Сравнение эффективности CPU и GPU реализаций некоторых комбинаторных алгоритмов на задачах обращения криптографических функций

В.Г. Булавинцев

Институт динамики систем и теории управления СО РАН

Проводится сравнение эффективности CPU и GPU реализаций некоторых комбинаторных алгоритмов, используемых в криптоанализе. В частности, анализируются причины, по которым не удается эффективно реализовать на GPU алгоритмы, осуществляющие «интеллектуальный перебор». Показывается, что применение специальных техник трансформации потока управления позволяет существенно компенсировать потери производительности, возникающие из-за неэффективного исполнения условных переходов на SIMD-устройстве. Однако ограничения, которые накладывают механизмы работы с памятью, применяемые в современных GPU, для рассматриваемых алгоритмов оказываются непреодолимыми. В качестве тестовых задач рассматриваются задачи обращения криптографических функций DES и A5/1.

1. Введение

Современные GPU предоставляют выгодное соотношение цены, производительности и энергопотребления. Многие суперкомпьютерные кластеры имеют высокий рейтинг в ТОП500, благодаря использованию GPU [1]. Однако, будучи специализированными устройствами, рассчитанными на потоковую обработку однотипных данных, GPU на многих алгоритмах не показывают значимого преимущества перед процессорами традиционной архитектуры [2]. Это обусловлено тем, что современные GPU используют SIMD-архитектуру [3] и специально рассчитанные на работу с ней контроллеры памяти [4]. GPU являются «массивно-параллельными» процессорами, т. е. вычислительными устройствами, на которых одновременно выполняется большое (по сравнению с CPU) количество вычислительных потоков. Многие алгоритмы, применяемые в криптографии и криптоанализе, имеют возможность почти неограниченного масштабирования на параллельных вычислительных архитектурах. Криптоанализ методом прямого перебора, обслуживание криптовалют [5], построение rainbow-таблиц [6] – популярные алгоритмы демонстрирующие высокую эффективность на GPU. В то же время перспективным направлением в криптоанализе является применение алгоритмов, основанных на различных подходах к «интеллектуальному» сокращению перебора. В частности, в последние годы появился ряд работ, в которых для решения задач криптоанализа используются современные SAT-решатели [7, 8]. При реализации таких алгоритмов на GPU обнаруживается, что эта архитектура совершенно не приспособлена для их эффективного выполнения: малый объем кэш-памяти, проблемы с условными переходами, узкоспециальные механизмы работы с памятью – все эти факторы приводят к тому, что даже после тщательной адаптации алгоритма в соответствии с рекомендациями производителя, GPU сильно проигрывают CPU в скорости и эффективности их исполнения.

До сих пор основное внимание исследователей (за редким исключением, [2]) было сосредоточено на положительных аспектах применения GPU. Настоящая работа посвящена исследованию основных причин низкой производительности «интеллектуальных» комбинаторных алгоритмов на GPU на примере алгоритма DPLL. Для сравнения используются алгоритмы, демонстрирующие на GPU высокую производительность (криптоанализ DES и A5/1 методом полного перебора). В работе продемонстрирован метод, позволяющий уменьшить потери производительности от неэффективной обработки условных переходов на GPU – первой из двух основных причин низкой производительности DPLL на этой платформе. Потери от второй причины – специализированного механизма доступа к памяти – к сожалению, на современных GPU компенсированы быть не могут.

1.1 Криптоанализ генератора A5/1 на GPU методом полного перебора

Генератор потокового шифра A5/1 является стандартным для применения в современных сетях мобильной связи GSM. Он представляет собой [9] 3 независимых друг от друга регистра сдвига с линейной обратной связью (РСЛОС) длиной 19, 22 и 23 бит (всего 64 бит). РСЛОСы тактируются условно, по т. н. функции большинства («majority function»), взятой от одного бита из каждого регистра. Для получения ключевого потока выходы РСЛОСов смешиваются друг с другом. Секретным ключом является начальное заполнение РСЛОСов. Шифрование осуществляется побитовым смешиванием ключевого потока, порожденного генератором, с открытым текстом с помощью функции XOR. Протокол шифрования A5/1 продолжает использоваться, несмотря на то, что различными исследовательскими группами были продемонстрированы практические атаки на него [6, 9].

Далее мы рассматриваем задачу криптоанализа A5/1 на основе известного фрагмента ключевого потока [10]. Простейший метод криптоанализа в этом случае – полный перебор: для каждого из всех возможных вариантов секретного ключа генерируется соответствующий фрагмент ключевого потока, который сравнивается с заранее известным образцом. Для генератора A5/1, чтобы однозначно определить секретный ключ достаточно фрагмента ключевого потока длиной 64 бит [10]. Таким образом, в худшем случае требуется для каждого из 2^{64} ключей-кандидатов получить 64 бита выхода генератора и сравнить их с известным образцом ключевого потока. Поскольку шифр является потоковым, нет необходимости генерировать сразу все 64 битов выхода. Достаточно генерировать ключевой поток по 1 биту и, в случае его несовпадения с образцом, прерывать проверку текущего ключа-кандидата.

Мы использовали специальную реализацию алгоритма A5/1. Поскольку РСЛОСы независимы друг от друга и их длина невелика (19-23 бит), для каждого из них можно сгенерировать соответствующую непериодическую часть порождаемой ими двоичной последовательности (т.н. М-последовательность). Для используемых в A5/1 РСЛОС получаемые последовательности занимают в сумме около 2 Мбайт оперативной памяти. Затем состояние любого бита РСЛОСа для любого его такта при любом начальном заполнении может быть получено выборкой n-ого значения из соответствующей последовательности по формуле: n=(S+t+b), где S – смещение от начала последовательности, кодирующее начальное заполнение РСЛОСа; t – номер такта; b – номер бита в РСЛОСе. В таком случае вместо перебора начальных заполнений РСЛОСов в лексикографическом порядке, перебор проходит в порядке, заданном М-последовательностью. Поскольку значения в М-последовательности не повторяются, лишняя работа не производится. Эта версия алгоритма может быть реализована как на CPU, так и на GPU. Она не предъявляет специальных требований к возможностям оборудования[1].При переносе на GPU ускорение достигается за счет одновременной проверки многих ключей-кандидатов. Каждый из тысяч вычислительных потоков GPU проверяет один вариант ключа. Данные по скорости реализации описанной атаки на GPU приведены в конце статьи (Табл. 3).

