

Massively Threaded Digital Forensics Tools

Lodovico Marziale¹, Santhi Movva², Golden G. Richard III¹,
Vassil Roussev¹, Loren Schwiebert²
{vicodark@gmail.com, santhimovva@wayne.edu, golden@cs.uno.edu,
vassil@cs.uno.edu, loren@cs.wayne.edu}

¹Dept. of Computer Science
University of New Orleans
New Orleans, LA 70148

²Dept. of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

Digital forensics comprises the set of techniques to recover, preserve, and examine digital evidence and has applications in a number of important areas, including investigation of child exploitation, identity theft, counter-terrorism, and intellectual property disputes. Digital forensics tools must exhaustively examine and interpret data at a low level, because data of evidentiary value may have been deleted, partially overwritten, obfuscated, or corrupted. While forensics investigation is typically seen as an off-line activity, improving case turnaround time is crucial, because in many cases lives or livelihoods may hang in the balance. Furthermore, if more computational resources can be brought to bear, we believe that preventative network security (which must be performed on-line) and digital forensics can be merged into a common research focus. In this chapter we consider recent hardware trends and argue that multicore CPUs and Graphics Processing Units (GPUs) offer one solution to the problem of maximizing available compute resources.

1 Introduction

The complexity of digital forensic analysis continues to grow in lockstep with the rapid growth of the size of forensic targets—as the generation of digital content continues at an ever-increasing rate, so does the amount of data that ends up in the forensic lab. According to FBI statistics (Federal Bureau of Investigation, 2007), the average amount of data examined per criminal case has been growing at an average annual rate of 35%—from 83GB in 2003 to 277GB in 2007. However, this is just the tip of the iceberg—the vast majority of forensic analyses are in support of either civil cases or internal investigations and can easily involve the examination of terabyte-scale data sets.

Ultimately, a tiny fraction of that information ends up being relevant—the proverbial ‘needle in a haystack’—so there is a pressing need for high-performance forensic tools that can quickly sift through the data with increasing sophistication. As an illustration of the difficulty of the problem, consider the 2002 Department of Defense investigation into a leaked memo with Iraq war plans. It has been reported (Roberts, 2005) that a total of 60TB of data were seized in an attempt to identify the source. Several months later, the investigation was closed with no results. The Enron case involved over 30TB of raw data and took many months to complete. While these examples might seem exceptional, it is not difficult to come up with similar, plausible scenarios in a corporate environment involving large amounts of data. As media capacity continues to double every two years, such huge data sets will be increasingly the norm, not the exception.

Current state-of-the-art forensic labs use a private network of high-end workstations backed up by a Storage Area Network as their hardware platform. Almost all processing for a case is done on a single workstation—the target is first pre-processed (indexed) and subsequently queried. Current technology trends (Patterson, 2004) unambiguously render such an approach as unsustainable: I/O capacity increases at a significantly faster rate than corresponding improvements in throughput and latency.

This means that, in relative terms, we are falling behind in our ability to access data on the forensic target. At the same time, our raw hardware capabilities to process the data have kept up with capacity growth. The basic problem that we have is two-fold: a) current tools do a poor job of maximizing compute resource usage; b) the current index-query model of forensic computation effectively neutralizes most of the gains in compute power by traversing the I/O bottleneck multiple times.

Before we look at the necessary changes in the computational model, let us briefly review recent hardware trends. Starting in 2005, with the introduction of a dual-core Opteron processor by AMD, single-chip multiprocessors entered the commodity market. The main reason for their introduction is that chip manufacturing technologies are approaching fundamental limits and the decades-old pursuit of speedup by doubling the density every two years, a.k.a. keeping up with Moore's Law, had to make a 90 degree turn. Instead of shrinking the size and increasing the clock rate of the processor, more processing units are packed onto the same chip and each processor has the ability to simultaneously execute multiple threads of computation. This is an abrupt paradigm shift towards massive CPU parallelism and existing forensic tools are squarely not designed to take advantage it.

Another important hardware development that gives us a peek into how massively parallel computation on the desktop will look in the near future is the rise of Graphics Processing Units (GPUs) as a general-purpose compute platform. GPUs have evolved as a result of the need to speedup graphics computations, which tend to be highly parallelizable and follow a very regular pattern. As a result, GPU architectures have followed a different evolution from that of the CPU. Instead of having relatively few, very complex processing units and large caches, GPUs have hundreds of simpler processing units and very little cache on board.

For many years, GPU manufacturers produced very specialized, idiosyncratic hardware that was narrowly tailored for its graphics workload and was incompatible across generations. This all changed in 2007 with the introduction of the NVIDIA G80 processor and the accompanying release of the CUDA development kit. CUDA enabled general-purpose computation in the GPU using a dialect of the C language. The general-purpose use of GPUs is a significant development. In terms of raw computational power, current generation GPUs can achieve a theoretical rate of 1000 GFLOPS, whereas CPUs max out around 50 GFLOPS. Figures 1 and 2 illustrate the peak computational speed (in GFLOPS) and peak memory bandwidth (in GB/sec) of some NVIDIA GPUs and Intel CPUs. The performance of ATI's GPUs are similar, although the programming interfaces are different; AMD recently demonstrated a system with dual R600 GPUs that achieved 1 TFLOP and ATI has since released even faster GPUs.

This is not to suggest that GPUs will supplant CPUs as the primary compute engines in commodity computer systems, but rather that GPUs make available significant computational resources that can be harnessed to speed up a variety of application workloads. Specifically, GPUs can be used to significantly speed up digital forensics and network security applications with only modest associated costs.

As we discussed above, the current index-then-query computational model for processing digital evidence, which closely resembles that of traditional databases, is unsustainable. This, unfortunately, is well illustrated by the backlogs at large government forensics labs (where backlogs are measured in months). While there are multiple contributing factors for these backlogs, waiting for days for current-generation forensics tools to index a large target is certainly a serious contributor.

The only reasonable alternative is to use a stream-oriented processing model in which only one or two *sequential* passes over a forensic target are made. Data is read at the maximum sustained rate for the media, and all processing is performed at line speed. For current generation large hard drives, the reference rate is about 100MB/s, which means that the forensic tool must do all required processing at that rate. Obviously, this is a challenging task and is orders of magnitude outside the capability of current tools in use. The good news, however, is that this is a sustainable model because improvements in CPUs/GPUs play directly into its strengths and give us a chance to keep turnaround time steady in the face of growing forensic target size.

Regardless of the specific architectural model adopted by a forensic tool, it should be clear by now that it needs to be able to take maximum advantage of all the hardware parallelism available on the platform. For the rest of this chapter, we survey recent research in the area of high-performance forensic tools and outline the essential ideas that will form the basis of next-generation tools. By and large, these are proof-of-concept research prototypes that implement specific functions and are not yet ready to completely replace existing software. They do, however, demonstrate the tremendous potential of well-designed massively-threaded applications to significantly speedup the forensic process.

2 Background

2.1 GPU Programming for General Purpose Computing

Until recently, general purpose programming for GPUs was both difficult and typically targeted at very specific problems. To perform non-graphical calculations required techniques that recast data as textures or geometric primitives and expressed the calculations in terms of available graphics operations. Other difficulties included non-IEEE compliant floating point representations, the lack of integer arithmetic, and lack of support for random memory writes.

Newer GPUs, including the Gxx series from NVIDIA and the R600 from ATI, have large numbers of scalar processors and excel at executing massively threaded algorithms. In addition, both NVIDIA and ATI have released high-level SDKs for general purpose GPU programming. NVIDIA's SDK is the Common Unified Device Architecture (CUDA) (NVIDIA, 2008), which allows NVIDIA GPUs to be programmed using a re-

stricted subset of C. ATI's GPU programming SDK is Brook+ (ATI, 2008), based on Brook, which is itself an extension to C that was originally developed at Stanford University for parallel computation (Brook, 2008). We will discuss CUDA in more detail in Section 3, because most of the existing work on accelerating security applications is currently based on the NVIDIA CUDA platform.

