Hardware Acceleration of Secure Hashing (HASH)

Christophe Brown
Columbia University
New York, NY
cb3534@columbia.edu
christophejbrown@gmail.com

***Abstract*—Society operates in an information age of big data. The proliferation of sensors, networks, and mobile devices have created enormous amounts of data. In many cases, this data includes information that is owned by private companies, government agencies, or individuals. With increasing amounts of sensitive data, a greater interest arises in securing and encrypting the information to deter cyber attacks. Additionally, innovations in computer hardware have produced graphics processing units (GPUs) used to accelerate workflows that can be parallelized. In this paper, the potential for performance boosts in cryptography will be explored by using GPU processing to accelerate the SHA-256 hash algorithm. Preliminary results show that GPU-related speedup may be possible at scale, yet proves to be difficult in the experimental configuration detailed below.**

***Index Terms*—Graphics Processing Unit (GPU), Central Processing Unit (CPU), PyCUDA, Parallelism, Thread, hash, SHA, encryption, hardware acceleration**

Overview

SHA-2 is a family of cryptography functions created by the United States National Security Agency at the turn of the 21st century. Although no longer used by the government, it is still widely popular for basic authentication systems, personal or private records, and the blockchain industry. They're designed to serve as a one-way function to convert text of arbitrary length to a string of 256 bits. This is accomplished through converting the string into ASCII binary values and performing bit-wise operations, modular additions, and compression functions. The algorithm as a whole relies on many sequential processes, namely iteratively populating array indices and performing the aforementioned mathematical and logical operations on them. The expansion function can be seen in Fig. 1 from [10] and the compression function of the SHA-2 family can be seen in Fig. 2 from [4].

Problem

The array operations mentioned above are computed in constant space, yet with more input data, the the array operations occur more times. This is because for SHA-256, the input data is padded with zeros until its length in bits is a multiple of 512. The input string is broken into 512-bit "chunks." Each chunk is split into 16 array elements, each being 32 bits in length. Afterwards, more calculations "expand" the 16 elements into 64 elements, and the main compression function iteratively produces the hash string.

The issue this paper addresses is the longer run times associated with processing larger data. While hashing is only one way of obscuring information (encryption is another

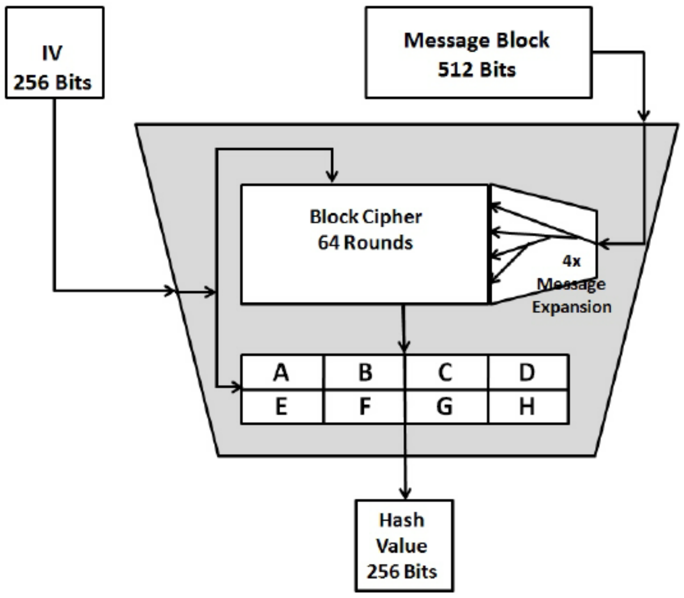


Fig. 1. SHA-256 algorithm visualized with expansion function.

