Построение коллизии для 75-шаговой версии хэш-функции SHA-1 с использованием ГПУ-кластеров

Е.А. Гречников[1], А.В. Адинец[2,3]

МГУ им. М. В. Ломоносова, механико-математический факультет[1],
Научно-исследовательский вычислительный центр МГУ им. М. В. Ломоносова[2],
Объединённый институт ядерных исследований[3]

Криптографические хэш-функции, в частности, SHA-1, в настоящее время широко используются в современных информационных технологиях. Важным свойством таких функций является устойчивость к коллизиям, т.е. сложность построения двух различных входных сообщений, значения хэш-функции от которых совпадают. Мы развиваем метод разностных (дифференциальных) атак для поиска коллизий хеш-функции SHA-1 и её сокращённых вариантов. В данной статье описывается реализация метода характеристик для хэш-функции SHA-1 на кластере из графических процессоров. Использование различных оптимизаций позволило достичь эффективности ГПУ-реализации 50%, и ускорения в 39 раз по сравнению с реализацией на одном ядре ЦПУ. Оптимизации также включали модификацию метода поиска характеристик. На текущий момент наши улучшения метода и использование ГПУ-раздела суперкомпьютера «Ломоносов» позволили найти коллизию для SHA-1, сокращённой до 75 шагов (полная функция состоит из 80), что является мировым рекордом.

1. Введение

В современной криптографии широко используются различные хеш-функции. Хеш-функция есть функция, отображающая сообщения (строки из 0 и 1) произвольной длины в последовательности из 0 и 1 ограниченной длины — значения хеш-функции или хеш-значения. Значение хеш-функции есть как бы отпечаток всего сообщения, а роль его подобна отпечаткам пальцев в процессе идентификации.

Для любой хеш-функции существуют сообщения, имеющие одинаковые хеш-значения, ведь множество сообщений бесконечно, а множество хеш-значений конечно. Если удаётся явно построить два различных сообщения, имеющие совпадающее хеш-значение, иначе говоря, построить коллизию, то хеш-функция считается скомпрометированной, что имеет разрушительные последствия для некоторых криптографических приложений.

Хеш-функция, обозначаемая SHA-1 (Secure Hash Algorithm), преобразует сколь угодно длинные сообщения (в стандарте максимальная длина равна $2^{64} - 1$ бит) в 160-битные хеш-значения. Эта хеш-функция была опубликована NIST (National Institute of Standards and Technology) в США в 1995 г. и в настоящее время используется как важная составная часть различных государственных и индустриальных стандартов безопасности, таких как электронная цифровая подпись, аутентификация пользователей, обмен ключами и построение псевдослучайных последовательностей. SHA-1 внедрена почти во все коммерческие системы безопасности.

В течение ряда лет в различных странах предпринимаются активные попытки компрометации функции SHA-1, и, хотя коллизия для этой функции не построена, исследователи продвинулись в этой деятельности достаточно далеко. В настоящее время NIST проводит конкурс на построение новой хеш-функции, которая смогла бы заменить SHA-1. Предполагается ввести новую функцию в действие в 2012 году.

Криптоаналитические задачи, как правило, достаточно легко распараллеливаются на любое доступное количество вычислительных ресурсов. Поэтому тем более логично ис-

пользовать ГПУ для решения таких задач. И хотя ГПУ активно используются для задачи подбора пароля [1], нам не встречались случаи использования ГПУ для поиска коллизий для хэш-функций.

Наша цель — построить коллизию хеш-функции SHA-1. На настоящий момент с использованием ГПУ-раздела суперкомпьютера «Ломоносов» мы установили мировой рекорд [3], построив коллизию для хеш-функции, устроенной аналогично SHA-1, но с меньшим значением одного из параметров схемы: 75 шагов вместо 80 шагов в SHA-1.

Вклад данной статьи можно кратко описать следующим образом:

- Нами впервые предложена реализация поиска коллизий для хэш-функций, использующая графические процессоры
- При помощи нашей реализации построена коллизия для 75-раундовой версии SHA-1, что на данный момент является мировым рекордом

Данная статья организована следующим образом. В разделе 2 описывается устройство хэш-функции SHA-1. В разделе 3 описывается устройство разностных атак, в том числе применительно к хэш-функции SHA-1, а в разделе 4 описывается используемый алгоритм поиска характеристики. Особенности ГПУ-реализации приводятся в разделе 5, а в разделе 6 описываются проведённые расчёты и приводится полученная коллизия. Наконец, раздел 7 содержит подведение итогов и благодарности.

