Building a Collision for 75-Round Reduced SHA-1 Using GPU Clusters

Andrew V. Adinetz[1,2] and Evgeny A. Grechnikov[3]

[1] Lomonosov Moscow State University, Research Computing Center
adinetz@gmail.com
[2] Joint Institute for Nuclear Research
[3] Lomonosov Moscow State University, Faculty of Mechanics and Mathematics
grechnik@mccme.ru

Abstract. SHA-1 is one of the most widely used cryptographic hash functions. An important property of all cryptographic hash functions is collision resistance, that is, infeasibility of finding two different input messages such that they have the same hash values. Our work improves on differential attacks on SHA-1 and its reduced variants. In this work we describe porting collision search using method of characteristics to a GPU cluster. Method of characteristics employs backtracking search, which leads to low GPU performance due to branch divergence if implemented naively. Using a number of optimizations, we reduce branch divergence and achieve GPU usage efficiency of 50%, which gives 39× acceleration over a single CPU core. With the help of our application running on a 512-GPU cluster, we were able to find a collision for a version of SHA-1 reduced to 75 rounds, which is currently (February 2012) the world's best result in terms of number of rounds for SHA-1.

1 Introduction

A *cryptographic hash function* is a function which maps *messages* (bit-strings of arbitrary length) into *hash values*, or *hashes* (bit strings of fixed length). Such functions are widely used in modern cryptography and information security. A hash serves as a fingerprint for a message. An important property for practical applications of cryptographic hash functions is *computational infeasibility of finding a message with a given hash value*. A *collision* is a pair of different messages which give the same hash value. Due to limited size of hash value, collisions exist for any hash function; however, they are hard to find. If a collision has been built, then the cryptographic hash function is considered to be *compromised*, and is no longer suitable for practical applications. Collision search is therefore an important part of cryptoanalysis of hash functions.

Hash function called SHA-1 (Secure Hash Algorithm 1) maps messages of any length (maximum of $2^{64} - 1$ specified by the standard) into 160-bit hashes. It was published by NIST (National Institute of Standards and Technology) in 1995 and is now widely used in different government and industrial security standards, such as electronic digital signature, user authentication, key exchange

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 933–944, 2012.

and generation of pseudo-random sequences. SHA-1 is available in almost all commercial security systems.

Attempts to compromise SHA-1 have been performed for a number of years. They advanced far enough, though as of February 2012, no full SHA-1 collision has been built. Currently, NIST is holding the competition for a new cryptographic hash function to replace SHA-1. The new function is expected to be announced in 2012.

As a rule, cryptoanalytic problems are easily parallelized and scale well to any available computational resources. It seems therefore logical to solve them using GPUs. And though GPUs are quite widely used to solve problems such as password cracking [1], so far we haven't found any working application of GPUs to collision search.

The contribution of this paper can be summarized as follows:

- We have ported collision search using method of characteristics for SHA-1 to GPUs, and after performing optimizations we proposed, obtained 39× acceleration compared to a single CPU core
- With our application running on a GPU cluster, we have found a collision for reduced 75-round SHA-1, which is, as of February 2012, world's best result in terms of number of rounds for SHA-1.

This paper is organized as follows. We describe SHA-1 hash function and differential attacks in section 2. Section 3 describes the characteristic search algorithm. GPU implementation of message search are described in section 4. We describe computational experiments in section 5 and conclude in section 6.

