

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Exploring security implications of the AMD STREAM PROCESSOR

Permalink

<https://escholarship.org/uc/item/0r38818x>

Author

Lee, Seung-won

Publication Date

2008

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Exploring security implications of the AMD STREAM PROCESSOR

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in
Computer Science

by

Seung-won Lee

Committee in charge:

Professor Hovav Shacham, Chair
Professor Stefan Savage
Professor Geoffrey M. Voelker

2008

The thesis of Seung-won Lee is approved and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2008

TABLE OF CONTENTS

Signature Page	iii
Table of Contents	iv
List of Figures	vi
List of Tables	vii
Abstract	viii
I Introduction	1
II CAL PLATFORM	5
A. Introduction	5
B. Programming Model	5
C. CAL Runtime	7
1. System	8
2. Device Management	8
3. Context Management	8
4. Memory Management	10
5. Program Loader	12
6. Computation	12
D. Sample application	14
III AMD Memory Controller	20
A. Memory Controller	20
B. PCIeExpress	22
C. Hypermemory	25
IV Windows Memory Forensic	28
A. Cycle counts of read and write	28
B. Acquisition of memory	29
1. Hardware based acquisition tools	29
2. Software based acquisition tools	31
C. Memory Analysis	32
V Windows Kernel Debugger	36
A. Converting virtual address to physical address	36
B. Windows Kernel Debugger	40
1. Commands and Results	40

VI	Conclusion	47
	A. Future Work	47
	B. Summary	49
	Bibliography	51

LIST OF FIGURES

Figure I.1:	Hardware Abstraction (CTM)	2
Figure I.2:	AMD Stream Computing Software Stack	3
Figure II.1:	CAL System	6
Figure II.2:	CAL Platform	7
Figure II.3:	CALdeviceattrs	9
Figure II.4:	CALdevicestatus	9
Figure II.5:	Local and Remote Memory	10
Figure II.6:	CAL code generation	12
Figure III.1:	Ring-bus diagram	21
Figure III.2:	PCI system layout	22
Figure III.3:	The shared bus	23
Figure III.4:	The shared switch	24
Figure III.5:	Links and lanes	24
Figure III.6:	Memory allocation and use with HyperMemory	26
Figure III.7:	PCI Express memory auxiliary memory channel	26
Figure IV.1:	VAD tree for notepad.exe	34
Figure V.1:	Virtual to Physical (x86)	37
Figure V.2:	Virtual to Physical in PAE enabled x86 system)	38
Figure V.3:	!process 0 0 command	41
Figure V.4:	“!vtop” command	43
Figure V.5:	Physical address range associated with PCI bus	44
Figure V.6:	!pfm output	45
Figure V.7:	GPU writing to memory region written by CPU	46

LIST OF TABLES

Table IV.1:	Cycle counts of read and write	30
Table V.1:	!process 0 0 output	42

ABSTRACT OF THE THESIS

Exploring security implications of the AMD STREAM PROCESSOR

by

Seung-won Lee

Master of Science in Computer Science

University of California, San Diego, 2008

Professor Hovav Shacham, Chair

Since its invention, the graphics card has become one of the most important components of a computer system. In early days of computer architecture, the CPU was responsible for performing most of the computations needed for the transfer of the image data from the software state onto the screen, while the graphics card's role was limited to determining where the resulting image was to be placed on the screen. However, recently developed graphics cards not only can handle manipulating and displaying images, but are also making computational inroads into some of the tasks formerly performed by the CPU due to their fast, highly-parallel embedded graphics processing unit (GPU). Modern, high-end GPUs have faster computation rates compared to CPUs when dealing with heavy floating-point computations, and their usage has spread into a variety of fields, such as computing complex algorithms in Artificial Intelligence, Bioinformatics, etc.

ATI Technologies has recently developed the AMD Stream Processor (also known as the AMD FireStream) that uses a modified stream processor to allow a General Purpose Computing on Graphics Processing Units (GPGPU). The stream processing hardware comes with Close-to-Metal (CTM), a hardware interface that exposes the GPU architecture to provide direct communication to the device. The CTM allows the developer to directly access the instruction set, along with the memory space used by the AMD graphics card. The developer not only can allocate and access resources in

the GPU local memory, but also has the ability to control resources in the CPU system memory.

In this thesis, we particularly focus on the CTM's ability to allow developers to directly allocate and access memory space in the CPU. The core of our research aims to understand how the AMD Stream Processor writes to the system memory, including whether it writes to the system memory independent of the CPU, and whether a malicious user can inject malicious code through the GPU without being noticed by any anti-virus program.

I

Introduction

The General Purpose Computing on Graphics Processing Units (GPGPU) is a technique of using a high-performance, parallel GPUs to perform computation in applications traditionally handled by the CPU. Modern graphics cards have additional programmable states in the rendering pipelines to allow software developers to use GPU on non-graphics data [2].

Unfortunately, developers have had a hard time utilizing the powerful GPU due to the lack of an Application Programming Interface (API) that can access the graphics hardware. The OpenGL and the Direct3D are, at present, the two most widely used APIs in the graphics field; however, these APIs are not designed for non-graphics applications, and hide most of the architectural details of the GPU that may be useful or necessary to application programmers.

The AMD Stream Processor is a stream processor that utilizes the GPGPU concept for heavy floating-point computations. The Stream Processor comes with a new hardware abstraction called Close-to-metal (CTM). The CTM exposes the GPU architecture as a data-parallel processor array and a memory controller, fed by a command processor [5]. Figure I.1 illustrates how the CTM exposes GPU. The applications can directly communicate with the GPU by calling the CTM APIs that control the command processor. The Data Parallel Processor Array (DPP) executes the CTM

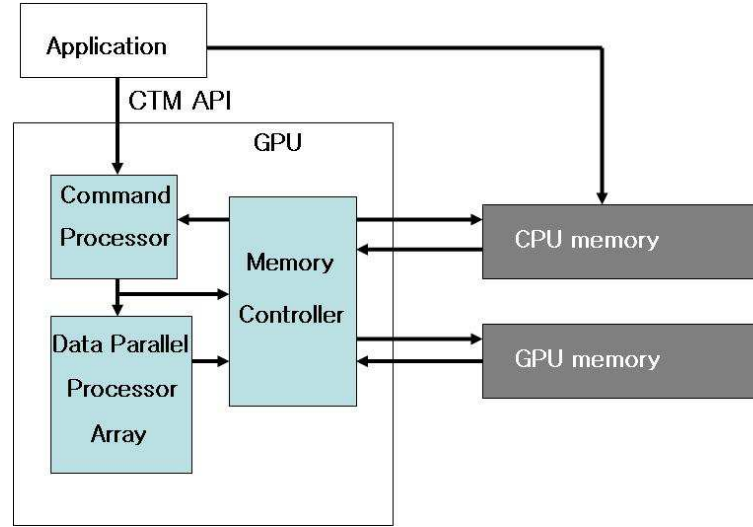


Figure I.1: Hardware Abstraction (CTM)

program, and interacts with the Memory Controller (MC) to read from and write to memory space in both CPU (System) and GPU (Local). The CTM hides all other graphics-specific features and eliminates driver-implemented procedural APIs to push policy decisions to application and remove constraints imposed by graphics APIs [41].

Our intuition of the CTM was that it gives too much leverage to the developers by allowing them to read/write directly to/from CPU (system) memory. If the *write* function occurs independent of the CPU, a malicious user not only can inject malicious code, but can also overwrite the memory space used by the kernel. In addition, virus scanning programs may have a hard time detecting malicious behavior if the injected code resides on the GPU side.

Unfortunately, AMD has revamped the CTM and has released a new hardware abstraction called the Compute Abstraction Layer (CAL). The CTM Software Development Kit (SDK) is no longer available, and the developers now must use the Stream Computing SDK available at [9]. We actually had use of both the CTM and the CAL SDK, but decided to use the CAL, since the CTM SDK does not support high-end graphics cards, and the CAL supports most of the features present in the CTM,

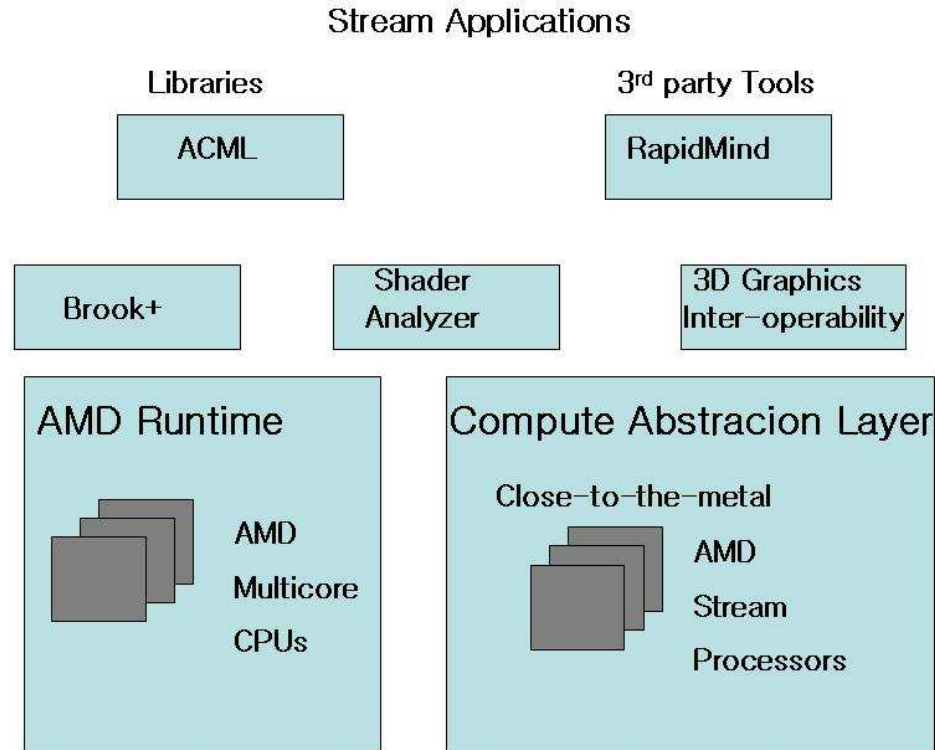


Figure I.2: AMD Stream Computing Software Stack

including the ability to allow developers to directly access the CPU system memory.

The Stream Computing SDK is composed of Brook+, a modified Brook open source compiler and CAL, a cross-platform interface to the GPU [9]. Figure I.2 shows the software stack of stream applications.

The Brook+ is an open source Brook C/C++ compiler for developers creating non-graphics applications able to run in the GPU environment. Ordinary developers should first try using Brook+ to write applications, since Brook+ is more straightforward, in a sense that it allows users to program a C-level code for the kernel that runs on the GPU. In addition, Brook+ sits on top of CAL and provides more user-friendly APIs. However, we used CAL to write our application, since CAL allows users to allocate and access resources in CPU system memory. The details of the rest of the components in the software stack of the AMD stream applications can be found

at [8].

The remainder of this thesis is about our understanding of how the AMD Stream Processors accesses the CPU system memory under Windows XP Service Pack 2.

In Chapter II, we first describe CAL platform along with the application we have used to invoke the GPU to write to the system memory. In Chapter III, we describe the memory controller used in the AMD Stream Processor, including how it accesses system memory through the PCI Express interconnect. In Chapter IV and V, we describe various tools and experiments that we have used to verify whether the GPU indeed writes to the system memory, independent of the CPU. Finally, in Chapter VI we conclude our discussion, along with an overview of future work.

