Russian Technological Journal.
2021;9(6):7–15

ISSN 2782-3210 (Print)
ISSN 2500-316X (Online)

УДК 004.051
https://doi.org/10.32362/2500-316X-2021-9-6-7-15

ОБЗОР

Сравнительный анализ методов оптимизации программного обеспечения для борьбы с предикацией ветвлений на графических процессорах

И.Ю. Сесин [@], **Р.Г. Болбаков**

МИРЭА – Российский технологический университет, Москва, 119454 Россия
[@] *Автор для переписки, e-mail: isesin@protonmail.com*

Резюме. Технология GPGPU (General Purpose computing for Graphical Processing Units – расчеты общего назначения на графических процессорах) является мощным инструментом для переноса задач параллельной обработки информации на GPU (Graphical Processing Unit – графический процессор). Эта технология находит применение практически в любой области, требующей проведения массы параллельных расчетов, и применяется как в научной и коммерческой, так и в любительской среде. Разработчики программ общего назначения, запускаемых на GPU, неизбежно сталкиваются с падением производительности ввиду предикации ветвления кода. В условиях предикации ветвления исполняются обе ветви условного оператора вне зависимости от истинности условия, но посредством маскирования выполняемых инструкций программа учитывает только результат работы верной ветви. Из-за этого программы общего назначения, имеющие большие участки кода, скрытые за условными операторами, становятся существенно менее производительными на графических процессорах. В статье рассматриваются существующие в предметной области методы и подходы к увеличению производительности программного обеспечения в рамках их применимости к решению проблемы падения производительности при предикации. Приводится описание методов, их сильных и слабых сторон, а также рамок их применимости, на базе чего делается заключение о возможности их использования на GPU. В число рассмотренных методов и подходов вошли следующие: оптимизирующие компиляторы, JIT-компиляция, предсказатель переходов, спекулятивное исполнение, адаптивная оптимизация, специализация алгоритма во время исполнения, оптимизация на основе профилирования. Показано, что указанные аппаратные и программные подходы к увеличению производительности программного обеспечения преимущественно ориентированы на решение проблем специфичных для CPU (Central Processing Unit – центральный процессор) и в целом неприменимы для разрешения потерь производительности при предикации на GPU. Указывается на необходимость создания отдельного подхода, ориентированного именно на решение проблемы предикации ветвления на GPU.

Ключевые слова: расчеты общего назначения на графических процессорах, оптимизирующие компиляторы, предикация

• **Поступила:** 22.03.2021 • **Доработана:** 26.05.2021 • **Принята к опубликованию:** 25.07.2021

Для цитирования: Сесин И.Ю., Болбаков Р.Г. Сравнительный анализ методов оптимизации программного обеспечения для борьбы с предикацией ветвлений на графических процессорах. *Russ. Technol. J*. 2021;9(6):7–15. https://doi.org/10.32362/2500-316X-2021-9-6-7-15

Прозрачность финансовой деятельности: Никто из авторов не имеет финансовой заинтересованности в представленных материалах или методах.

Авторы заявляют об отсутствии конфликта интересов.

REVIEW ARTICLE

Comparative analysis of software optimization methods in context of branch predication on GPUs

Igor Yu. Sesin [@], **Roman G. Bolbakov**

MIREA – Russian Technological University, Moscow, 119454 Russia
[@] *Corresponding author, e-mail: isesin@protonmail.com*

Abstract. General Purpose computing for Graphical Processing Units (GPGPU) technology is a powerful tool for offloading parallel data processing tasks to Graphical Processing Units (GPUs). This technology finds its use in variety of domains – from science and commerce to hobbyists. GPU-run general-purpose programs will inevitably run into performance issues stemming from code branch predication. Code predication is a GPU feature that makes both conditional branches execute, masking the results of incorrect branch. This leads to considerable performance losses for GPU programs that have large amounts of code hidden away behind conditional operators. This paper focuses on the analysis of existing approaches to improving software performance in the context of relieving the aforementioned performance loss. Description of said approaches is provided, along with their upsides, downsides and extents of their applicability and whether they address the outlined problem. Covered approaches include: optimizing compilers, JIT-compilation, branch predictor, speculative execution, adaptive optimization, run-time algorithm specialization, profile-guided optimization. It is shown that the aforementioned methods are mostly catered to CPU-specific issues and are generally not applicable, as far as branch-predication performance loss is concerned. Lastly, we outline the need for a separate performance improving approach, addressing specifics of branch predication and GPGPU workflow.

