FAST HASHING IN CUDA

Neville Walo

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Describe in concise words what you do, why you do it (not necessarily in this order), and the main result. The abstract has to be self-contained and readable for a person in the general area. You should write the abstract last.

1. INTRODUCTION

Hash functions are one of the most important operations in cryptographic applications, like digital signature algorithms, keyed-hash message authentication codes and the generation of random numbers. Furthermore, hashing is also used in many data structures and applications such as hash tables and for calculating checksums to compare files. The acceleration of this routine is therefore of great importance for many areas.

Motivation. With the recent rise of the cryptocurrencies, it is as important as never before to hash as fast as possible. Many cryptocurrencies are based on the *proof of work* [1] principle, in which one party (the prover) proves to others (the verifiers) that a certain amount of computational effort has been expended for some purpose. For example, in the Bitcoin protocol [2], users have to find a *nonce* such that the SHA-256 hash of the nonce and the current block is smaller than the current target of the network. Since only the first miner who finds a nonce that fulfills the target receives a reward, it is important to try out many SHA-256 hashes as fast as possible. Today mostly ASICs (application-specific integrated circuit) are used to mine Bitcoins, as ASICs work more efficient and compute more hashes per second than traditional hardware.

While the original SHA-256 implementation, as proposed in the Secure Hash Standart (SHS) [3] by the National Institute of Standards and Technology (NIST), does not allow for much parallelization due to sequential dependencies, it is possible to use the compression function of SHA-256 along with the Sarkar-Schellenberg composition principle [4] to create a parallel collision resistant hash function: PARSHA-256 [5].

Implementing PARSHA-256 efficiently, even within a single GPU or CPU, is a complex challenge that needs to take the memory mode land architectural details into account.

In this work we try to speed up hashing by writing hashing algorithms in CUDA [6] to execute them on a GPU. We have divided this project into two sub-projects. The first one is the "Bitcoin" scenario, where the goal is to calculate many independent SHA-256 computation in parallel. The second case is PARSHA-256, where the goal is to implement the proposed algorithm in CUDA.

Related work. To our knowledge there is no comparable implementation of PARSHA-256 which runs on a GPU. There exists only the implementation of the original paper, which uses multithreading [5]. On the other hand, there are countless implementations of Bitcoin Miners in CUDA [7, 8], as this was the most prominent way to mine Bitcoins before ASICs were introduced.

Our contribution is an implementation of SHA-256 and PARSHA-256 in CUDA.

2. BACKGROUND

This section gives an overview of how PARSHA-256 and SHA-256 and work, mainly taken from the original sources [3, 5]. The focus is on the technical implementation (how the algorithm works), not on the theoretical properties (why the algorithm is secure). A *word* size of 32-bits is assumed.

2.1. SHA-256 [3]

The SHA-256 algorithm can be described in two stages: preprocessing and hash computation. Preprocessing involves padding a message, parsing the padded message into 512-bit blocks, and setting initialization values to be used in the hash computation. The hash computation generates a *message schedule* with size 64 words from the padded message and uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is called a *message digest* and it's length is 256 bits.

2.1.1. Preprocessing

Preprocessing consists of three steps: padding the message, parsing the message into message blocks, and setting the initial hash value.

Padding. The purpose of padding the input message is to ensure that the padded message is a multiple of 512 bits, since SHA-256 assumes a block size of 512 bits. Suppose that the length of the message, M, is ℓ bits. The bit "1" is appended to the end of the message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $\ell + 1 + k \equiv 448 \mod 512$. Then the 64-bit block that is equal to the number ℓ expressed using a binary representation is appended to the message. This will result in a message that can be divided into 512-bit blocks.

Parsing the Message. The message and its padding are parsed into N 512-bit blocks, $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$.

Initial Hash Value. Before hash computation begins, the initial 8 hash values, $H_0^{(0)}$ up to $H_7^{(0)}$, must be set. These can be taken from the official document [3].

