The Quarterly Magazine for Digital Forensics Practitioners

DIGITAL FORENSICS / MAGAZINE

WIN! A COPY OF BELKASOFT EVIDENCE CENTRE

ISSUE 14
FEBRUARY 2013

INSIDE

/ FUZZING RISKS FOR RICH HTML APPLICATIONS
/ THE CRIMINAL CONNECTION
/ BLACKBERRY FILE DELETION
/ REMOTE DATA COLLECTION

CUDA & GPU
FOR SECURITY & FORENSICS

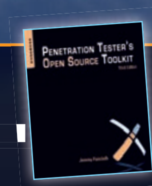
Mark Osborne takes an in-depth look at the number crunching capabilities of the Graphics Processing Unit

/ REGULARS
ROBSERVATIONS, 360, NEWS, IRQ & MORE...

/ FROM THE LAB
TARANTULA UNCOVERED – THE LATEST RELEASE

/ INTRODUCING
REMOTE DATA COLLECTION FOR FORENSIC ANALYSIS

/ BOOK REVIEWS
PENETRATION TESTERS OPEN SOURCE TOOLKIT



CUDA & GPU FOR SECURITY & FORENSICS

Exploring the opportunities that CUDA holds for revolutionizing information security.

by Mark Osborne

/ ADVANCED

This article must hold the world record for the longest time taken to write. Let me explain, in 2008, flushed with success; I had just designed/innovated a cyber security probe that had been incredibly successful. It had out performed anything on the market at that time because it used special custom hardware in conjunction with a normal Intel based CPU, the special sauce in this combination was a cooperative processing model where the Pentium CPU worked in parallel with the FPGA custom hardware. The CPU instructed the FPGA to make certain calculations and apply certain filters not statically according to some arcane policy but dynamically as processing continued. I won't be as bold to suggest it was AI, it certainly wasn't but it could do very necessary things that other products still struggle to do.

Overly flushed by my achievements, I became fascinated by the use of Graphics Processing Units (GPU) as mini supercomputers for non-graphic number crunching applications. Overly taken in by GPU marketing claims, I attended a week's intensive course on CUDA programming at a leading University and then included a GPU in later models of my probe. I was convinced that hardware assist security was the way to go. I even persuaded the long-suffering Editor of this journal to invest in an expensive CUDA capable PC. I formed my own firm with 2 dozen pre-orders on the books for my probe so I had visions of Champagne & Caviar. But my involvement with this firm can be measured in nanoseconds due to the EuroCrisis [here's a joke – how many Greek investment bankers does it take to fund a new IT company. Answer: It doesn't matter as they don't have enough money] so I looked elsewhere to follow my weird dream of cooperative hardware.

Excited by Intel's' acquisition of McAfee, I spent hours in interviews to become their UK Chief Architect. The thinking being that with Intel's massive resource in the world of hardware and with MacAfee's' lead in the world of security, their combined forces must be an endeavour to protect the world's information assets with embedded hardware; they would be just the right people to share my dream, but there were two big problems:

- They didn't [share my dream that is]; and
- They didn't like me very much

So it was not till 2012 when I narrowly missed popping my clogs and required a long period of convalescence after major surgery that I had the time to pick the article back up. Not bad nearly 4 years in the writing.

/ OBJECTIVE

The article may not be as fresh as it could have been, but I have scanned the Internet and there isn't that much out there. In this article, I want to explore the opportunities that CUDA holds for revolutionizing information security. Back in 2008, I saw that many security problems could not be solved because of lack of cheap horsepower. Wirespeed AV and Intrusion Detection Systems (IDS) were just a couple of the problems looking for an answer.

Without appearing like a catalogue for video card manufacturers, Table 1 shows the power of the home user/ graphics card range from the leading card. The key number here is the "Cores" column, processors to you and me.

Model	Cores	Memory
GeForce GTX 690	3072	2048 MB
GeForce GTX 680	1536	2048 MB
GeForce GTX 670	1344	2048 MB
GeForce GTX 660 Ti	1344	2048MB
GeForce GTX 590	1024	3072MB (1536MB per GPU)
GeForce GTX 580	512	1536 MB
GeForce GTX 570	480	1280 MB
GeForce GTX 560Ti	384	1024 MB
GeForce GTX 560	336	1024 MB
GeForce GTX 480	480	1536 MB
GeForce GTX 470	448	1280 MB
GeForce GTX 465	352	1024 MB
GeForce GTX 460	336	1 GB / 768 MB
GeForce GTX 550 Ti	192	1024 MB
GeForce GTS 450	192	1 GB
GeForce GT 640	384	2048MB
GeForce GT 630	96	512MB or 1024MB
GeForce GT 440	96	512 MB GDDR5
GeForce GT 430	96	1 GB
GeForce GT 620	96	1024MB
GeForce GT 610	48	1024MB
GeForce GT 520	48	1024MB (DDR3)
GeForce 210	16	512 MB
GeForce 8400 GS	8	256 MB