Keywords: general-purpose computing for graphical processing units, optimizing compilers, predication

• **Submitted:** 22.03.2021 • **Revised:** 26.05.2021 • **Accepted:** 25.07.2021

For citation: Sesin I.Yu., Bolbakov R.G. Comparative analysis of software optimization methods in context of branch predication on GPUs. *Russ. Technol. J*. 2021;9(6):7–15 (in Russ.). https://doi.org/10.32362/2500-316X-2021-9-6-7-15

Financial disclosure: The authors have no a financial or property interest in any material or method mentioned.

The authors declare no conflicts of interest.

ВВЕДЕНИЕ

Графические процессоры или GPU (Graphical Processing Unit), являются специализированным аппаратным обеспечением, выполняющим обработку графической информации. В отличие от центрального процессора (CPU) графические процессоры специализированы для параллельной обработки большого количества данных, что обуславливает определенные различия в их архитектурном построении:

- CPU имеют малое количество физических ядер (от 1 до 32), у GPU количество физических ядер на порядки больше (может исчисляться сотнями и тысячами[1,2] в зависимости от выбранного производителем аппаратного обеспечения баланса между числом ядер и их мощностью);
- ядра в CPU обладают дополнительным аппаратным функционалом, таким как собственные кеши, конвейеры инструкций и модули предсказания ветвлений. Ядра GPU являются очень простыми арифметическими ядрами, ориентированными на быструю работу с числами с плавающей точкой. Ряд современных GPU также включает специализированные ядра для трассировки лучей и тензорных вычислений [1, 2];
- CPU отделен от оперативной памяти, GPU имеют прямой доступ к видеопамяти, размещенной на той же плате;
- ядра GPU не могут записывать данные в области памяти, отведенные под исполняемый код;
- CPU попадает в классификацию MIMD (Multiple Instruction, Multiple Data), GPU попадает в классификацию SIMD (Single Instruction, Multiple Data) по Флинну [3].

С использованием технологии GPGPU (General Purpose computing on Graphical Processing Units – расчеты общего назначения на графических процессорах) становится возможным запуск программ на GPU, отличных от узкоспециализированных программ-шейдеров. Эта технология находит применение во многих областях – от майнинга криптовалют до обсчета сворачивания белков[3].

Применяя технологию GPGPU, приходится столкнуться с определенными особенностями построения программ и их поведения в условиях исполнения на графическом процессоре. Так, например, в ходе написания GPGPU программы авторами настоящей статьи был замечен тренд падения ее производительности по мере добавления функционала. При этом потери в производительности были гораздо более существенны, чем можно было бы предположить исходя из сложности алгоритма добавляемого функционала.

В ходе исследования вышеуказанной проблемы было установлено, что это случается в силу специфики SIMD архитектуры графических процессоров: при исполнении программой условного оператора возникает одна особенность – будут выполнены обе ветви, но операции неправильной ветви не будут применены. Эта особенность называется предикацией [4, с. 168] веток исполнения, и она нужна первоочередно для замещения зависимости от потока исполнения на зависимость от данных. Сама необходимость подобных мер оправдана тем, что большая аппаратная часть графического процессора архитектурно попадает в класс SIMD (Single Instruction, Multiple Data) таксономии Флинна [3]. Отдельные модули системы не могут иметь свои потоки исполнения и, соответственно, не могут следовать по различным ветвям условного оператора.

