CSIT Conference 2019, Yerevan, Armenia, September 23-27

Heterogeneous Hashing Optimization with OpenCL and CUDA

Nikita Storublevtcev
St-Petersburg State University, St-Petersburg, Russia
e-mail: 100.rub@mail.ru

Victor Smirnov
St-Petersburg State University, St-Petersburg, Russia
e-mail: ariox41@gmail.com

Alexander Bogdanov
St-Petersburg State University and Russian Economical University, St-Petersburg, Russia
e-mail: bogdanov@csa.ru

ABSTRACT

Modern GPU-based heterogeneous systems offer an exceptional computation potential, but require specific expertise on the part of the developer to fully exploit it. There are also multiple technologies on the market that support GPU-based computation, but they all have different demands, quirks and range of supported hardware. It is important for any project aiming use heterogeneous systems to make an informed choice on what technology to employ or support. In this paper we compare performance of GOST 34.11-2012 ("Stribog") hashing algorithm implemented using CUDA and OpenCL. This algorithm also has a lot of potential in blockchain technology. We also discuss the optimization techniques used and their effectiveness in terms of both computation speed-up and resource investment.

Keywords

Computing, CUDA, OpenCL, Hashing, Performance, Heterogeneous, C++

1. INTRODUCTION

Hashing is a widely-used operation in today's internet infrastructure. Everything from search functions and data protection protocols to blockchain platforms rely on it to function. Hashing of large amounts of data, particularly in case of secure hashing algorithms, is resource-intensive, so any optimization in that area is welcome. It is well-known that most hashing algorithms can be easily adapted to heterogeneous GPU-based systems, and receive significant boosts in performance from that. Such porting has been done to all of the popular algorithms, but less-known ones are frequently forgotten. There are also some algorithms that are specifically designed to be ASIC-resistant or even GPU-resistant.

The algorithm of interest to us is GOST 34.11-2012 "Stribog", the use of which is mandated by Russian Government. We were unable to find any open-source heterogeneous implementation of this algorithm, so we elected to create and benchmark it on the test case of long multi-block message hashing. We use that case instead of cryptocurrency mining, as it is more relevant to the general public. We also measure the performance in Megabytes per second, instead of hashes per second, though these units can be converted into one another in most cases.

In this paper we present the performance results of two implementations of Stribog-512 hashing algorithm, one based on CUDA technology and another on OpenCL, and compare them to CPU-based performance and to each other. We also discuss the algorithm's suitability for GPU acceleration and techniques that can be used to accelerate it.

2. ANATOMY OF STRIBOG

Stribog, detailed in GOST 34.11-2012 federal standard of Russian Federation, is a cryptographic hash function, created to replace an obsolete GOST R 34.11-94 standard. It is based on the MerkleDamgrd construction, with block size of 512 bit. Overall structure resembles that of the old standard, with significant changes to compression function, which operates in MiyaguchiPreneel mode and uses a 12-round cipher with 512-bit block and key [1].

When a message is processed by Stribog it is first broken down into 512-bit blocks. If message's length is not a multiple of 512, the last block (one with a size less than 512) is padded to the desired size. Each block is then processed sequentially as each block processing updates internal constants for the next block. Processing of a block includes a compression function, which, in turn, includes a 12-round AES-like cipher based on LPS transformations [2], where L stands for linear transformation, P marks byte reordering, and S is a non-linear bijective transformation. Stribog can also work in 256-bit mode by changing its initial internal state and truncating the output hash.

The byte order is a contentious issue in Stribog as it is not explicitly dictated by the standard, which leads to difficulties in determining if the implementation is done correctly. Our implementations are based on ones by RustCrypto [3] and Oleksandr Kazymyrov [4]. These implementations use little-endian byte order and, therefore, do not pass comparison tests against big-endian ones. Based on them, we created big-endian table implementations that pass the comparison tests with popular non-table big-endian ones.

Since its release, Stribog has passed cryptanalysis trials by the international community with several reduced-round version collision attacks published, and the only concerning result being as-of-yet unconfirmed weakness of the S-Box generation algorithm [5]. It is considered secure overall, but still has a very low adoption rate, despite having recently been included into the Linux kernel as one of the available hashing algorithms. Only the future will show if it sees wide adoption.

