Symmetric Key Cryptography on Modern Graphics Hardware

Jason Yang and James Goodman

Advanced Micro Devices, Inc.
Graphics Product Group
{jasonc.yang,jim.goodman}@amd.com

Abstract. GPUs offer a tremendous amount of computational bandwidth that was until now largely unusable for cryptographic computations due to a lack of integer arithmetic and user-friendly programming APIs that provided direct access to the GPU's computing resources. The latest generation of GPUs, which introduces integer/binary arithmetic, has been leveraged to create several implementations of the AES and DES symmetric key algorithms. Both conventional and bitsliced implementations are described that achieve data rates on the order of 3-30 Gbps from a single AMD HD 2900 XT graphics card, yielding speedups of 6-60x over equivalent implementations on high-performance CPUs.

1 Introduction

In recent years, there has been significant interest from both academia and industry in applying commodity graphics processing units (GPUs) toward general computing problems [1]. This trend toward *general-purpose computation on GPUs* (GPGPU) is spurred by the large number of arithmetic units and the high memory bandwidth available in today's GPUs. In certain applications, where there is a high compute to memory bandwidth ratio (a.k.a., arithmetic intensity) the GPU has the potential to be orders of magnitude faster than conventional CPUs due to the parallel nature of GPUs versus CPUs, which are inherently optimized for sequential code. In addition, the computational power of GPUs is growing at a faster rate than what Moore's Law predicts for CPUs (Figure 1).

With the introduction of native integer and binary operations in the latest generation of GPUs, we believe that bulk encryption and its related applications (e.g., key searching) are ideally suited to the GPGPU programming model. In this paper we demonstrate the viability of the GPGPU programming model for implementing symmetric key ciphers on GPUs. We examine high-efficiency bitsliced implementations of the AES and DES algorithms, as well as compare conventional block-based implementations of AES on previous/current generation GPUs. We demonstrate AES and DES running on an AMD HD 2900 XT GPU to be up to 16 and 60 times faster respectively than high end CPUs.

The following section describes previous work related to implementing symmetric cryptographic algorithms on GPUs and vector-based processors. Next we

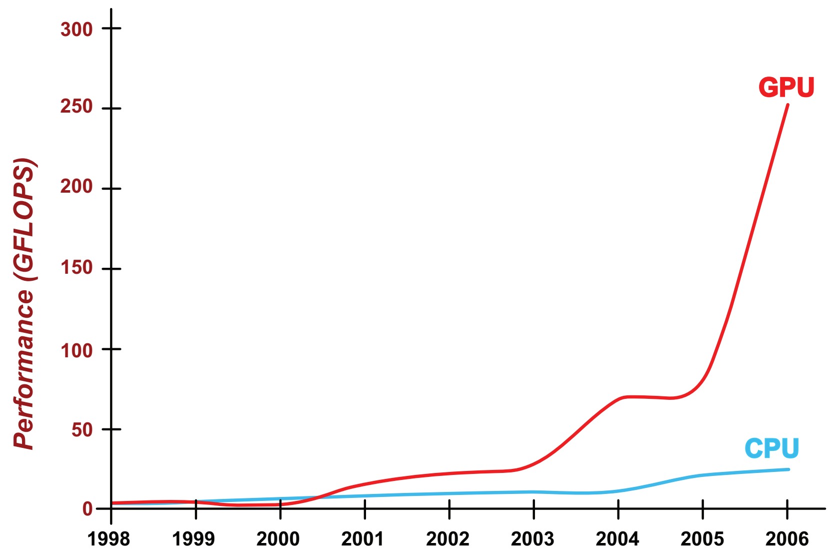

Fig. 1. GPU vs. CPU GFLOPS performance over time

describe GPU hardware architecture and programming APIs to provide context for the GPGPU programming model. Bitsliced implementations of DES and AES are then described in the context of high-performance GPGPU-accelerated key searching applications that demonstrate the potential speedup of GPUs over conventional CPUs in certain classes of problems. Lastly, a comparison of a conventional block-based implementation of AES on both the current and previous generations of GPUs is presented to illustrate the computational advantages of the latest generation of GPUs.

2 Previous Work

Cook et al. [2] were the first to investigate the feasibility of using GPUs for symmetric key encryption. Using OpenGL they implemented AES on various previous-generation GPUs. Unfortunately, the limited capability of the graphics programming model they used limited their performance and prevented them from exploiting some of the programmable features of their hardware. Instead they were forced to use a fixed-function pipeline, rely on color maps to transform bytes, and exploit a hardware XOR unit in the output-merger stage. A complete execution of AES required multiple passes through the pipeline, which significantly impacted their performance. Their experiments found that the GPU could only perform at about 2.3% of the CPU rate when both were running code optimized for their individual instruction sets. A recent OpenGL implementation [3] on a NVIDIA Geforce 8800 GTS achieves rates of almost 3 Gbps.

Vector processors have been considered for implementation of symmetric algorithms such as DES [4], and cryptography in general [5], which yielded some performance increase. Recently, Costigan and Scott [6] implemented RSA using the vector units of the Cell processor. They were able to achieve rates up to 7× faster using 6 vector units (SPU) over the onboard PowerPC unit (PPU).