Even using the new SDKs for general purpose GPU programming, such as CUDA and Brook+, programmers face a relatively steep learning curve and must also pay close attention to the architectural details of the GPU. Achieving maximum performance when offloading computations onto modern GPUs still requires careful attention to a number of details. Modern GPUs use a Single Instruction Multiple Thread (SIMT) execution model, a slightly less restrictive version of the more widely known Single Instruction Multiple Data (SIMD) model. Transfer bandwidth between the host and the GPU is limited by bus speed, mandating that GPU computations have significant "arithmetic intensity" to offset the cost of moving data between the host and GPU. Furthermore, modern GPUs have complicated memory hierarchies, with available memory divided into a number of classes with varying sizes, caching strategies, and access costs. The purpose of this chapter is to motivate our belief that despite these difficulties, the programming effort associated with GPU programming is worthwhile and that proper utilization of GPUs can have a positive impact on the performance of security tools.

2.2 Current Generation Digital Forensics Tools

A variety of digital forensics tools are in wide use, including "point and click" forensics suites such as AccessData's Forensic Toolkit (FTK) and Guidance Software's Encase Forensic Edition, low-level filesystem analysis tools such as Brian Carrier's Sleuthkit, evidence correlation tools (Garfinkel, 2006), and file carving applications (Richard, 2005). We briefly describe file carving here and then motivate why studying file carving provides a good basis for investigating performance-enhancing techniques for digital forensics.

File carving applications use a database of descriptive information for a number of file types of interest to perform file recovery. The descriptive information typically includes file headers and footers (predictable strings that appear at the beginning and end of files of a particular type) and other relevant binary strings. File-type specific rules may be applied to match the binary strings for example, to associate the footer closest to a discovered header or farthest away from the header, inclusion or exclusion of the footer in the carved file, the presence of recognizable milestones, or evaluation of other syntactical or semantic information. The database is used to search physical disks or disk images for recognized file types, with the goal being identification of the starting and ending locations of interesting files in the disk images. Once the locations are known, the file carving application can "carve" (copy) the corresponding sequences of bytes into regular files.

File carving is a particularly powerful technique because files can be retrieved from raw disk images, regardless of the type of filesystem, even if file metadata has been destroyed. For example, a file deposited on a FAT partition can often be recovered even if the partition is reformatted as NTFS, then ext2, then FAT again, even if bad block checks (which are generally read-only operations) are applied. While a filesystem's metadata can be quite fragile, file data is much more resilient. In addition, file carving can find

data stored in slack space, swap files, and data “hidden” outside the filesystem. File carving is applicable to both “dead” machines—machines shut down for investigation—and “live” machines.

Treating file carving as a “typical” digital forensics technique makes sense in our work because issues common to many forensics tools (particularly with respect to performance) arise. First, disk activity must be minimized, since file carvers typically must make multiple passes over a disk image. Simple header/footer-based file carving may make only two sequential passes, but more sophisticated processing such as file fragmentation support may require additional passes. Second, file carving applications must perform very efficient binary string searches, because a number of patterns must be matched against a large amount of binary data. The fact that current generation file carvers typically err on the side of completeness and generate many false positives simply increases the number of binary string comparisons that must be performed. Finally, the sophistication of file carving is increasing, with the development of techniques for effectively reducing false positives (through verification or deeper analysis of type-specific file structures) and detecting and processing fragmented files. These new techniques will in turn require more computational resources, making file carving applications less I/O-bound and even more compute intensive. This trend of deeper processing of forensic evidence, led by researchers increasingly interested in the field of digital forensics, naturally results in an increased need for computational resources to process evidence. By creating massively threaded digital forensics tools that specifically take advantage of modern multi-core CPUs and GPUs, these needed computational resources can be effectively provided.

2.3 GPU-enhanced Security Applications

The application of GPUs to increasing the performance of security software is still in its infancy. We believe that the primary reason for this is that usable APIs for truly general-purpose computing on GPUs are still quite new. Prior to the introduction of NVIDIA’s CUDA and ATI’s Brook+ SDKs for GPU programming, there were substantial barriers to effective utilization of GPUs for many non-graphical applications. While GPU programming is still not straightforward, we anticipate that in the near future there will be increased interest in the use of GPUs for enhancing the performance of computer security tools. This section briefly examines some of the published work to date. Section 4 discusses some of this work in more detail.

2.3.1 Forensics

GPU-enhanced Scalpel (Marziale, 2007) was the first digital forensics tool to rely on GPUs for speeding evidence processing. The initial work used NVIDIA 8800GTX GPUs and a simple binary string search algorithm to perform parallel header/footer searches and speed up a popular file carving application. Both the preliminary work and enhancements made since the publication of that paper are discussed in the Case Studies in Section 4. We are not aware of any other forensics tools that currently take advantage of GPUs for enhancing performance, but expect to see more tools in the near future.

2.3.2 Network Intrusion Detection

Jacob and Brodley (Jacob, 2006) describe PixelSnort, a port of the popular open-source network intrusion detection system (NIDS), Snort. Their system offloads IDS packet processing (specifically, comparison with Snort rules) from the host's CPU to an NVIDIA 6800GT. Jacob and Brodley convert Snort rules to textures and then use the fragment processors to match network packets with rules. While this work is important for raising awareness of the possible impacts of GPU programming on network security tools, PixelSnort offers only modest performance gains, limited by the complicated software architecture dictated by the 6800GT's lack of direct support for general computations.

A subsequent port of Snort described in (Vasiliadis, 2008) targets NVIDIA Gxx series GPUs and takes advantage of the CUDA SDK to offer improved NIDS performance. This work offloads pattern matching against network packets to the GPU and uses a multi-pattern matching algorithm to increase performance by a factor of 2-3 over an unmodified snort installation. gnort also takes advantage of enhancements made in the later generation NVIDIA GPUs to overlap CPU and GPU operations. Section 4 discusses gnort in greater detail, examining both the architecture and some of the associated design decisions.

2.3.3 Encryption on GPUs

Recently, (Manavski, 2007) demonstrated that GPUs can be used for efficient implementation of symmetric key encryption schemes, such as AES. In this work, two approaches were considered: a "traditional" OpenGL-based implementation running on an ATI X1900 and a CUDA implementation on an NVIDIA G80. For the "traditional" approach on ATI hardware, data had to be mapped into the OpenGL memory model, which uses floating point for calculations. Every byte of the input was mapped into a 16-point floating point number in texture memory. A further complication is the lack of bitwise operations on the programmable shaders of the ATI card, necessitating the use of lookup tables. This approach fared poorly not only because of these limitations, but because host $\leftrightarrow$ GPU bandwidth is very limited on the ATI X1900, making the OpenGL implementation uncompetitive with an implementation on a 3GHz Pentium 4.

For the CUDA implementation, many of the limitations encountered in the ATI X1900 study were relaxed. For example, the G80 has a 32 bit logical XOR operation, which greatly increases the speed of bit operations over the use of a lookup table. A single AES round in the CUDA implementation can be performed with 16 table look-ups and 16 32-bit XOR operations. The CUDA implementation is still limited by host $\leftrightarrow$ GPU bandwidth, but the available bandwidth is substantially higher than for the X1900. This implementation delivered maximum performance of approximately 8.3Gb/sec on 8MB inputs with AES-128 and approximately 6.6Gb/sec on 8MB inputs with AES-256. This is approximately 20X faster than an implementation of AES-128 and 15X faster than an implementation of AES-256 on a 3GHz Pentium 4.

2.3.4 Password Cracking

Elcomsoft has released a parallel password cracking application based on NVIDIA's CUDA SDK for computers equipped with NVIDIA GPUs. The product can be used to

crack a number of different password schemes, including Windows passwords and WPA/WPA2 wireless networking keys. With two GeForce GTX-280 graphics cards, WPA2 keys can be cracked approximately 100X faster than with a typical CPU-based approach. Since the software supports a scalable, distributed attack, with support for multiple GPUs per node, rapid brute force attacks become feasible for a number of encryption schemes.