II

CAL PLATFORM

In this chapter we describe the structure of the Compute Abstraction Layer (CAL) and the various CAL runtime components, along with their relevant APIs. For further understanding we present, at the end of the chapter, a sample application that causes the GPU to write to the system memory. Most of the contents in this chapter can be found in the “AMD Compute Abstraction Layer Programming Guide” [6] and “CAL Platform” [12].

II.A Introduction

“The AMD CAL is designed to provide an easy-to-use, forward compatible interface to the high-performance, floating-point, parallel processor arrays found in the AMD Stream Processors” [6]. The computational model of CAL is processor-independent, and it allows users to switch from directing a computation from the GPU to the Central Processor Unit (CPU), or vice versa.

II.B Programming Model

Figure II.1 illustrates a CAL system. It has one master process driving one or more GPU devices. The CPU is responsible for running the master process and

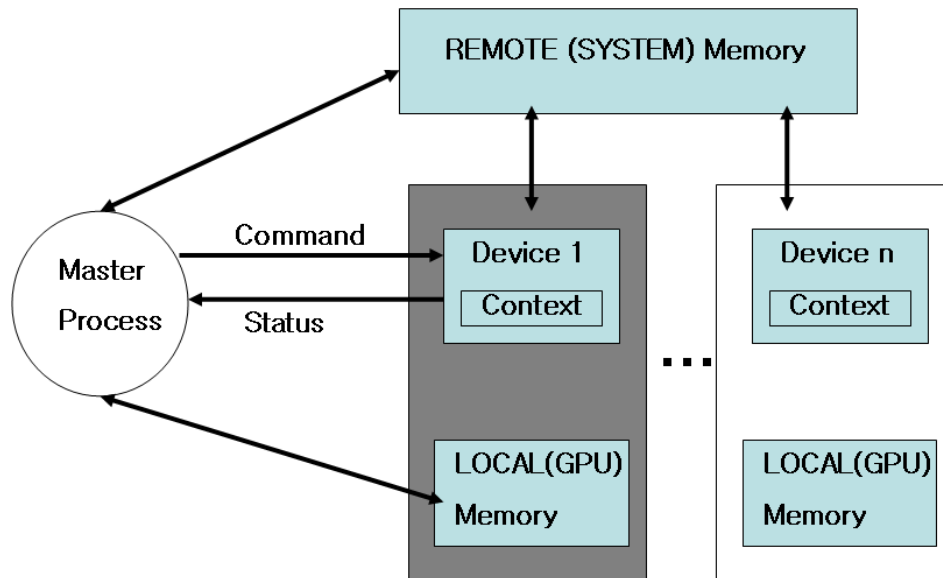


Figure II.1: CAL System

for sending commands to the GPU through the CAL APIs. Each GPU is a physical device that runs CAL programs. The GPU receives commands from the CPU, and runs the computational function specified by the application running on the CPU side. Both the GPU device driver program and the CAL run on the CPU side. A device is connected to two memory subsystems: the local (GPU) and the remote (system) memory. Context on a device can read from and write to its GPU local memory through fast memory interconnects, and it can also access remote (system) memory through the PCI Express interconnect. The details of PCI Express are presented in chapter III. The master process executing on the CPU side also has access to both of the memory pools. The CAL abstraction has two key types of commands: device commands and context commands. The device commands involve resource allocation to both local and remote memory. A context is composed of a queue of commands that are sent to a device. All resources are created on devices, and must be mapped into a context to provide scoping and access control. This feature allows multiple contexts to access the same memory resource; synchronization of these contexts should be handled by the programmer.

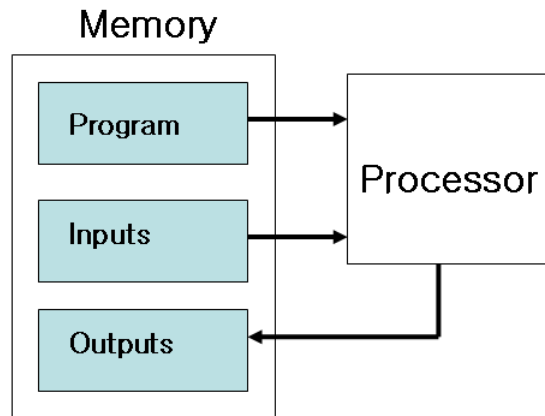


Figure II.2: CAL Platform

A device has one or more computational stream processors. The stream processor receives input, executes computational functions (also referred to as kernel), and subsequently writes the results to an output resource. Both inputs and outputs may reside in either the local or the remote memory. Figure II.2 shows the computational model of CAL.

II.C CAL Runtime

The CAL runtime is composed of multiple parts: the system, the device management, the context management, the memory management, the program loader, and the program execution. In this section, we describe what these components are responsible for, and how some of the key APIs are related to each of these components.

II.C.1 System

The system component is responsible for both initializing and shutting down of the CAL system. The *calInit* routine initializes the CAL system and recognizes all the running CAL devices on the system. This routine should be invoked at the beginning of an application, before calling any other operations. Similarly, *calShutdown* should be called before closing the application, in order to cleanly shut down every component. The application can query the CAL version running on the system through the *calGetVersion* routine.

II.C.2 Device Management

The device management component manages both opening and closing devices. Multiple devices can run in the CAL system, and each device is identified by a unique number in the range $[0..N-1]$, where N is the number of available CAL devices on the system. The *calDeviceGetCount* routine is used to query the number of devices, and GPUs running on the system. The general device information, including the device type and the maximum dimensions of buffer resources, can be retrieved by the *calDeviceGetInfo* routine. Before running any device operations, a dedicated connection between the device and application should be created using the *calDeviceOpen* routine. Similarly, an application needs to shut down the device before it exits through the *calDeviceClose* routine. In addition to retrieving the basic information of a device returned by the *calDeviceGetInfo* routine, the application can extract more detailed information using the *calDeviceGetAttribs* and the *calDeviceGetStatus* routines. Figure II.3 and II.4 are the structures returned by the routines.

II.C.3 Context Management

The context management component is responsible for creating and destroying contexts on a particular device. A context can be thought of as an abstraction representing all the device states that affect the execution of a CAL kernel. The ap-

```

Struct CALdeviceattribs
{
    CALuint    struct_size:      /*size of CALdeviceattribs struct
    CALtarget  target:           /*asic identifier
    CALuint    physical RAM:      /*amount of local GPU RAM
    CALuint    uncachedRemoteRAM: /*amount of uncached remote RAM
    CALuint    cachedRemoteRAM:  /*amount of cached remote RAM
    CALuint    engineClock:       /*GPU device clock rate
    CALuint    memoryClock:       /*GPU memory clock rate
    CALuint    wavefrontSize:     /*Wavefront size
    CALuint    numberOfSIMD:      /*number of SIMDs
    CALboolean doublePrecision:   /*double precision supported
    CALtarget  reserved1-4:       /*reserved
    CALboolean memExport:         /*memory export supported
}

```

Figure II.3: CALdeviceattribs

```

Struct CALdevicestatus
{
    CALuint struct_size:      /* size of struct
    CALuint availLocalRAM:     /*available local RAM
    CALuint availUncachedRemoteRAM: /* available uncached remote memory
    CALuint availCachedRemoteRAM: /*available cached remote memory
}

```

Figure II.4: CALdevicestatus

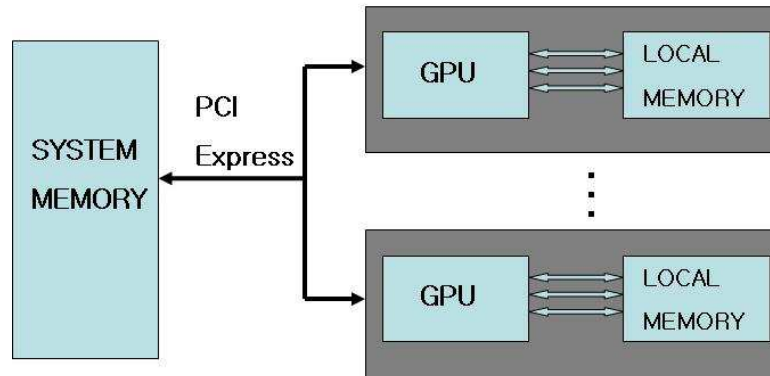


Figure II.5: Local and Remote Memory

plication needs to have a valid CAL context before executing any kernel on a CAL device. The application can create or destroy a CAL context by calling the *calCtxCreate* and the *calCtxDestroy* routines, respectively. Destroying a context involves releasing all modules and memory related to the specified context. As we mentioned earlier, a resource needs to be mapped onto a context for scoping and access control. The *calCtxGetMem* routine is used to map a resource onto the specified context, and it generates a memory handle of the resource. Before closing the application, a generated memory handle needs to be released through the *calCtxReleaseMem* routine.

II.C.4 Memory Management

The memory management component manages allocating and freeing memory resources in both local and remote memory. All CAL devices have access to local and remote memory. Remote memory corresponds to memory that is not local to the given device (GPU). In this document, remote memory, however, refers to system memory only (Figure II.5).

The following steps are required before using the memory buffers in the CAL kernel:

- Allocating memory resources (local or remote).
- Initializing the input and constant contents by mapping the resource onto application address space.
- Creating memory handles for each resource.
- Binding memory handles to corresponding parameters in CAL kernel

‘Resource’ refers to all physical memory blocks allocated in both local and remote memory. Local resources are allocated using the *calResAllocLocal* routine. Similarly, remote resources can be allocated through the *calResAllocRemote* routine. At the time of allocation, both the data type and the format of the element in resources need to be specified. The supported formats are 8, 16, 32-bit, signed and unsigned integer types, as well as 32 and 64-bit floating-point types. An allocated remote resource uses uncached system memory by default, and may slow down the system when accessed by the CPU. However, uncached memory does give better performance for non-CPU involving operations (e.g. Direct Memory Access (DMA)). CAL resources are used as inputs, outputs and constants to CAL kernels. Inputs and constants need to be initialized by the host application through the *calResMap* routine. The routine maps the resource to the applications address space and returns a host-side memory pointer. The routine is synchronous and blocks every other operation until the CPU pointer is valid. For a local surface, this may require a copy of a resource; however, for remote surfaces a pointer is always returned without copying. This function is one of the key APIs to keep in mind, since the returned pointer is equivalent to the virtual address of the resource which is required to perform various Memory Forensic techniques, which we describe in chapter IV and in chapter V. A mapped resource cannot be accessed by the CAL kernel; therefore, the resource needs to be unmapped through the *calResUnmap* routine before being used by the kernel.