1.2 Криптоанализ алгоритма DES на GPU методом полного перебора

Блочный шифр DES являлся федеральным стандартом США с 80-х годов и активно применялся вплоть до конца XX века. DES - симметричный шифр, построенный на сети Фейстеля. Длина ключа в DES составляет всего 56 бит, что и является главной его уязвимостью. С середины 90-х годов были неоднократно продемонстрированы практичные атаки на DES, использующие «метод грубой силы» (например, [11]).

Мы реализовали брутфорс-атаку на DES в условиях известного открытого текста. Для ускорения работы алгоритма мы применили технику bitslice, повышающую эффективность использования возможностей современных вычислительных архитектур [12].

Опишем вкратце технику bitslice. Шифр представляется в виде схемы, составленной из логических вентилей. Используя соответствующую этой схеме последовательность побитовых логических операций, можно одновременно проверять столько ключей-кандидатов (или шиф-

[1]Например, для эффективной реализации алгоритма A5 в технике bitslice требуется, чтобы устройство поддерживало аппаратную инструкцию bitselect или аналогичную ей [6].

ровать столько блоков секретного текста), сколько двоичных разрядов содержит один регистр общего назначения (РОН) вычислительного устройства. Например, на 32-разрядной платформе в технике bitslice каждый РОН содержит в себе по 1 биту каждого из 32 ключей-кандидатов. Таким образом, на 32-разрядной платформе bitslice позволяет проверять одновременно 32 ключа. Следует отметить, что построение оптимальной схемы из вентилей является нетривиальной задачей, и исследователи до сих пор работают над улучшением такой схемы для DES. В нашей работе мы использовали лучшую известную на данный момент схему для DES на стандартных вентилях (И, ИЛИ, НЕ, XOR) [13].

Как и в случае A5/1, при переносе на GPU ускорение достигается за счет одновременной проверки многих ключей. Однако в данном случае каждый вычислительный поток GPU проверяет 32 ключа – за счет использования техники bitslice (для удобства сравнения версия для CPU работает в 1 поток и проверяет на нем за счет техники bitslice одновременно также 32 ключа). Данные по достигнутой в результате применения описанного подхода скорости перебора пространства ключей DES на GPU приведены в конце статьи (табл. 3)

2. Реализация алгоритма DPLL на GPU

Многие задачи криптоанализа можно рассматривать в контексте более общей задачи обращения дискретных функций. Последняя задача эффективно сводится к задаче поиска выполняющего набора булевой формулы (т.н. SAT-задача). Наиболее эффективные программы-решатели SAT-задач построены на базе классического алгоритма DPLL [14]. Он представляет собой направленный обход дерева, дополненный правилом распространения булевых ограничений) Boolean constraint propagation, BCP. Для ускорения процедуры BCP, которая занимает до 95% времени работы решателя, применяются специальные «ленивые» структуры данных, т. н. «watched literals» [15]. Также, в современных решателях используется стратегия Conflict-Driven Clause Learning (CDCL): обнаружив конфликт в присвоении переменных, решатель анализирует его причины и добавляет информацию о них к основной базе ограничений в виде т. н. «конфликтного дизъюнкта». Это позволяет ускорить обход дерева, пропустив проверку переменных, не имеющих отношения к конфликту («backjumping») [16]. В дальнейшем конфликтный дизъюнкт может ускорить вывод по BCP.

Далее мы кратко описываем сделанную нами GPU-реализацию алгоритма DPLL. Рассмотрев различные подходы к распараллеливанию алгоритма DPLL, мы пришли к выводу, что единственный способ полностью задействовать возможности GPU, не ориентируясь при этом на особенности SAT-задач, – это запускать в каждом вычислительном потоке отдельный экземпляр алгоритма. При этом общее дерево поиска расщепляется по значениям n отдельных переменных на 2^n подзадач, и каждый вычислительный поток GPU решает одну такую подзадачу. Когда поток заканчивает работу над своей подзадачей, он ищет другой поток, у которого еще есть работа, и расщепляет его подзадачу, забирая часть работы себе (при этом используется техника «work stealing» [17]).

2.1 Адаптация структур данных, используемых в DPLL, для GPU

Малый объем памяти современных GPU является одним из препятствий для реализации полноценной стратегии CDCL. Размер КНФ в задачах криптоанализа может достигать 10^4–10^6 литералов [7]. В то же время, при 2048 запущенных на GPU потоках, на каждый поток приходится не более 512 Кб из 1 Гб памяти устройства. Поэтому мы реализовали ограниченную версию CDCL, с хранением только тех дизъюнктов, которые необходимы для корректной работы нехронологического бэктрекинга [16].

Для представления дизъюнктов в памяти GPU мы разработали новые структуры данных, сокращающие потребление памяти решателем с 32 бит до 1 бита на каждый литерал КНФ. Требования к памяти GPU в нашем решателе могут быть определены по формуле: $M=M_o+n_п M_и$, где M_o – общие для всех потоков данные, $M_и$ – индивидуальные рабочие данные потоков, $n_п$ – число потоков. К примеру, для КНФ: A5/1 M_o=1160 Кбайт, $M_и$=114 Кбайт (с учетом требований выравнивания памяти для работы на GPU). Эти 2048 потоков и общие данные КНФ при решении A5/1 (119700 литералов, 7817 переменных) занимают в памяти устройства около 270

Мбайт. Если бы мы применили схему, в которой каждый поток хранит все данные КНФ полностью, для размещения данных тех же 2048 потоков понадобилось бы не менее 1200 Мбайт. Кроме того, при запуске большого числа потоков примененная нами схема хранения является более «дружественной» к использованию кэш-памяти, т. к. до 50% запросов к памяти в алгоритме DPLL составляют обращения к литералам. Поскольку общие для всех потоков данные занимают около 1 Мб, они могут почти полностью разместиться в кэш-памяти, быстрым доступом к которым могут воспользоваться все потоки. В случае традиционной схемы ресурсы кэш-памяти устройства были бы распределены между всеми потоками, что привело бы к значительному снижению эффективности кэширования.