3 GPGPU Programming Model

The latest generation of GPUs (e.g., Nvidia's 8000 series or AMD's HD 2000 series) has adopted the unified shader programming model pioneered by AMD in the Xbox 360's GPU [7]. In the unified shader model, all graphics functions are executed on programmable ALUs that can handle the different types of programs (i.e., shader programs) that need to be run by the different stages of the conventional graphics pipeline. The programmable nature of these ALUs can be exploited to implement non-graphics functions using a virtualized SIMD processing programming model that operates on streams of data. In this programming model, arrays of input data elements stored in memory are mapped one-to-one onto the virtualized SIMD array, which executes a shader program to generate one or more outputs that are then written back to output arrays in memory. Each instance of a shader program running on a virtualized SIMD array element is called a thread. The GPU and its components map the array of threads onto a finite pool of physical shader processors (SPs) by scheduling the available resources in the GPU such that each element of the virtual SIMD array is eventually processed, at which point additional shader programs can also be executed until the application has completed. A simplified view of the GPGPU programming model and mapping of threads to the GPUs processing resources is shown in Figure 2.

Modern GPUs are designed to be very efficient at running large numbers of threads (e.g., thousands/millions) in a manner that is transparent to the application/user. The GPU uses the large number of threads to hide memory access latencies by having the resource scheduler switch the active thread in a given SP whenever the current thread finds itself stalled waiting for a memory access to complete. Time multiplexing is also used in the SPs' ALUs to execute multiple threads concurrently and hide the latency of ALU operations via pipelining. Both of these techniques require that a thread contains a large number of calculations to improve the ability of the resource scheduler to hide the aforementioned latencies. When that condition is satisfied, the entire computational bandwidth of the GPU can be utilized to help GPGPU applications achieve performance increases on the order of $10 - 100\times$ over conventional CPUs.

DirectX [8] and OpenGL [9] are the standard programming APIs for GPUs and provide high-level languages for writing shader programs (e.g., HLSL and GLSL). However, these APIs are optimized for graphics and are difficult to use for non-graphics developers. Recently several projects have begun to try and

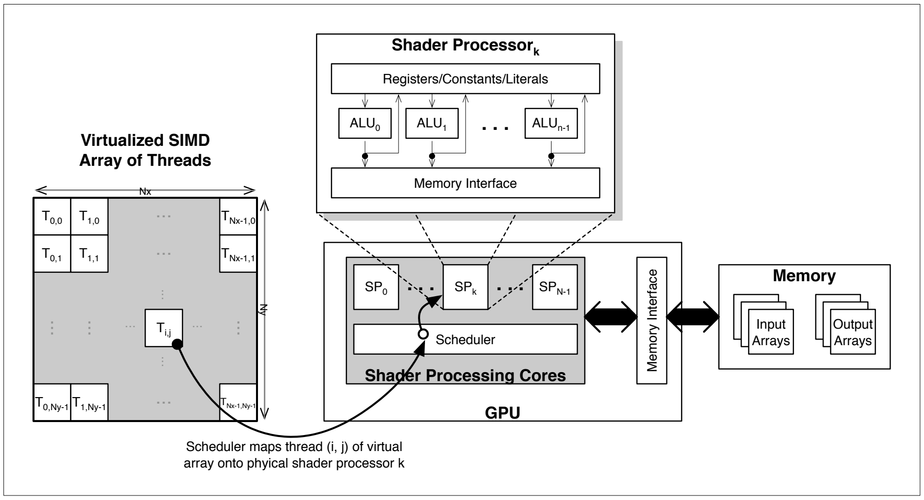

Fig. 2. Simplified view of the GPGPU programming model and thread mapping

abstract away the graphics-specific aspects of traditional GPU APIs ([10], [11], [12]). In this paper we use both DirectX and CTM [13], AMD's GPU hardware interface API, which treats the GPU as a data parallel virtual machine. CTM allows shader programs to be written in both high-level (e.g., HLSL) and low-level (e.g., native GPU ASM) languages. Writing high-level shaders is similar to writing C code, except there are additional vector data types with multiple (up to four) accessible components. See [14] for a more complete description. Our implementations written in DirectX can run on any DirectX capable hardware. The bitsliced implementations described in the following sections could also be implemented on most modern graphics hardware.

All of the experiments in this work were conducted on either an AMD Radeon X1950 XTX or an AMD Radeon HD 2900 XT GPU. The HD 2900 XT is the latest generation of AMD GPUs and uses a unified, superscalar shader processing architecture. Shader processors also share a limited number of memory fetch units, which are the physical devices that access memory. Table 1 summarizes the relevant GPU feature sets. With significantly more ALUs than memory fetch units, GPUs perform better on applications with high arithmetic intensity.

4 High-performance Bitsliced DES Key Searching Application

Bitslicing was first suggested by Biham in [15] as a means of exploiting large word widths in conventional CPUs to increase the bandwidth of software implementations of symmetric algorithms. The HD 2900 XT can be utilized in a variety of configurations due to its flexible superscalar architecture. For this

Table 1. GPU characteristics

	X1950 XTX	HD 2900 XT
# of SP Units	48	64
# of ALU Units	192	320
# of Memory Fetch Units	16	16
SP Frequency	650 MHz	750 MHz
Memory Frequency	1 GHz	825 MHz
Memory Bandwidth	64 GB/s	105.60 GB/s
Local Memory Size	1 GB	1 GB

application we utilized it as a $2 \times 64-$bit wide processor with 64 individual processing cores to implement a bitsliced implementation of DES [16] for use in a key search application implemented using AMD's CTM GPGPU programming infrastructure. The full width of the GPU (160-bits) was not used as the resulting register requirements to store the entire cipher state and key vector would limit the number of threads executing at any given time, reducing overall program performance.

The key search application partitions the key space of size 2^{56} into 2^{22} independent jobs that each check 2^{34} keys. Each job is composed of 2^{12} (64×64) individual program invocations (threads), each of which is run on a shader processor using an optimized bitsliced DES shader program written in the GPUs native assembly language. Each shader program computes 64 DES calculations in parallel, and iterates a total of 2^{16} times, for a total of 2^{22} key checks per thread. In general such a brute force searching application is of limited use, but combined with a directed, template-based approach, such as that used in popular password recovery utilities, or in conjunction with side channel techniques that are used to find a subset of the secret key bytes, it can prove to be a very potent tool capable of operating substantially faster than conventional CPU implementations.

