CryptoGraphics: Secret Key Cryptography Using Graphics Cards

Debra L. Cook[1], John Ioannidis[1], Angelos D. Keromytis[1], Jake Luck[2]

[1] Department of Computer Science, Columbia University, New York, NY, USA
{dcook,ji,angelos}@cs.columbia.edu
[2] 10K Interactive
jake301@10k.org

Abstract. We study the feasibility of using Graphics Processing Units (GPUs) for cryptographic processing, by exploiting the ability for GPUs to simultaneously process large quantities of pixels, to offload symmetric key encryption from the main processor. We demonstrate the use of GPUs for applying the key stream when using stream ciphers. We also investigate the use of GPUs for block ciphers, discuss operations that make certain ciphers unsuitable for use with a GPU, and compare the performance of an OpenGL-based implementation of AES with implementations utilizing general CPUs. While we conclude that existing symmetric key ciphers are not suitable for implementation within a GPU given present APIs, we discuss the applicability of moving encryption and decryption into the GPU to image processing, including the handling of displays in thin-client applications and streaming video, in scenarios in which it is desired to limit exposure of the plaintext to within the GPU on untrusted clients.

Keywords: Graphics Processing Unit, Block Ciphers, Stream Ciphers, AES.

1 Introduction

We investigate the potential for utilizing Graphics Processing Units (GPUs) for symmetric key encryption. The motivation for our work is twofold. First, our initial motivation was a desire to exploit existing system resources to speed up cryptographic processing and offload system resources. Second, there is the need to avoid exposing unencrypted images and graphical displays to untrusted systems while still allowing remote viewing. While we show that moving symmetric key encryption into the GPU offers limited benefits compared to utilizing general CPUs with respect to non-graphics applications, our work provides a starting point towards achieving the second goal, by determining the feasibility of moving existing symmetric key ciphers into the GPU. Our initial intent is to determine the use of standard GPUs and configurations for cryptographic applications, as opposed to requiring enhancements to GPUs, their drivers, or other system components. Avoiding specialized requirements is necessary to provide a benefit to generalized environments. The focus of our work is on symmetric key ciphers (as opposed to asymmetric schemes) due to the general use of symmetric key ciphers for encryption of large quantities of data and the lack of basic modular arithmetic support in the standard API (OpenGL) for GPUs.

In a large-scale distributed environment such as the Internet, cryptographic protocols and mechanisms play an important role in ensuring the safety and integrity of the interconnected systems and the resources that are available through them. The fundamental building block such protocols depend on are cryptographic primitives, whose algorithmic complexity often turns them into a real or perceived performance bottleneck to the systems that employ them [4]. To address this issue, vendors have been marketing hardware cryptographic accelerators that implement such algorithms [7, 12, 14, 16, 17]. Others have experimented with taking advantage of special functions available in some CPUs, such as MMX instructions [1, 15].

While the performance improvement that can be derived from accelerators is significant [13], only a relatively small number of systems employ such dedicated hardware. Our approach is to exploit resources typically available in most systems. We observe that the large majority of systems, in particular workstations and laptops, but also servers, include a high-performance GPU, also known as a graphics accelerator. Due to intense competition and considerable demand (primarily from the gaming community) for high-performance graphics, such GPUs pack more transistors than the CPUs found in the same PC enclosure [18] at a smaller price. GPUs provide parallel processing of large quantities of data relative to what can be provided by a general CPU. Performance levels equivalent to the processing speed of 10Ghz Pentium processor have been reached, and GPUs from Nvidia and ATI are functioning as co-processors to CPUs in various graphics subsystems [18]. GPUs are already being used for non-graphics applications, but presently none are oriented towards security [11, 27].

With respect to our second goal, limiting exposure of images and graphical displays to be within a GPU, implementing ciphers within the GPU allows images to be encrypted and decrypted without writing the image temporarily as plaintext to system memory. Potential applications include thin clients, in which servers export displays to remote clients, and streaming video applications. While existing Digital Rights Management (DRM) solutions provide decryption of video within the media player and only allow authenticated media players to decrypt the images, such solutions are still utilizing the system's memory and do not readily lend themselves to generic applications exporting displays to clients [20].

Our work consists of several related experiments regarding the use of GPUs for symmetric key ciphers. First, we experiment with the use of GPUs for stream ciphers, leveraging the parallel processing to quickly apply the key stream to large segments of data. Second, we determine if AES can be implemented to utilize a GPU in a manner that allows for offloading work from other system resources (*e.g.*, the CPU). Our work illustrates why algorithms involving certain byte-level operations and substantial byte-level manipulation are unsuitable for use with GPUs given current APIs. Finally, we investigate the potential for implementing ciphers in GPUs for image processing to avoid the image being written to system memory as plaintext.

1.1 Paper Organization

The remainder of the paper is organized as follows. We provide background on the OpenGL commands and pixel processing used in our implementations in Section 2. Section 3 explains how GPUs can be utilized for the combination of a stream cipher's

keystream with data in certain applications, and includes performance results. Section 4 describes the representation of AES which we implemented in OpenGL and includes a general discussion of why certain block ciphers are not suitable candidates for use with a GPU given the existing APIs. Section 5 provides an overview of our implementation of AES that utilizes a GPU and provides performance results. We discuss the potential use of GPU-embedded versions of symmetric key ciphers in image processing and thin client applications in Section 6. Our conclusions and future areas of work are covered in Section 7. Appendix A describes the experimental environments, including the minimum required specifications for the GPUs. Appendix B contains pseudo-code for our OpenGL AES encryption routine.