2.2. Реализация алгоритма DPLL на GPU с учетом особенностей SIMD

Современные GPU NVIDIA состоят из нескольких SIMD-ядер, называемых «мультипроцессорами». Каждый мультипроцессор обладает собственным набором регистровой памяти, кэшем L1, общей памятью, несколькими АЛУ, диспетчером команд, контроллером условных переходов и т.д. Мультипроцессоры независимы друг от друга и подключены к основной памяти устройства через кэш L2.

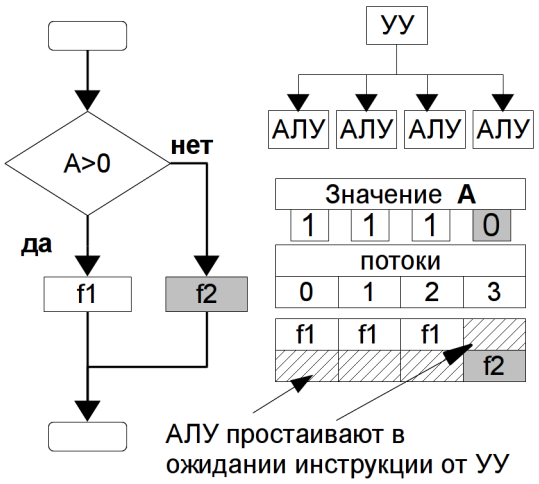

Рис. 1: Сериализация инструкций в результате условного перехода в SIMD

Для выполнения на мультипроцессоре вычислительные потоки GPU объединяются в SIMD-группы размером в 32 потока, именуемые «варпами» («warp»). Потоки в варпе выполняются в режиме «lockstep», т.е. «1 шаг – 1 одинаковая инструкция для всех потоков».

Условные переходы в SIMD обрабатываются особым образом. Если во время исполнения оператора условного перехода возникает ситуация, когда часть потоков в варпе должна выполнить один блок инструкций, а оставшиеся потоки – другой, происходит последовательное выполнение этих блоков (сериализация). Вначале будет выполнен блок, назначенный наибольшему числу потоков, затем – назначенный остальным. При сериализации потоки, не получившие инструкции, простаивают в бездействии (Рис. 1). Это явление называется «warp divergence», оно является следствием экономии на управляющей логике в SIMD-архитектурах. К примеру, в результате последовательности из 5 вложенных условных переходов может возникнуть ситуация, когда из всех 32 потоков варпа в каждый момент времени будет выполняться только 1. Производительность мультипроцессора на таком участке кода при этом ухудшается пропорционально сериализации, до 32 раз. Чтобы уменьшить потери производительности от этого «SIMD-эффекта», можно применять различные техники программирования, позволяющие сократить в коде количество условных переходов и/или степень их вложенности. Далее мы описываем соответствующие техники, использованные нами при реализации на GPU нехронологического алгоритма DPLL.

Алгоритм DPLL можно условно рассматривать как конструкцию, состоящую из 4-х вложенных циклов:

1. цикл обхода дерева поиска;

2. цикл обработки очереди переменных в BCP;
3. цикл обработки дизъюнктов одной переменной;
4. цикл обработки литералов в отдельном дизъюнкте (цикл нижнего уровня).

Цикл обхода дерева поиска включает в себя процедуры угадывания новых переменных и backjumping'a после обнаружения конфликтного присвоения. Однако по сравнению с процедурой BCP эти процедуры используются редко – около 95% всего времени занимает BCP.

Процедуры этих циклов оптимизированы так, чтобы выполнять только те итерации, которые действительно необходимы. Дизъюнкты могут иметь разную длину и невозможно определить заранее, сколько литералов из него понадобится просмотреть. Подмножество дизъюнктов, требующих проверки, также отличается для разных переменных и зависит от текущего их состояния и пути, пройденного решателем. Выведенные по BCP литералы добавляются в очередь, поэтому и ее длину невозможно предсказать заранее, она меняется динамически по ходу решения. Путь решателя в дереве также непредсказуем. Таким образом, каждый из 4-х циклов будет иметь непредсказуемую длину, зависящую от решаемой подзадачи, порядка угадывания переменных и т.д. Вычислительное время, необходимое для обработки любого из этих циклов также может варьироваться в широких пределах. Поскольку число итераций этих циклов заранее неизвестно, выход из них происходит по условию. В результате для DPLL глубина вложенности условных переходов составляет, как правило, 4 и более, что крайне негативно сказывается на производительности алгоритма на GPU, за счет усиления «SIMD-эффекта».