Степень влияния вышеуказанной особенности на программу разнится от программы к программе и коррелирует с количеством возможных настроек. Так, если программа изначально предназначена для решения одной максимально конкретизированной задачи без возможности настройки (например, расчет определенной хеш-функции), то влияние предикации будет сведено к минимуму.

Более подвержены падению производительности программы, выполняющие более общие задачи, и, следовательно, обладающие большим списком подключаемого или необязательного функционала, который будет задействован в процессе работы программы только при определенных входных данных.

Возьмем в качестве примера такой программы рендерер трехмерных сцен. Рендерингом в области компьютерной графики принято называть процесс получения изображения из определенных данных, характеризующих объекты отображаемой сцены. Программу, реализующую такой процесс, называют рендерером. Профессиональные рендереры могут включать в себя тысячи различных опций, изменяющих поведение программы – от настройки угла поля зрения камеры до детальной конфигурации двулучевой функции отражательной способности для каждой поверхности.

Взяв за пример программу-рендерер, представим ситуацию, в которой осуществляется рендеринг некоторой сцены с набором примитивов, которые могут лишь рассеивать или излучать свет, но не отражать его. В таком случае код, ответственный за расчет отражений, не должен вызываться, но само его наличие замедлит программу из-за предикации.

---

[1] Advanced Micro Devices, Inc, Graphics Specifications, 2021. URL: https://www.amd.com/en/products/specifications/graphics, дата обращения 01.03.2021. [Advanced Micro Devices, Inc, Graphics Specifications, 2021. URL: https://www.amd.com/en/products/specifications/graphics. Accessed March 1, 2021.]

[2] NVIDIA. Сравнение спецификаций видеокарт RTX 30. URL: https://www.nvidia.com/ru-ru/geforce/graphics-cards/30-series/compare/?section=compare-specs, дата обращения 01.03.2021. [NVIDIA. Comparison of specifications of RTX 30 video cards. URL: https://www.nvidia.com/ru-ru/geforce/graphics-cards/30-series/compare/?section=compare-specs. Accessed March 1, 2021.]

[3] Houston M. General Purpose Computation on Graphics Processors (GPGPU). ATI HD 2000 Series. Launch, Tunis, Tunisia; 2007. URL: https://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf, дата обращения 01.03.2021. [Houston M. General Purpose Computation on Graphics Processors (GPGPU). ATI HD 2000 Series. Launch, Tunis, Tunisia; 2007. URL: https://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf. Accessed March 1, 2021.]

Для включения или выключения желаемого поведения программы в подавляющем большинстве случаев используются именно условные операторы, зависящие от некоторых входных данных программы, будь то просто данные или конфигурационные настройки.

Вместе с ростом функционала программы будет расти число условных операторов, соответственно, будут расти и издержки, затрачиваемые графическим процессором на обработку условных ветвей программы. Это, в свою очередь, приводит к появлению определенного потолка усложнения программы – после определенной критической массы ветвей программа замедлится настолько, что сработает внутренняя защита драйвера (при ее наличии) и выполнит аварийную остановку исполнения.

Примечательно, что с точки зрения программиста само наличие дополнительного кода замедляет программу, несмотря на то, что код не должен исполняться. Подобное поведение крайне нехарактерно для программ, исполняемых на CPU. Более того, правильное использование условных переходов зачастую является ключом к написанию более эффективных программ за счет предсказателя переходов, встроенного в ядра современных CPU.

ИССЛЕДОВАНИЕ СУЩЕСТВУЮЩИХ ПРОГРАММНО-АППАРАТНЫХ СРЕДСТВ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Рассмотрим существующие технологии оптимизации программ и увеличения производительности, оценивая их применимость для разрешения описанной проблемы.