2 SHA-1 and Differential Attacks

Notational conventions used in this paper are presented in Table 1. SHA-1 hash function [2] works as follows. First, the message is padded with bits, including message length, and split into 512-bit message blocks $M_1, \ldots, M_k$. The *compression function* $g(M, H)$ is then applied sequentially to compute $H_i = H_{i-1} + g(M_i, H_{i-1})$. H_0 is the *initial value* provided by the standard,

Table 1. Notational Conventions Used in This Paper

Notation	Description
X	32-bit unsigned integer related to 1st message
X^*	32-unsigned integer related to 2nd message
X^2	a pair of 32-bit unsigned integers (X, X^*)
$X \oplus Y$	exclusive OR (XOR)
$X + Y$	2^{32} wrap-around addition
$[X]_i$	i-th bit of X ($i = 0$ — least significant bit)
$X \lll i$	left rotation by i bits
$X \ggg i$	right rotation by i bits

and H_k is the hash value of the message. For building a collision, it is sufficient to provide two messages $(M_1, \ldots, M_k)$ and $(M_1^*, \ldots, M_k^*)$ of equal length so that $H_k = H_k^*$.

The compression function consists of 80 rounds and maps a 160-bit *input vector* H and 512-bit message block M into the new 160-bit value. Input vectors consist of 5 32-bit unsigned integers $H = (A_0, B_0, C_0, D_0, E_0)$, $M = (M_0, \ldots, M_{15})$, $g(M, H) = (A_{80}, B_{80}, C_{80}, D_{80}, E_{80})$. Computing the compression function consists of the *message expansion* and the *state update transformation*. 16-uint messagge M_i is expanded to 80 variables W_i as described by (1)

$$\begin{array}{ll} W_i = M_i & 0 \leqslant i < 16 \\ W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1 & i \geqslant 16 \end{array} \tag{1}$$

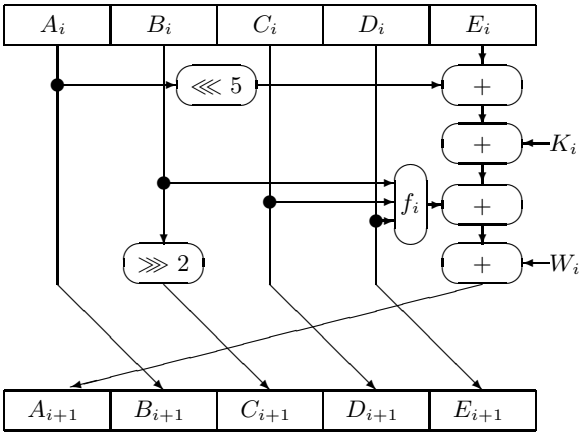

Fig. 1. One Round of SHA-1's Compression Function

One round of the state update transformation is described in Fig. 1. Constants K_i and functions f_i are defined by (2)

$$\begin{array}{lll} K_i = \texttt{0x5A827999}, & f_i(b,c,d) = (b \wedge c) \vee (\overline{b} \wedge d), & 0 < i \leq 20 \\ K_i = \texttt{0x6ED9EBA1}, & f_i(b,c,d) = b \oplus c \oplus d, & 20 < i \leq 40 \\ K_i = \texttt{0x8F1BBCDC}, & f_i(b,c,d) = (b \wedge c) \vee (b \wedge d) \vee (c \wedge d), & 40 < i \leq 60 \\ K_i = \texttt{0xCA62C1D6}, & f_i(b,c,d) = b \oplus c \oplus d, & 60 < i \leq 80 \end{array} \tag{2}$$

It's obvious that $B_i = A_{i-1}$, $C_i = A_{i-2} \ggg 2$, $D_i = A_{i-3} \ggg 2$, $E_i = A_{i-4} \ggg 2$, so having only A_i is enough. This is the notation used for the rest of the paper. $A_{-4}, \ldots, A_0$ give initial values while $A_{76}, \ldots, A_{80}$ can be used to compute the hash value. As building a collision for full 80 rounds requires very large computational resources which are not currently available, in our case we reduce the compression function to 75 rounds.

Differential attacks have been developed for some time. The main stages of their development (including attacks on other hash functions) are described in [4] (MD4), [7] (35-step SHA-0), [8] (full SHA-0), [5] (MD5), [6] (58-step SHA-1), [9] (64-step SHA-1), [10] (70-step SHA-1). We improve on the method described in works on 64 and 70-step SHA-1.