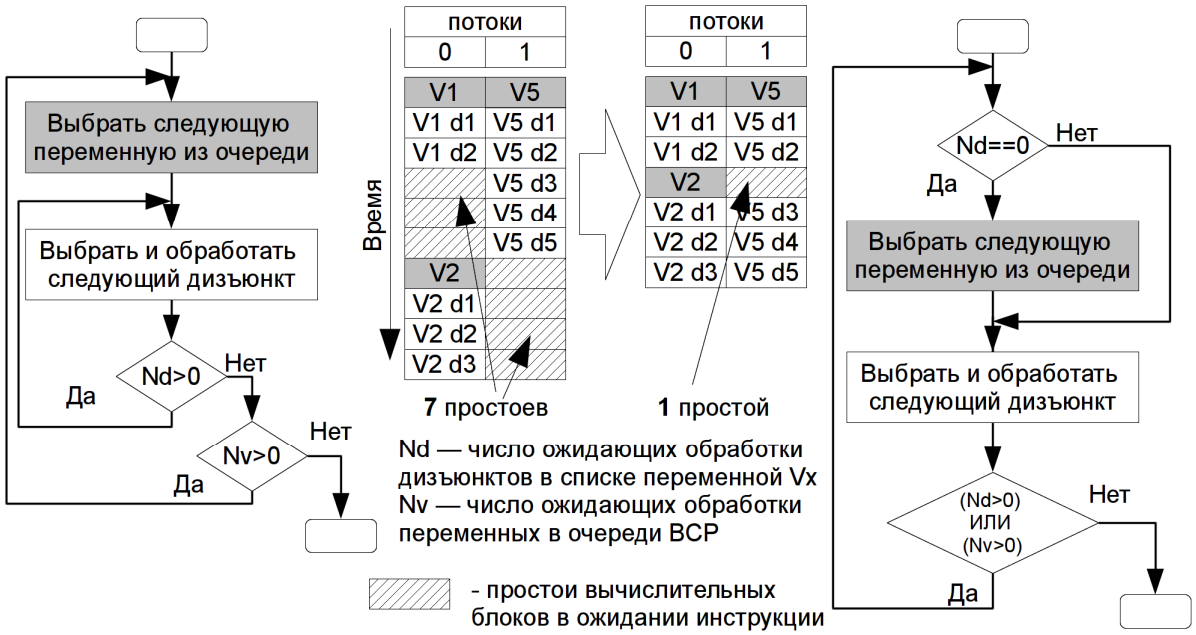

Рис. 2: Уменьшение степени вложенности циклов (на примере алгоритма DPLL).

Для того чтобы уменьшить потери производительности от SIMD-эффекта, можно реорганизовать вложенные циклы таким образом, чтобы уменьшить степень их вложенности. Пример такой трансформации для алгоритма DPLL приведен на Рис. 2. Основная ее идея состоит в том, чтобы использовать обратную дугу внешнего цикла в качестве обратной дуги внутреннего цикла, тем самым оставив лишь одну обратную дугу и один естественный цикл. Проверка необходимости повтора внешнего цикла объединяется с той же проверкой для внутреннего цикла. Операции, относящиеся к циклу верхнего уровня, при этом переставляются под условный переход-предикат. Полученный в результате граф потока управления не будет эквивалентен исходному, но при выполнении алгоритма последовательность базовых блоков всегда будет совпадать с первоначальной. Эта трансформация должна положительно сказываться на производительности алгоритма не только на GPU, но и на современных суперскалярных CPU, для которых важна возможность конвейеризации потока команд. Применение этой техники в нашей реализации DPLL для объединения циклов обработки очереди переменных с циклом обработки дизъюнктов одной переменной позволило повысить ее производительность на GPU на 30-50%.

В целом применение этой и других подобных техник реорганизации условных переходов позволило увеличить производительность DPLL на GPU в 2-5 раз (в зависимости от характера решаемой задачи).

Объединение циклов может привести и к ухудшению производительности приложения на SIMD-архитектуре (Рис. 3).

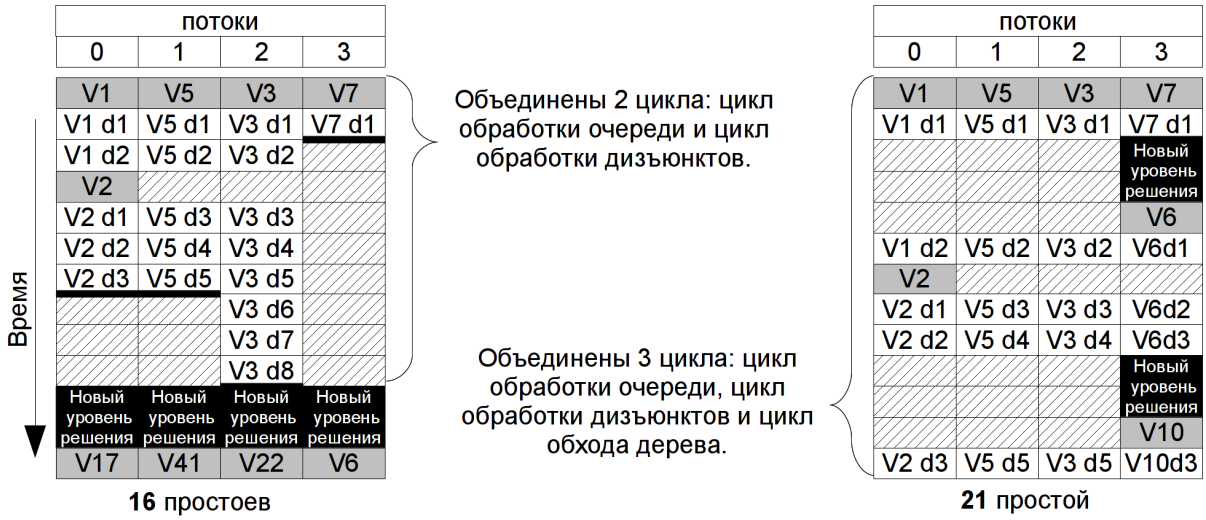

Рис. 3: Потери производительности из-за избыточного объединения циклов (на примере DPLL).

Преимущества от более полного использования АЛУ при обработке команд внутреннего цикла могут быть нивелированы простоями при обработке стоящих под предикатом команд внешнего цикла, если их выполнение требует слишком много времени. Выясним, в каких случаях одной SIMD-группе выгодно перейти к обработке внешнего цикла. Введем следующие обозначения:

T_a - время, выполнения инструкций внешнего цикла (цикла «a»), необходимых для обеспечения потоков заданием для внутреннего цикла;

T_b - время, необходимое для выполнения инструкций одной итерации внутреннего цикла («b»);

N - общее число потоков (в большинстве случаев равно размеру варпа);

n - число остановившихся потоков, закончивших задания во внутреннем цикле и готовых перейти ко внешнему циклу, чтобы получить новые задания;

$P_{iter}(b, N)$ - наиболее вероятное число итераций внутреннего цикла, после которого хотя бы у одного из N потоков закончится задание;

$T_b^{'} = T_b P_{iter}(b, N)$ - среднее время которое проведут потоки на внутреннем цикле, если у всех N потоков будут задания;

$(N-n)T_a$ - цена получения новых заданий остановившимися потоками (т. е. простои при переходе ко внешнему циклу тех потоков, у которых работа еще есть).

