Effects of GPU and CPU Loads on Performance of CUDA Applications

M. Bobrov[1], **R. Melton**[1], **S. Radziszowski**[2], **and M. Łukowiak**[1]
[1]Department of Computer Engineering, Rochester Institute of Technology, Rochester, New York, USA
[2]Department of Computer Science, Rochester Institute of Technology, Rochester, New York, USA

Abstract — *General purpose computing on GPUs provides a way for certain applications to benefit from a commonly available massively parallel architecture. As such deployment becomes more widespread, multiple GPU applications will have to execute on the same hardware in systems that have only one GPU. The aggregate loads of the GPU and CPU impact the performance of each application. This work investigates the effects of CPU and GPU loads on the performance of two CUDA GPU applications with significantly different CPU-GPU interaction profiles: implementations of the AES encryption and Keccak hashing algorithms. The percentage degradation in performance of these applications from CPU and GPU loads indicates dependence on the total execution time of the application, with the greatest degradation for the shortest execution times. Performance degradations as high as 22% and 36% were observed for CPU and GPU loads, respectively.*

Keywords: CUDA; GPGPU; GPU; load; performance

1 Introduction

The advent of NVIDIA's Compute Unified Device Architecture (CUDA) and ATI's FireStream Technology has shifted Graphics Processing Units (GPUs) from primarily graphics enabling devices to general purpose stream processing systems. These GPU architectures are a cost effective alternative to traditional parallel processing machines, (e.g., clusters), with comparable performance for certain applications [1]. This change ushers in a new era in computing, which allows any modern personal computer to take advantage of parallel processing capabilities previously available only in specialized systems.

For such applications, processing may occur primarily on the GPU or may be partitioned between the GPU and CPU. The first configuration will efficiently support only a certain class of applications whose computations fit the single program, multiple data (SPMD) paradigm with a sufficient ratio of computations to memory accesses. On the other hand, the second configuration with the workload partitioned between GPU and CPU provides the opportunity for a wider range of applications to benefit from GPU computing by offloading only the part of the computation that can best benefit from the GPU architecture.

If a CPU/GPU system is not dedicated to execution of an application, performance of that application will be affected by the other applications targeting the same GPU. In other words, there is the potential for additional CPU and/or additional GPU loads. As offloading tasks from the CPU to the GPU on standard desktop configurations becomes more common, the likelihood of having multiple loads from different applications increases. Such is the case for a general desktop user who will not have a dedicated GPU for non-graphics related tasks. Thus, performance in a typical system will be affected by other applications run by the user.

This research investigates the effects of additional CPU and GPU loads on CUDA performance. The Advanced Encryption Standard (AES) and Keccak hashing algorithms were selected as the test cases. CPU and GPU loads were simulated using various applications, and the performance of the encryption and hashing algorithms was recorded. These values were then used to determine the effects of CPU and GPU loads on performance.

2 CUDA

CUDA is a highly parallel computing architecture of recent NVIDIA GPUs [2]. Unlike traditional GPUs, CUDA GPUs are designed with greater focus on data processing as opposed to flow control and caching. They are capable of executing thousands of lightweight threads simultaneously with many more queued. This high degree of parallelism leads to an immense increase in potential performance such that current generation GPUs can vastly outperform contemporary CPUs in certain applications [3]. However, not all applications can realize these benefits. CUDA is based on the stream processing model, which is an extension of the SIMD (single instruction, multiple data) paradigm. This design paradigm makes CUDA optimal for performing a single program instruction many times on different data elements. The rest of this section briefly describes CUDA stream processing, and it follows the presentation in [2–4].

The CUDA architecture consists of two main components: the memory and the processing cores, which work in conjunction. Their interaction must be considered carefully when designing a CUDA application. Memory is divided into five categories: global, constant, textured, shared, and local. Each type of memory has distinct features with regard to location, caching, and access. The overall architecture is shown in Fig. 1.