Следует отметить, что в рамках этой статьи не будет попыток численного сравнения рассматриваемых методов. Это связано с тем, что практически все методы повышения производительности программного обеспечения опираются на определенную специфику оптимизируемой программы, аппаратной платформы или языка программирования. Методы, эффективные для одних программ, могут быть бесполезны и даже вредны для других. Качество внедрения того или иного метода также оказывает прямое влияние на получаемый результат, и один и тот же подход, внедренный по-разному, может давать разительно отличающиеся результаты. Конкретная метрика успешности применения методов зависит от многих факторов, как количественных, так и качественных, которые не могут быть исключены из исследования, не ставя под удар воспроизводимость самого исследования. С другой стороны, включение этих факторов сузит исследование до сравнения частных случаев, а именно, сравнения конкретных программ на конкретном аппаратном обеспечении в конкретной конфигурации, что нерепрезентативно для описания общей картины.

В силу вышеуказанного авторы рассматривают главным образом качественные характеристики в ходе определения применимости методов улучшения производительности ПО.

Одну из ведущих ролей в повышении производительности программного обеспечения продолжают занимать оптимизирующие компиляторы. В общем случае оптимизирующим компилятором можно назвать любой компилятор, выполняющий специальные операции над кодом для улучшения его производительности.

Изначально это подразумевало замену определенных операций на эквивалентные, но более эффективные, как например, замена умножения или целочисленного деления на числа, являющиеся степенями двойки, на операции сдвигов. По мере развития научной области, методологий написания кода и аппаратного обеспечения компиляторы обзавелись обширным арсеналом возможностей оптимизации программного кода.

Их принято разделять на низкоуровневые и высокоуровневые оптимизации.

Низкоуровневые оптимизации подразумевают внесение изменений в генерируемый машинный код таким образом, чтобы максимально эффективно использовать особенности аппаратной платформы. Это включает в себя использование более производительных конструкций машинного кода, в том числе и с использованием специальных команд, доступных в целевой архитектуре, векторизации операций, встраивания функций и т.д.

Высокоуровневые оптимизации оперируют на уровне абстрактных алгоритмических элементов, составляющих программы – циклов, ветвлений, базовых блоков. Они используют данные о структуре программы, чтобы преобразовывать промежуточное представление программы в более эффективную форму.

На практике граница между этими группами размыта, так как многие методы вовлекают в себя элементы как высокоуровневого анализа кода, так и низкоуровневого управления генерацией машинных команд.

В число широко применяемых компиляторами оптимизаций входят:

- схлопывание константных выражений – если определенное выражение состоит только из констант, то оно рассчитывается на этапе компиляции и его результат подставляется вместо оригинального выражения;
- устранение «мертвого кода» (англ. – dead code [5]) – исключение участков кода, которые не могут быть достигнуты программой;

- устранение «тупиковых записей» (англ. – dead store [6]) – исключение записи значения в переменную, значение которой не используется в коде далее;
- оптимизация использования регистров общего назначения процессором [7] реорганизует код таким образом, чтобы минимизировать количество обращений к памяти в процессе работы программы посредством удержания наиболее часто используемых переменных в определенных регистрах общего назначения процессора;
- распараллеливание операций – изменение порядка операций таким образом, чтобы они могли быть запущены параллельно на уровне потоков, памяти или инструкций;
- снижение стоимости операций – замена медленных операций на эквивалентные, но более быстрые на целевой архитектуре;
- оптимизации циклов – обширная группа методов, ориентированная на работу с циклами, включает такие подходы как вынос инвариантов за рамки цикла, инверсию циклов, развертку циклов, разделение и слияние тел циклов, вынос условных операторов из цикла и т.д.;
- выбор инструкций [8] – позволяет выбрать из нескольких вариантов машинного кода, выполняющих одно и то же действие, наиболее эффективный для архитектуры целевого процессора;
- Instruction Scheduling [9, 10] – реорганизация инструкций таким образом, чтобы не вызывать простои конвейера центрального процессора в результате долгих обращений к памяти, исчерпания ресурсов процессора или ветвления.