The bitsliced DES shader program utilizes the XOR, AND, OR, and NOT instructions of the GPU to implement the necessary functions, which are primarily the eight DES S-boxes. Matthew Kwan's optimized DES S-box implementations [17] were utilized as the basis for our implementation. Modifications were made to both the data format and S-box functions to enable two S-boxes to be computed concurrently (e.g., sbox15 = sbox1 and sbox5) as a means of reducing the execution time by almost a factor of 2. Table 2 compares the performance of the conventional and parallelized S-box implementations. The even/odd round distinction is required due to the alternating write-back of the left and right cipher states in the even/odd rounds when you leave the cipher state in place to eliminate DES' right/left state swapping. The difference in instruction counts between the even/odd versions is due to the insertion of NOPs to avoid write conflicts within the ALU/register interface.

Table 2. Comparison of DES S-box instruction counts

S-box	Odd Round Instruction Count	Even Round Instruction Count	S-box	Instruction Count
sbox15	69	72	sbox1	67
sbox26	65	64	sbox2	60
sbox37	63	63	sbox3	61
sbox48	61	61	sbox4	46
Total	**258**	**260**	sbox5	66
			sbox6	61
			sbox7	61
			sbox8	58
			Total	**480**

S-box parallelization, combined with a reduction in the number of registers needed by the shader program, more than offset the fact that we are only able to use less than half of the full 160-bit width available in the shader processor for bitslicing. The net effect is approximately 2.5× increase in overall performance using the 64-bit solution with S-box parallelization compared to a full-width (i.e., 128-bit) bitsliced solution.

The resulting bitsliced implementation is shown graphically in Figure 3. The main loop consists of 16 rounds of S-box applications, along with short setup functions that mix in the necessary key bits for each round. The InitCipherState function loads IP-permuted plaintext(s) into the GPU using constants as they don't change during the shader program's execution. The CheckResult function compares the pre-IP^{-1} permuted output to a similarly formatted reference ciphertext, generating a 64-bit bitmask of each bitsliced calculation where a "1" indicates a match was found (i.e., the reference plaintext encrypted with the key corresponding to that slice generated the reference ciphertext). Note that multiple plaintexts and ciphertexts can be utilized as those values are passed in as simple parameters. When a match is found the necessary information required to reconstruct the corresponding key is written to the output array where it can be scanned by the application running on the CPU while the next job is being processed by the GPU, thereby incurring no overall result-checking performance penalty. The IncrementKey function increments the bitsliced key vector stored within the GPU using a simple bitsliced bit-serial addition on the 16 key bits that track the iteration number.

The theoretical peak bandwidth of the GPU for the bitsliced DES calculation can be determined by computing the maximum rate that can be achieved by all 64 SPs operating at their peak rate, ignoring any degradation in performance due to memory accesses and overhead:

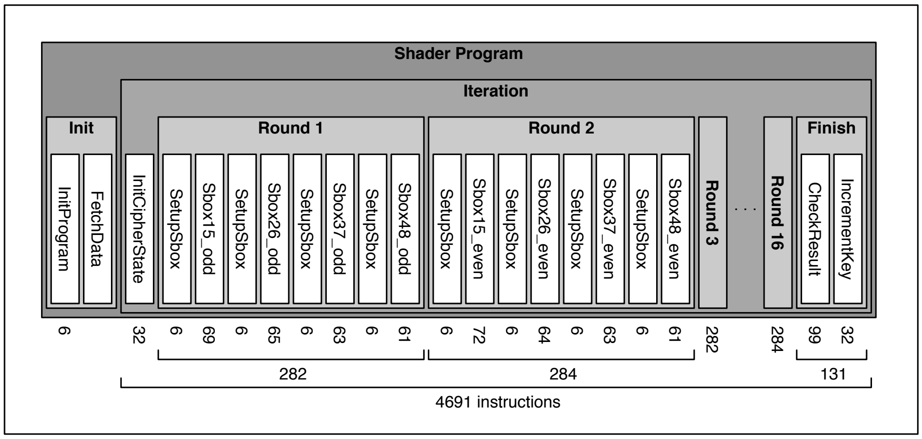

Fig. 3. Bitsliced DES implementation instruction count

$$
\begin{aligned}
PeakRate &= \frac{64\ SPs \times 750\ Minstructions/s \times 64\ blocks/iteration}{4691\ instructions/iteration} \\
&= 654.9\ Mblocks/s
\end{aligned}
$$

The execution time of the shader program is key-invariant. The performance measured on HD 2900 XT hardware is shown in Figure 4. All measurements were based on timing the program across multiple iterations for several minutes of real time execution. The implementation achieves a maximum device utilization of 83% for a maximum key checking rate of 545 Mkeys/s (i.e., encrypting 545M DES blocks per second, or 34.9 Gbps of data, though memory read/write bandwidth limitations may constrain this general case). The remaining 17% of the available performance is lost to the overhead associated with the scheduling and execution of the shader program on the GPU, along with the costs of reading/writing memory during execution.

Figure 4 also shows the performance advantage of using the HD 2900 XT compared to a comparable bitsliced DES key search program using Matthew Kwan's optimized S-boxes executing on a dual-core AMD 2.8 GHz Athlon FX-62 system. The CPU-based solution had a measured key checking rate of 9 Mkeys/s. Hence, a single-GPU solution can deliver on the order of a $19-60\times$ increase in performance over a single-CPU solution for this application.