3. GPU SPECIFICS

GPU-based computational systems excel at massive parallelism, especially in data-parallel cases, so we have to analyze Stribog with that in mind. Processing of separate messages can always be parallelized, because they are independent of each-other. One message is unlikely to take up all the available GPU resources in the system, so it is an obvious and easy way to increase overall performance. However, it requires having several different messages that have to be hashed in the first place, which might now always be the case. Additionally, it is not always easy to organize a non-exclusive use of the GPU, that would allow for the message hashing to only use as much resources as needed and not the whole device. NVIDIA provides some virtualization capabilities in its products, which show inconsistent results, but OpenCL has no such functionality. This means that most of the time we will have to manually organize the parallel use of GPU resources to process several messages at the same time. Things become even more complex when the messages are of different length. We can either package one block from each message and send them for processing as a batch, or send the maximum possible number of blocks (limited by the shortest message) from each, which will reduce the memory transfer overhead. Overall, this approach can be implemented for Stribog.

Looking deeper, we immediately see that inter-block parallelism within a single message is not possible, because processing of each block changes global internal constants used in further calculations. We have no choice but to leave this level sequential, and look deeper for more parallelism opportunities.

LPS transformations may be exactly the thing we are looking for. All of them are vector operations, which have an excellent potential for GPU optimization. On the other hand, each operation is always done on a 512-bit block so it does not scale very well, as each block still has to be processed sequentially. As each block undergoes multiple sequential transformations, it may be a good idea to place it in local memory space of the GPU to cut down on access delays. This alone has the potential to significantly increase the performance. The whole algorithm, in general, makes heavy use of various constants, which are ideal candidates for placing into the register memory, reducing access times even further.

As we cannot go deeper than vector operations, this constitutes all the possible heterogeneous optimizations available for this algorithm. After reviewing all the options, a table version of Stribog was chosen for implementation. Table version combines LPS transformations, by preemptively calculating constants and combining them into a single table. This increases the number of constants used, but reduces the number of reads required for each.

4. RESULTS

CPU implementation was written in Rust programming language, and makes uses of automatic SIMD optimization provided by the compiler. The resulting performance is roughly equivalent to that of other correct and optimized implementations. Faster implementations do exist, but they don't pass the comparison tests, and we cannot verify their correctness.

GPU implementations were created using OpenCL 1.2 and CUDA 9.2. OpenCL host-side code written in Rust, and kernel code written in C. CUDA version was written entirely in C++. Test platform consists of Intel Xeon E5-2690 v4 CPU and NVIDIA GP102GL Quadro P6000 GPU, running under Fedora 26 operating system.

Table 1 contains the benchmark results for processing of one message at a time. It is obvious that CPU implementation is faster here, as CPU has a much faster clock speed, and we don't make use of the massive parallelism of GPU. However, the difference in performance is not proportional to the frequency difference. Here we see how the CPU-GPU memory transfer overhead tanks the resulting performance of GPU versions. Next, let us look at the best-possible situation for GPU.

Table 1: Performance results for individual message processing

Implementation	Peak performance (MB/s)
CPU	19.05
OpenCL	2.63
CUDA	3.34

Table 2 showcases the peak performance when processing multiple messages at the same time. As predicted, it was not possible to data-parallelize the whole algorithm because some parts of it have to remain sequential, but we managed to parallelize the processing of each individual block of 64 bytes using 8 threads for each. The number of threads is determined by the specifics of LPS transformations, where each block is divided into 8 64-bit parts. For this case we used messages of constant length of 512 blocks (32 MB) that were all processed at the same time. The GPU finally outperformed the CPU after 128 concurrent messages being processed, and plateaued at around 4096 messages.

Table 2: Performance results for concurrent message processing

Implementation	Peak performance (MB/s)
CPU	201.33
OpenCL	380.92
CUDA	479.95

5. CONCLUSION

Results suggest that Stribog hashing algorithm is relatively GPU-resistant. It does not scale well for the massively-parallel GPU architecture, mandating several parts of its structure to remain sequential, but can be adapted for parallel processing of multiple independent messages. The extent of that use-case is not clear, as it is quite rare to have hundreds of messages that require hashing at the same time. However, there is a potential for further optimization of vector operation used at the lowest level of the algorithm.

REFERENCES

[1] GOST R 34.11-2012: Hash Function `https://tools.ietf.org/html/rfc6986`

[2] GOST-R-34-11-2012 `http://docs.cntd.ru/document/gost-r-34-11-2012`

[3] https://github.com/RustCrypto/hashes/tree/master/streebog

[4] https://github.com/okazymyrov/stribog

[5] Reverse-Engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1 https://eprint.iacr.org/2016/071