2 OpenGL and GPU Background

Before describing our work with symmetric key ciphers in GPUs, we give a brief overview of the OpenGL pipeline, modeled after the way modern GPUs operate, and the OpenGL commands relevant to our experiments. The two most common APIs for GPUs are OpenGL and Direct3D [19]. We use OpenGL in order to provide platform independence (in contrast to Microsoft's Direct3D). We choose to avoid higher level languages built on top of these APIs in order to ensure that specific OpenGL commands are being used and executed in the GPU when using full hardware acceleration. Examples of such languages include Cg [8] (HLSL in DirectX [19]) and, from more recent research, Brook (the BrookGPU compiler [3] uses Cg in addition to OpenGL and Direct3D). Higher level languages do not allow the developer to specify which OpenGL commands are utilized when there are multiple ways of implementing a function via OpenGL commands and do not even guarantee the operations will be transformed into OpenGL commands but instead may transform it into C code. For example, code in a higher level language that XORs two bytes will likely be transformed into code executed in the operating system rather than converted into OpenGL commands that converts the bytes to pixels and XORs pixels. We use the OpenGL Utility Toolkit (GLUT) [28] to open the display window. GLUT serves as a wrapper for window system APIs, allowing the code to be independent of the window system.

Our implementations process data as 32 bit pixels treated as floating point values, with one byte of data stored in each pixel component.[3] We do not use OpenGL's capabilities of processing pixels as color and stencil indices, and we do not use OpenGL's vertex processing (refer to [21] and [28] for a complete description). OpenGL version 1.4 was used in all experiments. Figure 1 shows the components of the OpenGL pipeline that are relevant to pixel processing when pixels are treated as floating point values. While implementations are not required to adhere to the pipeline, it serves as a general guideline for how data is processed. We also point out that OpenGL requires support for at least a front buffer (image is visible) and a back buffer (image is not visible) but does not require support for the Alpha pixel component in the back buffer. This limits us to three bytes per pixel (the Red, Green, Blue components) when performing operations

[3] When using 32 bit pixels, 1 byte is typically dedicated to each of the Red, Green, Blue and Alpha components. A format with 10 bits for each of the Red, Green and Blue components and 2 bits for the Alpha component may also be supported.

in the back buffer. It is worth mentioning that while a 32 bit pixel format is used, the 32 bits cannot be operated on as a single 32 bit value, but rather is interpreted in terms of pixel components. For example, it is not possible to add or multiply two 32 bit integers by representing them as pixels.

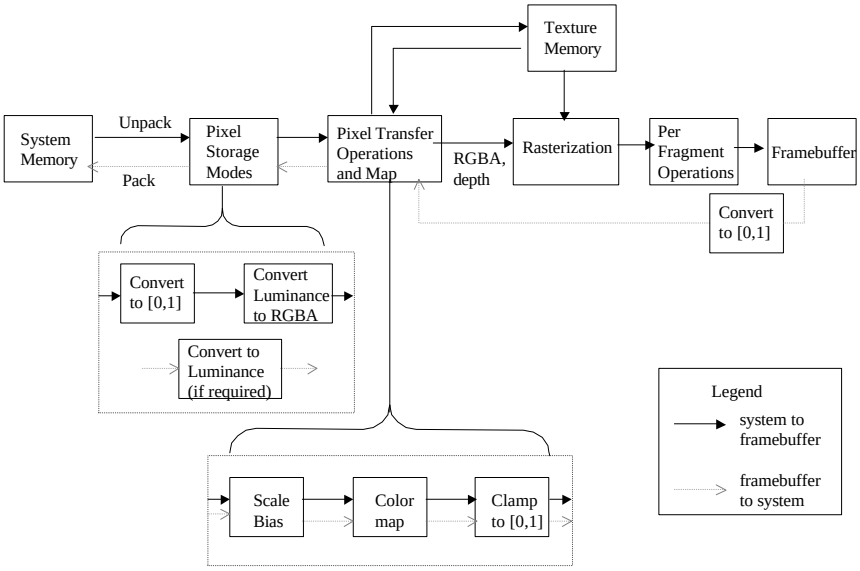

Fig. 1. OpenGL Pipeline for Pixel Processing

A data format indicating such items as number of bits per pixel and the ordering of color components specifies how the GPU interprets and packs/unpacks the bits when reading data to and from system memory. The data format may indicate that the pixels are to be treated as floating point numbers, color indices, or stencil indices. The following description concerns the floating point interpretation. When reading data from system memory, the data is unpacked and converted into floating point values in the range $[0, 1]$. Luminance, scaling and bias are applied per color component. The next step is to apply the color map, which we describe later in more detail. The values of the color components are then clamped to be within the range $[0, 1]$.

Rasterization is the conversion of data into fragments, with each fragment corresponding to one pixel in the frame buffer. In our work this step has no impact. The fragment operations relevant to pixel processing include dithering, threshold based tests, such as discarding pixels based on alpha value and stencils, and blending and logical operations that combine pixels being drawn into the frame buffer with those already in the destination area of the frame buffer. Dithering, which is enabled by default, must be turned off in our implementations in order to prevent pixels from being averaged with their neighbors.

When reading data from the frame buffer to system memory, the pixel values are mapped to the range $[0, 1]$. Scaling, bias, and color maps are applied to each of the RGBA components and the result clamped to the range $[0, 1]$. The components or luminance are then packed into system memory according to the format specified. When copying pixels between areas of the frame buffer, the processing occurs as if the pixels were being read back to system memory, except that the data is written to the new location in the frame buffer according to the format specified for reading pixels from system memory to the GPU.

Aside from reading the input from system memory and writing the result to system memory, the OpenGL commands in our implementations consist of copying pixels between coordinates, with color mapping and a logical operation of XOR enabled or disabled as needed. Unfortunately, the copying of pixels and color maps are two of the slowest operations to perform [28]. The logical operation of XOR produces a bitwise-XOR between the pixel being copied and the pixel currently in the destination of the copy, with the result being written to the destination of the copy.