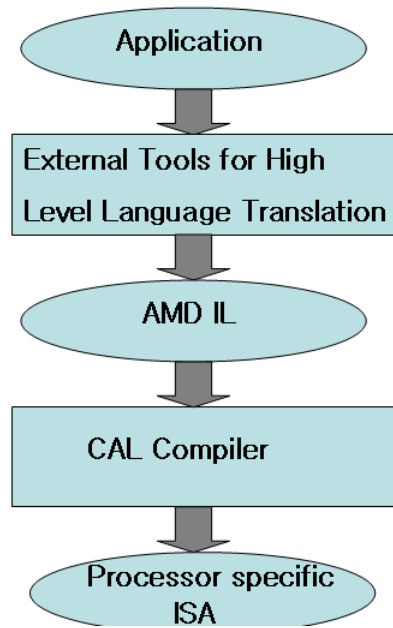


Figure II.6: CAL code generation

II.C.5 Program Loader

This component provides a mechanism to load an image (or program) onto a context. The CALimage executable file format must be the Executable and Linking Format (ELF) [11]. Creating an image involves compiling and linking a source written in one of many supported languages (e.g. Microsofts DirectX High Level Shading Language (HLSL); Brook+ [10]; or AMD Intermediate Language (IL) [7]).

The *calclCompile* routine invokes the CAL compiler to compile device kernels written in one of multiple supported interfaces, and generates a binary object specific to a given device (Figure II.6). After compilation, the object needs be linked into an image through the *calcllink* routine.

II.C.6 Computation

The Computation component handles setting up the inputs, outputs, and the entry point in the loaded program, along with executing a computation on a given

context.

Computation (Kernel execution) on a CAL device involves three high level steps: module loading, parameter binding, and kernel invocation.

- **Module Loading** A linked CAL image needs to be loaded as an executable module through the *calModuleLoad* routine. For execution, the entry point within the module needs to get specified before running the kernel. This entry point can be queried through *calModuleGetEntry* routine.
- **Parameter Binding** The CAL runtime also provides interfaces to setup various parameters, such as inputs, outputs, and constants, which are required by the CAL module. Each parameter is identified by IL-style variable name. For instance, *i#* represents inputs, *o#* represents outputs, and *cb#* is used for constant buffers. The *calModuleGetName* routine returns a variable handle within the module. These variable handles can be bound to any of the memory handles created by the *calCtxGetMem* routine in context component, and can later be called by the kernel at runtime.
- **Kernel Invocation** The *calCtxRunProgram* routine launches a kernel by specifying context, entry point and the domain of execution. The *calCtxRunProgram* is an asynchronous routine and the application is free to call any other routines while the computation is being done. The programmer can use *calCtxIsEventDone* routine to check for the completion of the event. *calMemCopy* routine also invokes CAL kernel, and is used to copy data buffers between remote and local devices. The routine uses a dedicated DMA engine in memory controller and is also asynchronous. The use of DMA can improve data transfer rates by accessing system memory directly independent of the CPU.

II.D Sample application

In this section, we provide an application we have used to invoke GPU to write to system memory using CAL APIs.

```
/////Header files /////
#include "cal.h"
#include "calcl.h"
#include <string>
```

The following code snippet is the kernel code written in AMD Intermediate Language. It basically loads the input buffer to the register *r0* using “sample_resource(0)_sampler(0)” and writes the value to the output buffer *o0*. The details of each command can be found at AMD Intermediate Language (IL)[7]).

```
std::string programIL =
"il_ps_2_0\n"
"dcl_input_interp(linear) vObjIndex0.xy\n"
"dcl_output_generic o0\n"
"dcl_resource_id(0)_type(2d)_fmtx(SINT)_
  fmty(SINT)_fmtz(SINT)_fmtw(SINT)\n"
"sample_resource(0)_sampler(0) r0, v0.xyxx\n"
"mov o0, r0\n"
"ret_dyn\n"
"end\n";
```

The following code snippet initializes CAL, retrieves the number of available devices in the system and opens the first (0th) CAL device. The application then retrieves the device attributes and creates a context on the opened device.

```
int main(int argc, char** argv)
{
```



```

//Initializing CAL
calInit();
CALuint numDevices = 0;
calDeviceGetCount(&numDevices);

//Opening device
CALdevice device = 0;
calDeviceOpen(&device, 0);

//Querying device attribs
CALdeviceattribs attribs;
attribs.struct_size = sizeof(CALdeviceattribs);
calDeviceGetAttribs(&attribs, 0);

//Creating context to opened device
CALcontext ctx = 0;
calCtxCreate(&ctx, device);

```

The following code compiles the device kernel via *calclCompile* and links the generated object file into an image via *calclLink*.

```

CALobject obj = NULL;
CALimage image = NULL;
CALlanguage lang = CAL_LANGUAGE_IL;
std::string program = programIL;
std::string kernelType = "IL";

calclCompile(&obj, lang, program.c_str(), attribs.target);
calclLink(&image, &obj, 1);

```

The following code allocates memory for input and output buffers used by the device kernel. Input buffer gets allocated in GPU local memory via *calResAllocLocal*, while output buffer gets allocated in system memory through *calResAllocRemote*. By allocating the output buffer in system memory, GPU is forced to write the resulting value to CPU remote memory. The application then initializes the input buffer by filling in a constant value 588 and maps both input and output buffer into a context.

```
// Input and output resources
CALresource inputRes = 0;
CALresource outputRes = 0;

//allocating resource in GPU memory
calResAllocLocal2D(&inputRes, device, 256, 256,
                  CAL_FORMAT_INT_1, 0);

//allocating resource in CPU system memory.
calResAllocRemote2D(&outputRes, &device, 1, 256, 256,
                  CAL_FORMAT_INT_1, 0);

//Initializing input buffer by filling in value(588)
int* fdata = NULL;
CALuint pitch = 0;
calResMap((CALvoid**)&fdata, &pitch, inputRes, 0);
for (int i = 0; i < 256; ++i)
{
    int* tmp = &fdata[i * pitch];
    for (int j = 0; j < 256; ++j)
    {tmp[j] = 588;}
}
```

```
//Mapping buffers into context
calCtxGetMem(&inputMem, ctx, inputRes);
calCtxGetMem(&outputMem, ctx, outputRes);
```

As we mentioned in previous section, computation (kernel execution) is done by executing three high level steps: *Module Loading*, *Parameter Binding*, and *Kernel Invocation*. The following code demonstrates how each step can be done in application.

```
// Creating module using compiled image
CALmodule module = 0;
calModuleLoad(&module, ctx, image);

// Defining symbols in module
CALfunc func = 0;
CALname inName = 0, outName = 0;

// Defining entry point for the module(parameter binding)
calModuleGetEntry(&func, ctx, module, "main");
calModuleGetName(&inName, ctx, module, "i0");
calModuleGetName(&outName, ctx, module, "o0");

// Setting input and output buffers used in the kernel
calCtxSetMem(ctx, inName, inputMem);
calCtxSetMem(ctx, outName, outputMem);

// Setting domain
CALdomain domain = {0, 0, 256, 256};
```

```
// Event to check completion of the program
CALevent e = 0;
```

```
//kernel invocation
calCtxRunProgram(&e, ctx, func, &domain);
```

```
//Waits till the completion of an event
while (calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);
```

The final code snippet shows the process to close the application cleanly. All module, image, object needs to be freed and every allocated resource should be de-allocated before killing the context.

```
// Unloading the module
calModuleUnload(ctx, module);

// Freeing compiled program binary
calclFreeImage(image);
calclFreeObject(obj);
```

```
// Releasing resource from context
calCtxReleaseMem(ctx, inputMem);
calCtxReleaseMem(ctx, outputMem);
```

```
// Deallocating resources
calResFree(outputRes);
calResFree(inputRes);
```

```
// Destroying context
calCtxDestroy(ctx);
```

```
// Closing device
calDeviceClose(device);

// Shutting down CAL
calShutdown();

return 0;
}
```

III

AMD Memory Controller

In Chapter II, we described the structure of Compute Abstraction Layer (CAL) along with various CAL runtime components, including memory management. In this Chapter, we focus on AMD GPU's memory controller (MC), including its structure and the way it accesses system memory through the PCI Express.

III.A Memory Controller

Every GPU includes a memory controller, which is responsible for managing complex task of multiple read and write requests issued by the processors [14]. This can be thought of as a client/server arrangement, where clients such as shader processors units, texture units, and the PCI express interface, sending memory requests to memory controller which acts as a server. As graphics performance scales upward, the need for the amount of data transfer over a link or interface has scaled with it. The faster the GPU processes data, the more memory bandwidth must be provided to run the system at peak efficiency. There are multiple ways to increase memory bandwidth. The most common ways are to increase the width, clock speed or data rates of the interface. However, these methods force not only the GPU, but also the attached memory devices to implement the supporting technologies. On the other hand, methods like latency hiding, caching, and compression techniques, can be implemented only in

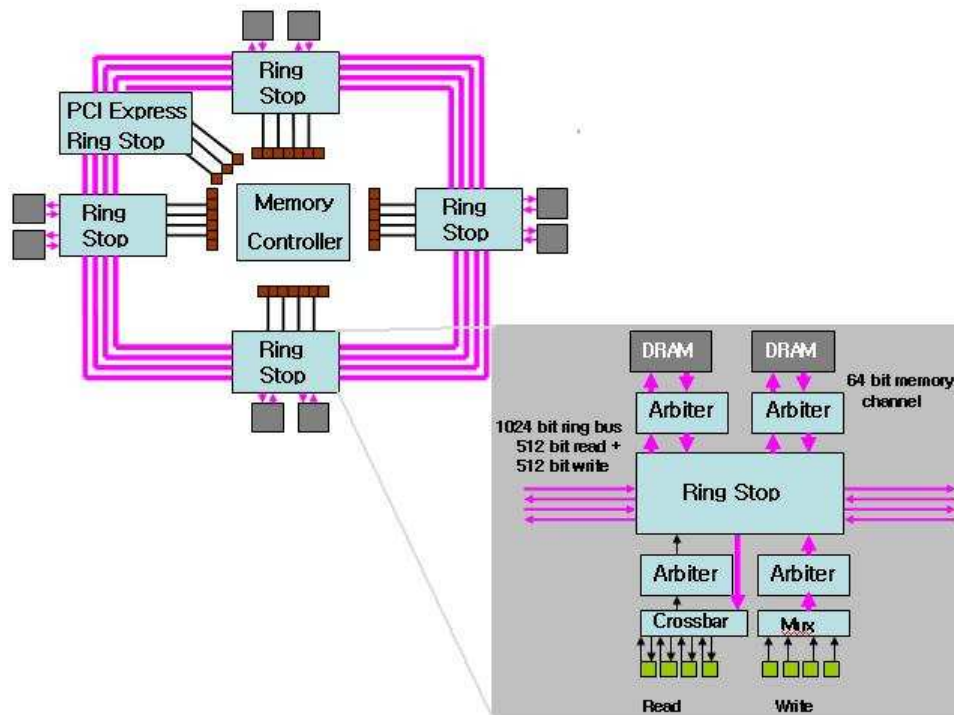


Figure III.1: Ring-bus diagram

the GPU, without modifying the attached memory devices. The ATI Technology has invented a ring bus memory controller that increases the memory performance, using all of these techniques.

