Copyright

by

Mrinal Deo

2011

The Report committee for Mrinal Deo

Certifies that this is the approved version of the following Report

A MEMORY PROFILER FOR 3D GRAPHICS APPLICATIONS USING BINARY INSTRUMENTATION

APPROVED BY
SUPERVISING COMMITTEE:

Donald S. Fussell, Supervisor

Mattan Erez

A MEMORY PROFILER FOR 3D GRAPHICS APPLICATIONS USING BINARY INSTRUMENTATION

by

Mrinal Deo, B.E

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2011

Dedicated to my wife Shobhna, for her love and support.

A MEMORY PROFILER FOR 3D GRAPHICS APPLICATIONS USING BINARY INSTRUMENTATION

Mrinal Deo, M.S. E.

The University of Texas at Austin, 2011

Supervisor: Donald S. Fussell

This report describes the architecture and implementation of a memory profiler for 3D graphics applications. The memory profiling is done for parts of the program which runs on the graphics processor and is responsible for rendering the image. The shaders are parsed and every memory instruction is instrumented with additional instruction for profiling. The results are then transferred from the video memory to CPU memory. Profiling is done for a frame and completes in less than three minutes. The report also describes various analyses that can be done using the results obtained from this profiler. The report discusses the design of an analytical cache model that can be used to identify candidate memory buffers suitable for caching among all the buffers used by an application. The profiler can segregate results for reads and writes separately, can handle all formats of texture access instructions and predicated instructions.

Table of Contents

LIST OF TABLES ..... **VII**

LIST OF FIGURES ..... **VIII**

CHAPTER 1. INTRODUCTION **1**

CHAPTER 2. HETEROGENEOUS PLATFORMS **2**

2.1 Architecture 2

2.2 Platform Design Choices 2

2.3 Platform Design Tools 4

CHAPTER 3. 3D GAMES AS A WORKLOAD **6**

3.1 DirectX pipeline 6

3.2 3D games memory usage pattern 7

CHAPTER 4. MEMORY PROFILER **9**

4.1 Motivation 9

4.2 DirectX driver workflow 10

4.3 Memory profiler design 11

4.3.1 Memory Requirement and Allocation 11

4.3.2 Instrumenting shader code. 13

4.3.3 Performance of the tool 20

CHAPTER 5. RESULTS **21**

5.1 Verification 21

5.2 Results 22

5.3 Analytical Cache Model 26

CHAPTER 6. CONCLUSION **29**

6.1 Summary & Future work 29

BIBLIOGRAPHY **31**

LIST OF TABLES

Table 1. Number of Draw Calls for chosen frames in each game. 8

Table 2. List of various forms of Texture access instruction supported by ISA. ..17

Table 3: List of various forms of Load and Store instruction supported by ISA...18

LIST OF FIGURES

Figure 1: Block diagram of current generation of Heterogeneous Architectures. ...4

Figure 2: A 1920x1200 surface divided into tiles. 13

Figure 3: function prototype of Direct3D API to create Shader Object. 13

Figure 4: Sample texture and predicated instruction. 15

Figure 5. A sample code of instrumented Pixel Shader. 19

Figure 6: Render Target for DirectX SDK sample 23

Figure 7: Texture used in BasicHLSL.. 23

Figure 8: Surface representing the access for RT and ZB. 24

Figure 9: Surface representing the access for Texture 24

Figure 10. Sample Frame from Unigine 2.1 25

Figure 11. Memory access per tile for one of the surface used in Unigine 25

Figure 12: Design of analytical cache model. 27

Figure 13: Results using our analytical cache model for Unigine Heaven. 28

CHAPTER 1

INTRODUCTION

It is getting increasingly difficult for CPU designers to provide the performance gains that the industry has seen in the past three decades. Therefore architects are looking at multi cores to continue to provide the performance gain. There is renewed interest in the field of parallel computer architecture. However, this time around the approach is slightly different and the solution space includes a new architecture, the Graphics Processing Unit (GPU), which has proven to be very successful in providing performance gains for a certain types of workloads. CPU designers now want to make use of the computation power provided by the GPU by incorporating it into the same die. The resultant architecture is loosely referred as the Heterogeneous architecture comprising of a CPU, a GPU and the glue logic, all on the same die. The design and analysis of heterogeneous architectures present new set of challenges that demands new set of tools and methodologies.

In this report I present the design and implementation of one such tool which is targeted towards getting insights into the memory access patterns of 3D games running in the GPU. This tool gathers the memory footprint and access patterns of the portion that is responsible for rendering images and runs on the GPU. This is the portion that now needs to interact with a unified memory architecture serving the needs of a GPU and a CPU. The two main components, CPU and GPU have evolved in a separate direction owing to the different nature of the workloads and now there is a need to find the commonality between the two if they are to co-exist efficiently on the same die. I also present some initial results using this tool and provide insights as how to use this tool to answer difficult questions faced architects.

CHAPTER 2

HETEROGENEOUS PLATFORMS

2.1 ARCHITECTURE

Let us look at examples of current designs that incorporate a CPU and GPU on the same die. These designs are now readily available from Intel and AMD. The current set of designs feature a CPU, a GPU but they noticeably miss any advanced glue logic. By glue logic I refer to the interconnect between the two chips and logic which will make the system homogenous. The programming model has not changed to reflect one homogenous system. CPU works as the master which feeds the command and data to the GPU. There is no hardware support for coherency between the two systems. One important shortcoming of these designs is the fact that the CPU and GPU do not share the same address space even though they share the same physical memory. I would however also like to mention the fact that this is not a problem that can be solved by the hardware designs alone. We may need changes in the entire software stack from the operating system to the applications.