2. Хэш-функция SHA-1

Используемые в настоящей статье в формулах обозначения приведены в таблице 1 Хеш-функция SHA-1 [2] устроена следующим образом. В конец входного сообщения определённым образом добавляется несколько бит, зависящих от длины сообщения, так, чтобы результат состоял из нескольких 512-битных блоков $M_1, \ldots, M_k$. 160-битное хеш-значение представляет из себя набор из 5 32-битных переменных, для вычисления которых к начальному набору H_0 последовательно «подмешиваются» блоки M_i следующим образом: для $i = 1, \ldots, k$ вычисляется $H_i = H_{i-1} + g(M_i, H_{i-1})$. Константа H_0 является частью стандарта. Хеш-значением входного сообщения является H_k.

Таблица 1. Обозначения, используемые в данной статье

Обозначение	Описание
X	32-битное число, связанное с первым сообщением
X^*	32-битное число, связанное со вторым сообщением
X^2	пара 32-битных чисел (X, X^*)
$X \oplus Y$	побитовое исключающее ИЛИ (XOR)
$X + Y$	сложение по модулю 2^{32}
$[X]_i$	i-й бит числа X ($i = 0$ — младший бит)
$X \lll i$	циклический сдвиг влево на i бит
$X \ggg i$	циклический сдвиг вправо на i бит

Для нахождения коллизии достаточно построить два набора $(M_1, \ldots, M_k)$ и $(M_1^*, \ldots, M_k^*)$ одной и той же длины, для которых $H_k = H_k^*$. Поскольку длины одинаковы, то биты, добавляемые в конец сообщений при вычислении SHA-1, также будут одинаковы, так что последующие значения H_i и H_i^* совпадут.

Сжимающая функция $g(M, H)$ строит новое 160-битное значение по старому 160-битному значению H и 512-битному блоку сообщения M. Входные и выходные данные представляются как 32-битные переменные, $H = (A_0, B_0, C_0, D_0, E_0)$, $M = (M_0, \ldots, M_{15})$, $g(M, H) =$

$(A_{80}, B_{80}, C_{80}, D_{80}, E_{80})$. Вычисление сжимающей функции состоит из двух частей: расширения сообщения и обновления состояния.

Расширение сообщения линейно дополняет 16 переменных M_i до 80 переменных W_i, которые будут использоваться при обновлении состояния:

$$\begin{aligned} W_i &= M_i, \quad 0 \leqslant i < 16; \\ W_i &= (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, \quad i \geqslant 16. \end{aligned}$$

Обновление состояния состоит из 80 шагов следующего вида, изображённого на рис. 1.

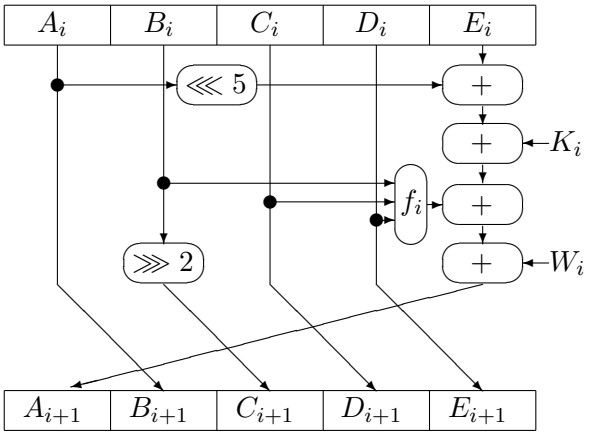

Рис. 1. Вид одного раунда хэш-функции SHA-1

Здесь K_i — константы, а f_i — следующие функции:

- на первых 20 шагах $K_i =$ `0x5A827999`,

$$f_i(b, c, d) = f_{IF}(b, c, d) = (b \wedge c) \vee (\overline{b} \wedge d);$$

- на вторых 20 шагах $K_i =$ `0x6ED9EBA1`,

$$f_i(b, c, d) = f_{XOR}(b, c, d) = b \oplus c \oplus d;$$

- на третьих 20 шагах $K_i =$ `0x8F1BBCDC`,

$$f_i(b, c, d) = f_{MAJ}(b, c, d) = (b \wedge c) \vee (b \wedge d) \vee (c \wedge d);$$

- на последних 20 шагах $K_i =$ `0xCA62C1D6`,

$$f_i(b, c, d) = f_{XOR}(b, c, d) = b \oplus c \oplus d.$$

Легко видеть, что $B_i = A_{i-1}$, $C_i = A_{i-2} \ggg 2$, $D_i = A_{i-3} \ggg 2$, $E_i = A_{i-4} \ggg 2$. Таким образом, достаточно рассматривать только переменные A_i; начальные значения $(A_0, \ldots, E_0)$ задают $(A_{-4}, \ldots, A_0)$, конечные значения $(A_{80}, \ldots, E_{80})$ получаются из $(A_{76}, \ldots, A_{80})$.