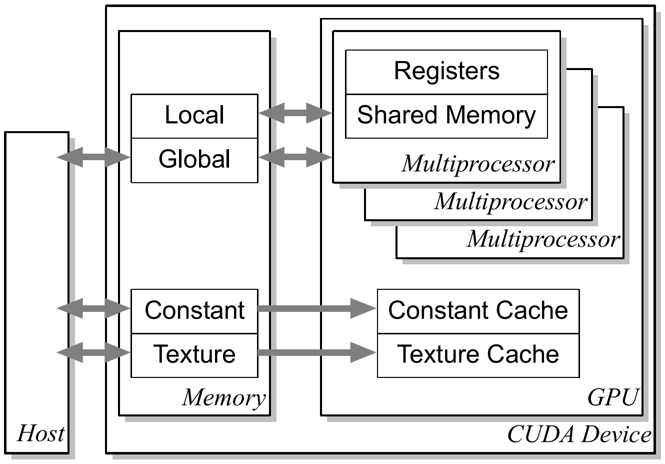


Figure 1. CUDA hardware architecture [4]

The most basic unit of execution in the CUDA architecture is the thread, and threads are organized in a hierarchy, as depicted in Fig. 2. Each thread is allocated a segment of local memory for local variables. Threads may be grouped into 1-dimensional, 2-dimensional, or 3-dimensional blocks, which consist of up to 512 threads per block. Each block has a unique section of shared memory allotted to it, which can be accessed by all threads belonging to that block. All blocks execute independently, but all threads within a block execute simultaneously. There is a mechanism to synchronize threads within a block. At the top level of the thread hierarchy, blocks are grouped into 1-dimensional or 2-dimensional grids. All grids have access to global memory, constant memory, and texture memory.

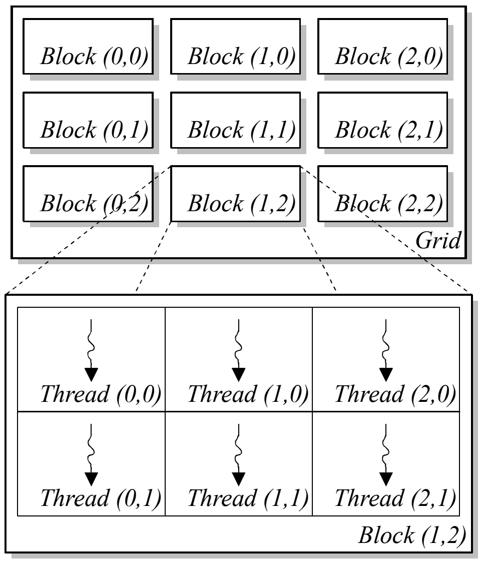


Figure 2. CUDA thread hierarchy [2]

To facilitate software design, CUDA implements numerous extensions to ANSI C. Applications are divided into two categories: code designed to execute on the host CPU and code designed to execute on the GPU. The code that is to execute on the GPU is called the kernel. Communication between the CPU and GPU is achieved through memory reads and writes.

CUDA processing consists of four steps, as indicated in Fig. 3: 1) data transfer to GPU memory, 2) CPU invocation of kernel, 3) GPU kernel execution, and 4) data transfer from GPU memory. The first step before executing a kernel is a transfer of data for GPU processing to memory on the GPU. Next, the CPU initiates kernel execution on the GPU. Once execution is complete, the CPU retrieves the processed data from the GPU. Since communication between a CPU and its peripherals is relatively slow, the process of copying data back and forth can often be a major bottleneck. Therefore, an important aspect of an efficient CUDA implementation is the ability to overlap GPU communication from/to the CPU or from/to PC memory with GPU computation.

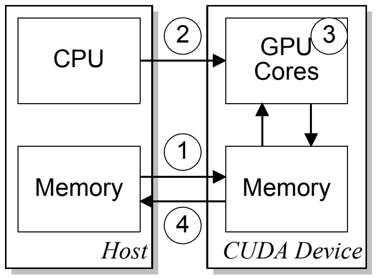


Figure 3. CUDA process flow

3 Test applications

Given the roles of both computation and communication in GPU performance, two general types of algorithms were identified for evaluation of the performance effects of other CPU and GPU loads: communication intensive and computation intensive. A GPU application that is communication intensive requires a significant amount of data transfer from/to the CPU or from/to PC memory, to the extent that execution time is dominated by this communication. In contrast, GPU computation time dominates the execution time of a GPU application that is computation intensive.

Analyzing the performance effects of additional loads on each type of application required suitable candidates. The performance results of GPU implementations of basic cryptographic algorithms [5] provided insight for selecting a test algorithm to represent each type of GPU application. AES was selected as an algorithm whose execution time is dominated more by CPU-GPU communication, and Keccak was selected as an algorithm whose execution time is dominated more by GPU processing.

