Реализация алгоритма шифрования "Кузнечик"с помощью технологии NVIDIA CUDA

А.И. Борисов[1], Е.В. Мясников[1]

[1]Самарский национальный исследовательский университет им. академика С.П. Королева, Московское шоссе 34А, Самара, Россия, 443086

Аннотация. В статье рассматриваются различные варианты реализации блочного алгоритма шифрования "Кузнечик" с помощью технологии NVIDIA CUDA. В качестве основы используется реализация на основе таблиц поиска. Исследуется влияние размера блока нитей и места расположения таблиц на скорость шифрования. Лучшие результаты получены при хранении таблиц поиска в глобальной памяти при размере блока в 384 нити. Пиковая скорость шифрования на графическом процессоре NVIDIA GeFOice 850M (версия памяти DDR3) достигает 2,7 ГБ/с.

1. Введение

В настоящее время криптографическая защита информации является неотъемлемой частью современной IT-инфраструктуры. Постоянно возрастают как объемы обрабатываемой информации, так и вычислительная мощность компьютеров. Соответственно, растут требования как к стойкости алгоритмов, так и к их скорости.

Идея использования графических процессоров для ускорения существующих алгоритмов шифрования появилась практически одновременно с самой идеей их использования для вычислений общего назначения. Как известно, наибольший эффект от применения графических процессоров достигается при возможности полного распараллеливания задачи. Неудивительно, что наиболее заметные результаты в этой сфере были получены для блочного шифрования в режимах ECB (электронной кодовой книги) и CTR (режим гаммирования), поскольку блоки открытого текста в данном случае обрабатываются независимо.

К настоящему времени написан целый ряд работ, посвященный ускорению блочных шифров с помощью CUDA [3, 4]. Наиболее популярным алгоритмом является американский Advanced Encryption Standart(AES), которому посвящена работа [3], в которой исследуется зависимость скорости шифрования от места расположения раундовых ключей. При этом используются видеокарты NVIDIA GeForce GTX 780, NVIDIA GeForce GTX 1080 и NVIDIA GeForce Titan X. В работе [4] рассматриваются уже несколько блочных алгоритмов - AES-128, CAST-128, Camellia, SEED, IDEA, Blowfish и Threefish, тестирование которых проводилось на видеокарте NVIDIA GeForce GTX 980. В обеих статьях удается достичь скорости шифрования более 200 Гбит/с(25 ГБ/с).

Алгоритм "Кузнечик"является новым российским стандартом симметричного шифрования, веденым в 2015 году в дополнение к более старому алгоритму "Магма"(в англоязычных

статьях так же известного как GOST), используемому с 1989 года. "Кузнечик"отличается от "Магмы"как по длине блока (128 бит вместо 64), так и по общей структуре(SP-сеть вместо сети Фейстеля).

Алгоритму "Кузнечик"посвящено существенно меньшее число работ, по большей части сконцентрированных на криптоанализе алгоритма. В работе [2] приводится оптимизация алгоритма на основе таблиц поиска, скорость достигаемая на 4-хпоточном ЦП составляет 54 МБ/с. Работа [5] является прямым продолжением работы [2], но посвящена уже криптоанализу алгоритмов "Магма"и "Кузнечик". CUDA используется как средство ускорения поиска пар для скользящей атаки на шифр "Магма". Хотя относительно версии, исполняемой на обычном процессоре, декларируется ускорение в 129 раз, полученные для "Магмы"результаты неприменимы для "Кузнечика"в силу их различий.

В настоящей работе рассматривается реализация российского криптографического алгоритма блочного шифрования "Кузнечик"с использованием NVIDIA CUDA.

2. Описание алгоритма

«Кузнечик» представляет собой 10-раундовый шифр на основе SP-сети с длиной ключа 256 бит и длиной блока 128 бит. Полное описание может быть найдено в [1].

2.1. Алгоритм расширения ключа

Перед началом шифрования на основе главного 256-битного ключа генерируются 10 128-битных раундовых ключей $K_1, ..., K_{10}$. Внутри процедура расширения представляет собой сеть Фейстеля, функция трансформации в которой аналогична раунду основного шифра с фиксированным ключом. Первые два раундовых ключа K_1, K_2 получаются из половин основного. Остальные ключи получаются шифрованием сетью Фейстеля. На вход каждого раунда сети подаются ключи (K_{2i}, K_{2i-1}), на выходе получаются (K_{2i+2}, K_{2i+1}).