A color map is applied to a particular component of a pixel when the pixel is copied from one coordinate to another. A color map can be enabled individually for each of the RGBA components. The color map is a static table of floating point numbers between 0 and 1. Internal to the GPU, the value of the pixel component being mapped is converted to an integer value which is used as the index into the table and the pixel component is replaced with the value from the table. For example, if the table consists of 256 entries, as in our AES implementation, and the map is being applied to the red component of a pixel, the 8 bits of the red value are treated as an integer between 0 and 255, and the red value updated with the corresponding entry from the table. In order to implement the tables of equation (III) in Section 4 as color maps, the tables must be converted to tables of floating point numbers between 0 and 1, and hard-coded in the program as constants. The table entries, which would vary from 0 to 255 if the bytes were in integer format, are converted to floating point values by dividing by 255. Because pixels are stored as floating point numbers and the values are truncated when they are converted to integers to index into a color map, 0.000001 is added to the result (except to 0 and 1) to prevent errors due to truncation.

3 Graphics Cards and Stream Ciphers

As a first step in evaluating the usefulness of GPUs for implementing cryptographic primitives, we implemented the mixing component of a stream cipher (the XOR operation) inside the GPU. GPUs have the ability to XOR many pixels simultaneously, which can be beneficial in stream cipher implementations. For applications that pre-compute segments of key streams, a segment can be stored in an array of bytes which is then read into the GPU's memory and treated as a collection of pixels. The data to be encrypted or decrypted is also stored in an array of bytes which is read into the same area of the GPU's memory as the key stream segment, with the logical operation of XOR enabled during the read. The result is then written to system memory. Overall, XORing the data with the key stream requires two reads of data into the GPU from system memory and one read from the GPU to system memory.

The number of bytes can be at most three times the number of pixels supported if the data is processed in a back buffer utilizing only RGB components. The number of bytes can be four times the number of pixels if the front buffer can be used or the back buffer supports the Alpha component. If the key stream is not computed in the GPU, the cost of computing the key stream and temporarily storing it in an array is the same as in an implementation not utilizing a GPU. At least one stream cipher, RC4 [26], can be implemented such that the key stream is generated within the GPU. However, the operations involved result in decreased performance compared to an implementation with a general CPU. Our work with AES serves to illustrate the problems with implementing byte-level operations within a GPU and thus we omit further discussion of RC4 within this paper. Others, such as SEAL [24] which requires 9-bit rotations, involve operations which make it difficult or impossible to implement in the GPU given current APIs.

Table 1. XOR Rate Using System Resources (CPU)

	CPU		
	1.8 Ghz	1.3 Ghz	800 Mhz
XOR Rate	139MB/s	93.9MB/s	56MB/s

We compared the rate at which data can be XORed with a key stream in an OpenGL implementation to that of a C implementation (Visual C++ 6.0). We conducted the tests using a PC with a 1.8Ghz Pentium IV processor and an Nvidia GeForce3 graphics card, a laptop with a 1.3Ghz Pentium Centrino Processor and a ATI Mobility Radeon graphics card, and a PC with a 800Mhz Pentium III Processor and an Nvidia TNT2 graphics card. Refer to Appendix A for additional details on the test environments. We provide the results from the C implementation in Table 1. We tested several data sizes to determine the ranges for which the OpenGL implementation would be useful. As expected, the benefit of the GPU's simultaneous processing is diminished if the processed data is too small. Table 2 indicates the average encryption rates over 10 trials of encrypting 1000 data segments of size $3Y^2$ and $4Y^2$, respectively, where the area of pixels is Y by Y. For the number of pixels involved in our images, the transfer rate to the GPU was measured to be equal to the transfer rate from the GPU, thus each read and write contributed equally to the overall time.

Notice that the encryption rate was fairly constant for all data sizes on the slowest processor with the oldest GPU (Nvidia TNT2). Possible explanations include slow memory controller, memory bus, or GPU, although we have not investigated this further. With the GeForce3 Ti200 card, the efficiency increased as more bytes were XORed simultaneously. On the laptop the peak rates were obtained with 200x200 to 400x400 square pixel areas.

When using the RGB components, the highest rate obtained by the GPUs compared to the C program is 58% for the Nvidia GeForce3 Ti200 card, 48.5% for the ATI Mobility Radeon card, and 51.4% for the Nvidia TNT2 card. With both the GeForce3 Ti200 and the ATI Radeon cards, results with the 50x50 pixel area was significantly slower than with larger areas due to the time to read data to/from system memory representing a larger portion of the total time. In both cases the rate is approximately 25% of that

Table 2. XOR Rate Using GPUs - RGB and RGBA Pixel Components

	Using RGB components			Using RGBA components		
Area (in pixels)	Nvidia GeForce3 Ti200	ATI Mobility Radeon 7500	Nvidia TNT2	Nvidia GeForce3 Ti200	ATI Mobility Radeon 7500	Nvidia TNT2
50x50	35.7MBps	23.5MBps	27.8MBps	49.3MBps	26.3MBps	37.0MBps
100x100	53.4MBps	38.5MBps	28.8MBps	69.2MBps	38.1MBps	38.4MBps
200x200	64.5MBps	45.5MBps	26.0MBps	86.8MBps	45.7MBps	32.0MBps
300x300	70.1MBps	45.0MBps	26.0MBps	94.8MBps	42.3MBps	32.0MBps
400x400	75.4MBps	43.0MBps	27.0MBps	95.9MBps	49.0MBps	32.8MBps
500x500	77.3MBps	38.0MBps	26.6MBps	97.5MBps	37.0MBps	32.6MBps
600x600	81.2MBps	41.7MBps	27.7MBps	105.0MBps	41.5MBps	32.8MBps

of the C program. When using the RGBA components, the highest rates on the Nvidia GeForce Ti200, ATI Radeon and Nvidia TNT2 cards are 75.5%, 52% and 68% of the C program, respectively.

