available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Massive threading: Using GPUs to increase the performance of digital forensics tools[☆]

Lodovico Marziale, Golden G. Richard III, Vassil Roussev*

Department of Computer Science, University of New Orleans, New Orleans, LA 70148, United States

Keywords:
GPU
Parallel programming
Digital forensics
Distributed digital forensics
File carving
SIMD

ABSTRACT

The current generation of Graphics Processing Units (GPUs) contains a large number of general purpose processors, in sharp contrast to previous generation designs, where special-purpose hardware units (such as texture and vertex shaders) were commonly used. This fact, combined with the prevalence of multicore general purpose CPUs in modern workstations, suggests that performance-critical software such as digital forensics tools be ‘‘massively’’ threaded to take advantage of all available computational resources.

Several trends in digital forensics make the availability of more processing power very important. These trends include a large increase in the average size (measured in bytes) of forensic targets, an increase in the number of digital forensics cases, and the development of ‘‘next-generation’’ tools that require more computational resources. This paper presents the results of a number of experiments that evaluate the effectiveness of offloading processing common to digital forensics tools to a GPU, using ‘‘massive’’ numbers of threads to parallelize the computation. These results are compared to speedups obtainable by simple threading schemes appropriate for multicore CPUs. Our results indicate that in many cases, the use of GPUs can substantially increase the performance of digital forensics tools.

1. Introduction

This paper investigates the role that Graphics Processing Units (GPUs) can play in enhancing the performance of digital forensics tools. Traditionally, GPUs have been both difficult to program and 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. A new class of GPUs, such as the NVIDIA G80, have large numbers of general purpose stream processors that excel at executing massively threaded algorithms. Considering their speed, GPUs are relatively cheap and modern architectures allow adding several GPUs to a single computer. The peak performance of the NVIDIA line of GPUs, compared to the peak performance of the Intel line of general purpose CPUs, is shown in Fig. 1.

The goals of the experiments described in this paper included measuring the effectiveness of offloading processing common to digital forensics tools to a GPU and, even more importantly, comparing the resulting performance improvement with that attainable by using simple threading techniques on multicore CPUs. GPU programming, even on modern GPUs, is substantially more difficult than developing multithreaded

[☆] This work was supported in part by the National Science Foundation under Grant No. CNS-0627226.
[*] *Corresponding author.*
E-mail addresses: vico@cs.uno.edu (L. Marziale), golden@cs.uno.edu (G.G. Richard III), vassil@cs.uno.edu (V. Roussev).

1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.014

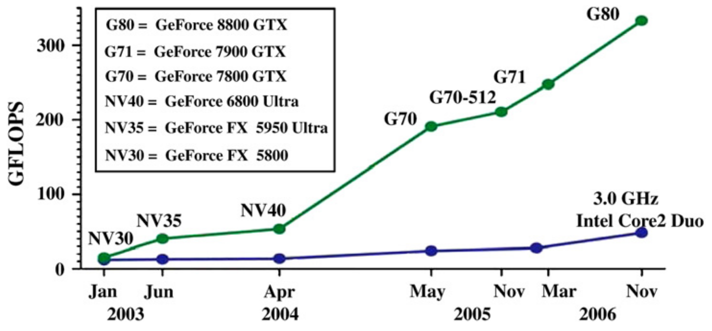

Fig. 1 – Estimated computational power, in gigaflops, of representative GPUS. Source: NVIDIA CUDA 0.8 SDK.

applications suitable for execution on multicore CPUs such as the Intel Core2Duo or AMD Opteron. The question is whether this additional effort is worth it. Our results suggest that the answer is yes.

Several trends in digital forensics make the availability of more processing power to support investigations an urgent need. The first is a vast increase in the average size of forensic targets encountered by investigators, which is directly attributable to the availability of cheap storage devices. This results in long turnaround times for critical cases and ultimately causes large case backlogs. Another trend is the increasing sophistication of digital forensics tools, fueled by growing interest in digital forensics as a research area and by a realization that feeds back into the first trend. This realization is that "manual" investigative methods, such as searching for child pornography by viewing thumbnails or listening to every audio file on a drive, are completely impractical when terabytes of data must be processed. Finally, the number of digital forensics cases is rising for a number of reasons, including better awareness of digital forensics techniques in law enforcement and in the private sector.

Currently, most digital forensics tools run on a single workstation. For very large cases, only distributed computing (e.g., using a system like DELV, Roussev and Richard, 2004) will offer enough processing power. But the performance of tools running on individual workstations can be increased substantially, through a number of means. One is very careful attention to the design of digital forensics tools, to minimize disk accesses and data copying. Unlike commodity applications like word processors, where the ever-increasing computational power of modern CPUs can hide sloppy programming or the excessive use of expensive abstractions, digital forensics software must execute as quickly (and accurately) as possible. In some cases, lives, economic prosperity, or freedom may hang in the balance.