3.1 AES

Advanced Encryption Standard (AES) is based on the principles of substitution-permutation networks (SP networks) [6]. To begin, the plaintext to be encrypted is divided into fixed-length blocks of data. These blocks are then converted into a 4×4 array of bytes, known as the state, as illustrated in Fig. 5, where the shading signifies grouping of bytes in columns (words).

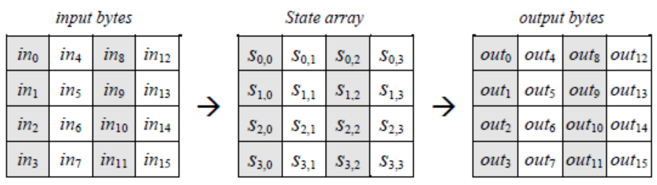

input bytes
in_0 in_4 in_8 in_{12}
in_1 in_5 in_9 in_{13}
in_2 in_6 in_{10} in_{14}
in_3 in_7 in_{11} in_{15}
State array
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
output bytes
out_0 out_4 out_8 out_{12}
out_1 out_5 out_9 out_{13}
out_2 out_6 out_{10} out_{14}
out_3 out_7 out_{11} out_{15}

Figure 5. AES state array [6]

Multiple rounds of substitutions, permutations, and key-based operations are performed on the input data to obtain the encrypted ciphertext. The substitution portion of the cipher is a simple replacement of each byte in the array with its entry in a fixed 8-bit Rijndael substitution box (S-box). Next, the permutation portion of the cipher consists of two steps: shift and mix. Shift consists of a row permutation in the form of a left-circular shift, starting with a zero shift for the top row and increasing the shift stride by one for each consecutive row. Mix is then a linear transformation of bytes forming columns of the state matrix. Finally, the round key is determined using Rijndael's key schedule, and the key is then added to the state via a bitwise exclusive or (XOR).

To increase the security and usability of block ciphers like AES, numerous modes of operation have been developed [7]. These modes extend the algorithm in order to ensure that identical message blocks encrypted at different positions in the plaintext with identical keys will not produce equal values.

The tested implementation of AES uses counter (CTR) mode, which performs the AES encryption on a counter value and XORs the result with the corresponding message block to obtain the encrypted output. CTR mode has two characteristics that are favorable for an efficient CUDA implementation. First, it preserves block-level parallelism, which represents the bulk of parallelism available in AES. Second, its encryption of a counter value instead of the plaintext provides the potential for reducing data transfers between the CPU and GPU; the final XOR can be performed either on the GPU or on the CPU.

3.2 Keccak

Keccak is a hash function based on sponge construction [8], and it is one of five finalists in the National Institute of Standards and Technology (NIST) Cryptographic Hash Algorithm Competition to select SHA-3 [9]. The sponge construction, depicted in Fig. 6, consists of two steps: absorbing and squeezing. It operates on a state, which is arranged in a 5×5 array of 64-bit lanes as shown in Fig. 7.

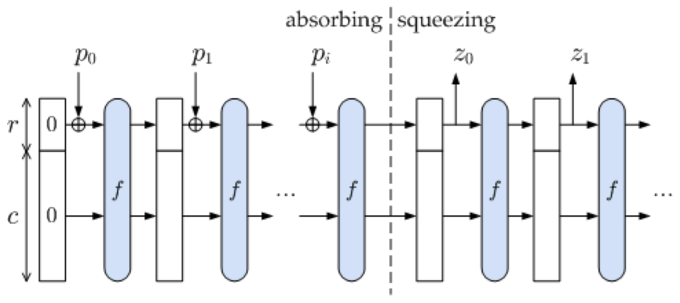


Figure 6. Sponge construction [8]

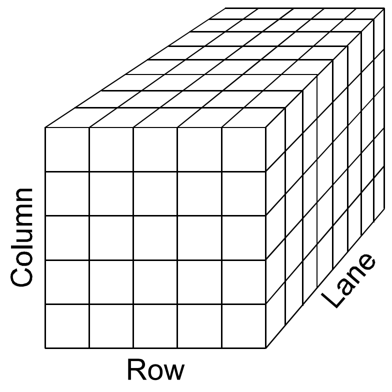


Figure 7. Keccak state [10]

Multiple rounds of squeezing and absorbing are performed on the input data to obtain the final hash. The absorption phase absorbs one r-bit block of the input message at a time by XORing the block with the state and then scrambling the result using a function f. Absorption continues until all blocks of the message have been absorbed. Likewise, the squeezing phase uses the function f to scramble the data further.