Lastly, Figure 4 demonstrates the effect of amortizing a portion of the fixed-cost overhead of processing on the GPU across multiple iterations, indicating that over 87% of the application's maximum performance can be realized with as few as 32 iterations.

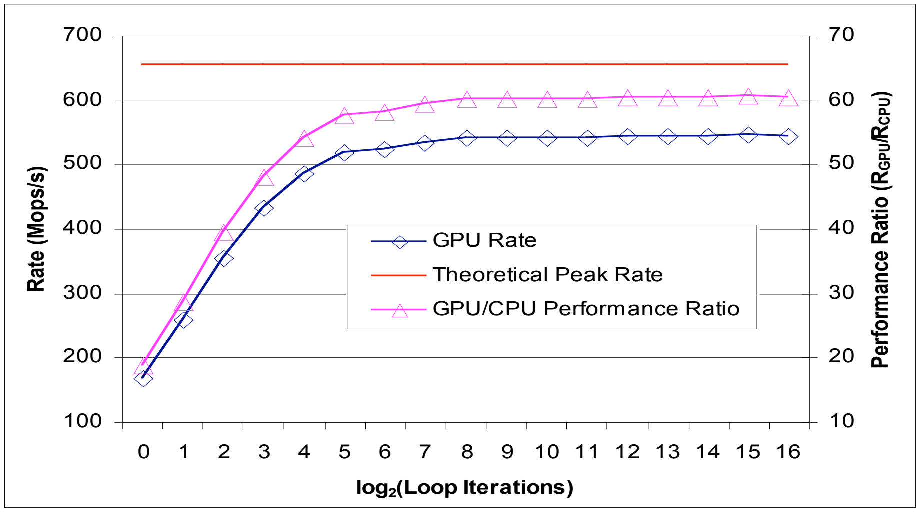

Fig. 4. Measured bitsliced DES performance

5 High-performance Bitsliced AES Key Searching Application

A more relevant algorithmic exploration was undertaken to implement an efficient bitsliced AES [18] version of the aforementioned DES key search application. The AES key search application partitions the key space of size 2^{128} into 2^{95} independent jobs that each check 2^{33} keys. Each job is composed of 2^{12} (64×64) individual threads, each of which executes an optimized bitsliced AES shader program written in the GPU's native assembly language. Each shader program computes 32 AES calculations in parallel, and iterates a total of 2^{16} times, for a total of 2^{21} key checks per thread. With such an enormous key space of 2^{128}, the only realistic use of a brute-force AES-based key search application is as a component of the aforementioned directed, template-based key searching utilities, or helping to find missing key bytes in side channel attacks. In this sort of application having an accelerated AES engine can prove very beneficial to greatly reduce the search times over conventional CPU-based solutions.

For bitsliced AES the HD 2900 XT shader processor is utilized as a $4 \times 32-$bit wide processor that processes four columns of 32 bitsliced AES state arrays in parallel. The bitsliced implementation computes the encryption key schedule on-the-fly using a transposed key array stored in the register file. The transposition is required to maximize the performance of the round key generation function. The bitsliced state and key array to register mappings are shown in Figure 5.

The bitsliced AES shader program utilizes an optimized AES ByteSub/ShiftRow implementation that computes four columns in parallel, requiring four invocations to process the entire state array (i.e., 4 ByteSub/ShiftRow operations = SubBytes/ShiftRows operation defined in [18]). The AES S-boxes were im-

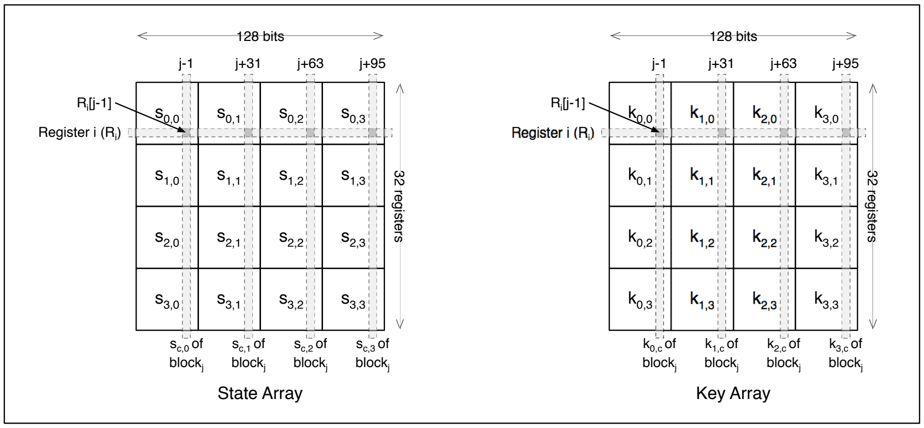

Fig. 5. Bitsliced AES register mapping of state and key arrays

plemented using the optimized normal basis composite S-box implementation described in [19] and shown in Figure 6. Additional optimizations to eliminate redundant calculations/storage were used to yield a final implementation requiring 126 instructions, which is substantially less than previously reported bitsliced AES S-box solutions (e.g., 205 instructions in [20]).