Поскольку вычисления для функции SHA-1 достаточно трудоёмки, для демонстрации новых и оптимизации существующих методов принято использовать так называемые сокращённые варианты, которые отличаются от «полной» функции меньшим количеством шагов в сжимающей функции, сами шаги при этом остаются теми же.

3. Разностные атаки

Если взять в качестве блока сообщения случайный набор бит, то хеш-функция SHA-1 выдаст случайное 160-битное хеш-значение с приблизительно равномерным распределением среди всех возможных 160-битных строк; таким образом, если взять два случайных блока, то вероятность того, что они образуют коллизию, составляет примерно 2^{-160}, что, разумеется, слишком мало для практических целей.

Разностные атаки эволюционировали со временем, различные этапы (в том числе атаки на другие хеш-функции, построенные по той же схеме) отмечены в работах [4] (MD4), [7] (35 шагов SHA-0), [8] (полный SHA-0), [5] (MD5), [6] (58 шагов SHA-1), [9] (64 шагов SHA-1), [10] (70 шагов SHA-1). Мы усовершенствуем метод характеристик, описанный в двух последних работах.

Ключевая идея разностных атак заключается в том, чтобы вести перебор не по всем возможным парам сообщений, а только по парам сообщений с фиксированными разностями по модулю 2 $\delta M_i = M_i \oplus M_i^*$, за что атаки и получили такое название. Помимо фиксирования δM_i, полезным также оказывается фиксировать отдельные биты в M_i, а также δA_i и A_i. Более точно, назовём *характеристикой* набор из $(80+85)\cdot 32$ элементарных условий на пары бит $([W_i]_j, [W_i^*]_j)$ и $([A_i]_j, [A_i^*]_j)$. Каждое элементарное условие какие-то из 4 возможных вариантов разрешает, остальные запрещает; все 2^4 возможных элементарных условий собраны в следующей таблице вместе с условными обозначениями для них.

Таблица 2. Обозначения условий на биты, применяемые при записи характеристик

∇_i	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
`*`	✓	✓	✓	✓
`-`	✓	–	–	✓
`x`	–	✓	✓	–
`0`	✓	–	–	–
`u`	–	✓	–	–
`n`	–	–	✓	–
`1`	–	–	–	✓
`#`	–	–	–	–
`3`	✓	✓	–	–
`5`	✓	–	✓	–
`7`	✓	✓	✓	–
`A`	–	✓	–	✓
`B`	✓	✓	–	✓
`C`	–	–	✓	✓
`D`	✓	–	✓	✓
`E`	–	✓	✓	✓

Множество пар (X, X^*), удовлетворяющих всем 32 условиям на соответствующую переменную, будем обозначать ∇X. Пусть известна некоторая характеристика и мы хотим организовать перебор по этой характеристике для нахождения коллизии. Будем последовательно подбирать $M_0^2 = (M_0, M_0^*)$, затем M_1^2 и так далее. На каждом новом шаге мы выбираем очередной вариант для пары M_i^2 с учётом условий из характеристики, вычисляем пару A_{i+1}^2, проверяем условия из характеристики, пока не исчерпаем все возможности или не найдём нужной пары. Если пара найдена, продвигаемся на шаг вперёд, если возможности исчерпаны, отступаем на шаг назад. После того, как мы подобрали M_{15}^2, сообщение становится полностью определено, так что последующие шаги состоят только из проверок условий.

Число вариантов пары A^2_{i+1}, удовлетворяющих условиям характеристики, может быть меньше, чем у соответствующей пары M^2_i. В таком случае мы перебираем A^2_{i+1} и вычисляем M^2_i; при фиксированных предыдущих значениях A_i, A^*_i соответствие взаимно-однозначно.

Перебор будем вести до нахождения первой коллизии или до исчерпания пространства перебора. Предположим, что перебор завершится успехом, и попробуем оценить его сложность. Для этого введём несколько определений.

Набор $(W^2_0, \ldots, W^2_i)$ назовём непротиворечивым, если его можно продолжить до полной пары расширенных сообщений, удовлетворяющих характеристике. *Степень свободы сообщения на шаге i (обозначается $\tilde{F}_W(i)$)* – это число непротиворечивых наборов $(W_0, \ldots, W_i)$, продолжающих непротиворечивый набор $(W_0, \ldots, W_{i-1})$. При $i \geqslant 16$, очевидно, $\tilde{F}_W(i) = 1$. Если в характеристике условия на $W_{16}, \ldots, W_{79}$ тривиальны, то при $i < 16$ степень свободы сообщения на шаге i равна просто $|\nabla W_i|$. В общем случае условия характеристики на $W_{16}, \ldots, W_{79}$ влекут за собой какие-то линейные соотношения на отдельные биты $[M_i]_j$; если есть m таких независимых соотношений, то $\tilde{F}_W(i)$ равно $\frac{|\nabla W_i|}{2^m}$.