Table 1. Card Comparison

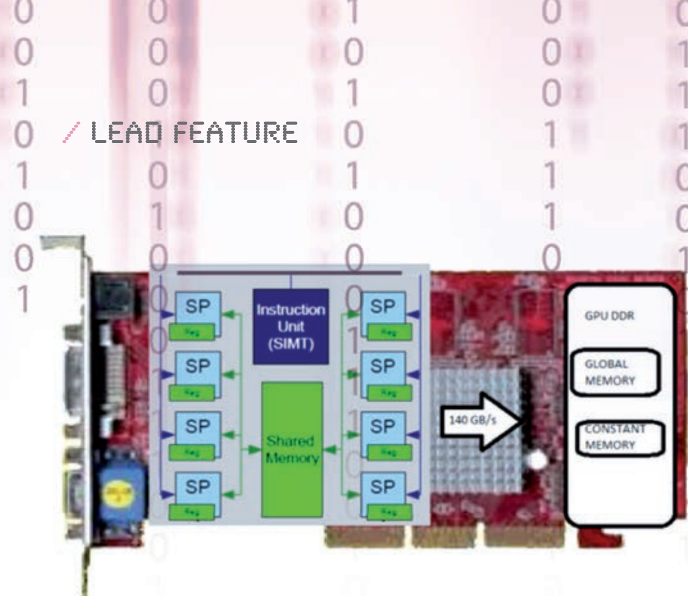


Figure 1. Basic Card Arrangement

The basic arrangement of a card showing the all too key bus speed (this will become clearer later in the article) and the memory types is shown in Figure 1.

/ THE ANALYSIS FRAMEWORK

For this article I envisaged several typical heavyweight security functions that might lend themselves to being more time efficient written in CUDA as opposed to plain old "C" (that's right no Ansi "C" or C90). To provide this "fair" comparison I will write simple "proof-of-concept" code in both languages and compare the elapse time for both.

Why elapse time? Well we don't care about the GPU processor time, we assume this baby (the GPU for the CUDA) has been installed solely to run our example code hopefully hundreds of times faster, this will give us more time at home with the wife and kids. We would expect that both system and user time would be considerably less on the host, but this is not the core objective, which is to prove the hypothesis that spending £50 on each or key systems in an environment will yield potentially marvellous benefits.

At the end of the article after we have displayed some quite shocking stats, we will highlight some of the drawbacks of the CUDA framework.

/ THE SCENARIOS

The three scenarios selected were:

- A GPU v CPU based file checksum calculator. Leading file integrity checkers are programs such as Tripwire and AFICK, these have become popular again as a result of PCI-DSS.
- A GPU v CPU based simple version of GREP.
- A simple rainbow table generator.

/ A SIMPLE GPU/CPU FILE CHECKSUM CALCULATOR

This was a simple program, it simply read every byte of the file and treating each one as an integer, generated a modulo-11 checksum. This brought back memories, I am sure I remember such an example program in a famous text by the venerable Spaf & Garfinkel.

Both the GPU and CPU function share a routine to read the data from the file and then depending on a switch passed to the program, it would perform the function either on the GPU or the CPU. The host function is shown below. It is simplicity itself, loop through every byte of the file record, dividing it by 11 and adding any remainder into a checksum.

```
//Host function to generate checksum
//
//read whole buffer and
//generate a Modulo 11
void
Hcalc_check (long *c, int *d, int inx)
{
int ind;
long sum = 0;
for (ind = 0; ind < inx + 1; ind++)
{
sum += c[ind] % 11; // divide by 11, sum remainder
}
//Leave checksum of record
//buffer into host_b[0] array
d[0] = sum;
}
```

The CUDA version isn't too difficult. The processing units are organised into threads grouped into blocks to form an array, that's just the way it works. Each dispatchable processing unit can access arrays in this format. By superimposing this array over the record buffer, each thread gets its own byte to work on.