Большинство традиционных методов оптимизации программ, используемых в оптимизирующих компиляторах, можно исключить из рассмотрения, так как они ориентированы на особенности CPU, не присущие арифметическим ядрам, использующимся в графических процессорах. Те методы, что могут быть применены, либо уже внедрены компиляторами программ для GPU, либо не решают представленную проблему.

Перейдем к рассмотрению более динамических методов увеличения производительности.

Технология JIT-компиляции [11] (just-in-time) позволяет улучшить производительность программ на языках, компилирующихся в байт-код. Байт-кодом называют промежуточное представление программного кода [12–14], которое исполняется виртуальной машиной, что является его ключевым отличием от машинного кода, исполняемого непосредственно процессором.

В рамках технологии JIT-компиляции байт-код программ компилируется в машинный код по мере надобности прямо во время работы программы[4]. Это позволяет ускорить работу программ сразу с нескольких сторон:

- значительно уменьшается задержка при изначальном запуске программы, которая без этой технологии вызывается длительной обработкой исходного кода; компиляция из исходного кода в байт-код происходит существенно медленнее, чем компиляция из байт-кода в машинный код;
- использование особенностей конкретного аппаратного обеспечения для улучшения производительности;
- оптимизация с использованием данных, полученных во время работы программы;
- возможность динамического подключения библиотек без накладных расходов, присущих компилируемым языкам.

Однако применение JIT-компиляции в рамках программ, запущенных на GPU, на данный момент не представляется возможным из-за архитектурных ограничений на динамическое изменение исполняемого машинного кода на графических процессорах.

Помимо этого, встраивание компиляции в сам цикл работы графического процессора потребует массу как аппаратных, так и программных изменений и в целом не даст такого же прироста производительности, как на центральном процессоре, т.к. на GPU нет проблемы подключения библиотек (ввиду статической компиляции программ). При этом промежуточное представление программы и так будет полностью скомпилировано в машинный код в процессе переноса программы на графический процессор. В целом, указанные проблемы делают применение JIT нецелесообразным на GPU.

Одним из ключевых элементов, обеспечивающих быстродействие современных CPU, является предсказатель переходов[5]. Предсказатель переходов позволяет процессору, использующему конвейерную архитектуру, начать загружать инструкции одной из ветвей оператора ветвления в конвейер прежде, чем будет определена истинность условия.

---

[4] Croce L. Just in Time Compilation. Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22_Croce_JIT.pdf, дата обращения 01.03.2021. [Croce L. Just in Time Compilation. Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22_Croce_JIT.pdf. Accessed March 1, 2021.]

[5] Fog A. The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers. Technical University of Denmark. URL: https://www.agner.org/optimize/microarchitecture.pdf, дата обращения 01.03.2021. [Fog A. The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers. Technical University of Denmark. URL: https://www.agner.org/optimize/microarchitecture.pdf. Accessed March 1, 2021.]

Это играет важную роль в повышении производительности за счет параллельного исполнения инструкций, оперирующих на взаимно независимых данных. В общем случае, без предсказателя конвейер процессора не будет знать, инструкции из какой ветви следует использовать, и сможет начать их загрузку только после вычисления истинности условия в условном операторе, что означает простой конвейера на каждом из условий. Этого можно избежать, предсказав результат условия и начав загрузку инструкций из соответствующей ветки в конвейер заранее. В случае ошибки предсказателя произойдет простой конвейера, пока не будут загружены правильные инструкции.

Само предсказание ветвления осуществляется эвристически [15–17], как правило, опираясь на статистику исполнения данного участка кода. Временны́е издержки при ошибке предсказателя неизбежны по причине несовершенности механизмов предсказания. Однако если механизм достаточно точен, то временны́е потери от неправильного предсказания становятся незначительными по сравнению с выигрышем, полученным в ходе верных предсказаний.

Подход предсказания ветвления получил развитие в виде спекулятивного исполнения[6] – помимо загрузки инструкций на конвейер также осуществляется и исполнение предсказанной ветви прежде, чем истинность условия будет установлена. В случае ошибки вырастают временны́е потери, т.к. требуется целиком очистить конвейер и загрузить в него инструкции правильной ветви.