Design must also take into account the trend to use lower clock speeds and multiple compute cores in modern CPUs. Many current generation tools are single-threaded and without modification will be unable to take advantage of modern hardware, including multicore processors. We argue in this paper that new, multithreaded designs should also consider the role that GPUs can play. GPUs excel at single instruction, multiple data (SIMD) computations and examples of these kinds of calculations definitely appear in the tools we develop in the digital forensics research community.

2. Related work

2.1. Distributed digital forensics

In some cases, only tens, hundreds, or thousands of general purpose processors, coupled with large amounts of RAM, will suffice to solve a large case within a reasonable timeframe. A distributed solution, such as a digital forensics framework running on a compute cluster (Roussev and Richard, 2004), may be necessary. Such systems can address both I/O and processing constraints, using aggressive data caching techniques and performing investigative operations in parallel. The research described in this paper is complimentary to that approach, since GPUs may be able to speed up some cases sufficiently so they can run on a single workstation, freeing cluster resources to process larger cases. The techniques can also be used to build faster clusters, by augmenting each node in a cluster with one or more GPUs. This approach was used with older GPUs in Fan et al. (2004).

2.2. GPUs in computer security software

A recent paper by Jacob and Brodley (2006) describes PixelSnort, a port of the popular open-source intrusion detection system (IDS), Snort. Their system offloads IDS packet processing (specifically, comparison with Snort rules) from the host's CPU to an NVIDIA 6800GT. Since the 6800GT does not present a programming model with general purpose processors (unlike the G80 used in our research), GPU programming is complicated. The 6800GT provides vertex and fragment processors and programs can be written to control either processor. Jacob and Brodley convert Snort rules to textures and then use the fragment processor to match network packets with rules. This involved writing a fragment shader (in Cg) that performs string searches. When a packet matches a rule, the fragment shader writes to the framebuffer, otherwise the packet is simply discarded. They detect matches using a graphics technique called occlusion-query, which is supported by OpenGL. PixelSnort offers modest performance gains, probably limited by the complicated software architecture dictated by the 6800GT's lack of direct support for general computations. The programming situation for GPUs is improving, as we point out in subsequent sections of this paper.

2.3. GPUs for "General Purpose" computing

The tremendous increases in power in today's GPUs have led to a need for ways to utilize them for non-graphics applications. For example, the NVIDIA 8800GTX is capable of a theoretical maximum of 350 GFLOPS at a cost of $570 (April 2007). Compare this to a 3.0 GHz Intel Core2Duo, which is capable of around 40 GFLOPS, at a cost of about $266 (April 2007). This gives $.95/GFLOP for the 8800GTX and $6.65/GFLOP for the Core2Duo. Memory bandwidth is also much higher on the GPU: 86.4 GB/s vs. 6 GB/s. Clearly there is reason to want to exploit the tremendous power of the GPU.

On the downside, the prevailing GPU architecture and other GPU implementation details lead directly to several difficulties when doing general purpose programming. Floating point numbers are generally non-IEEE compliant. Until recently, there was no support for integer arithmetic. There were no random memory writes. The massively parallel nature of the GPU incurs added cost at each branching operation. As threads diverge, the GPU has to begin executing them serially instead of in parallel. To combat these constraints, algorithms have to be re-engineered to exploit parallelism. In addition, the memory hierarchy can be complex, with several types of memory of differing access granularity, speed, and size requiring strict attention to memory allocation details. And, also until recently, coding had to be done through a graphics API which was not particularly friendly to non-graphics programming. Last, since moving data into and out of the GPU is an added cost, algorithms which do not exhibit a certain level of "arithmetic intensity" and, therefore take advantage of the power of the GPU, may not overcome the added costs.

Most coding done for GPUs is through one of a handful of APIs. OpenGL is an open 2D and 3D API developed by Silicon Graphics in 1992. It is a cross-platform, cross-language set of around 250 functions for building complex graphics from a set of primitives. Competing with OpenGL is Direct3D. This proprietary offering from Microsoft, part of the DirectX package, is the basis for the graphics API on the Xbox and Xbox 360 console systems. Close to the Metal is ATI's API exposing the hardware and instruction set of its newer stream processors. Sitting above the graphics APIs are graphics and general purpose languages. Developed by NVIDIA, C for Graphics, or Cg, is a high-level graphics language based on C. It shares syntax with C and adds some new features that make it more suitable for GPU programming. In the general purpose arena is BrookGPU from Stanford University's Graphics Group. It consists of a compiler and runtime for their general purpose stream programming language, called Brook.