4 Graphics Cards and Block Ciphers

We now turn our attention to the use of GPUs for implementing block ciphers. The first step in our work is to determine if AES can be represented in a manner which allows it to be implemented within a GPU. We describe the derivation of the OpenGL version of AES and its implementation in some detail, in order to illustrate the difficulties that arise when utilizing GPUs for algorithms performing byte-level operations. We also briefly comment on the suitability of using GPUs for block ciphers in general. While GPUs are advantageous in various aspects, the use of floating point arithmetic and the fact that the APIs are not designed for typical byte-level operations, as required in most block ciphers, present severe obstacles. For 128-bit blocks, the AES round function for encryption is typically described with data represented as a 4x4-byte matrix upon which the following series of steps are performed:

(I) SubBytes (S-Box applied to each entry)
ShiftRows (bytes within each row of the 4x4 matrix are shifted 0 to 3 columns)
MixColumns (a matrix multiplication; absent in last round)
AddRoundKey (the 4x4 matrix is XORed with a round key)

Ten rounds are performed, with the data XORed with key material prior to the first round and the MixColumns step omitted in the last round. The round function for decryption differs from encryption in that inverse functions for SubBytes, ShiftRows and MixColumns are used. Refer to [9] for a complete description of each function.

A faster implementation for environments with sufficient memory operates on 32-bit words and reduces the AES round function to four table lookups and four XORs. If A denotes a 4x4 matrix input to the round, $a_{i,j}$ denotes the i^{th} row and j^{th} column of

A, $j - x$ is computed modulo 4, and Tk are tables with 256 32-bit entries, the round function is reduced to the form:

(II) $$A'_j = T0[a_{0,j}] \oplus T1[a_{1,j-1}] \oplus T2[a_{2,j-2}] \oplus T3[a_{3,j-3}] \oplus RoundKey$$

where A'_j denotes the j^{th} column of the round's output. Refer to pages 58–59 of [6] for a complete description. The entries in the tables in (II) are concatenations of 1, 2, and 3 times the S-Box entries. This version is due to the fact that the order of the SubBytes and ShiftRows steps can be switched and the MixColumn step can be viewed as the linear combination of four column vectors, which is actually a linear combination of the S-Box entries.

The AES round function cannot easily be implemented with OpenGL as the standard series of four steps. The SubBytes can be performed using a color map, and the ShiftRows and AddRoundKey can be performed by copying pixels to change their location or to XOR them with other pixels. However, the MixColumn step would have to be expanded to a series of color maps to perform individual multiplications and copying of pixels to perform additions due to the lack of a corresponding matrix multiplication with modular arithmetic in OpenGL. The view of AES as four table lookups and XORs also cannot be implemented in OpenGL due to the lack of a 32-bit data structure. While the RGBA format is 32 bits, it is not possible to use all 32 bits as an index into a color map or to swap values between components, both of which would be necessary to implement the version in (II). As a result, we use an intermediate step in the transformation of the standard algorithm to the version in (II). Letting A'_j and $a_{i,j}$ be defined as in (II) and letting $S[a_{i,j}]$ denote the S-Box entry corresponding to $a_{i,j}$, the encryption round function for rounds 1 to 9 is represented as:

(III) $A'_j =$

$$\begin{pmatrix} 02S[a_{0,j}] \\ 01S[a_{0,j}] \\ 01S[a_{0,j}] \\ 03S[a_{0,j}] \end{pmatrix} \oplus \begin{pmatrix} 03S[a_{1,j-1}] \\ 02S[a_{1,j-1}] \\ 01S[a_{1,j-1}] \\ 01S[a_{1,j-1}] \end{pmatrix} \oplus \begin{pmatrix} 01S[a_{2,j-2}] \\ 03S[a_{2,j-2}] \\ 02S[a_{2,j-2}] \\ 01S[a_{2,j-2}] \end{pmatrix} \oplus \begin{pmatrix} 01S[a_{3,j-3}] \\ 03S[a_{3,j-3}] \\ 02S[a_{3,j-3}] \\ 01S[a_{3,j-3}] \end{pmatrix} \oplus Roundkey$$

If three tables, representing 1, 2, and 3 times the S-Box entries are stored, (III) reduces to a series of table lookups and XORs. This allows AES to be implemented using color maps and copying of pixels. The 10^{th} round is implemented as (III) with all the coefficients of 2 and 3 replaced by 1. Since decryption uses the inverses of the S-Box and matrix multiplication, five tables need to be stored, representing 0E, 0B, 0D, 09 and 01 times the S-Box inverse. Notice that this representation of AES processes data as individual bytes, instead of 4-byte words. However, the manner in which the pixel components are utilized in the implementation when encrypting multiple blocks allows 4 bytes to be processed simultaneously per pixel, compensating for the loss of not being able to use 32-bit words as in (II).

In general, algorithms performing certain byte and bit-level operations are not suitable for GPUs given current APIs. While simple logical operations can be performed efficiently in GPUs on large quantities of bytes, as shown in Section 3, the byte and bit-level operations typically found in symmetric key ciphers, such as shifts and rotates, are

not available via the APIs to GPUs. Modular arithmetic operations are also not readily available. While some operations, such as defining masks of pixels and using multiple copy commands to perform rotations and shifts on single bytes, can be performed via combinations of OpenGL commands, other operations, such as shifts across multiple bytes and table lookups based on specific bits, prove to be more difficult. For example, there is no straightforward way to implement in OpenGL the data dependent rotations found in RC6 [23] and MARS [5]. Also consider the DES S-Boxes [10]. The index into the S-Box is based on six key bits XORed with six data bits. Two of the bits are used to select the S-Box and the remaining four are the index into the S-Box. Masks of pixels copied onto the data can be used to "extract" the desired bits, but to merely XOR the six key bits with six data bits requires copying the pixel containing the desired key bits onto the pixel containing the mask with XOR turned on, doing the same for the data pixel, then copying the two resulting pixels to the same position. Color maps are required to emulate the S-Box. Overall, to use OpenGL for the S-Box step in DES, a larger number of less efficient operations are required than in a C implementation.