Прерывать обработку внутреннего цикла и выходить на внешний цикл имеет смысл только в том случае, если от этого будет выигрыш в работе: $nT_b^{'} - (N-n)T_a > 0$. Преобразовав это выражение, получим:

$$\frac{n}{N} > \frac{T_a}{T_a + T_b^{'}}$$

Компилятор и диспетчер потоков мультипроцессора не знают ничего о средней продолжительности внутреннего цикла, которая, вообще говоря, зависит от решаемой задачи. Как правило, диспетчер действует по правилу «большинства голосов» - переходит по той ветви условного перехода, которую требует в текущий момент большинство активных потоков. Очевидно, что такая стратегия не всегда будет наиболее эффективной.

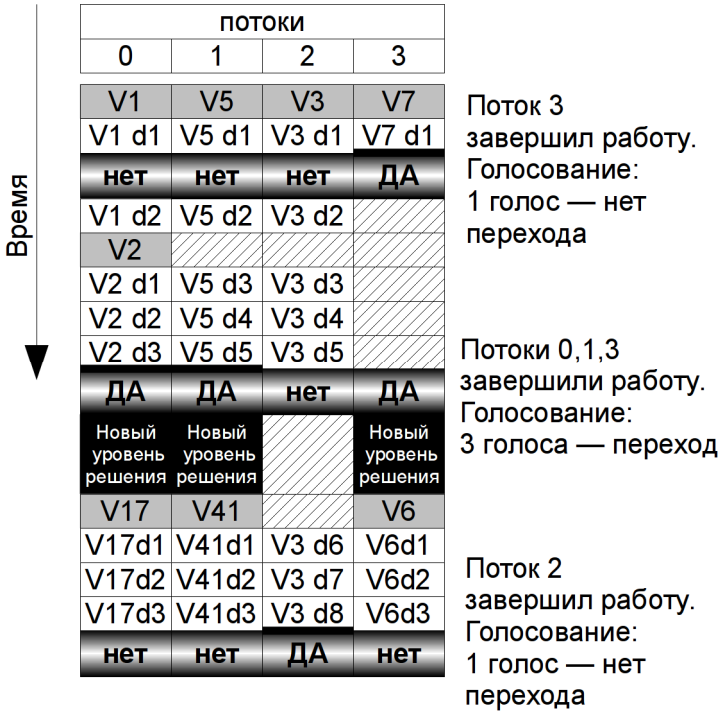

Рис. 4: Стратегия "голосований" за условный переход в SIMD (на примере DPLL).

Поэтому мы реализовали «ручное» управление выходом в цикл обхода дерева из цикла BCP в DPLL. Для этого мы применили стратегию «голосований» (Рис. 4) с использованием встроенных инструкций GPU Fermi [4]. Результаты голосования внутри SIMD-группы сравниваются с пороговым значением, при достижении которого группа выходит на внешний цикл, чтобы простаивающие потоки могли получить новое задание. Голосование проводится каждый раз, когда в цикле BCP у одного из потоков заканчивается работа. Эксперименты с различными классами задач и значениями порога голосования показали, что порог должен подбираться индивидуально, в зависимости от характера решаемой задачи. В наших экспериментах в отдельных случаях удачно подобранное значение порога позволяло добиться ускорения работы решателя до 15%. Интересным развитием этой техники может стать реализация динамического изменения порога, в зависимости от накопленной во время решения задачи статистики. По сути, управление значением порога является формой механизма предсказания ветвлений. Важно заметить, что метод голосования не является специфическим для DPLL, он может применяться в любых алгоритмах со сложной структурой вложенных циклов.

3. Особенности доступа к памяти в GPU

Каждый мультипроцессор GPU способен обрабатывать несколько SIMD-групп одновременно, динамически переключаясь между ними. Это позволяет компенсировать латентность основной памяти устройства: в каждый момент времени обрабатывается та SIMD-группа, данные для исполнения команд которой уже доставлены из основной памяти устройства. Если данные для текущей группы еще не готовы, мультипроцессор переключается на обработку следующей группы, и т.д. Чем больше одновременно запущено на мультипроцессоре SIMD-групп – тем лучше удается скрыть латентность памяти, тем полнее используется доступная пропускная способность памяти (ПСП). Возможность работы нескольких SIMD-групп на одном мультипроцессоре ограничена его ресурсами: регистровой памятью («registry file») и общей памятью («shared memory») [4].

3.1 Группировка запросов к памяти

В случае SIMD-архитектур обработка данных производится пакетами, состоящими из нескольких одинаковых элементов, предназначенных различным выч. потокам одной SIMD-

группы. При этом запросы к памяти от всех 32-х потоков SIMD-группы должны укладываться в границы одной кэш-линии кэша L2 (128 байт для Fermi) – тогда они будут доставлены мультипроцессору в виде единого пакета данных [4]. В противном случае происходит сериализация обращений к памяти, создающая до 32 запросов вместо одного (Рис. 5). Этот «эффект разреженного доступа» («uncoalesced access») приводит к кратному уменьшению эффективной ПСП.

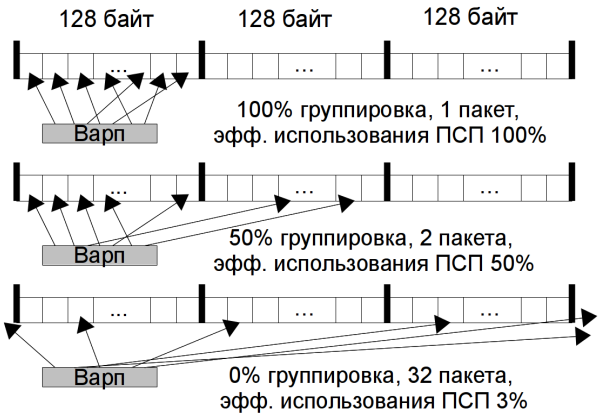

Рис. 5: Группировка запросов к памяти GPU.