Several significant GPGPU projects have been implemented recently. At Stony Brook's Center For Visual Computing, a parallel flow simulation using the Lattice-Boltzmann Model (LBM) was implemented on a cluster with 32 nodes with dual Xeon 2.4 GHz processors (Fan et al., 2004). Each node was equipped with an NVIDIA GeForce FX 5800 Ultra, resulting in a 4.6 times speedup over their CPU cluster implementation. Stanford University's Folding@Home project has produced a (beta) GPU client for the ATI X series of GPUs. It provides 20–40 times faster processing over general purpose CPUs in many of the calculations needed to simulate the folding of proteins. They have also created a client for the PS3 cell processor, which is about 40 times faster than a regular CPU.

The PeakStream Application Platform (PeakStream, 2006) from PeakStream was used to perform Monte Carlo simulations for pricing financial instruments. The GPU implementation provides a 16 times speedup vs. dual 3.6 GHz Xeon processors. At the University of North Carolina at Chapel Hill, algorithms have been developed for performing fast computation of several common database operations on GPUs (Govindaraju et al., 2004). Database operations were broken down into three basic types: conjunctive selections, aggregations, and semi-linear query. They achieved as high as an order of magnitude performance gain for certain query types. ATI has recently presented a new virtual machine abstraction of a GPU, the Data Parallel Virtual Machine (Peercy et al., 2006). It exposes the hardware as a data parallel processor array and a memory controller fed by a simple command processor in a platform independent way. This allows a developer to fully exploit the hardware without being locked into a graphics-centric framework. Interestingly, multiple virtual machines can operate on one GPU or a single virtual machine can operate across multiple GPUs.

We discuss NVIDIA's CUDA architecture, used for our work, in a subsequent section.

2.4. File carving

File carvers (e.g., DFRWS, 2006; Foremost; Richard and Roussev, 2005) read sets of rules, traditionally, databases of header and footer definitions, and search one or more target disk images for streams of bytes which potentially represent recoverable files (or file fragments).

File carving is a very important data recovery technique because files can be retrieved in the absence of filesystem metadata, e.g., after this metadata is destroyed by a format operation. While a filesystem's metadata is fragile, file data is much more resilient.

Treating file carving as a "typical" digital forensics technique makes sense because many common issues arise. First, disk activity must be minimized, since file carvers typically must make multiple passes over a disk image. Second, they must perform very efficient binary string searches, because a number of patterns must be matched against a large amount of binary data. Finally, the sophistication of file carving is increasing, with the development of techniques for 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.

3. Overview: NVIDIA G80 and CUDA

In this section we briefly describe the architecture of the NVIDIA G80 GPU, the 8800GTX graphics card used in our experiments, and the Compute Unified Device Architecture (CUDA) SDK, which is used to program the G80 GPU. This section is a summary of the information available in the CUDA SDK documentation at NVIDIA.

The G80 contains a set of multiprocessors, each of which contains a set of stream processors which operate on SIMD (Single Instruction Multiple Data) programs. A high-level design of the G80 is depicted in Fig. 2. 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, the stream processors in the G80 are general purpose. Despite this, care must still be taken to write code which executes quickly on the GPU. One relevant architectural constraint is that stream processors within a multiprocessor share an instruction unit; if control flow "diverges", then thread execution is serialized. Another constraint is that access to device memory, the largest general purpose pool of memory on the device, is uncached and much slower than access to the other memory pools.

A unit of work issued by the host computer to the GPU is called a *kernel* and defines the computation to be performed by a large number of threads, organized in thread *blocks*. Each multiprocessor executes one or more thread blocks, with each group organized into *warps*. A warp is a fraction of a thread group, comprised threads that are currently executing on a particular multiprocessor.

