AES Encryption Implementation and Analysis on Commodity Graphics Processing Units

Owen Harrison and Dr. John Waldron

Computer Architecture Group, Trinity College Dublin, Dublin 2, Ireland,
harrisoo@cs.tcd.ie, john.waldron@cs.tcd.ie

Abstract. Graphics Processing Units (GPUs) present large potential performance gains within stream processing applications over the standard CPU. These performance gains are best realised when high computational intensity is required across large amounts of mostly independent input elements. The GPU's success in general purpose stream processing has been demonstrated in many diverse fields, though attempts to port cryptographic algorithms to the GPU have thus far met little success. In recent years, GPU architectures have continued to develop a more flexible and uniform programming environment. These developments have overcome a lot of previously encountered restrictions in cipher implementations. We present novel approaches for the implementation of the AES block cipher encryption algorithm on these GPUs. This work also serves as a precursor for future cipher implementations on the most advanced GPU architecture, the recently released Nvidia G80, which now includes integer support and a simplified programming interface.

Keywords: AES, Graphics Processor, GPU, Hardware Accelerated

1 Introduction

Graphical Processing Units are becoming increasingly important in the space of applications which involve data parallel processing. Over the last few years there has been an acceleration in processing power found within these commodity chips which exceeds both Moore's predictions and recent advancements in CPU performance [1]. This increase in performance is due to the distributed architecture within the GPU in the form of large numbers of simple processing units. Other processor architectures are starting to follow this model, for example the Cell processor [2]. Standard Intel and AMD chips are attempting to follow Moore's curve by increasing the number of processors available on a single die rather than increasing the core clock speed. This is the fundamental design driver behind the GPU architecture which currently boasts up to 128 parallel processors.

A recent key development within GPU design which is pertinent to this paper is the increase in its programmability. The ability to create and run a defined program within these parallel processors is the key to GPUs moving into the general purpose processing scene. Previously all control of graphics processors was through parametrised function calls using graphics programming APIs. This is ill suited for the level of hardware control required to implement a large array of general applications - we will see an example of this in Section 2. The most commonly available and existing graphics processor generation provides floating point processing capabilities only and thus encryption is not an obvious target application. This paper shows that it is possible to achieve respectable secret key cryptographic performance using this generation of GPU.

Motivation: The motivation for targeting cryptographic ciphers for GPU processing is that certain cipher uses show good characteristics for data parallel processing - high computation intensity and independent work loads. Also, as security is becoming increasingly important in the public's eyes, there is a continuing trend to secure data in all its uses, from communication to active and archived storage. This trend requires increased processing power which is being met by a combination of the standard CPU and hardware extensions in the form of cryptographic accelerators. The GPU is now ubiquitous and for the vast majority of its life is spent grossly underutilised. Unless playing games, up to 165GFlops of processing power and 54GB/s local memory bandwidth [1] are largely going to waste. There exists the potential to use this available power in the capacity of a co-processor in a similar role that existing hardware cryptographic solutions play.

This co-processing can take part or fully carry out the cryptographic needs on consumer and server platforms ideally for communications applications such as IPsec, SSL, though more probable, for bulk data encryption tasks such as secure backup/restore applications. This paper does not set out to show the GPU's suitability for each of the various security applications, but solely to demonstrate a viable possibility. A secondary motivation for employing the GPU to perform cryptographic tasks is the possibility of creating a reduced trusted computing base which is designed to hide data from the CPU. This could be used for example in the transfer of encrypted video/remote displays which is only decrypted once on the GPU. The potential for such a system has been further discussed in a paper by D. Cook et al. [3].

AES: We have selected the Advanced Encryption Standard [4] symmetric block cipher as our example cryptographic algorithm for implementation. AES was selected due to its compact nature, well documented implementation techniques and its available optimisations [5]. We have simplified our investigation to cover AES encryption using 128 bit key size only, which provides sufficient details to demonstrate the feasibility and performance of the proposed implementation approaches. Another simplification is the use of the insecure ECB [6] mode of operation. Although insecure, this mode serves as the simplest representation of modes which are suitable for parallelisation, such as CTR [7] and CWC [8]. The ability to parallelise an application is necessary with respect to achieving performance on the GPU architecture. All results presented within this paper use the OpenGL [9] graphical programming API running on Linux Fedora Core 4 using both a Geforce 6600GT AGP8x and a Geforce 7900GT PCIe graphics cards running with a 2GHz AMD CPU.

Organisation: This paper presents an overview of related work in Section 2. Sections 3 and 4 cover the relevant GPU concepts which are sufficient for following the graphical processing terminology presented. Within Section 5 we provide a brief introduction to the AES algorithm, the optimisations used and present general AES algorithm mappings to GPU hardware. Section 6 demonstrates the fundamental operation of AES, a bitwise exclusive or (XOR), its various implementation approaches and their performance results. We present the details for three different AES implementation approaches in Section 7 including results and analysis. Section 8 investigates the effectiveness of using a GPU as a parallel co processors and its interference with separate CPU running processes. Finally we present our conclusion in Section 9.

2 Related Work

Using non general purpose hardware for implementing AES or other forms of cryptographic ciphers is not new to the field of cryptography. Specific to AES recent implementations include ASIC [10] [11] [12] and custom FPGA [13] [14] designs. Using a non custom hardware approach for the execution of any algorithm will always under perform when compared to its custom counter part. The possible speeds of AES processing by custom silicon designs, such as the theoretcial 30-70Gbps proposed by A. Hodjat et al. [15], will continue to demonstrate superior performance compared to commodity approaches. The advantage of commodity ASIC design, such as the GPU, lies in its economies of scale, allowing the possibility of cryptographic co-processing for a low price per byte and also the fact that virtually all users have GPUs at their disposal by default thus any extra effective processing gained being an advantage.

