# Fast, Scalable, and Secure encryption on the GPU

Rishabh Mukherjee   M. Suhail Rehman
Kishore Kothapalli   P. J. Narayanan   Kannan Srinathan
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India – 500 032.
Email:{rishabh_m@research., rehman@research.} iiit.ac.in
{kkishore@, pjn@, srinathan@} iiit.ac.in

## Abstract

*In this paper we present GPU based implementations of popular encryption schemes Blowfish and the Advanced Encryption Standard (AES). The performance of these implementations is better than a conventional CPU based implementation by a factor of* 40*x. These implementations scale linearly with a growing input size as well as the number of computational cores available on the GPU. We further analyze the security of these implementations against standard attacks on implementations (also known as* side channel attacks*), specifically timing attacks. Based on our analysis we also present versions of these encryption schemes that are resistant to such attacks. Using an* Nvidia GTX280 *we are able to obtain encryption speeds of up to 32GBps.*

## 1. Introduction

In recent years the growth in computing power of GPUs has significantly out paced that of CPUs. Contemporary GPUs have computing power in excess of Tflops and memory bandwidth in the order of several GBps. This computing power combined with a very low cost (courtesy of a huge Video Games market) makes GPUs a very viable platform for large scale computation beyond traditional computer graphics.

Until recently there were only two mainstream API's available to utilize the computing power of GPU's: The openGL standard and the proprietary Microsoft Direct3D API. These API's are primarily geared towards graphics rendering and offer very little scope for multi purpose programming. With the evolution of Nvidia's C like programming interface called *Compute Unified Device Architecture* (CUDA) and ATI/AMD's offering called Close to Metal(CTM) it has now become possible to have a high level view of the GPU as a massively multi-core computing platform with an interface close to the traditional multi threaded programming style.

A combination of the aforementioned massively multi-core architecture and the CUDA programming environment makes the GPU adept at solving problems that fall into the Single Instruction Multiple Data (SIMD) model[4]. In this model the same routine runs on different data elements in parallel. The GPU excels particularly when these routines have a high arithmetic intensity. Symmetric key cryptography appears to be a natural fit in this environment as the same encryption/decryption routine runs in parallel on different blocks of data (with possibly separate key) with a large number of arithmetic operations per encryption.

### 1.1. Motivation

While symmetric key encryption schemes are relatively much faster than asymmetric key encryption schemes, they are not fast enough to be used in real time without a substantial loss in performance or without employing expensive dedicated cryptographic accelerators. The benefits of using GPUs are manifold. GPUs can be easily integrated into all platforms (Desktops, Laptops, Servers and even mobile devices) without any fundamental changes to the hardware. GPUs are available of the shelf in a wide range of performance tiers. One may chose a GPU according to their requirements. GPUs spend most of their time rendering user displays which is almost equivalent to being idle, so it makes sense to shift compute heavy operations like encryption/decryption onto them. A GPU driven encryption system (if fast enough) could undoubtedly consume much less space and power in data intensive contexts like data centers, video conferencing and real time hard disk encryption.

### 1.2. Challenges

Implementing encryption algorithms on the GPU however is not a straightforward task. The GPU does not support many hybrid operations that modern CPUs do (like shift and rotate). Moreover, not all operations have a uniform cost on the GPU. Operations critical to cryptography like the modulus operation takes as much 20 times an ordinary addition. Optimizing programs in such an environment poses a significant challenge. To make matter worse, the GPU does not support elementary data types like the boolean data

| Algorithm | Hardware | Previous results (Throughput) | Our Performance | Speed up Vs CPU |
|---|---|---|---|---|
| Blowfish(448 Bit Key) | $GTX280$ | None | 31Gbps | 41.7x |
| Constant Time Blowfish(448 Bit Key) | $GTX280$ | None | 2.54Gbps | No Reported Work |
| AES(128 Bit Key) | $8800GTX$ | 8.28Gbps[11], 15.01Gbps[6] | 17.95Gbps | 37.1x |
| Constant Time AES(128 Bit Key) | $8800GTX$ | 18.5Gbps[14] | 13.5Gbps | 12x [7] |

Table 1: Some of Our Results. CPU results are with OpenSSL on a 2.66GHz Core 2 Duo

type which is essential for implementing many cryptographic primitives. The design choices available on the GPU (thread organization, choice of memory etc) give rise to many possible configurations, choosing from which also requires substantial effort.

Recent advances in cryptography [1], [2] have underlined the need to produce cryptography software that runs in the same fixed time(hence the name *constant time*) for any input. Writing constant time encryption on the GPU is extremely difficult. Nvidia GPUs and the CUDA platform are proprietary systems and information about significant parts of the hardware (like memory controller designs) and the software platform (like scheduling) are not public knowledge. Abstracting out the known parts and then using only them for implementing the encryption scheme and then trying to achieve a constant time, all while not sacrificing speed is quite a challenge.