2.2 PLATFORM DESIGN CHOICES

There are many ways in which we can chose to integrate the CPU and the GPU on the same die. Figure 1 shows two such example implementation made by AMD and Intel respectively. For the GPU to provide a computational horsepower of 1 TFLOP it needs to be fed by a very high bandwidth memory interface. This presents an interesting challenge to the designers who would like to put the GPU on the same die. First choice they need to

make is whether the unified memory interface should look more like the GPU interface or the CPU interface or should it a hybrid design catering to the need of both the cores. A CPU has a memory bandwidth interface with an order of magnitude less bandwidth. If we want high bandwidth we can choose GDDR5 as the memory interface for the entire die replacing the DDR3. This however has not been chosen by either design house because GDDR5 is more power hungry and is not scalable in terms of capacity that each chip can provide. Now, given that both the design houses have incorporated DDR3 as the memory interface, let us shift our focus to another important component of the memory subsystem of a chip, namely on chip memory or caches. We know that each CPU today comes with large caches with capacities ranging from a MB to a few MB. Ideally the cache memory available on die can act as bandwidth amplifier and take some pressure off the DDR interface. But when we look at the two block diagrams we see that the AMD design has not chosen to share the last level cache with the GPU. We try to understand what is the rationale behind the different choice made when it comes to cache sharing in these two implementations? Traditionally GPUs have had very small amount of specialized caches with capacity in KB built into the chip and the design might need some modifications if it needs to be able to work with a large cache efficiently. The designers also need to keep in mind that the main purpose of the GPU, even today, is to render images in 3D games or applications written using APIs like DirectX or OpenGL and these applications do not assume a presence of large caches on chip. The GPUs have traditionally been designed to tolerate cache miss and the resultant long latency fetches. An emerging class of applications targeted at GPUs is data parallel applications which would benefit from large caches. For these GPGPU [1] applications cache efficient implementations exist by the virtue of the fact that they were being optimized for CPU platforms up until today.

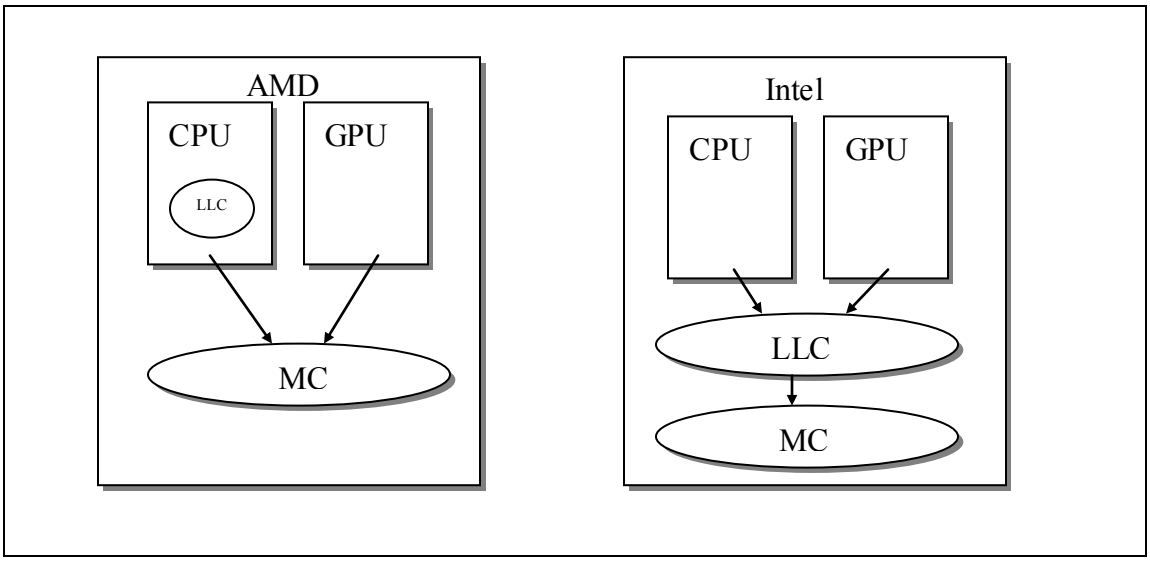

Figure 1: Block diagram of current generation of Heterogeneous Architectures.

2.3 PLATFORM DESIGN TOOLS

If we were to influence the architecture and design of future chips we need to answer few specific questions. We need to understand which of the two options for cache sharing is optimal. Measuring performance on these chips will not provide the answer as the capabilities of the individual components (CPU and GPU) are very different on these two chips. We need to analyze the behavior of different workloads in this unified memory subsystem. The performance of 3D games is of paramount importance to GPU designers. Therefore we need to understand the behavior of 3D games in a unified memory subsystem. Specifically, we need to find the answer to the question - Can 3D games benefit from large caches? What is the size of the cache that is optimal? Does the size depend on factors like screen size? To be able to answer these questions we need specialized tools.

3D games have long been described as a ‘streaming’ workload. I use the word ‘streaming’ to denote the use-once behavior where ‘buffers’ are read or written to only once. I use the term ‘buffer’ to denote these surfaces Render target, Z-Buffer and Texture. I use the word ‘buffer’ and ‘surfaces’ interchangeably as from a driver perspective all these are buffers that need memory to be reserved. However we do not know for sure if this is true for current generation of games. We need to investigate if there is anything inherent in the nature of 3D applications that that makes it not conducive for caching. Or we can approach this problem in a different way. We know that applications running on the CPU generally benefit from larger caches and so does GPGPU style of applications. Given this, the question to answer would be - Can 3D games make good use of caches if they are made available? To tackle this broad problem I suggest that we should first evaluate the 3D graphics workloads in isolation and as a next step worry about the interplay between the CPU and the GPU traffic. The scope of this report is to provide tools and methodologies to categorize and understand 3D games as a workload in isolation.