There has been little use of graphics processing technology in the space of cryptography due to its previously poor suitability to the problem space. This was due to its lack of programmability and integer processing support. One notable attempt to use a GPU for AES implementation was made by D. Cook et al. [16]. Here it can be seen that the imaging subset of the graphics pipeline was used to achieve its AES lookup functionality. The imaging subset is a fixed function part of the pipeline which allows the construction of color maps. These color maps were used by [16] to simulate XOR instructions within the GPU. The authors [16] present a successful implementation of AES though the reported speeds were in the range of 184Kbps-1.53Mbps. The main obstacles encountered were due to the poor feature set available within graphics hardware at that time. For example, there was no ability to programme the most powerful components within the GPU(fragment and vertex processors) and thus the reliance on the underpowered imaging subset. The most advanced graphics processor used in this research [16] was the Geforce3 Ti200 which is currently 4 generations behind.

The approaches we present rely heavily on the ability to program the pixel pipeline for round encryption implementation. It was noted also within [16] that the CPU registers at 100% utilisation when the GPU program is running, this is still a pertinent issue with current cards and graphics drivers. We fully explore this issue within Section 8 and present a method and results which demonstrate how GPU programs share the CPU with other CPU bound processes. Further work by D. Cook et al [3] has been carried out with regards to using the GPU to encrypt video streaming showing its feasibility, however the same hardware was used as previously described and thus the same performance issues exist.

One of the first publications [17] which shows the concept of using graphical hardware to solve cryptographic problems was based on a study on cracking Unix passwords using the PixelFlow [18] architecture. This paper provides the insight that the type of hardware suitable for solving graphical problems, which requires large arrays of simple parallel processors, is suitable to more general stream based problems encountered within cryptography. Buck [19] and Venkatasubramanian [20] give more contemporary insights into the use of current generation graphical hardware for general purpose processing solving data parallel tasks. General purpose computing on graphical hardware can now be seen across a wide array of application areas such as database research [21], computer vision [22], audio and signal processing and data mining [23]. For a fuller description of general purpose processing on graphical hardware and a survey of applicable application areas, refer to Owens et al [24] and the active GPGPU community [25] involved in all types of general purpose computing on GPUs.

3 GPU Background

Within this section we present a brief overview of the current graphics processing architecture and its graphical programming model. We also pay particular attention to the GPU facets which are relevant to the implementation processes presented later in the paper. The current GPU architecture designs largely follow the structure of the programming pipeline as used by the graphic APIs. This pipeline, see Figure 1, is divided into vertex processing, rasterisation, fragment processing and raster operations stages of operation. The general approach to graphics programming is to provide the graphics driver with a list of vertices which exist within a 3 dimensional space. These vertices act as primitive descriptors, such as triangles or quadrilaterals. The vertex processing stage is responsible for transforming vertex co-ordinates and vertex attributes before passing on to the next stage. The rasterisation stage is responsible for accepting these primitives and generating a pixelised view of the particular primitive. This pixelised view comes in the form of arrays of fragments, or potential pixels, which may or may not be rendered to the screen. The rasterisation stage hands off these fragments to the fragment processing stage, which can manipulate the fragment attributes such as its colour. These fragments are outputted to the final stage, raster operation, which is ultimately responsible for writing the final pixel colour values to the active framebuffer (usually the screen framebuffer).

The hardware designs map closely to this pipeline layout, the vertex processing and fragment processing are carried out on vertex processors and fragment processors. These processors contain the majority of the processing power found within a GPU, concerning the GPU generation under study, both processors con-

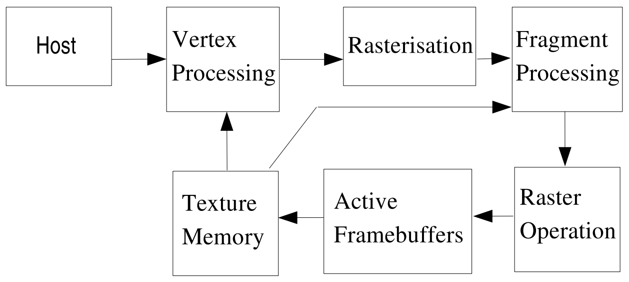

Fig. 1. A simplified view of the graphics pipeline.

tain 4 wide 32bit vector floating point processing units. The fragment processors traditionally have the most intensive task within the graphics pipeline and accordingly have the most processing power. For example the GeForce 7900 comes equipped with 48 parallel floating point processing units within the fragment processing stage compared to 8 within the vertex processing stage. The reason for GPUs outperforming CPUs in terms of performance progress is due to the expenditure of transistor budget on processing power (ie. fragment processors) rather than data movement and complex memory hierarchies. The downside to this being that the fragment processors run independently of each other, thus with no ability for fine grained synchronisation, only naturally data parallel tasks are suited to processing on GPUs. This is the reason for only supporting parallel modes of operation such as CTR and CWC. Attempts can be made to support chaining modes of operations, however performance is likely to suffer unless an efficient approach can be designed which allows a large number of multiple independent messages to be encrypted simultaneously.