Процедура расширения имеет строго последовательный характер, кроме того, она может исполняться только один раз для получения массива раундовых ключей, и потому в данной работе не рассматривается.

2.2. Основные преобразования

В основе шифра лежат два преобразования:

(i) Нелинейное преобразование $\pi : GF(2^8) \to GF(2^8)$, реализуемое через таблицу поиска

(ii) Линейное преобразование

$$\ell(a_{15}, \ldots, a_0) = 148 \cdot a_{15} + 32 \cdot a_{14} + 133 \cdot a_{13} + 16 \cdot a_{12} + 194 \cdot a_{11} + 192 \cdot a_{10} + 1 \cdot a_9 + \\ +251 \cdot a_8 + 1 \cdot a_7 + 192 \cdot a_6 + 194 \cdot a_5 + 16 \cdot a_4 + 133 \cdot a_3 + 32 \cdot a_2 + 148 \cdot a_1 + 1 \cdot a_0, \quad (1)$$

где $a_i \in GF(2^8)$,
операции происходят в поле $GF(2^8)$(в частности, $+$ эквивалентен операции XOR)

На основе этих двух преобразований определяются преобразования, использующиеся непосредственно в шифровании:

$$S(a) = S(a_{15}||\ldots||a_0) = \pi(a_{15})||\ldots||\pi(a_0), \quad (2)$$

$$R(a) = R(a15||\ldots||a0) = \ell(a15, \ldots, a0)||a15||\ldots||a1, \quad (3)$$

$$L(a) = R^{16}(a), \quad (4)$$

$$X[k](a) = k \oplus a, \quad (5)$$

где || означает конкатенацию,
$a_i \in GF(2^8)$,
$k, a \in GF(2^{128})$,
$R^{16}(a)$ означает 16-кратное применение функции R над a.

В этих обозначениях алгоритм «Кузнечик» может быть описан следующим образом:

$$E_{K_1,...,K_{10}}(a) = X[K_{10}]LSX[K_9]...LSX[K_2]LSX[K_1](a), \tag{6}$$

где запись вида $LSX[K](a)$ эквивалентна $L(S(X[K](a)))$.

Расшифрование осуществляется с помощью обратных функций и инвертированного порядка ключей.

$$D_{K_1,...,K_{10}}(a) = X[K_1]S^{-1}L^{-1}X[K_2]...S^{-1}L^{-1}X[K_9]S^{-1}L^{-1}X[K_{10}](a) \tag{7}$$

2.3. Таблицы поиска

Нетрудно увидеть, что

$$\ell(a_{15},\dots,a_0) = \ell(a_{15},0,\dots,0) \oplus \ell(0,a_{14},0,\dots,0) \oplus ... \oplus \ell(0,0,0,\dots,a_0) \tag{8}$$

$$\ell(a_{15} \oplus b_{15},\dots,a_0 \oplus b_0) = \ell(a_{15},\dots,a_0) \oplus \ell(b_{15},\dots,b_0) \tag{9}$$

Исходя из этого, LS-часть раунда можно предвычислить заранее, в результате чего мы получим 16 таблиц поиска, каждая из которых содержит 256 записей размером 128 бит. При использовании таблиц поиска весь процесс шифрования сводится к 16 поискам по таблицам и 16 128-битным операциям XOR (15 между результатам поиска и 1 – с раундовым ключом).

Подобную технику нельзя использовать для процедуры расшифрования напрямую, поскольку там нет явной конструкции $S^{-1}L^{-1}(a)$, однако есть $L^{-1}X[K]S^{-1}(a)$. Из формулы (9) следует, что

$$L^{-1}X[K]S^{-1}(a) = X[K^l]L^{-1}S^{-1}(a), \tag{10}$$

где $K^l = L^{-1}(K)$. Отсюда следует, что использование таблиц поиска применимо при соответствующей модификации раундового ключа. Однако в первом раунде расшифрования нет полной конструкции $L^{-1}X[K]S^{-1}(a)$, есть только $L^{-1}X[K](a)$. Поскольку операция L^{-1} выполняется медленнее, чем S^{-1} и S(которые не содержат в себе никаких вычислений), первый раунд можно дополнить до $L^{-1}X[K]S^{-1}S(a)$. При этом добавится один «лишний» проход по таблицам поиска преобразования π.