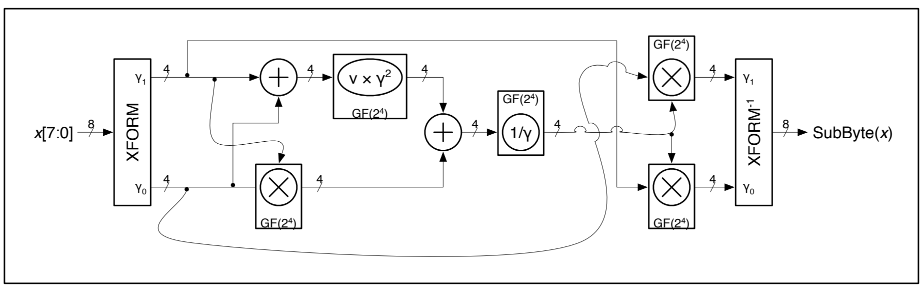

Fig. 6. Composite normal basis S-box implementation

The round key update function (Figure 7) exploits the transposed key array and optimized ByteSub/ShiftWord function to yield a 160 instruction operation. The transposition of the key array is undone when the round key is XORed into the state array using a transposed XOR operation that has no performance penalty since the transposition is done via register addressing.

The resulting bitsliced AES implementation is summarized graphically in Figure 8. The main loop adds some additional initialization as both state and key arrays need to be reset. The ByteSub/ShiftRow, UpdateRoundKey, and AddRoundKey functions have already been discussed. The MixColumns func-

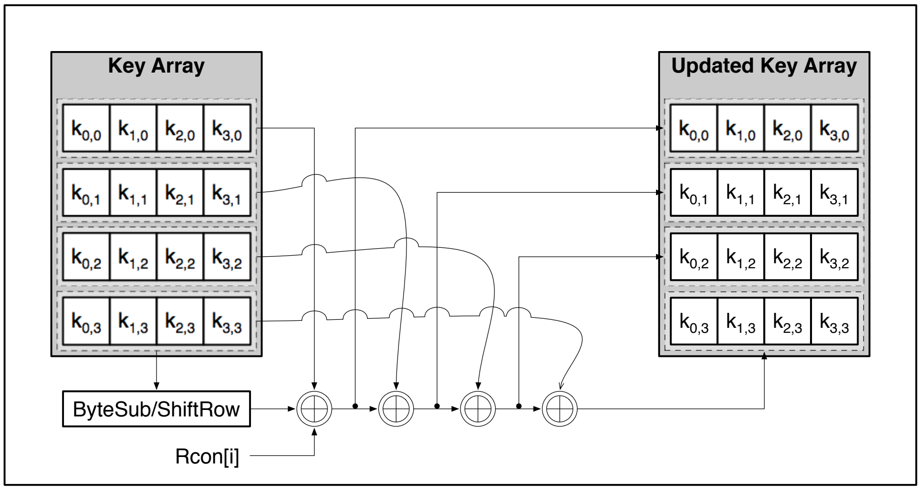

Fig. 7. Round key update function

tion processes all four columns in parallel, in-place, and in a single invocation. The CheckResult and IncrementKey functions are functionally equivalent to previously described bitsliced DES functions. As in the case of DES, arbitrary plaintexts and ciphertexts can be used, and, as previously mentioned, the key schedule is computed on-the-fly. With pre-generated keys, the performance could be increased by 23%.

The theoretical peak bandwidth of the GPU for bitsliced AES calculations can be computed as with DES using the formula:

$$\begin{aligned} PeakRate &= \frac{64\ SPs \times 750\ Minstructions/s \times 32\ blocks/iteration}{8560\ instructions/iteration} \\ &= 179.4\ Mblocks/s\ (w/key\ generation) \end{aligned}$$

The execution time of the shader is key-invariant. The performance measured on HD 2900 XT hardware is shown in Figure 9. All measurements were based on timing the program across multiple iterations for several minutes of real time execution. The implementation achieves a maximum device utilization of 81% for a maximum key checking rate of 145 Mkeys/s (i.e., encrypting 145M blocks per second, or 18.5 Gbps of data, though memory read/write bandwidth limitations may constrain the general case).

Figure 9 also compares the performance on the GPU to two previously reported software implementations ([20], [21]). The authors of [20] describe a non-bitsliced implementation on an AMD Athlon 64 3500+ CPU running @ 2.2 GHz at a rate of 2200 MHz / 170 cycles/block $\sim$ 13 Mblocks/s. The authors of [21] describe a non-bitsliced implementation on an AMD Opteron 64 CPU running @ 2.4 GHz at a rate of 2400 MHz / 254 cycles/block $\sim$ 9 Mblocks/s. Unfortu-

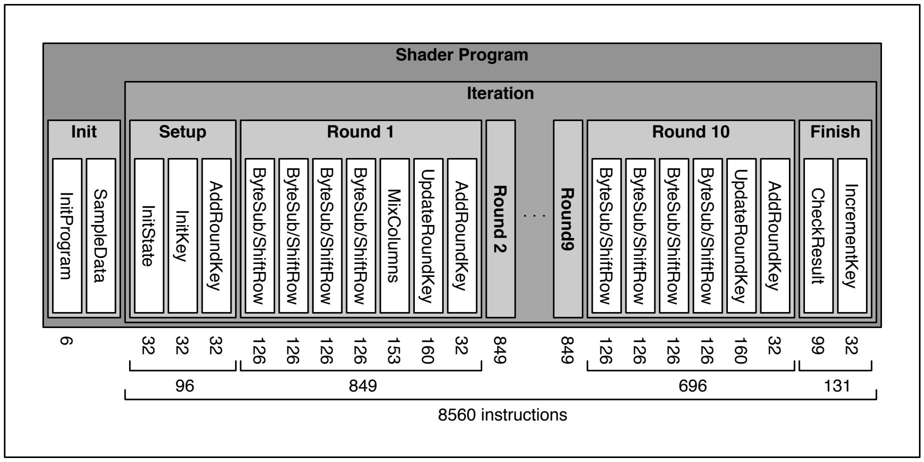

Fig. 8. Bitsliced AES implementation instruction count

nately, simple comparisons to our work aren't possible as neither implementation generates their key schedule on the fly, which is required in a key searching application. Figure 9 attempts to normalize the key generation process out of the equation by removing the key generation portion of our implementation since we don't have the necessary information to derate the results of [20] and [21]. Hence Figure 9 shows GPU implementation's results prorated by the aforementioned 23% attributed to round key generation. Hence, a single-GPU solution