2.1.2. Hash Computation

SHA-256 uses 8 working variables with size 1 word and six logical functions ($Ch(x, y, z)$, $Maj(x, y, z)$, $\sigma_0(x)$, $\sigma_1(x)$, $\Sigma_0(x)$,$\Sigma_1(x)$), where each function operates on 32-bit words, which are represented as x, y, and z. The result of each function is a new 32-bit word. The exact specification of these functions can be found in the official document [3].

Each message block is processed in order, using the following steps:

1. The Message Schedule $\{W_t\}$ of the current block is prepared using the following approach:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) \ + \ W_{t-7} \ + & 16 \leq t \leq 63 \\ \sigma_0(W_{t-15}) + W_{t-16} & \end{cases}$$

2. The eight working variables are initialized with the $(i-1)^{st}$ hash values. This means that block i receives as working variables the message digest of block $i-1$, while the first block receives the initial hash values as working variables.

3. In 64 rounds the working variables are permuted using the above functions, the message schedule and predefined constants. The exact specification found in the official document [3].

4. To compute the i^{th} intermediate hash value, the 8 working variables are added to the $(i-1)^{th}$ hash value.

After repeating steps one through four a for every block, the resulting 256-bit message digest of the message, M, is the hash value of the final block.

2.2. PARSHA-256 [5]

PARSHA-256 uses the compression function of SHA-256 along with the Sarkar-Schellenberg [4] principle to create a parallelizable collision resistant hash function. The overall approach does look not that different from SHA-256, but there is one major change. While the task graph in SHA-256 is linear, because the previous block has to be processed to process the next one, in PARSHA-256 the processors are arranged in a binary tree (similar to a reduction), which allows to work on more than one block at a time.

2.2.1. Compression Function

Let $h()$ be the compression function. In the case of SHA-256, the input to $h()$ consists of 24 32-bit words (768 bits) and the output consists of 8 32-bit words (256 bits). In the rest of the paper we set $n = 768$ and $m = 256$.

2.2.2. Processor Tree

PARSHA-256 uses a binary tree of processor height T, note that T is an argument to the hash function and a different T can produce a different result. There are 2^T processors in the tree and the children of processor P_i are P_{2i} and P_{2i+1}, see Fig. 1. The arcs denote the data flow and go from the children to the parent.

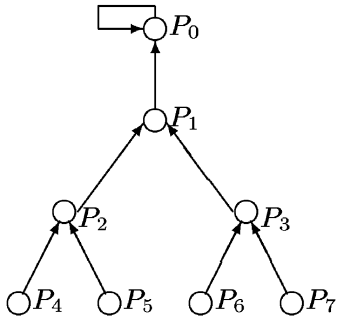

Fig. 1. Processor Tree with T = 3. Source: [5]

The behaviour of any processor P_i with input y is described as follows:

$$P_i(y) = \begin{cases} h(y) & \text{if } |y| = n \\ y & \text{else} \end{cases} \quad (1)$$

2.2.3. Formatting the Message

Similar to SHA-256, the incoming message has to be padded and divided into blocks. While for SHA-256, this procedure was relatively simple, as the message should be a multiple of 512 bits, which is also the block size, in PARSHA-256 it is more complicated.

In PARSHA-256 the message undergoes two kinds of padding. In the first kind of padding, called *end-padding*, zeros are appended to the end of the message to get the length of the padded message into a certain form. The second kind of padding is called *IV-Padding*. The **I**nitialization **V**ector (IV) ensures that no invocation of $h()$ gets only message bits as input. Using an IV is relatively simple in the Merkle-Damgard composition scheme (used by SHA-256). The IV has to be used only in the first invocation of the compression function (in SHA-256 the IV is called initial hash values). In PARSHA-256 however, the IV has to be used at several points. In PARSHA-256 the length l of the IV can be either 0,128 or 256 bits.

The exact procedure how the message is padded and divided into blocks is rather cumbersome and is therefore omitted here. The reader can find more information in the original paper [5].