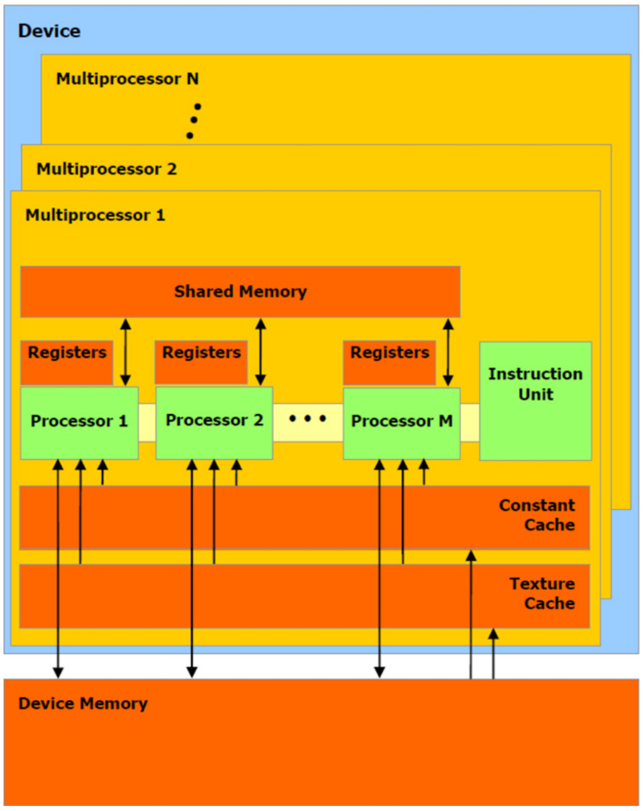

Fig. 2 – G80 architecture. On the 8800GTX, there are 16 multiprocessors, each containing eight stream processors, for a total of 128 processors. The stream processors within a multiprocessor share an instruction unit, so maximum parallelism is obtained only when the stream processors execute the same instruction stream (potentially on different data). Source: NVIDIA CUDA 0.8 SDK.

Fig. 3 illustrates the organization of executing threads on the G80 and their relationships with 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:

- *Private registers* are local to a particular thread and readable and writeable only by that thread.
- *Constant memory* is 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.
- *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.
- *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.

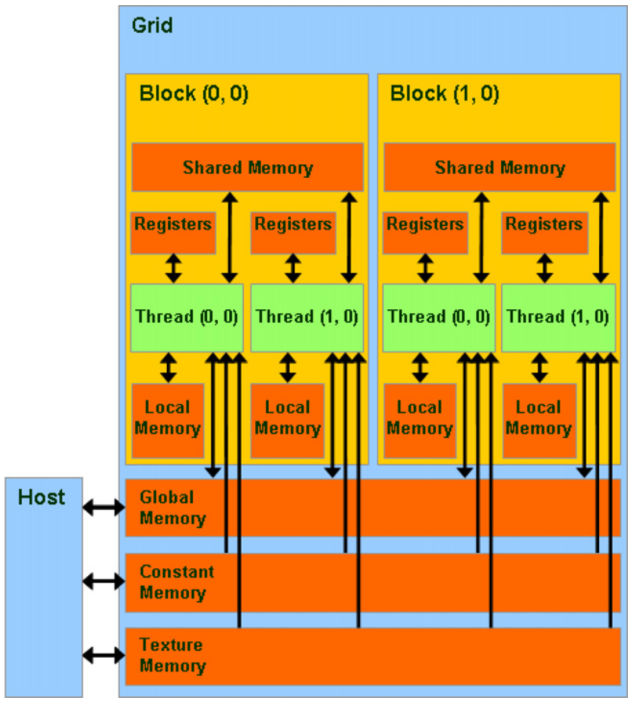

Fig. 3 – Organization of a grid of threads in CUDA and the relationship between threads and available memory spaces. Source: NVIDIA CUDA 0.8 SDK.

- Finally, *global memory* is uncached device memory, readable and writeable by all threads in a kernel and by the host. Accesses to global memory are expensive, requiring 200 or more cycles of memory latency.

The 8800GTX card we used has a single G80 GPU, 768 MB of device RAM, and 128 stream processors, organized into 16 multiprocessors. Each stream processor executes at 1.35 GHz. The raw (theoretical) compute power of the 8800GTX is approximately 350 GFLOPS. Some specific limits of the 8800GTX relevant to our work are:

- A maximum of 512 threads per thread block is allowed.
- 16 KB of shared memory is available per multiprocessor, organized into 1 K banks.
- A total of 64 K of constant memory is available, with a cache size of 8 K per multiprocessor.
- Thread warp size on 8800GTX is 32 threads.

We now very briefly discuss the CUDA SDK, used to conduct our GPU experiments. CUDA programs are written in C/C++, with CUDA-specific extensions, and are compiled using the *nvcc* compiler, under either Microsoft Windows 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, other than kernel invocations blocking the calling host thread. CUDA provides functions for managing the GPU, memory management functions which allow allocating and initializing device memory, texture handling, and support for OpenGL and Direct3D.

The code executing 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. The most important performance constraints are maximizing use of shared memory, limiting access to global memory as much as possible, and keeping threads within a warp in "lockstep", since violations of the SIMD execution model result in thread serialization.

4. GPU-enhanced digital forensics tools: case study

4.1. *Background*

To test the ability of current generation GPUs to speed digital forensics operations, we modified the file carver Scalpel (Richard and Roussev, 2005) to support multithreaded operation. Different threading models were developed for execution on multicore CPUs and on GPUs such as the NVIDIA G80. The component of Scalpel most amenable to parallelization is header/footer searches, which involve a large number of binary string search operations. Since searching for binary strings is a building block of many digital forensics techniques, this is a reasonable place to start.