Figure III.1 shows the ring-bus memory controller used in AMD ATI graphic cards. The controller is fully distributed instead of using a central arbiter. The internal memory bus uses ring topology and consists of four bidirectional ring buses. The memory is connected to the buses at the so-called “Ring Stops” [40]. Each Ring Stop sends out data to the requesting client, according to memory controllers instructions. The memory controller is divided up into eight separate 64-bit memory channels, for a total of 512-bits [30]. Each channel accesses portions of the frame buffer, and memory accesses are done by circling the ring bus to find the proper STOP. Four ring stops are used for local GPU memory and PCI Express Ring Stop is dedicated for accessing

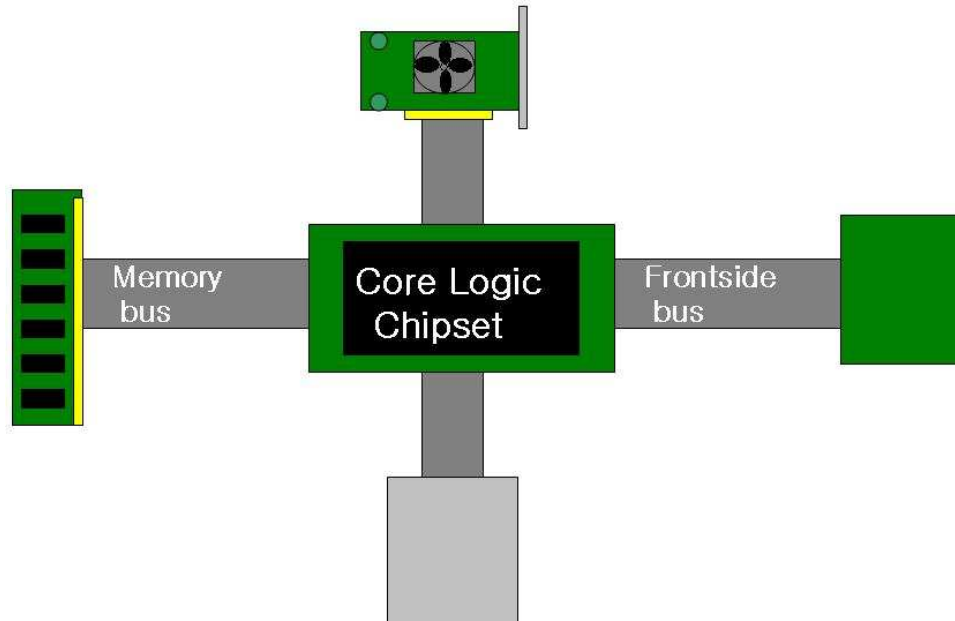


Figure III.2: PCI system layout

the PCI Express memory space in the CPU. From the GPU point of view, all memory space, including both the local and the remote one, are thought of as one linear memory space. More details are presented at the end of this chapter.

III.B PCIeExpress

We have mentioned the PCI Express on several occasions so far; in this section, we take a detailed look at PCI Express: what it is, and how it improves the interconnect scheme. Before we go into the PCI Express, it is worthwhile to take a look at the PCI system layout.

Figure V.5 shows the layout of a PCI system. “The core logic chipset acts as a switch or router, and it routes I/O traffic among the different devices that make up the system” [22]. In reality, the chipset is split into two parts: the Northbridge and the Southbridge. The Northbridge handles communication between CPU, main memory, and the video card, while the Southbridge routes traffic from different I/O

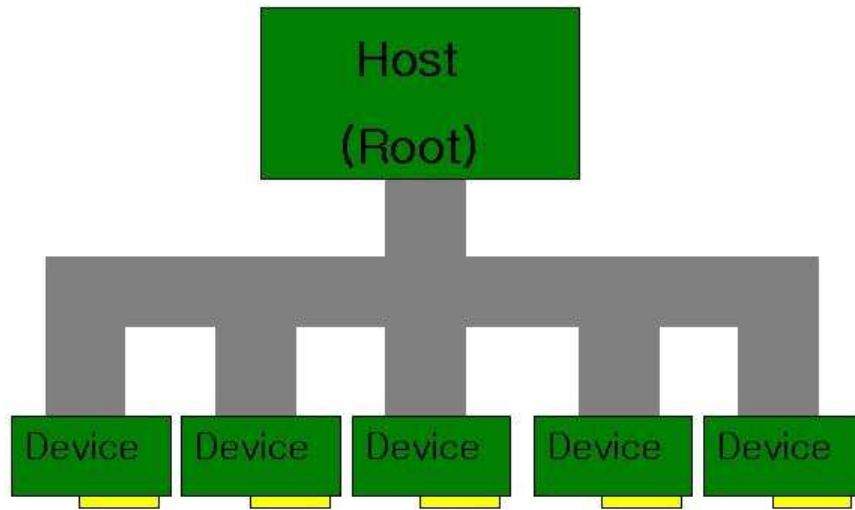


Figure III.3: The shared bus

devices such as, hard drive, USB ports, Ethernet ports, etc. The PCI bus is attached to the Southbridge and is the slowest bus in modern PC system. PCI uses a shared bus topology to allow communication among different I/O devices on the bus as shown in figure III.3.

The bus arbitration, which can be thought of as a scheduler, is in place to order which device gets access to the system. This scheme is very simple, cheap and easy-to-use; however, the main bottleneck is performance, especially when multiple devices try to access the system simultaneously.

PCI Express was invented to overcome this bottleneck. The main difference between PCI bus and PCI Express is that PCI Express uses a point-to-point bus topology instead of shared bus topology (Figure III.4).

In a point-to-point topology, a switch is the single shared resource, and each

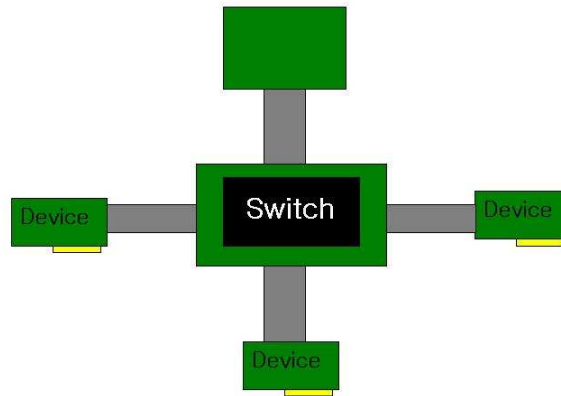


Figure III.4: The shared switch

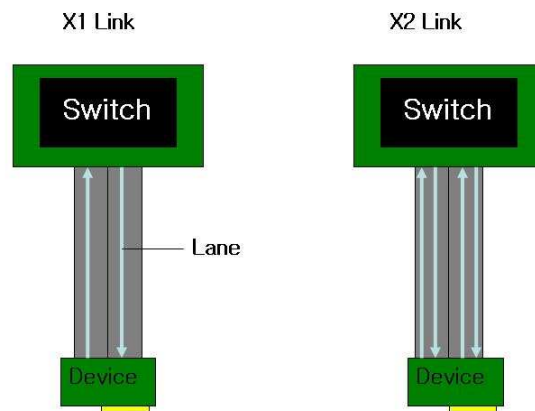


Figure III.5: Links and lanes

device has an exclusive access to the switch. The CPU can talk to any of the PCIe devices by accessing the address space of that device and opening up a direct communication link. “The fundamental building blocks that make up the fabric of the PCI Express interface are lanes and links” [16]. Two PCI Express devices are connected through a link which is composed of one or more lanes. Each lane is a dual-unidirectional communication channel between two PCI Express devices, and is capable of transmitting one byte at a time in both directions. (Figure III.5).

The biggest advantage of this scheme is that it allows aggregating multiple lanes to form a single link. In other words, you can double the bandwidth of a link by coupling two lanes together to form a single link.

III.C Hypermemory

The final piece of information we mention, is HyperMemory, a technology developed by ATI to use system memory as part of the video card’s frame-buffer memory. HyperMemory technique is not used in the high-end graphics cards. However, the concept of HyperMemory will help reader to understand how GPU accesses the system memory. HyperMemory relies on a fast data transfer mechanism within the PCI Express and has two key components [13].

- “Pre-emptive virtual memory along with the intelligent memory allocation to allow GPU to have less local memory without suffering from performance loss.”
- “A memory controller connected to the GPU’s PCI Express interface to allow the PCI Express interface treated as a part of GPU local memory resources.”

Figure III.6 illustrates the memory allocation and use by the application when HyperMemory memory manager is in place. Hypermemory manager preemptively removes unused memory blocks from the local GPU memory to higher-latency, remote system memory. It also re-orders memory allocations requested by the application, and intelligently decides where to actually store the data.

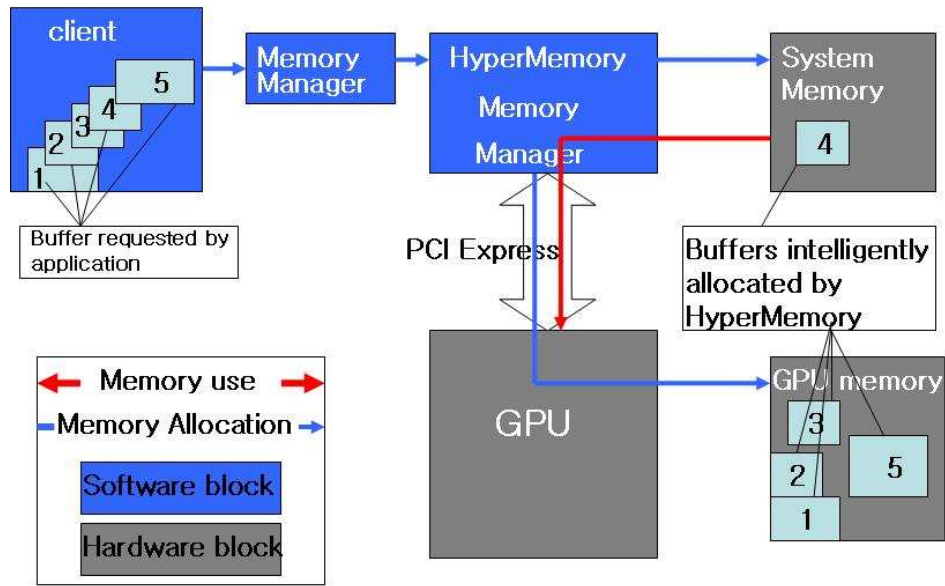


Figure III.6: Memory allocation and use with HyperMemory

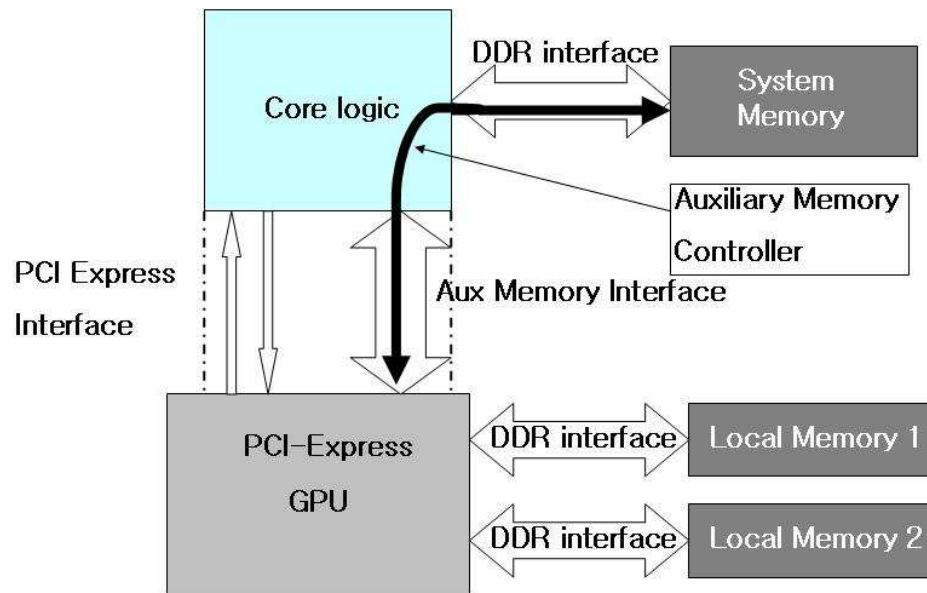


Figure III.7: PCI Express memory auxiliary memory channel

Figure III.7 illustrates how the PCI Express auxiliary memory channel fits into a system. There are two things that can be drawn from HyperMemory technique: The first is that the allocation of resources in system memory does not involve any interaction with the CPU. The CPU (or the operating system) initially decides and allocates a region of memory space (PCI Express memory space) that is accessible by GPU; however, as we have seen in Figure III.6, it is the memory manager who is responsible for allocating and storing resources within the PCIe memory space. The other thing to keep in mind is that the GPU do not have any knowledge of whether it is accessing a local GPU memory or a remote system memory. Instead, both of them are part of one large linear memory space from the GPU's stand point of view.