3 Toward Massively Threaded Tools

Improving the performance of the next generation of digital forensics tools will involve harnessing all available computational resources, including multicore CPUs, and cluster-based resources (e.g., Beowulf clusters). Traditional multithreading (e.g., using the popular PThreads library), appropriate for programming multicore CPUs, and distributed computing on clusters are both fairly well understood. Researchers and commercial tool developers must simply adopt these paradigms and adapt them to the processing of digital evidence. With commercial vendors such as AccessData incorporating some support for multithreading and distributed execution into their forensics suites (e.g., FTK 2.0+), it appears that the advice offered in (Roussev, 2004) is being heeded.

GPUs will also play an important role in next-generation digital forensics and both the roles they can play and how best to harness their computational resources is less well-understood. GPUs require development of massively-threaded software under a restricted execution model and have complicated memory organizations. The necessary optimizations required for best performance will not be intuitive to inexperienced programmers. Therefore GPU programming is hard, but a number of factors make the effort worthwhile. First, for some classes of computation, offloading work to GPUs can result in speedups not attainable on traditional CPUs. Second, as researchers craft more sophisticated digital forensics techniques, the hunger for processing power to analyze digital evidence will increase. And finally, while GPUs may not be able to offer the speedup possible with large-scale cluster-based approaches, GPUs are a cost effective mechanism for high-performance digital forensics processing, particularly for those organizations for which purchase of traditional high-performance computing equipment (e.g., a Beowulf cluster) is cost-prohibitive. The remainder of this section discusses the basics of GPU programming, concentrating on NVIDIA's Compute Unified Device Architecture (CUDA) SDK. Section 4 then covers a number of case studies that introduce algorithmic and design issues and how carefully engineered cooperation between GPUs and CPUs can maximize performance.

3.1 GPU Programming Basics

3.1.1 Introduction

In this section we provide some background on GPU programming, based on NVIDIA's CUDA SDK, appropriate for use with NVIDIA's G80 and subsequent GPU architectures. This section is a summary of the information available in the CUDA SDK documentation at (NVIDIA, 2008); interested readers are referred to that document for more expansive coverage.

The primary hurdles that a programmer unfamiliar with GPU programming will face are a complicated memory hierarchy, a Single Instruction Multiple Thread (SIMT) execution model (a relaxed version of the more common Single Instruction Multiple Data (SIMD) model), and the need to stage data in the GPU for processing. The programmer will also face familiarization with some other issues to obtain maximum performance. First, unlike multithreaded programming on traditional CPUs, where thread creation is expensive and therefore reusable thread pools are often used, GPUs create and destroy large numbers of threads with very low overhead. Furthermore, GPUs perform floating point arithmetic very quickly and memory accesses (particularly to the large, un-cached pool of device memory called global memory) are relatively expensive. This is likely to contradict the experience of programmers who have focused exclusively on traditional computer systems. The remainder of this section discusses architectural details relevant to developers creating massively threaded tools for processing of digital evidence.

3.1.2 Execution Model

The latest NVIDIA GPUs are organized as a set of multiprocessors, each of which contains a set of scalar processors which operate on SIMT (Single Instruction Multiple Thread) programs. The G80 GPU (used on commodity graphics cards such as the 8800GTX) is typical, having 128 scalar processors organized into 16 multiprocessors with 8 scalar processors each. Each of these scalar processors executes at 1.35GHz, for a theoretical maximum compute capability of approximately 350 GFLOPs. A total of 768MB of RAM is available on the 8800GTX. Subsequent models, such as the GTX280, have more scalar processors, more RAM, and higher performance. Unlike earlier GPU designs, which had fixed numbers of special-purpose processors (e.g., vertex and fragment shaders), very limited support for arbitrary memory accesses (scatter/gather), and little or no support for integer data types, these scalar processors are general purpose. In addition to the set of scalar processors, each NVIDIA multiprocessor contains two units for computation of transcendental functions, a pool of shared memory, and a multi-threaded instruction unit. Barrier synchronization and thread scheduling are implemented in hardware.

To utilize the GPU, the host computer is responsible for copying data to the GPU, issuing units of work to be performed, and then copying results back from the GPU once the units of work have completed. Each unit of work is called a *kernel* and defines the computation to be performed by a large number of threads. The highest level organization of the threads is a *grid*, with areas of the grid organized into *blocks*. Each multiprocessor executes the threads making up a block in parallel. Since the number of threads in a block may exceed the number of scalar processors in a single multiprocessor, the threads of a block are further organized into *warps*. A warp is a fraction of a thread block, comprised of a set of threads that are currently executing on a particular multiprocessor. As thread blocks complete execution, new thread blocks are launched on available multiprocessors.

The SIMT model deserves additional attention. SIMT is related to the more familiar SIMD execution model, but differs in some important ways. In NVIDIA's SIMT model, each scalar processor executes a single thread and maintains its own register state and current instruction address. A multiprocessor executes sets of threads called warps in par-

allel, with each thread able to execute instructions and perform branches independently. Since a multiprocessor has only one instruction unit, a warp executes one common instruction at a time. If control flow diverges between threads because of a conditional branch (e.g., taking opposite branches in an IF/THEN/ELSE), then the multiprocessor executes the divergent instruction streams serially until control flow resynchronizes. A key difference between SIMD and SIMT is that for the sake of correctness, programmers need not worry about control flow divergence. Only efficiency is impacted when threads execute independent code paths, which offers greater programming flexibility.

3.1.3 Memory Architecture

NVIDIA GPUs provide a number of available memory spaces, through which threads can communicate with each other and with the host computer. These memory areas, with restrictions and associated costs, are:

- *A private set of 32-bit registers*, local to a particular thread and readable and writable only by that thread.
- *Shared memory* can be read and written by threads executing within a particular thread group. The shared memory space is divided into distinct, equal-sized banks which can be accessed simultaneously. This memory is on-chip and can be accessed by threads within a warp as quickly as accessing registers, assuming there are no bank conflicts. Requests to different banks can be serviced in one clock cycle. Requests to a single bank are serialized, resulting in reduced memory bandwidth.
- *Constant memory* is a global read-only memory space initialized by the host and readable by all threads in a kernel. Constant memory is cached and a read costs one memory read from device memory only on a cache miss, otherwise it costs one read from the constant cache. For all threads of a particular warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. The cost scales linearly with the number of different addresses read by all threads.
- *Texture memory* is a global, read-only memory space shared by all threads. Texture memory is cached and texture accesses cost one read from device memory only on texture cache misses. Texture memory is initialized by the host. Hardware texture units can apply various transformations at the point of texture memory access.
- *Global memory* is un-cached device memory, readable and writable by all threads in a kernel and by the host. Accesses to global memory are expensive, but programmers can use a set of guidelines discussed in the CUDA Programmer's Reference Manual to increase performance.

The most important issue governing these memory spaces is that access to the largest general purpose pool of memory, global memory, is flexible but expensive. Simply stag-

ing data that needs to be processed in global memory without attention to the other memory pools is easier, but will generally result in poor performance.

3.1.4 The Common Unified Device Architecture (CUDA) SDK

We now very briefly discuss the CUDA SDK, used for development of the tools described in the case studies in Section 4. CUDA is a compiler, set of development tools, and libraries for writing GPU-enabled applications. CUDA programs are written in C/C++, with CUDA-specific extensions, and are compiled using *nvcc* compiler (included in CUDA), under Microsoft Windows, Mac OS X, or Linux.

A CUDA program consists of a host component, executed on the CPU, and a GPU component, executed on the GPU. The host component issues bundles of work (kernels) to be performed by threads executing on the GPU. There are few restrictions on the host component. CUDA provides functions for managing the kernel invocations, memory management functions that allow allocating and initializing device memory, texture handling, and support for OpenGL and Direct3D. On the other hand, code that executes on the GPU has a number of constraints that are not imposed on host code. Some of these limitations are “absolute” and some simply reduce performance. In general, standard C library functions are not available in code executing on the GPU. CUDA does provide a limited set of functions for handling mathematical operations, vector processing, and texture and memory management. Recursion, static variables within functions, and functions with a variable number of arguments are not supported on the GPU component.