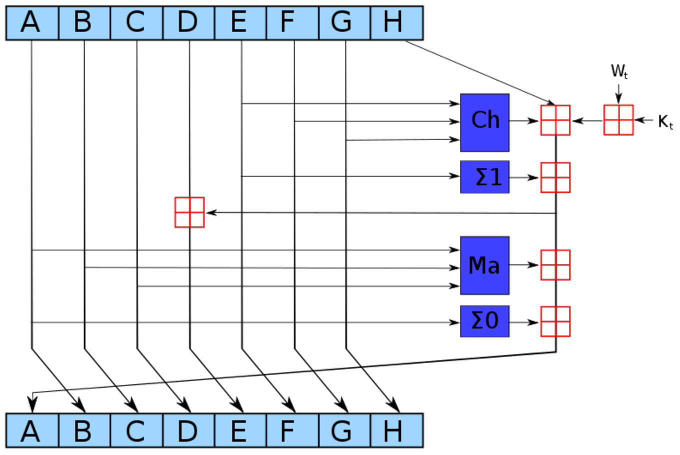


Fig. 2. Main compression function for the SHA-2 family.

important topic), approaches to reduce the time it takes to process sensitive data could introduce new security algorithms and workflows applicable security in big data.

A. *Prior Work*

[2] assesses the performance of 4 different algorithms, citing that hashing algorithms MD5 and SHA-1 will remain inferior unless they can become faster than modern algorithms like SHA-256 and SHA-512. The experiment performs each algorithm on identical input strings, and analyzes the performance with increasing string length and varying cache resources. The study concludes that with more data (longer strings) hashed with greater complexity (i.e. more reliability, SHA-256 and SHA-512), more time is taken to compute the hashes. This observation was drawn by varying string lengths less than 100 characters. For organizations and companies that operate with data in the gigabyte and terabyte range, there could be a vested interest in speeding up hash computations.

I. Description

The experiment in this paper proposes adding parallelism to the splitting of the 512-bit array into 16 elements to achieve speedup. To test this, a custom implementation of the SHA-256 algorithm was written in Python 2.7, alongside the NVIDIA CUDA computer kernel written in C. The code was executed on a remote ssh server with an NVIDIA GK110BGL [Tesla K40c] GPU clocked at 33MHz.

A. *Objectives and Technical Challenges*

Successful work in this experiment would demonstrate a noticeable change in compute time when running some of this algorithm on a GPU. Parallelism is introduced only in a portion of the algorithm, so extraordinary boosts are not expected. The SHA-256 algorithm consists of several sequential operations (operating on fixed-size arrays), which from a distance would appear to be parallelizable. However, with a closer inspection of the algorithm one will find that there are dependencies between iterations of the array, such that what is computed on an iteration *i-1* is needed for the computation on iteration *i*. This introduces a challenge and a trade off. One must consider the value in using GPUs for performance boosts provided that only a portion of the high-computational workflow benefits in this experiment.

This is not to suggest that other areas cannot be parallelized, however. The bit-wise operations performed on each compression iteration can, in theory, be parallelized. For example, the SHA-256 compression function computes a *Majority* and *Choose* function [3] on each loop iteration, and the calculations are independent, and therefore could be pipelined for the entire algorithm. Two important factors should be considered at this point. The raw speed of bit-wise operations are fast and the transfer of data from the host to the device incurs delays. This experiment does not test for GPU speed up of bit-wise calculations so results are therefore inconclusive regarding the benefits of their parallelization.

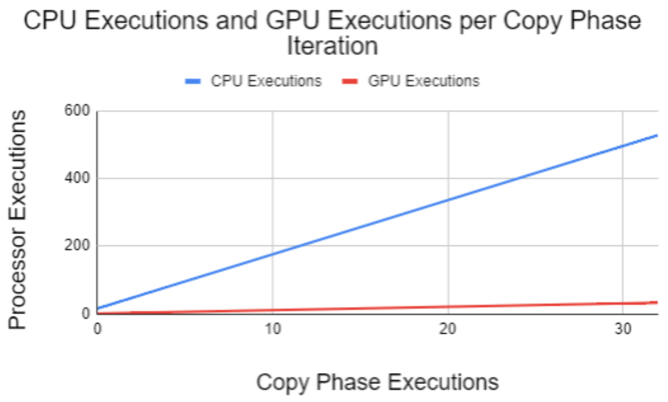


Fig. 3. For very large data where the copy phase executes many times, the GPU is expected to have a lower utilization time over the CPU.

B. *Problem Formulation and Design*

There are two metrics used to assess the impact of the GPU in this experiment. The first being the speedup, or lack thereof, on the single sweep of the copy phase. The second being the overall execution time of the algorithm. An important distinguishment should be made between the two.