5 OpenGL Version of AES

5.1 Implementation Overview

We describe an implementation of AES's encryption and decryption functions for 128-bit blocks that works with any GPU supporting 32-bit pixels and OpenGL. The key schedule is not implemented inside the GPU. While the GPU allows for parallel processing of a large number of blocks, due to the simplicity in which AES can be implemented in software as a series of table lookups and XORs, the overall encryption rate using the GPU is below the rate that can be obtained with a C implementation utilizing only system resources.

The code consisted of C, OpenGL and GLUT. The C portion of the code sets up the plaintext or ciphertext and key. The OpenGL and GLUT commands are called from within the C program. GLUT commands are used to open the display window. All of the encryption and decryption computations are performed with OpenGL functions, with data being stored and processed as pixels. To accomplish this, it is necessary to represent AES in a manner that requires only the specific transformations or functions supported by the graphics hardware. As explained in Section 4, we use a representation that can be implemented in OpenGL solely via color maps and pixel copying. The implementation allows encrypting $4 * n$ blocks simultaneously, where n is the number of pixels utilized for the data being encrypted or decrypted and may be any integer less than the display's maximum pixel height supported by the GPU. The encryption of multiple blocks simultaneously from the same plaintext is useful if ECB or CTR mode are used. Alternatively, we can process one block from several messages in parallel.

Figure 2 illustrates the pixel coordinates utilized by the algorithm. The initial data blocks are read into the 16 x n area starting at the origin, indicated by "DATA" in the diagram. One byte of data is stored in each pixel component, allowing us to process $4*n$ blocks of data when all of the RGBA components are used. The i^{th} column contains the i^{th} byte of each block. This area is also used to store the output from each round. To maximize throughput, for each data block one copy of the expanded key is read into

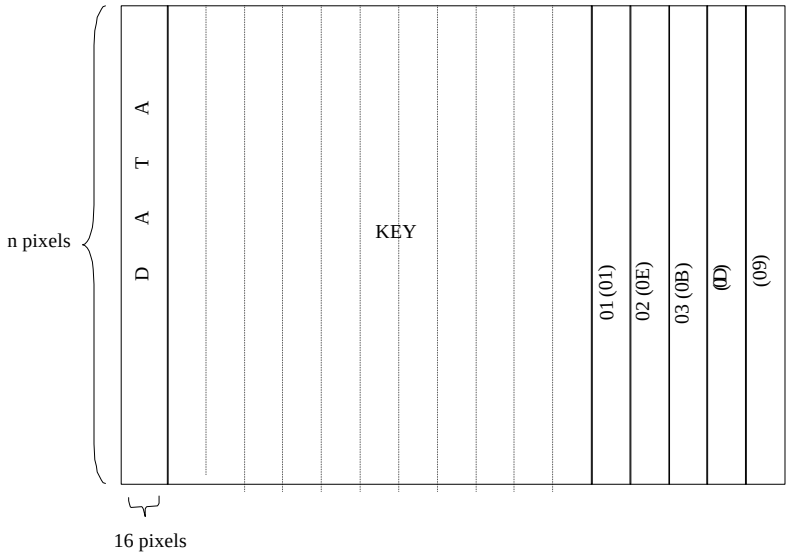

Fig. 2. Layout of Data in Pixel Coordinates used in OpenGL Version of AES

the area labeled "KEY" in the diagram. This area is 176 x n pixels starting at (17, 0) and the round keys are stored in order, each encompassing 16 columns. The tables are stored as color maps and do not appear in the layout. The data stored in the first 16 columns is copied 3 times for encryption and 5 times for decryption, applying a color map each time. The results are stored in the areas indicated by the hex values in the diagram and are computed per round. The values in parenthesis indicate the location of the transformations for decryption. The hex value indicates the value by which the S-Box (or inverse S-Box, when decrypting) entries are multiplied. See Appendix B for pseudo-code of the GPU AES encryption process. Figure 3 shows an example of the resulting display when the front buffer and RGB components are used to encrypt 300 identical data blocks simultaneously.

Two C implementations of AES are used for comparison. The first is the AES representation corresponding to variant (I) in Section 4, with the multiplication steps performed via table lookups, and reflects environments in which system resources for storing the tables required by variant (II) are not available. The second is a C implementation of variant (II), which offers increased encryption and decryption rates over (I) at the cost of requiring additional memory for tables. The code for (II) is a subset of [22].

5.2 Experiments

We compare the rate of encryption provided with the GPU to that provided by the C implementation running on the system CPU. Tests were conducted using the same

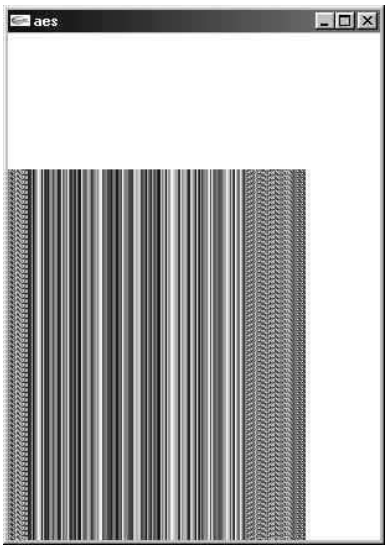

Fig. 3. Encryption of 300 Identical Blocks in RGB Components

three environments used for the stream cipher experiments. When describing the results, AES-GL indicates the implementation using OpenGL and AES-C indicates the C implementations, with the specific variant from Section 4 indicated by I and II. The AES-C programs have a hard-coded key and single 128-bit block of data. The programs expand the key then loop through encrypting a single block of data, with the output from the previous iteration being encrypted each time. No data is written to files and the measurements exclude the key setup (which is common for all variants). The AES-GL program uses a hard-coded expanded key and one or four blocks of data in the cases when the red or RGBA pixel components are used, respectively. Both the key and data are read in n times to provide n copies. Similar to the AES-C programs, the AES-GL program loops through encrypting blocks of data, with the output from the previous iteration being encrypted each time. The times exclude reading in the initial data and key, and no data is read from or written to system memory during the loop. Trials were conducted with the values of n ranging from 100 to 600 in increments of 100. The rates for values of $n \geq 300$ varied by less than 2% and the rates across all values of n varied by at most 8%. The results for AES-GL in Table 3 are the averages over $n \geq 300$ when a single pixel component and all of the RGBA pixel components are utilized. The corresponding decryption rates for the C and OpenGL implementations will be slightly lower than the encryption rates due to a small difference in the number of operations in the decryption function compared to that of the encryption function.