Finally, the *compute capability* of the GPU should be taken into account when using CUDA to develop GPU-enhanced tools. The compute capability of a GPU device is defined by a major revision number and a minor revision number. The initial series of NVIDIA GPUs that supported CUDA, including the G80, have compute capability 1.0, which provides basic features. Devices with compute capability 1.1 support all of the features of 1.0 devices plus the addition of atomic operations on 32 bit words in global memory. These operations include simple atomic arithmetic (e.g., *atomic_add()*), atomic swaps, min/max functions, etc. Compute capability 1.2 adds support for 64-bit atomic operations, support for “warp voting” functions, and doubles the number of available registers per multiprocessor. Warp voting functions allow the evaluation of a predicate across all currently executing threads on a multiprocessor and return a Boolean value. The latest compute capability as this chapter is written is 1.3, which provides support for double precision floating point numbers (devices with earlier computer capabilities support only single precision).

3.2 Maximizing Performance: Unifying Multicore and GPU Computation

In general, maximum performance will be gained when both multicore CPU and GPU resources are harnessed in unison. While CPUs are well-suited to general-purpose computation, work assigned to GPUs must have significant “arithmetic intensity” to overcome the cost of transferring data between the host and GPU memories. In the foreseeable future, host $\leftrightarrow$ GPU transfer rates will remain substantially below on-GPU memory bandwidth because of bus speed limitations. Compression of data being transferred to and from the GPU may be helpful, but this issue has not yet been properly investigated.

Below certain arithmetic intensity thresholds, it makes sense to simply use a multi-threaded implementation on a multi-core CPU and avoid the development effort associated with design of software components for GPUs.

When offloading computation onto GPUs makes sense, it generally also makes sense to simultaneously take advantage of resources offered by the multicore CPUs in the host and to overlap GPU and CPU-based computations. Since digital forensics processing generally involves significant I/O, a massively threaded approach to tool design will also involve overlapping I/O with CPU-bound operations, a paradigm that is rarely encountered in current-generation digital forensics tools. Newer NVIDIA GPUs such as the GTX280 and GTX260 have larger numbers of scalar processors, more on-board RAM, and provide much better support for overlapping I/O and computation. They can not only overlap CPU and GPU computation and computation across multiple GPUs, but also overlap computation with host $\leftrightarrow$ device memory transfers. This has significant benefits for I/O-heavy digital forensics tools. By allowing overlapping data transfer and computation, newer GPUs lower the arithmetic intensity threshold, making GPU computation applicable to a wider range of problems by hiding the significant costs of moving data between the host and GPUs.

Finally, we note that since GPU performance is significantly improved by reducing thread divergence (essentially, violations of SIMD execution semantics), there are interesting opportunities for load balancing computations between CPUs and GPUs. For example, depending on the particular string comparison algorithm in use, a GPU implementation might excel when it *doesn't* find matching strings, because matches cause thread divergence (as the location of the match is stored, etc.). Implementations might take advantage of situations like this by, e.g., offloading searches for infrequently encountered data (but important, nonetheless) to the GPUs and performing searches for the most commonly encountered data on host CPUs.

4 Detailed Case Studies

4.1 File Carving

4.1.1 Overview

This section briefly summarizes the work described in (Marziale, 2007) and includes additional work performed by the authors since the publication of that paper. Scalpel, a typical single-threaded file carving application, was modified to support both multi-threaded operation on multicore CPUs and to offload computation to one or more GPUs. Scalpel processes disk images in two passes, with the first pass reading the input in 10MB chunks and performing header/footer searches to locate potentially interesting data. Between the two passes, a schedule is constructed so the second pass can perform carving operations. Different threading models were developed for execution on multicore CPUs and on GPUs such as the NVIDIA G80. The Scalpel component that benefits most from parallelization is header/footer searches, involving a large number of binary string search operations. In this case study, a simple binary string search algorithm is used to locate header/footer strings on the GPU. More sophisticated string search algorithms are considered in Section 4.2.

As a base case, to measure speedup on multicore CPUs, Scalpel v1.60 was modified to support multithreaded header/footer searches, with a threading model based on the POSIX Pthreads library. Multithreaded Scalpel creates a thread pool with the number of threads equal to the number of different file types being recovered and these threads sleep while Scalpel fetches a 10MB block of data. The threads are then awakened to perform parallel header/footer searches on the 10MB block, before sleeping again. Each thread uses the Boyer-Moore binary string search algorithm (Boyer, 1977) to search for headers and footers. In this version, disk accesses are not overlapped with computation because the goal is to simply measure the performance of sequential, multithreaded, and GPU-enhanced file carving. Overlapping disk I/O with computation does offer a substantial performance gain, as we discuss later in this section. The multithreaded version of Scalpel for multicore CPUs is called Scalpel-MT in the performance study below.

For GPU-enhanced file carving, we consider two approaches to illustrate important points. These are named Scalpel-GPU1 and Scalpel-GPU2. Both Scalpel-GPU1 and Scalpel-GPU2 stage the carving rules for particular file types in the GPU's constant memory pool. Since constant memory is not cleared across kernel invocations, this operation is performed only once. Before each kernel invocation, a 10MB block of data is copied to global memory on the device. For both Scalpel-GPU1 and Scalpel-GPU2, a single host thread is used to invoke GPU operations. This thread blocks during kernel invocation; this decision is deliberate, so that we can more directly measure the performance of the GPU. Both Scalpel-GPU1 and Scalpel-GPU2 use a very simple "introduction to programming" sequential string search algorithm.

Scalpel-GPU1 invokes a kernel that creates 65,536 threads to process the 10MB block of data. These threads are created and torn down for each kernel invocation. Each GPU thread is responsible for searching approximately 160 bytes of the 10MB block, read directly from global memory. Results are then copied from the GPU to the host as a vector that encodes the locations of discovered headers and footers. Although the number of threads spawned is large when compared with typical multithreading approaches for code executing on traditional CPUs, increasing the number of threads substantially will significantly increase performance. This is counterintuitive to programmers used to conserving resources by creating thread pools and trained to fear substantial overhead for thread creation and teardown on traditional architectures. Thread creation overhead is non-existent on NVIDIA GPUs and an increase in the number of threads can help hide memory latencies, increasing performance.

Scalpel-GPU2 eliminates iteration over the 10MB buffer in the string search technique. Instead of spawning a relatively small number of threads, each searching a fixed portion of the 10MB block of data, Scalpel-GPU2 spawns one thread per byte—10 million threads—to process the input buffer. Each thread "stands in place", searching for all relevant headers and footers starting at a specified index in a small area of shared memory. Shared memory mirrors a portion of the 10MB buffer (in device memory) to further increase performance. This threading model is even more counterintuitive than Scalpel-GPU1 for programmers trained to use common multithreading techniques, because the overhead of managing so many threads would typically be prohibitive. But the NVIDIA GPUs excel at thread management and this modification substantially increases performance, as discussed in Section 4.1.2, below.

4.1.2 Experimental Results

To measure the performance of GPU-enhanced file carving, we ran carving operations on 20GB and 100GB disk images using a set of 30 carving rules. All carving operations used Scalpel's "preview" mode, which supports in-place carving (Richard, 2007). Experiments were conducted on a Dell XPS 710 workstation with a single Core2Duo processor (dual core) running at 2.6GHz. This machine had 4GB of RAM and a 500GB, 7200rpm SATA hard drive. An NVIDIA 8800GTX with 128 scalar processors and 768MB of RAM was used. All of the experiments on this computer were conducted under Linux, running a 32-bit 2.6-series SMP kernel and the ext3 filesystem.