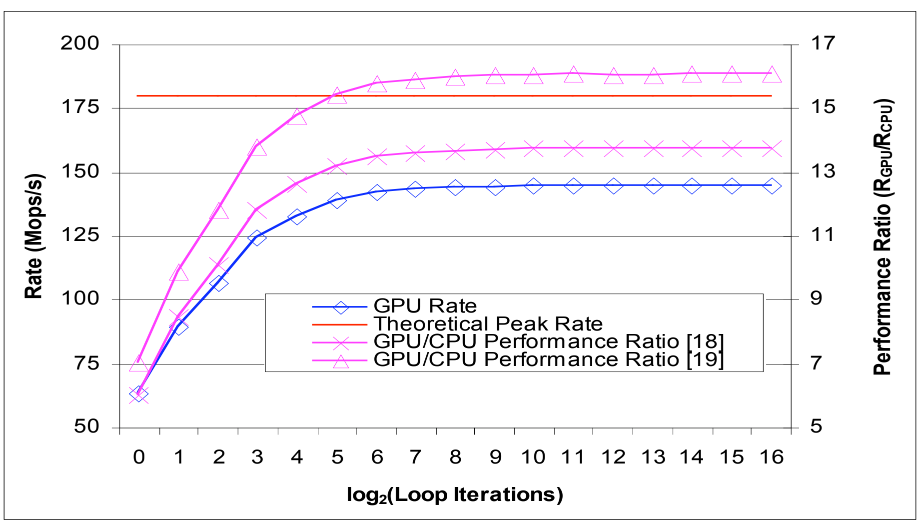

Fig. 9. Measured bitsliced AES performance

can deliver on the order of $6 - 16\times$ increase in performance over a single-CPU solution for this application.

As with the bitsliced DES implementation, Figure 9 demonstrates the amortization effect of running multiple loop iterations, indicating that over 85% of the application's maximum performance can be realized with as few as 8 iterations.

6 Conventional Block-based AES Implementation

In this section, we describe the implementation of a conventional block-based AES decryption implementation on both the previous-generation X1950 XTX GPU, which only has floating point ALU units, and the current HD 2900 XT GPU that features an enhanced instruction set with full integer support. Even with the availability of full integer support, it is still important to understand implementations on earlier GPUs because they are still used in low-cost graphics cards.

6.1 Implementation using only Floating Point Hardware

The entire 128-bit state array is transformed in parallel using four registers containing 4 bytes each stored in the transposed, unpacked format shown in Figure 10. When reading in an integer value, floating point GPU hardware normalizes the input to range from 0 to 1, which is accounted for in the shader program that implements AES.

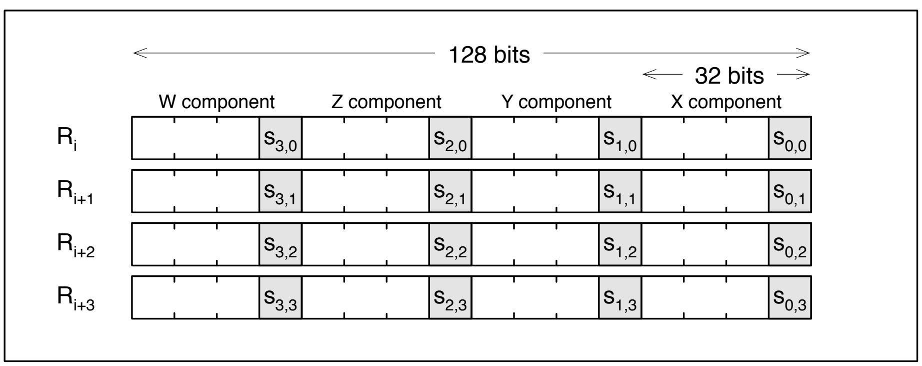

Fig. 10. AES state array register storage mapping

The internal floating-point representation introduces complications with the required XOR operation. [2] proposed using XORs in the output stage which incurs a steep penalty due to the overhead involved with issuing multiple passes through the GPU's pipeline. One alternative is to use the GPU's native instruction set to implement a XOR function at the cost of 20 instructions per

$4 \times 8-$bit row of the state array. A more economical solution is to utilize a 256×256 table-lookup in local memory to implement each 8-bit XOR operation in a single instruction. The cost of this approach is the memory latency associated with performing the lookup, but GPUs are optimized to hide these latencies by efficiently switching to other threads whenever a stall occurs due to fetching data from memory. However, a 256×256 (64 KB) lookup table is actually quite large, so a hybrid approach can also be used that processes the 8-bit XOR as two 4-bit XORs through a combination of a 16×16 (256 bytes) lookup table or ALU instructions. Table 3 compares the performance of the different XOR alternatives; however, actual performance in the full AES implementation will depend on shader instruction ordering.