The major draw back is that the GPU and CPU can't share memory so it has to be transferred to the GPU and back via the PCI bus. This is achieved by the "cudaMemcpy()" function.

```
// Kernel that executes on the CUDA device
__global__ void
kernel_cksum (long *a, int *b, int N)
{
int idx = blockIdx.x * blockDim.x + threadIdx.x; // turn
grid into
// a single dimension linear array
int tempsum=0;
if (idx > N)
return;
tempsum = a[idx] % 11; // checksum
// Atomicadd get exclusive control of memory and adds a
num to it
atomicAdd(&b[0], tempsum ); // waits until exclusive control
}
// Set up cuda memory & call kernel
void
Dcalc_check (long *c, int *d, int inx)
{
b_h[0] = 0;
// copy host array (*_h ) to array on CUDA device (*_d)
cudaMemcpy (a_d, a_h, size, cudaMemcpyHostToDevice);
cudaMemcpy (b_d, b_h, size, cudaMemcpyHostToDevice);
// Do calculation on device
kernel_chsum <<< n_blocks, block_size >>> (a_d, b_d, N);
//
cudaThreadSynchronize();
// Retrieve result from device and store it in host array
cudaMemcpy (b_h, b_d, size ,cudaMemcpyDeviceToHost);
cudaMemcpy (a_h, a_d, size , cudaMemcpyDeviceToHost);
//
}
```

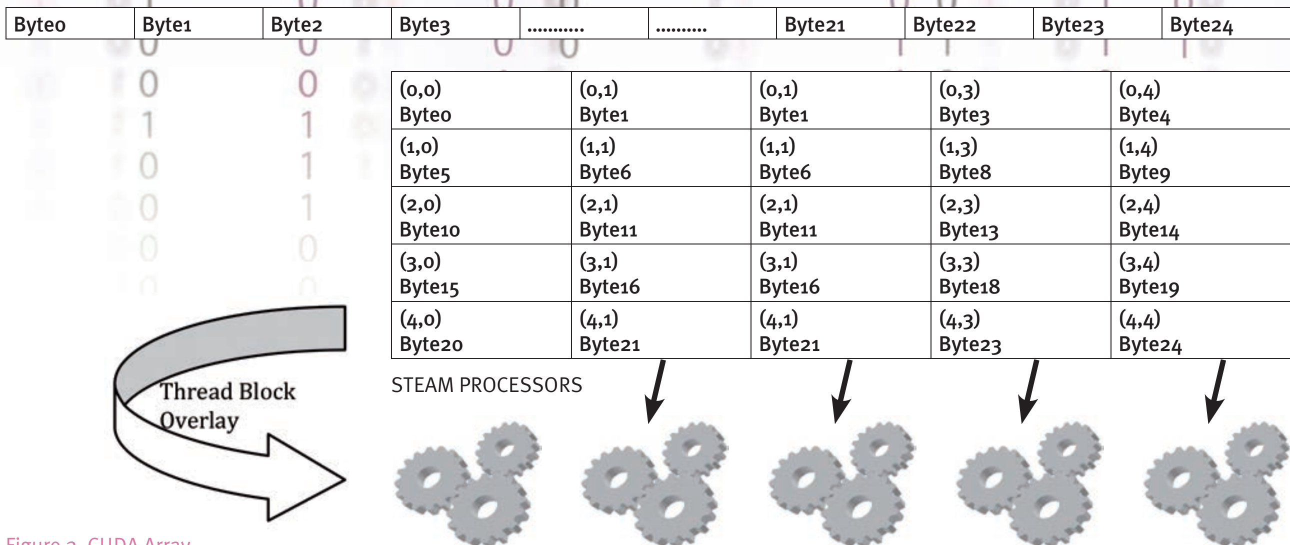


Figure 2. CUDA Array

/ RESULTS

The results blew me away and are shown in Figure 3. How could the CUDA program be so slow? Well there is a fundamental reason that is due to the inherent design of the architecture:

Lesson 1: GPU memory and processors are fast; one might believe there is some difference in relative speed but even if all processes were performed sequentially, this large difference in elapse times could not be credibly attributed to processor speed. It isn't.

Data getting to the GPU must pass over the PCI bus, this is slow; in fact, because the amount of processing per byte transferred was so small, no advantage to parallel process could be utilised. To emphasise this fact, if we add the same "cudaMemcpy" statements to host CPU code, the host program delivers slow results that are very similar. (i.e. Where the average elapse time for CUDA is 26 seconds [the mid data point], if we add the CUDA "memcopy" code into the host based code we see a rise from 0.2sec to 22.5sec; practically equal).

/ A GPU V CPU BASED SIMPLE VERSION OF GREP

This program encapsulated the rationale for me looking at hardware assisted processing. Back in the dark ages, I made a name for myself in the IDS space. One of the "bread winners" for me was IDS that couldn't keep up and needed tweaking. Mostly, there were real speed restrictions well below manufacturers' specs on the throughput capability of the devices, but that usually wasn't the problem. The problem was usually unbounded string searches.