The speed of the system's CPU was shown to be fast enough to write all 16 elements in the same time as a GPU can. Several reasons explain this. First, clock speed of modern CPUs are as much as an order of magnitude faster than modern GPUs, so it makes sense that a CPU can write 10 elements sequentially in the time it takes the GPU to write 10 elements in parallel, as it is 10 times as fast. Second, GPU computation requires a transfer of memory away from the core. While imperceivable in real time, the delays experienced in this transfer compound for larger data. It is expected to suffer a loss here.

The overall execution time is arguably a more important metric, as the job of hashing is not done until a hash string is produced. An intermediate value produced, such as the array of 16 elements from the copy phase is not useful for anything beyond the next phase of the algorithm. Fig 3 shows that, for each execution of the copy phase, the GPU must perform one computation, whereas the CPU must perform 16 computations. This poses the hypothesis that a GPU may yield throughput improvements for very large data.

II. Software Design

Many implementations of the SHA-2 algorithms exist. A custom implementation was used for this experiment to be readily compatible with NVIDIA CUDA. The source code can be found in the references in [1]. The pseudocode from [4] and [5] were used to create the implementation and tested against publicly-available SHA-256 hash generators. The software algorithm was written such that it would be executed in segments so that parellelization could be explored in different segments, of which include the *preprocess*, *copy*, *extension*, and *compression* phases. Pre-processing simply converts the input string into ASCII in binary values, and pads the string of bits with zeros until its length is divisible by 512. The copy phase takes each chunk of 512 bits and breaks them into an array of 16 elements of 32 bits each. Extension expands the each array of 16 elements into 64 elements. The compression phase iterates over all the data and compresses it into a 256-bit value, often represented in hexidecimal. The source code also includes unit tests, which are useful for development as well as ensuring the CUDA implementation works properly.

The CUDA kernel is assigned a 16x32 thread block. Mentioned earlier, every "chunk" of 512 bits is split and copied into 16 segments of 32 bits each. So each thread in the y-dimension indexes an array element while each each thread in the x-dimension indexes a bit in the element. This produces a threading formula:

$$bit = 32 * ty + tx \tag{1}$$

where `tx` is the CUDA C `threadIdx.x` and `ty` is the `threadIdx.y`. An early consideration was to use threading for only array indexing, but using threading in two dimensions removes the need for writing a `for` loop inside the kernel to write each array element's bit iteratively. It is important to note that additional blocks on the grid could be used, however as mentioned earlier, the copy phase executes in constant space regardless of the data size, so a 16x32-thread block will suffice in this experiment.

Another initial goal was to parallelize the extension phase of the algorithm. After the copy phase, bit-wise operations on the 16-element array expand the array into 64 elements. This process of the calculation happens on the same core 512 bits that were broken up to begin with, so one's intuition may suggest that it can be done in another GPU kernel. Late into development, this turned out to be false. The expansion uses backwards indexing in its calculations. For example, let `w` be an index in the array from elements 15 to 64. The calculation for the value stored at `w` is dependent on the values that are already written at `w-2`, `w-7`, `w-15`, and `w-16`. This would not be a problem for elements 16 to 31 (their dependencies were written during the copy phase), but elements at index 32 and beyond cannot happen without all prior elements being written. A more complex design could yield an implementation with performance varying between full parallelism and no parallelism. Such a design was not explored.

III. Results

Figs. 4 and 5 show the times taken for full hash compute and copy phase execution time, respectively. With the available compute resources, up to 10 megabytes of data could be processed for this experiment. Marginal differences can be observed in the copy times, CPU execution is dominant for small file sizes, with GPU incentives beginning to show at the 1 megabyte range. Both CPU hash time and GPU hash time experience exponential increases in compute time.