## 1.3. Previous Work

Using GPUs for cryptography is not new. The first such work to deal with symmetric key cryptology appears in [3]. A similar but more refined approach is presented in [12]. Both these approaches utilize openGL and map AES to openGL constructs. This approach is limited by the unsuitability of openGL for such tasks. CUDA based approaches for AES are presented in [11], [6], [14][1]. These works explore only the AES. While AES is popular, many legacy systems continue to run various other encryption schemes like DES and Blowfish. Also these works implement a bare bones version of the AES system and do not examine the security of the implementation at all. To the best of our knowledge, the performance or security of other symmetric key crypto systems on the GPU has not been evaluated.

Our paper aims to fill this void. We have chosen two very different yet effective crypto systems to underline the suitability of the GPU across different generations of encryption schemes.

## 1.4. Our results

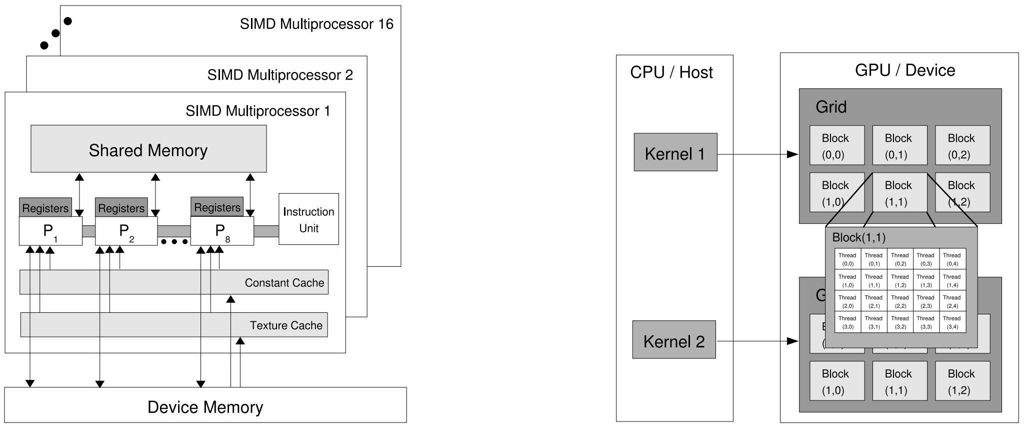
Our implementations are using Nvidia's CUDA GPGPU programming interface on the *Nvidia 8xxx* and *GTX2xx* series cards. Our results are.

- We achieve encryption rates of $31$ Gbps using Blowfish(448bit key) and $36$ Gbps using AES(128 bit key) on a *Nvidia GTX280* GPU. These are the best encryption speeds in terms of throughput per core. In absolute terms as well, our implementations outperform several previous, works. Some of our performance benchmarks and comparisons to previous works are shown in Table 1.
- Unlike other implementations that are tied specifically to a fixed GPU, our implementation is portable across any GPU. Our implementations scale linearly with input sizes as well as the number of GPU cores.
- We evaluate the performance of these implementations in terms of how efficiently they are able to utilize the computing power of the massively multi-core hardware.
- We explore possible sources of irregular behavior and hence timing attacks on the GPU. As per our knowledge this is the first attempt to analyze the security of any GPU based cryptosystem.
- We present the first constant time versions of Blowfish and AES on the GPU. The constant time version of Blowfish provides a throughput of 2.5Gbps and constant time AES provides a throughput of $24.5$ Gbps on the *GTX280*.
- We also try to ascertain what kind of encryption schemes are likely to do well with GPGPU implementations and describe problems with using schemes like Blowfish and DES on the GPU.

## 1.5. Organization of this paper

Section 2 provides an introduction to the Nvidia CUDA GPGPU platform. Section 3 contains a brief description of the three encryption schemes we have implemented. In section 4 we present details of the CUDA implementations of each of the schemes. In section 5 we showcase the benchmarks of these implementations and analyze their working. In section 6 we discuss the security of these implementations and present modified versions of these encryption schemes that address the security concerns raised. In section 7 we present some concluding remarks.

1. The implementation presented in [14] is a bitsliced version of AES. This needs a lot of input dependent preprocessing and is thus completely unsuitable for any high performance applications



(a) The CUDA memory layout (b) CUDA thread organization

Figure 1: The CUDA architecture

## 2. The Nvidia CUDA Platform

The GPU is a massively multi-threaded architecture containing hundreds of processing elements or *cores*. Each core comes with a four stage pipeline. Eight cores are grouped in SIMD fashion into a *symmetric multiprocessor (SM)*, hence all cores in an SM execute the same instruction. The GTX280 has 30 of these SMs, which makes for a total of 240 processing cores with 8 cores per SM.

The CUDA API allows a user to create a large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and multiple blocks are grouped into *grids*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads. An SM executes one warp at a time. CUDA has zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp. For this purpose, NVIDIA recommends 1024 threads (across multiple blocks) be assigned to an SM to keep it fully *occupied*.