2.2.4. Hash Computation

This subsection demonstrates the hash computation of a message of suitable length with a processor tree of height $T = 3$ and $l = 0$. The message length is $L = 2^T(p + 2)(n - 3) - (n - 2m)$, for some integer $p \geq 0$.

The whole computation will be done in $(p + 4)$ parallel rounds. In each round some or all of the processors work in parallel and invoke the compression function on its input to produce its output.

The input message will be broken up into disjoint substring of length n or $n - 2m$. These substrings will be provided as input to the processors in the different rounds. We call u_i the substring of the input message provided to the processor P_i in a particular round and z_i the output message of processor P_i in a particular round.

The description of the rounds is as follows, see Fig. 2:

1. In the first round, all processors get an n-nit substring u_i from the input message and produce an m-bit output z_i by invoking the compression function.

2. In rounds 2 to $(p + 1)$ the computation proceeds as follows:

   - The processors which are a leaf in the tree (P_4, P_5, P_6, P_7) each get an n-bit substring from the input message.
   - All non leaf processors (P_0, P_1, P_2, P_3) get an $(n - 2m)$-bit substring from the input message and concatenate this with the two m-bit messages z_{2i}, z_{2i+1} from their children from the previous round.

3. In round $(p + 2)$ all non leaf processors get an $(n - 2m)$-bit substring from the input message and concatenate this with the two m-bit messages z_{2i}, z_{2i+1} from their children from the previous round. The leaf processors do not receive any new input.

4. In round $(p + 3)$ only P_0 and P_1 get an $(n - 2m)$-bit substring from the input message and concatenate this with the two m-bit messages z_{2i}, z_{2i+1} from their children from the previous round. The other processors do not receive any new input.

5. In round $(p+4)$ only P_0 gets an $(n-2m)$-bit substring from the input message and concatenates this with the two m-bit messages z_{2i}, z_{2i+1} from their children from the previous round. The other processors do not receive any new input. This input is hashed to obtain the final message digest.

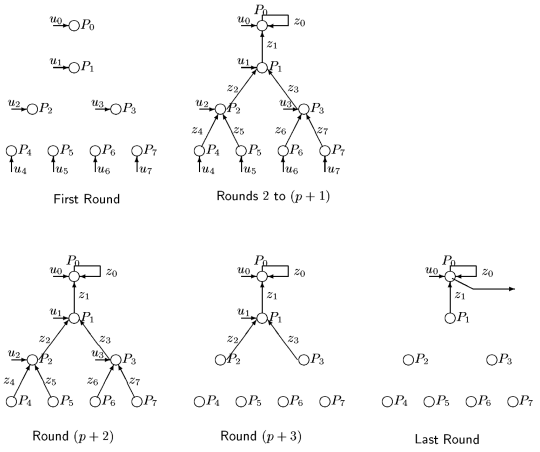

Fig. 2. Example Hash Computation. Source: [5]

3. OUR WORK

This section describes our implementation of SHA-256 and PARSHA-256 in CUDA.

3.1. SHA-256

SHA-256 is specified using binary input and output, but most implementations, including ours, work with strings as input and output, since the char sequence can be interpreted as a binary sequence.

The message padding is performed on the host and then the padded message is copied into the global memory of the GPU. In a real Bitcoin Miner the message has to be adapted to the *threadIdx.x* and *blockIdx.x*, as each thread has to process a different nonce. The modification should be performed locally to each thread using its registers and not on the globally stored message. However, we omitted this modification and all threads work on exactly the same input message, since our goal was not to create a Bitcoin Miner, but to get a performance estimate of the involved hashing.

After the message is stored in global memory, the kernel is called with the required number of threads and blocks. All constants used in the hash calculation are declared directly in the kernel using *constexpr*, such that the compiler can optimize access and does not store the constants in memory. Similarly, the initial hash values are declared directly in the kernel. All functions used in the computation are marked with *__inline__* and use *const* input such that the compiler can inline and optimize them correctly.