The function f used by Keccak is a permutation, which incorporates innovative security improvements [8]. The permutation performed by Keccak can be considered either as an SP network with five-bit wide S-boxes or as a combination of linear transforms followed by a very simple nonlinear transform [11]. The tested implementation of Keccak consists of 24 rounds, which is the recommendation for SHA-3 [8].

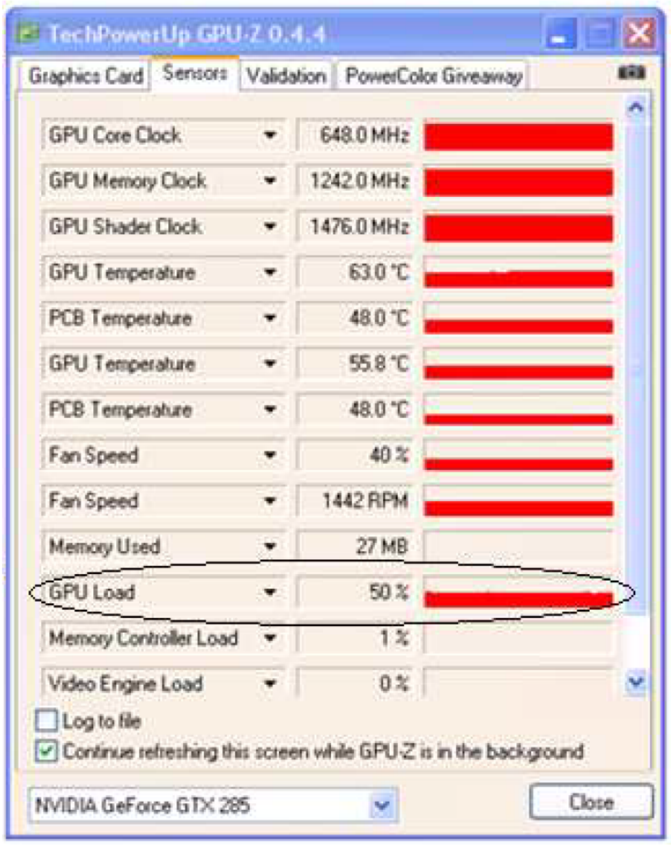
4 Test methodology

The test system utilized consists of a dual-core AMD Athlon 5600+ CPU and an NVIDIA GeForce GTX 285 GPU. The GTX 285 contains 240 processing cores and 1 GB of memory. Although this particular model is a midrange GPU, it is generally representative of CUDA enabled GPUs.

For each application, the total execution time (t_{Total}) and the GPU execution time (t_{GPU}) were measured. The total execution time is the time required to encrypt or hash a dataset, including time to transfer data between the CPU and the GPU. The GPU time consists only of the time required to compute the encryption or hashing on the GPU. For each dataset size and application variant, (e.g., AES-128, AES-192, AES-256, Keccak-224, Keccek-256, Keccak-384, and Keccak-512), these timings were measured 1000 times. The average time for each was then calculated.

To measure the effects of CPU and GPU loads additional to the encryption algorithms, CPU and GPU loads were simulated using custom applications. The CPU load application was a simple infinite loop with varying sleep times to achieve the desired 20%, 40%, and 60% loads. The GPU load application consisted of rotating a number of displayed images. Varying the number of images adjusted the GPU load to 25%, 50%, or 75%.

The loads of these applications were determined using Microsoft Perfmon for the CPU and TechPowerUp GPU-Z version 4.4 [12] for the GPU, (shown in Fig. 8). These industry proven tools provide an estimate of the average load over a given period of time. Both applications were run in parallel with the test code. They produced files containing measured CPU loads and GPU loads, respectively. These loads were recorded every second; the average load over the execution period for each test was calculated based on the values found in these files. Throughput measurements were made for loads ranging between 0% and 60% on the CPU and 0% to 75% on the GPU. The throughputs calculated under each load were then compared to the unloaded values.