CHAPTER 3

3D GAMES AS A WORKLOAD

3.1 DIRECTX PIPELINE

DirectX [2] is a z-buffer based pipeline. The different stages of DirectX pipeline are:

1. Input Assembler: Reads in vertex data from an application supplied vertex buffer and feeds down the pipeline.
2. Vertex Shader: Performs operation on a single vertex at a time, such as transformations skinning and lighting
3. Geometry Shader: Can write out the previous stage's result to memory. This is useful to re-circulate data back into the pipeline.
4. Rasterizer: Converts the primitives into pixels, feeding these into the pixel shader. The Rasterizer may also perform other tasks such as clipping and interpolation.
5. Pixel Shader: Works on each pixel as input from the previous stage. Computes lighting equations specified by the user. This typically involves looking up textures and other buffers.
6. Output merger: Merges the various types of output data (blending) to produce the final result.

All objects that need to be rendered are represented as a triangle mesh. Each of these objects or triangle mesh goes through the basic steps described above. Though this pipeline does not suggest much reuse of data but many rendering algorithms work in a producer-consumer fashion. This producer-consumer communication is typically

modeled as different pipeline pass or Draw calls, where one pipeline pass produces the data and another pipeline pass uses the data. One classic example of such algorithm is shadow mapping [4] which is used in all 3D games. And whenever we come across such producer-consumer patterns in applications we can try to use on chip caches to store the output from the producer.

3.2 3D GAMES MEMORY USAGE PATTERN

Every pipeline pass in a Direct3D application is identified by calls eg. DrawPrimitive() or DrawIndexedPrimitive(). I started my investigations by collecting data to substantiate the theory from section 2.1. I instrumented the Nvidia Corp. DirectX driver to count the number of pipeline passes (draw calls) needed to render a frame in a DirectX game. Table 2 provides some typical numbers for some chosen frames from popular games and benchmarks.

The total number of surfaces allocated by each frame is 100 on an average across the games in Table 2. Each draw call works on certain surfaces or buffers which needs memory to be allocated for them. The driver is responsible for all the memory allocation. It makes an attempt to allocate the buffer in video memory whenever possible. Some buffers are read-only e.g. textures whereas others are both read from and written to. To understand the memory reuse patterns of 3D games we look at read-only reuse, write-only reuse and read-write reuse i.e. essentially a producer-consumer model.

Before a DrawPrimitive() command is pushed into the GPU the application is required to set up the state corresponding to that Draw call. It is also required to attach the Render Target, Z-Buffer, Textures and other buffers needed by the Draw call. I dumped the surfaces or buffers for each Draw call as they were sent down the pipe from the application. Each surface is uniquely identified by an 'id' – a handle needed by the driver. I counted the reuse of each surface individually for a frame. The numbers where encouraging as few surfaces being "used" close to 500 times. However, one important point to realize is that count of surface reuse means very little. Even if a draw calls

continues to render to a surface, it could actually be rendering to different portions of the buffer in each of the 500 Draw calls in which it uses this surface. This information even though encouraging can mislead.

Game/Benchmarks	**Number of Draw Calls**
3D Mark 11 (GT1)	2300
3D Mark 11(GT2)	2109
Unigine 2.1	2488
Crysis 2	1610
Dirt2	1880

Table 1. Number of Draw Calls for chosen frames in each game.

CHAPTER 4

MEMORY PROFILER

4.1 MOTIVATION

After having conducted experiments described in chapter 2 it lead me to believe that 1) there is an opportunity for some bandwidth saving due to memory buffer reuse and 2) all the experiments that I conducted had some shortcomings and what we needed is a holistic approach to tackle this problem.

What would be really nice to achieve is to be able to categorize the memory used by any 3d games into streaming and non-streaming. Streaming memory is defined as a piece of memory that is used only once by the applications. Non-streaming is a piece of memory that is used (read or written) more than once. If all the memory the application uses is of streaming nature then all we need is a small amount of cache to exploit spatial locality in access. There is no temporal locality which warrants a need for large caches. In fact all graphics architectures already incorporate small amount of cache storage primarily to exploit spatial locality. There is temporal reuse assumed only for small buffers e.g. Index and vertex buffers which can fit into the small (few hundred KB) caches.

The requirements suggest that we profile the 3D game for memory accesses. There are different approaches that we can take to profile an application. I used the ideas on the line of binary instrumentation [4]. Another requirement for the tool was to be able to collect this information at a finer granularity as collecting information on a per surface basis is not sufficient. If we say that a particular surface X is used N number of times it does not guarantee a reuse of the same bytes. To further clarify, let us say the surface was a Render Target representing the screen, there can be different object drawn at different

coordinates of the screen and in this case even though we use the same surface we are actually not using the same memory location. This argument holds for all kinds of buffers. This may suggest that we need to collect information on a per word (pixel or texel) basis. This approach although most accurate is however an overkill and would require a lot of memory to store all counters and metadata per surface. The memory required would also make profiling some applications impossible as the driver would run out of memory to allocate. There is also an issue of performance of the tool itself. Therefore the solution that I implement is to profile on a cache line basis. More detailed explanation on how this is done is found in the next section.

4.2 DIRECTX DRIVER WORKFLOW

Before describing the design of the memory profiler it is important to understand the workflow and the workings of a DirectX driver running on windows platform. This work was done using the production driver at Nvidia Corp. A 3D application running on windows use the DirectX API. It gets compiled by what is called the D3D runtime. This runtime compiles the code into an intermediate form. The driver supplied by vendors like Nvidia and AMD is a dynamically linked library (.dll) that is placed in the system directory on windows. This .dll file is responsible for providing the implementation mapping for all DirectX API calls. The application writers code the shaders in what is called as High Level Shading Language (HLSL) [5]. The runtime compiles the shaders into an intermediate form or bytecode.

After this step the driver takes control of the compiled code. This code is however in an intermediate form which is platform agnostic. Converting to intermediate code is an essential step so that the windows operation system can support GPUs provided by different hardware vendors. The vendors providing the GPU do not expose the Instruction Set Architecture (ISA) that they support. The ISA changes from one vendor to another. It can also change from one generation of a chip to another for a given vendor.