I shows ratios of the GPU execution time to the CPU execution time. "Small" refers to the brief string "the quick brown fox jumped over the lazy dog". While the GPU was expected

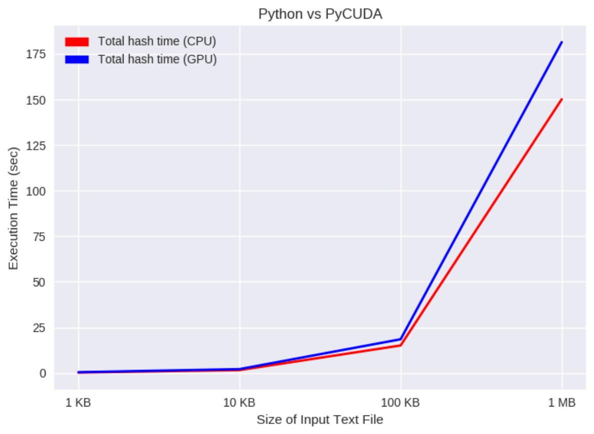


Fig. 4. Elapsed time for the GPU and non-GPU execution versus input file size.

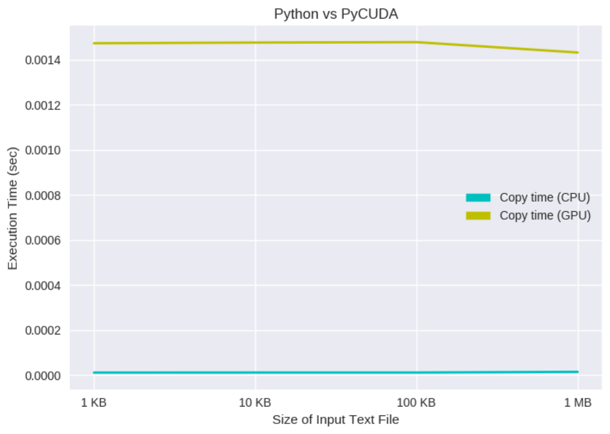


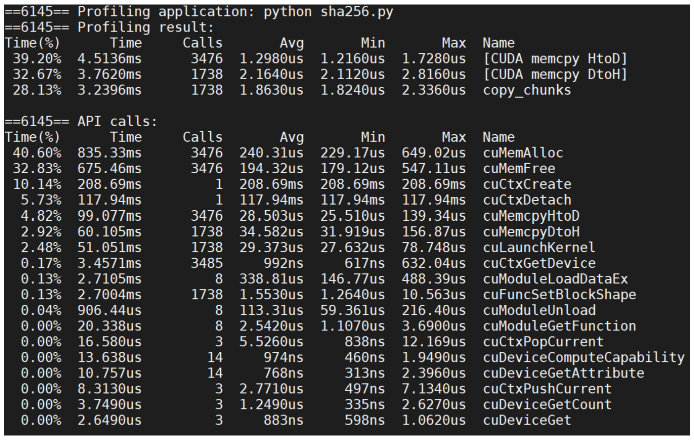
Fig. 5. Elapsed time for the GPU and non-GPU copy time versus input file size.

to perform relatively poorly, ratios trend down as input data size increases logarithmically until about 10 megabytes in file size. A minima exists between 100 kilobytes and 10 megabytes - likely indicating the limits of this specific parallelism configuration. Before GPU could show a respectable speed increase, delays accumulated significantly.

Fig. 6 is a screenshot taken from the NVIDIA Profiling tools after running several GPU hashes. Roughly two-thirds of

String Size	GPU Time	CPU Time	GPU/CPU Ratio
Small	1.10E-05	0.57	5.17E+04
1 KB	0.16	0.38	2.34
10 KB	1.49	2.03	1.36
100 KB	14.87	18.55	1.25
1 MB	150.14	179.27	1.19
10 MB	1482.13	1796.5	1.21