The general programming model employed for general purpose processing on a GPU, and the method followed closely for the implementations presented here, is based on rendering a quadrilateral to the active framebuffer. The key is to ensure that the final rendering view port maps to the size of the rendering quadrilateral. This allows textures to be uploaded and have a one to one mapping between generated fragments from the rasteriser and the texture elements. This sets up a streaming processing model whereby all input data in the form of texture elements are made available individually and independently to each fragment processor. The fragment processor is then responsible for outputting the generated result based on the input data to its fixed output location, described below. The output can be written to the screen framebuffer or more usefully written to another texture using the OpenGL Framebuffer Object extension. This programming model can be implemented using the OpenGL or DirectX graphics libraries or can also use the CTM or CUDA frameworks recently introduced by the ATI and Nvidia corporations. There are also higher level languages which allow programmers to interact with GPUs using a more standardised programming approach such as Brook [26], however when implementing complex applications on the GPU performance can be gained by having more direct control of the graphics hardware through the aforementioned libraries/framework.

4 The GPU and AES

The GPU memory model is based on access to graphical textures, which can be viewed as 2 dimensional arrays of memory, though 1D and 3D textures also exist. These textures are available for access from both the vertex and fragment processors. These textures are accessed via the use of texture co-ordinates within the corresponding dimensional space. The following are restrictions and potential bottlenecks to memory usage in all GPU generations which are relevant to our AES implementations.

High Data Throughput: There is a high data throughput requirement, both to and from the graphics card. This data transfer must occur across the system bus which in recent years has improved with the introduction of the PCIe bus standard. We address this potential bottle neck within Section 6.

Texture Lookups: Our implementation approaches rely heavily on texture lookups, these lookups come largely in the form of sequential and dependent lookups. Dependent texture lookups are those which use the retrieved data from an initial texture lookup to form the basis of new texture coordinates to execute a further lookup. This type of lookup generally results in random gather patterns from the accessed texture and results in large slowdowns to performance. Example results from the GpuBench [27] tool shows the dramatic fall off in access speed depending on the different types of texture access, ranging from over 60GB/s for sequential access to less than 4GB/s for random access. The reason for this reduction in speed is due to the small cache sizes on the GPUs, which are normally sufficient for graphical purposes which show a high degree of spacial locality of reference. There is an emphasis on all implementation techniques to try to reduce the memory footprint of lookup tables in Section 5 and 6 and to increase the reuse patterns of memory access in Section 7. With these techniques we try to minimise the last two types of cache misses as discussed in Hill et al. [28], namely conflict and capacity misses.

Gather and Scatter: Gather is supported in terms of texture reads from various locations, however, a notable restriction is the common lack of native scatter support within the fragment processors. Each fragment processor can output a small number of results (normally between 1 and 4), however these results must be written to a predetermined memory location within the active output framebuffers. This is due to traditional graphics programming where each potential pixel is associated with only one pixel location on the screen/framebuffer. This as we will see in Section 7 restricts our output format for our AES implementation strategies and also causes a further restriction on the input format for one of them.

Another relevant area of the GPU is the availability of a logical operation stage within the final stage of the pipeline. There is hardware support for this type of operation and more specifically XOR within the raster operations units

(ROPs) of current designs. This allows the combination of the fragment processor output and the existing data within the active framebuffer to be combined using XOR. There does not exist support for XOR within current (prior to DirectX10 hardware support) designs of the fragment processors. ROPs can only be used at the end of the rendering pipeline and exist in fewer numbers compared to the fragment processors. We use this type of XOR functionality in both Section 6 and Section 7 and further discuss the restrictions imposed by its availability in the ROP only.

The last GPU feature of note is the fragment processors ability to implement swizzle operations for free. Data stored within textures can be addressed and operated upon within the various processing stages having the option of being represented as groups of 4 8bit components. This is due to traditional graphics processing commonly requiring work on RGBA (red, green, blue, alpha) groups, each of which is referred to as a component. Within the fragment processor there is the ability to change the ordering of these RGBA vectors during operations. This provides a useful means for cheaply executing byte rotates and ultimately leads to an optimisation of the Rijndael cipher to further reduce active memory footprint, which we will explore in Sections 5 and 7.

5 AES Background

The Advanced Encryption Standard (AES) [4] was introduced in 2001 by the National Institute of Standards and Technology in response to the aging concerns of DES [29]. The standard adopted a restricted version of the Rijndael [5] symmetric block cipher which can encrypt and decrypt plaintext blocks of size 128 bits using a key size of 128-bit,192-bit or 256-bit length. The Rijndael cipher was selected due to its compact simple structure and suitability to commonly available 8-bit and 32-bit processing platforms. The cipher is based on executing a number of round transformations on plaintext, each round's output is the next round's input. The number of rounds is determined by the key length, 128-bit uses 10 rounds, 192-bit uses 12 and 256-bit uses 14. We have selected to use only 128-bit and thus 10 rounds in all AES implementations within this paper.

Each round consists of largely the same steps except for an extra addition of a round key before starting and the lack of a MixColumns step in the last iteration. These steps operate on 128 bits of data called the State which transform the input State into 128 bits of output State ready for the next stage. The State, which consists of a 16 byte block, is generally viewed as a 4 x 4 table of bytes. The round stages are Sub Bytes (non-linear byte substitution using an S-box lookup table), Shift Rows (cyclical shifting of bytes in each row), Mix Columns (linear transformation which mixes column State data) and Add Round Key (XOR addition of a scheduled round key with State data).