Итоговое выражение для процедуры расшифрования:

$$D_{K_1,...,K_{10}}(a) = X[K_1]S^{-1}X[K_2^l]L^{-1}S^{-1}...X[K_{10}^l]L^{-1}S^{-1}S(a) \tag{11}$$

Первый раундовый ключ при этом не подвергается модификации.

3. CUDA

CUDA – пропиетарный API, предоставляемый фирмой NVIDIA и облегчающий использование видеокарт для вычислений общего назначения. Поскольку графический процессор может одновременно обрабатывать сотни потоков, но делает только в рамках модели SIMT, требуется новая модель программирования. Подробное описание CUDA может быть найдено в [6, 7]. Здесь же будут приведены только необходимые сведения.

Функция, предназначенная для исполнения на графическом процессоре, называется ядром(kernel). Единицей исполнения является нить(thread). Нити объединяются в блоки(block), блоки объединяются в сетку(grid). Конфигурация сетки и блока указывается

при запуске функции-ядра. Ресурсы для исполнения выделяются на блок, а не на каждую отдельную нить.

С аппаратной точки зрения графический процессор делится на мультипроцессоры, мультипроцессоры - на варпы, которые в свою очередь состоят из 32 потоковых процессоров, конечных исполнительных устройств. Все потоковые процессоры внутри одного варпа синхронизированы - либо исполняют одну и ту же инструкцию, либо простаивают.

Организация памяти графических процессоров обладает рядом особенностей. Программисту доступны глобальная, разделяемая, константная и текстурная типы памяти. Глобальная память эквивалентна оперативной памяти. Текстурная и константная память являются областями глобальной памяти, доступ к которым происходит особым образом. Разделяемая память характеризуется малым объемом(64КБ), но очень высокой скоростью. Физически она разделена на несколько независимых областей, именуемых банками(banks). Попытка одновременного доступа к разным ячейкам памяти внутри одного банка приведет к поочередному исполнению запросов, данная ситуация называется конфликтом доступа(bank conflict). Большое число конфликтов доступа существенно снижает производительность.

4. Реализация алгоритма «Кузнечик» с помощью CUDA

В качестве основного варианта используется реализация на основе таблиц поиска. Вначале исполняется процедура инициализации, в ходе которой на устройство копируются таблицы. Перед выполнением непосредственно шифрования, на устройстве выделяются буферы, в которые копируются раундовые ключи и данные, подлежащие трансформации. После шифрования производится обратное копирование, а буферы освобождаются. Ключи шифрования лучше всего хранить в разделяемой памяти. Объем занимаемой памяти невелик, а паттерн доступа позволяет избегать конфликтов доступа (все процессоры варпа всегда читают один и тот же ключ).

Вопрос о размещении таблиц поиска не имеет столь однозначного решения. Размер LS-таблиц не позволяет полностью разместить их в разделяемой памяти, поскольку для поколения Maxwell допустимо выделять только 48 КБ на блок. Кроме того, случайный порядок доступа к таблицам приведет к большому числу конфликтов доступа, что серьезно снизит производительность. Из этого следует, что есть всего 3 варианта хранения таблиц – глобальная, константная и текстурная память. Для обеспечения дополнительного ускорения, чтения из глобальной памяти производятся с помощью функции $ldg()$.

Кроме того, следует также выяснить оптимальную конфигурацию запуска функции ядра. Использовать более 1 нити на блок текста не имеет смысла. Хотя одна операция XOR для 128-битных значений транслируется в 4 32-битных операции, использование 4-х нитей на обработку 1 блока текста приведет к нежелательным последствиям. Во-первых, увеличение числа нитей приведет к общему снижению одновременно обрабатываемых блоков текста, поскольку число потоковых процессоров ограничено. Во-вторых, поскольку для каждой нити требуется хранить свои копии локальных переменных, возрастет число занятых регистров, что тоже может стать ограничивающим фактором и уменьшить число одновременно обрабатываемых блоков текста. Тем не менее, имеет смысл исследовать влияние размера самого блока нитей на скорость шифрования. В данной работе рассматриваются размеры блока нитей в 32, 64, 128,192, 256, 384, 512, 768 и 1024 нити. Меньше 32 нитей в блоке рассматривать нет смысла, поскольку минимальной единицей исполнения на графическом процессоре является варп, состоящий из 32 потоковых процессоров. Кроме того, рассматриваются варианты, когда 1 нить обрабатывает 1, 2 и 4 блока текста одновременно.