The GPU also has different types of memory at each level (Figure 1(a)). A set of 32-bit registers is evenly divided among the threads in each SM. scratch pad memory of 16 KB, known as *shared memory*, is present at every SM and can act as a user-managed cache. This is shared among all the blocks scheduled on an SM. The GTX 280 also comes with 1 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but incurs hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp. The general, read-write access of the global memory is not cached. Thus, locality of memory access over time by a thread provides no advantages. Simultaneous memory accesses by multiple adjacent threads, however, are *coalesced* into a single transaction if they are adjacent.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Each kernel executes a grid. Prior to launching a kernel, all the data required for the computation must be transferred from the host (CPU) memory to the GPU (global) memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data (Figure 1(b)). Barrier synchronization for all threads in a block can be defined by the user in the kernel code. Apart from this, all threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches.

The other GPUs used for benchmarking i.e. the *8600GT* (32 Cores), the *8800GS*(96 Cores), the *8800GTX*(128 Cores) and the *GTX260*(196 Cores) differ only in the number of cores present on them. The architecture and memory layout is identical on these GPUs.

| Resource Parameter | Limit |
|---|---|
| No of Cores | 240 |
| Cores per SM | 8 |
| Threads per SM | 1,024 threads |
| Thread Blocks per SM | 8 blocks |
| 32-bit Registers per SM | 16,384 registers |
| Active Warps per SM | 32 warps |

Table 2: Constraints of GTX 280 on CUDA.

## 3. The Encryption Schemes

In this section we present a brief overview of the two encryption schemes we have implemented on the GPU.

### 3.1. Blowfish

Blowfish is a symmetric key encryption algorithm designed by Bruce Schneier[13]. It was intended to be used as a replacement for the aging DES. Blowfish is a 16 round Fiestal Cipher. It operates on a block size of 64 bits using a key size that varies from 32 to 448 bits. Blowfish uses a two sub-keys, an array of 18 32-bit sub-keys called a P-array and four S-Boxes with 256 32-bit sub-keys of the form $S[i][j]$.

**3.1.1. Sub-Key Generation.** All the values of P and S are initialized using a string. A simple way would be to set the string to the hexadecimal notation of $pi$. The first entry in the P-array is XORed with the first 32 bits of the key, the second entry in P with the next 32 bits of the key. Cycle through the key bits till all the entries of P have been XORed. Now encrypt a zero string with the blowfish routine. Replace $P[1]$ and $P[2]$ with the output. In the next step encrypt another zero string and replace $P[3]$ and $P[4]$ with the output. Continue doing this till all the elements of P and S have been replaced.

**3.1.2. The Encryption Scheme.** A 64 bit block of data is divided into two equal halves $X_L$ and $X_R$. One *round* involves the following operations on these two halves.

$$\begin{gathered} X_L = X_R \oplus P_i \\ X_R = F(X_L) \oplus X_R \\ \text{Swap}(X_L, X_R) \end{gathered}$$

Where $i$ is the round number and $F(X)$ is defined as

$$F(X) = ((S_{1,a} + S_{2,a} \bmod 2^{32}) \oplus S_{3,c}) + S_{4,d}$$

where $a, b, c$ and $d$ are the four octets of $X$ and $S_{i,j}$ refers to the element S[i][j].This is repeated for 16 rounds, i.e. from $i = 1$ to $i = 16$. This is followed by

$$\begin{gathered} \text{Swap}(X_L, X_R) \\ X_R = X_R \oplus P_{17} \\ X_L = X_L \oplus P_{18} \\ \text{Recombine}(X_L \text{ and } X_R) \end{gathered}$$

Decryption involves the same operations with the order of sub keys reversed. Further details may be obtained from [13]

### 3.2. The Advanced Encryption Standard (AES)

The AES is an encryption standard adopted by the U.S. Government. It refers to the Rijndael encryption scheme [5]. AES is a substitution permutation network. The cipher operates on a block length of 128 bits with possible key sizes of 128, 196 and 256 bits. AES uses two S-boxes $S_1$ and $S_2$ each with 256 1-byte entries, an encryption key $K$ (variable size) and four temporary tables $T_0$, $T_1$, $T_2$, and $T_3$ consisting of 256 4-byte entries. Described below is the encryption scheme for a 128 bit key.

Pre-computation. The four tables $T_0$, $T_1$, $T_2$, and $T_3$ are built using entries from the two S-boxes $S_1$ and $S_2$. They are expanded into the four $T_i$ tables as

$$\begin{gathered} T_0[b] = (S_2[b], S_1[b], S_1[b], S_1[b] \oplus S_2[b]) \\ T_1[b] = (S_1[b] \oplus S_2[b], S_2[b], S_1[b], S_1[b]) \\ T_2[b] = (S_1[b], S_1[b] \oplus S_2[b], S_2[b], S_1[b]) \\ T_3[b] = (S_1[b], S_1[b], S_1[b] \oplus S_2[b], S_2[b]) \end{gathered}$$

These tables are pre-computed but can also be generated on the fly during the encryption process if memory is a constraint.