As we discussed in Section 1, comparing the effectiveness of using a GPU to increase the performance of a digital forensics tool vs. simply creating a multithreaded version of the tool for execution on one or more multicore CPUs is important. One reason is that the GPU may be able to offer substantially more performance at a much lower price than adding additional CPUs to a workstation. But this performance comes at a price, namely, increased programming effort. To address this issue we conducted our experiments on both a relatively expensive workstation, with two multicore processors and a GPU, as well as a more modest workstation, with a single dual core processor and the same GPU.

For the following experiments, we modified Scalpel v1.60 to support multhreading on both multicore CPUs (using the POSIX Threads Pthreads library) and on the G80 GPU (using CUDA). Scalpel processes disk images in two passes, with the first pass reading the input in 10 MB chunks and conducting header/footer searches. Between the two passes, a schedule is constructed so the second pass can perform all of the carving operations (or, for in-place carving, Richard et al., 2007, construct only a set of file fragment offsets and lengths).

We parallelized the header/footer processing in the first phase as follows. For multicore machines, Scalpel was modified to spawn a thread for each file carving rule. These threads form a pool that sleeps while Scalpel fetches a 10 MB block of data, then wakes to perform header/footer searches on the block, before sleeping again. In our prototype, we do not currently hide disk access times by fetching additional blocks while the threads search, but this will be implemented in the future. Overlapping disk I/O with computation will speed both the multicore and GPU versions of the code.

For our initial attempt at multithreading on the GPU, carving rules are copied to the 8800GTX'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 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 10 MB block, read directly from global memory. 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. A simple sequential string search is used in this version. Once all search operations for a 10 MB buffer have been completed, the locations of discovered headers and footers are copied back to the host. Note that for the GPU-assisted carving, a single host thread is used which blocks during the kernel invocation. This is deliberate, so that the performance of the GPU (rather than the host processor) can be more accurately measured. Before releasing the code, we intend to create a hybrid strategy which utilizes both the main CPU's cores *and* the GPU to maximize use of available resources.

4.2. *Experimental results*

To measure the performance of GPU-enhanced file carving, we ran carving operations on 20 GB and 100 GB disk images

Table 1 – Results for carving 20 GB disk image on dual processor, dual core Sun Ultra 40 (2.6 GHz AMD Opteron 2218 processors, 16 GB RAM)

Scalpel 1.60 ‘‘vanilla’’ (s)	2672
Scalpel 1.60 ‘‘new q’’ (s)	1784
Scalpel 1.70MT-multicore (s)	1054
Scalpel 1.70MT-gpu-0.20 (s)	860

Thirty file types, ~3 M files carved. Each result is the average of multiple, sequential runs.

using a set of 30 carving rules. All carving operations used Scalpel’s ‘‘preview’’ mode, which supports in-place carving (Richard et al., 2007). The first set of experiments was conducted on a Sun Ultra 40 with dual AMD Opteron 2218 processors, each running at 2.6 GHz. This machine had 16 GB of RAM and a 250 GB, 7200 rpm SATA hard drive. The Opteron 2218 is a dual core processor, so this machine has a total of four CPU cores. The stock graphics card in this box, an NVIDIA Quadro 5500, was removed. An NVIDIA 8800GTX graphics card was installed, which is based on the G80 GPU. The 8800GTX has 128 stream processors and 768 MB of device RAM. All of the experiments on this computer were conducted under Linux, running a 32-bit 2.6-series SMP kernel and the ext3 filesystem.

Before discussing the performance results, a brief note about the performance of Scalpel 1.60 is required. During the code review for the current research, an inefficiency in how Scalpel 1.60 handles the scheduling of its second pass over a disk image was noted and corrected. ‘‘Vanilla’’ Scalpel 1.60 performance (without the fix) is noted in the table, along with the improved version (labeled Scalpel 1.60 ‘‘new q’’). The multicore and GPU-enhanced versions of Scalpel are based on the improved version of 1.60.

The results for the 20 GB disk image are presented in Table 1. The released 1.60 version of Scalpel required 2672 s to carve approximately 3 M files of 30 different types. The improved (sequential) version of 1.60 required only 1784 s to process the 20 GB image. The multicore version (running multiple header/footer processing threads on the host CPUs) offers significantly better performance, requiring only 1054 s. Finally, when header/footer processing is offloaded to the GPU, execution time is reduced to 860 s.

Table 2 presents the results for the 100 GB disk image. The released 1.60 version of Scalpel required 13,067 s to carve approximately 15 M files of 30 different types. By improving the carve scheduling (as discussed previously) in 1.60, this time was reduced to 8725 s. The multicore version (running multiple header/footer processing threads on the host CPUs) completed processing of the disk image in 4958 s. Offloading processing to the GPU and using only a single host thread results in an execution time of 5185 s.