These rounds can be reduced into a simplified equation, see Equation 1, as presented in the original Rijndael cipher proposal [5], which we will be using with our implementations. This equation reduces the number of operations involved by using 4 1K table lookups whose results need to be XORed with each other and the round key. In an attempt to reduce the active memory footprint used within each round we also have adopted a variation of the further reduced equation shown in Equation 2 from the same proposal. We can see that it reduces the table lookup to a single 1K table which will reduce the caching demands of this part of the implementation. This equation incurs a penalty of three extra rotates per column per round on the output of each table lookup, these rotates can be implemented using the free swizzle operations leading to an further optimisation of this equation as covered in Section 7. As we can see from the two equations that have been selected for implementation, which we describe as the noROT and ROT approaches, the operations involved are byte selects (swizzle), XORs (denoted by $\bigoplus$) and table lookups (denoted by $T_i[]$). We will separately address the issue of efficient XOR implementation on current GPU hardware in Section 6.

$$\begin{aligned} e_j &= T_0[a_{(0,j)}] \oplus T_1[a_{(1,j-c1)}] \oplus \\ &\quad T_2[a_{(2,j-c2)}] \oplus T_3[a_{(3,j-c3)}] \oplus k_j \ . \end{aligned} \tag{1}$$

$$\begin{aligned} e_j &= k_j \oplus T_0[a_{(0,j)}] \oplus Rot(T_0[a_{(1,j-c1)}] \oplus \\ &\quad Rot(T_0[a_{(2,j-c2)}] \oplus Rot(T_0[a_{(3,j-c3)}]))) \ . \end{aligned} \tag{2}$$

As part of the results presented within this paper we include a comparative result from a CPU AES implementation. The CPU result is based on the maximum result of the following two approaches. We have implemented a modified version of the standard rijndael fast implementation [30] effectively running in ECB mode, which reduces the API overhead of operating on a single message block at a time. Within a single function it operates directly on large message arrays reducing function call overhead and pointer manipulation, which more closely simulates the required input layout of AES running on the GPU as discussed later. The second approach is to use the built in OpenSSL [31] speed test which encrypts memory located plaintext using AES. We have been careful to only compare a single core single processor CPU with a single GPU so as to provide some form of comparison base. Both CPUs and GPUs can be scaled in different ways, for example multi core, multi way CPUs or SLI, multi GPU boards. We have used a single core 2GHz processor for the CPU tests and single GPU processor boards for GPU tests.

6 XOR Approaches

In this section we present three approaches for implementing XOR on the GPU which will be used for the AES implementations. First we discuss the common issues concerning all approaches, the data round trip to and from the card and data storage formats. There are three main choices when using OpenGL concerning texture (input/output data) storage format. They are external(cpu side) data format, external data type and internal(gpu side) data format. For AES we will use an unsigned byte format and use the swizzled(re-ordered) four component BGRA textures and suggest to the driver that it maintain the RGBA format and layout internally. Note that you can only ever suggest/hint to the driver the required format for storage. The reason for using pre-swizzled components is that internally graphics processors store 8bit components in this format, if we use RGBA externally the driver would have to swizzle these before sending the data across the bus and thus causing a performance slow down. Also we have used the OpenGL Pixel Buffer Object extension to transfer the data which facilitates DMA data transfers directly from driver memory space. We have selected to use texture sizes of 1024x1024 pixels for all data transfers in both the XOR and AES implementations, these dimensions tend to show good performance in all cases. The performance drops for transfers as the texture size is reduced and shows little gain when increased over that size [32]. Here we present the three different approaches used for performing XOR on the GPU. These approaches only concern the currently existing GPUs as the recently introduced G80 contains integer support.

Approach 1: This approach involves the use of a lookup table to perform the XOR operation on two 8-bit values. The table uses 65,536 (256x256) entries, representing the precomputed results of the XOR operation for all 8-bit values. The lookup table is stored as a texture which uses the single component GL_ALPHA external and internal format. This format is used to reduce the internal memory necessary to represent the lookup table. Two four component 1024x1024 textures are used to store the input data, each texture element (texel) holds four bytes. The bytes at corresponding locations within these textures are XORed together, thus using a sequential data access pattern across the input textures. For example the first component of the texel at location x,y in texture 1 is XORed with the first component of the texel at the same x,y location in texture 2. The approach renders a quadrilateral with dimensions 1024x1024 to match the size of the input textures. The generated fragments from the rasteriser are sent to a loaded fragment program which is designed to load in each pair of texels corresponding to the currently set texture coordinates. Each of the four pairs of bytes from the pair of texels are used in turn to execute a dependent texture lookup within the 256x256 XOR lookup texture. The result of each XOR lookup form one of the four components of the output fragment.

Approach 2: We have noted that dependent texture lookups have a severe performance penalty. One way to reduce this penalty is to make the dependent texture lookups access a reduced lookup space and thus ease the caching requirements. This approach uses a similar method to the above 256x256 8-bit lookup table, however to help reduce the size of the table we split each 8 bit input value into two 4-bit values and use a smaller precomputed 256 entry table. The issue with this approach is that there is no integer support or bitwise operators, all values read into the fragment processor are represented as floating point numbers. Thus splitting the input floating point values representing the high and low 4-bit values must be achieved using a different method.

The method requires a 16x16 entry texture with the wrap mode set to GL_REPEAT to store our precomputed XOR values. Note that all byte reads from textures within fragment programs are clamped between 0 and 1, there is no way to avoid this at present when dealing with byte values. Due to this clamping, when byte values are used as the coordinates for a dependent texture lookup into the 16x16 XOR table, they will automatically retrieve the XOR of the 4 high bits (most significant). We then multiply the original input pair of input values by 16, which when combined with the repeating nature of the lookup texture, cancels out the effect of the high bits and will retrieve the XOR of the 4 low bits. After retrieving the two 4-bit results they are recombined by multiplying the 4 high bit value by 16 and adding the low bit resultant value. It should be noted that all retreived values required multiplication by a correction factor of slightly less that 1 to avoid rounding error when used in depended texture lookups.