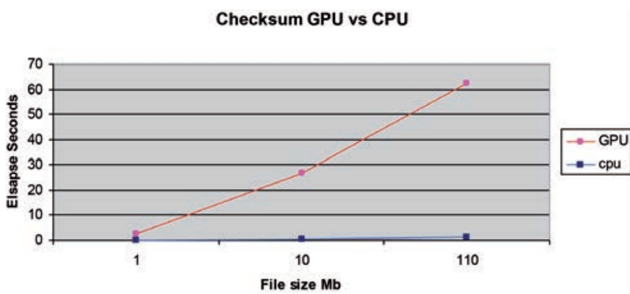


Figure 3. Checksum GPU v CPU Results

In example, I wrote two versions to look for the word "CAT" in any file. Sorry it is very basic, but I am simple man. The host code is simple. It reads a file. It loops thru the buffer to find the literal "CAT", simples!

```
void host_search()
{
char *p;
char out[10]= " ";
int cnt = LUMP_SIZE+1;
//
p = h_a ; // h_a contains data to be searched from file
// Could have just used strstr but wanted to keep code
similar
//so
// Skip thru file finding every letter "c"
while ( (p = (char *) memchr( p ,h_c[0] ,cnt )) != NULL
)
{
// if the "c" is followed by "at" i.e. cat
// then output and exit
if ( strncmp(p ,h_c,3 ) == 0 )
{
strncpy(out,p ,3 ) ;
printf("host buffy= %s \n" , out );
exit (0);
}
p++;
}
}
```

I ENVISAGED SEVERAL TYPICAL HEAVYWEIGHT SECURITY FUNCTIONS THAT MIGHT LEND ITSELF TO BEING MORE TIME EFFICIENT WRITTEN IN CUDA AS OPPOSED TO PLAIN OLD C

The GPU/CUDA version is less complex. Basically, each thread is assigned a byte in the record buffer with a different offset. When the code runs, it checks to see if its byte is “c”. If it is, the code checks if the next byte is “a”. Lastly it checks if the 3rd byte is “t”. If any of these criteria isn’t true, it sets a flag and leaves.

```
//////////////
__global__ void
ss(char *haystack, int *d, char *needle, int needleLen)
{
int startIndex = blockDim.x*blockIdx.x + threadIdx.x;
//
int fMatch = 1;
for (int i=0; i < needleLen; i++)
{
if (haystack[startIndex+i] != needle[i]) fMatch = 0;
}
if (fMatch)
atomicMin(&d[0], startIndex ) ;
//
}
```

/ RESULTS

Less unexpected now, the results of the GPU code was still disappointing (see Figure 4).

The amount of instructions per “memcopy” are still too small to show any elapse time improvement but there are more reasons for the poor performances.

Lesson 2: The fifty, 250 or 512 processors in your GPU are designed to run free on its own bit of data. Shared data can lead to a race condition where multiple processes updating one byte can corrupt it, because the two processes are using it at the same time. To avoid that we used the “atomicMIN()” routine, which locks the field while updating takes place. Where the “program” effectively is one or two operations, this means everything is single streamed. In this case, there is no advantage in using a GPU.

Good GPU code tends to use complex formulae to perform such mathematical functions to avoid locks, if you come across the term data reduction, this is what it means.

Lesson 3: GPUs are designed to run serial streams of instruction. Complex conditions cause what is known as stream divergence. Effectively, instead of running one set of commands, which has a branch, the compiler builds two separate streams bound to two threads. The despatcher on the GPU dispatches one of the two streams conditionally. Obviously this is relatively ineffective and slow.

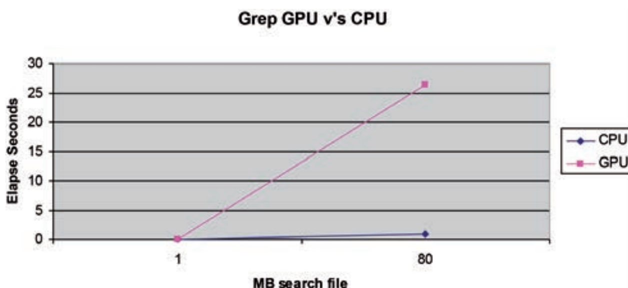


Figure 4. GREP GPU v CPU Results

Stream divergence can be avoided by using maths. In the example above, the following statement could cause divergence:

```
if (haystack[startIndex+i] != needle[i]) fMatch = 0;
```

Divergence might be avoided if the following arithmetic was used to determine if there was a difference between the two numbers. Something like this might be a good starting place.

```
fMatch += abs(haystack[startIndex+i] - needle[i]) ;
```