To compute the message schedule, an array of 64 words is allocated. This array is filled using two for-loops. The first loops copies 16 words from the input message into the array using vector loads. The second for loops perform the calculation described in step 1 of 2.1.2. Both loops are marked with *#pragma unroll*, as the iteration count is known in advance.

Afterwards, the 64 rounds of permutations, as described in step 3 of 2.1.2, is performed, using again an unrolled for loop.

We note that only 16 words of the message schedule are used simultaneously, which means, it would be possible to integrate the generation of the message schedule into the permutation loop to save registers. However, since all loops are unrolled and all functions are inlined, the compiler can, by looking at the dependencies, find the correct instruction order, which minimizes register usage to maximize occupancy.

After all blocks are processed the final result is written back to global memory using vector stores.

3.2. PARSHA-256

Similar to SHA-256, all preprocessing is performed in the host. Once the message is padded, the tree structure and all necessary information to hash the input message is known, 3 global buffers are allocated. One buffer stores the padded input message, the other two buffers store the output for each thread in a specific round. Using two buffers to store the intermediate results allows to read from one buffer while the other one is written to which significantly reduces the amount of synchronization needed between threads.

Each thread knows the *id* of its children and thus the address, from where the input can be read for the next round. One kernel is launched for each round, as described in 2.2.4. Four different kernels were implemented, one for the first round, one for round 2 to $(p+1)$, one for the last round and one for all the other rounds, in which some threads copy the input to the output buffer. As described in Equation 1, in each round each thread calls the compression function using its input or copies its input to the output buffer. After each round, the two buffers are swapped, such that the output from the previous round is the input for the next round.

3.2.1. Shared Memory and Registers

The exclusive use of global memory and multiple kernel launches for communication and synchronization between threads makes the code relatively elegant, since execution is determined only by which thread has which children. However, this approach is also slow and does not make use of the more advanced memory and synchronization mechanisms of modern GPUs, such as shared memory, registers or warp shuffle instructions.

The first problem, if these mechanisms are to be used, is the following: The PARSHA-256 specification only specifies a maximum tree height T, but if the message is sufficiently small, a smaller tree with effective tree height t is used instead. This means that the exact structure of the tree is not known in advance and therefore it is also not how many threads will be launched.

The second problem concerns older models of NVIDIA GPUs in which all threads in a warp share the same instruction pointer. If this is the case, it can happen that one thread in warp has to communicate through global memory while all other thread can communicate through shared memory or registers. In the code itself these behaviors would have to be distinguished with an *if*-statement. Since all threads share the same instruction pointer, both cases are executed sequentially and the latency has basically doubled. In newer architectures this behaviour does not occur any more since every thread has its own instruction pointer.

Assuming that the GPU used and the specified tree are known in advance, each thread has its own instruction counter, it is possible to implement PARSHA-256 more efficiently by using more advanced mechanisms. We will now present implementations for some special cases.

Warp PARSHA-256. If $t \leq 5$ at most 32 threads are launched, which means the whole calculation can fit into in a single warp, see Fig. 3. In that case warp shuffle instruction can be used to exchange the data, *__syncwarp()* can be used to synchronize the threads and the intermediate results can be stored into registers.

Block PARSHA-256. If $t \leq 10$ at most 1024 threads are launched, which means the whole calculation can fit into in a single block, see Fig. 4. In that case shared memory and *__syncthreads()* can be used to communicate and synchronize between the warps. Note that in this case, the property

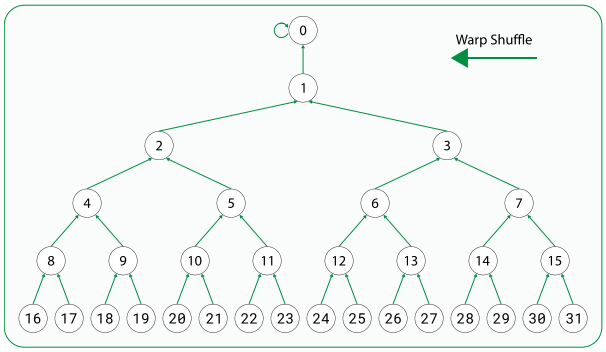

Fig. 3. Warp PARHSA-256. The threads communicate using registers,*__syncwarp()* and warp shuffle instructions.

that a thread with the id i has two children with the id $2i$ and $2i + 1$ no longer applies, but a more complicated relationship is required so that communication across warps is minimized.