The results for the 20GB disk image are presented in Table 1. The sequential version of Scalpel required 1,260 seconds to process the 20GB image. Scalpel-MT (the multi-threaded version running multiple header/footer processing threads on the host CPU) executed in 861 seconds. Offloading processing to the GPU, using Scalpel-GPU1, reduced execution time to 686 seconds. Finally, Scalpel-GPU2, in which a thread is spawned for each byte in the 10MB buffer, further reduced execution time to 446 seconds. Table 2 presents results for processing the 100GB disk image. Scalpel v1.60 required 7,105 seconds. Scalpel-MT reduced the execution time to 5,096 seconds. Scalpel-GPU1 required 4,192 seconds. Massively-threaded Scalpel-GPU2 had the best running time, 3,198 seconds.

For both disk images, simple multithreading results in substantially better performance than the sequential version of the file carver; this is not unexpected. But the versions of Scalpel using GPU-assisted carving have substantially better performance, and this occurs even though Scalpel-GPU1 and Scalpel-GPU2 are using a very simplistic string search technique, while the multithreaded Scalpel-MT is using Boyer-Moore (which is also used in the sequential version of Scalpel). To illustrate the importance of this point, we conducted another experiment, in which the Boyer-Moore string search algorithm in Scalpel-MT was replaced with the simple string search algorithm used in Scalpel-GPU1 and Scalpel-GPU2. For the 20GB disk, execution time for Scalpel-MT increased from 861 seconds to 3,544 seconds, or 4X slower. This doesn't mean, of course, that implementation of the Boyer-Moore algorithm in Scalpel-GPU1 and Scalpel-GPU2 would provide a corresponding improvement—in fact, it doesn't. But it does illustrate that even with the burden of host $\leftrightarrow$ GPU data transfer overhead and a simple search technique, the GPU performs remarkably well. There is clearly substantial room for improvement, as the rest of this section illustrates.

4.1.3 Overlapping computation with I/O

4.1.4 Overlapping Computation with I/O

In order to further push the limits of file carving performance, we have recently made some additional performance improvements to the Scalpel file carver, and commenced another round of experiments. The following demonstrates the performance improvements which can be gained from overlapping disk IO with CPU computation, as well as overlapping transfers to and from the GPU with computation on the GPU. To facilitate this, Scalpel was modified as follows: an asynchronous CPU thread is responsible for reading blocks of a disk image into a synchronized queue. This thread never waits for

CPU processing to occur, so the disk can be read in at the maximum speed for the device. Blocks are removed from the queue and sent to the GPU via the CUDA streams mechanism for processing.

In CUDA, the "streams" mechanism allows for concurrent computation and memory transfers on GPU devices. A stream is a logical sequence of actions, commonly of the form: transfer data to the device, perform some computation on the data, and transfer the results off the device. CUDA allows the user to create multiple streams, which act asynchronously, allowing overlapping computation and data transfer. Scheduling what will happen in what order is taken care of by the CUDA runtime so that no conflicts occur. The use of these mechanisms however adds another layer of complexity to an already complex system. CPU processing of a large dataset generally requires reading in the dataset a chunk at a time from secondary storage. In cases where a single logical unit of work requires more than 1-byte of input, the programmer must take into account that the first part of the input could be in the current chunk of the dataset, while the rest of this unit of input is in the next chunk of the dataset (which has yet to be read). These overlapping work units (border cases) are easy places to make mistakes, so they must be thoroughly checked. In CUDA programming for large datasets the difficulty is multiplied. First the dataset must be broken into chunks by the host application when reading from secondary storage as in the CPU case above. Then, potentially, this chunk must be further broken down for transfer to the limited amount of GPU RAM for processing. Once on the GPU, the smaller chunk is effectively processed in still smaller chunks by independently operating thread blocks. Overlap must be accounted for very carefully at each level. Since the GPU streams operate semi-independently, on different parts of the dataset, this is an additional opportunity for a border case to appear. In cases like these, the coding logic can become quite complex. The payoff though is demonstrated by the following results.

A set of experiments was performed on a Dell XPS 720 with a quad-core Core2 Extreme, 4GB of RAM, and 2 750GB 7200rpm SATA drives. This machine was equipped with an NVIDIA GTX260 GPU with 192 scalar processors and 896MB of RAM. Scalpel was executed in preview mode searching for 25 patterns on each of a 20GB disk image and a 100GB disk image (distinct from those in the previous experiments). Results are presented for 4 different configurations of Scalpel: single-threaded (Scalpel-1.60), multi-threaded (Scalpel-MT), multi-threaded with asynchronous disk reads (Scalpel-MTA), and GPU-enabled with asynchronous disk reads (Scalpel-GPU2A); these results appear in Tables 3 and 4. It is worth noting that the GPU-enabled version is operating at the transfer rate of the drive (~70MB/s). Scalpel-GPU2A is probably capable of higher performance if a faster storage medium was used.

4.2 Case Study: Better File Carving: Using Multi-Pattern String Matching Algorithms

Finding matching patterns is obviously an important problem in digital forensics, as well as other areas of security, like network intrusion detection. Examples include searching for text strings within documents and looking for files of specific types on a disk image (file carving). In this case study, we again focus on file carving, but consider the problem of searching for multiple file types in a more efficient manner. Since file carving applications must often search for multiple types of files, such as JPG and GIF files, or

MS Word and MS Excel files, it is logical to search for all these file types at the same time. For single pattern matching, there is a large body of work in which a single string is to be searched for in the text. More formally, the Multi-Pattern Matching problem searches a body of text for a set of strings. Let $P = \{p_1, p_2, \dots, p_k\}$ be a set of patterns, which are strings of characters from a fixed alphabet Σ . Let $T = t_1, t_2, \dots, t_N$ be a large sequence of bytes, again consisting of characters from Σ . The problem is to find all occurrences of all the patterns of P in T . One can trivially extend a single pattern matching algorithm to be a multiple pattern matching algorithm by applying the single pattern algorithm to the search text for each search pattern. For example, the Boyer-Moore string search algorithm is a particularly efficient string searching algorithm, and it has been the standard benchmark for the practical string searching field (Boyer, 1977).

A single pattern string matching algorithm becomes a multiple pattern string matching algorithm by applying a single pattern algorithm (such as Boyer-Moore) to the search text for each search pattern. Obviously, this does not scale well to larger sets of strings to be matched. Instead, multi-pattern string matching algorithms generally preprocess the set of input strings, and then search for all of them simultaneously over the body of text. The three most efficient multiple pattern matching algorithms are the Aho-Corasick algorithm (Aho, 1975), Commentz-Walter algorithm (Commentz-Walter, 1979), and Wu-Manber algorithm (Wu, 1994). The Aho-Corasick algorithm is a direct extension of the Knuth-Morris-Pratt algorithm (Knuth, 1977), which combines their algorithm with a finite state machine to model the patterns as a finite number of states with transitions between those states, and associated actions.

Commentz-Walter (Commentz-Walter, 1979) proposed an algorithm that combines the Boyer-Moore technique with the Aho-Corasick algorithm (Aho, 1975). It is often substantially faster than the Aho-Corasick algorithm for small pattern sets and long minimum pattern lengths. Hume designed a tool called *gre* based on this algorithm, and version 2.0 of *fgrep* by the GNU project is using it. The Commentz-Walter algorithm constructs a converse state machine from the patterns to be matched. Each pattern to be matched adds states to the machine, starting from the right side and moving to the first character of the pattern. In the searching stage, a match is attempted by scanning backward through the input string. At the point of a mismatch something is known about the input string (by the number of characters that were matched before the mismatch). This information is used as an index into a pre-computed shift table to determine the distance by which to shift before commencing the next match attempt.

The Wu-Manber algorithm was developed by Udi Manber and Sun Wu (Wu, 1994). It is a high performance multiple pattern matching algorithm based on Boyer-Moore. Three tables are built in the preprocessing stage, a SHIFT table, a HASH table, and a PREFIX table. The SHIFT table is similar, but not exactly the same, as the regular shift table in a Boyer-Moore type algorithm, because the SHIFT table determines the shift based on the last B characters rather than just one character. When the shift value is 0, the HASH and PREFIX tables are used to determine which pattern is a candidate for the match and to verify the match.