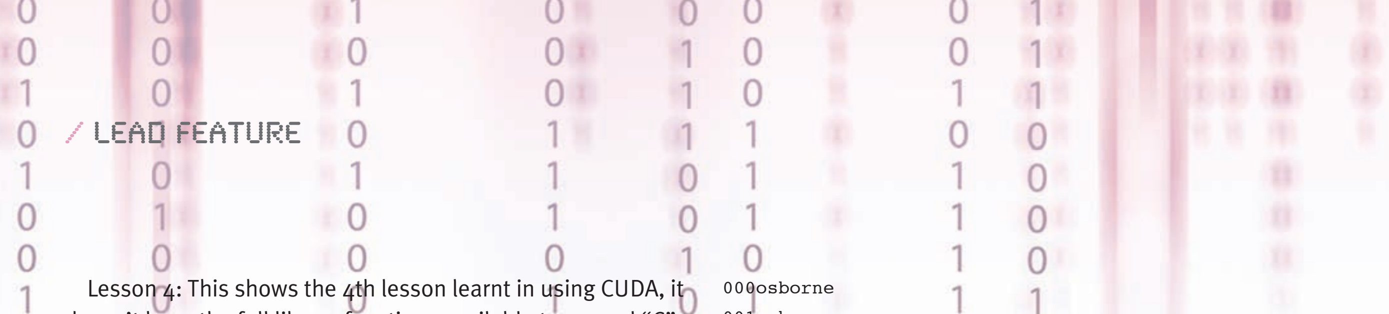
/ GO FURTHER

Frustrated by my own simplicity, I decide to make it more processor intensive by making the string search case insensitive. This means CAT, cat, Cat, CAt and caT all would match. Below is the CPU based code.

```
void host_search()
{
char *p; int cnt =0 ;
char out[10]= " ";
// LUMP_SIZE is buffer length
p = h_a ; // h_a contains data to be searched from file
while ( cnt < LUMP_SIZE + 1 )
{ // h_c contains .cat
if ( strncasecmp(p+cnt ,h_c,3 ) == 0 )
{
strncpy(out,p+cnt ,3 ) ;
printf("host buffy= %s \n" , out );
break ;
}
cnt++;
}
}
```

Below is the GPU based code.

```
//// ss - Search String Cuda Kernel ///////////
__global__ void
ss(char *haystack, int *d, char *needle, int needleLen)
{
int startIndex = blockDim.x*blockIdx.x + threadIdx.x;
uint low_hay, low_needle;
int crounds;
int fMatch = 1;
for (int i=0; i < needleLen; i++)
{
// implement to lower()
low_hay = haystack[startIndex+i] ;
if ( low_hay >= 'A' && low_hay <= 'Z')
low_hay = low_hay - 'A' + 'a';
// implement tolower()
low_needle = needle[i] ;
if ( low_needle >= 'A' && low_needle <= 'Z')
low_needle = low_needle - 'A' + 'a';
// Set flag if any character doesnt match
if ( low_hay != low_needle) fMatch = 0;
}
// store lowest occurrence of match pointer
if (fMatch)
 atomicMin(&d[0], startIndex ) ;
}
/////////////
```



Lesson 4: This shows the 4th lesson learnt in using CUDA, it doesn't have the full library functions available to normal "C". In these examples, we have had to write our own versions of tolower(), strncasecmp() etc.

Although this was an interesting exercise, the results did not change much. Using unsophisticated brute-force testing, we discovered that the GPU began to establish equality when doing greater than 10000 comparisons on 1500 bytes, the approximate size of an Ethernet frame. Putting this in practical terms, the last time I checked a typical IDS rule set it contained 8000 rules out of which around 5000 having multiple string searches. This appears to be bigger enough to justify the GPU.

As previously mentioned, since the inclusion of PCRE based regular expressions in many IDS rule sets, writers include them routinely without the proper understanding. A type PCRE rule can decompose into the equivalent of dozens of compound conditions. This suggests that GPUs could have a place in IDS processing. The Gnort Project, a research project where the PCRE plug-in has been GPU-ised, has suggested this. This project published significant results. But given the slow speed of memory movement, involved with transferring data over the PCI bus, its application for IPS would result in massive network latency. Except with special hardware, this would make GPU and IPS a marriage made in hell.

/ LASTLY, A NON-TRIVIAL RAINBOW TABLE GENERATOR

A rainbow table is a simple table that contains password hashes. These can be simply matched with a hash extracted from an OS, to reduce the time a bad guy gets in.