The layout of the pixels was chosen to simplify indexing while allowing for a few thousand blocks to be encrypted simultaneously. Since the layout does not utilize all of the available pixels, the number of blocks encrypted at once can be increased if the

Table 3. Encryption Rates for AES

	AES Version			
PC and GPU	AES-GL R	AES-GL RGBA	AES-C (I)	AES-C (II)
800Mhz Nvidia TNT2	184Kbps	732Kbps	1.68Mbps	30Mbps
1.3Ghz ATI Mobility Radeon	55Kbps	278.3Kbps	2.52Mbps	45Mbps
1.8Ghz Nvidia GeForce3	380Kbps	1.53Mbps	3.5Mbps	64Mbps

display area is utilized differently. For example, if the number of blocks is n^2, the layout can be altered such that the various segments are laid out in $n \times n$ areas instead of as columns. Performance recommendations for OpenGL include processing square regions of pixels as opposed to processing narrower rectangles [28]. We tried a modification of the program, which performed the same number of steps on square regions instead of the configuration shown in Figure 2. There was no change in the encryption rate, most likely because the program appears to be CPU-bound as we discuss in the next section. Furthermore, using square areas makes indexing more difficult and requires the number of blocks to be a perfect square for optimal utilization of the available pixels.

5.3 Performance Analysis

With the two Nvidia graphics cards, AES-GL's encryption rate was just under 50% that of AES-C (I). However, when compared to AES-C (II), the AES-GL rate was 2.4% of the AES-C version. The ratio was lower in both cases when using ATI Mobility Radeon graphics card, with the AES-GL encryption rate being 11% of AES-C (I)'s rate and less than 1% of AES-C (II)'s rate.

To determine the factors affecting AES-GL's performance, additional tests were performed in which AES-GL and AES-C were run while monitoring system resources. When we use either AES-C or AES-GL, the CPU utilization is 100% for the duration of the program. While we expect high CPU utilization for AES-C, the result is somewhat counter-intuitive for AES-GL. We believe that this happens because of the rate at which commands are being issued to the graphics card driver. Due to the simplicity in which AES is represented, a single OpenGL command resulted in one operation from AES being performed: either the table lookup or the XORing of bytes.

We do not consider the difference between the AES representations used by AES-GL and AES-C to be a factor. While the representation of AES used in AES-GL processes data as individual bytes instead of as the 32-bit words used in AES-C (II), even when excluding the processing of n pixels simultaneously the use of the RGBA components allows 4 bytes to be processed simultaneously per pixel, compensating for the loss of not being able to use 32-bit words. We also reiterate that the actions performed upon the pixels (color maps and copying) are two of the slowest GPU operations.

6 Decryption of Images Inside the GPU

The fact that symmetric key ciphers can be implemented within a GPU implies it is possible to encrypt and decrypt images in a manner that does not require the image to ever be present outside the GPU in unencrypted format. If the decrypted image is only available in the GPU, an adversary must be able to execute reads of the GPU's memory for the area utilized by the window containing the image while the image is being displayed. As a proof of concept, we use the AES-GL implementation with the image read into the card's memory in an area not utilized by AES. The data area for AES is populated by copying the image pixels into the area *in lieu* of reading data from system memory. Trivially, the image can have a stream cipher's key stream applied to it in the GPU by XORing the image with the pixel representation of the key stream.

One potential application is encrypted streaming video in which the video frames are decrypted within the back buffer of the GPU prior to being displayed, as opposed to decrypting within the system when the data is received. Typical media player screens vary from 320 x 200 pixels to 1280x1024 pixels. For low-end video, 10 frames per second (fps) is sufficient, while full-motion video requires 15 to 30 fps, with minimal perceived difference between the two rates. Assuming 8 bits per RGB component, the decryption rate must be 1.92 MBps to support 10 fps and 2.88 MBps to support 15 fps when displaying video to a 320 x 200 pixel window, rates within the limits supported by the GPUs when using stream ciphers but which exceed the rate currently obtained with AES-GL. The AES C (I) implementation also does not support these rates. For a 1280x1024 screen, 39.25 MBps support is required for 10 fps, a rate which is supported when using a stream cipher in two of the three GPUs. At 15 fps, 58.9MBps must be supported, which can only be achieved with the Nvidia GeForce3 Ti200.

A second application, less intensive than streaming video, concerns processing of displays in thin client applications. In such applications, a server sends only the updated portion of a display to the client. For example, when a user places the mouse over a link on a web page, the server may send an update that results in the link being underlined or changing color by sending an update only for the display area containing the link, thus requiring only a small amount of data to be decrypted.

When encrypting and decrypting images within the GPU, a few issues need to be resolved, such as image compression. If an image is encrypted prior to compression, ideally no compression should be possible; therefore, when encrypting and decrypting images in the GPU compression and decompression will also need to be migrated to the GPU. Second, as mentioned previously, dithering needs to be turned off. This may produce a visible side affect if the algorithm is used on large images. However, on small images, typical of a media player when not set to full screen, the lack of dithering is not likely to be noticeable. An option would be to decrypt the image in the back buffer then have dithering on when transferring the image to the front buffer, allowing decrypted images and video to be displayed with dithering.