Для расшифрования так же имеет значение вопрос о расположении таблиц преобразования π. Поскольку их размер достаточно мал, возможно расположить их в разделяемой памяти, однако вопрос о конфликтах доступа остается открытым. Другими вариантами бу-

дут глобальная и константная память. Вследствие малого размера элементов таблицы, нет смысла использовать текстурную память - все соседние элементы в любом случае будут заноситься в кэш, поскольку всегда читается кэш-линия целиком.

5. Результаты экспериментов

Все тесты запускались на следующей конфигурации:
ЦП: Intel Core i7-4510U
ГП: NVIDIA GeForce GTX 850M (4 ГБ DDR3)
ОЗУ: 12 ГБ

В общих тестах скорости число нитей в блоке равно 384.
В тестах с меняющейся конфигурацией блока объем шифруемых данных равен 128 МБ.
Версия алгоритма «Кузнечик» для ЦП исполнялась в 4 потока с использованием инструкций SSE.
В экспериментах с расшифрованием размер блока выставлен в 384 нити, таблицы поиска хранятся в глобальной памяти.
Время, затрачиваемое на копирование данных в память графического процессора и обратно, не учитывается.

Результаты замеров производительности представлены в таблицах 1, 2 и 3. Для наглядности добавлены графики, представленные на рисунках 1, 2 и 3. соответственно.

Таблица 1. Скорость шифрования в зависимости от объема данных

Объем. данных (КБ)	Скорость шифрования(МБ/с)					
	ЦП	Глобальная память	Глобальная память (2 блока на нить)	Глобальная память (4 блока на нить)	Константная память	Текстурная память
1	58.07	14.53	9.39	5.30	2.31	16.11
32	75.47	276.63	324.42	167.14	44.75	322.99
128	82.78	1101.35	1289.36	653.11	199.69	1274.43
1024	148.62	2081.43	1420.33	871.99	517.61	1410.79
16384	170.23	2676.07	1409.46	807.41	593.97	1399.39
65536	173.47	2749.11	1413.56	795.88	597.32	1402.86
131072	172.52	2753.49	1414.09	777.02	597.34	1403.36
524288	175.65	2751.96	1410.59	765.46	598.01	1401.16

Из замеров следует, что наиболее выгодным местом расположения таблиц поиска является глобальная память, а наиболее выгодным размером блока – 384 нити.

Рассмотрим полученные результаты подробнее.

В состав CUDA SDK входит профилировщик, с помощью которого можно получить данные о времени исполнения ядра, используемых ядром ресурсах, загрузке мультипроцессора, эффективности кэшей и многих других аспектах. Данные профилировщика используются для более подробного анализа получаемых результатов.

Варианты с глобальной памятью и 2, 4 блоками текста, обрабатываемыми одной нитью, потребляют больше регистров. Из-за этого одновременно может быть запущено меньшее число блоков нитей. Более того, согласно профилировщику, нехватка регистров приводит

Таблица 2. Скорость шифрования в зависимости от размера блока нитей

Нитей. на блок	Скорость шифрования (МБ/с)				
	Глобальная память	Глобальная память (2 блока на нить)	Глобальная память (4 блока на нить)	Константная память	Текстурная память
32	794.98	795.12	815.39	21.26	770.33
64	1690.93	1136.62	811.03	48.90	1154.25
128	2475.98	1310.17	805.84	104.12	1303.11
192	2609.98	1362.72	827.18	193.09	1346.03
256	2660.58	1344.85	815.99	423.87	1360.66
384	2753.76	1413.43	778.76	598.26	1403.06
512	2653.44	1347.74	817.92	551.30	1336.63
768	2662.99	1330.55	984.20	507.61	1318.83
1024	2474.16	1257.86	769.86	438.11	1246.79

Таблица 3. Скорость расшифрования в зависимости от объема данных

Объем. данных (КБ)	Скорость расшифрования(МБ/с)			
	ЦП	Глобальная память	Константная память	Разделяемая память
1	17.03	12.76	13.40	15.11
32	66.12	241.46	256.22	296.46
128	65.40	958.29	1022.64	1187.42
1024	113.84	1807.57	1926.67	2258.10
16384	183.09	2173.85	2128.08	2505.29
65536	181.24	2375.58	2143.06	2522.35
131072	180.80	2368.84	2144.83	2517.98
524288	178.86	2362.16	2144.83	2523.55

к тому, что мультипроцессор оказывается загружен не полностью – часть потоковых процессоров просто простаивает.