For both disk images, multithreading results in substantially better performance than the sequential version of the file carver; this is not unexpected. For this set of experiments, our GPU code is handling header/footer processing at least as well as four 2.6 GHz host CPU cores. While these results are promising, additional optimizations to the code running on the GPU can yield even better performance.

Next, we substantially increased the number of threads executed on the GPU and eliminated iteration over the 10 MB buffer in the string search technique. Instead of spawning a relatively small number of threads, each searching a fixed portion of the 10 MB block of data read by Scalpel, we spawned one thread per byte (e.g., approximately 10 million threads) for the input buffer. Each thread simply ‘‘stands in place’’, searching for all relevant headers and footers starting at its location in a small area of shared memory, which mirrors a portion of the buffer in device memory. This threading model is counter-intuitive for execution on commodity CPUs, because the overhead of managing so many threads would typically be prohibitive. But the G80 GPU excels at thread management and this modification substantially increases performance. Note that the string search technique being used is still very simple; we will return to this issue later in the section. The performance increase obtained by using ‘‘massive’’ threading on the GPU is detailed in the second set of experiments, described below.

The second set of experiments was conducted on a Dell XPS 710 with a single Core2Duo processor running at 2.6 GHz. This machine had 4 GB of RAM and a 500 GB, 7200 rpm SATA hard drive. The Core2Duo is a dual core processor. The same NVIDIA 8800GTX used in the Sun Ultra 40 was used. We moved our experiments to the Dell XPS because we wanted to measure the performance of multicore and GPU-based threading on a box with specifications (and cost) that more closely matched those of a ‘‘typical’’ investigative machine. At the time this paper is written (April 2007), the Sun Ultra 40 with the Quadro 5500 replaced with the 8800GTX costs approximately $9500, while the Dell XPS costs approximately $3500. 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 20 GB disk image are presented in Table 3. The improved version of Scalpel 1.60 was used as a baseline and required 1260 s to process the 20 GB image. The multicore version (running multiple header/footer processing

Table 2 – Results for carving 100 GB disk image on dual processor, dual core Sun Ultra 40 (2.6 GHz AMD Opteron 2218 processors, 16 GB RAM)

Scalpel 1.60 ‘‘vanilla’’ (s)	13,067
Scalpel 1.60 ‘‘new q’’ (s)	8725
Scalpel 1.70MT-multicore (s)	4958
Scalpel 1.70MT-gpu-0.20 (s)	5185

Thirty file types, ~15 M files carved. Each result is the average of multiple, sequential runs.

Table 3 – Results for carving 20 GB disk image on single processor, dual core Dell XPS 710 (2.4 GHz Core2Duo processor, 4 GB RAM)

Scalpel 1.60 ‘‘new q’’ (s)	1260
Scalpel 1.70MT-multicore (s)	861
Scalpel 1.70MT-gpu-0.20 (s)	686
Scalpel 1.70MT-gpu-0.30 (s)	446

Thirty file types, ~3 M files carved. Each result is the average of multiple, sequential runs.

Table 4 – Results for carving 100 GB disk image on single processor, dual core Dell XPS 710 (2.4 GHz Core2Duo processor, 4 GB RAM)

Scalpel 1.60 "new q" (s)	7105
Scalpel 1.70MT-multicore (s)	5096
Scalpel 1.70MT-gpu-0.20 (s)	4192
Scalpel 1.70MT-gpu-0.30 (s)	3198

Thirty file types, ~15 M files carved. Each result is the average of multiple, sequential runs.

threads on the host CPUs) executed in 861 s. Offloading processing to the GPU, using our original search technique (0.2), reduces execution time to 686 s. Finally, searching "in place" by spawning 10 million threads on the GPU (0.3) further reduces execution time to only 446 s. We instrumented Scalpel to determine how much time was spent in binary string searches. For the 20 GB cases, approximately 85% of execution time was used searching for headers and footers. The remainder was largely consumed by disk operations.

Table 4 presents results for processing the 100 GB disk image on the Core2Duo machine. The improved version of Scalpel 1.60 requires 7105 s. Threading on the Core2Duo reduces the time to 5096 s. Offloading searches onto the GPU, using the original search technique (0.2), requires 4192 s. The massive threading approach on the GPU (0.3) has the best running time, 3198 s.