Можно также посчитать $\tilde{F}_A(i) = |\nabla A^2_{i+1}|$. Если $\tilde{F}_A(i) \geqslant \tilde{F}_W(i)$, то мы перебираем M^2_i и вычисляем A^2_{i+1}; иначе мы перебираем A^2_{i+1} и вычисляем M^2_i. В первом случае положим $F_W(i) = \tilde{F}_W(i)$, во втором положим $F_W(i) = \frac{\tilde{F}_A(i)}{2^m}$; тогда $F_W(i)$ — число потомков вершины дерева перебора на шаге i с поправкой на неявные линейные ограничения.

В определении хеш-функции значение A_{i+1} вычисляется на каждом шаге по значениям $A_{i-j}, 0 \leqslant j \leqslant 4$, и W_i. Для оценок введём величины, которые характеризуют шаг хеш-функции; временно забудем, что A получаются по сообщениям, и рассмотрим вероятностное пространство, в котором $A_{i-j}, 0 \leqslant j \leqslant 4$, и W_i — независимые величины, удовлетворяющие соответствующим ограничениям из характеристики.

Неконтролируемая вероятность на шаге i (обозначается $P_u(i)$) — это вероятность того, что результат шага i будет удовлетворять характеристике при условии, что на всех предыдущих шагах состояния и расширенные сообщения удовлетворяют характеристике. То есть при $\tilde{F}_A(i) \geqslant \tilde{F}_W(i)$

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

а при $\tilde{F}_A(i) < \tilde{F}_W(i)$

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

Контролируемая вероятность на шаге i (обозначается $P_c(i)$) – это вероятность того, что найдется хотя бы одна пара W^2_i, удовлетворяющая характеристике, такая что результат шага i будет удовлетворять характеристике при условии, что на всех предыдущих шагах состояния удовлетворяют характеристике. То есть

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

(Контролируемая вероятность не меняется при перемене ролей A и W.)

Можно с некоторой погрешностью оценить сложность успешного перебора, считая приближённо, что все шаги независимы. А именно, на i-м шаге в среднем придётся обойти $N_S(i)$ вершин дерева перебора, где:

- $N_S(80) = 1$ (достаточно найти одну коллизию),
- $N_S(i) = \max\left\{\frac{N_S(i+1)}{F_W(i)P_u(i)}, \frac{1}{P_c(i)}\right\}$ (с одной стороны, в среднем у вершины дерева перебора будет $F_W(i)$ потомков, среди которых доля $P_u(i)$ будет давать вершину следующего уровня; с другой стороны, с вероятностью $P_c(i)$ вершина дерева перебора безнадёжна для получения вершин следующего уровня).

Величину

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

зависящую только от характеристики, будем называть *фактором объёма перебора* характеристики. Чем меньше фактор объёма перебора, тем характеристика лучше.

4. Подбор характеристики

Подбор характеристики состоит из трёх этапов.На первом этапе выбирается *линейная* характеристика, состоящая только из условий `-` и `x` (что эквивалентно, фиксирует все δW_i, δA_i, но не отдельные биты). Для этого рассматривается *линеаризация* хеш-функции, при которой структура функции сохраняется, но все нелинейные операции (включая сложение по модулю 2^{32}) заменяются на подходящие линейные. Идея заключается в том, чтобы найти характеристику с возможно меньшим количеством условий `x`; поскольку любая операция на совпадающих входах даёт совпадающие выходы, то хеш-функция может «разойтись» с линеаризацией только из-за отличающихся бит, соответствующих `x` в линеаризации. Линейные операции достаточно просты. Задачу о нахождении линейной характеристики с малым числом условий `x` можно сформулировать как поиск вектора малого веса в некотором линейном коде, теория кодов даёт решение этой задачи.

Мы строим коллизию, состоящую из двух блоков. Характеристика для каждого блока своя, но строится по одной и той же линейной характеристике. Хеш-значение вычисляется как $H_2 = H_1 + g(M_2, H_1) = H_0 + g(M_1, H_0) + g(M_2, H_1)$, $H_2^* = H_0 + g(M_1^*, H_0) + g(M_2^*, H_1^*)$. Линейная характеристика задаёт $g(M_1, H_0) \oplus g(M_1^*, H_0)$ и $g(M_2, H_1) \oplus g(M_2^*, H_1^*)$; поскольку она одна и та же для обоих блоков, то фиксацией значений различающихся бит можно добиться того, чтобы разность выходов сжимающей функции на первом блоке была противоположна по знаку разности выходов сжимающей функции на втором блоке. Тогда будет $H_2 = H_2^*$.