Approach 3: We use the native XOR found in the ROP units at the end of the rendering pipeline, i.e. the output from the fragment processors can be XORed with the values within the framebuffer. The advantage to this is that it will perform well, however the disadvantage is that the XOR operation can only be applied to the final stage of the rendering process meaning that to reuse previously XORed values a full render pass must occur. In comparison, the previous two approaches which simulate XORs within the fragment processors can immediately reuse the values for input into other operations within the fragment program. The ping-pong method must be employed when requiring the previous render pass output to be used within the next pass input. This involves making the output textures the input textures of the next rendering pass, and switching the current pass's input texture to be the output textures of the next pass.

A bi-product of the ROP stage XOR is that only a single input can be XORed to the existing results in the framebuffer. To equalise this fundamental difference, when benchmarking all three approaches we only transfer the data for one input texture in approaches 1 and 2, thus only one set of data is changing, as in approach 3. We have taken this into account when reporting the amount of bytes XORed in the results section.

Results: Table 1 shows the results of the three approaches including CPU results. The 8bit and 32bit CPU results portray the performance using bytes and integers respectively as the data units for the xor operations. The GPU results include figures for running the approaches with full data round trip and without. As one would expect the full data round trip approaches incurs large slow downs due to the transmission of the input and results across the system bus. The reason for including the results for non round trip XORs is due to the reality that when used within AES the data will not have to be transferred across the system bus after every XOR operation. We can see that the native XOR results far exceed those of the others, however it is worth bearing in mind the previously mentioned restrictions to using this approach. The native speeds as expected are close to the full rendering speeds with the additional overhead of a texture lookup and a framebuffer read per pixel per pass. Note that the theoretical pixel fill rate of the 7900GT is 7200Mpixels/s, that is equivalent to 28,800MB/s. We can also see that there is a significant increase in XOR performance when using the 4-bit lookup table over the 8-bit lookup table. This increase and the fact that the major difference between the table lookup approaches and the native approach is the execution of dependent texture reads, suggest that the lookup table approaches are memory bound.

Table 1. Results of the various XOR implementation approaches quoted in MBytes/s.

	GeForce 6600GT			GeForce 7900GT			CPU	
	8-bit	4-bit	Native	8-bit	4-bit	Native	8-bit	32-bit
W/O Round Trip	181.26	1068.0	4160	672.0	3510	12249	118.29	437.18
With Round Trip	79.61	126.7	141.0	334.83	472.7	475.4		

7 AES Approaches

As previously mentioned we use the encrypt part of the AES cipher using 128-bit key length and thus 10 rounds. We have maintained the same texture size of 1024x1024 as used within Section 6. All approaches use Pixel Buffer Objects for efficient data transfer both to and from the graphics card, this data round trip is included in all implementations to show realistic performance results. As introduced in Section 5 both forms of the optimised table lookup techniques of AES cipher implementation, as presented in the original Rijndael proposal, are implemented within each approach. All approaches use a technique called multiple render target, which involves the use of 4 output textures as output targets for the fragment programmes.

Memory Access Techniques: In general we read plaintext data from textures which have an internal format of four bytes (components) per texture element (texel). Each texel makes up a single column within the input block State. Each texel is written out to the destination framebuffer when the round or stage processing, depending on the approach used, is finished and represents the new State value of the corresponding column. Within all approaches we attempt to increase the patterns of memory access by altering the layout of the plaintext data across the input textures. We explore three different input gather techniques which include the use of multiple tables and single tables. These techniques are included in the implementation of each approach where appropriate.

- In Figure 2(a) we can see that the input data is read from four different textures at the same texture coordinate, this provides for good predictability though as Govindaraju et al. [33] points out texture memory is read in the form of blocks of data. This would mean that 4 independent texture blocks are requesting residency within the texture cache at all times. We label this technique as Multi Input in the results.

- In Figure 2(b) we have adopted a different memory gather approach reading all input plaintext from a single texture. The layout of the 16 byte blocks use four component texels one after the other in a horizontal fashion which we hope would require less active memory blocks within the texture cache at the one time. The rasterisation pattern which is responsible for handing off fragments to the fragment processor in a cache friendly order is proprietary so we can only guess at the most efficient access patterns. To reduce the overhead in calculating the 4 different gather texture coordinates within the fragment program we construct the rendered quadrilateral with multiple texture coordinates per vertex. We configure each texture coordinate set to be appropriately out of line with the rendered quadrilateral so that the interpolated coordinates generated within the rasterisation stage will automatically fall on the correct texel. This allows the rasterisation stage of the pipeline to be utilised thus incurring no computational overhead within the fragment processors. This technique is labeled as Single Input Hgather.

- The third approach shown in Figure 2(c) is similar to the previous technique, reading from a single texture, however to cater for an access pattern which suits 2 dimensional block structures better we organise the input plaintext data into a square. This has similar gather requirements to standard texture filter reads used within traditional graphics programming. The same method involving multiple texture coordinates as stated in the previous technique is also used here. This technique is labeled as Single Input Sgather.

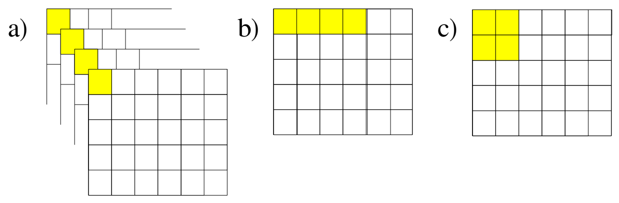

Fig. 2. Illustrations of the different gather techniques employed across the AES approaches.

The AES implementation approaches are presented here.