TechPowerUp GPU-Z 0.4.4

GPU Core Clock	648.0 MHz
GPU Memory Clock	1242.0 MHz
GPU Shader Clock	1476.0 MHz
GPU Temperature	63.0 °C
PCB Temperature	48.0 °C
GPU Temperature	55.8 °C
PCB Temperature	48.0 °C
Fan Speed	40 %
Fan Speed	1442 RPM
Memory Used	27 MB
GPU Load	50 %
Memory Controller Load	1 %
Video Engine Load	0 %

☐ Log to file
☑ Continue refreshing this screen while GPU-Z is in the background

NVIDIA GeForce GTX 285

Figure 8. GPU load application

5 Results

To determine the effects of GPU offloading for the applications without any additional system workload, the effective CPU time (t_{CPU}) of each GPU implementation was calculated by subtracting the measured GPU execution time (t_{GPU}) from the measured total execution time (t_{Total}): $t_{\text{CPU}} = t_{\text{Total}} - t_{\text{GPU}}$. To determine the percentage of CPU time saved by offloading computation to the GPU (t_{Saved}), this effective CPU time (t_{CPU}) was then compared to the time required for the application to compute solely on the CPU without any GPU offloading of computations (t_{NoGPU}).

$$t_{\text{Saved}} = \frac{t_{\text{NoGPU}} - t_{\text{CPU}}}{t_{\text{NoGPU}}} 100\% \qquad (1)$$

Figs. 9 and 10 show the percentage of CPU time saved versus dataset size from offloading AES and Keccak, (respectively). A performance benefit from GPU offloading of these cryptographic applications was observed for datasets of 256 KB and larger. The best improvement was in AES-256, for which GPU offloading reduced CPU time by 60%. All implementations realized a time savings of at least 20% for datasets of 256 KB and larger, and the time savings increased with dataset size.

The remainder of this section describes performance results for the test applications running in a system with additional workloads. First the results of additional CPU loads are given. Next the effects of additional GPU loads are presented.

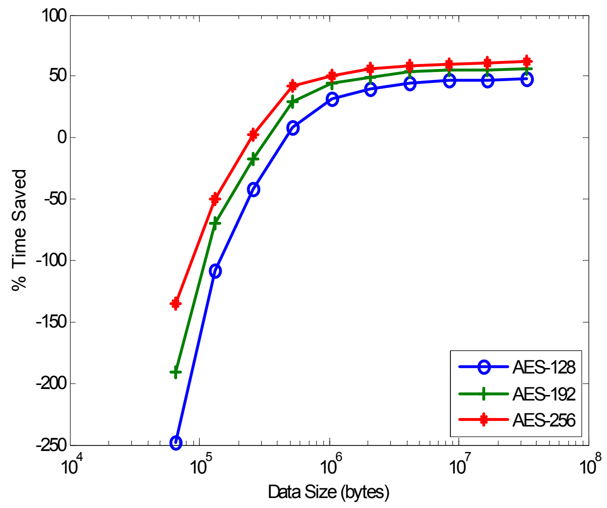


Figure 9. Offloading effects for AES

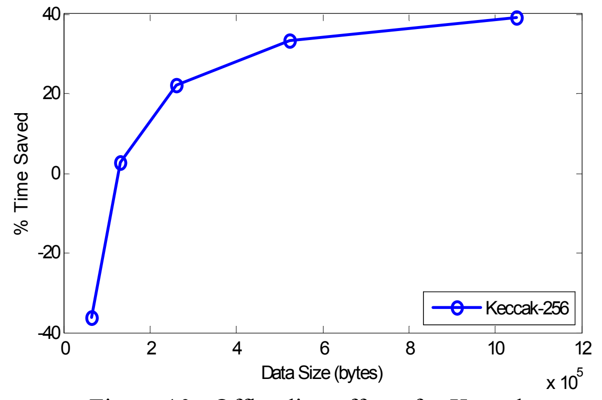


Figure 10. Offloading effects for Keccak

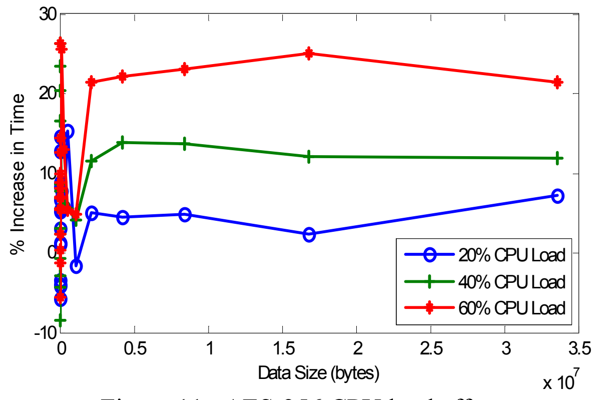


Figure 11. AES-256 CPU load effects

5.1 CPU load effects

Fig. 11 shows the effects of CPU load for AES-256. AES-128 and AES-192 performed similarly. CPU loading effects for Keccak-512 are illustrated in Fig. 12. For dataset sizes over 1 KB, the total computation time for Keccak is significantly longer than for AES.