Второй этап начинается с отбрасывания всех условий линейной характеристики на первых 12 шагах на A_i и подстановки начальных условий для $A^2_{-4}, \ldots, A^2_0$. Для первого блока начальным условием является константа H_0, для второго блока — результат вычислений по первому блоку (таким образом, строить характеристику для второго блока можно только после того, как подобран первый блок коллизии). Кроме того, условия вида `xx` (на две подряд идущих пары бит) можно заменить на `-x`, если различие в старшем из этих бит может быть обусловлено переносом из младшего (это не всегда так из-за циклических сдвигов). На втором этапе следует найти какой-нибудь «путь» (согласованный набор условий) от начальных условий до линейной характеристики. Для этого будем выбирать случайные позиции в A_i^2, на которых ещё нет ограничений, накладывать ограничение `-` и вычислять, какие дополнительные ограничения будут выполнены как следствие всех текущих. Кроме того, при появлении ограничений `x` в A_i^2 (два соответствующих друг другу битов различаются) оказывается полезным дополнительно фиксировать значения различающихся битов (к примеру, требовать, чтобы бит первого сообщения был нулевым, а соответствующий бит второго сообщения был единичным). При обнаружении противоречивых условий возвращаемся к последней фиксации условия `x` и используем вторую возможную фиксацию значения бит. Второй этап заканчивается, когда все условия имеют вид одного из `-xun01`.

На третьем этапе последовательно улучшается фактор объёма перебора найденной характеристики. Для этого перебираем все позиции в характеристике, рассматриваем возможные усиления условия на позиции, вычисляем дополнительные ограничения, появляющиеся как следствие усиления, вычисляем фактор объёма перебора для новой характеристики. После окончания перебора фиксируем то из условий, для которого фактор объёма перебора был минимален.

В подборе характеристики отметим только несколько важных моментов:

- Величины F_W, P_u, P_c вычисляются последовательно от младших битов к старшим при помощи перебора элементарных условий и возможных переносов.

- Следствия от введения нового условия для одного шага вычисляются в два прохода, вначале от младших бит к старшим запоминаются возможные переносы, затем с учётом переносов вычисляются возможные следствия ограничений. Эта процедура быстрая, но находит не все возможные дополнительные ограничения. Поэтому для поиска следствий мы используем цикл по позициям бит, в котором временно фиксируем возможные значения бит и смотрим, не находит ли быстрая процедура противоречий. На третьем этапе цикл включает все биты, на втором этапе — только биты, «соседние» с изменёнными.

- Для повышения эффективности перебота по характеристике на ГПУ важна *когерентность*, т.е. одинаковость путей исполнения ядра в соседних потоках. Когерентность оказывается выше, если сильные ограничения сконцентрированы в начале характеристики. Поэтому на втором этапе мы выбираем позиции в начале с несколько меньшей вероятностью (чтобы условия - в среднем тяготели к шагам с большим номером). Подобная оптимизация даёт повышение производительности поиска на ГПУ более чем на 80%.

5. Реализация поиска на ГПУ

Перейдём к вопросам перебора по готовой характеристике. Именно он является наиболее вычислительно сложной частью задачи. Сначала отметим некоторые особенности, которые не зависят от выбора платформы реализации:

- Итоговые характеристики всегда состоят только из условий, входящих в список `-xun01`. Следовательно, набор условий на каждую пару 32-битных переменных эквивалентен $X \oplus X' = a$, $X \wedge b = c$, где a, b, c — некоторые константы. Предвычисление a, b, c позволяет быстро проверять пару переменных.

- Величины A_i на двух последних шагах не участвуют в вычислениях f_i, а потому вместо двух условий из предыдущего пункта достаточно проверять, что $X - X' = a$. Кроме того, при вычислении первого блока условия на двух последних шагах можно вообще игнорировать, поскольку любую возможную разность можно будет сократить на втором блоке без увеличения количества необходимых условий. Значения $N_S(i)$ в приводимых нами таблицах скорректированы с учётом этого.

- Эффективный перебор всех пар X таких, что $X \wedge b = c$: первый элемент множества есть $X = c$, следующий после x задаётся формулой $(((x \vee b) + 1) \wedge \overline{b}) + c$, перебор заканчивается, когда формула из-за переполнения даёт c.

- Линейные соотношения на M_i, возникающие из условий на W_k при больших k, могут либо выражать какой-то бит $[M_i]_j$ через какие-то биты предыдущих сообщений, либо давать какую-то связь между двумя или более битами $[M_i]_j$. Первый случай меняет только то, что числа a, b, c зависят не только от характеристики, но и от предыдущих сообщений. Соотношений, соответствующих второму случаю, мало, от них можно избавиться, просто наложив дополнительные искусственные ограничения, которые не меняют существенно фактор объёма перебора.