The Aho-Corasick algorithm is a linear-time algorithm, so it is optimal in the worst case, but as the regular string-searching algorithm by Boyer and Moore demonstrated, it may be possible to skip a large portion of the text while searching, leading to a faster than

linear algorithm in the average case. The performance of the Wu-Manber and Commentz-Walter algorithms is dependent on the minimum length of the patterns, because both of them are based on Boyer-Moore. In practice the Wu-Manber algorithm often has the best average performance.

Our goal was to design a multi-pattern matching algorithm for the GPU. Since we want to limit thread divergence, we adopt a simple algorithm that is similar to the Aho-Corasick algorithm in the sense that we maintain a Finite State Machine (FSM). However, we maintain a much simpler structure because the additional memory requirements for the type of FSM that is required by the Aho-Corasick algorithm will be too large to fit into the fast constant memory on the GPU, and furthermore, traversal of the FSM will lead to additional thread divergence, which we seek to avoid.

For this new algorithm we have created the pattern table, which is an array of patterns, and the preprocessing table for storing the index of each pattern of the pattern table. The preprocessing table is an array of structures with each structure having two integer variables. The array index for the preprocessing table is the value (0-255) of the first character of each pattern of the pattern table. The first integer variable of the structure stores the pattern table index and the second integer variable stores the array location for the next one with the same first character. For instance, if pattern 3 starts with an 'a' then array element 97 (the ASCII value of 'a') has 3 in the first field of the structure. If pattern 45 also starts with 'a', then, assuming information on pattern 45 is in array location 285, we store 285 in the second field.

Initially when we start with the first character of the 10MB text block, we need to search for an entry for that character in the preprocessing table. If we have such an entry then we can retrieve the pattern from the pattern table with this character as the first character. We can compare the other characters of the text with the rest of the characters in that pattern. If we don't have an entry in the preprocessing table then we move to the next character of the text. We repeat this process until the end of the block is reached.

4.2.1 Implementation and Testing

The following experiments were conducted using an NVIDIA 8600GTS graphics card. The 8600GTS card has 768MB of device RAM, and 32GB/sec memory bandwidth. It contains 4 multiprocessors, each of which contains 8 stream processors, for a total of 32 stream scalar processors.

Our new algorithm is massively threaded. In the host component, which is executing on the CPU, we have created the pattern table using 60 patterns and the preprocessing table for storing the index of each pattern of the pattern table. We copied the preprocessing table and the pattern table into the GPU's constant memory area. Since constant memory is not cleared across kernel invocations, this operation is performed only once. Before each kernel invocation, a 10 MB block of data is copied to global memory on the device. The host component issues the kernel, searching for headers and footers in the input buffer to the device component, which is executing on the GPU. In this massive threading implementation, approximately 10 million threads are spawned in the device component and each thread performs searches starting at a particular index in the input buffer. Each thread searches for relevant headers and footers starting at its index in a small area of shared memory, which mirrors a portion of the buffer in device memory.

Initially, when we start at the first character of the text in the input buffer, the first thread needs to search for an entry for that character in the preprocessing table. All the threads search the input buffer similarly in parallel. If we have such an entry then we can get the pattern from the pattern table with this character as the first character. We can compare the other characters from the shared memory with the rest of the characters in that pattern. We repeat the same process until the end of the input buffer. The indices of the matching patterns are written to a data structure that is subsequently retrieved by the host component once kernel execution is completed.

4.2.2 Results

For measuring the performance of GPU enhanced file carving, we ran carving operations on 2GB, 5GB, and 20GB disk images for different number of carving rules such as 10, 20, 30, 40, 50, and 60. For all carving operations we used Scalpel's "preview" mode, which supports in-place carving (Richard, 2007). On the GPU we have tested the following Multiple Pattern Matching algorithms: Simple String Search, the Boyer-Moore algorithm, the Wu-Manber algorithm, and our new algorithm. We did not include the Aho-Corasick and Commentz-Walter algorithms because Wu-Manber showed better performance for the initial tests with our pattern table. Simple String Search is implemented on the GPU as a simple sequential string searching algorithm and uses massive GPU threading [9]. We spawned one thread per byte for the input buffer. Each thread stands in place, searching for all relevant headers and footers starting at its location. For the Boyer-Moore algorithm and for Wu-Manber we did not use the massive GPU threading; we used a different multi-threading method. Both of these algorithms also copy a 10MB block of data to global memory on the device. For the Boyer-Moore algorithm, the host then invokes a kernel that creates 65,536 threads to process the block. Each GPU thread is responsible for searching approximately 160 bytes of the 10MB block, read directly from global memory. For the Wu-Manber algorithm we have created 8192 threads to process the block, each thread is responsible for searching 1,280 bytes. The sections of the buffer processed by individual threads are chosen to overlap by a number of bytes equal to the longest search string to accommodate headers and footers that lie across section boundaries. These two algorithms used larger blocks so the benefits of skipping characters could be tested. However, Boyer-Moore and Wu-Manber algorithms are more complicated, require more memory and use skipping of some characters which causes thread divergence by using massive GPU threading, then thread execution is serialized.

We have conducted experiments using different image files with sizes 2GB, 5GB, 20GB, using the above pattern table for Boyer-Moore, Wu-Manber, Simple String Search and our new algorithm. We created 6 different pattern files by copying the first 10, 20, 30, 40, 50, 60 patterns, respectively. For the Wu-Manber algorithm because of a bug in the open-source Agrep implementation when constructing the hash tables we were not able to conduct experiments with more than 45 patterns. For this reason we have not given the results for Wu-Manber for more than 40 patterns. The mean and standard deviation for the 5 2GB size image files are given below. Since the other image file sizes showed similar performance, only the 2GB results are shown in Table 5.

In Table 5, for Boyer-Moore and for Wu-Manber, the standard deviations are 10% or more when compared to the mean. This is caused by the unpredictable effects of thread

divergence. The performance of Boyer-Moore algorithm and Wu-Manber algorithm depend on the shortest pattern length. Since file headers and footers are rather short, there is little opportunity to skip characters. However, neither algorithm is competitive with the new algorithm. Although Boyer-Moore is faster than simple string search, the small number of characters that can be skipped and the resulting thread divergence limit the performance improvement. The Wu-Manber algorithm has additional overhead because the relatively large data structures do not fit in constant memory, so memory accesses are more expensive. It can be seen that the time taken for the file carving for any image file using our new algorithm is much less than the other algorithms. Although omitted because of space constraints, our experiments show that the Wu-Manber algorithm outperforms the new algorithm on a CPU. This is strong evidence that algorithms should be optimized for the GPU and the best algorithm on a CPU may not offer the best performance on a GPU.

4.3 GPU-enabled Network Intrusion Detection

In this section we briefly describe the GPU-enabled version of snort, named gnort, which was recently presented in detail in (Vasiliadis, 2008). We chose this application because it illustrates similarities between digital forensics and network security approaches and the potential for these disciplines to converge, if sufficient computational resources are available. This work also considers some interesting design alternatives for GPU-enabled NIDS. The authors concentrate on the performance of a GPU-enabled version of snort compared to a single-threaded version. A performance study for multi-threading snort on multicore processors was not done.

4.3.1 Overview

NIDS is similar to file carving in that large amounts of binary data must be matched against sets of patterns. Performance is even more critical, however, because processing must typically be done in real time, at line speed. gnort transfers packets for processing on the GPU by copying batches of packets to the GPU, rather than moving individual packets. After a short delay, packets are copied even if the corresponding batch to be transferred isn't yet full. These packet transfers are overlapped with GPU processing, so that while one batch of packets is being analyzed, another batch is transferred. gnort uses the Aho-Corasick algorithm (mentioned in the previous section) for pattern matching.