The key idea of differential attacks is to restrict the search to message pairs with a fixed difference modulo 2 $\delta M_i = M_i \oplus M_i^*$, hence the name. It turns out to be convenient to fix some bits also in M_i, A_i, and δA_i. Precisely, a *characteristic* is a set of $(80 + 85) \cdot 32$ elementary conditions on bit pairs $([W_i]_j, [W_i^*]_j)$ and $([A_i]_j, [A_i^*]_j)$, each allowing only certain combinations of bit pair values. There are $2^{2^2} = 16$ possible bit-pair conditions, the six actually used for collision search are described in Table 2.

Table 2. Bit Conditions Used in Characteristics

∇_i	(0, 0)	(1, 0)	(0, 1)	(1, 1)
-	✓	–	–	✓
x	–	✓	✓	–
0	✓	–	–	–
u	–	✓	–	–
n	–	–	✓	–
1	–	–	–	✓

Let ∇X be the set of pairs (X, X^*) satisfying all 32 bit-pair conditions for a variable. We want to perform exhaustive search over a given characteristic to find a collision. For each i, we search through values of M_i^2 allowed by chararacteristic, compute A_{i+1}^2 and check it against characteristic for state. If a suitable value is found, the search proceeds to round $i + 1$; if not, it backtracks to $i - 1$. After finding M_{16}^2, the message is fully defined and further steps perform only checking. If the *input freedom* for states A_{i+1}^2 is less than for messages M_i^2, we search through the values of state instead, as there is one-to-one correspondence between message and state once values for previous rounds A_i^2 are fixed. The search continues either until a collision is found, or the search space is exhausted.

We will now estimate complexity of the search, assuming that it is successful. A set $(W_0^2, \ldots, W_i^2)$ is *consistent* if it can be extended to a full set of expanded messages satisfying the characteristic. *Input freedom for the message* $\tilde{F}_W(i)$ *at step* i is the number of consistent sets $(W_0, \ldots, W_i)$ which extend the consistent set $(W_0, \ldots, W_{i-1})$. It is obvious that $\tilde{F}_W(i) = 1$ when $i \geqslant 16$. When conditions for $W_{16}, \ldots, W_{79}$ are trivial, for $i < 16$ we have $\tilde{F}_W(i) = |\nabla W_i|$. In general case, conditions for $W_{16}, \ldots, W_{79}$ impose linear equations on bits of $[M_i]_j$. When there are m independent equations, $\tilde{F}_W(i) = \frac{|\nabla W_i|}{2^m}$. Input freedom for the state is $\tilde{F}_A(i) = |\nabla A_{i+1}^2|$. When $\tilde{F}_A(i) \geqslant \tilde{F}_W(i)$, we search through M_i^2 and compute A_{i+1}^2. Otherwise, we search through A_{i+1}^2 and compute M_i^2. In the first case we assume $F_W(i) = \tilde{F}_W(i)$, and in the second $F_W(i) = \frac{\tilde{F}_A(i)}{2^m}$. Thus defined, $F_W(i)$

is the number of children nodes of the search tree at step i when implicit linear equations are taken into account.

For SHA-1, A_{i+1} is computed at each step based on $A_{i-j}, 0 \leqslant j \leqslant 4$, and W_i. For our estimation, we assume that $A_{i-j}, 0 \leqslant j \leqslant 4$, and W_i are simply independent random variables (irrespective to hash function) which satisfy the characteristic.