IV

Windows Memory Forensic

After understanding how the CAL system works, we need to next check whether the GPU is indeed writing onto the system memory, and if so, which region of the memory is written on.

In recent years, memory forensic has become one of the most popular techniques to identify offensive attacks such as viruses, worms and trojans. “Traditional forensics procedures examined hard disks by acquiring an exact sector-by sector copy for later analysis” [19]. However, acquiring disk image was not sufficient to detect any volatile information residing in random access memory (RAM), such as process information, network connections, command history, etc.

Our first intuition was that this memory forensic technique could help us understand how the GPU writes to the system memory.

In this chapter, we describe various techniques of acquiring and analyzing Windows physical memory, and the experience we’ve had dealing with them.

IV.A Cycle counts of read and write

Before using any of the forensic tools, we first ran a simple experiment to check how many cycles get consumed when the GPU reads or writes to the system memory. We then compared those numbers to the cycles counts consumed when the

CPU reads/writes to its own system memory. The cycle counts should not show a huge difference if the GPU is indeed writing to the system memory.

Table IV.A are the results gained from our experiments.

- **CPU-CPU without CAL** represents the case where the CPU reads/writes to its own system memory, without invoking the CAL system;
- **CPU-GPU** represents the case where the CPU reads/writes to the GPU memory under CAL;
- **CPU-CPU** represents the case where the CPU accesses its own memory under CAL;
- **GPU-CPU** represents the case where the GPU writes to the system memory.

We ran this experiment multiple times by changing the size of the allocated memory region (array). As we expected, the cycles consumed by the *write* operation were quite similar in all cases. However, the number for *read* operations was on a bigger order of magnitude under the CAL system. The explanation of this behavior is presented in chapter V

IV.B Acquisition of memory

In order to analyze volatile information, we first need to collect all data that resides in the RAM. There are two approaches to acquire physical memory images: hardware and software oriented.

IV.B.1 Hardware based acquisition tools

Hardware-based acquisition relies on a dedicated hardware (physical device) to access physical memory through a dedicated communication port. Two most widely used tools are: Tribble and FireWire bus.

Table IV.1: Cycle counts of read and write

Case	Write	Read
CPU-CPU without CAL	152	1256
CPU-GPU	472	600
CPU-CPU	224	1944
GPU-CPU	4308	N/A

(a) 1x1 array

Case	Write	Read
CPU-CPU without CAL	42920	36656
CPU-GPU	38232	341792
CPU-CPU	37656	367912
GPU-CPU	77432	N/A

(b) 64x64 array

Case	Write	Read
CPU-CPU without CAL	147856	134904
CPU-GPU	144656	1267776
CPU-CPU	144888	1288712
GPU-CPU	179448	N/A

(c) 128x128 array

Case	Write	Read
CPU-CPU without CAL	372000	323520
CPU-GPU	345776	3155488
CPU-CPU	348952	2940032
GPU-CPU	333432	N/A

(d) 196x196 array

Case	Write	Read
CPU-CPU without CAL	586312	530296
CPU-GPU	569904	5197191
CPU-CPU	570112	4943512
GPU-CPU	563416	N/A

(e) 256x256 array

- **Tribble.** This technique requires a dedicated PCI card, and needs to be installed before incident occurrence.
- **FireWire bus.** “FireWire bus, also known as IEEE 1394 bus, supports high speed communication and data-transfer, and physical access to system memory” [19].

The main advantage of the hardware-based acquisition technique is that it uses Direct Memory Access (DMA) to prevent any interaction with the operating system, and it gives the user the exact memory image. However, the installing process can be a little tricky and may not be worthwhile to use.

IV.B.2 Software based acquisition tools

- **Data Dumper (DD)** Data Dumper available on G.M. Garner website [20] is the most widely used software-based acquisition tool. This solution yields a very light footprint and is very easy to use. Data Dumper does not support the latest version of Windows, since DD retrieves the memory image by accessing the special device “\Device \Physical-Memory”, which is prohibited since Windows 2003 Service Pack 1. Recently, ManTech International Corporation released MDD, which generates a DD-style memory dump and supports Windows 2000, XP, Vista, and Windows Server [23]. The major drawback of Data Dumper is that it can last for several hours of operation, according to the size of the RAM.
- **Userdump** Microsoft also has its own tool for memory dump, called Userdump [34]. This software creates a dump file (.DMP format) of a process, instead of dumping the entire RAM, and is only compatible with Microsoft tools. If you are only interested in memory contents written by a specific process and are planning to use one of the Microsoft tools to analyze the dump file, Userdump should be the best tool to use.
- **Crash dump Utility** Last technique is to be used with the Windows crash dump utility. In Windows, when a crash dump occurs, the system state and the contents

of the memory are copied to a disk. The output file is in .DMP format and is only compatible with Microsoft tools. “Crashes can be easily induced through a specific keyboard shortcut when the following registry key [CrashOnCtrl-Scroll] is set to REG-DWORD” [31]. Currently, complete dumps are not available anymore and are mostly used for debugging purpose.

IV.C Memory Analysis

Collecting volatile memory is just the first step of the job: We need additional tools to read the contents of the memory image, and reconstruct important data structures from a raw memory file:

- **String.exe** This tool should only be used to search predefined key words, since it does not have the ability to reconstruct any data structures from raw memory files. However, it can still be useful to check whether certain contents have been written to memory.
- **BinText** BinText is also a text extractor that has the ability to find ASCII, Unicode text, and Resource strings [18]. This tool also has the ability to filter out unwanted text.
- **PTfinder** “Process and thread finder (PTfinder) is a Perl script created by Andreas Schuster that detects all running processes and threads at the time of RAM acquisition” [32]. This tool is useful in the way that it identifies `_EPROCESS` and `_ETHREAD` structures in Windows memory dump files. However, that is basically all it can do; it does not include any other features to help user perform a deep analysis on dump file. PTfinder supports all versions of Windows, from 2000 to Windows server 2003.
- **KnTTools with KnTList** This tool can both acquire the physical memory (KnTTools) and interpret the structures in memory (KnTList) [21]. I personally did not

have a chance to use this tool, since it is commercial software; however, it seems that KnTTools is equivalent to Data Dumper, and KnTList is similar to PTfinder.

- **Volatility** The Volatility Framework, originated from Volatools [35], is a completely open collection of tools, implemented in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory images (DD-style) acquired from Microsoft Windows XP SP2.[36].

The extraction capabilities from images include:

1. Date and time information from an image;
2. Running processes;
3. Open network sockets;
4. Open network connections;
5. DLLs loaded for each process;
6. Open files for each process;
7. OS kernel modules;
8. Mapping physical offsets to virtual addresses;
9. Virtual Address Descriptor (VAD) information.

This tool is much more powerful than PTfinder, since it does not only have the ability to detect all running processes, it can also identify all loaded modules for process and kernel. Mapping the physical offsets to virtual addresses, do not seem to work correctly. Volatility also has the ability to dump Virtual Address Descriptor (VAD) used by windows memory manager. VAD describes the memory ranges used by a process as they are allocated [17]. When a process allocates memory with VirtualAlloc, an entry in the VAD tree is created and the corresponding page directory and page table entries are not created unless the process references that particular entry in the VAD tree. This behavior can save a significant amount of memory space if a process allocates a huge amount of memory,

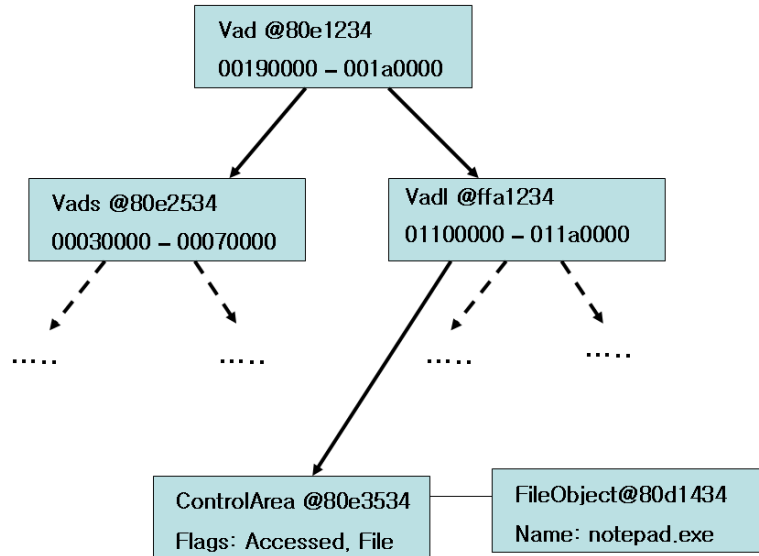


Figure IV.1: VAD tree for notepad.exe

but rarely accesses it. A VAD tree is a self balancing tree, with memory address lower than the current node staying in the left side of the sub-tree and higher one in the right (Figure IV.1).

I personally had a hard time extracting any useful information from VAD nodes. The tool is still at an early stage of development, and if one is interested in locating the physical page ranges associated with a process, one still needs to know the page directory and page table entries of a process. Nonetheless, Volatility was the most straightforward tool to use, and was especially useful in identifying loaded modules.

- **Windbg** Windbg, a debugger with a graphical user interface and a console interface, is one of the tools included in Microsoft Windows Debugging Tool[39]. Windbg supports analyzing the dump file with a .DMP format. Should one decide to use windbg, either “Userdump” or “Crashdump Utility” must be used to generate the dump file. The installing and learning phase of Windbg took significant amount of time; however, this tool seems to be more useful than any other tools

mentioned above. User can retrieve the assembly code of a process, stack information, register values, and the contents for specified virtual address. Windbg also supports retrieving the Virtual Address Descriptor (VAD) nodes related to a process. The list of windbg commands can be found at [15]

Memory forensic is a fairly new area that not many people are familiar with. It can be very useful in retrieving Process information, Network connections, Network status, Command history, Services/driver information and Logged users with their authentication credentials [33]. However, it is still at an early stage of development and it requires further research. The analysis can vary among different versions of Windows Operating System. In addition, the acquired memory image may not contain the information that users are interested in, since the pages in the system memory constantly get swapped in and out. Therefore, examiner should expect a significant amount of time and effort in using memory forensic technique.