We also conducted a number of experiments that used only a small number of carving rules. We observed the worst performance on GPU-enhanced carving when the number of carving rules was minimal and the size of the target was quite large. For example, Table 5 presents results for a 500 GB disk image for which only two file types (GIF and JPEG) were carved. 73,303 files were recovered, with the sequential and multicore versions of Scalpel taking almost exactly the same amount of time: 9946 and 9922 s, respectively. There is limited room for speedup under this scenario, since the time spent performing disk operations overwhelms the small amount of time dedicated to header/footer searches. In this experiment, GPU-enhanced Scalpel performs poorly, requiring 12,168 s. The cause is memory transfer overhead; because host memory and device memory are distinct, we must copy each chunk of the disk image to the GPU, process it, and then copy results back to host RAM. For a 500 GB image, this requires about 1 TB of device ↔ GPU memory transfers plus decoding of the results, which is performed sequentially on the host. Since there is little work to parallelize, the cost of memory transfers to and from the GPU plus the processing of results exceeds any possible speedup. While data compression is expected to help, there must be a certain level of arithmetic intensity to offset the cost of data transfer.

Table 5 – Results for carving 500 GB disk image on single processor, dual core Dell XPS 710 (2.4 GHz Core2Duo processor, 4 GB RAM)

Scalpel 1.60 "new q" (s)	9946
Scalpel 1.70MT-multicore (s)	9922
Scalpel 1.70MT-gpu-0.30 (s)	12,168

Two file types, ~73,000 files carved.

The last experiment is interesting for a number of reasons. One observation is that software using a GPU should incorporate measures of potential parallelism and below certain thresholds, the GPU should not be used. The second observation is that to overcome host to GPU and GPU to host transfer costs in the earlier experiments, the GPU was actually exhibiting remarkable speedups. The transfer rate between the G80 GPU and host, under the beta distribution of CUDA, is limited to 2 GB/s maximum. In practice, researchers are seeing much lower transfer rates. Currently, CUDA uses DMA to transfer data between the host and the GPU, but these transfers are synchronous, with computation on the GPU blocked during the entire transfer. NVIDIA has indicated that this restriction may be removed in a future release. Since data transfer rates are ultimately limited by the PCI Express bus speed, this experiment also exposes the need for a compression scheme for efficient transfer of data to the GPU and transfer of results back to the host.

4.3. *Discussion*

Our experiments reveal that incorporating GPU support is a viable method for substantially increasing the performance of digital forensics software that relies on binary string searches. We expect that relevant computations that exhibit higher "arithmetic intensity" will similarly exhibit even higher speedups on GPUs.

There are several factors in our current work that are limiting GPU performance. The first issue is that our current GPU work is based on CUDA 0.8, which is a beta release. The compiler does not generate fully optimized code (for instance, it does not perform full loop unrolling and does not effectively minimize register usage) and contains a number of bugs which require us to greatly simplify our implementation. The raw data transfer rate between the host and GPU is also not as fast as we would expect. The full release of CUDA, due out soon, should increase performance.

Another issue is that while the sequential version of Scalpel v1.60 and the multicore, multithreaded version of Scalpel are using an optimized, efficient binary string search algorithm (a modified version of the classic Boyer–Moore technique), our GPU code is currently using a simple sequential search algorithm. To illustrate this point, consider the performance of the multithreaded (for multicore CPUs) version of Scalpel running on the Dell XPS box. Under the 20 GB experiment on the Dell XPS, the multicore version of Scalpel takes 861 s. If the Boyer–Moore string search algorithm is replaced with the simple one implemented on the GPU, the time increases to 3544 s, almost four times slower. This does not necessarily represent speedup obtainable on the GPU, because the Boyer–Moore algorithm is more complicated, requires more resources, and may increase thread divergence. But it does illustrate substantial room for improvement. We are currently working on implementing an improved GPU binary string search algorithm, but this work is not yet complete.

To avoid confusing the reader, we note that in every test, the Dell XPS significantly outperformed the more expensive

Sun Ultra 40. Disk speed benchmarks illustrated virtually no difference, so this is not the reason. However, we have seen other evidence that the current generation Core2Duo processors perform significantly better than their Opteron counterparts. Thus, the sets of experiments on either box should be considered independently, rather than as pitting a quad-core machine against a dual core machine. We did not have timely access to another machine with a PCI-E-16 slot and an appropriate power supply for the 8800GTX, but will expand the number of machines in our testbed in the future.

5. Conclusion

The size of the targets that digital forensics investigators must process continues to grow and the current generation of digital forensics tools is already struggling to deal with even modest-sized targets. In addition, cutting edge tools are offering more sophisticated analysis, in an effort to reduce manual investigative techniques. This means that the computational resources of a single workstation are severely strained. As a result, digital forensics researchers must use every means available to increase the performance of their tools. Some of the possible means include paying critical attention to designing the most efficient software possible, developing software that can take advantage of modern multicore CPUs (through multithreading), using distributed processing, and as demonstrated in this paper, considering the use of commodity GPUs to speed appropriate computation.