Для того чтобы запросы к памяти от одной SIMD-группы всегда выполнялись максимально эффективно, достаточно чтобы соблюдалось 2 условия:

1. Тип данных элементов массива должен быть выровнен в соответствии с требованиями устройства.
2. При выполнении инструкции потоки одной SIMD-группы должны обращаться к элементам массива, последовательно расположенным в памяти устройства.

Для описанных выше атак полного перебора (в применении к A5/1 и DES) эффект разреженного доступа не создает потерь эффективности, поскольку в этих алгоритмах рабочие данные потоков умещаются в регистрах и кэше L1, и не требуют хранения в основной памяти устройства.

В DPLL последовательность обращений к элементам массивов основной памяти заранее предсказать невозможно, поэтому разреженный доступ имеет вероятностный характер и может привести к весьма существенным потерям эффективности.

3.2 Сравнение характеристик кэш-памяти и основной памяти GPU и CPU.

Для компенсации эффектов фон Неймановского ограничения [18] в современных CPU и GPU применяется кэш-память, расположенная непосредственно на кристалле устройства. Объем, скорость и алгоритмы работы кэш-памяти в CPU и GPU сравнимы друг с другом, однако на практике кэш-память выполняет в них разные роли. Например, в GPU Nvidia, построенных на основе архитектуры Fermi, кэш-память 1-го уровня выделяется из расчета 16 Кбайт (или 48 Кбайт) на один блок (одну или несколько SIMD-групп, выполняемых на одном мультипроцессоре). Такой объем кэш-памяти 1-го уровня сравним с объемом кэш-памяти 1-го уровня в CPU Intel архитектуры Nehalem. Однако, в пересчете на 1 вычислительный поток (при 32 потоках в блоке) объем ее составит в лучшем случае 1,5 Кбайт. Этого достаточно лишь для того, чтобы частично компенсировать нехватку локальных регистров мультипроцессора, используемых для хранения промежуточных результатов вычислений (т.н. «registers spilling» [4]). Кэш память 2-го уровня в Fermi составляет 768 Кбайт и является общей для всего устройства. Ее объем сравним с объемом кэшей 2-го и 3-го уровней в Nehalem. Однако, в случае 2048 потоков, запущенных на GPU GTX 560Ti, на один поток придется всего 0,375 Кбайт этой кэш-памяти. Очевидно, что кэш-память 2-го уровня в архитектуре Fermi не может вместить «горячие» данные всех потоков. Она используется в основном для ускорения доступа к небольшому объему данных, одновременно используемых всеми потоками, и для обработки инструкций атомарного доступа к памяти. В случаях, когда обращения вычислительных потоков к глобальной памяти GPU не локализованы, наличие кэш-памяти L2 не дает почти никаких преимуществ.

Таблица 1. Сравнение подсистем памяти GPU и CPU[2]

Уровень подсистемы памяти	CPU (Intel Nehalem)			GPU (Nvidia Fermi)	
	Объем	**ПСП 128 бит**	**ПСП 32 бита**	**Объем**	**ПСП 32-10243 бита**
Кэш L1	32 Кбайт	45 Гбайт/с	11 Гбайт/с	16/48 кбайт	~8000 Гбайт/с
Кэш L2	256 Кбайт	31 Гбайт/с	8 Гбайт/с	768 кбайт	16-520 Гбайт/с
Кэш L3	8 Мбайт	26 Гбайт/с	6 Гбайт/с	-	-
Основная память	1-24 Гбайт	20 Гбайт/с	5 Гбайт/с	1-2 Гбайт	4-128 Гбайт/с

Данные, представленные в табл.1показывают, что алгоритм, скорость которого ограничена ПСП в режиме передачи 32-битных слов и хранящий данные в основной памяти устройства, покажет 0,8-21-кратное ускорение при переносе с CPU Nehalem на GPU Fermi, в зависимости от того, насколько сгруппированными окажутся обращения к памяти при решении конкретной задачи. «Дружественность» структур данных к кэш-памяти и особенности конкретной задачи могут улучшить эту оценку.

В DPLL при 2048 потоках, запущенных на GPU занимаемый рабочими данными потоков, объем памяти устройств, составит около 300Мбайт, в зависимости от числа литералов и переменных в КНФ. Поскольку «горячая область» памяти в каждой подзадаче/потоке будет разной, маловероятно, что наличие кэша L2 существенно повлияет на производительность устройства.

В описанных выше алгоритмах криптоанализа полным перебором в случае DES кэш L1 активно используется для компенсации нехватки регистров. В случае же A5/1 наличие у GPU кэша L2 заметно влияет на производительность, поскольку он способен вместить большую часть данных, к которым обращаются все потоки во время работы.

3.3 Более детальная картина обращений к памяти в DPLL.

До 95% времени работы алгоритма DPLL занимает процедура BCP. В основном она состоит из непредсказуемой последовательности обращений к массивам литералов, watched-флагов и состояний переменных. Поэтому скорость DPLL ограничена, как правило, латентностью памяти в режиме произвольного доступа. Если же латентность памяти не является сдерживающим фактором, производительность алгоритма будет зависеть от реальной ПСП устройства в режиме передачи коротких слов (1-32 бит).

Для оценивания эффективности различных механизмов использования памяти в CPU и GPU мы построили CPU-версию нашего GPU-решателя (с теми же структурами данных и алгоритмическими особенностями, которые были описаны выше). При его тестировании на SAT-задачах, кодирующих криптоанализ A5/1, было обнаружено, что все 100% запросов к памяти обслуживаются из кэша. Аналогичная картина наблюдается и для сторонних решателей (например, minisat [20]). При работе с памятью GPU на тех же самых тестовых задачах мы наблюдаем совершенно иную картину.

