AES Decryption Using Warp-Synchronous Programming

Saddam Quirem and Byeong Kil Lee
Department of Electrical and Computer Engineering
The University of Texas at San Antonio, San Antonio, Texas, USA
s_quirem@ieee.org

***Abstract*—Programming for CUDA devices presents the paradigm of warp-synchronous nature. This paper covers an implementation of an AES decryption kernel that makes use of warp-synchronicity. The operation of the various types of memories found in a CUDA device also required some analysis for a warp-synchronous implementation. The CUDA based implementation of AES256 was tested on several CUDA devices of various compute capabilities against the original single-threaded CPU implementation on various machines using encrypted messages of sizes ranging from 2MiB to 1GiB. The time taken for the GPUs to decrypt a message has varied between GPU architectures, with some achieving a 6x speedup.**

Keywords-CUDA, GPGPU, AES, Rijndael, Decryption, CBC

I. INTRODUCTION

NVIDIA GPUs organize threads into what are known as warps, a group of 32 threads. The GPU's warp scheduler is responsible for selecting which warp will be issued its next instruction [1]. The key to warp synchronous programming is the fact that the warp scheduler issues the next instruction to each thread in warp simultaneously in an SIMD fashion. This means every thread in warp will be executing the same instruction at the same time. However, when reading from and writing to memory RAW, WAR, and WAW hazards may still occur. The importance of warp-synchronous programming is due to the degradation in performance caused by threads in a warp diverging from each other. This paper will use AES decryption as an example of the benefits arising from the use of warp-synchronous programming. One approach will use warp-synchronous programming (4 threads/cipher block), and the other will look at the results of using a standard approach to parallelizing AES decryption. Performance and hardware profiles will be presented and compared for both approaches.

II. KERNEL & DEVICE MEMORY CONFIGURATION

The decryption kernel is launched with 256 threads per block for optimal multiprocessor occupancy. In order to avoid thread divergence, extra memory is allocated on the GPU to ensure that extra threads will not have to branch out of the kernel or cause memory access violations. This is done in several steps starting by selecting a transfer size; this can be hard-coded into the program or some algorithm can be used to determine an appropriate transfer size. This transfer size has to be a multiple of 16 (16B is the block size), and at least 32B in size, because an XOR operation needs to be performed with the previous cipher block. The size of the x-dimension of the grid is calculated with the following equation, where 16 is subtracted from the transfer size, which is in bytes, because the first 16 byte would be a cipher block that was decrypted in a previous kernel call. This number is divided by 4 times the block size because each thread is going to be operating on a state column composed of 4 bytes.

$$gridSize.x = \sqrt{\frac{transferSize - 16}{4 \cdot CUBLOCKSIZE}}$$

The size of the kernel grid in the y-dimension is set equal to the grid's x-dimension and incremented once or twice if it is determined the grid is not yet large enough.

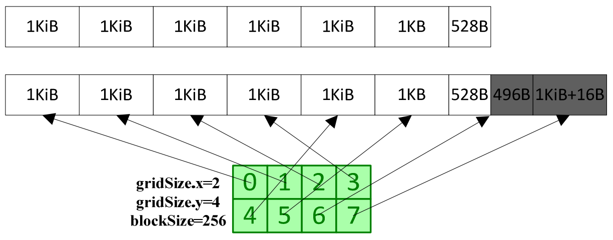

Figure 1. Given an Encrypted Message and a CUDA Block Size, CUDA Blocks And Enough Memory Allocation should be Provided.

The size of the memory allocated on the GPU is large enough to allow each thread a piece of data to independently work on. The size of the memory to be allocated on the device would be calculated as:

$$allocSize = gridSize.x \cdot gridSize.y \cdot (4 \cdot CUBLOCKSIZE) + 16$$

Figure 1 shows how much extra memory would be allocated with kernel grid containing 256 threads per block. Despite the extra memory allocated on the GPU, the ciphertext read from the input file and the useful plaintext decrypted by the kernel is all that needs to be transferred to and from the device.

III. WARP-SYNCHRONOUS DECRYPTION KERNEL

During kernel configuration, the amount of ciphertext that actual needs decryption does not need to be known. The kernel only requires the device pointer to the ciphertext buffer and the device pointer to the plaintext output buffer. Each thread operates on a 32-bit word which compromises of a single column within the 128-bit state. The other 3 columns within the state must be known when a thread performs an AES round. For this reason, all states are stored in shared memory.

Global memory accesses on devices of compute capabilities 1.x are performed in half-warp (16 threads). Due to each thread operating on a single 4B column, the groups of 4 threads handling each AES block would not fall out of sync during a global memory access. An AES round starts off with the loading of other columns. Within a group of four threads, thread 0 would be accessing columns 0, 3, 2, and 1 and shifting them by 0, 8, 16, and 24 bits respectively in that order. There is a potential for a RAW hazard occurring at this step, because the threads are reading from sections of the shared memory. However, each thread is set to first read from a column that corresponds to a thread's position in a group of 4 threads. Thread 0 reads from columns 0, 3, 2, 1 and thread 1 reads from columns 1, 0, 3, 2 in these orders, thus the RAW hazard is avoided, as shown in Figure 2. By the time a thread reads from a different column, the other threads in the warp would have finished writing to their respective columns for that round. This approach is beneficial if two threads with adjacent thread IDs are actually present in different half-warps.