Therefore this intermediate stage becomes a necessary evil. After this step it is the responsibility of the driver to interpret and compile the intermediate code into instructions that can be understood by their hardware. The driver has an inbuilt compiler to compile the shader into assembly language understood by the hardware. The memory profiler intercepts the driver compiler and instrument the code before it is compiled so that we can collect the information we care about.

4.3 MEMORY PROFILER DESIGN

The basic design of the memory profiler is described as follows. We need to collect reuse information for each surface or buffer. Therefore we need some kind of temporary storage where we will accumulate the access counts per buffer. The memory profiler makes a request to the driver to provide this memory. The tool instruments the shader code and inserts an additional instruction selectively before those instructions that read or write into any buffer. The result buffer is copied back from the video memory to host memory, periodically. At the end of the frame we get a cumulative count of the accesses made to each surface at a granularity we have chosen.

4.3.1 Memory Requirement and Allocation

For a screen with 1920x1200 resolution we have 2M pixels. So for a game application that wishes to render at such a resolution it will need surfaces that can represent 2M pixels uniquely. There are various surface formats that DirectX supports. We have surfaces where each pixel can be represented with varying bits of precision e.g. 32, 64 or 128 bits. For memory profiling if we were to collect information per pixel we would need 2M * 2Bytes (16bit) = 4MB of memory per surface. Now, let us consider surfaces that support any form of anti-aliasing (AA) [6]. Typically 3D games support formats like 4xAA to 16XAA. This increases the memory requirements to 16MB (4x) or

64MB (16x) per surface. However, in a 3D game not all surfaces use AA. The total available memory on a high end card is approx 1GB. Typical games would use most of the 1 GB memory for rendering. Not all surfaces that the game allocates need to be resident in Video memory. There are some surfaces that can be pushed far out to reside on the host memory (though this will result in approximately 2X decrease in performance), but some surfaces are mandated by the programmers to be resident in the video memory. The driver will try to accommodate memory allocation requests made by the profiler but it is on a best effort basis without making the game unplayable (not able to run due to lack of memory). Given these memory constraints the driver can spare around 200 MB of memory which allows us to track approximately 50 surfaces (without AA) using a total memory of 50 * 4 = 200MB. We will not be able to profile games which uses lot more than 50 surfaces or if they make heavy use of AA surfaces.

The solution therefore is not to track memory usage per pixel. We use a compression technique. The surface being tracked is logically thought of as being composed of many small tiles. Each tile represents a rectangular block of pixels. Here is how I go about choosing the tile size. Let us say that the surface is of ARGB8 (32 bit) format. For memory usage patterns it makes a lot of sense to track accesses per cache line. Therefore we choose a tile to be a cache line size. GPU architecture support cache line size of 128 bytes. Therefore we can fit in 128/4 = 32 pixels per tile. If the end goal is to calculate the amount of data fetched from memory counting on a cache line basis will still give us a correct number. This is because even if the shader accesses any byte of the cache line the memory controller will end up fetching the entire cache line from the memory. Tracking accesses using this compression scheme allows us to shrink the memory usage by 32. This is also a scalable design where we trade-off lower precision with decrease in memory pressure. If we are still memory constrained all we need to do is make the tile size bigger i.e. comprising of 64 pixels. This comes very handy when we are tracking surfaces which use 128 bit precision as there are only 8 pixels in a cache line of that format. Figure 2 explains this in more detail.

Tile 00	*Tile 10*	*Tile 20*	*Tile 30*
Tile 01	*Tile 11*	*Tile 21*	*Tile 31*
Tile 02	*Tile 12*	*Tile 22*	*Tile 32*
Tile 03	*Tile 13*	*Tile 23*	*Tile 33*

Figure 2: A 1920x1200 surface divided into tiles. A tile is 32 pixels i.e. 32*4 = 128 bytes.

4.3.2 Instrumenting shader code.

There are few pieces of software needed to be able to instrument the shader code. 1) We need a module in the driver code which can be called before the shader is compiled and attached to the draw call. 2) This module in the driver needs a parser for the shader bytecode to be able to identify instructions of interest. 3) Lastly we need a module for analyzing the output. The design of this module is pretty flexible and we can choose to simply aggregate the count or use an analytical cache model to predict cache behavior.

```
HRESULT CreatePixelShader(
[IN]   const void  *pShaderBytes,
[IN]    SIZE_T       BytecodeLength,
[OUT] ID3D10PixelShader  **ppPixelShader
}
```

Figure 3: function prototype of Direct3D API to create Shader Object.

Using DirectX API programmers make call a function which creates shader objects. Figure 3 gives a prototype of one such call. *pShaderBytes is a pointer to the bytecode of the shader. The driver needs to do some book keeping and processing before it can attach the shader to the Draw call. The most important of these tasks is compiling the shader bytecode into native GPU ISA. The profiler needs to intercept this compilation step. The instrumentation of the shader is done on the compiled code and not on the byte code representation. Instrumenting at the ISA level allows me to handle Nvidia specific instructions and also get additional information from these instructions. However, what this means is that I need to be able to parse the assembly generated from the compiler, identify instructions and then resend them for compilation. This is an additional step but is necessary. Each compile of the shader is done in a few microseconds and we do not incur a performance hit due to this extra compile. Table 2 provides a list of load/store instructions in the ISA. I cannot elaborate on the details of these instructions as this information is confidential and propriety.

There are two types of memory operations happening in the hardware. First, the ones that are explicitly coded as shader instructions, inside a Vertex Shader, Geometric Shader, Hull Shader or Pixel Shader. The second class of memory operations is implicitly encoded in the pipeline. For e.g. after the rasterization we get many more fragments then the final pixels. For each fragment there is an implicit read of the depth buffer if depth read/write is enabled and also an implicit read of the Render Target if color write is enabled. Another important example of implicit access is related to blending. This is done in the final stages of the DirectX pipeline. Typical blending operation consists of read-modify-write kind semantic. Though the buffers touched for blending are typically the same i.e. Render Target and Z-buffer. If we wish to collect information for reads and writes separately we would like to treat blending differently. When the shader code is instrumented we would like to record both the implicit and explicit memory operations.