Далее мы демонстрируем, как группировка запросов к памяти и SIMD-эффект влияют на эффективность выполнения DPLL на GPU. Для этой цели мы специально построили различные тестовые версии решателя, в которых искусственно вводились негативные и позитивные факторы, влияющие на сериализацию запросов к памяти и инструкций в SIMD-группе, и, как следствие - на производительность приложения на GPU (Табл. 2).

Опишем модификации, применявшиеся для получения тестовых вариантов приложения:

1. Нормальная / перемешанная индексация памяти. Если случайным образом перемешать номера вычислительных потоков между SIMD-группами, SIMD-группе придется «собирать» данные из разбросанных областей памяти, что гарантированно спровоцирует эффект разреженного доступа.

[2]Вместительность и ПСП кэшей и основной памяти CPU и GPU приведены на примере архитектур Intel Nehalem и GPU Nvidia Fermi. ПСП для CPU приведены для 1 потока/ядра [19], для GPU – для всего устройства целиком (оценены по материалам [4]).

[3]В зависимости от степени от степени выраженности описанного выше «эффекта разреженного доступа».

2. Одинаковые / разные подзадачи в SIMD-группе. Отдавая всем потокам внутри SIMD-группы на решение одну и ту же подзадачу, можно гарантировать отсутствие SIMD-эффекта, т.к. последовательность выполнения потоками инструкций при одинаковых начальных условиях строго детерминирована.
3. Параллельное / последовательное выполнение инструкций в SIMD-группе. Для гарантированного получения 100% SIMD-эффекта, достаточно сделать так, чтобы внутри одной SIMD-группы алгоритм, выполнялся потоками последовательно: вначале код алгоритма полностью выполняется только одним потоком, затем полностью выполняется другим потоком и т.д.

В (Табл. 2) представлены 8 тестовых вариантов приложения, реализующих все возможные сочетания описанных выше модификаций нашей реализации DPLL для GPU. Тестирование производилось на КНФ кодирующей задачу криптоанализа DES. Обмены задачами между потоками (work stealing) были выключены. Порядок угадывания переменных в различных подзадачах выбирался случайным образом, что соответствует состоянию решателя после продолжительной работы. В качестве меры производительности использовалась скорость просмотра литералов решателем в процедуре BCP.

Таблица 2. Производительность DPLL на GPU в зависимости от режима доступа к памяти и SIMD-эффекта. Тестовый вариант №6* соответствует рабочему режиму решателя.

Тестовый вариант приложения	**№1**	**№2**	**№3**	**№4**	**№5**	**№6***	**№7**	**№8**
Индексация памяти: (+) нормальная, (-) перемешанная.	-	+	-	+	-	+	-	+
Подзадачи в SIMD-группе: (+) одинаковые, (-) разные.	-	-	+	+	-	-	+	+
Обр. инструкций: (+) паралл., (-) последовательная.	-	-	-	-	+	+	+	+
(+) 100% эффективность доступа к памяти.	-	-	-	-	-	-	-	+
(+) 100% отсутствие «SIMD-эффекта».	-	-	-	-	-	-	+	+
Производительность DPLL, млн. просмотров в секунду.	17,5	17,5	17,0	15,6	28,9	31,2	37,6	471,3

Из данных (Табл. 2) становится видно, что только при 100% эффективности доступа к памяти (вариант №8) DPLL демонстрирует высокий уровень производительности на GPU. Вариант №8 в десятки раз опережает другие тестовые варианты, включая №6, соответствующий рабочему режиму решателя. 100% эффективность доступа к памяти в варианте №8 является следствием полного отсутствия сериализации обращений к памяти в результате эффекта разреженного доступа, SIMD-эффекта, или по иным причинам. Если бы на производительность DPLL на GPU негативно влиял только «SIMD-эффект», разница в производительности между вариантами №7 и №8 не была бы столь велика.

Непредсказуемость пути поиска в DPLL, перемешанная индексация памяти, последовательное выполнение инструкций – любой из этих факторов приводит к 100% сериализации обращений к памяти, что буквально обрушивает производительность GPU, снижая ее в десятки раз. Следует обратить внимание на то, что искусственное форсирование 100% сериализации описанными выше способами слабо влияет на скорость выполнения DPLL на GPU в рабочем режиме (переход от №6 к №7). Это говорит о том, что обращения к памяти в рабочем режиме DPLL совершенно хаотичны, вызывают сильнейший «эффект разреженного доступа» и, как следствие, обрабатываются почти на 100% в серийном режиме. Фактически, это сводит на нет все преимущества SIMD-архитектуры.

Кроме того, в силу сложности алгоритма, скомпилированный для GPU код DPLL использует довольно много регистровой памяти. Из-за этого нам пришлось искать компромисс между запуском числа SIMD-групп, достаточного для компенсации латентности памяти (высокий показатель «occupancy»), и эффектом переполнения регистровой памяти («register spilling») [4]. Это означает, что требования DPLL к числу регистров слишком велики для GPU поколения Fermi.

Таким образом, архитектура памяти современных GPU совершенно не подходит для выполнения сложных комбинаторных алгоритмов, таких как DPLL, в силу следующих причин:

1) Латентность доступа к основной памяти GPU высока.

2) Низкое число регистров в мультипроцессорах не позволяет запускать на них достаточно SIMD-групп, чтобы компенсировать латентность доступа к памяти за счет переключения между ними.

3) Контроллер доступа к памяти в GPU рассчитан на выборку сплошных диапазонов памяти, лежащих в пределах одной кэш-линии и передачу этой информации мультипроцессору целиком. Однако, при решении реальных задач с помощью алгоритма DPLL вероятность того, что хотя бы 2 потока одной SIMD-группы обратятся одновременно к одной кэш-линии, крайне мала. На практике это приводит к очень высокому эффекту разреженного доступа и падению эффективной ПСП в десятки раз.