Approach 1: This approach is based on the 8-bit implementation of XOR as described in Section 6 and both forms of AES optimisation as described in Section 5. Each execution of the fragment program reads a full 16 byte block via 4 texels using the gather techniques described above. The other input textures used within the fragment program are the round key texture, the XOR texture and the Te (using the terminology in reference [30]) lookup textures. The round key texture is a 1D texture which contains a pre-generated schedule of round keys which is provided by the CPU part of the implementation. The appropriate texture co-ordinates for the round key are dynamically generated within the fragment program. There are either 5 or 2 1D Te lookup textures, representing the first form of the cipher optimisation lookup tables (noROT) or the second form which only involves a single lookup table (ROT). The extra lookup table is used for the last round which consists of a precomputed set of results for this round which excludes the MixColumn step. A single execution of the fragment program processes the 16 input bytes and produces the final round output of 16 bytes.

Approach 2: Based on the 4-bit version of XOR approach described in Section 6 the vast majority of implementation detail is the same as the above 8-bit AES approach. The number of XOR lookups are doubled due to both the high and low bit values being dealt with separately. The high and low bit values are only recombined at the end of each round when necessary for use as a single 8-bit Te table lookup value. This recombining could be further delayed by using 2D Te lookup tables based on 4-bit by 4-bit lookups though was deemed unnecessary as the ALU instructions are not presenting a bottleneck. This could be shown by the removal of all ALU instructions within the algorithm implementation which resulted in no performance difference.

Approach 3: This approach is based on the ROP provided XOR native implementation shown in Section 6. As scatter is not supported within fragment programs, the output is restricted to writing to a fixed location within the four active output framebuffers(textures). This restriction dictates that the input format must use the Multi Input gather technique as described above. The Single Input Gather techniques are not incorporated into the implementation of this approach. As only one XOR operation per fragment output can be executed per pass, we require five rendering passes per round of execution plus one initial clear fragments command to reset all output values to zero ready for the next round after the input and output textures have been swapped, see ping-pong above. Each stage of the optimised AES implementation equation is implemented by a different fragment program specifically written to execute the correct component based lookup within the Te textures. To save having to read in all four input textures each render pass and due to the free nature of the swizzle operation(ie. RotByte * n) we can rearranged the ROT version of the AES implementation technique to permit only a single active input texture per pass, thus reducing the active cache footprint of a single pass. Equations 3 and 4 show the first two column equations suitably rotated to facilitate a single column reference per rendering pass, note the column references are matching vertically. This in effect means that we are only referring to a single column at each stage and generating full stage output for all columns, XORing it with the appropriately rotated result. The OpenGL Vertex Buffer Object extension was employed when implementing this approach to reduce the overhead of vertex transfer due to the high number of render passes.

$$e_0 = k_0 \oplus T_0[a_{(0,0)}] \oplus Rot(T_0[a_{(1,1)}]) \oplus Rot2(T_0[a_{(2,2)}]) \oplus Rot3(T_0[a_{(3,3)}]) \quad (3)$$

$$e_1 = k_1 \oplus Rot3(T_0[a_{(3,0)}]) \oplus T_0[a_{(0,1)}] \oplus Rot(T_0[a_{(1,2)}]) \oplus Rot2(T_0[a_{(2,3)}]) \quad (4)$$

Results: Table 2 shows the results of the various implementation strategies described above running on both graphics cards mentioned in Section 1. We can see that the performance figures predictably follow the results trend presented in the XOR Approaches in Section 6. There is a consistent slight speed up when using the ROT version of the AES implementation over the noROT version. There is no appreciable difference in speed when using the different gathering techniques which suggests that the bottleneck lies with the XOR table lookups or that the sequential nature of all gather techniques do not incur conflicts or capacity misses in the first place. In general it is good practice to structure reusable memory access patterns which benefit from spacial locality of reference. These techniques can be applied when implementing future approaches using the new DirectX10 architectures where XOR bottlenecks will not exist. It is worth noting that the speeds of the CPU implementations under perform when compared to 64 bit optimised versions reported on [34].

Table 2. Results of the various AES implementation approaches quoted in MBytes/s.

Gather Technique		GeForce 6600GT			GeForce 7900GT			CPU
		8-bit	4-bit	Native	8-bit	4-bit	Native	
Multi Input	ROT	6.24	11.47	45.15	25.86	39.23	108.86	46.13
	noROT	6.11	11.19	44.89	25.71	39.01	108.55	
Single Input Sgather	ROT	6.22	11.40	N/A	26.06	39.18	N/A	
	noROT	6.11	11.22	N/A	25.92	39.12	N/A	
Single Input Hgather	ROT	6.20	11.41	N/A	25.99	39.16	N/A	
	noROT	6.15	11.30	N/A	25.69	39.08	N/A	

Figure 3 demonstrates the encryption throughput effectiveness of the GPUs studied when using different packet sizes. A packet of data is defined as a separate block of data which is delivered to the GPU in isolation and delivered back to the CPU after the entire data block is encrypted. In practice the data packet size refers to the amount of data transferred as textures across the system bus before rendering and subsequent readback happens. The figure quite clearly shows that as the packet size reduces the throughput also reduces. The causes of this are the inefficiencies in transferring multiple small data loads across the system bus which leads to an increase in the number of CPU-GPU interactions. Also in general terms, as data workloads reduce in size it becomes increasingly difficult to ensure all processors in a highly multi processor environment are kept busy, which in turn leads to difficulty in effectively leveraging the potential processing power. The implication of the noted behaviour in Figure 3 with regard to cryptography is an ineffectiveness of the GPU to assist in small data unit encryption and decryption. Applications such as IPsec rely heavily on this type of behaviour and thus in order to assist, the small packet size throughput bottlenecks would have to be significantly reduced. Applications which require bulk data encryption and decryption are thus more suited to the GPU.