The last piece of information needed to form the instruction which we will insert is the position of access in screen space by the original instructions. For explicit memory access this can be derived by looking at the runtime values of the operands. However, it

is very important that we correctly calculate the position for the implicit memory accesses. This information can be made available if we enable specific setting when we compile the shader. This puts the rasterized coordinates in specific registers which can be read by the code that we insert.

Predicate instructions need to be handled carefully. Let us look at the predicate instructions. In the Nvidia architecture each instruction can be predicated. From figure 4 @P2 LDLK P3, R1, [R4]. The semantic of this instruction is - if P2 equals 1, load 32 bits in R1 from address in R4, request lock and write the result in P3. This means that when we instrument instructions we need to be aware of the value of the predicate register at runtime. The module responsible for inserting instructions checks to see the presence or predicates and inserts the instruction predicated on the same register.

```
TLD.LZ   R0, R4, 0, 2D, 0xf;
TLD.MS R0, R4, R6, 0, 2D, 0xf;
@P2 LDLK   P3, R1, [R4];
```

Figure 4: Sample texture and predicated instruction.

Similarly, texture instructions (Table 2) have rules that needs to be understood and followed. Let us look at a sample Texture instruction (figure 4). It has many operands. Let us examine them one by one. The .LZ means that we are accessing the zero MipMap Level and .MS means we are in a MultiSample mode provided by the value in R6. The first constant "0" gives the number of the texture Id. Direct3D API allows 128 textures that can be bound. This is used to identify the texture that is bound. The next parameter "2D" denotes the type of the texture format. It can be 1D, 2D or 3D. I do not support profiling of 3D textures because even after compression it would need lots of

memory. The final constant "0xf" denotes a mask indicating what all channels are active. Each bit can be used to mask off Red, Green, Blue, and Alpha channels respectively. The values in the registers R0, and R4 store the u, and v coordinates. These u, v coordinates are normalized. To find out the exact coordinates we need to find the dimension in X and Y and using them de-normalize the u, v coordinates. The size information of each texture can be retrieved from the texture that is attached. Once we have the de-normalized coordinates we use the same logic as is used for any other load or store instructions.

What we are really gathering is an access count per tile per surface. This is a counter that needs to be incremented possibly, by more than one thread. There are 2 possible choices to make it thread safe. First is to avoid the conflict, which means that each warp will have its own set of buffers. This is a problem because we will end up bloating the memory footprint of the profiler. Second option is to use thread safe memory operation like locks. The memory profiler use Atomic instructions to accumulate the count.

To be accurate in our measurements we also need to pay attention to the number of active lanes. In Nvidia GPU architecture [7] the threads are organized in lanes and all the lanes execute the same piece of code. If a lane is not active, the hardware does not actually commit any instructions to the buffer. The memory scans the first active thread in a warp, counts the number of active threads and uses both the information to accumulate the count. Say if thread id 0 is active and in total only 10 threads are active, then for every memory access it will increment the counter by 10.

Figure 5 provides a sample of what the shader looks like after it has been instrumented by the memory profiler. The code in the figure omits the actual shader algorithm as it is propriety information and I provide a skeleton with enough information for us to decipher the resultant modified shader.

TEX, TEX_N	Texture Fetch
TLD	Texture Load
TLD4	Texture Load4
TMML	Texture Get MipMapLod
TXA	Texture Virtual AA Resolve
TXD	Texture Fetch With Derivatives
TXQ	Texture Query
STP, STP_N	Set Texture Phase

Table 2. List of various forms of Texture access instruction supported by ISA.

Opcode	**Definition**
LD, LD_N, LDU_N	Load
LDU	Load Uniform
LDLK	Load and Lock
ST, ST_N	Store
STUL	Store and Unlock
ATOM	Atomic operation
RED	Reduction operation
LDL, LDL_N, LDS, LDS_N	Load Local, Load Shared
LDSLK	Load Shared and Lock
STL, STL_N, STS, STS_N	Store Local, Store Shared
STSUL	Store Shared and Unlock
LD_LDU	Load || Load Uniform
LDS_LDU	Load Shared || Load Uniform
SULD	Surface Load
SULEA	Surface Load Effective Address
SUST	Surface Store
SURED	Surface Reduction
SUQ	Surface Query

Table 3: List of various forms of Load and Store instruction supported by ISA.

```
SHADER_START_LABEL

<Read Raster Pos X and Pos Y>     // Inserted top of each Shader for RT and ZB

<Find the tile#>

<increment tile# usage in the Attached Profile Buffer >

FMUL  ….

FMAD ….

DOT ……..

<Read "u" and "v" coord for TEX>
< find TEX Width and Height – Denorm U,V>
<Use the Denorm Value to get the Texel #>
< Map Texel to Tile #>
<increment tile# usage in the Attached Profile Buffer >

TEX …..
FADD ….

SHADER_END_LABEL
```

Legend :

- *Italic* (Code Inserted by Profiler for RT/ZB)
- *Italic* (Code Inserted by Profiler for Texture)
- Black (Pixel Shader original code)

Figure 5. A sample code of instrumented Pixel Shader.