The *uncontrolled probability* $P_u(i)$ *at step* i is the probability that the result of step i satisfies the characteristic if all state and extended message values at previous steps satisfy the characteristic. That is, for $\tilde{F}_A(i) \geqslant \tilde{F}_W(i)$ and $\tilde{F}_A(i) < \tilde{F}_W(i)$ by it is defined by (3) and (4), respectively.

$$P_u(i) := Pr(A^2_{i+1} \in \nabla A_{i+1} | A^2_{i-j} \in \nabla A_{i-j}, 0 \leqslant j \leqslant 4, W^2_i \in \nabla W_i) \tag{3}$$

$$P_u(i) := Pr(W^2_i \in \nabla W_i | A^2_{i-j} \in \nabla A_{i-j}, 0 \leqslant j \leqslant 4, A^2_{i+1} \in \nabla A_{i+1}) \tag{4}$$

The *controlled probability* $P_c(i)$ *at step* i is the probability that at least one pair W^2_i satisfying the characteristic exists, such that the result of step i satisfies the characteristic on the condition that state values at all previous steps satisfy the characteristic. Formally (independent of whether A or W is enumerated) it is defiend by (5)

$$P_c(i) := Pr(\exists W^2_i \in \nabla W_i : A^2_{i+1} \in \nabla A_{i+1} | A^2_{i-j} \in \nabla A_{i-j}, 0 \leqslant j \leqslant 4). \tag{5}$$

We now estimate the complexity of a successful search. At step i the number of nodes $N_S(i)$ that must be traversed is, on average:

- $N_S(80) = 1$ (we need just a single collision),
- $N_S(i) = \max\left\{\frac{N_S(i+1)}{F_W(i)P_u(i)}, \frac{1}{P_c(i)}\right\}$ (on the one hand, a search tree node has on average $F_W(i)$ children, among which the fraction of $P_u(i)$ give the next level node; on the other hand, with probability $P_c(i)$ the node won't give any next level nodes).

We call the value defined by (6)

$$\sum_{i=0}^{80} N_S(i), \tag{6}$$

which depends on the characteristic only, the *work factor* of the characteristic. The less the work factor is, the better the characteristic is.

3 Finding a Characteristic

Finding a characteristic consists of three stages. At the first stage, a *linear* characteristic is searched for, which consists only of `-x` conditions; it fixes differences, but not bits. To do that, we construct a *linearization* of the hash functions by

replacing non-linear operations with their linear "approximations". The goal of this stage is to minimize the number of `x` in the characteristic, which lead to differences between the function and linearization. A search for a linear characteristic with small `x` conditions is expressed as searching for small-weight vector in some linear code, which is a known problem from the coding theory.

We construct a 2-block collision. The characteristic for each block is different, but is constructed based on the same linear characteristic. Resulting hash is given by $H_2 = H_1 + g(M_2, H_1) = H_0 + g(M_1, H_0) + g(M_2, H_1)$, $H_2^* = H_0 + g(M_1^*, H_0) + g(M_2^*, H_1^*)$. Linear characteristic gives $g(M_1, H_0) \oplus g(M_1^*, H_0)$ and $g(M_2, H_1) \oplus g(M_2^*, H_1^*)$; as it is the same for both blocks, we can make differences of first and second block values differ only in sign by fixing the bits that differ. This leads to $H_2 = H_2^*$, that is, a collision.

The second stage begins with discarding conditions for A_i at first 12 steps and replacing them with conditions for $A_{-4}^2, \ldots, A_0^2$. The initial condition is H_0 for the 1st block, and the result of the first block for the 2nd block. Therefore, we can construct 2nd block characteristic only after finding the 1st block of the collision. We also replace `xx` condition pairs for successive bits with `-x`, if the difference can be satisfied due to the carry. This is not always true because rotations are involved. At the second stage we need to find some "path" (a consistent set of conditions) from initial conditions to the linear characteristic. To do this, we choose random positions in A_i^2 which have no conditions, add `-` condition and find which additional conditions are satisfied based on ones already enforced. Also, when `x`-type conditons appear in A_i^2 it is useful to fix values of differing bits. If we find a contradiction, we backtrack to the last fixing `x` and choose an alternative fixing. The second stage finishes when all conditions have the form `-xun01`.