In this example, I imagined a Unix like password scheme where our passwords are hashed before they are stored. As we are security aware, the passwords are appended to a numeric salt ranging from 000 to 999 and the hashed with MD5. Also like Unix, we go thru a series of "rounds" of re-hashing (only in our case we just re-perform the original md5 hash as a simulation, as I like checking the MD5 output in the table). Given a password of "osborne", the os (and so our rainbow table) might store the hashed value of:

ONE OF THE "BREAD WINNERS" FOR ME WAS IDS THAT COULDN'T KEEP UP AND NEEDED TWEAKING. MOSTLY, THERE WERE REAL SPEED RESTRICTIONS WELL BELOW MANUFACTURERS' SPECS ON THE THROUGHPUT CAPABILITY OF THE DEVICES, BUT THAT USUALLY WASN'T THE PROBLEM

```
000osborne
001osborne
/\/\/\/\/\/\/\/\/
998osborne
999osborne
```

The CPU code appears below:

```
mycrack(char *text_string, char *hash_out)
{
int totalLen = 0;
uint c[4];
c[1] = 0, c[2] = 0, c[3] = 0, c[0] = 0;
totalLen = strlen( text_string);
//get the md5 hash of the word
md5_vfy(text_string ,totalLen, &c[0], &c[1], &c[2],
&c[3]);
// put a hex string into hex_out ;
sprint_as_hex ( c, hash_out, 16 ) ;
}
Main()
{
…..excluded loads of general defs and initialisation
char hash_out[32];
// iterate to emulate a bigger table
for ( rounds = 0; rounds < 100 ; rounds++ )
{
printf(" r: %i ", rounds);
for ( n = 0; n < TOTAL_WORDS ; n++ )
{
for ( x = 0; x < TOTAL_HASHS ; x++ )
{
sprintf ( p ,"%03i%s", x , words[n] );
mycrack( p, hash_out);
memcpy( rainbow [n][x] , hash_out , 32);
}
}//end of 100 rounds
}
```

The GPU below is simple and shown in two parts.

```
//GPU kernel
__global__ void mycrack(char *cuda_words, char *cuda_
rainbow)
{
//compute our index number
uint linear_hash_idx = (blockIdx.x*blockDim.x +
threadIdx.x);
uint hash_idx = threadIdx.x;
uint word_idx = blockIdx.x;
char cuda_hash_out[32];
char cuda_salt_n_word[103];
int totalLen = 0;
uint c[4];
c[0] = 0; c[1] = 0; c[2] = 0; c[3] = 0;

char ccc[] = "0123456789";
int thous, huns, tens, ones;
memset(cuda_salt_n_word,0,103 ) ;
huns = hash_idx /100 ;
cuda_salt_n_word[00] = ccc[ huns ] ;
```

```
tens = ( hash_idx - (huns *100) ) /10 ;
cuda_salt_n_word[01] = ccc[ tens ] ;
ones = hash_idx %10 ;
cuda_salt_n_word[02] = ccc[ ones ] ;
memcpy( cuda_salt_n_word+3, cuda_words+(word_idx*32), 32 );
totalLen = cuda_strlen( (char *) cuda_salt_n_word );

// cuda_salt_n_word[02] = ccc[ totalLen %10] ;
memcpy( cuda_rainbow + (linear_hash_idx*32) , cuda_
salt_n_word , 32);
if ( totalLen < 0)
{
cuda_salt_n_word[00] = cuda_salt_n_word[01] = cuda_
salt_n_word[02] = '!' ;
return;
}
//get the md5 hash of the word
md5_vfy((unsigned char *) cuda_salt_n_word ,totalLen,
&c[0], &c[1],&c[2],&c[3]);
//sprint_as_hex is a sprint in hex -- puts hash in target
sprint_as_hex ( (unsigned char *) c, cuda_hash_out, 16
) ;
memcpy( cuda_rainbow + (linear_hash_idx*32) , cuda_hash_
out , 32);
__syncthreads();
}
```

The invocation of the GPU Kernel is shown below.

```
main()
//
..........general setup etc
//
// iterate to emulate a bigger table
for ( rounds = 0; rounds < 100 ; rounds++ )
{
printf(" r: %i ", rounds);
cudaMemcpy( cuda_words , words , wordsize ,
cudaMemcpyHostToDevice);
//run the kernel
int xxx = TOTAL_WORDS ;
int yyy = TOTAL_HASHS ;
//run the kernel
dim3 dimGrid( xxx ) ;
dim3 dimBlock( yyy ) ;
mycrack<<<dimGrid, dimBlock>>>(cuda_words, cuda_
rainbow);
} // end of forced iteration
cudaMemcpy(rainbow, cuda_rainbow , rainbowsize
,cudaMemcpyDeviceToHost);

} //end dummy iteration
```