The Sub-Key generation. A copy of the key $K$ is stored in the variable $x$. $x$ may be viewed as four 4-byte arrays $x_0, x_1, x_2, x_3$. We then compute a 4-byte array $e$ as

$$e = (S_1[x_3[1]] \oplus C[i], S_1[x_3[2]], S_1[x_3[3]], S_1[x_3[0]])$$

where $C$ is the constant vector

$$C = (1, 2, 4, 8, 16, 32, 64, 128, 27, 54)$$

and $i$ is the current round. We then update $x$ as

$$\begin{gathered} x_0 = e \oplus x_0 \\ x_1 = e \oplus x_0 \oplus x_1 \\ x_2 = e \oplus x_0 \oplus x_1 \oplus x_2 \\ x_3 = e \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3 \end{gathered}$$

The Encryption Scheme. The plaintext $n$ is stored in the variable $y$. It is then XORed with the key $K$. $y = n \oplus K$. We may view $y$ as four 4-byte arrays $y_0, y_1, y_2$, and $y_3$. These are then modified as

$$\begin{gathered} y_0 = T_0[y_0[0]] \oplus T_1[y_1[1]] \oplus T_2[y_2[2]] \oplus T_3[y_3[3]] \oplus x_0 \\ y_1 = T_0[y_1[0]] \oplus T_1[y_2[1]] \oplus T_2[y_3[2]] \oplus T_3[y_0[3]] \oplus x_1 \\ y_2 = T_0[y_2[0]] \oplus T_1[y_3[1]] \oplus T_2[y_0[2]] \oplus T_3[y_1[3]] \oplus x_2 \\ y_3 = T_0[y_3[0]] \oplus T_1[y_0[1]] \oplus T_2[y_1[2]] \oplus T_3[y_2[3]] \oplus x_3 \end{gathered}$$

This process is repeated 9 times. During the tenth round, we use the $S_1$ array rather than the $T_i$'s. The final value of $y$ is the encrypted value. Decryption is essentially the same process, the details of which can be obtained from [5]. This view of AES is picked up from [1].

## 4. The CUDA implementations

As discussed in section 2, the CUDA platform exposes various types of memory for storing data. Different schemes need different types of memory usage schemes. This section details what schemes are used by each scheme.

### 4.1. Implementing Blowfish

The first step in the Blowfish encryption scheme is the generation of the round keys or the $P$ array. The generation of the $P$ array is carried out by the CPU as it is an inherently sequential task. The entire data i.e. the $P$ array, the S-boxes and the plaintext are then transferred to the GPU.

| Algorithm | Plaintext Location | $P$ Array Location | $S$ Array Location |
|---|---|---|---|
| Blowfish | Global(Coalesced) | Constant(Conflict Free) | Shared Memory(With Conflicts) |
| Secure Blowfish | Global(Coalesced) | Constant Cache(Conflict Free) | Global(Perfectly Non Coalesced) |

Table 3: Mapping Blowfish Data to GPU Memory. Brackets contain information about access patterns

| Algorithm | Plaintext Location | Key $K$ | $S_1, S_2$ | $T_i$ |
|---|---|---|---|---|
| AES | Global(Coalesced) | Constant(Conflict Free) | Not on GPU | Shared Memory(With Conflicts) |
| Secure AES | Global(Coalesced) | Constant Cache(Conflict Free) | Shared Memory(No Conflicts) | Not Used |

Table 4: Mapping AES Data to GPU Memory. Brackets contain information about access patterns

**4.1.1. Memory usage scheme.** The *Constant Cache* described in section 2, is well suited for storing small amounts of data that a large number of threads utilize. Using the constant cache to store the round keys would seem to agree with that idea. In the case of Blowfish, it would seem obvious to use the constant cache to store the $P$ array. The S-box $S$ however has a large size and requires frequent and random access by the threads. It is thus best suited to being stored in the *Shared Memory*. As mentioned earlier the Shared Memory, is a fast on chip memory with 16 banks for parallel access. As different threads in a warp may access random values from the S-Box, there is a possibility that different threads will try to read values accessible through the same bank, thereby causing *bank conflicts*. While this causes some degradation in performance, it results in performance significantly better then using any other memory. The plaintext and the ciphertext are both directly read and written to global memory. As consecutive threads read/write to consecutive locations, all the global memory transactions are *coalesced*. For a summary look at Table 3.

**4.1.2. Thread organization.** Each CUDA thread reads, encrypts and writes back one block(64 bits) of plaintext. We use 512 of these threads per *CUDA Block*. This number helps maintain sufficient work in the pipeline for the GPU. Other configurations such as 256 threads or 128 threads result in slightly lower throughput as more *CUDA blocks* need to be created for the same number of Blowfish blocks.

## 4.2. Implementing AES

The first step in the AES encryption scheme is the generation of the round keys. The round keys are precomputed by the CPU. The S-Boxes $S_1$ and $S_2$ are then expanded into the four tables $T_0$, $T_1$, $T_2$, and $T_3$ on the CPU as well. Both of the pre-computation tasks are inherently sequential and are not suited to be carried out on the GPU.

