IEEE*, 25(2):41 – 51, march-april 2005.

[45] Stephen L Morein. System, method, and apparatus for early culling, February 14 2006. US Patent 6,999,076.

[46] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. Shader performance analysis on a modern gpu architecture. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 10 pp. –364, nov. 2005.

[47] V. Moya, C. González, J. Roca, A. Fernández, and R. Espasa. A single (unified) shader gpu microarchitecture for embedded systems. *High Performance Embedded Architectures and Compilers*, pages 286–301, 2005.

[48] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[49] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, first edition, 2007.

[50] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56 –69, march-april 2010.

[51] K. Nose and T. Sakurai. Analysis and future trend of short-circuit power. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(9):1023–1030, 2000.

[52] J. Owens. Gpu architecture overview. In *ACM SIGGRAPH*, volume 1, pages 5–9, 2007.

[53] J.T. Russell and M.F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, pages 328–333. IEEE, 1998.

[54] Pedro V Sander, David Gosselin, and Jason L Mitchell. Real-time skin rendering on graphics hardware. In *the proceedings of SIGGRAPH*, 2004.

[55] T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago. Evaluation of architecture-level power estimation for cmos risc processors. In *Low Power Electronics, 1995., IEEE Symposium on*, pages 44–45. IEEE, 1995.

[56] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, et al. Larrabee: A many-core x86 architecture for visual computing. *IEEE micro*, 29(1):10–21, 2009.

[57] A. Sharif and H.H.S. Lee. Total recall: a debugging framework for gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 13–20. Eurographics Association, 2008.

[58] Allen Sherrod. *Beginning DirectX 11 game programming*. Cengage Learning, 2012.

[59] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc. of PATMOS*. Citeseer, 2001.

[60] VCS Synopsys. Verilog simulator, 2004.

[61] T. Tamasi. Evolution of computer graphics. *Proc. NVISION*, 8, 2008.

[62] D. Tarjan, S. Thoziyoor, and N.P. Jouppi. Cacti 4.0. *HP laboratories, Technical report*, 2006.

[63] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[64] Alexandre Valdetaro, Gustavo Nunes, Alberto Raposo, Bruno Feijó, and R de Toledo. Understanding shader model 5.0 with directx11. In *IX Brazilian symposium on computer games and digital entertainment*, volume 1, page 13, 2010.

[65] Andries Van Dam, Steven K Feiner, Morgan McGuire, and David F Sklar. *Computer graphics: principles and practice*. Pearson Education, 2013.

[66] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 95–106, New York, NY, USA, 2000. ACM.

[67] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 214–222. ACM, 1977.

[68] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21:1–1874919424, 2011.

[69] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50 –59, march-april 2011.

[70] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[71] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 340–345, New York, NY, USA, 2000. ACM.

[72] Gordon Yip. Expanding the synopsys primetime® solution with power analysis. *[Online document] June*, 2006.

[73] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. 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, july 2011.