The third stage iteratively improves the work factor of the characteristic. To do this, we search through possible tightenings of conditions, *propagate* the new conditions, i.e. look at the additional conditions which follow from the new set of conditions, and compute the work factor for the new characteristic. At the end of the search the characteristic with the smallest work factor is chosen.

We'll note several important aspects of characteristic search here:

- F_W, P_u, and P_c are computed sequentially, from least to most significant bits by searching through elementary conditions and possible carries.
- The propagation of conditions is calculated in two passes. First, possible carries are evaluated from least to most significant bits, and then new conditions are evaluated taking carries into account. This is fast, but sometimes doesn't find all possible conditions (due to an interference between consecutive steps). To propagate further, we loop through bit positions, fix possible bit values and check if the fast procedure finds any contradictions. In the second stage we check only those bits who are close to some bit that was changed. In the third stage we check all bits.
- *Coherency*, i.e. similarity of control flow and memory accesses in neighbouring threads, in important for efficient GPU execution. Coherency can be improved by concentrating strong conditions in the middle of the initial

rounds of the characteristic. This is achieved by choosing these positions for - conditions with less probability at the second stage. This is the first GPU-related optimization, and it improves GPU search efficiency by 80%.

4 Message Search Implementation on GPU

Searching for a message which satisfies the characteristic is the most computationally intensive part of collision search. There is a number of points which do not depend on hardware:

- Characteristics always consist of conditions of type `-xun01`. Therefore, condition set for each 32-bit variable can be expressed as a pair of equations $X \oplus X' = a$, $X \wedge b = c$, where a, b, c are 32-bit values which depend only on characteristic.
- The following procedure is an efficient way to enumerate the set $\{X : X \wedge b = c\}$. The first element is $X := c$, every next element is given by the equation $X := (((X \vee b) + 1) \wedge \overline{b}) + c$, the enumeration is over when this equation gives $X = c$ due to overflow.
- As A_i at two last steps are not used in computing f_i, it is sufficient to check on those steps that $X - X' = a$. Moreover, for the 1st collision block conditions at two last steps can simply be ignored, as any difference due to them could be compensated by the 2nd block without increasing the number of conditions.
- Linear equations on M_i, appearing due to conditions on W_k for $k \geq 16$, can either express a bit $[M_i]_j$ through bits of previous message words, or give an equation involving values of 2 or more bits of M_i. In the first case, a, b, and c depend also on previous messages, and must be recomputed for each round. In the second case, these equations can be removed by adding some artificial conditions (e.g. imposing an additional restriction $[M_i]_j = 0$), without significantly changing the work factor.

The search is naturally divided into *generation* phase, which searches through message pairs, and *check* phase, which checks the rest of the characteristic for the pair of messages. Generation phase is a back-tracking search, and check is simply a function which is called at the last round of generation. Generation can in turn be divided into *host* part and *device* or *GPU* part. On the host, the search tree is expanded to a certain *host depth* to generate enough *search stacks* to make use of GPU parallelism. Host depth is specified individually based on the characteristic and available computational resources. Too little depth leads to insufficient parallelism, while with too large depth, search stacks won't fit into GPU memory. To utilize a single GPU efficiently, about 10^5 search stacks are needed.

During GPU part, the search is performed in parallel on a large number of GPUs. Stacks for which the search is finished are removed, and no new stacks are generated. The main GPU kernel implements back-tracking search and message check. In this kernel, each GPU thread processes only a single search stack for

a fixed number of search iterations. The main kernel also collects statistics on the number of traversed nodes, check rounds and maximum depth reached by the search. Between kernel calls, the depth is checked and the defunct stacks are removed from the array of search stacks.