4.3.3 Performance of the tool

As it turned out during the experiments I conducted, transferring the results from video memory to host memory was the most performance sensitive part of the pipeline. Initially I was expecting the increased amount of code or the atomic increments to be a problem but the real limiter turned out to be the data movement across the PCI Express lanes. After every draw call completes execution on the GPU, I transfer the results that have been collected and resident on the video memory back to the host memory. We are transferring result buffers for Render Targets (up to 8), a depth buffer and 128 possible Textures. However, not all surfaces are touched in each draw and we can come up with optimizations which copies results only for surfaces that have been touched. Currently with memory profiler enabled it take close to 3 minutes to run one frame. This is close to 6000X slower as compared to the regular mode. This is primarily due to the fact that we starve the GPU and feed it one draw call at a time so that we can transfer the results across to host memory. If we choose to run longer and transfer results later in batches we run the risk of running out of memory. Another possible approach is to insert a compute kernel after each draw call which accumulates the result into one unified buffer per surface. This will result in very small amount of data being copied at the end of the frame and we will also release the video memory which can be used for collecting results for another draw call. However, I did not feel the need to implement this approach as the scope of the tool is to profile one frame. In case a need arises to profile complete benchmarks which are say 3 minutes long (30*3 = 90 frames) or more, this tool will take a couple of hours to profile a frame. Then we surely need to think about speeding up the tool.

CHAPTER 5

RESULTS

5.1 VERIFICATION

The memory profiler was validated using some simple tests. The sample test is the program "BasicHLSL" from the DirectX SDK. This is a DirectX10 program. To be able to verify the correct functionality we would like to see the correct access count being reported for a given draw call. Also, we can verify if the tiles are correctly placed. This is easy if we can verify using one draw call. The draw call should be simple enough with basic loads and stores instructions and also some texture access. BasicHLSL program actually consists of 10 different draw calls which are used to render basic control buttons. I removed all the unnecessary draw calls and kept only the single draw call which renders the body. This single draw renders the entire body. Another piece of the memory profiler code that needs to be validated is related to copying and merging the results. To be able to test that I added 25 instances of the same draw call. What we should see is that the surface is touched 25 times. I do not report the number of fragments per tile but it was verified to match the tile size (in pixels) for fully covered tiles. For semi-covered tiles we did see fewer fragments being counted. In Figure 6 we see the image that was extracted using Nsight [8]. Figure 6 shows the image that is finally rendered and figure 7 shows the texture that is associated with this frame. We need to verify the access patterns for Render Target (RT), Z Buffer (depth buffer) and the textures. Visual verification of the Render Target and Z buffer is easy, the tiles carves out the same shape. To verify the count reported by the profiler I plot the aggregate access per tile. The graph reports the use count per draw call. It does not report the count of the number of threads that write to each tile. With 25 draws each in the application each cell or tile has a count of 25 denoting the over draw or access on that surface.

For verification of the texture we need to observe the image more carefully. Firstly, there is only 1 texture representing the clothes, face, shoes etc of the character. The application carefully chooses the u,v texture coordinates. When we look at the surface which represents the texture access we will see that back portion of the characters pants are not accessed. Here the count is zero. Similarly the count is zero for the texture regions that represent the back portion of the characters face. The results for render target and z-buffer is shown in figure 8 and the texture access in figure 9.

5.2 RESULTS

Once we have verified the results of the memory profiler using simple DirectX examples we can move onto larger frames. With larger frames it quickly gets very tricky to be able to inspect visually. Figure 11 presents the results for a frame in figure 10 from the Unigine Benchmark version 2.1 [Ref] running in DirectX mode. I make a best guess to select the surface which is the final render target but it is very difficult to be accurate. This might as well be the surface representing some intermediate stage. We can do some manual inspection using commercial profiling tools like Nsight but they will work for certain surfaces like the Render Target and we cannot visualize the texture access using this tool.

Figure 6: Render Target for DirectX SDK sample app “BasicHLSL”

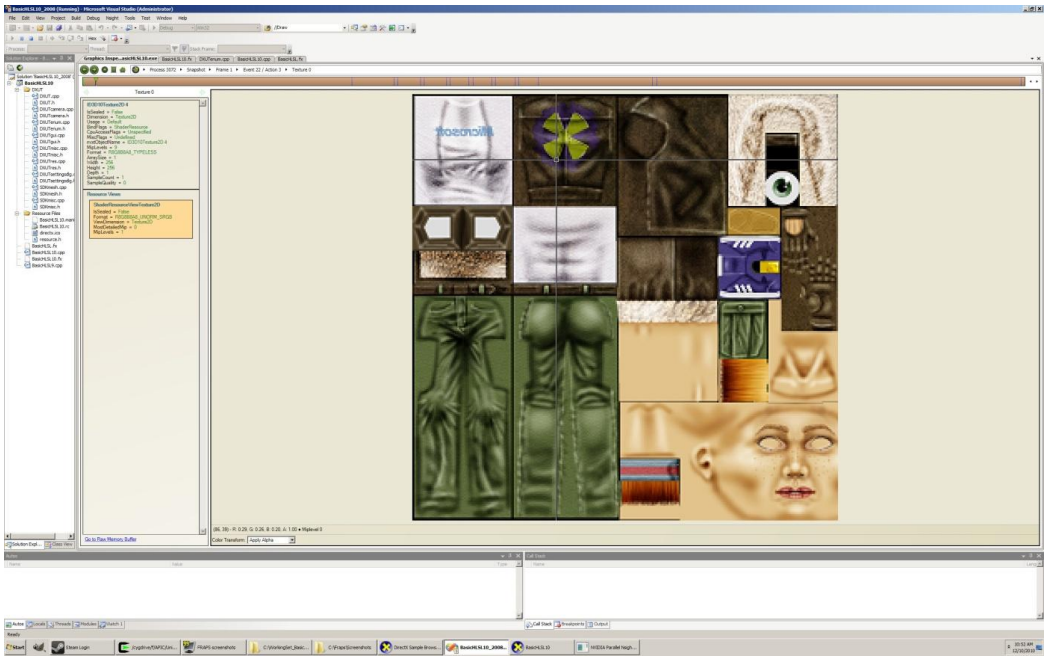

Figure 7: Texture used in BasicHLSL. Note the coordinates of rear portion of the trousers.

Figure 8: Surface representing the access for RT and ZB.