For AES with its much lower total time, a CPU load may have a significant negative effect. With a 20% load, datasets larger than 1 MB experienced an average performance degradation of 4%. Average degradations of 12% and 22% are experienced with 40% and 60% loads, respectively, for datasets larger than 1 MB. The precise effects on datasets smaller than 1 MB are difficult to measure as they are generally encrypted quite fast and are prone to experience large percentage increases and decreases with small variations in performance. However, the effects on smaller datasets are generally minimal in terms of time. Thus, the effects of CPU loads are expected to increase as total time of the algorithm decreases.

In contrast, for the higher total time of Keccak, a CPU load has a negligible impact because the majority of the total time is from GPU processing. CPU loading effects for Keccak-512 are illustrated in Fig. 12. The bottom graph shows results for datasets larger than 128 KB, which are not distinguishable in the top graph. No degradation in execution time greater than 1.5% was measured for files larger than 32 KB. Again, the effects on datasets smaller than 32KB are difficult to measure but are minimal in terms of time.

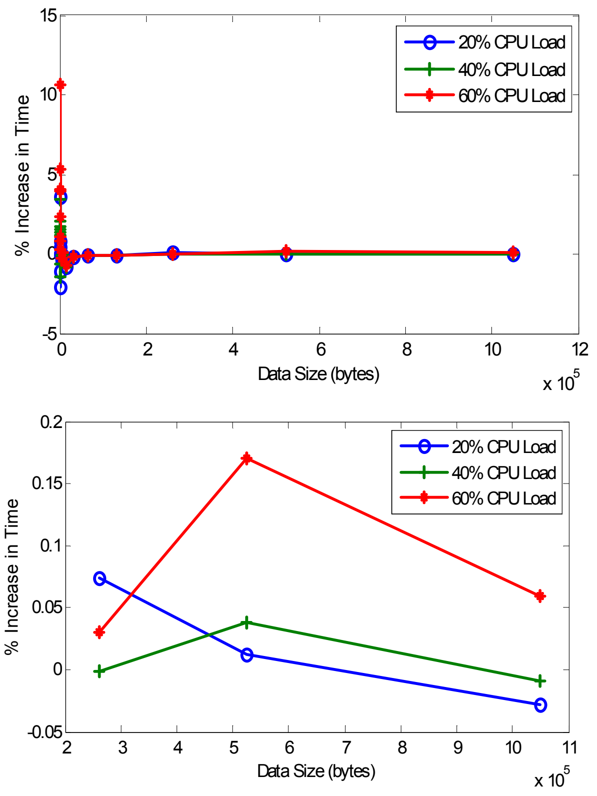


Figure 12. Keccak-512 CPU load effects

5.2 GPU load effects

Like the CPU load, the effects of introducing a GPU load are highly dependent on the execution time of the algorithm. However, the GPU load effects are quite different in nature from the CPU load effects. A GPU load consumes some portion of the resources available on the GPU and therefore makes those resources unavailable for a test application. Furthermore, a typical application producing the GPU load would also be expected to transfer data between the CPU and GPU. This additional data transfer requirement increases the load on the already slow PCI Express bus.

The primary effect of GPU utilization is on total time, which is shown in Fig. 13 for AES-256. The corresponding graph for Keccak-512 is not included here since the wide variation in magnitude among the data points causes them to appear compressed along the axes when plotted on the same scale. Instead, the top graph in Fig. 14 plots dataset sizes smaller than 8 KB, and the bottom graph shows 8 KB and larger sizes. Since GPU utilization affects both the GPU and CPU, it produces a greater increase in total execution time than does a purely CPU load. The overall trend, however, is similar to that of CPU load effects.