**4.2.1. Memory Usage Scheme.** For AES we use the *Constant Cache* for storing the round keys and the *Shared Memory* for storing the look-up tables $T_i$. It is here that we make a departure from previous works. In [11] the look-up tables are stored in the constant cache while in [6] a single copy of the $T_i$ tables is placed in the shared memory. We place four copies the $T_i$ tables in the shared memory to consume the entire 16KB. The tables are placed in such a manner that each memory channel of the shared memory contains an entire copy of a single $T_i$ table.For example Bank 1 contains the entire $T_1$ table, Bank 2 contains the entire $T_2$ table and this goes on in a cyclic fashion till we come to bank 5 which again contains $T_1$, Bank 6 that contains $T_2$ and so on. This provides improved redundancy while accessing the $T_i$ tables. Greater redundancy results in fewer bank conflicts, which is the main factor in an improved performance as compared to [11] and [6] (See Table 1). As with Blowfish, the plaintext and ciphertext are stored in global memory. As consecutive threads read/write to consecutive locations, all global memory accesses are coalesced. For a summary look at Table 4.

**4.2.2. Thread Organization.** Each CUDA thread reads, encrypts and writes back one AES block (128 bits) of data. There are 256 of these threads per *CUDA Block*. This provides better throughput compared to other configurations like 128 or 512 threads per block.

## 5. Results and analysis

The GPUs are connected to the system through a PCIe 2.0 expansion slot. The bandwidth of this interface is significantly lower than the possible bandwidth that the GPU can provide. The throughput when considering transfers to the system memory depends on this bandwidth and also the system memory speed. We also show the comparative speeds using the OpenSSL implementations of Blowfish and AES on an Intel Core 2 Quad Q6600 clocked at 2.4 GHz(Single threaded). All test were run on a system running Fedora 9 Linux.

**Putting the results in perspective.** Recent work[10] on building performance models for the CUDA platform provides us with a framework to analyze the efficiency of our

| Algorithm | Throughput | CPU Throughput | Relative Speed Up |
|---|---|---|---|
| Blowfish | 3763 MB/s | 91 MB/S | 41.7x |
| Secure Blowfish | 326 MB/s | N/A | 3.5x(Vs Normal Blowfish on CPU) |
| AES | 4219 MB/s | 114 MB/s | 37x |
| Secure-AES | 3171 MB/s | N/A | 28x(Vs. Normal AES on CPU) |

Table 5: Throughput of the encryption schemes on the GPU Vs. CPU

code and to gauge whether we are tapping into the full potential of the GPU or not. The model establishes the relationship between the various design parameters like the number of threads per block, the amount of computation and scheduling among other parameters and uses this information to predict bounds in which the running time of the implementation is likely to lie. First we provide a brief description of the model and then use it to analyze our implementations.

### 5.1. The CUDA performance prediction model

The various design parameters of the GPU are related as

$$T(K) = \frac{N_B(K) \cdot N_w(K) \cdot N_t(K) \cdot C_T(K)}{N_C \cdot D \cdot R} \tag{1}$$

where $T(K)$ denotes the time taken by the kernel to complete execution, $N_B(K)$ denotes the number of blocks that run in serial on a SM, $N_w(K)$ denotes the number of warps per CUDA block, $N_t(K)$ denotes the number of threads per warp, $C_T(K)$ denotes the number of GPU cycles of needed for a thread to finish execution, $N_C$ denotes the number of cores, $D$ denotes the depth of pipelining and $R$ denotes the clock rate of the GPU. The details of scheduling are not public and so we consider the two extreme cases of scheduling, while evaluating the number of cycles per thread. The best effect of scheduling is to completely hide the latency incurred due to a memory access. In such a case we take the maximum of the amount of computation and the amount of memory access time. This is called the MAX model. The other extreme of scheduling is a complete failure to hide any latency. In this case we add the cycles taken for computation and memory access. This is called the SUM model. The gap between the predictions essentially compensates for any possible behavior the scheduling might have. An implementation that is satisfactorily utilizing the computing resources of the GPU is expected to have it's actual run time lie between these two bounds.

### 5.2. Analyzing the Blowfish implementation

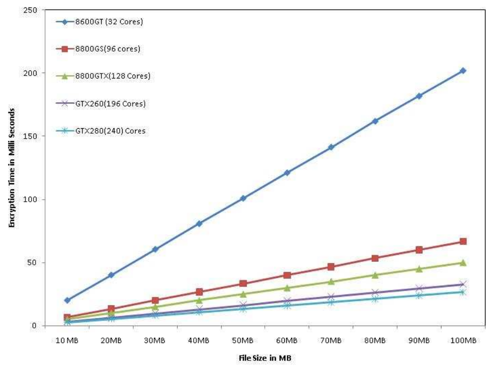
As input to our blowfish encryption program we use an input file of $N$ Bytes. Since each thread is responsible for the encryption of 8 bytes, we spawn $t = N/8$ threads. These threads are further grouped into $N/4096$ blocks of 512 threads each. These $N/4096$ blocks are divided over the 30 SM's of the GPU at an average of $N_B = N/122,880$ blocks per SM. Each of these blocks are grouped into $N_w = 16$ warps of $N_t = 32$ threads each. An inspection of the blowfish algorithm gives us an idea of the arithmetic and memory operations required. Adding up the cost of these operations tells us that a single thread needs about 2000 cycles for computation and another 1100 for memory access. For the MAX model, the above works out to $N \times 2.1 \times 10^{-10}$ ms and for the SUM model, this works out to $N \times 2.8 \times 10^{-10}$ ms. This bounds us within a throughput range of in between 4541 MB/s and 3405 MB/s for the MAX and SUM models respectively. The observed 3763 MB/s is well within these bounds.