TABLE I
Ratios of GPU hash time to CPU hash time



```
==6145== Profiling application: python sha256.py
==6145== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 39.20%  4.5136ms      3476  1.2980us  1.2160us  1.7280us  [CUDA memcpy HtoD]
 32.67%  3.7620ms      1738  2.1640us  2.1120us  2.8160us  [CUDA memcpy DtoH]
 28.13%  3.2396ms      1738  1.8630us  1.8240us  2.3360us  copy_chunks

==6145== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 40.60%  835.33ms      3476  240.31us  229.17us  649.02us  cuMemAlloc
 32.83%  675.46ms      3476  194.32us  179.12us  547.11us  cuMemFree
 10.14%  208.69ms         1  208.69ms  208.69ms  208.69ms  cuCtxCreate
  5.73%  117.94ms         1  117.94ms  117.94ms  117.94ms  cuCtxDetach
  4.82%  99.077ms      3476  28.503us  25.510us  139.34us  cuMemcpyHtoD
  2.92%  60.105ms      1738  34.582us  31.919us  156.87us  cuMemcpyDtoH
  2.48%  51.051ms      1738  29.373us  27.632us  78.748us  cuLaunchKernel
  0.17%  3.4571ms      3485     992ns     617ns  632.04us  cuCtxGetDevice
  0.13%  2.7105ms         8  338.81us  146.77us  488.39us  cuModuleLoadDataEx
  0.13%  2.7004ms      1738  1.5530us  1.2640us  10.563us  cuFuncSetBlockShape
  0.04%  906.44us         8  113.31us  59.361us  216.40us  cuModuleUnload
  0.00%  20.338us         8  2.5420us  1.1070us  3.6900us  cuModuleGetFunction
  0.00%  16.580us         3  5.5260us     838ns  12.169us  cuCtxPopCurrent
  0.00%  13.638us        14     974ns     460ns  1.9490us  cuDeviceComputeCapability
  0.00%  10.757us        14     768ns     313ns  2.3960us  cuDeviceGetAttribute
  0.00%  8.3130us         3  2.7710us     497ns  7.1340us  cuCtxPushCurrent
  0.00%  3.7490us         3  1.2490us     335ns  2.6270us  cuDeviceGetCount
  0.00%  2.6490us         3     883ns     598ns  1.0620us  cuDeviceGet
```

Fig. 6. NVPROF Profiler screenshot of utilization for running GPU hashes for a simple message and 1 KB, 10 KB, and 100 KB text files.

device time is consumed by memory transfers alone, with the remaining time used for kernel compute. The time allotted to the kernel is appreciable, but leaves more to be desired for the ideal performance enhancement. The sections on Discussion and Further Work as well as the Conclusion go into further detail on this.

IV. Discussion and Further Work

Two areas for further investigation open up here. The first is to explore additional methods of paralleism for the SHA-256 algorithm. Either attempting to partially parallelize the expansion or compression functions. With enough of the workflow pipelined, a GPU may prove to be more robust in the algorithm. It should be noted that the parts of the algorithm left to run sequentially are performing very fast bit-wise operations, so dramatic increases are not likely. The second area is to explore other algorithms in hashing such as the MD5 or the SHA-512. The unique workflows may have workflows that can be pipelined unlike that of the SHA-256 algorithm. Various encryption algorithms like the Advanced Encryption Standard (AES) may also host the potential for parallelism.

On the note of additional CUDA tools, many parallel computing problems will leverage more niche resources such as shared memory. Shared memory is useful for tasks where data reuse occurs frequently and memory access is coalesced. The advantage is that shared memory exists very close to the processor core, unlike global memory. While smaller in capacity, the small distance to travel on the processor die reduces data transfer delays. A commonly-referenced example is tiled matrix convolution, where a tile (or kernel) or scalars experience high reuse during matrix multiplications. This study examines an algorithm that is very different from matrix convolution, and is much less able to benefit from shared memory. SHA-256 has a set of constants used in this calculation, see [4]. They are the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers. One hypothesis was to use constants involved with the SHA-2 algorithm stored in shared memory. The compression phase, which uses the constants, occurs for every 512-bit chunk in the input data could possibly benefit from shared memory. A problem that arises here is that, while the constants get used many times, the amount of time in between use of constants is fairly long, less than one use every 10 computations. This contrasts with shared memory in matrix convolution where nearly every operation makes use of the shared tile. Shared memory would therefore be unlikely to add any advantage to the operation.