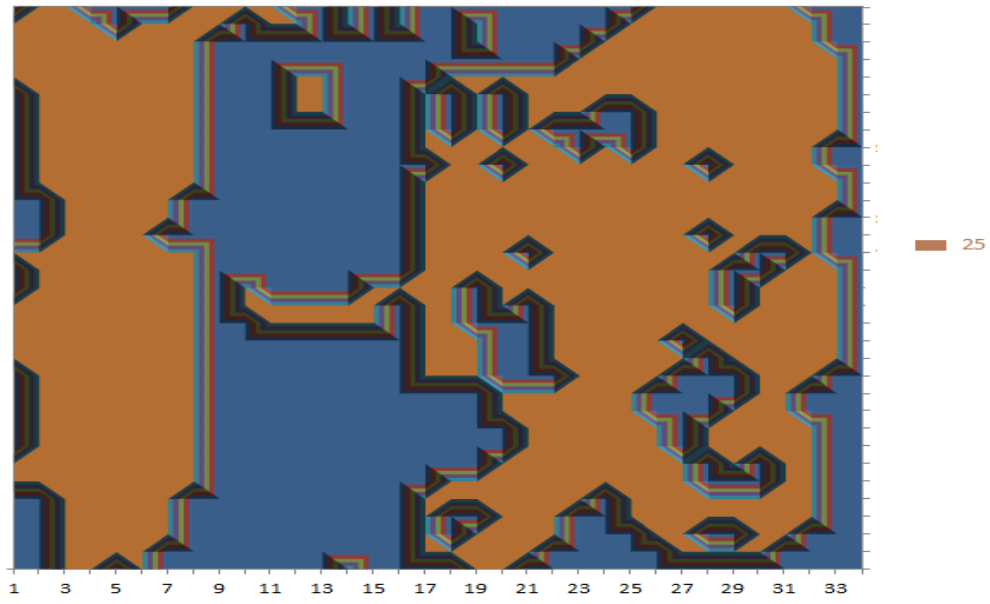

Figure 9: Surface representing the access for Texture

Figure 10. Sample Frame from Unigine 2.1

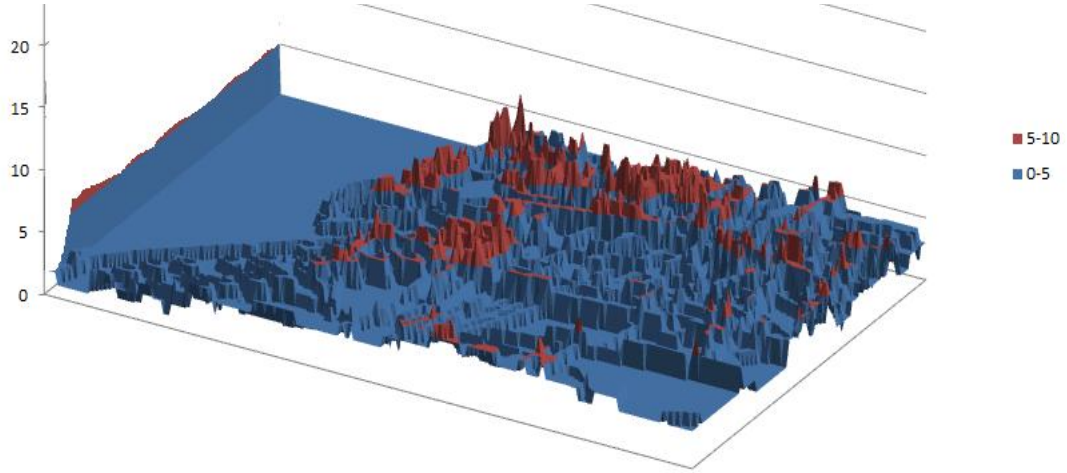

Figure 11. Memory access per tile for one of the surface used in Unigine

5.3 ANALYTICAL CACHE MODEL

Once we have profiling data collected per surface, we can make further use of the data and do some offline analysis. I experimented with a simple analytical cache model to provide some insight into reuse of data across Draw calls within a frame. We could also look at caching behavior within a Draw call or across frames. We possibly cannot make the cache large enough to exploit any reuse of data across frames. Each frame working set is very large to the tune of 100s of MB. Also, since the memory profiler does not have a time stamp recorded for each access made it is very difficult to predict the cache behavior within a draw call (assuming a small 1 to 2MB cache size). For these small sizes the temporal access pattern from different cores of the GPU will have an impact on the cache hit rate for a single Draw. Therefore the scope of this work is restricted to analyzing cache performance across Draw calls.

For this analytical cache model the assumption is that the cache is small enough that we cannot protect data across draw calls. For now, I assume a 100% eviction probability of a data at the end of a Draw. This can be configured by taking into account the working set of the draw call and the cache size. We would like to assess the benefit of increasing cache size by looking at data reuse patterns across draw calls. Figure 12 shows the design of the analytical cache model. The 'X' in black denotes access to a tile and 'X' in red ink denotes access to the same tile in subsequent draw. The memory profile allows us to calculate the number of tiles with 'X' in red ink i.e. reused across draw calls. This is a good approximation for calculating the reuse. When we do this analysis for each surface in isolation and what we are looking for is surfaces with good amount of reuse across draw calls. Once we find such a surface with good reuse across draw calls, we also need to look at the absolute memory use count. If both these counts are high then we have found a good candidate for caching. Figure 13 shows results across one frame.

The final part of the puzzle is after we identify a surface which can benefit from caching, we need to find a way to protect or pin this surface in the cache. If we don't do

this than any other surface can evict this data. This can either be done transparently by hardware with the help of page level caching attribute. The application writer can request memory accesses for such surfaces to be made from such pages which has high priority in the caches. We can simulate such behavior by biasing the LRU bits of when data is pulled from such pages. Another approach could be via some DirectX language extension which allows the application writer to provide hints when it allocates the surface and deem it amenable to caching.