### 5.3. Analyzing the AES implementation

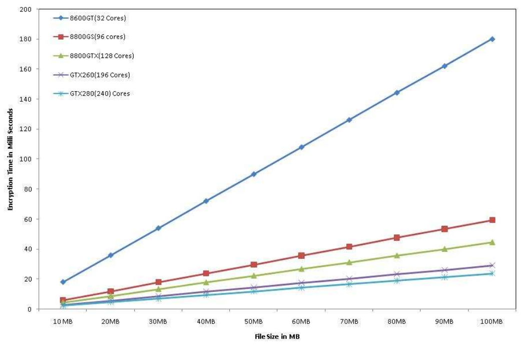
As input to our AES encryption program we use an input file of $N$ Bytes. Since each thread is responsible for the encryption of 16 bytes, we spawn $t = N/16$ threads. These threads are further grouped into $N/4096$ blocks of 256 threads each. These $N/4096$ blocks are divided over the 30 SM's of the GPU at an average of $N_B = N/122,880$ blocks per SM. Each of these blocks are grouped into $N_w = 8$ warps of $N_t = 32$ threads each. An inspection of the AES algorithm gives us an idea of the arithmetic and memory operations required. Adding up the cost of these operations tells us that a single thread needs about 1700 cycles for computation and another 850 for memory access. For the MAX model, the above works out to $N \times 1.8 \times 10^{-10}$ ms and for the SUM model, this works out to $N \times 2.65 \times 10^{-10}$ ms. This bounds us within a throughput range of in between 5341 MB/s and 3761 MB/s for the MAX and SUM models respectively. The observed 4219 MB/s is well within these bounds.

### 5.4. Scalability

When designing high performance systems, scalability is of great importance. This is particularly important in the case of cryptography applications as use cases vary from encrypting small files in a laptop to encrypting in bulk in data centers. While measuring scalability one must consider how an existing GPU scales with increasing input size as well as how increasing the computing power affects performance. We benchmark our Blowfish and AES implementations by running the implementation on variable



(a) Blowfish Performance Scaling



(b) AES Performance Scaling

Figure 2: Blowfish and AES Performance Scaling.

file sizes on a range of GPUs with varied computing power. Our benchmark process involves running the GPU implementations of encryption schemes on increasing file sizes in steps of 10MB. The benchmarks are run across five GPUs that range from the lowest GPU (the *8600GT*) to the mid range(the *8800GTX*) to the high end (the *GTX280*). As seen in Fig 2 Both Blowfish and AES scale linearly (and virtually identically) with increasing input size and varying computing power.

### 5.5. CPU Usage

For the GPU to be a viable cryptographic co-processor, it is essential that when it is being used for encryption it doesn't start to tax the host machine's CPU. In our implementations the role of the CPU is to pre-calculate the round keys and generate some tables. This is rather negligible in the context of the total amount of computation being performed. As expected, during our test runs there was virtually zero extra CPU usage on account of the GPU running the encryption program. This makes a very strong case for using the GPU to completely offload any encryption from the CPU.

## 6. Security of GPU based implementations

When we talk about the security of an implementation of a cryptographic scheme on a particular platform, we normally consider side channel attacks like Timing attacks[1], [8], [2] and Power attacks[9]. In this paper our focus is on dealing with protecting against timing attacks. Dealing with attacks based on power and heat dissipation is beyond the scope of this work.

### 6.1. Timing attacks and the GPU

Timing attacks can happen when an attacker can establish some sort of a correlation between the running times of the algorithm and the nature of the key and possibly data. This typically happens when the same operation takes different times for different inputs. Common sources of trouble include optimizations for certain types of data, caching effects and even interrupts. To defeat such attacks, the idea is to develop implementations that run in a constant number of cycles, independent of any combination of key and data, so as to make any timing based analysis impossible. To defend against such attacks it is extremely important to completely understand each and every architectural detail of a platform. Unfortunately all the aspects of the GPU architecture have not been made public. We analyze the GPU for some of these typical problems that we would face with CPU based implementations and try to make modifications according to those architectural details that are public knowledge.

**6.1.1. Effects of caching.** : Modern CPUs have clearly defined multi level caching schemes. This produces variations in run time depending upon cache hits or misses. The GPU on the other hand has no sort of implicit multi level caching of any form. The user has to explicitly store data in particular memory location, and the GPU doesn't move it anywhere at any point.

**6.1.2. Interrupts.** : On normal CPU based systems, other processes may vie for CPU time and as a result the encryption process may be swapped out in place of another process. Upon being swapped back in, not all the data associated with the encryption scheme may be present in memory resulting in cache misses and even page misses. Such a problem is not likely to occur as the GPU does not have any mechanism to *swap* a process (CUDA kernel) out.

**6.1.3. Hidden Optimizations.** : Many hardware and compiler designers implement certain optimizations that reduce the amount of computation in the case of special arithmetic operations like calculating powers of 2 or division by 2.