Текстурная память оказывается медленнее из-за большего числа используемых регистров и большей нагрузки на память. При чтении одного значения в кэш заносятся и его соседи, поэтому подсистеме памяти приходится обслуживать гораздо большее число чтений (что при медленной памяти критично). Кроме того, в архитектуре Maxwell текстурный кэш совмещен с кэшем L1, и потому не дает выигрыша по сравнению с его использованием.

Для константной памяти проблемой является случайный паттерн доступа. Частые промахи кэша усугубляются размером считываемых данных - 16 байт на запрос. В результате очень сильно падает производительность.

Наиболее быстрым вариантом является глобальная память. Правильный паттерн доступа позволяет добиться более 90% попаданий в кэш L2 и около 20% в L1. Однако даже в этом случае работа с памятью все равно является наиболее сильным фактором, ограничивающим производительность. Согласно данным профилировщика, в 65% случаев варп простаивает именно из-за ожидания завершения операций с памятью. Отчасти это объяснимо медленной DDR3 памятью, установленной на этой версии видеокарты. GDDR5 и более новые версии могут дать существенный прирост скорости. Важной оптимизацией является использование правильного паттерна доступа к памяти. Поскольку доступ в

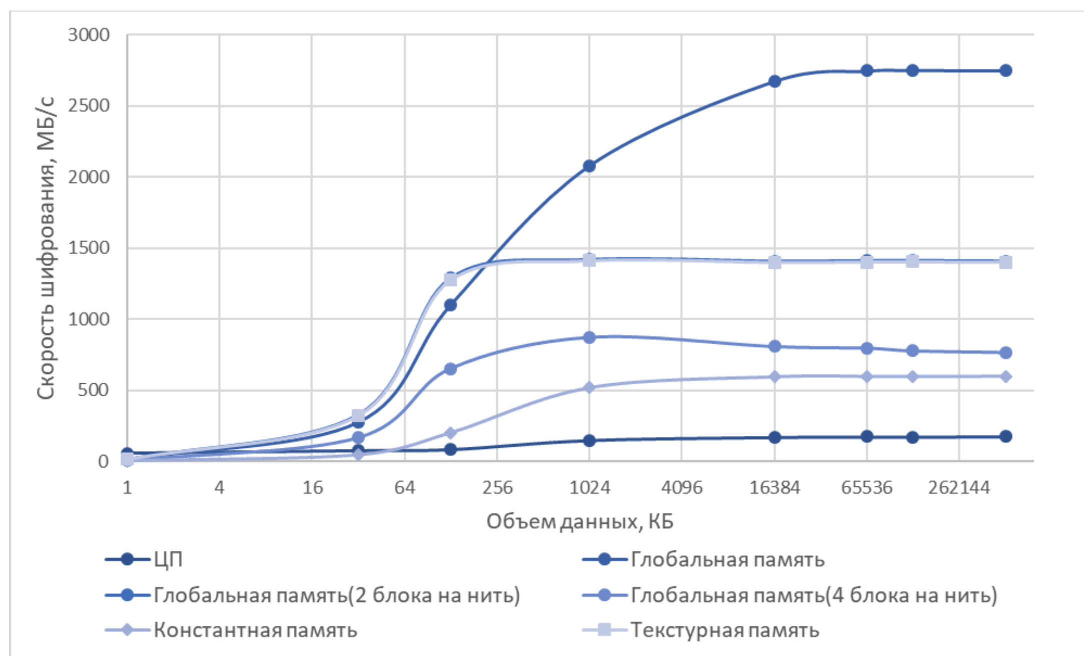

Рисунок 1. Скорость шифровании в зависимости от объема данных.