The main goal of our memory analysis was to identify the physical memory region written by a process (or GPU), and to extract the flag settings of a page (Write-Copy, Write, Read, Execute, etc), in order to tell if the GPU is indeed writing to the system memory. Unfortunately, none of the tools were able to extract the page table information of a process. Nonetheless, we believe the contents of this chapter can still help many researchers to decide whether he/she should use this technique.

V

Windows Kernel Debugger

After trying out various memory forensic tools, we were still facing the same issue: How can we find the physical address ranges written by a process (or GPU)? In other words, we had to check whether the backing pages written by GPU were in the range associated with the System board, or in the range associated with the PCI bus. We decided to try using the Windows Kernel Debugger, which is also a part of Microsoft Debugging Tools. In this section, we first describe how to convert virtual address to physical address in an X86-system, followed by the experience and results we have obtained by using the Windows Kernel Debugger.

V.A Converting virtual address to physical address

Before we go into Windows Kernel debugger, it is worthwhile to take a look at how a virtual address gets translated into a physical address in X86 system.

Figure V.1 illustrates how a virtual address gets translated into a physical address on a non PAE-enabled X86 system. CR3, also known as a Page-Directory Base Register (PDBR), is the CPU's control register, which contains the physical base address of the page-directory, and each process has a distinct CR3 value.

The virtual address (32 bit address) is a combination of three fields [37]:

- Page Directory Index = Upper 10 bits of virtual address;

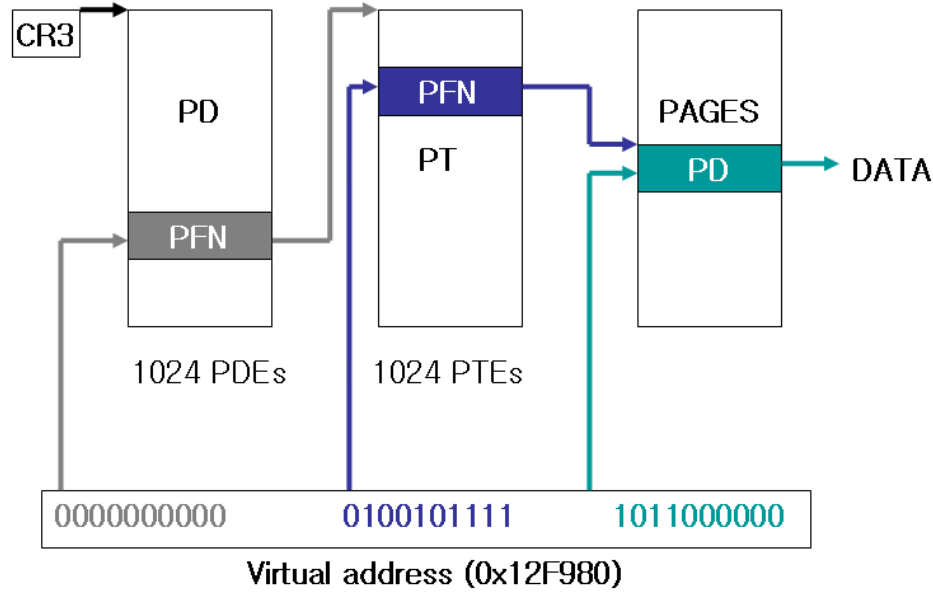


Figure V.1: Virtual to Physical (x86)

- Page Table Index = Next 10 bits of virtual address;
- Offset into data page pointed by PTE = Lower 12 bits.

CR3 register, along with upper 10 bits of virtual address, gives us the Page Frame Number (PFN) of Page Directory Entry (PDE). The value stored in the PFN is the base address of the Page Table Entry (PTE), and along with Page Table Index (next 10 bits of virtual address), the PFN of the PTE can be retrieved. The value stored in the PFN of the PTE points to the top of a 4 KB physical page that contains the data written by a process, and the lower 12 bits of the virtual address are then used to locate the physical address that holds the desired data.

Physical Address Extension (PAE) refers to a feature of x86 and x86-64 processors that allows more than 4GB of physical memory to be used in a 32-bit system [29]. In Windows XP Service Pack2 and later versions, the PAE mode is turned on by default on processors with no-execute (NX) or execute-disable (XD) features

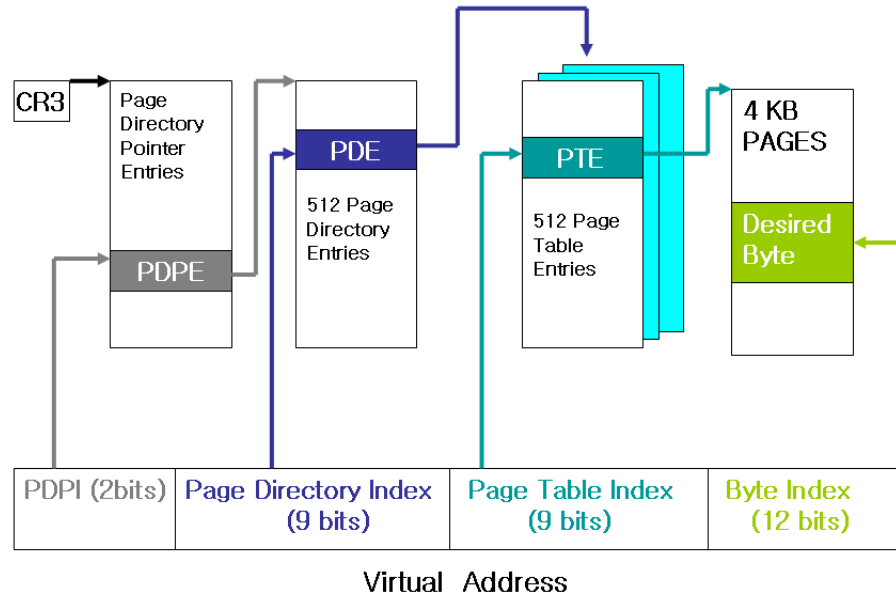


Figure V.2: Virtual to Physical in PAE enabled x86 system)

[28]. In PAE-enabled X86 system, virtual address translation uses a three-level address translation instead of two-level linear address [26]. Figure V.2 illustrates how address translation works in a PAE-enabled x86 system.

In a PAE-enabled system, virtual address is split into four components, instead of three, as in the case of a non-PAE system:

- Page Directory Pointer Index (PDPI) = Upper 2 bits of virtual address;
- Page Directory Index = Next 9 bits of virtual address;
- Page Table Index = Next 9 bits of virtual address;
- Offset into data page pointed by PTE = Lower 12 bits.

The CR (Control Register) now points the Page Directory Pointer Index (PDPI), which is 2-bits wide. The first two bits of virtual address are the index to the Page Directory Pointer Entry (PDPE), which stores a pointer to a Page Directory. The next 9 bits of virtual address are then used to retrieve the Page Directory Entry

(PDE) in Page Directory. The resulting PDE, along with the next 9 bits of virtual address, points to the Page Table Entry (PTE) in the Page Table. Finally, this PTE value points to a 4KB page in the memory and the lower 12 bits of virtual address works as an offset on that page to retrieve the desired data.

Once PAE mode is enabled, user may have trouble locating the Page Table Entries, since most of the tools (including Windbg and Kernel Debugger) expect the system to run in non-PAE mode. One can check whether a system is running on a PAE mode by running the Winver.exe, located in C: \Windows\ System folder or run *Volatility* described in chapter IV. The simplest solution is to just disable the PAE feature by adding “\ noexecute=Alwaysoff” clause in boot.ini file. The details of editing boot.ini file can be found at [25].

Even after understanding how address translation works, it is still challenging to recover the physical address, unless user can dump Page Frame Number (PFN) of Page Directory and Page Table. However, the user can at least recover the PTE address with three additional pieces of information:

- The size of the PTE, which is 4 bytes on non PAE x86 systems;
- The size of a page which is 0x1000 bytes;
- The PTE_BASE virtual address, which is 0xC0000000 on non PAE system.

Using the data above, the Page Frame Number of PTE can be retrieved by:

$$\begin{aligned}
 PTEaddress = PTE_BASE + (PageDirectoryIndex) * PAGE_SIZE \\
 + (PageTableIndex) * sizeof(PTE)
 \end{aligned}
 \tag{V.1}$$

V.B Windows Kernel Debugger

The next step was to find the tool that actually dumps the physical memory, in order to verify the contents stored in the area pointed by PTE. Windows Kernel Debugger is part of the Windows Debugging tool which is a set of extensible tools for debugging device drivers for the Microsoft Windows family of operating systems and is useful in debugging of [39]:

- “Applications, services, drivers, and Windows Kernel”.
- “Live targets and dump files”.
- “Microsoft Windows NT 4.0, Microsoft Windows 2000, and Windows XP”.
- “x86-based and Itanium-based target systems”.
- “Local and remote targets”.
- “User-mode programs and kernel-mode programs”.

There are two types of Windows Kernel Debugging: Local Kernel Debugging (LocalKD) and Live Kernel Debugging (LiveKD). Live Kernel Debugging requires two systems connected through a null modem cable, a communication method to connect two DTEs (computer, terminal, printer, etc.) [38]. One machine runs the debugger acting as a “host”, while the “target” machine runs the kernel to be debugged. On the other hand, Local Kernel Debugging only requires a single machine, and acts as both the “host” and “target” system. We decided to use the Local Kernel Debugging due to its simple installation process. The details of performing Local Kernel Debugging can be found at [27].

V.B.1 Commands and Results

In this section, we present some of the built-in commands supported by kernel debugger that helped us understand how GPU writes to system memory. The description of all supported commands can be found at [4] and [24].

```

C:\WINDOWS\system32\cmd.exe - kd -kl
Immunity Debugger
!kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8a3b9830 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: 0b140000 ObjectTable: e1001e78 HandleCount: 911.
Image: System

PROCESS 8a131da0 SessionId: none Cid: 0448 Peb: 7ffde000 ParentCid: 0004
DirBase: 1842a000 ObjectTable: e1da1280 HandleCount: 21.
Image: smss.exe

PROCESS 8a18fda0 SessionId: 0 Cid: 0490 Peb: 7ffde000 ParentCid: 0448
DirBase: 1a819000 ObjectTable: e175b250 HandleCount: 783.
Image: csrss.exe

PROCESS 88cc4da0 SessionId: 0 Cid: 04b8 Peb: 7ffde000 ParentCid: 0448
DirBase: 2261e000 ObjectTable: e178eb88 HandleCount: 613.
Image: winlogon.exe

PROCESS 8a188328 SessionId: 0 Cid: 04e4 Peb: 7ffda000 ParentCid: 04b8
DirBase: 22bce000 ObjectTable: e1f76df8 HandleCount: 351.
Image: services.exe

PROCESS 89f1b740 SessionId: 0 Cid: 04f0 Peb: 7ffde000 ParentCid: 04b8
DirBase: 22b95000 ObjectTable: e180f740 HandleCount: 448.
Image: lsass.exe

PROCESS 8a04d020 SessionId: 0 Cid: 05b8 Peb: 7ffde000 ParentCid: 04e4
DirBase: 233f5000 ObjectTable: e26464c0 HandleCount: 106.
Image: ati2evxx.exe

PROCESS 88c3fda0 SessionId: 0 Cid: 05cc Peb: 7ffde000 ParentCid: 04e4
DirBase: 23b40000 ObjectTable: e25fe570 HandleCount: 200.
Image: svchost.exe

PROCESS 8a0a0238 SessionId: 0 Cid: 0608 Peb: 7ffde000 ParentCid: 04e4
DirBase: 2377a000 ObjectTable: e260e2b0 HandleCount: 420.
Image: svchost.exe

PROCESS 8a032410 SessionId: 0 Cid: 00d4 Peb: 7ffdc000 ParentCid: 04e4
DirBase: 24040000 ObjectTable: e27a73c0 HandleCount: 1773.
Image: svchost.exe

```

Figure V.3: !process 0 0 command

In previous section, we described how to convert Virtual address to Physical address in x86 system. Fortunately, for those of you who are not familiar with address translation, the Windows Debugger offers two extensions that translate virtual address to physical address: *!vtop* and *!pte*. We tried using both of the extensions, however, only *!vtop* was successful in returning the correct physical address. The ***!vtop*** extension receives two inputs, the virtual address and the page frame number of the directory base of a process, and returns the corresponding physical address. The directory base of a process is the value stored in CR3 register, and can be retrieved via the *!process 0 0* command. As shown in Figure V.3, *!process* command dumps all active (running) processes. The Table V.B.1 describes the information returned by *!process 0 0*.