These optimizations may leak some information about the data. According to [4] there are no such implicit optimizations from the side of the manufacturer. As per our knowledge no evidence of such optimizations has ever been presented either.

**6.1.4. Cache bank conflicts.** : Access to fast caches is generally through a fixed number of parallel channels called banks. When more then one request ends up falling into the same access bank, the requests get serialized and take a bit longer. This may leak some information about the data. On the GPU there are a few ways to get past this. If the lookup tables are small enough(as in the case of AES) we can create multiple copies of them in the cache and stripe them across bands to make sure that there is always one entire copy of the table available through each bank.

**6.1.5. The missing links.** : The aspects of the GPU architecture that are not understood pose interesting challenges to both the attacker and the developer. Perhaps the most crucial unknown is the Scheduling scheme. As mentioned above random scheduling would make statistical inferences difficult. However one may argue that if the scheduling is not truly random (in the cryptographic sense) then, the statistical changes due to scheduling may be considered no more then *noise*. The memory controller for Global memory access is also not documented. As such there is no way to understand the effects of reading from global memory.

## 6.2. Constant Time versions

A *Constant time* encryption algorithm takes a fixed number of cycles for encrypting one block of data independent of the input data. The versions of Blowfish and AES presented so far do not do so. In this section we first identify parts of the GPU that we have full information about. We then identify possible implementation situations that may leak information about the key/data. Finally we discuss possible modifications to the implementations of these schemes to make them *Constant Time*.

**6.2.1. Identifying what to use and what not to.** To be able to standardize run time, we have to use components that we definitively understand. This would preclude those components of the GPU whose workings are not completely documented. From our point of view memory and scheduling are the major issues. The working of global and texture memory is not completely documented. The way the access banks are setup and other settings like CAS latency generally associated with Graphical memory (GDDR in this case) is not made public. Shared memory and Constant cache on the other hand are on chip registers and have a constant access time of 1 cycle. In any massively multi-core system, scheduling plays a very important role. Since the CUDA scheduling scheme is unknown, we need to make sure that the largest component unaffected by scheduling runs in constant time. From section 2 we may recall that threads are executed in half warps. In the current CUDA architecture a half warp is 16 threads. Essentially we have to make sure that these 16 threads run in constant time. This is a slight departure from the notion of constant time encryption as understood on a sequential machine where making every single encryption work in constant time is the goal.

**6.2.2. Measuring time on the GPU.** To measure time on the GPU we use the *clock()* call. This provides a GPU-cycle accurate time stamp and measures the wall-clock time, i.e. the value returned by *clock()* differs by the total number of GPU cycles elapsed between two calls to it. This GPU call however is not officially exposed by Nvidia because the CUDA compiler has a tendency to displace the position of the call during compilation. To make sure that this doesn't happen, we compiled the code to GPU level assembly code also known as *PTX*. We found the calls to the *clock()* construct to be correctly placed. To be absolutely sure we went one step ahead and inspected the actual byte-code running on the GPU hardware using a CUDA disassembler. Once again the calls to *clock()* were correctly placed. While measuring the run times, we started the clock once the plaintext block was copied onto the registers of the GPU, and stopped the clock just before the ciphertext was about to be copied out from the registers.

**6.2.3. Constant time Blowfish.** The only part of our Blowfish implementation that does not take the same time for the same operation is the look-up from the $S$ table stored in shared memory. Unlike AES, where we can use a condensed version of the tables, Blowfish tables are 4KB and we cannot create 16 copies of it in a striped fashion. A solution would be to move the $S$ look-up table to memory that does not have any issue of bank conflicts. The only other memory that we completely understand, is constant cache, but it is not large enough to fit 4KB of data. When analyzed for the number of cycles it took to perform a single encryption, every encryption was performed in 7484 cycles. However, when more than one encryption was performed, the run times started to vary, sometimes by even a hundred cycles as the number of simultaneous encryption increased. This is clearly due to bank conflicts between the various threads. The other alternative to was to place the $S$ table in the global memory in such a way that there could be no possible optimizations or conflicts while it was accessed. When an item is read from global memory, the memory controller picks up segments of memory 128 bytes wide. If multiple requests can be served from the same segment, then the time taken to service multiple request starts to depend on what is requested. To ensure that each memory access takes the same amount of time, we have to ensure no coalescing takes place. To ensure that, each entry of the $S$ table is placed in one 128 Byte segment. Essentially $S[0]$ would be placed at address $0$ in global memory, $S[1]$ would be placed at address $128$, $S[2]$ at address $256$ and so on. An appropriate modification to the statements used to access the $S$ table in the Blowfish kernel complete the task. As one would expect, accessing data from global memory in an non-coalesced manner would incur a severe performance penalty. Our measurement of 48,583 cycles per encryption would agree with the same.