Table 3. Performance of 8-bit XOR operations on the X1950 XTX

Shader Type	XORs per sec
ALU Only	6307 M
256x256 Table	778 M
16x16 Table	2980 M
Hybrid	4877 M

The GPU-based AES implementation is performed using the T-box approach described in the original Rijndael submission [22] to the AES contest:

$$\begin{aligned}
&\quad\;\; \mathtt{T_{round}[x]} \qquad\qquad \mathtt{T_{round}[x]} \qquad\qquad \mathtt{T_{round}[x]} \qquad\qquad \mathtt{T_{round}[x]} \\
\mathtt{s'_{0,c}} &= (\mathtt{0E{\cdot}Sbox[s_{0,c}]}) \wedge (\mathtt{0B{\cdot}Sbox[s_{1,c}]}) \wedge (\mathtt{0D{\cdot}Sbox[s_{2,c}]}) \wedge (\mathtt{09{\cdot}Sbox[s_{3,c}]}) \\
\mathtt{s'_{1,c}} &= (\mathtt{09{\cdot}Sbox[s_{0,c}]}) \wedge (\mathtt{0E{\cdot}Sbox[s_{1,c}]}) \wedge (\mathtt{0B{\cdot}Sbox[s_{2,c}]}) \wedge (\mathtt{0D{\cdot}Sbox[s_{3,c}]}) \\
\mathtt{s'_{2,c}} &= (\mathtt{0D{\cdot}Sbox[s_{0,c}]}) \wedge (\mathtt{09{\cdot}Sbox[s_{1,c}]}) \wedge (\mathtt{0E{\cdot}Sbox[s_{2,c}]}) \wedge (\mathtt{0B{\cdot}Sbox[s_{3,c}]}) \\
\mathtt{s'_{3,c}} &= (\mathtt{0B{\cdot}Sbox[s_{0,c}]}) \wedge (\mathtt{0D{\cdot}Sbox[s_{1,c}]}) \wedge (\mathtt{09{\cdot}Sbox[s_{2,c}]}) \wedge (\mathtt{0E{\cdot}Sbox[s_{3,c}]})
\end{aligned}$$

Using the above implementation, each column of the state array would require 4 lookups to compute the $GF(2^8)$ multiplications (each fetch can return $4 \times 8-$bit values simultaneously) and 12 lookups for computing the 8-bit XORs, assuming 1 fetch per XOR, for a total of 64 lookups per round. The number of lookups can be reduced to 24 by combining two $GF(2^8)$ multiplications and XORs into a single lookup table. Hence, every lookup of $T_{round}[x, y]$ would return a 4-tuple containing $[0E{\cdot}\, x \wedge 0B{\cdot}\, y, 09{\cdot}\, x \wedge 0E{\cdot}\, y, 0D{\cdot}\, x \wedge 09{\cdot}\, y, 0B{\cdot}\, x \wedge 0D{\cdot}\, y]$ which reduces each state array column update to 6 lookups (2 for the multiplications and 4 for the XORs), or 24 MixColumns lookups per round. With swizzling, the ability for hardware to arbitrarily access register components, only one table is required.

AddRoundKey is implemented using a similar lookup based technique that requires us to pre-process the key expansion table and XOR it with the range of 8-bit values forming a 2D lookup table that can be accessed using 16 lookups.

Every byte in every round maps to a specific entry in the key expansion, so every table access is of the form T_{keyadd}[byte_value, key_entry]. For the last round, which has no MixColumns operation, the S-Box transform is also included.

The following shader program pseudo-code processes one complete column of the round function:

```
float4 a, b, t, c0;
a = T_round[r0.w, r3.z];
b = T_round[r2.y, r1.x];
t = XOR(a, b);
c0.w = T_keyadd[t.x, round_offset];
c0.z = T_keyadd[t.y, round_offset + 1];
c0.y = T_keyadd[t.z, round_offset + 2];
c0.x = T_keyadd[t.w, round_offset + 3];
```

Assuming a single lookup per 8-bit XOR, the complete round function is 40 lookups.

When the shader program has processed all 10 rounds the 128-bit state array is written out to memory. The hardware can write four outputs simultaneously, which is used to write back the state as four, $4 \times 8-$bit values, each representing a row in the transposed state array (e.g., $s_{c,0}$, $s_{c,1}$, $s_{c,2}$, or $s_{c,3}$ in Figure 10).

The measured performance of this straightforward implementation is approximately 315 Mbps on a X1950 XTX and 380 Mbps on a HD 2900 XT. This assumes all input blocks use the same key and does not include the key expansion which can be computed on the CPU in parallel with previous GPU computations such that it can be effectively hidden in a well-balanced implementation. The performance is limited due to the number of lookups, which can be a penalty if there are not enough threads and ALU instructions to hide the associated memory access latencies. This is why performance does not scale by the number of ALU units, because both GPUs have the same number of memory fetch units. In addition, the random nature of the fetches due to the mixing properties of the AES algorithm impacts the ability of the GPU to use caching to minimize the memory access latencies of the lookups.

One possible optimization replaces the 2D round processing lookup tables with a 3D table that incorporates three $GF(2^8)$ multiplies and two XORs, as well as a 2D table that incorporates the fourth $GF(2^8)$ multiply and round key XOR. This reduces the entire round function to 24 lookups. In this mode, performance increases to 770 Mbps. However, the memory requirements are greatly increased as we now need a $256 \times 256 \times 256$ (16 MB) lookup table.

Taking advantage of latency hiding, a fully optimized shader using hybrid XORs performs at 840 Mbps on a X1950 XTX and 990 Mbps on a HD 2900 XT.

6.2 Implementation on the HD 2900 XT

AMD's HD 2900 XT allows for native integer operations and data types, as well as the ability to access data structures in memory (i.e., lookup tables) using integer values. XORs can be computed using the native XOR instruction of the GPU, so all 256×256 byte lookup tables with precomputed XORs from the previous section can be replaced with much smaller 256×4 byte tables (similar to CPU implementations) and their results summed using explicit XOR operations. Hence, the round operation shader code can be greatly simplified:

```
float4 c0, r0;
c0 = txMCol[r0.w].wzyx ∧ txMCol[r3.z].xwzy ∧
     txMCol[r2.y].yxwz ∧ txMCol[r1.x].zyxw;
r0 = c0 ∧ T_keyadd[round_offset];
```

With swizzling, only a single table is needed to represent an entire state array column update (e.g., four S-Box transforms and four $GF(2^8)$ multiplies) in one lookup.