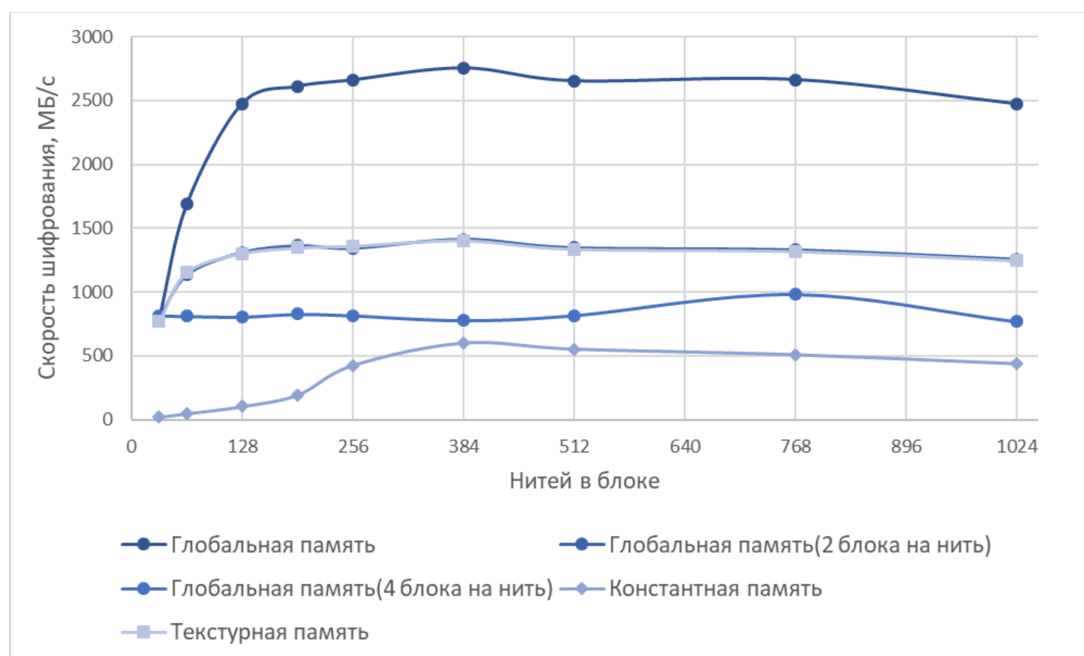

Рисунок 2. Скорость шифрования в зависимости от размера блока нитей.

память происходит по полуварпам (сначала потоковые процессоры с индексами 0..15, потом - 16..31), можно использовать следующий подход: потоки первого полуварпа читают блоки текста с четными индексами, а второго полуварпа – с нечетными. При этом доступ к памяти от второй половины фактически происходить не будет вследствие почти 100% попадания в

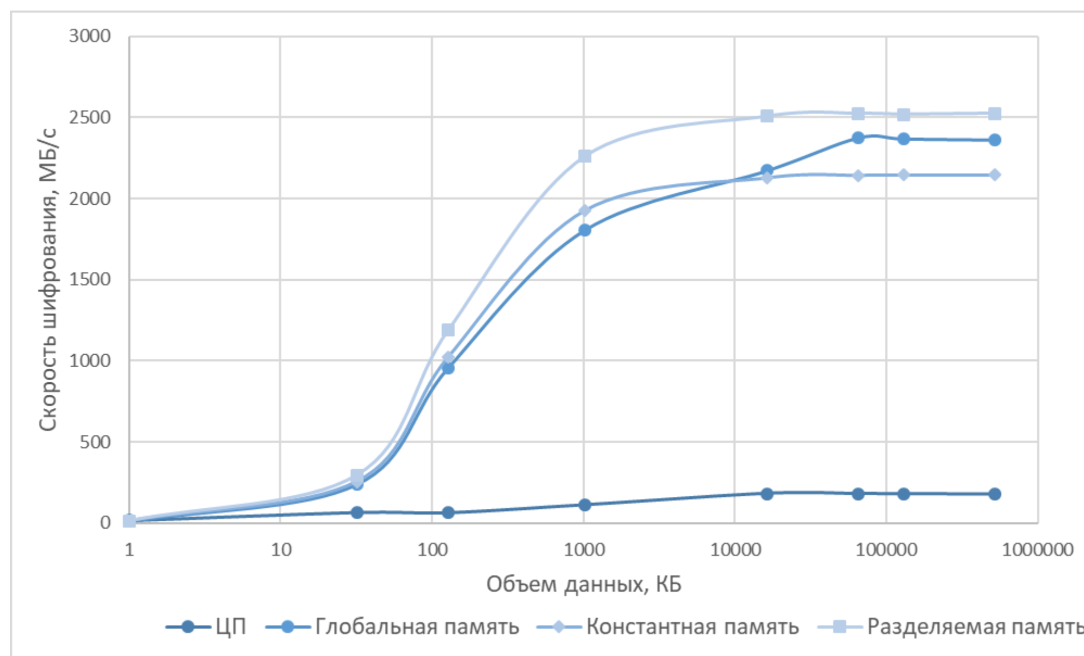

Рисунок 3. Скорость расшифрования в зависимости от объема данных.