Figure V.4 shows the three steps to retrieve and verify the corresponding physical address. We first ran *!process 0 0 hellocal.exe* to retrieve the directory base

Table V.1: !process 0 0 output

Element	Meaning
Process	The hexadecimal address of the EPROCESS block.
Cid	The identification number of a process.
Peb	The hexadecimal address of the process environment block.
ParentCid	The identification number of the parent process.
DirBase	The value stored in CR3 register.
ObjectTable	The hexadecimal address of process object.
HandleCount	The number of handle references for the object.
Image	The module that owns the process.

of the process (hellocal.exe). The first parameter of the *!vtop* command is the page frame number (PFN) of the directory base, which is simply the directory base without three trailing zeroes (i.e. The PFN of “5ebbe000” is “5ebbe”). The virtual address “16d000” was retrieved through the *CalResMap* routine described in Chapter II. The *!vtop* command returns the page frame number, which is the address of the beginning of a 4KB physical page. In our example, this value is “1e740000”. We then need to add the byte index, the lowest 12 bits of the virtual address, to the returned page frame number. In our case the byte index is “000”. Now we finally have the corresponding physical address (“1e740000”) that holds the contents written by the GPU. The final step was to verify whether the computation was correct by looking at the contents of the address. The *!dd* command displays the memory at a specified physical address. The output of the *!dd* extension shows “24c” which is “588” in decimal (Figure V.4). This is the exact value that we were looking for, since we used the same application presented in Chapter II, which writes “588” to 256x256 array allocated in the system memory.

After retrieving the physical address range written by the GPU, we then checked whether this range was associated with System board or PCI bus. As we mentioned in Chapter II, the GPU only has access to the region of system memory specified by the operating system. We wanted to check whether the specified region is part of the System board or the PCI bus to get a sense which region of the system

```

C:\WINDOWS\system32\cmd.exe - kd -kl
lkd> !process 0 0 hellocal.exe
PROCESS 8a1bc270 SessionId: 0 Cid: 0650 Peb: 7ffda000 ParentCid: 07e8
DirBase: 5ebbe000 ObjectTable: e53e3920 HandleCount: 22
Image: hellocal.exe
lkd> !vtop 5ebbe 16d0000
Pdi 5 Pci 2a0
016d0000 1e740000 pfn(1e740)
lkd> !dd 1e740000
#1e740000 0000024c 0000024c 0000024c 0000024c
#1e740010 0000024c 0000024c 0000024c 0000024c
#1e740020 0000024c 0000024c 0000024c 0000024c
#1e740030 0000024c 0000024c 0000024c 0000024c
#1e740040 0000024c 0000024c 0000024c 0000024c
#1e740050 0000024c 0000024c 0000024c 0000024c
#1e740060 0000024c 0000024c 0000024c 0000024c
#1e740070 0000024c 0000024c 0000024c 0000024c
lkd>

```

Figure V.4: “!vtop” command

memory gets reserved for the PCI Express Interconnect. Figure V.5 shows the physical memory layout of our system, which can be accessed by opening the **Device Manager** in the Windows system. The physical address “1e740000” obtained via the *!vtop* extension, belongs to the memory range [00100000 - 7FEDFFFF] associated with the System board. In other words, the PCI Express memory, the memory region specified by the CPU or the operating system accessible by the GPU, is part of the memory region used by the System.

The *!pfn* is an extension provided by the Windows Debugger that displays information of a specified page frame. In Chapter IV, we showed an experiment that aimed to check how many cycles get consumed when the GPU reads or writes to the system memory. As a result, the *read* operation took an order of magnitude more cycles than the *write* operation. The *!pfn* extension helped us understand this behavior. Figure V.6 shows the output returned from the *!pfn* extension. The result shows that the *Read* and *Write* operations are in progress and the page flag is set to Write Combine (WriteComb). “Write combine is a computer bus technique for allowing data to be combined and temporarily stored in a buffer - the write combine buffer (WCB) - which is later to be released together in burst mode instead of writing (immediately) as single bits or small chunks” [3]. In other words, the number of cycles consumed during the *write* operation represents the time spent writing to the write combine buffer, instead

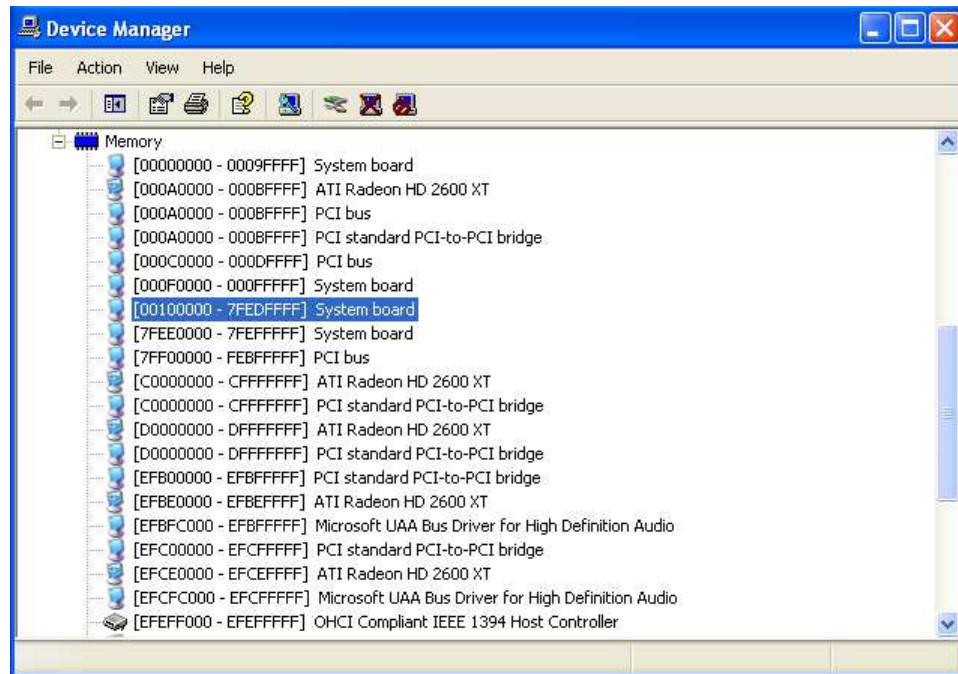


Figure V.5: Physical address range associated with PCI bus

```

C:\WINDOWS\system32\cmd.exe - kd -kl
lkd> !process 0 0 hellocal.exe
PROCESS 89edcda0 SessionId: 0 Cid: 0144 Peb: 7ffdd000 ParentCid: 0e14
DirBase: 5d624000 ObjectTable: e49144b0 HandleCount: 22.
Image: hellocal.exe

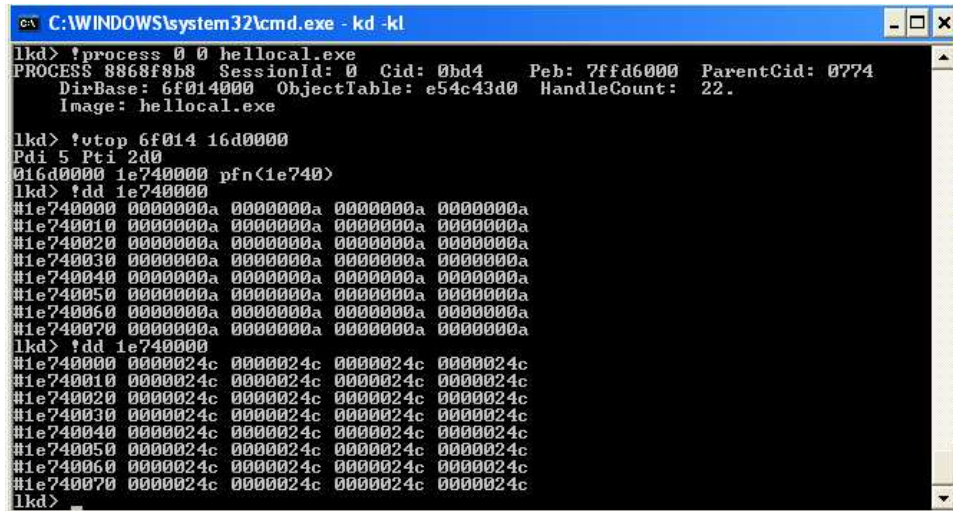
lkd> !vtop 5d624 1790000
Pdi 5 Pti 390
01790000 1f340000 pfn(1f340)
lkd> !dd 1f340000
#1f340000 0000024c 0000024c 0000024c 0000024c
#1f340010 0000024c 0000024c 0000024c 0000024c
#1f340020 0000024c 0000024c 0000024c 0000024c
#1f340030 0000024c 0000024c 0000024c 0000024c
#1f340040 0000024c 0000024c 0000024c 0000024c
#1f340050 0000024c 0000024c 0000024c 0000024c
#1f340060 0000024c 0000024c 0000024c 0000024c
#1f340070 0000024c 0000024c 0000024c 0000024c
lkd> !pfn 1f340
PFN 0001F340 at address 81AECE00
flink 00000000 blink / share count 00000001 pteaddress 0007CD01
reference count 0001 WriteComb color 0
restore pte 00000080 containing page FFEDCB Active RW
ReadInProgress WriteInProgress
lkd>

```

Figure V.6: !pfn output

of on the actual physical memory. On the other hand, the number of cycles consumed by the *read* operation represents the time spent fetching the data from the physical memory.