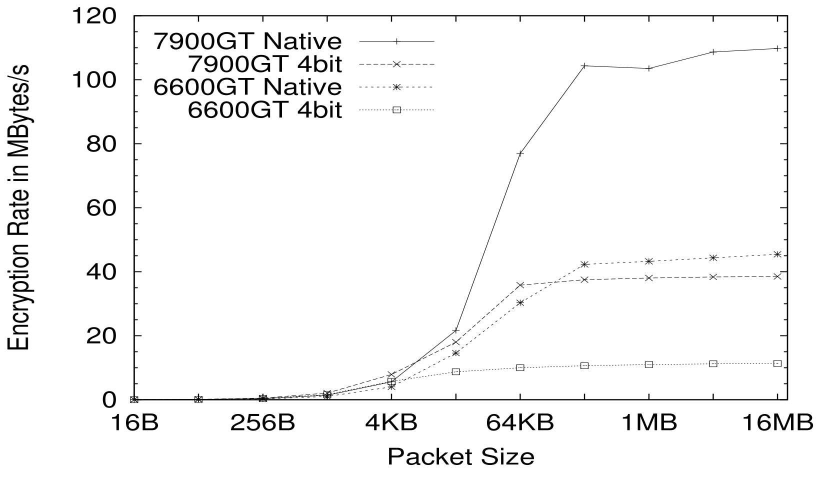

Fig. 3. Effects of packet size variation on encryption throughput.

8 GPU as an AES Co-Processor

The results shown in Section 7 are somewhat encouraging in that current GPUs can provide assistance as a cryptographic co-processor, however it was noted that the operating system reported CPU utilisation lies at 100% during all runs of the above approaches. D. Cook et al. [16] also reported the same issue for their implementations. There is little point in using the GPU as a cryptographic co-processor if it must be run in series with CPU tasks. We present a formalised investigation into this behaviour and corresponding results in this Section.

We define % CPU Idle Time as the amount of idle CPU time during the execution of a GPU task as a percentage of the total runtime of the GPU task. For example two CPU bound programs which must run in series would have a % CPU Idle Time of 0% and conversely tasks which can run perfectly in parallel have a % CPU Idle Time of 100%. % CPU Idle Time for GPU tasks can be calculated as follows: create a CPU bound task which requires a known amount of runtime, called CPU Task Time; note the length of time the GPU tasks takes on an otherwise idle CPU, called GPU Task Time; the CPU Task Time must be sufficiently longer than the GPU Task Time such that it starts first and always finishes last; run both the CPU and GPU tasks together, starting the CPU task first and note the total run time of the CPU task (which should always finish last), called the Combined Task Time; GPU Task Used CPU Time = Combined Task Time - CPU Task Time, this follows as the amount of CPU time demanded by the GPU must be the extra time the CPU task takes to finish when run in parallel with the GPU task; GPU Task Idle CPU Time = GPU Task Time - GPU Task Used CPU Time, this also follows as the amount of time the GPU task consumes that it is not running on the CPU must be in the form of idle CPU cycles; % CPU Idle Time = GPU Task Idle CPU Time / GPU Task Time * 100.

Results: In Table 3 we can see that in general the GPU performs well as a co-processor in that most of the percentages are quite high, thus leaving a high percentage of idle CPU time for other CPU tasks. There is a notable reduction in % CPU Idle Time for scenarios which demonstrate a high transfer rate. This is expected as the amount of CPU overhead will remain more or less consistent across the presented GPU tasks even though the overall transfer time has dropped: thus resulting in a high percentage of its running time occupying the CPU. Care has to be taken when interpreting these figures given that the faster AES approaches are not necessarily disadvantaged over the slower ones in terms of % CPU Idle Time, but rather there is a price to pay for the increased transfer rates. The GPU tasks which transfer at faster rates can artificially generate the same % CPU Idle Time as the slower GPU tasks by adding sleep cycles. This table clearly demonstrates that the high transfer rates come at a price.

Table 3. % CPU Idle Time based on 16MB packet sizes.

Gather Technique		GeForce 6600GT			GeForce 7900GT		
		8-bit	4-bit	Native	8-bit	4-bit	Native
Multi Input	ROT	96.69%	94.19%	86.75%	87.42%	90.61%	74.84%
	noROT	95.96%	94.10%	85.98%	88.79%	89.79%	74.57%
Single Input SGather	ROT	99.18%	96.75%	N/A	88.06%	93.54%	N/A
	noROT	98.24%	95.32%	N/A	88.65%	92.34%	N/A
Single Input HGather	ROT	98.76%	96.59%	N/A	88.70%	93.02%	N/A
	noROT	98.56%	96.46%	N/A	88.49%	93.34%	N/A

9 Conclusions

Within this paper we have presented new approaches to solving AES block cipher encryption on pre G80 GPU hardware. We have compared each approach's resulting performance to each other and to standard CPU implementations. We have achieved rates of up to 870.8Mbits/s using a Raster Operations Unit based approach and 313.84Mbits/s using a fragment processor based XOR simulation on a GeForce 7900GT. Comparing to some sources of AES performance figures for optimised implementations on standard CPUs [34], the reported GPU approaches under perform. Given that the GPU is ubiquitous and generally available by default in a highly underutilised state, it can still act to alleviate AES or potentially similar cryptographic loads from a CPU allowing it to spend time on other tasks. It was demonstrated that the GPU performs best using large packet sizes and thus suits applications which require bulk data encryption/decryption. This paper also demonstrates that the GPU can be used effectively as a co-processor contrary to the operating system reports of 100% CPU load during GPU task execution.