Перебор естественным образом разделяется на *генерацию*, где выполняется собственно перебор и генерация пар сообщений, и *проверку*, где характеристика проверяется для оставшихся раундов для построенной пары сообщений. Генерацию можно рассматривать как поиск с возвратом, и проверка просто добавляется как вызов функции в последний раунд

генерации. Генерация разделяется на часть, выполняемую на хосте, и часть, выполняемую на устройстве. На хосте дерево перебора раскрывается до определённой глубины, чтобы сгенерировать достаточное количество *стеков поиска* для задействования ресурсов ГПУ-параллелизма. Эта глубина сейчас задаётся вручную для каждой характеристики. Слишком маленькая глубина приведёт к недостаточному ресурсу параллелизма, а при слишком большой глубине стеки поиска просто не поместятся в память ГПУ. В результате экспериментов мы установили, что примерно 100 тыс. стеков на ГПУ достаточно для эффективного использования его ресурсов.

Во время ГПУ-части поиск по всем стекам идёт параллельно на большом количестве ГПУ. Если поиск для стека завершён, после завершения ГПУ-ядра стек удаляется. Новые стеки во время ГПУ-части не генерируются. Главное ГПУ-ядро реализует поиск с возвратом и проверку построенного сообщения. В этом ядре каждый ГПУ-поток обрабатывает свой стек поиска. Во время вызова ядра каждый поток выполняет фиксированное число итераций поиска, после чего завершает свою работу. Главное ядро также собирает статистические данные по количеству перебранных вершин, общему количеству раундов проверки, а также максимальной достигнутой при проверке глубине. Между вызовами ядра выполняется проверка на достижение необходимой глубины и сжатие стека поиска.

Вычисление на распределённом кластере реализовано при помощи MPI. Каждый MPI-процесс работает только с одним ГПУ. Используется блочно-циклическое распределение стеков поиска между процессами. Во время хост-части каждый процессор генерирует все стеки, после чего оставляет себе только те, что ему принадлежат. В первой реализации после каждого вызова ГПУ-ядра использовался глобальный барьер, в том числе для обмена статистическими данными. Однако эксперименты показали, что ввиду разбалансировки загрузки, возникающей вследствие неравномерности завершения работы над стеками, для некоторых характеристик ГПУ простаивают 50% времени. Поэтому в текущей реализации от глобального барьера мы отказались. Вместо этого мастер-процесс (с рангом 0) порождает 2 дополнительных потока, один для сбора статистики, и один для показа статистики через заданный промежуток времени (обычно 1 минута). Все процессы отсылают собственную статистику потоку сбора статистики в процессе с номером 0.

Для реализации на ГПУ использовался язык Nemerle и система NUDA (Nemerle Unified Device Architecture) [11], набор расширений этого языка для программирования ГПУ. Система NUDA была выбрана вследствие её свободного распространения и реализации высокоуровневой поддержки ГПУ. При этом низкоуровневые детали, такие как передача данных и параметров ядра, скрыты от программиста и обрабатываются самой системой. Для генерации ГПУ-кода для циклов и взаимодействия с ГПУ NUDA использует технологию OpenCL.

В нашей первой реализации поиск с возвратом был реализован как один цикл, в котором этапы, требующие специальной обработки, были реализованы в виде условий. Таких этапа было 2: переключение между раундами поиска, где требуется предвычислить большое количество значений, и проверка сгенерированного сообщения. Тестирование было реализовано в отдельной функции, а цикл по раундам тестирования полностью развёрнут при помощи аннотации `inline`, доступной в NUDA.

Однако производительность первой реализации была хороша только для некоторых характеристик, и очень плоха для других. Например, на 73-1 была достигнута эффективность 60% (от пиковой производительности ГПУ), в то время как на 72-2 достигнутая эффективность составляла лишь 15%. Низкая эффективность была вызвана низкой когерентностью между потоками одного варпа. Для повышения когерентности мы использовали две оптимизации.

Первая оптимизация состояла в сортировке стеков после каждого вызова основного ядра. Использование устойчивого алгоритма сортировки давало лучшие результаты по сравнению с неустойчивыми алгоритмами (например, быстрой сортировкой), поскольку она со-

храняла порядок значений с одинаковыми ключами, а значит, сохраняла когерентность. Мы пробовали сортировку по разным ключам, однако в конце концов остановились на значении поиска (состоянии или сообщении) на текущем раунде. На характеристике 72-2, просто устойчивая сортировка давала 45%-ый прирост производительности по сравнению с изначальной реализацией. Для реализации устойчивой сортировки на ГПУ использовался алгоритм побитовой сортировки [12], и эксперименты показали, что время сортировки стеков достаточно мало по сравнению с временем выполнения основного ядра. Сортировка сочеталась с модификацией цикла поиска сообщения: выход из цикла разрешался только при переключении раунда. Сортировка по значению поиска вместе с модификацией цикла даёт прирост производительности 87%.