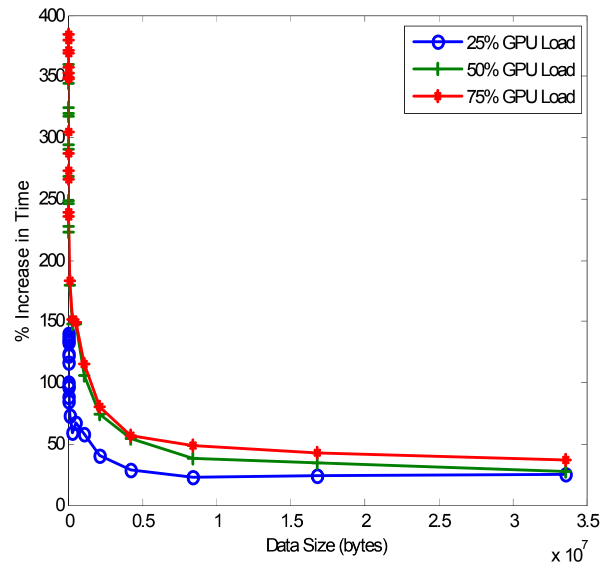
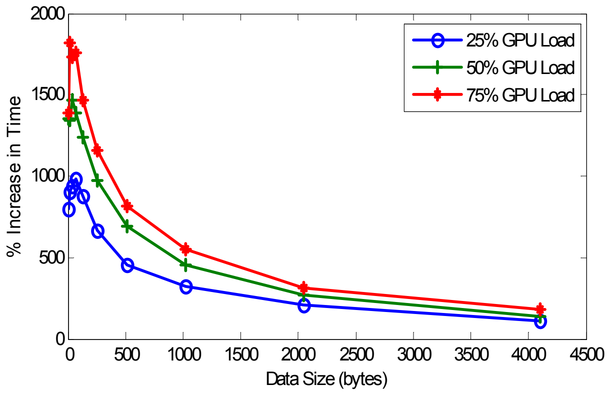


Figure 13. AES-256 GPU load effects on total time



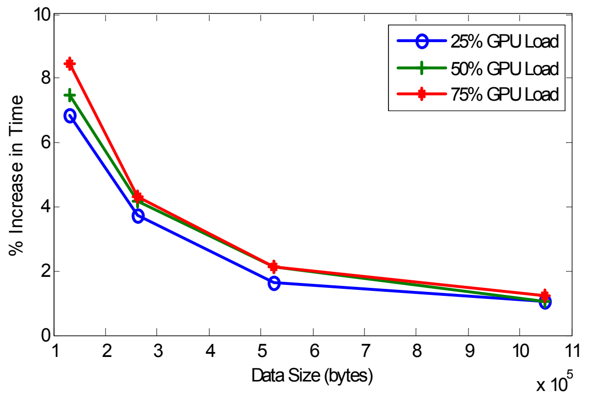


Figure 14. Keccak-512 GPU load effects on total time

As observed with CPU loading, the GPU loading results are highly dependent on total time of the algorithm. With slower algorithms such as Keccak-512, as the dataset size increases, the increase in time approaches 2%. These results likely indicate that the GPU resources were underutilized by the Keccak implementation and thus are available to support additional computation. On the other hand, for faster algorithms like AES, as the dataset size increases, the increase in time is much larger. In the case of AES-256, the increase in time approaches 25%, 27%, and 36% for loads of 25%, 50%, and 75%, respectively. Smaller datasets are not capable of masking the effects of the load well and thus are more affected with greatly increased execution times up to 2000%.

The effects of GPU utilization on GPU time are less noticeable than those on total time, as seen in Fig. 15 for Keccak-512 on datasets smaller than 8 KB. (Again, a single graph of all data points is not included because the data points appear compressed along the axes. The general trend is similar to Fig. 14, and the percent increase in time is less than 0.2% for datasets of 8KB and larger.) Smaller datasets are affected more since they are hashed in very short times, which do not mask the overhead as well as larger datasets. They experience an increase in GPU time of 6–11% for Keccak-512. As the size of the dataset increases, the overhead is better masked, decreasing the percentage to 0–1% beyond 1 MB for AES-256 and beyond 32 KB for Keccak-512.