The computation is distributed among cluster nodes using MPI. Each MPI process uses only a single GPU. Search stacks are distributed between nodes in block-cyclic way. During host part, each MPI process generates all search stacks and discards those belonging to other processes. There is a global barrier at the end of host part, but after that, all MPI processes run independently and asynchronously. This means that if search is finished on some stacks and some processes will have less search stacks than the others, there will be no load imbalance; just some of the stacks will be searched through quicker. The only communication involved is sending statistics to statistics collection thread of master process. The master process also spawns one more thread, which prints out aggregated statistics at fixed time intervals.

The application is implemented using Nemerle, an extensible .NET language, and NUDA (Nemerle Unified Device Architecture), a system of Nemerle extensions [11] for programming GPUs. NUDA was chosen due to its free availability and support for high-level GPU programming, including automatic host-GPU data synchronization and generation of kernels. Internally, OpenCL is used to interact with GPUs and as a target for GPU code generation. mono is used to run .NET applications on Linux, and MPI .NET provides .NET bindings for MPI.

We first implemented GPU back-tracking search as a single loop, with branches inside the body handling specific conditions. There were 2 such conditions: round switch and message check. A round switch can arise when a successful message word is found, when all words are exhausted or when the kernel starts. In any case, large number of additional precomputing is required. Message check was implemented as a separate function, with loop on rounds fully unrolled using `inline` annotation available in NUDA.

The performane of our initial implementation, however, was unsatisfactory. While the application scaled well due to little communication, computational efficiency was only 15% on some characteristics. We define computational efficiency as the ratio of really executed integer operations to peak GPU performance in terms of integer operations. Low efficiency was due to low coherency between threads in a single warp, so we concentrated on improving coherency. The first optimization, described in section 3, was modifying search algorithm, which gave 1.8× improvement of efficiency.

The second optimization was sorting search stacks after each GPU pass. We quickly found out that stable sort was better than unstable (quicksort) in maintaining coherency. We also tried different keys, including round number, search value, round change direction (delta), number of search steps to nearest round change (njd) and their combinations. Additionally, we modified the search loop to allow exiting the kernel only on round change; we call this *snapping*. Results of our experiments for characteristic for 2nd block for 72-round collision (72-2)

are presented in Fig. 2. We have finally chosen stable sort by search value and snapping, which gave 1.87× efficiency improvement. This was implemented using GPU radix sorting [12], and experiments have shown that sorting takes less than 1% of total computing time.

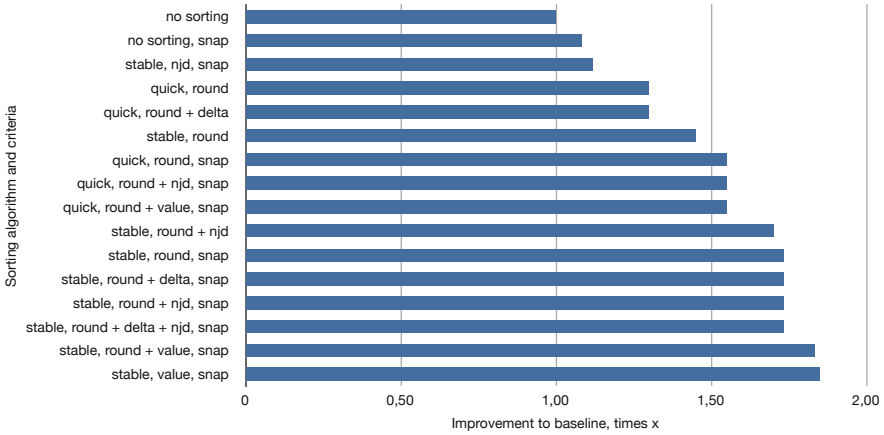

Fig. 2. Comparison of different sorting and snapping approaches

The third optimization was replacing one-loop implementation of backtracking search with a nest of 3 loops. Innermost loop iterates over search values of a single round until either all are exhausted, or a successful message word is found. The second loop works only for 16th round, and iterates over messages that must be checked. Our experiments have shown that more than 75% is spent is message check for some characteristics, so doing that in a separate loop improves performance. The outermost loop switches between rounds, and also checks thread termination condition. As search can be exited from outermost loop only, this ensures automatic snapping. On 75-1 characteristic, triple loop gave 1.25× improvement compared to stable sort and snapping only. Together, triple loop and stable sort give more than 2-fold performance improvement.