Второй оптимизацией была замена одинарного цикла в поиске возвратов на тройной цикл. Самый вложенный цикл выполняет перебор на одном раунде и проверяет соответствие характеристике, пока либо не будет найдена хорошая вершина, либо не будут перебраны все вершины в этой ветке дерева на этом раунде. Второй по вложенности цикл реализует проверку сообщения, и имеет смысл только на последнем раунде перебора. Так как более 75% времени затрачивается на проверку сообщения, это повышает производительность. Наконец, внешний цикл выполняет переходы между раундами, и проверяет условие завершения ядра. Конструкция с тройным циклом, хотя и может понижать когерентность за счёт разного количества итераций внутреннего цикла у соседних потоков, ещё и препятствует расхождению соседних потоков. В целом мы обнаружили, что использование тройного цикла совместно с устойчивой сортировкой по значению поиска даёт более чем 2-кратное увеличение производительности на характеристике 75-1 по сравнению с изначальной реализацией.

Выполнялись и другие оптимизации. Это включало использование константной памяти для хранения данных характеристик, а также использование разделяемой памяти на чипе для хранения стеков поиска. К нашему удивлению, использование разделяемой памяти улучшило производительность только на 2.5% по сравнению с хранением стеков в глобальной памяти. Это показывает, что используемый алгоритм работает на каждом мультипроцессоре с небольшим множеством адресов, которое хорошо помещается в кэш 1-го уровня на ГПУ NVidia Fermi. Финальная версия, с помощью которой проводился расчёт, использует все описанные оптимизации, включая описанную ранее модификацию алгоритма нахождения характеристики. Эффект от влияния различных оптимизаций на производительность отражён в таблице 3.

Таблица 3. Влияние различных оптимизаций на произовдительность поиска на ГПУ

Оптимизация	**Ускорение**
Модификация поиска характеристики	1.8×
Устойчивая сортировка по значению поиска	1.87×
Тройной цикл	1.25×
Прочее	1.12×
Итого	**4.2**×

Поскольку по предварительным оценкам поиск второй пары блоков должен был длиться очень долго, в приложение была добавлена поддержка контрольных точек. А поскольку ожидалось, что количество доступных узлов будет меняться, реализация контрольных точек поддерживала сохранение с одним числом процессов и последующее восстановление в другое число процессов. Поскольку во время ГПУ-части нет глобальной синхронизации, каждый процесс писал файл контрольной точки независимо от остальных, через фиксированные промежутки времени, по умолчанию каждый час.

6. Результаты вычислений

Первые эксперименты и настройка параметров выполнялись на ГПУ-кластере «ГрафИТ!», установленном в НИВЦ МГУ. Эта система состоит из 16 узлов, на каждом из которых стоит по 3 ГПУ NVidia Fermi M2050 с 3 ГБ памяти на каждом. Итого это даёт 48 ГПУ, но мы проводили расчёты не более чем на 30 из них.

Финальные расчёты для 1-го и 2-го блока коллизии выполнялись на ГПУ-разделе кластера «Ломоносов», также установленного в НИВЦ МГУ. На каждом узле ГПУ-раздела установлено 2 ГПУ NVidia Fermi X2070, с 6 ГБ памяти на каждом ГПУ. Поскольку во время расчётов система находилась в режиме бета-тестирования, не все узлы были доступны для вычислений. Использованные для расчётов характеристики в данной статье для краткости опущены, однако они доступны в [3].

Объём перебора для нахождения первого блока оценивался как 2^{58} вершин. Ввиду ограничения на сообщения, характеристика была разделена на 4 части, и только одна из них выбрана для расчёта. Расчёт выполнялся на 264 ГПУ, и занял 11000 секунд. Количество обработанных вершин дерева перебора было $2^{54.06}$. Здесь в качестве «вершин» учитываются как собственно вершины дерева, так и раунды проверки сообщения, хотя один раунд проверки требует в 2.5 раз больше вычислений, чем вершина собственно перебора. Для первого блока раундов проверки было примерно 40% от общего числа «вершин».

Объём перебора для 2-го блока оценивался как $2^{63.01}$ вершин. Перебор начался на 320 ГПУ, и несколько раз перезапускался с контрольных точек ввиду сбоев на узлах или ввода в строй дополнительных узлов с ГПУ. В конце расчёт проходил на 512 ГПУ, а в среднем число использованных ГПУ составило 455. Весь расчёт длился 1904252 секунды, или примерно 22 дня 45 минут. Было обработано $2^{61.92}$ вершин, примерно 58.8% из которых составили раунды проверки. Эффективность составила 52% от пиковой производительности всех ГПУ. Если бы нам был доступен весь кластер с 1554 ГПУ, для завершения расчёта всё равно потребовалась бы целая неделя.