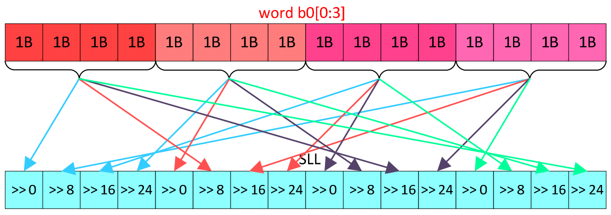

Figure 2. A Group of 4 Threads Operate on the Same.

IV. Kernel Profiles and Performance

The NVIDIA Visual Profiler was used to determine the efficiency of the decryption kernel. The profiler's outputs have shown positive results for the kernel on Fermi processors (compute capabilities 2.0/2.1) as displayed in Table 1. Similar or better results could probably be achieved on newer hardware.

Table 1. Warp-Synchronous Decryption Kernel Profile.

Property	Warp-Synchronous	Standard
Registers/Thread	14	20
Global Load Efficiency	66.6%	66.7%
Global Store Efficiency	100%	100%
Branch Divergence Overhead	0%	0%
Shared Memory Replay Overhead	0%	0%
Global Memory Replay Overhead	9.2%	0%
Local Cache Replay Overhead	0.3%	0.6%
Achieved Occupancy	85%	75.8%

The key achievement of the warp-synchronous approach is the reduced number of registers/thread, and thus occupancy. The warp-synchronous implementation generally suffered very little overhead. However, the standard 1 thread/block implementation had no replay overhead at all. Regardless, the performance results showed that the single stream implementations had roughly the same performance.

Encrypted messages used to collect results began with an encrypted message of 2MiB plus 16B for the initialization vector. The decrypted message would be exactly 2MiB. The encrypted message sizes for the test were doubled from 2MiB up to, if possible, 1GiB message sizes. Not all message sizes were tested for all the cards, because some had very limited message sizes. The Quadro NVS 295 was limited to 512MiB, while the Tesla C1060 and C2075 contained 4GiB and 6GiB of global memory respectively. There is maximum amount of global memory that may be allocated on the devices at once, which is obtained through the *cudaMemGetInfo* function, which also limits the possible size of messages that may be decrypted in a single kernel call. We compared the timings of all the CUDA devices tested against the various message sizes.

One of the greatest issues in data transfer times is that they are not related to the performance of graphics processor, but instead the bandwidth of the PCI-e bus responsible for transferring data between the device and host. Adding a larger number of shader processors, multiprocessors, or raising the clock frequencies may improve the kernel execution time, but the transfer times would be unaffected. Theoretically, even if the messages were decrypted instantaneously, the overall performance would still be limited. If the implementation of the decryption program were to be unchanged, with message transfers and kernel executions happening sequentially, then maximum performance gain of the GPU over the CPU can be calculated as such:

$$Speedup_{MAX} = T_{CPU}/(T_{H2D} + T_{D2H})$$

Against the Xeon X5670 and Core i7-2600K, the Tesla C1060 was the first device to actually attain a faster total decryption time. Table 2 shows the measured times of a 64MiB decryption for all CUDA devices, the two best performing CPUs, and the average speedups compared to the Xeon X5650. The Core i7-2600K and Xeon X5650 both had similar performance, with each outperforming the other when decrypting different message sizes. Although the CPUs used for comparison were not present on all the machines, any performance loss induced by the use of a lesser CPU is negligible.

Table 2. 64MiB Decryption Times and Speedups Relative to X5650.

Device	Decryption Time	Speedup
C1060	179.2ms	2.445x
GTX 460	157.1ms	2.864x
GTX 560 Ti	91.2ms	4.897x
C2075	114.2ms	3.963x
Xeon X5650	454.7ms	1x

V. Conclusion

As the memory models, and size, of CUDA devices continue to advance, the pitfalls of warp-synchronous programming will continue to disappear. The measured performance of the kernels has shown that efficient global memory access patterns are not of great importance when compared to the optimization the kernel. The practice of simply allocating extra memory to avoid branching out extra threads is also shown to be beneficial regardless of device architecture, as it simplifies the operation of the kernel with the only cost being larger memory allocations. The interior of the kernel was most important in demonstrating warp-synchronicity. Because shared memory access was coalesced and did not present a RAW hazard, the kernel implemented was effectively 1/4th the size of the second implementation, while also using less registers/thread. The described techniques to avoiding divergence and hazards can generally be used by other applications which may use warp-synchronicity.

Acknowledgements

This work was supported in part by the National Science Foundation under awards CCF-1063106.

References

[1] "NVIDIA CUDA C Programming Guide v. 4.2", NVIDIA Corporation. 4/5/2012.
[2] Hubert Nguyen, "GPU Gems 3". NVIDIA Corporation. 2008.
[3] Svetlin A. Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography", presented at ICSPC 2007, Dubai, UAE.
[4] Jacob T. Adriaens, *et al.* "The Case for GPGPU Spatial Multitasking", presented at HPCA 2012, New Orleans, LA.
[5] CUDABench.cu, Visual Molecular Dynamics, University of Illinois, http://www.ks.uiuc.edu/Research/vmd/doxygen/CUDABench_8cu.htm