Подходы предсказания ветвления и спекулятивного исполнения исполнены аппаратно и не видны программисту. Тем не менее, при определенных построениях программ возможно получать снижение производительности, что наиболее заметно в случае, когда программа стабильно заставляет модуль выдать неправильное предсказание перехода. Существующая практика оптимизации низкоуровневых программ для более удачного взаимодействия с предсказателем переходов является прямым следствием несовершенности применяемых способов предсказания.

Само наличия механизма предсказания ветвления и спекулятивного исполнения также открыло путь целому классу аппаратных уязвимостей, позволяющих программе получить несанкционированный доступ к информации. Специальным образом сконструированная программа может установить содержимое ячеек оперативной памяти при помощи косвенных данных (как правило, о быстродействии), возникающих в ходе ошибочных предсказаний ветвлений [18–20].

По мнению авторов настоящей статьи, предсказание переходов на GPU не имеет смысла ввиду отсутствия конвейера на его ядрах, а также отсутствия ветвлений в исполняемом машинном коде в типовом их понимании из-за предикации. Вместо этого код структурирован линейно и при исполнении условных операторов не происходит «прыжков» регистра счетчика инструкций.

Даже если предположить, что данное препятствие будет преодолено, внедрение модуля предсказания перехода на GPU крайне затруднительно ввиду сложности подобного модуля. Типовой графический процессор включает в себя большое количество слабых арифметических ядер, на порядки превышающее количество ядер на типовом рыночном центральном процессоре. Добавление к каждому из них предсказателя увеличит их размер и энергопотребление до неприемлемых значений.

Метод адаптивной оптимизации [21] полагается на специальный инструментарий, схожий с JIT-компиляторами, чтобы производить рекомпиляцию кода программы в процессе ее работы. Этот метод нацелен на применение различных подходов к оптимизации программы в зависимости от продуктивности кода. Так, например, адаптивная оптимизация будет применять более агрессивные методы оптимизации в «узких местах», где код тратит больше всего процессорного времени. Подобные агрессивные методы оптимизации занимают много времени, и применять их на весь объем программы было бы слишком расточительно. Поэтому участки программы, которые вызываются редко, будут оптимизированы более примитивными, но более быстрыми приемами, позволяя сэкономить время в целом, пусть и за счет того, что этот код будет работать медленнее.

Применение вышеуказанного подхода в отношении GPU невозможно по схожей с JIT причине – для его работы требуется возможность для программы перезаписывать свой код, что не поддерживается современными GPU.

Рассмотрим метод специализации алгоритма во время исполнения (англ. – run-time algorithm specialization) – подход, пришедший из области автоматического доказательства теорем [22]. Указанный подход подразумевает создание частных реализаций

[6] Gabbay F. Speculative execution based on value prediction. Research Proposal towards the Degree of Doctor of Sciences. Technion-Israel Institute of Technology (IIT), Department of Electrical Engineering. 1996. 65 p. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.5397&rep=rep1&type=pdf, дата обращения 01.03.2021. [Gabbay F. Speculative execution based on value prediction. Research Proposal towards the Degree of Doctor of Sciences. Technion-Israel Institute of Technology (IIT), Department of Electrical Engineering. 1996. 65 p. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.5397&rep=rep1&type=pdf. Accessed March 1, 2021.]

сложных функций для определенных входных данных, для которых возможно представить исходную функцию в более простой форме. Иногда это может выражаться в создании табличных значений для функции и кешировании отдельных результатов для часто повторяющихся значений. В более сложных случаях может создаваться несколько вариантов кода, соответствующих частным вариантам функции, где, например, исключается часть расчетов из-за специфических значений входных данных.

Вышеуказанный подход и схожие с ним методы [23] требуют не только записи в область памяти исполняемого кода, но и большого количества дополнительной памяти под код полученных специализаций, что делает его применение в рамках технологии GPGPU невозможным.