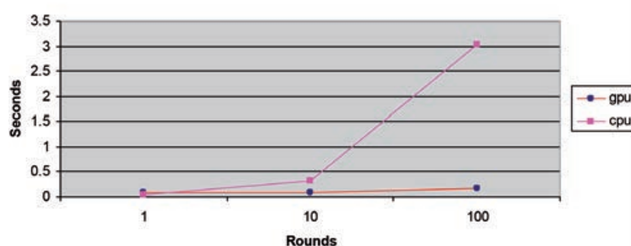


Figure 5. MD5 CPU v GPU Results

dictionary of ten words of variable length. Each of these generated a thousand table entries (i.e. 000password thru 999password) with associated hashes. To increase the workload, we varied the number of rounds (to demonstrate a Unix-like scheme but also allow us to extrapolate a dictionary of 1000 words). The GPU treated all workload with clear distain. The results are shown in Figure 5.

At last we had a task that is processor intensive enough. The GPU's performance is stunning. We can generate a million hashes in 5 or so seconds. The CPU takes a similar time to generate several 1000. This effectively means that many of our crypto techniques can be rather ineffective.

Take the Unix example, many Linux password files and reasonably recent HP-UX do not separate the hashes in a separate "shadow" file (as the previously mentioned volume from "Spaff" and Garfinkel described in the early 1990s) every user would be able to run a cracker on it. With results like this, we can extrapolate that with no argument that the cracker would certainly come up with viable password.

/ CODIFYING THE SELECTION OF APPLICATIONS TO BE GPU-ISED

So it is absolutely clear why the GPU's have not resulted in the revolution I had hoped for. This is because:

The code cannot simply be recompiled with GPU directives. Code often needs to be redesigned to operate at all. Particularly, there is an absence of libraries and most conventional code would be inefficient. This means that there is likely to be a significant software cost, as redevelopment would be required.

The PCI bus is a big constraint. To be a useful candidate, the following must be true:

```
Time transfer all units between GPU
< Time to complete workload on CPU
```

Although, this might seem obvious, I must admit at being very surprised at the slowness of the transfer and just how nippy the old PC CPU was, but I guess when you look at the relative bus speeds (as shown below) and account for the fact that most CUDA programs will have to read data into CPU based ram and then transfer it to the GPU effectively resulting in a sum of latency, the results become more explainable (Figure 6).

There are few things that could be done to minimise, the impact of this truism. CUDA supports an asynchronous copy verb with memory pinned into DMA and multiple streams, which can reduce the latency involved at the cost of complexity.

A real world example of this (if you are sad enough to have gone thru the source of SSH to establish the root cause of network latency of file transfers) is the SSH client. This uses two record buffers that are encrypted and transferred independently in a flip-flop arrangement to reduce wait time. Yet again, hardly the plug and play solution we were searching for. This is a natural candidate for such a conversion (i.e. two streams for each buffer).