Two different approaches for processing packets are considered. The first splits each packet into 32 fixed-size fragments and each thread in a 32-thread warp searches one of the fragments in parallel. The fragments are cached in local memory to increase performance. In the second approach, each thread is assigned an individual packet. This approach has the disadvantage of assigning unequal amounts of work to each thread, since packets typically vary in size.

4.3.2 Performance Evaluation

The authors of (Vasiliadis, 2008) explore a number of performance considerations. We focus on the performance study which considered real network traffic. All of their experiments were conducted using an NVIDIA GeForce 8600GT card, which contains 32 stream processors organized in 4 multiprocessors, operating at 1.2GHz. The card has a

total of 512 MB of device memory. The CPU they used was a 3.40 GHz Intel Pentium 4 with 2GB of RAM. A second PC is used to replay real network traffic for processing using the *tcprelay* application. *gnort* used 5467 rules with 7878 patterns. The performance of the GPU-enabled version of *snort* was substantially higher than that of the vanilla *snort*. *gnort* is able to process network data at a sustained rate of 600Mb/sec, approximately 2X faster than the 300Mb/sec attainable by a single CPU.

5 Future Trends

The work presented in this chapter reflects only the beginning in terms of utilizing massive parallelism that is increasingly available on commodity hardware. Having approached fundamental limits and diminishing returns, hardware manufacturers have refocused their attention from the ‘gigahertz wars’ to the ‘parallelism wars’. In other words, instead of using manufacturing advances to push execution rates, they are working on increasing the level of thread parallelism supported by the processors.

Looking ahead, it is a safe bet that for the foreseeable future, hardware-supported parallelism will continue to increase at a fast rate. In the next few years we can expect GPU architectures to advance at a faster rate relative to CPUs, due to the novelty of the technology. We may also see the rise of more tightly integrated CPU-GPU hybrid architectures, similar to IBM’s Cell Broadband Engine, as AMD seeks to integrate its CPU and GPU products, and Intel seeks to enter the high-performance graphics arena.

The fast introduction of large-scale parallelism finds software developers largely unprepared—several generations of programmers and software development tools have been steeped in algorithms and performance optimizations for sequential programming. Parallelism was the domain of supercomputing and a small minority of software experts. Currently, there is a widespread consensus among researchers and developers that there is dearth of mature software technologies that will help the average programmer transition to parallel coding techniques. This is particularly true for performance-sensitive applications, such as digital forensics. The current generation of GPU software development tools are in their infancy and require a lot of patience and expertise to achieve the kind of performance results described earlier. We can expect these to mature quickly and to gradually start providing higher-level primitives. In particular, more and more of the performance optimization will be handled automatically. At present, the developer has to worry about too many details—synchronous thread execution (non-divergence), overlapping host-device memory transfers with computation, managing the GPU memory hierarchy and memory access costs, etc. We expect that future libraries will provide optimized versions of common algorithms, such as sorting and searching, as well as data structures optimized for parallel computations.

It is unlikely that performance-conscious tools will ever be completely decoupled from the underlying hardware; however, programmers can expect significant help in the form of automated source code analysis (to identify parallelism), optimized libraries, profiling, and so on. Intel’s C/C++ compiler suite for multi-core CPUs, which is significantly more mature than GPU tools, provides a peek into what is needed to make massively-parallel programming more accessible.

Most importantly, it is application developers who need to fundamentally change their way of thinking from sequential to parallel. There will be a transitional period of more than a few years until parallel algorithmic approaches become the default. As some of the presented work here demonstrates, algorithms optimized for sequential execution are not always good candidates for parallelization, so algorithmic problems considered ‘solved’ for many years will have to be re-examined. Significant legacy research on parallelism will speed up this process; however, it is likely that many of these solutions need to be adjusted for the realities of modern-day architectures.

Looking still further ahead, we consider it likely that standard data formats will have to change to facilitate parallel data processing. For example, many compressed data formats, such as JPEG, ZIP, and PNG, are fundamentally designed for sequential processing. We believe that fundamental forensic functions, such as hashing, will also change to accommodate parallelism. For example, at least one of the competing designs for the next generation crypto hash functions—Skein—is designed explicitly with parallel implementation in mind. Taken together with the exponential growth in target size, all of these developments mean that as soon as five years from now, a large part of the existing forensic code base will be obsolete and will have to be rewritten from scratch to accommodate parallelism.

6 Conclusion

The exponential growth in the number and size of forensics targets, particularly disk storage, necessitates the use of more powerful methods for processing information. The recent trend in processor design toward multicore processors, along with the increasing affordability of high-performance GPUs, makes massively multithreaded forensics tools a promising approach to these demands. As multicore processors and powerful GPUs become commonplace, programmers must be prepared to exploit these resources. Hence, the reach of multithreaded tools will continue to expand.

In this chapter, we have explained the potential of multithreaded code for meeting the demands of forensic investigators and presented case studies that illustrate how the effective use of multithreading can improve the throughput of digital forensics and network security tools. Scaling the performance as the number of cores per processor increases is a significant research challenge for keeping up with the growth in forensic targets. Additional research will also be required to integrate multicore CPUs and high-performance GPUs in the most efficient manner.

7 References

Aho, A. V., & Corasick, M. J. (1975). Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6), 333-340.

ATI (2008). GPU Technology for Accelerated Computing. Retrieved 11/18/2008 from <http://ati.amd.com/technology/streamcomputing>.

Boyer, R. S., & Moore, J. S. (1977). A Fast String Searching Algorithm. *Communications of the ACM*, 20(10), 762-772.

B. Commentz-Walter, B. (1979). A String Matching Algorithm Fast on the Average. Proc. 6th International Colloquium on Automata, Languages, and Programming, 118-132, Graz, Austria.

Brook (2008). "BrookGPU. Retrieved 11/18/2008 from <http://graphics.stanford.edu/projects/brookgpu>.

Federal Bureau of Investigation (2007). Regional Computer Forensics Laboratory Program annual report FY2007. Retrieved 11/18/2008 from <http://www.rcfl.gov>.

Garfinkel, S. (2006). Forensics Feature Extraction and Cross-Drive Analysis. Proceedings of the 2006 Digital Forensics Research Workshop (DFRWS 2006), Lafayette, Indiana.

Jacob N. & Brodley C. (2006). Offloading IDS computation to the GPU. Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC2006), Miami Beach, FL.

Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast Pattern Matching in Strings. SIAM Journal on Computing, 6(2), 323-350.

Manavski S. (2007). CUDA-Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. Proceedings of the 2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007), Dubai, United Arab Emirates.

Marziale, L., Richard III, G. G., & Roussev, V. (2007). Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools. Proceedings of the 7th Annual Digital Forensics Research Workshop (DFRWS 2007), Boston, MA.

NVIDIA (2008). Common Unified Device Architecture (CUDA). Retrieved 11/18/2008 from <http://www.nvidia.com/cuda>.

Patterson D. (2004). Latency Lags Bandwidth. Communications of the ACM, Vol. 47(10), 2004.

Richard III, G. G., & Roussev, V. (2005). Scalpel: A Frugal, High-Performance File Carver. Proceedings of the 2005 Digital Forensics Research Workshop (DFRWS 2005), New Orleans, LA.

Richard III, G. G., Roussev, V., & Marziale, L. (2007). In-place File Carving. Research Advances in Digital Forensics III, 217-230, Springer.

Roberts P. (2005). DOD seized 60TB in Search for Iraq Battle Plan Leak. Computer-world (Australia), Retrieved 11/18/2008 from <http://www.computerworld.com.au/index.php/id;266473746>.

Roussev, V. & Richard III, G. G. (2004). Breaking the Performance Wall: The Case for Distributed Digital Forensics. Proceedings of the 2004 Digital Forensics Research Workshop (DFRWS 2004), Baltimore, MD.

Vasiliadis G., Antonatos S., Polychronakis M., Markatos E. & Ioannidis S. (2008). Gnort: High Performance Network Intrusion Detection Using Graphics Processors. Proceedings of the 11th International Symposium On Recent Advances In Intrusion Detection (RAID). Boston, MA.