In this paper, we illustrated that at least one type of operation common to many types of digital forensics software, namely, binary string searches, can be sped up substantially by offloading work to a GPU. While the 8800GTX used in our experiments is still relatively expensive, it will very soon be a commodity graphics card. Furthermore, future GPU designs, also based on general purpose stream processors, will offer even more computational power. Our primary purpose in writing this paper is to make it clear that it is worth the effort to develop GPU-aware digital forensics software.

6. Future work

Several efforts related to the research presented in this paper are underway. First, experimental results indicate that the data transfer to and from the host is a significant bottleneck and we are implementing a compression scheme for data streaming into and out of the GPU to address this issue.

Second, since workstations will increasingly have both (several) multicore CPUs and (potentially several) powerful GPUs, we are developing adaptive multithreading schemes that allow threads to execute on both the host CPU(s) and the GPU(s) in parallel. By executing I/O-bound and CPU-bound threads on the host CPU(s) and appropriate CPU-bound threads on the GPU, it will be possible to hide host to GPU data transfer and disk overhead.

Finally, we are also investigating alternative binary string search algorithms for the GPU. The degree to which a more complex algorithm will lead to performance gains in light of potentially increased thread divergence is still an open question.

Acknowledgments

The authors are grateful to Fareed Quaddora, the UNO CS IT Manager, for much emergency systems administration help, which allowed us to concentrate on hacking. To Mahdi Abdelguerfi for allowing us to take apart (and then take over, for nearly two weeks) his brand new four processor Sun Ultra 40. We are also very grateful to Daryl Pfeif and to Jaime Nino for helpful comments on the paper.

REFERENCES

ATI close to the metal, <http://ati.de/companyinfo/researcher/documents/ATI_CTM_Guide.pdf>.

BrookGPU, <http://graphics.stanford.edu/projects/brookgpu/>.

Cg (C for Graphics), <http://developer.nvidia.com/page/cg_main.html>.

2006 Digital forensics research workshop file carving challenge, <http://www.dfrws.org/2006/challenge/submissions/index.html>.

Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. In: Proceedings of the ACM/IEEE supercomputing conference, 2004.

Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha, D. Fast computation of database operations using graphics processors. In: Proc ACM SIGMOD, 2004.

High performance modelling of derivative prices using the PeakStream platform. PeakStream Financial Services Technical Note; September 2006.

Jacob N, Brodley C. Offloading IDS computation to the GPU. In: Proceedings of the 22nd annual computer security applications conference (ACSAC2006), 2006.

Microsoft DirectX, <http://www.microsoft.com/windows/directx/default.mspx>.

NVIDIA Compute Unified Device Architecture (CUDA), <http://developer.nvidia.com/object/cuda.html>.

OpenGL, <http://www.microsoft.com/windows/directx/default.mspx>.

Peercy M, Segal M, Gerstmann D. A performance-oriented data parallel virtual machine for GPUs. In: Proc ACM SIGGRAPH, 2006.

Richard III GG, Roussev V. Scalpel: a frugal, high-performance file carver. In: Proceedings of the 2005 digital forensics research workshop (DFRWS 2005).

Richard III GG III, Roussev V, Marziale V. In-place file carving, Research advances in digital forensics III. Springer; 2007.

Roussev V, Richard III GG. Breaking the performance wall: the case for distributed digital forensics. In: Proceedings of the 2004 digital forensics research workshop (DFRWS 2004).

The foremost file carver, <http://foremost.sourceforge.net>.

Lodovico Marziale is a graduate student and research assistant in the Computer Science Department at the University of New Orleans, and a member of the Digital Forensics Research Group there. He received a B.S. in Finance, and M.S. in Computer Science from the University of New Orleans, and is currently pursuing a M.S. in Mathematics and a Ph.D. in Computer Science. His research currently focuses on digital forensics, machine learning,

and parallel and concurrent programming. He has been spied on more than one occasion preaching the Linux gospel to random passers-by.

Golden G. Richard III is a Professor of Computer Science at the University of New Orleans and co-founder of Digital Forensics Solutions, LLC. He received the B.S. in Computer Science from the University of New Orleans (honors) and M.S. and Ph.D. degrees in Computer Science from The Ohio State University. He is a co-director of the Networking, Systems Administration, and Security Laboratory (NSSAL) at the University of New Orleans. His research interests include digital forensics, computer security, and operating systems internals. He is extremely unlikely to eat foods whose recipes contain the word "packet" or "can". It is a New Orleans thing.

Vassil Roussev is an Assistant Professor of Computer Science at the University of New Orleans. His research interests include digital forensics, high-performance computing, distributed collaboration, and software engineering. Dr. Roussev holds B.S. and M.S. degrees in Computer Science from Sofia University (Bulgaria) and M.S. and Ph.D. degrees in Computer Science from the University of North Carolina – Chapel Hill.