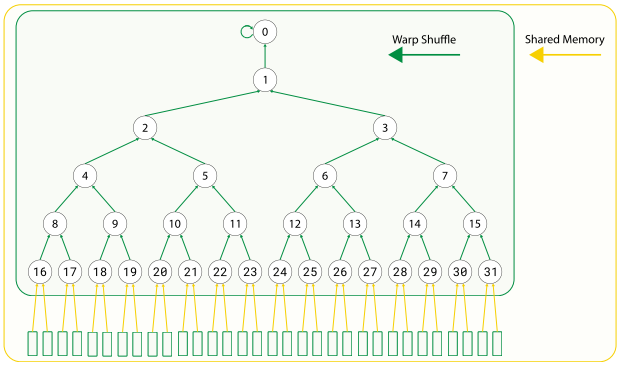

Fig. 4. Block PARHSA-256. The warps communicate using shared memory and *__syncthreads*.

If $t > 10$ the calculation does not fit into a single block and global memory and multiple kernel launches are necessary to communicate and synchronize between threads.

4. EXPERIMENTAL RESULTS

We evaluate the performance of PARSHA-256 and SHA-256 .

Experimental setup. The benchmarks were performed using the *Ault* cluster of the CSCS (Swiss Natinonal Supercomputing Center). The used nodes adhere to the specification described in Table 1. All kernels were compiled using NVCC (NVIDIA CUDA Compiler [9]) version 11.1 with the following flags: `-O3 -std=c++11 -gencode arch=compute_70,code=sm_70`. To measure the performance of the kernels *nvprof* [10] was used with the following options: `--concurrent-kernels off --csv --profile-from-start off --print-gpu-trace` . The turbo boost of the accelerators were disabled using `export CUDA_AUTO_BOOST=0`. As host compiler gcc [11] version 10.1 was used with the following flags: `-O3 -march=native -std=c++14`. Host code was measured using `std::chrono::high_resolution_clock`. It was only measured how long it takes to hash the padded message, the time for preprocessing the message was excluded as it is the same for every message.

Table 1. Specification of the Ault05/Ault06 nodes of the Ault cluster.

	Ault05/Ault06
Sockets	2
Cores/Socket	18
Threads	72
CPU Model	Intel® Xeon® Gold 6140
RAM/Node	768 GB
Accelerator	4 × NVIDIA Tesla V100 (32GB PCIe)

To collect performance results, every benchmark was performed 110 times with the first 10 iterations being warm-up rounds, where the time was not measured. In the next 100 iterations the kernel time was measured.

Results. A normal SHA-256 implementation requires 3384 integer operations to hash a 512 bit block. In CUDA, it is possible to use the *funnel shift* instruction to implement rotations and 3-way arithmetic instructions, which reduces the instruction count drastically. In our main loop there are 1385 arithmetic instruction, that contribute to hashing, and only 22 non arithmetic instructions, which mean our implementation can reach up to 98.4% of the reachable peak performance. Note that we compare against reachable peak performance and not *real* peak performance, since in order to achieve real peak performance one has to use instructions that have a higher throughput than regular instructions (e.g. Fused Multiply Add), but these do not appear in SHA-256.

In Fig 1, we can see ... Next divide the experiments into classes, one paragraph for each. In each class of experiments you typically pursue one questions that then is answered by a suitable plot or plots. For example, first you may want to investigate the performance behavior with changing input size, then how your code compares to external benchmarks.