The current AES-GL implementation reads the expanded key from the system. Alternate methods of storing the key or conveying the key to the GPU must be considered to make the key storage secure as well. We are currently experimenting with the use of remotely-keyed encryption [2] in which a smartcard or external system conveys an encrypted secret key that is decrypted within the GPU. Our current implementation re-

quires the ability to store a certificate within the GPU and utilizes RSA [25] to convey the secret key to the GPU, with limitations on the size of the RSA private key and modulus in order to contain decryption of the secret key within the GPU.

7 Conclusions

While symmetric key encryption is possible in GPUs, the lack of support via APIs for certain operations results in poor performance overall when using existing ciphers. Furthermore, current APIs do not permit some ciphers to be implemented within the GPU. The AES experiments prove it is possible to implement AES in a manner that utilizes a GPU to perform the computation while illustrating the difficulty in moving existing block ciphers into the GPU. The lessons learned from developing the OpenGL version of AES indicate GPUs are not suitable, given current APIs, for ciphers involving certain types of byte-level operations. GPUs can be used to offload a shared system CPU in applications using stream ciphers and which allow large segments of data to be combined with the key stream simultaneously.

Encryption and decryption of graphical displays and images may be moved into the GPU to avoid temporarily storing an image as plaintext in system memory. As GPU processing power and capabilities continue to increase, the potential uses will also increase. Our plans for future work include deriving a mechanism by which the key is not exposed outside the GPU and continuation of the work on remote keying of GPUs. We are also continuing work on the applicability to thin client and streaming video applications, such as video conferencing, and are designing a new cipher that can better exploit the capabilities of modern GPUs for use in these applications.

References

1. E. Biham, A Fast New DES Implementation in Software, *Workshop on Fast Software Encryption (FSE '97)*, LNCS 1267, Springer-Verlag, pages 260-272, 1997.
2. M. Blaze, J. Feigenbaum and M. Naor, A Formal Treatment of Remotely Keyed Encryption, *Proceedings of EUROCRYPT '98*, LNCS 1403, Springer-Verlag, pages 251-265, 1998.
3. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan, Brook for GPUs: Stream Computing on Graphics Hardware *Proceedings of SIGGRAPH*, 2004.
4. A. G. Broscius and J. M. Smith, Exploiting Parallelism in Hardware Implementation of the DES, *Proceedings of CRYPTO '91*, LNCS 576, Springer-Verlag, pages 367-376, 1991.
5. D. Coppersmith, et.al., The MARS Cipher, http://www.research.ibm.com/security/mars.html, 1999.
6. J. Daemon and V. Rijmen, *The Design of Rijndael: AES the Advanced Encryption Standard*, Springer-Verlag, Berlin, 2002.
7. W. Feghali, B. Burres, G. Wolrich and D. Carrigan, Security: Adding Protection to the Network via the Network Processor, *Intel Technology Journal*, 6(3), August 2002.
8. R. Fernando and M. Kilgard, *The Cg Tutorial*, Addison-Wesley, New York, 2003.
9. FIPS 197 Advanced Encryption Standard (AES), 2001.
10. FIPS 46-3 Data Encryption Standard (DES), 1999.
11. General Purpose Computation Using Graphics Hardware, http://www.gpgpu.org.
12. Helion Technology Limited, High Performance Solutions in Silicon, AES (Rijndael) Core, http://www.heliontech.com/core2.htm, 2003.

13. A. D. Keromytis, J. L. Wright and T. de Raadt, The Design of the OpenBSD Cryptographic Framework, *Proceedings of the USENIX Annual Technical Conference*, pages 181-196, 2003.
14. H. Kuo and I. Verbauwhede, Architectual Optimization for 1.82 Gbits/sec VLSI Implementation of Rijndael Algorithm, *Proceedings of CHES*, LNCS 2162, Springer-Link, pages 51-64, 2001.
15. Helger Lipmaa, IDEA: A Cipher for Multimedia Architectures? *Selected Areas in Cryptography '98*, LNCS 1556, Springer-Verlag, pages 248–263, 1998.
16. A. Lutz, J. Treichler, F.K. Gurkeynak, H. Kaeslin, G. Bosler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker and W. Fichtner, 2G bits/s Hardware Realizations of Rijndael and Serpent: A Comparative Analysis, *Proceedings of CHES*, LNCS 2523, Springer-Link, pages 144-158, 2002.
17. M. McLoone and J. McConny, High Performance Single Chip FPGA Rijndael Algorithms Implementations, *Proceedings of CHES*, LNCS 2162, Springer-Link, pages 65-76, 2001.
18. M. Macedonia, The GPU Enters Computing's Mainstream, *IEEE Computer Magazine*, pages 106-108, October 2003.
19. Microsoft DirectX, http://www.microsoft.com/windows/directx.
20. Microsoft Windows 9 Media Series Digital Rights Management, http://www.microsoft.com/windows/windowsmedia/drm.aspx, 2004.
21. OpenGL Organization, http://www.opengl.org.
22. V. Rijmen, A. Bosselaers and P. Barreto, AES Optimized ANSI C Code, http://www.esat.kuleuven.ac.be/ rijmen/rijndael/rijndael-fst-3.0.zip.
23. Rivest, Robshaw, Sidney and Yin, RC6 Block Cipher, http://www.rsa.security.com/rsalabs/rc6, 1998.
24. P. Rogaway and D. Coppersmith, A Software Optimized Encryption Algorithm, *Journal of Cryptology*, vol. 11, pages 273-287, 1998.
25. RSA Laboratories, PKCS1 RSA Encryption Standard, Version 1.5, 1993.
26. B. Schneier, *Applied Cryptography*, 2nd edition, John Wiley and Sons, New York, 1996.
27. C. Thompson, S. Hahn and M. Oskin, Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, *35th Annual IEEE/ACM International Symposium on Micro Architecture (MICRO-35)*, pages 306-317, 2002.
28. M. Woo, J. Neider, T. Davis and D. Shreiner, *The OpenGL Programming Guide*, 3rd edition, Addison-Wesley, Reading, MA, 1999.