Wu, S., & Manber, U. (1994). A Fast Algorithm for Multi-Pattern Searching. Technical Report TR 94-17, Department of Computer Science, University of Arizona, Tucson, AZ.

8 Key Terms and Their Definitions

Digital forensics: Digital forensics is the application of forensic techniques to the legal investigation of computers and other digital devices.

Multicore CPU: A multicore CPU is a single-chip processor that contains multiple processing elements.

Graphics Processing Unit (GPU): A GPU is a computing device that was traditionally designed specifically to render computer graphics. Modern GPU designs more readily support general computations.

File carving: File carving is the process of extracting deleted files or file fragments from a disk image without reliance on filesystem metadata.

Beowulf cluster: A Beowulf cluster is a parallel computer built from commodity PC hardware.

String matching algorithm: A string matching algorithm is a procedure for finding all occurrences of a string in a block of text.

Multi-pattern string matching: A multi-pattern string matching algorithm is a procedure for finding all occurrences of any of a set of text strings in a block of text.

Single Instruction Multiple Data (SIMD): SIMD is an approach to parallel computing where multiple processors execute the same instruction stream but on different data items.

Single Instruction Multiple Thread (SMT): SMT is an approach to parallel computing where multiple threads execute the same computations on different data items.

Figures and tables referenced in the chapter:

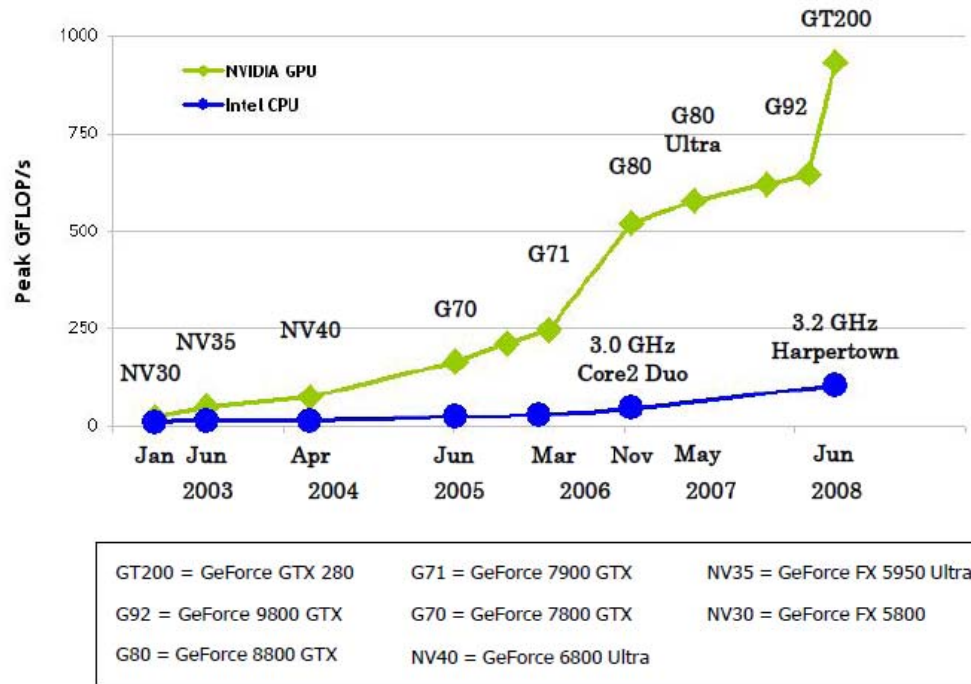


Figure 1. Peak computational capability (in GFLOPS) for Intel CPUs and NVIDIA GPUs.

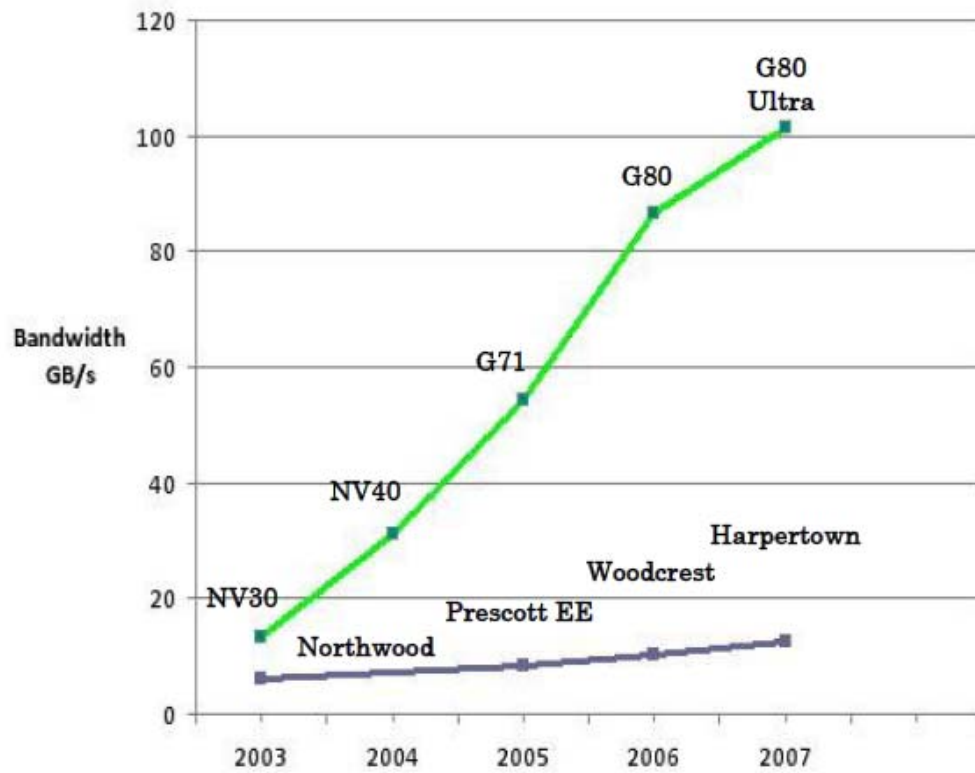


Figure 2. Peak memory bandwidth (in GB/sec) for Intel CPUs and NVIDIA GPUs.

Scalpel 1.60	1260 seconds
Scalpel-MT	861 seconds
Scalpel-GPU1	686 seconds
Scalpel-GPU2	446 seconds

Table 1. Results for file carving on 20GB disk image, 30 file types. Each result is the average of multiple runs.

Scalpel 1.60	7105 seconds
Scalpel-MT	5096 seconds
Scalpel-GPU1	4192 seconds
Scalpel-GPU2	3198 seconds

Table 2. Results for file carving on 100GB disk image, 30 file types. Each result is the average of multiple runs.

Scalpel 1.60	656 seconds
Scalpel-MT	393 seconds
Scalpel-MTA	296 seconds
Scalpel-GPU2A	282 seconds

Table 3. Results for file carving on 20GB disk image, 25 file types. Each result is the average of multiple runs.

Scalpel 1.60	3299 seconds
Scalpel-MT	1974 seconds
Scalpel-MTA	1529 seconds
Scalpel-GPU2A	1432 seconds

Table 4. Results for file carving on 100GB disk image, 25 file types. Each result is the average of multiple runs.

Number of Patterns	10		20		30		40		50		60	
	mean	std	mean	std	mean	std	mean	std	mean	std	mean	Std
Boyer-Moore	13.96	1.33	17.52	1.42	22.9	1.32	35.6	3.46	36.52	3.17	43.46	3.08
Wu-Manber	32.5	5.29	51.48	26.6	60.92	37.1	72.4	49.9				
Simple String Search	15.52	0.178	21.6	0.244	28.3	0.339	34.6	0.07	40.66	0.44	51.1	1.0
New Algorithm	10.16	0.14	10.48	0.07	11.34	0.21	11.62	0.28	11.84	0.28	14.26	0.56

Table 5. Mean and standard deviation for execution on 5 2GB image files.