References

1. I. Buck, A. Lefohn, P. McCormick, J. Owens, T. Purcell, R. Strzodka, General Purpose Computation on Graphics Hardware. IEEE Visualization 05, Minneapolis, USA, October 2005. Page 33.
2. J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, D. Shippy, Introduction to the Cell multiprocessor. IBM Journal of Research and Development, Volumn 49, Number 4/5, 2005. Pages 589-604.
3. D. Cook, R. Baratto, A. Keromytis, Remotely Keyed Cryptographics Secure Remote Display Access Using (Mostly) Untrusted Hardware. ICICS05 Conference Proceedings, December 2005.
4. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard. November, 2001. http://www.itl.nist.gov/fipspubs/.
5. J. Daemen, V. Rijmen, The Rijndael Block Cipher. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/, September 1999.
6. A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography. CRC Press, 1997.
7. M. Bellare, A. Desai, E. Jokipii, P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation. 38th Annual Symposium on Foundations of Computer Science (FOCS 97), 1997.
8. T. Kohno, J. Viega, D. Whiting, CWC: A high-performance conventional authenticated encryption mode. The Fast Software Encryption Workshop, Dehli, India, February 2004. Pages 408-426.
9. O. ARB, D. Shreiner, M. Woo, J. Neider, T. Davis. OpenGL Programming Guide: The Official Guide to Learning OpenGL. Version 2. 2005.
10. A. Satoh et al, "A Compact Rijndael Hardware Architecture with S-Box Optimization". ASIACRYPT 2001, LNCS 2248, pp. 239-254, 2001.
11. J. Wolkerstorfer, E. Oswald, M. Lamberger, "An ASIC Implementation of the AES Sboxes". RSA Conference 02, San Jose, CA, February 2002.
12. A.Hodjat, D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede, A 3.84 Gbits/s AES crypto coprocessor with modes of operation in a 0.18-um CMOS Technology. Proceedings of the 15th ACM Great Lakes Symposium on VLSI 2005, pages 60–63. April, 2005.
13. M. McLoone, J. McCanny, "High Performance Single Chip FPGA Rijndael Algorithm Implementations". Workshop on Cryptographic Hardware and Embedded Systems, Paris, 2001.

14. A. Elbirt, W. Yip, B. Chetwynd, C. Paar, "An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists". IEEE Trans. of VLSI Systems, 9.4, pages.545-557, August 2001.
15. A. Hodjat and I. Verbauwhede, Minimum Area Cost for a 30 to 70 Gbits/s AES Processor, 2004 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2004), Emerging Trends in VLSI Systems Design, pages 83–88, IEEE Computer Society, 2004
16. D. Cook and J. Ioannidis and A. Keromytis and J. Luck, "CryptoGraphics: Secret Key Cryptography Using Graphics Cards". In RSA Conference, Cryptographer's Track (CT-RSA), February 2005.
17. G. Kedem and Y. Ishihara, Brute Force Attack On Unix Passwords With SIMD Computer, Proceedings of the 8th USENIX Security Symposium, Washington, D.C., USA, August 23-26, 1999.
18. M. Olano and A.Lastra, A Shading Language on Graphics Hardware: The PixelFlow Shading System, Journal of Computer Graphics 1998, Pages 159-168.
19. I. Buck, Data parallel computing on graphics hardware. Siggraph 03: Graphics Hardware Panel, San Diego, USA, 2003.
20. S. Venkatasubramanian, The graphics card as a stream computer. DIMACS Workshop on Management and Processing of Data Streams, San Diego, USA, 2003.
21. N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, Gputerasort: High performance graphics coprocessor sorting for large database management. ACM SIGMOD/PODS, Chicago, USA, 2006.
22. J. Fung, S. Mann, and C. Aimone, Openvidia: Parallel gpu computer vision, ACM Multimedia, Singapore, 2005.
23. N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, Fast and approximate stream mining of quantiles and frequencies using graphics processors. ACM SIGMOD/PODS Baltimore, Maryland, USA, 2005.
24. J. D. Owens, A survey of general-purpose computation on graphics hardware. Eurographics, Dublin, Ireland, 2005.
25. The GPGPU Resources and Forums, available online at http://www.gpgpu.org/.
26. I. Buck, Ti. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, Brook for GPUs: Stream Computing on Graphics Hardware, SIGGRAPH Las Angeles, USA, 2004.
27. I. Buck, K. Fatahalian, and P. Hanrahan, Gpubench: Evaluating gpu performance for numerical and scientifc applications. ACM Workshop on General Purpose Computing on Graphics Processors, LA, USA, 2004.
28. M. Hill and A. Smith, Evaluating associativity in cpu caches. IEEE Transactions on Computers 38, 12, Pages 1612-1630.
29. National Institute of Standards and Technology (NIST). FIPS-46-3: Data Encryption Standard, 1976. http://www.itl.nist.gov/fipspubs/.
30. V. Rijmen, A. Bosselaers, P. Barreto. Optimised ANSI C code for the Rijndael cipher, Version 3.0. December 2000. http://homes.esat.kuleuven.be/∼rijmen/rijndael/
31. OpenSSL Open Source Project, can be accessed online at http://www.openssl.org/
32. O. Harrison, J. Waldron, "Optimising Data Movement Rates for Parallel Processing Applications on Graphics Processors", 25th International Conference on Parallel and Distributed Computing and Networks, February 13-15 2007, Innsbruck, Austria.
33. N. Govindaraju, S. Larsen, J. Gray, D. Manocha, A Memory Model for Scientific Algorithms on Graphics Processors, SC06, Florida, USA, 2006.
34. H. Lipmaa, "AES/Rijndael: speed", http://www.adastral.ucl.ac.uk/∼helger/research/aes/rijndael.html