**6.2.4. Constant time AES.** The only part of our AES implementation that does not take the same time for the same operation is the look-up from the $T_i$ tables stored in shared memory. Different threads of a half warp may try to read values from this table that can accessed through the same bank. These bank conflicts serialize the access for competing threads. Since there introduces some dependence of the time on the key and data it is theoretically conceivable that some information may be leaked. The solution to this would be to either use a memory that doesn't result in bank conflicts or change the access strategy altogether. Having a quick glance at section 3.3 would tell us that the tables $T_0$, $T_1$, $T_2$ and $T_3$ are formed by expanding $S_1$ and $S_2$. If we were to store only tables $S_1$ and $S_2$, and compute the values need from any of the $T_i$s when required, we would need only 512 bytes to store all the look up tables. With such a low memory foot print, these tables could be stored in the shared memory, in a striped fashion over the banks so as to ensure that there is a complete copy of both the tables available through each bank. A small branch condition in the AES routine(based on the thread id) would tell which thread where it would need to read from. To test our theory, we ran the encryption algorithm with a 1000 keys and a 1000 plain-texts resulting in 1000x1000 encryptions. Each time the encryptions took a fixed number of 5386 GPU cycles. This value remained the same whether we encrypted 1 block (resulting in 1 thread) or 16 blocks (resulting in 16 threads). On running 17 encryptions this number was double plus a non constant number of cycles that can be attributed to scheduling effects, as the 17'th thread would be part of another warp. With this modification the amount of computation that has to performed increases significantly. However a throughput of 3171 MB/s seems to suggest that a severe performance penalty is not incurred.

## 6.3. Can you really perform a timing attack ?

All this while we have taken an extremely fine tuned view of the GPU, i.e. we have sat inside the GPU and counted cycles. Will a real world attacker be afforded the same luxury ? The answer is likely to be no. All successfully demonstrated timing attacks have been on remote servers, where the host CPU provided was doing the encryption as well as providing the time-stamps. If the host were to use a GPU for encryption, there would be an additional overhead of moving data from the main memory of the host to the global memory of the CUDA device to the register on the GPU's and then once the encryption is completed moving the from the GPU's register to the GPU's global memory and to finally the hosts main memory. The variation in latencies involved in these movements run into the tens of thousands of GPU cycles on different runs of the same kay+data combination!!. The worst possible variation due to encryption is in the order of a hundred GPU cycles. Essentially the possible variation due to timing characteristics is very well within the bounds of variations due to things unrelated to the encryption. There also exists the possibility of writing a program that does time sharing on the GPU with the encryption scheme, specifically designed to reveal the memory accesses patterns of the encryption program. Such efforts may be defeated by simply locking the GPU to work with only the encrytpion Kernel. While all this does not rule out a theoretical possibility, it makes it real hard to use CPU based time-stamps to attack a GPU based cryptography implementation.

## 7. Conclusions

From our results one may conclude that cryptography on the GPU

- Is faster than any conventional CPU based implementation.
- Does not place any workload on the CPU and is thus a good cryptographic co-processor.
- Scales very well from GPUs present on laptops(like the *8600GT* to those present on workstations (like the *GTX280*) for all sorts of inputs.
- Is also safe against timing attacks in light of the work presented in this paper.

Despite the obvious benefits of using GPUs for cryptology, the lack of openness on part of GPGPU manufacturers like Nvidia about the hardware design and specifications, prevents a thorough investigation of many possible security issues. Through this work we hope to have laid to rest fears of timing based side channel attacks on AES. As was seen with Blowfish, given the current state of restricted knowledge it is not possible to write constant time versions for all encryption schemes. With the present architecture a shared memory of $S$KB with $L$ access lanes would have space for $L$ independent table(or set of tables) of maximum size $S/L$. Any encryption scheme that has lookup tables within that size range can possibly be implemented as a constant time version. Encryption schemes that require us to work with individual bits are unlikely to do well as we would either have to waste a byte storing a bit (and thus eventually run out of registers to store the plaintext) or have to unpack the bits from a byte every time we need to perform a computation(which would be too expensive). In short encryption schemes with that work with large blocks of data and small lookup tables are likely to perform the best on the GPU in terms of security and throughput.

## References

[1] Daniel J. Bernstein. Cache timing attacks on aes. *http://cr.yp.to/antiforgery/cachetiming-20050414.pdf*, 2005.

[2] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[3] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. Cryptographics: Secret key cryptography using graphics cards. In *CT-RSA*, pages 334–350, 2005.

[4] Nvidia Corporation. CUDA: Compute Unified Device Architecture Programming Guide. Technical report, 2007.

[5] Joan Daemen and Vincent Rijmen. Rijndael for aes. In *AES Candidate Conference*, pages 343–348, 2000.

[6] Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In *USENIX Security Symposium*, pages 195–210, 2008.

[7] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In *CHES*, pages 1–17, 2009.

[8] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.

[9] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[10] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan S. A performance prediction model for the cuda gpgpu platform. In *Proceedings of The 16th Annual International Conference on High Performance Computing (HiPC), Kochi, India*, pages 463–472, 2009.

[11] Svetlin A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Proceedings of the IEEE International Conference on Signal Processing and Communication*, pages 65–68, 2007.

[12] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[13] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *FSE*, pages 191–204, 1993.

[14] Jason Yang and James Goodman. Symmetric key cryptography on modern graphics hardware. In *ASIACRYPT*, pages 249–264, 2007.