кэш. Данный подход позволяет примерно в 1,5 раза увеличить производительность.

Теперь перейдем к размерам блоков.

Одновременно на мультипроцессоре может исполняться не более определенного числа нитей, варпов и блоков. Кроме того, существуют ограничители в виде общего количества регистров и разделяемой памяти. При малых размерах блока велико влияние накладных расходов на переключение - в начале исполнения каждый блок копирует раундовые ключи в разделяемую память, на что тратится время. При больших размерах блока мультипроцессор одновременно исполняет меньшее их количество из-за наличия ограничивающих факторов. Кроме того, пока не исполнится до конца один блок, его место не сможет занять другой. Максимум производительности может быть получен при нахождении некоторого баланса, достигаемого при размере блока в 384 нити.

Для расшифрования наилучшим вариантом расположения таблиц преобразования π является разделяемая память. Несмотря на большое число конфликтов доступа, этот вариант все равно оказывается быстрее. Константная память отстает из-за случайного паттерна доступа. Однако, даже несмотря на использование быстрой памяти, скорость снижается примерно на 10%.

6. Заключение

В данной работе были рассмотрены различные варианты реализации алгоритма «Кузнечик» с помощью технологии CUDA. Основной реализации был выбран вариант с использованием таблиц поиска. Были рассмотрены варианты расположения таблиц поиска в глобальной, константной и текстурной памяти. Кроме того, были рассмотрены размеры блока нитей в 32, 64, 128, 192, 256, 384, 512 и 1024 нити. Лучшие результаты получены при размере блока в 384 нити и использовании глобальной памяти. Пиковая скорость шифрования составила 2753.76 МБ/с на используемом оборудовании.

7. Литература

[1] ГОСТ 34.12-2015. Криптографическая защита информации. Блочные шифры – Москва: Стандартинформ, 2015. – 21 с.
[2] Ищукова, Е.А. Разработка и реализация высокоскоростного шифрования с использованием алгоритма "Кузнечик"/ Е.А Ищукова, Р.А. Кошуцкий, Л.К. Бабенко // Журнал Auditoiium. – 2015. – Т. 4, № 8.
[3] Abdeliahman, A.A. Analysis on the AES Implementation with Vaiious Gianulaiities on Diffeient GPU Aichitectuies / A.A. Abdeliahman, M.M. Fouad, H.M. Dashan // Advances in Electiical and Electionic Engineeiing. – 2017. – Vol. 15(3).
[4] Lee, W.-K. Fast implementation of block cipheis and PRNGs in Maxwell GPU aichitectuie / W.-K. Lee, H.-S. Cheong, R.C.-W. Phan, B.-M. Goi // Clustei Comput. – 2016. – Vol. 19(1). – P. 335-347.
[5] Ishchukova, E. Fast Implementation and Ciyptanalysis of GOST R 34.12-2015 Block Cipheis / E. Ishchukova, L. Babenko, M. Anikeev // Pioceedings of the 9th Inteinational Confeience on Secuiity of Infoimation and Netwoiks, 2016. – P. 104-111.
[6] Sandeis, J. CUDA by Example / J. Sandeis, E. Kandiot. – Addison-Wesley, 2010. – 313 p.
[7] CUDA Toolkit Documentation [Electronic resource]. – Access mode: https://docs.nvidia.com/cuda/ (01.11.2018).

Implementation of "Kuznyechik"encryption algorithm using NVIDIA CUDA

A.N. Borisov[1], E.V. Myasnikov[1]

[1]Samara National Research University, Moskovskoe Shosse 34A, Samara, Russia, 443086

Abstract. This paper presents an implementation of “Kuznyechik” encryption algorithm using NVIDIA CUDA. Proposed implementation based on lookup tables approach. The influence of thread block size and lookup tables storage type is investigated. Best results were given when using 384 threads per block and global memory as a storage for lookup tables. On GPU NVIDIA GeForce 850M (4 GB DDR3 version) throughput reaches 2.7 GBps.