Other optimizations included using constant and shared on-chip GPU memory. Quite unexpectedly, using on-chip shared memory gave only 2.5% performance improvement (1.025×). This indicates that previous optimizations did a good job of improving memory access coherency, so that most memory accesses hit the GPU cache available on Fermi GPUs. Using constant memory gave additional improvement of about 9%. Effects of individual optimizations and all of them combined are presented in Table 3. The final version uses all of the optimizations described above, and has efficiency of 63% on 75-1 in the short run. For the long run, efficiency is lower, but still remains above 50%, which we consider sufficient for our purposes.

Table 3. Effects of Different Optimizations on Performance of GPU Backtracking Search

Optimization	**Effects**
Characteristic search modification	1.8×
Stable sort by value + snap	1.87×
Triple loop	1.25×
Other	1.12×
Total	**4.2**×

We expected our application to run for a long time, so checkpoints were used. And as the number of available GPUs was expected to fluctuate significantly, our checkpointing scheme makes it possible to resume from a checkpoint with number of processes different from what was used to save it. As processes are independent, each process just writes its search stacks to a file independently at fixed time intervals, every hour by default. Cooperation is only needed to resume from a checkpoint.

5 Results

Final computation of 1st and 2nd collision blocks were performed at GPU partition of "Lomonosov" supercomputer installed in Research Computing Center, Moscow State University (RCC MSU). Each GPU node has 2 NVidia Fermi X2070 GPUs with 6 GB RAM, of which only 5.25 GB is available because of ECC. As the GPU partition was still in beta stage, not all GPU nodes were available, and the number of available nodes fluctuated. Characteristics used to search for messages are presented in [3].

The search factor for the 1st block was 2^{58}. 264 GPUs were used, and the computation took 11000 seconds. $2^{54.06}$ nodes were actually traversed. Here, "node" is either a search step or message check round; the latter requires 2.5× more computations than the former. About 40% of nodes were check rounds.

The search factor for the 2nd block was $2^{63.01}$ nodes. The computation started with 320 GPUs and finished with 512 GPUs, with 455 GPUs being used on average. It took 1904252 seconds, or 22 days and 45 minutes. $2^{61.92}$ nodes were actually traversed, about 58.8% of them were check rounds. We achieved 52% efficiency (of GPU peak performance). The resources required were really enormous: were all 1554 GPUs available, the computation would still take about a week.

The actual number of nodes traversed were smaller than estimates. It was 16 times smaller for the 1st block and 2 times smaller for the 2nd block. Were it not the case, the entire collision search would have taken 1.5 month.

We also compared the GPU code with our previous CPU implementation [13]. A single Intel Nehalem CPU core traverses about $2^{27.85}$ search nodes per second. A single check round requires 2.5× more operations than a search node; that is, our 22-day run is equivalent to $2^{62.83}$ search nodes, or $2^{33.14}$ search nodes

traversed per second per GPU. Based on those numbers, a single GPU is 39× faster than a single CPU core, or 9.75× faster than a 4-core CPU, i.e. a single CPU socket. The number varies slightly from one characteristic to the other, but the order remains the same. This means that our computation would have taken the same time if done on 17745 CPU cores, which is not much larger than the number of cores used for the previous computation. Obtaining this many "Lomonosov"'s cores for 3 weeks would be problematic. The GPU partition, however, wasn't oversubscribed, so we could easily use all the GPUs available.

The 75-round reduced SHA-1 collision we built is presented in Table 4.