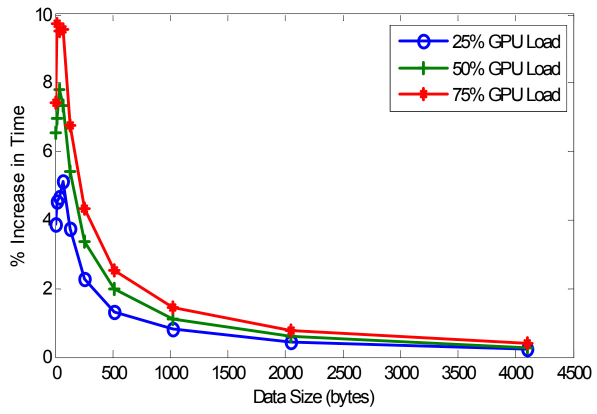


Figure 15. Keccak-512 GPU load effects on GPU time

6 Conclusions

To simulate a typical system environment, various CPU and GPU loads were introduced, and their effects on encryption and hashing performance using GPU co-processing were measured. CPU loads were found to have no effect on GPU time, but they did increase the total execution time. AES-256 experienced the largest increase by as much as 22% total execution time, but Keccak-512 saw minimal increases in total time because the CPU load effect was masked by the GPU execution time. GPU loads had similar effects on total time, although to a greater degree than CPU loads. The total time of AES-128 increased as much as 36%, whereas Keccak experienced minimal increases in total time.

For GPU processing, the effects of additional CPU loads are highly dependent on the total time of the algorithm. For such applications, the primary requirements of the CPU are for transferring data and initializing the kernel. Consequently, adding a CPU load has no effect on GPU processing time. However, total time can be adversely affected.

Similarly to CPU loads, the effects of introducing additional GPU loads are highly dependent on the total execution time of the algorithm; however, the effects are quite different in nature. A GPU load consumes some portion of the resources available on the GPU, which become unavailable for the GPU application. In addition to GPU computation, a typical GPU load also requires data transfer between the CPU and GPU. This extra data transfer requirement increases the load on the PCI Express bus, which increases total execution time.

As GPU processing for non-graphics applications becomes more common, such applications will certainly be deployed on platforms, such a general desktop computer, that have CPU and/or GPU loads in addition to the application. This investigation is a first step toward characterizing the effects of additional CPU and GPU loads on the performance of a GPU application. In this work, the effects of these additional types of loads have been considered independently of each other. Further investigation should evaluate both types of additional loads present simultaneously, as one would expect on a typical desktop system. Also, these effects have been evaluated only relative to the performance of two GPU encryption algorithms. Testing with other GPU applications is needed to see if they experience the same effects observed in this investigation. Also, performance effects on a variety of GPU applications with varying communication versus computation profiles need to be evaluated.

7 Acknowledgements

The authors wish to thank their colleagues at the Rochester Institute of Technology who contributed to this work: Dr. Andreas Savakis for equipment support from the Computer Engineering Real-time Vision and Image Processing Lab, and Dr. Muhammad Shaaban for helpful comments on drafts of this paper.

8 References

[1] David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Burlington, MA: Morgan Kaufmann Publishers, 2010.

[2] NVIDIA Corporation, "NVIDIA CUDA Programming Guide, Version 2.3.1," August 29, 2009, `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`.

[3] NVIDIA Corporation, "CUDA Zone," [July 2010] `http://www.nvidia.com/cuda`.

[4] NVIDIA Corporation, "NVIDIA CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3," July 2009, `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`.

[5] Max Bobrov, "Cryptographic Algorithm Acceleration Using CUDA Enabled GPUs in Typical System Configurations," master's thesis, Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY, August 2010.

[6] National Institute of Standards and Technology, "Advanced Encryption Standard (FIPS-197)," 2001.

[7] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation," NIST, Gaithersburg, MD, Special Publication 800-38A, 2001.

[8] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Keccak Sponge Function Family Main Document, Version 2.1," June 19, 2010, `http://keccak.noekeon.org/Keccak-main-2.1.pdf`.

[9] National Institute of Standards and Technology, "Cryptographic Hash Algorithm Competition," December 13, 2010, `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[10] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Keccak Specifications, Version 2," September 10, 2009, `http://keccak.noekeon.org/Keccak-specifications-2.pdf`.

[11] Meltem Sönmez Turan, Ray Perlner, Lawrence E. Bassham, William Burr, Donghoon Chang, Shu-jen Chang, Morris J. Dworkin, John M. Kelsey, Souradyuti Paul, and Rene Peralta, "Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithmic Competition," NIST, Gaithersburg, MD, Interagency Report 7764, February 2011.

[12] techPowerUp. "GPU-Z," July 2010, `http://www.techpowerup.com/gpuz/`.