Для каджого блока нам в определённом смысле «повезло», т.к. поиск занял меньше времени, чем ожидалось. Для 1-го блока «повезло» в 16 раз, а для 2-го — в 2 раза. Если бы нам не «повезло», весь расчёт занял бы полтора месяца.

Если сравнивать ГПУ-реализацию с предыдущей реализацией на кластере из обычных процессоров [13], 1 ГПУ обрабатывает данные в 39 раз быстрее, чем 1 ЦПУ. Если исходить из этого, то эквивалентное количество обычных ядер для нашего расчёта было бы 17745, что не сильно превышает количество ядер, использовавшихся для получения предыдущего результата. Однако получить столько ядер на такой промежуток времени на «Ломоносове» было бы достаточно проблематично. Спрос на ГПУ был намного меньше, и их получить в требуемом объёме было сравнительно легко.

В таблице 4 приводятся данные построенной коллизии.

7. Заключение

В данной статье предложена реализация поиска коллизий для хэш-функции SHA-1 при помощи метода характеристик на кластерах из ГПУ. Основываясь на результатах предыдущей работы, а также с помощью предложенных оптимизаций удалось достигнуть эффективности расчёта 50%. При помощи разработанной реализации удалось построить коллизию для 75-раундовой версии хэш-функции SHA-1, что в настоящее время является мировым рекордом для данной хэш-функции.

Поскольку сложность расчёта с каждым следующим раундом возрастает примерно в 8 раз, дальнейшее увеличение количества используемых раундов потребует хотя бы частичного пересмотра используемого метода. Предварительные работы в этом направлении ведутся, но о результатах говорить пока рано.

Мы выражаем благодарность НИВЦ МГУ им. М.В.Ломоносова за предоставленные

Таблица 4. Пример 75-шаговой коллизии SHA-1

i	Сообщение 1, первый блок				Сообщение 1, второй блок			
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	Сообщение 2, первый блок				Сообщение 2, второй блок			
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	XOR-разности в двух блоках совпадают							
1–4	F000006A	80000040	90000070	B0000053	F000006A	80000040	90000070	B0000053
5–8	30000008	00000043	D0000072	B0000010	30000008	00000043	D0000072	B0000010
9–12	F0000062	C0000042	00000030	E0000042	F0000062	C0000042	00000030	E0000042
13–16	20000060	E0000041	20000050	C0000041	20000060	E0000041	20000050	C0000041
i	Совпадающие хеш-значения							
1–5	3DF7F21E	130079F3	C2E6EFFF	FD9C4141	9AA8723A			

вычислительные ресурсы. Кроме того, мы выражаем благодарность группе поддержки суперкомпьютера «Ломоносов» и лично Антону Коржу за оперативное решение вопросов, возникающих во время использования кластера.

Литература

1. Teat C., Peltsverger S. The security of cryptographic hashes. // Proceedings of the 49th Annual Southeast Regional Conference, ACM, 2011, pp. 103–108.

2. National Institute of Standards and Technology (NIST). FIPS-180-2: Secure Hash Standard, August 2002. Available online at `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. Available online at `http://eprint.iacr.org/2011/641`.

4. Hans Dobbertin, *Cryptanalysis of MD4*, Fast Software Encryption, LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 53–69.

5. Xiaoyun Wang and Hongbo Yu, *How to Break MD5 and Other Hash Functions*, Eurocrypt 2005.

6. Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, *Finding Collisions in the Full SHA-1*, in Proceedings of CRYPTO, LNCS 3621, Springer, 2005, pp. 17–36.

7. Florent Chabaud, Antoine Joux, *Differential Collisions in SHA-0*, CRYPTO 1998.

8. Eli Biham, Rafi Chem, Antoine Joux, Patrick Carribault, Christophe Lemuet, William Jalby, *Collisions of SHA-0 and Reduced SHA-1*, Eurocrypt 2005.

9. Christophe De Cannière, Christian Rechberger, *Finding SHA-1 Characteristics: General Results and Applications*, In Proceedings of ASIACRYPT, LNCS 4284, pp. 1–20, Springer, 2006.

10. Christophe De Cannière, Florian Mendel, Christian Rechberger, *Collisions for 70-Step SHA-1: On the Full Cost of Collision Search*, In Proceedings of Selected Areas in Cryptography, LNCS 4876, pp. 56–73, Springer, 2007.

11. Andrew V. Adinetz. *NUDA Programmer's Guide*. URL: `http://nuda.sf.net`.

12. Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. *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 '10), pages 351-362. ACM, New York, NY, USA, 2010.

13. E. A. Grechnikov. *Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics*. Cryptology ePrint Archive: Report 2010/413, available at `http://eprint.iacr.org/2010/413`.