V. Conclusion

This paper seeks to explore advancements for computational cryptography using heterogeneous computing. With more sensitive data transmitted every day, companies and institutions may have an increased demand for performance security. The experiment conducted investigates, albeit at a shallow depth, the impact of GPU computing for hashing functions. To further speculate, more investments in this area should produce more profound security measures, such as hardware-based security versus software-based. Consider, for example, a heterogeneous platform where a hash or encryption is dependent on the configuration of gates and register used in a design. A propriety hardware encryption device may be near-impossible to reverse engineer in comparison to a software algorithm that can be exposed through trial-and-error. Consider also a high-density computation for encryption that can only compute in a respectable time with GPUs versus CPUs. The less-than-ethical will encounter new challenges in nefarious schemes without the resources to produce high-end decryption hardware.

Limitations in this experiment include the shortcomings of parallelism within the algorithm, the diversity of algorithms tested, and the underuse of advanced parallel computing techniques. Should these limitations be approached more closely, more insightful conclusions can be reached.

Heterogeneous computing and computer hardware is on the cusp of becoming a popular platform for developers and scientists alike. Innovations of all kinds should be receive ample attention to push this industry forward, as it may prove to be as influential in the industry as computer software.

VI. Acknowledgements

The author would like to acknowledge both the professor Dr. Zoran Kostić and teaching assistant Abhyuday Puri for their guidance and mentorship in preparation for this research undertaking. Additionally, acknowledgements should be given to Isaac Donis, Esha Trivedi, and Drew Afromsky for being valuable classmates and exceptional team players in the months leading up to this endeavor. Strides in innovation do not often happen by oneself. As the saying goes, ”none of us is as smart as all of us.”

References

[1] Project source code, $https : //www.github.com/ChristopheBrown/hash$

[2] https://tools.ietf.org/html/rfc4634 Lyudmil Latinov. $https : //automationrhapsody.com/md5 - sha - 1 - sha - 256 - sha - 512 - speed - performance$ AutomationRhapsody.com, 2018.

[3] U.S. Department of Commerce: Secure Hash Standard (SHS) $https : //csrc.nist.gov/$ Addison-Wesley, Reading, Massachusetts, 1993.

[4] Wikipedia: SHA-2 $https : //en.wikipedia.org/wiki/SHA-2$. 2019.

[5] Network Working Group: US Secure Hash Algorithms (SHA and HMAC-SHA) $https : //en.wikipedia.org/wiki/SHA - 2$. 2006.

[6] bitbucket.org: _sha256.py, $https : //bitbucket.org/pypy/pypy/src/tip/lib_pypy/_sha256.py$

[7] stackexchange.com: Self-contained SHA-256 implementation in C, $https : //codereview.stackexchange.com/questions/182812/self - contained - sha - 256 - implementation - in - c$

[8] Mikael Fargard: sha256.py, $https : //github.com/falgard/SHA256/blob/master/sha256.py$

[9] iwar.orgn: Hash-Buster, $http : //www.iwar.org.uk/comsec/resources/cipher/sha256 - 384 - 512.pdf$

[10] Nicolas Courtois: Descriptions of SHA-256, SHA-384, and SHA-512, $https : //www.researchgate.net/figure/One - compression - function - in - SHA - 256 - It - comprises - a - 256 - bit - block - cipher - with - 64 - rounds_fig3_258144528$

[11] freeformatter.com: Sha-256 Generator, $https : //www.freeformatter.com/sha256 - generator.html$

[12] Mark Gritter: How does the SHA256 expander work?, $https : //www.quora.com/How - does - the - SHA256 - expander - work$

[13] John Franco: Hash Algorithms, $http : //gauss.ececs.uc.edu/Courses/c6053/lectures/PDF/mdalgs.pdf$

[14] Jerry Coffin: Is there a limit for sha256 input?, $https : //stackoverflow.com/questions/17388177/is - there - a - limit - for - sha256 - input$

[15] Vincent Hindriksen: How expensive is an operation on a CPU?, $https : //streamhpc.com/blog/2012 - 07 - 16/how - expensive - is - an - operation - on - a - cpu/$

[16] Making a multi-processor string evaluation algorithm, $https : //stackoverflow.com/questions/31237286/making - a - multi - processor - string - evaluation - algorithm$

[17] sample-videos.com: Sample Text string genterator, $https : //www.sample - videos.com/sample - text.php$

[18] James Lloyd: What is SHA-256?, $https : //www.quora.com/What - is - SHA - 256$