For some tips on benchmarking including how to create a decent viewgraph see pages 22–27 in [**?**].

Comments:

- Create very readable, attractive plots (do 1 column, not 2 column plots for this report) with readable font size. However, the font size should also not be too large; typically it is smaller than the text font size.

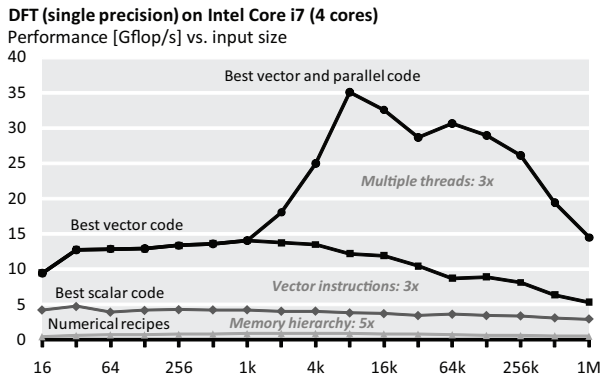

Fig. 5. Performance of four single precision implementations of the discrete Fourier transform. The operations count is roughly the same. The labels in this plot are maybe a little bit too small.

An example is in Fig. 5 (of course you can have a different style).

- Every plot answers a question. You state this question and extract the answer from the plot in its discussion.
- Every plot should be referenced and discussed.

5. CONCLUSIONS

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

6. FURTHER COMMENTS

Here we provide some further tips.

Further general guidelines.

- For short papers, to save space, I use paragraph titles instead of subsections, as shown in the introduction.
- It is generally a good idea to break sections into such smaller units for readability and since it helps you to (visually) structure the story.
- The above section titles should be adapted to more precisely reflect what you do.
- Each section should be started with a very short summary of what the reader can expect in this section. Nothing more awkward as when the story starts and one does not know what the direction is or the goal.
- Make sure you define every acronym you use, no matter how convinced you are the reader knows it.
- Always spell-check before you submit (to us in this case).
- Be picky. When writing a paper you should always strive for very high quality. Many people may read it and the quality makes a big difference. In this class, the quality is part of the grade.
- Books helping you to write better: [**?**] and [**?**].
- Conversion to pdf (latex users only):

  dvips -o conference.ps -t letter -Ppdf -G0 conference.dvi

  and then

  ps2pdf conference.ps

Graphics. For plots that are not images *never* generate the bitmap formats jpeg, gif, bmp, tif. Use eps, which means encapsulate postscript. It is scalable since it is a vector graphic description of your graph. E.g., from Matlab, you can export to eps.

The format pdf is also fine for plots (you need pdflatex then), but only if the plot was never before in the format jpeg, gif, bmp, tif.

7. REFERENCES

[1] Markus Jakobsson and Ari Juels, *Proofs of Work and Bread Pudding Protocols(Extended Abstract)*, pp. 258–272, Springer US, Boston, MA, 1999.

[2] Satoshi Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.

[3] "Secure hash standard," August 2015.

[4] Palash Sarkar and Paul Schellenberg, "A parallelizable design principle for cryptographic hash functions," 04 2002.

[5] Pinakpani Pal and Palash Sarkar, "Parsha-256 – a new parallelizable hash function and a multithreaded implementation," in *Fast Software Encryption*, Thomas Johansson, Ed., Berlin, Heidelberg, 2003, pp. 347–361, Springer Berlin Heidelberg.

[6] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2020, Version 11.1.0.

[7] *CudaMiner*, (accessed November 22, 2020), https://github.com/cbuchner1/CudaMiner/.

[8] *cuda_bitcoin_miner*, (accessed November 22, 2020), https://github.com/geedo0/cuda_bitcoin_miner.

[9] "NVIDIA CUDA Compiler," https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[10] "CUDA Profiling," https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[11] "GCC, the GNU Compiler Collection," https://gcc.gnu.org/.