Analysis Tools

Basic Analytical Cache Model

– Draw 'n' ➔ the profile buffer for Surface S looks like this

X			
X	X		
X	X	X	
X	X	X	

"X" against a tile means access

– Draw 'n+1' ➔ the profile buffer for Surface S looks like this

X	X	X	X
X	X	X	
X	X		
X			

"X" or "X" against a tile means access
"X" means that this part of buffer was reused across Draw Calls

Total memory traffic = ∑ ("X" + "X") x Size of each Tile.

*Based on worst case with no caches or that it gets evicted across draw calls. Open to debate ☺

* Within a Draw Call we do not count take into account multiple access for BW computation.

Total memory traffic Saving Opportunity = ∑ "X" x Size of each Tile.

Figure 12: Design of analytical cache model.

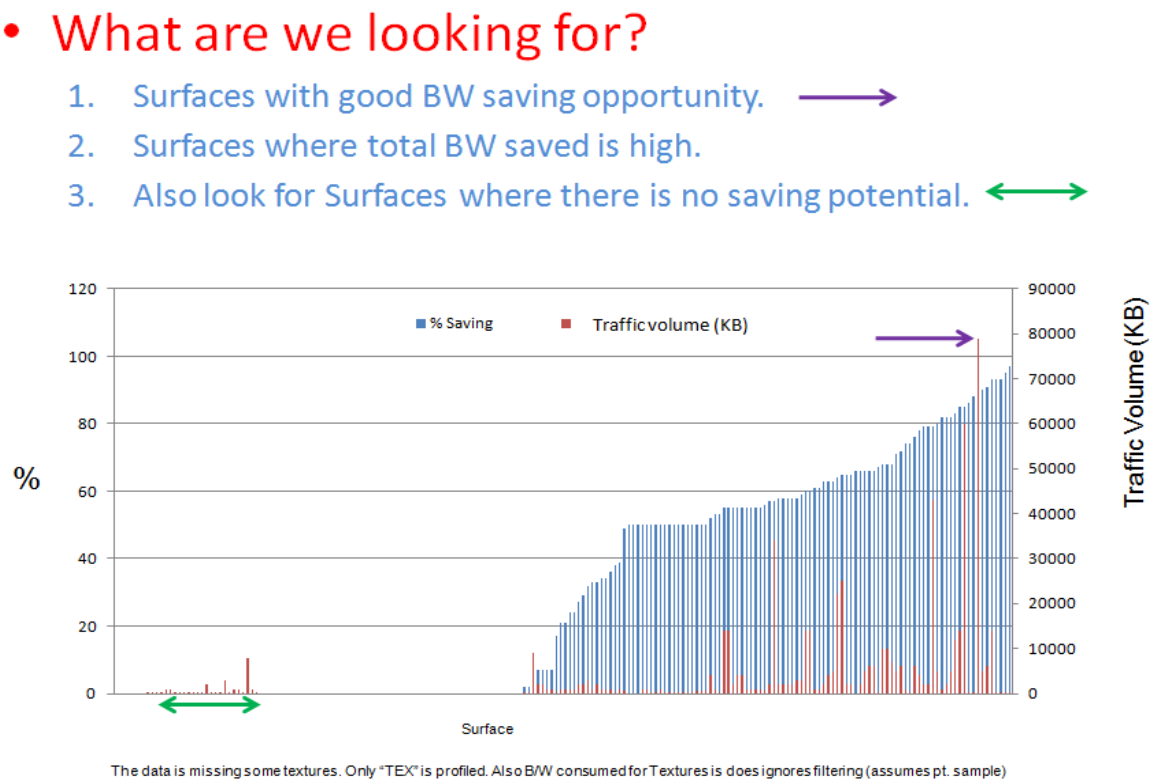

Figure 13: Results using our analytical cache model for Unigine Heaven.

CHAPTER 6

CONCLUSION

6.1 SUMMARY & FUTURE WORK

Heterogeneous platforms as we know today comprising of CPU and GPU brings together two inherently different architectures onto a single die. CPUs have targeted higher single thread performance with each new generation whereas the GPUs have provided better throughput with each generation. This approach is primarily driven by the workloads that are targeted for them. Now, with heterogeneous platforms they share the same memory subsystem. The architects and designers of the heterogeneous platforms have a difficult task at hand designing a common system to be shared by these architectures. Typically their expertise lies in either one of the domain. Therefore it is important that we provide tools to investigate the memory behavior of different workloads and try to find the commonality between them. 3D games are a unique workload and it is not very well understood by people outside the graphics community. Moreover they evolve at a very rapid pace. The tools should help break down the graphics workload into its basic components. These kinds of tools will help educate the architecture community at large about the behavior of graphics workloads.

In this report I demonstrate a working prototype of one such tool targeted specifically to answer architecture tradeoffs related to memory subsystem of heterogeneous architectures. My work is not complete and can be improved further. For instance we can add support for profiling the Hull Shader. There can also be better analysis tools that can take as input the data produced by this tool. These analysis tools can do the processing either online or offline. I believe the real value of this tool lies in the fact that the results can be analyzed offline in varied ways. We can also extend this tool to handle other API like OpenGL. The speed of this profiling tool is acceptable for a

couple of frames which is the typical use case. The tool can however be extended easily to handle long running benchmarks.

BIBLIOGRAPHY

[1] http://www.gpgpu.org/

[2] http://msdn.microsoft.com/en-us/directx/aa937781

[3] Crow, F.C.: Shadow algorithms for computer graphics. In Computer Graphics (SIGGRAPH '77 Proceedings), (1977) 242-248

[4] http://www.pintool.org/

[5] http://www.paradoxalpress.com/LayoutSamples/HLSLReference.pdf

[6] http://en.wikipedia.org/wiki/Antialiasing

[7] Erik Lindholm, John Nickolls, Stuart F. Oberman, John Montrym: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28(2): 39-55 (2008)

[8] http://developer.nvidia.com/nvidia-parallel-nsight

[9] http://unigine.com/