4) Малый объем кэш-памяти GPU не позволяет компенсировать проблемы с ПСП и латентностью памяти.

4. Сравнение производительности алгоритмов полного перебора и DPLL на CPU и GPU

В (Табл. 3) приведены результаты тестирования скорости криптоанализа генератора A5/1 и блочного шифра DES на GPU и CPU, реализованного с помощью алгоритмов прямого перебора и SAT-подхода (алгоритм DPLL). Для алгоритмов прямого перебора скорость оценивалась как число ключей, проверяемых за единицу времени («кл/с»).

В силу применения алгоритма work stealing поведение решателя на GPU не является строго детерминированным. Поэтому в качестве показателя скорости работы алгоритма DPLL, совместимого между GPU и CPU, использовалось число просмотров литералов в дизъюнктах в процедуре BCP за единицу времени («п/с»). При этом в версии DPLL для GPU число просмотров литералов суммировалось по всем потокам.

Для тестирования были использованы CPU Intel Core i7 930 и GPU Nvidia Geforce GTX560Ti. На CPU многопоточность не использовалось.

Также, для оценки потенциальных вычислительных возможностей GPU, каждый алгоритм был протестирован в специальной версии, в которой все вычислительные потоки устройства выполняли одинаковую подзадачу, независимо друг от друга. В таком случае доступ к памяти организован полностью в соответствии с рекомендациями производителя, и потери производительности из-за условных переходов сведены к нулю (100% «memory coalescence» и нулевой «warp divergence» в терминологии NVIDIA [4]).

Таблица 3. Сравнение скорости комбинаторных алгоритмов на CPU и GPU

Алгоритм	CPU	GPU	Коэфф. ускорения	GPU тест, одинак. подзадачи.	Коэфф. ускорения
Брутфорс A5/1	48 млн. кл/с	9761 млн. кл/с	203,3	12271 млн. кл/с	255,6
Брутфорс DES	12 млн. кл/с	1493 млн. кл/с	124,4	1908 млн. кл/с	159,0
DPLL A5/1	21 млн. п/с	84 млн. п/с	4,0	650 млн. п/с	30,9
DPLL DES	25 млн. п/с	195 млн. п/с	5,0	1053 млн. п/с	42,1

Данные, представленные в (Табл. 3) показывают, что алгоритм DPLL не выигрывает от переноса на GPU, в отличие от более простых способов криптоанализа.

Архитектура современных GPU плохо приспособлена для выполнения DPLL и подобных ему сложных поисковых алгоритмов. Причины этого – проблемы с условными переходами и произвольным доступом к памяти. И хотя техники реорганизации вложенных циклов и «голосований», описанные нами в п. 2, позволяют уменьшить негативные последствия «SIMD-эффекта», основным «узким местом» DPLL на GPU остается произвольный доступ к памяти.

Литература

1. http://www.top500.org

2. Lee V. W. et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU //ACM SIGARCH Computer Architecture News. – ACM, 2010. – T. 38. – №. 3. – C. 451-460.

3. Flynn M. Some computer organizations and their effectiveness //Computers, IEEE Transactions on. – 1972. – T. 100. – №. 9. – C. 948-960.

4. NVIDIA corp., CUDA C Best Practices Guide // CUDA SDK v.6.0. – 2014.

5. Percival C., Josefsson S. The scrypt Password-Based Key Derivation Function. – 2012.

6. Nohl K. Attacking phone privacy //Black Hat USA. – 2010.

7. Mironov I., Zhang L. Applications of SAT solvers to cryptanalysis of hash functions //Theory and Applications of Satisfiability Testing-SAT 2006. – Springer Berlin Heidelberg, 2006. – C. 102-115.

8. Semenov A. et al. Parallel logical cryptanalysis of the generator A5/1 in BNB-Grid system //Parallel Computing Technologies. – Springer Berlin Heidelberg, 2011. – C. 473-483.

9. Biryukov A., Shamir A., Wagner D. Real Time Cryptanalysis of A5/1 on a PC //Fast Software Encryption. – Springer Berlin Heidelberg, 2001. – C. 1-18.

10. Golić J. D. Cryptanalysis of alleged A5 stream cipher //Advances in Cryptology—EUROCRYPT'97. – Springer Berlin Heidelberg, 1997. – C. 239-255.

11. Kumar S. et al. Breaking ciphers with COPACOBANA–a cost-optimized parallel code breaker //Cryptographic Hardware and Embedded Systems-CHES 2006. – Springer Berlin Heidelberg, 2006. – C. 101-118.

12. Kwan M. Reducing the Gate Count of Bitslice DES //IACR Cryptology ePrint Archive. – 2000. – T. 2000. – C. 51.

13. http://www.openwall.com/john/

14. Davis M., Logemann G., Loveland D. A machine program for theorem-proving //Communications of the ACM. – 1962. – T. 5. – №. 7. – C. 394-397.

15. Moskewicz M. W. et al. Chaff: Engineering an efficient SAT solver //Proceedings of the 38th annual Design Automation Confe rence. – ACM, 2001. – C. 530-535.

16. Marques-Silva J. P., Sakallah K. A. GRASP: A search algorithm for propositional satisfiability //Computers, IEEE Transactions on. – 1999. – T. 48. – №. 5. – C. 506-521.

17. Blumofe R. D., Leiserson C. E. Scheduling multithreaded computations by work stealing //Journal of the ACM (JACM). – 1999. – T. 46. – №. 5. – C. 720-748.

18. Backus J. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs //Communications of the ACM. – 1978. – T. 21. – №. 8. – C. 613-641.

19. Molka D. et al. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system //Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on. – IEEE, 2009. – C. 261-270.

20. Een N., Sörensson N. MiniSat: A SAT solver with conflict-clause minimization //Sat. – 2005. – T. 5.

21. Chu G., Harwood A., Stuckey P.J. Cache Conscious Data Structures for Boolean Satisfiability Solvers //JSAT. – 2009. – T. 6. – №. 1-3. – C. 99-120.