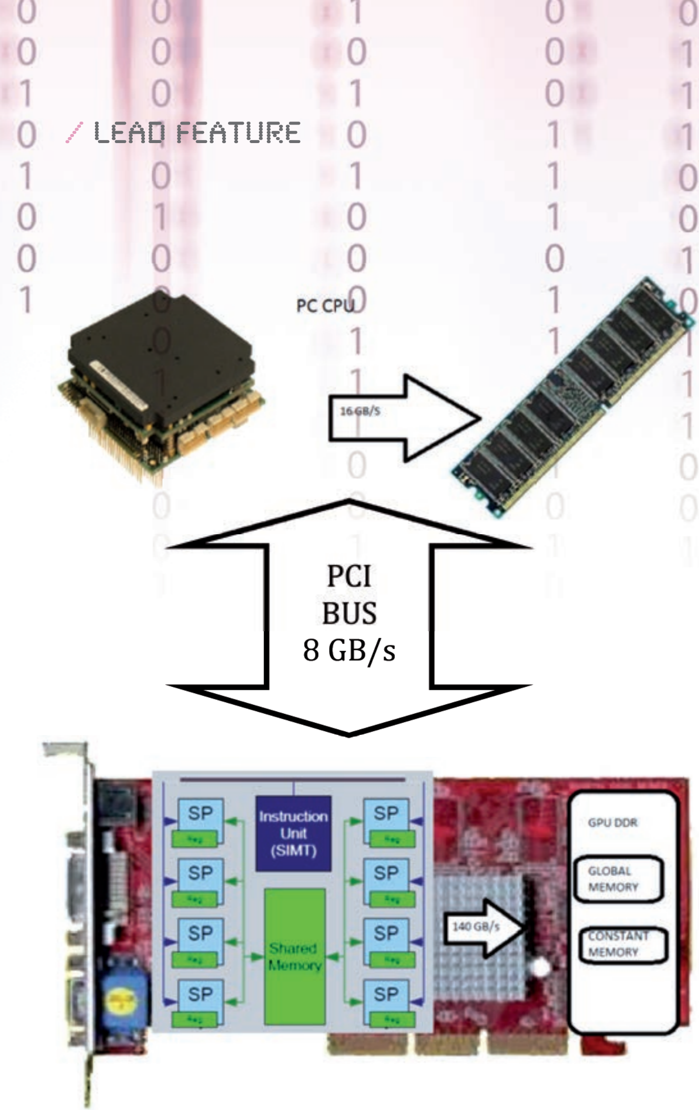


Figure 6. Relative Bus Speeds

Lastly, the impact of thread blocking and stream divergence means that many applications that are highly conditional in their nature and cannot be reduced into a mathematical formulae may remain unsuitable for porting to a GPU, so it is clear that GPUs are not the zero effort, universal panacea to all our problems, so where can they make an impact?

/ GOOD APPLICATIONS WOULD INCLUDE

Brute forcing password hashes, (as demonstrated above [and below]) is the dream application for security in a GPU. Unfortunately it is largely a negative application and is likely to render ubiquitous controls pointless. For example salting and complexity of passwords are rendered ineffective in any scheme where a password hash can be recovered. The GPU does not need to use dictionary style attacks it can attack a whole key space of 8 or 10 characters well within the average lunchtime.

Cracking encryption keys using a known text attack. Any standalone block of text encrypted using a standard algorithm will be easy prey, again a rather negative application. Finding MD5 collisions to spoof MD5 based x509 digital certificates, again a malevolent application. Developments on voice recognition or face recognition have shown some promise.

Applications that would seem to have an unfavourable byte transfer to processing ratio would seem to be File Integrity Monitoring. The processing per byte is just too low.

The jury is out regarding IDS but there is obviously some potential. In fact if you recall it was the work done on a GPU accelerators for SNORT (1) that I read about while I was designing FPGA based accelerators for SNORT that triggered my whole adventure. However, the author fervently believes that the doubling of memory transfers across the bus makes IPS difficult or impossible for latency reason. In a throw away comment in the GNORT reference (1) suggests that they are doing work on reading network data directly into the GPU. That would certainly impact the prognosis.

An area that I was hopeful that would be productive was AV. A typically antivirus product has upward of 33,000 signatures. A GPU could rattle through these comparisons at the speed of light. The GPU's are already used in generating signatures; a proof of concept to develop a GPU based system is described here (2). Apart from being a great article, it shouts a message I have been trying to convey for years; AV/IDS/IPS are not primarily there to identify an attack, they are there to provide assurance that a packet or program doesn't contain one. There is a big difference.

Perhaps an application that can really benefit is Spam processing on email because SMTP is effectively a store and forward process.

/ THE REAL WORLD

Look at "freshmeat" or "sourceforge" and you will see a bewildering array of password crackers powered by GPU's. In Backtrack4, the security testing distro, there are also several. A representative cross-section of the programs includes:

Hashcat – WPA cracking tool
Pyrit – a another WPA cracking
Multiforcer – a cracker of MD5 , MD4 and NTLM hashes.
Ikescan – a hacker tool for analysing IPSec vpns

If you download the ISO and have a play you can see just how so fast they go.

So to conclude, using GPU to solve complex muscle intensive problems instead of rendering the absolute detail of an AK47 in a "shoot'em" up game is a brilliant evolution. It is a shame that they can't be used in more utility functions, at the moment; their most viable application is cracking passwords and encryption, effectively an anti-security activity. /

REFERENCES
Gnort project, Giorgos Vasiliadis, Institute of Computer Science Foundation for Research and Technology Hellas
GPU Gems 3, Chapter 35. Fast Virus Signature Matching on the GPU, Elizabeth Seamans, Juniper Networks & Thomas Alexander, Polytime. http.developer.nvidia.com

/ AUTHOR BIO

Mark Osborne ran the KPMG security practice for many years (1993-2003). He has published several Zero-Day security vulnerabilities (e.g. Fatajack), and has also been an expert witness in the "cash-for-rides" case. Mark has designed the popular open-source wireless IDS/IPS (WIDZ), as well as the largest Cyber Security System in Europe. He is the author of "How To Cheat at Managing Information Security", which reached the Amazon.com Top 500.