Our final experiment was to let the CPU write to the resource allocated in system memory by mapping the resource into the application's address space and then invoking GPU to overwrite the contents written by the CPU. Figure V.7 shows the result obtained from our experiment. We first let the CPU to write the hexadecimal value "a" to the allocated region, after mapping the resource onto the application's address space through the *CalResMap* routine and then invoking GPU to write "24c" to the same region, without un-mapping the resource. As shown in Figure V.7, the contents written by the CPU do get overwritten by the GPU. This experiment assured us that when a resource is mapped onto the CPU side and the CPU is accessing it, the



```

C:\WINDOWS\system32\cmd.exe - kd -kl
lkd> !process 0 0 hellocal.exe
PROCESS 8868f8b8 SessionId: 0 Cid: 0bd4 Peb: 7ffd6000 ParentCid: 0774
DirBase: 6f014000 ObjectTable: e54c43d0 HandleCount: 22.
Image: hellocal.exe

lkd> !vtop 6f014 16d0000
Pdi 5 Pti 2d0
016d0000 1e740000 pfn<1e740>
lkd> !dd 1e740000
#1e740000 0000000a 0000000a 0000000a 0000000a
#1e740010 0000000a 0000000a 0000000a 0000000a
#1e740020 0000000a 0000000a 0000000a 0000000a
#1e740030 0000000a 0000000a 0000000a 0000000a
#1e740040 0000000a 0000000a 0000000a 0000000a
#1e740050 0000000a 0000000a 0000000a 0000000a
#1e740060 0000000a 0000000a 0000000a 0000000a
#1e740070 0000000a 0000000a 0000000a 0000000a
lkd> !dd 1e740000
#1e740000 0000024c 0000024c 0000024c 0000024c
#1e740010 0000024c 0000024c 0000024c 0000024c
#1e740020 0000024c 0000024c 0000024c 0000024c
#1e740030 0000024c 0000024c 0000024c 0000024c
#1e740040 0000024c 0000024c 0000024c 0000024c
#1e740050 0000024c 0000024c 0000024c 0000024c
#1e740060 0000024c 0000024c 0000024c 0000024c
#1e740070 0000024c 0000024c 0000024c 0000024c
lkd>

```

Figure V.7: GPU writing to memory region written by CPU

GPU writes to that resource are propagated to the CPU's side.

VI

Conclusion

We have explored how the AMD Stream Processor interacts with the system memory, and particularly how the stream processor writes to the system memory. During our experiments, we realized that AMD Stream Processor has a complex architecture and requires further research to discover whether malicious code can be injected through the GPU without notice. Below, we provide the future work that is necessary to identify possible vulnerabilities, followed by the summary of the thesis.

VI.A Future Work

Our initial goal was to identify whether the AMD Stream Processor writes to the system memory independent of CPU, and whether a malicious program sitting on the GPU side can overwrite the system memory without being noticed by the virus scanners. It is hard to conclude whether the GPU performs direct memory access (DMA) to the system memory. We do know that GPU writes to the system memory through the PCI Express Interface; however, we have to make sure that CPU does not get involved while the *writing* happens. One possible solution would be to disable the interrupts imposed by the CPU, and see whether GPU can still write to the system memory. If so, we can conclude that the GPU does write to the system memory, independent of the CPU. In the Linux system, the kernel provides functions for enabling

and disabling interrupts:

- `void enable_irq(int irq);`
- `void disable_irq(int irq);`

Unfortunately, our experiment was done under Windows XP SP2, and we could not find a way to disable the interrupts.

The next thing to explore is to discover how the CPU and the GPU (or memory controller in GPU) initially negotiate to setup the range of system memory accessible by the GPU. In Chapter V, we showed the range of the system memory accessible by the GPU. However, as far as hackers are concerned, writing arbitrary code to the area specified by the operating system is not going to be sufficient to take over the system. Instead, if one can find a way to overwrite the kernel space memory without gaining the “root” privilege, any malicious code can have devastating effects on the system: One can either modify or delete the system settings, and can even obtain critical information.

Programmability is one of the most interesting aspects of modern GPU architecture. In other words, GPU itself can contain the malicious code (kernel), without using the system memory. Virus scanning programs may have a hard time detecting the code, if the program sits on the GPU side, as it periodically overwrites the contents in system memory. In addition, if the GPU can overwrite the contents in the kernel space memory, no virus scanner will be able to detect or prohibit this behavior, since virus scanner itself is a user-land application, and do not have access to the kernel space.

Injecting the malicious code inside a game program will be an interesting experiment to prove this vulnerability. As 3D computer games are becoming the norm, not only the performance of the GPU, but the interaction between the GPU and the CPU is also becoming the crucial aspect for running the game programs. The GPU must not only perform image and data processing promptly, but the data transfer be-

tween the GPU and the CPU must also be done swiftly. If the game program contains a high level code (Brook+ or CAL) or the assembly code that invokes GPU to periodically overwrite the contents of the system memory, the system can get infected just by running the game program, and without notice.

VI.B Summary

Modern Graphics Cards have begun to expose their architecture through hardware APIs in order to help developers write non-graphics applications that can take advantage of fast, highly-parallel GPUs. AMD has released AMD Stream Processor with a “Close-to-metal hardware abstraction, which later evolved into Compute Abstraction Layer (CAL); NVIDIA also developed a Compute Unified Device Architecture (CUDA), which is a compiler and set of development tools that allow developers to use a variation of C code kernel for execution on the GPU in NVIDIA GeForce graphics card [1]. As the usage of GPU gets broader, the interaction between GPU and system memory will get more attention, since the memory operations always cause a bottleneck, compare to faster running processors (GPU or CPU).

In this thesis, we explored how AMD Stream Processor interacts with the system memory, particularly how the GPU writes to the system memory. We first described the Compute Abstraction Layer (CAL), a hardware abstraction used by AMD Stream Processors, along with a sample application that invokes the GPU to write its output to the system memory via CAL APIs. To understand the interaction between the GPU and the CPU system memory, we described the Memory Controller (MC) attached in AMD Stream Processors, followed by the details of the PCI Express Interconnect scheme. We also described various techniques we have used, including memory forensic, and Windows Kernel Debugger, for further understanding. From the experiments, we showed that the physical address range of system memory accessible by the GPU is part of the range used by System board. We also showed that the GPU can, indeed, overwrite the contents initially written by the CPU.

At this point, it is hard to conclude whether there is any vulnerability in the AMD Stream Processors architecture. Nonetheless, we believe that this paper can only help researchers in computer security community who are particularly interested in exploiting vulnerabilities in the graphics card, as well as prove useful for developers who are planning to create applications that utilize the AMD Stream Processors for numerically intensive computations.

Bibliography

- [1] Cuda. <http://en.wikipedia.org/wiki/CUDA>.
- [2] Gpgpu. <http://en.wikipedia.org/wiki/GPGPU>.
- [3] Write-combining. <http://en.wikipedia.org/wiki/Write-combining>.
- [4] Addison-Wesley. Undocumented windows 2000 secrets-appendix a. <http://undocumented.rawol.com/>, 2001.
- [5] AMD Inc. Amd close-to-the-metal programming guide.
- [6] AMD Inc. Amd compute abstraction layer programming guide. <http://ati.amd.com/technology/streamcomputing/sdkdnld.html>, 2008.
- [7] AMD Inc. Amd intermediate language (il) reference. <http://ati.amd.com/technology/streamcomputing/sdkdnld.html>, 2008.
- [8] AMD Inc. Amd stream computing software stack. ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf, 2008.
- [9] AMD Inc. Amd stream sdk. <http://ati.amd.com/technology/streamcomputing/sdkdnld.html>, 2008.
- [10] AMD Inc. Brook+ programming guide. <http://ati.amd.com/technology/streamcomputing/sdkdnld.html>, 2008.
- [11] AMD Inc. Cal image. <http://ati.amd.com/technology/streamcomputing/sdkdnld.html>, 2008.
- [12] AMD Inc. Cal platform. <http://ati.amd.com/technology/streamcomputing/sdkdnld.html>, 2008.
- [13] ATI Inc. Hypermemory-next generation memory management for pci express graphics. http://ati.amd.com/technology/HyperMemory_Whitepaper.pdf, 2004.

- [14] ATI Technology. Radeon x1800 memory controller. http://ati.amd.com/products/radeonx1k/whitepapers/X1800_Memory_Controller_Whitepaper.pdf, 2005.
- [15] Exploring windows 2000 memory. <http://undocumented.rawol.com/sbs-w2k-4-exploring-windows-2000-memory.p%df>.
- [16] W. Central. Pci express and windows. http://www.microsoft.com/whdc/system/bus/PCI/PCIE_Windows.mspx, 2004.
- [17] B. Dolan-Gavitt. The vad tree: A process-eye view of physical memory. Science Direct - Digital Investigation Volume 4, 2007.
- [18] Foundstone, Inc. Bintext. <http://www.foundstone.com/us/resources/proddesc/bintext.htm>, 2003.
- [19] G. L. Garcia. Forensic physical memory analysis: an overview of tools and techniques. TKK T-110.5290 Seminar on Network Security, 2007.
- [20] Garner George m. forensic acquisition utilities. <http://www.gmgsystemsinc.com/fau>, 2008.
- [21] GMC Systems, Inc. Knttools with kntlist. <http://gmgsystemsinc.com/knttools/>, 2005.
- [22] jon stokes. Pci express:an overview. <http://arstechnica.com/articles/paedia/hardware/pcie.ars>, 2004.
- [23] mdd. Mantech international corporation. https://sourceforge.net/project/showfiles.php?group_id=228865&release_id=618037, 2008.
- [24] microsoft. debugger. Help file in Debugging Tools for Windows.
- [25] microsoft Help and Support. How to edit the boot.ini file in windows xp. <http://support.microsoft.com/kb/289022>, 2003.
- [26] microsoft TechNet. How pae x86 works. <http://technet.microsoft.com/en-us/library/cc736309.aspx>, 2003.
- [27] Msdn. Performing local kernel debugging. <http://msdn.microsoft.com/en-us/library/cc266422.aspx>.
- [28] Nx bit. http://en.wikipedia.org/wiki/NX_bit.
- [29] Deploying windows xp service pack2 using software update services. [http://technet.microsoft.com/ko-kr/library/bb457097\(en-us\).aspx](http://technet.microsoft.com/ko-kr/library/bb457097(en-us).aspx), 2004.

- [30] PC perspective. Amd ati radeon hd 2900 xt review:r600 arrives. <http://www.pcper.com/article.php?type=expert&aid=406>, 2005.
- [31] N. Ruff. Windows memory forensics. SSTIC 2007 BEST ACADEMIC Papers, 2008.
- [32] A. Schuster. Searching for prcoesses and threads in microsoft windows memory dumps. Digital forensic research workgroup, 2007.
- [33] I. Sutherland, J. Evans, T. Tryfonas, and A. Blyth. Acquiring volatile operating system data tools and techniques. *SIGOPS Oper. Syst. Rev.*, 42(3):65–73, 2008.
- [34] Microsoft userdump. <http://support.microsoft.com/kb/241215/en-us>, 2008.
- [35] Volatools. <http://computer.forensikblog.de/en/2007/03/volatools.html>, 2007.
- [36] Volatile systems. http://computer.forensikblog.de/en/2008/06/volatility_1_1_2.html, 2008.
- [37] Converting virtual addresses to physical addresses. <http://msdn.microsoft.com/en-us/library/cc267483.aspx>.
- [38] Wikipedia. Null modem. http://en.wikipedia.org/wiki/Null_modem_cable.
- [39] Microsoft windows debugging tools. <http://technet.microsoft.com/en-us/library/bb742599.aspx>.
- [40] Xbit laboratories. Ati radeon x1000: Brand-new graphics architecture from ati explored (page 6). http://www.xbitlabs.com/articles/video/display/radeon-x1000_6.html#sect%0, 2005.
- [41] J. Yang. Amd stream computing. www.sharcnet.ca/events/ssgc2008/presentations/yang-sharcnet-final.pdf.