Appendix A: Environments

GPU Requirements

For our implementations, we use OpenGL as the API to the graphics card driver. All of our programs use basic OpenGL commands and have been tested with OpenGL 1.4.0. No vendor-specific extensions are used, allowing the program to be independent of the GPU. The GPU must support 32-bit "true color" mode, because 8-bit color components are required for placing the data in pixels. At a minimum, one color component and at a maximum all four of the RGBA components are utilized by our programs. The implementations of AES and stream ciphers can be set to work with one to four pixel components. To avoid displaying the pixels to the window as the encryption is occurring, the display mode can be set to use a front and back buffer, with the rendering performed in the back buffer and the results read directly from the back buffer to system

memory and never displayed on the screen. The support for the Alpha component in the back buffer is optional in OpenGL; therefore, it may be necessary to perform rendering in the front buffer and display the pixels to the screen when utilizing all of the RGBA components.

Processors

All tests were performed in three different environments, then a subset of the tests were run in other environments to verify the correctness of the implementations with additional GPUs. The environments were selected to represent a fairly current computing environment, a laptop and a low-end PC. Both Nvidia and ATI cards were used to illustrate our implementations worked with different brands of cards, but not to compare the performance of the different graphics cards. The three environments used for all tests are:

1. A Pentium IV 1.8 Ghz PC with 256KB RAM and an Nvidia GeForce3 Ti200 graphics card with 64MB of memory. The operating system is MS Windows XP.
2. A Pentium Centrino 1.3 Ghz laptop with 256KB RAM and an ATI Mobility Radeon 7500 graphics card with 32MB of memory. The operating system is MS Windows XP.
3. A Pentium III 800 Mhz PC with 256KB RAM and an Nvidia TNT32 M64 graphics card with 32MB of memory. The operating system is MS Windows 98.

In all cases, the display was set to use 32-bit true color and full hardware acceleration. Aside from MS Windows and, in some cases a CPU monitor, no programs other than that required for the experiment were running. The CPU usage averages around 8% in each environment with only the OS and CPU monitor running. All code was compiled with Visual C++ Version 6.0. Our implementations required opening a display window, though computations may be performed in a buffer that is not visible on the screen. The window opened by the program is positioned such that it does overlap with the window from which the program was executed and to which the output of the program is written. The reason for this positioning is that movement of the display window or overlap with another active window may result in a slight decrease in performance and can interfere with the results. GLUT commands were used to open the display window.

The other GPUs we tested our programs with included an Intel© 82845G/GL Graphics Controller on a 2.3 Ghz Pentium IV processor running MS Windows XP, and a Nvidia GeForce4 Ti 4200 on a Pentium III 1.4 Ghz processor running MS Windows 2000. The AES implementation was also tested using a GeForce3 Ti200 graphics card with 64MB of memory with X11 and Redhat Linux 7.3.

Configuration Factors

In order to determine configuration factors impacting performance, we ran a series of initial tests with the OpenGL implementations of AES and the stream cipher while holding the number of bytes encrypted constant. First, since the implementation required a

GPU that was also being utilized by the display, we varied the refresh rate for the display, but that did not affect performance. Second, we varied the screen area (not the number of pixels utilized for the cipher) from 800x600 to 1600x1200. This also did not affect performance, and in the results cited for AES, we set the screen area to the minimum of 800x600 and the dimension that accommodated the number of pixels required by the test. Third, we tested the use of a single buffer with the pixels displayed to the screen versus a front and back buffer with all work performed in the back buffer and not displayed to the screen. Again, there was no change in the encryption rate. A fourth test was run to determine if there was any decrease in performance by using the GLUT or GLX libraries to handle the display. GLX is the X Window System extension to support OpenGL. In the test, we executed two versions of the program, one using GLUT and one using GLX with direct rendering, from a server with a Pentium III running Redhat Linux 7.3. There was no noticeable difference between the rates from the GLUT and GLX versions of the program.

Appendix B: AES Encryption Using OpenGL

In our OpenGL version of AES, encryption was implemented as the following steps:

Define static color maps corresponding to 1, 2, 3 times the S-Box entries.

main {
- Load the data into the DATA area.
- Load the expanded key into the KEY area.
- Turn the logical operation of XOR on.
- Copy the first key from the KEY area to the DATA area.
- Turn the logical operation XOR off.
- for (i=0; i $<$ 9; ++i) {
  - Copy the DATA area:
  - to the 01 area with the color map corresponding to 1*S-Box turned on
  - to the 02 area with the color map corresponding to 2*S-Box turned on
  - to the 03 area with the color map corresponding to 1*S-Box turned on
  - Turn color mapping off
  - Copy the pixels from areas 01,02,03 corresponding to the first term on the right hand side of (III) to the DATA area.
  - Turn the logical operation of XOR on.
  - Copy the pixels from areas 01,02,03 corresponding to the 2^{nd}, 3^{rd} and 4^{th} terms on the right hand side of (III) to the DATA area.
  - Copy the ith round key from the KEY area to the DATA area.
  - Turn the logical operation XOR off.
- }
- Copy the DATA area to the 01 area with the color map corresponding to 1*S-Box turned on.
- Turn color mapping off.
- Copy the pixels from the 01 area back to the DATA area in the order corresponding to ShiftRows.

Turn the logical operation of XOR on.
Copy the last round key from the KEY area to the DATA area.
Turn the logical operation XOR off.
Read the DATA area to system memory.
}