Table 4. 75-Round Reduced SHA-1 Collision

i	Message 1, Block 1				Message 1, Block 2			
1–4	F01EE8EE	BDDFF313	B2F59EE4	BB37F2BB	F072633F	0D32226A	DFF74459	98507743
5–8	2F472A36	1C052F6A	96403EF0	F144298B	EEFE63DD	FE10D5C5	AFE33902	EF74984E
9–12	DAF5519C	7A90DD71	2BF3718E	A7E3DE6D	350272F7	DB382ABC	155B0414	B800179D
13–16	EFFA975E	9B00AA95	6056E3EE	2BA4483A	18ECD4BC	15497213	1505284C	60C4F869
i	Message 2, Block 1				Message 2, Block 2			
1–4	001EE884	3DDFF353	22F59E94	0B37F2E8	00726355	8D32222A	4FF74429	28507710
5–8	1F472A3E	1C052F29	46403E82	4144299B	DEFE63D5	FE10D586	7FE33970	5F74985E
9–12	2AF551FE	BA90DD33	2BF371BE	47E3DE2F	C5027295	1B382AFE	155B0424	580017DF
13–16	CFFA973E	7B00AAD4	4056E3BE	EBA4487B	38ECD4DC	F5497252	3505281C	A0C4F828
i	Colliding Hash Values							
1–5	3DF7F21E 130079F3 C2E6EFFF FD9C4141 9AA8723A							

6 Conclusion

We have proposed a GPU implementation of SHA-1 collision search using the method of characteristics. Based on our previous work and with GPU optimizations proposed, we were able to achieve 50% computational efficiency and 39× acceleration compared to a single CPU core. Using our implementation on a cluster of GPUs, we have found a collision for 75-round reduced SHA-1, which is world's best result in terms of number of rounds for SHA-1 as of February 2012.

However, as search complexity increases 8× with each additional round, searching for collisions with larger number of rounds would require modifications to the method of characteristics. While we are working in this direction, it is too early to talk about results.

Acknowledgements. We are thankful to Research Computing Center of Lomonosov Moscow State University for providing us with access to "Lomonosov" supercomputer. We are also thankful to "Lomonosov" support team and personally Anton Korzh for promptly resolving issues which appeared our using of the cluster. This work was supported by T-Platforms, Russian Fund for Basic Research (RFBR) grant 11-07-93960-SAR-a and CUDA Center of Excellence at Moscow State University.

References

1. Teat, C., Peltsverger, S.: The security of cryptographic hashes. In: Proceedings of the 49th Annual Southeast Regional Conference, pp. 103–108. ACM (2011)
2. National Institute of Standards and Technology (NIST). FIPS-180-2: Secure Hash Standard (August 2002), http://www.itl.nist.gov/fipspubs/
3. Grechnikov, E.A., Adinetz, A.V.: Collision for 75-step SHA-1: Intensive Parallelization with GPU // Cryptology ePrint Archive: Report 2011/641, http://eprint.iacr.org/2011/641
4. Dobbertin, H.: Cryptanalysis of MD4. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 53–69. Springer, Heidelberg (1996)
5. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
6. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
7. Chabaud, F., Joux, A.: Differential Collisions in SHA-0. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 56–71. Springer, Heidelberg (1998)
8. Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., Jalby, W.: Collisions of SHA-0 and Reduced SHA-1. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 36–57. Springer, Heidelberg (2005)
9. De Cannière, C., Rechberger, C.: Finding SHA-1 Characteristics: General Results and Applications. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 1–20. Springer, Heidelberg (2006)
10. De Cannière, C., Mendel, F., Rechberger, C.: Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 56–73. Springer, Heidelberg (2007)
11. Adinetz, A.V.: NUDA Programmer's Guide, http://nuda.sf.net
12. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Proceedings of the 2010 International Conference on Management of Data (SIGMOD 2010), pp. 351–362. ACM, New York (2010)
13. Grechnikov, E.A.: Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics. Cryptology ePrint Archive: Report 2010/413, http://eprint.iacr.org/2010/413