The AddRoundKey step requires the key expansion to be stored as a separate lookup table and the XOR is performed in the shader. In the very last round, SubBytes must be performed without the MixColumns. Previously we would have to precompute this into a dedicated lookup table, but now we perform separate lookups for all the S-Box transform values and then a final AddRoundKey.

With these changes, we can achieve rates of 3.5 Gbps on the HD 2900 XT compared to an optimized bitsliced implementation on a CPU running at 1.6 Gbps [20] and the floating point versions on X1950 XTX and HD 2900 XT GPUs running at 840 Mbps and 990 Mbps respectively. This is about $2\times$ faster than a CPU and $3.5\times$ faster than the floating point implementation. This is also comparable to the performance achieved by [3] using OpenGL on a NVIDIA Geforce 8800. Although the floating point implementation runs at half the rate of the CPU, this is still considerably better than 2.3% found by [2].

7 Conclusion and Future Work

In this work we have demonstrated both that GPUs can execute symmetric key ciphers, and that that they can perform significantly faster than CPUs in certain applications. Bitsliced DES on a single HD 2900 XT was shown to operate up to 60 times faster than on a CPU, and bitsliced AES was shown to run up to 16 times faster.

We also demonstrated the advantages of the latest generation of GPUs over the previous generation. A block-based GPU implementation of AES runs $4\times$ faster on the latest generation of GPUs versus the previous generation and $2\times$ faster than a CPU version.

It should be noted that the GPU is optimized for algorithms that are parallel in nature with high arithmetic intensity. Hence, when programs must be executed serially, such as when there are dependencies between threads, then CPUs will outperform GPUs. This will be the case for certain block cipher operating modes such as CBC encryption due to the dependencies between successive blocks, unless there are a sufficient number of streams that can be processed in parallel to provide the large number of independent threads required to extract the performance in the GPU.

We believe that the entire gamut of cryptography is waiting to be explored with current and future GPU hardware. Algorithmic exploration awaits on the symmetric algorithm front with investigations of efficient implementations of other block/stream ciphers, particularly those amenable to bitsliced implementations that can leverage the large datapath width inherent in modern GPUs. In addition, the word-level integer support should be exploitable in conventional hashing algorithms to achieve significant performance increases over conventional CPUs. One particularly interesting area of potential research is finding efficient mappings of the integer support on the latest generation of GPUs to DH/RSA/ECC, and other generic integer arithmetic algorithms. With processor design trending towards multi-core, and combining CPU(s) and GPU(s) on a single die, the GPU would appear to be a good research platform for future algorithm development.

Acknowledgements

We would like to thank Justin Hensley for typesetting and proofreading. Thanks also to Arcot Preetham and Avi Bleiweiss for proofreading.

References

1. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1) (2007) 80–113
2. Cook, D., Keromytis, A.: CryptoGraphics: Exploiting Graphics Cards For Security (Advances in Information Security). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
3. Yamanouchi, T.: AES Encryption and Decryption on the GPU. In: GPU Gems 3. Addison-Wesley Professional (August 2007)
4. Seidel, E.: Preparing tomorrows cryptography: Parallel computation via multiple processors, vector processing, and multi-cored chips. (Senior Honors Project, Lawrence University)
5. Fournier, J.J.A., Moore, S.W.: A vector approach to cryptography implementation. In: DRMTICS. (2005) 277–297
6. Costigan, N., Scott, M.: Accelerating ssl using the vector processors in ibm's cell broadband engine for sony's playstation 3. Cryptology ePrint Archive, Report 2007/061 (2007) `http://eprint.iacr.org/`.
7. Doggett, M.: Xenos: Xbox 360 gpu. (Game Developers Conference - Europe 2005, http://ati.amd.com/developer/eg05-xenos-doggett-final.pdf)
8. Blythe, D.: The direct3d 10 system. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, New York, NY, USA, ACM Press (2006) 724–734
9. Segal, M., Akeley, K.: Opengl 2.1 specification. Technical report, Silicon Graphics Computer Systems, Mountain View, CA,USA (2006)
10. McCool, M., Toit, S.D.: Metaprogramming GPUs with Sh. AK Peters Ltd (2004)

11. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: stream computing on graphics hardware. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, New York, NY, USA, ACM Press (2004) 777–786
12. NVIDIA Corporation: NVIDIA CUDA Programming Guide. (2007)
13. Advanced Micro Devices: ATI CTM Technical Reference Manual. (2006)
14. Microsoft Corporation: The DirectX Software Development Kit. (2007)
15. Biham, E.: A fast new DES implementation in software. Lecture Notes in Computer Science **1267** (1997) 260–272
16. National Institute of Standards and Technology: Data Encryption Standard (DES). U.S. Department of Commerce, FIPS pub. 46. (1977)
17. Kwan, M.: Bitsliced des s-box source code. (`http://www.darkside.com.au/bitslice/index.html`)
18. National Institute of Standards and Technology: Advanced Encryption Standard (AES). U.S. Department of Commerce, FIPS pub. 197. (2001)
19. Canright, D.: A very compact rijndael s-box. Technical Report NPS-MA-04-001, (Naval Postgraduate School)
20. Matsui, M.: How far can we go on the x64 processors? In: FSE. (2006) 341–358
21. Dai, W.: Crypto++ benchmarks for amd64 processor. (`http://www.cryptopp.com/benchmarks-amd64.html`)
22. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)