Перейдем к рассмотрению технологии оптимизации на основе профилирования (англ. – profile-guided optimization) – подхода к оптимизации программ, в котором процессом оптимизации управляют результаты быстродействия программы [24, 25]. Этот подход зачастую требует использования специального компилятора, который берет на себя задачу инструментации и снятия замеров времени исполнения программы. Подобный специальный компилятор транслирует, собирает и запускает программу множество раз, анализируя в процессе этих тестовых запусков частоту использования различных участков программы и скорость их вызова. Используя эти данные, компилятор применяет различные стратегии оптимизации к участкам кода, которые показали плохие результаты в ходе тестовых запусков. В число подобных стратегий входит подбор оптимального распределения регистров, встраивание функций, а также масса техник, завязанных на удачную группировку исполняемого кода в страницах памяти для ускорения работы кешей сразу на нескольких уровнях – от механизма управления виртуальной памятью на уровне ОС до кеша инструкций CPU.

В связи с тем, что собираемая подобным компилятором статистика репрезентативна только для тех действий, которые осуществлялись программой в процессе тестирования, типовой набор тестов должен включать в себя наиболее общие сценарии работы с программой. Это могло бы быть реализовано гораздо проще для GPU-программ в силу отсутствия у них интерактивности.

По мнению авторов статьи, применение этого метода для GPU-программ нецелесообразно. В условиях работы с GPU крайне затруднительно или даже невозможно проводить инструментацию кода и, соответственно, получать замеры производительности отдельных участков кода, необходимые для работы метода.

Из анализа существующих технологий повышения производительности видно, что проблема потери производительности с наращиванием функционала, характерная для графических процессоров, является атипичной и нетривиальной.

Ни один из рассматриваемых подходов не решает проблему предикации ветвлений на графических процессорах, однако они являются важной отправной точкой в процессе нахождения решения.

С одной стороны, для решения проблемы требуется применение динамических методов, чтобы сохранить гибкость функционала программы для графического процессора. С другой стороны, рассмотренные существующие решения, применяемые на CPU, не могут быть использованы в контексте графических процессоров на практике из-за характерных особенностей архитектуры, ориентированной на массово параллельные расчеты. Внесение изменений в устоявшуюся архитектуру GPU очевидным образом нецелесообразно ввиду ряда факторов, среди которых многократное увеличение цены производства и неизбежное снижение производительности.

В силу изложенных положений можно установить, что существует потребность в разработке подхода к оптимизации программ для GPU, совместимого с ограничениями графических процессоров. В то же время этот подход должен быть достаточно гибким, чтобы обеспечивать сохранение всего многообразия функционала оптимизируемых программ.

ЗАКЛЮЧЕНИЕ

В статье рассмотрена проблема уменьшения производительности программ для расчетов общего назначения на GPU, возникающая в ходе наращивания их функционала. Установлена связь этой проблемы с предикацией ветвления – существенной особенностью организации аппаратной платформы графических процессоров.

Рассмотрен ряд существующих подходов и технологий для увеличения производительности программ на CPU и показана малая степень их применимости в условиях программ, использующих технологию GPGPU.

Отмечено, что существующие методы, несмотря на невозможность их прямого применения, являются важной отправной точкой в дальнейших исследованиях.

Дальнейшее развитие указанной в статье проблематики подразумевает разработку специализированного метода оптимизации программ для GPU.

Вклад авторов. Все авторы в равной степени внесли свой вклад в исследовательскую работу.

Authors' contribution. All authors equally contributed to the research work.

СПИСОК ЛИТЕРАТУРЫ / REFERENCES

1. Markidis S., Chien S.W.D., Laure E., Peng I.B., Vetter J.S. NVIDIA Tensor Core Programmability, Performance & Precision. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Vancouver, BC, Canada; 2018, p. 522–531. https://doi.org/10.1109/IPDPSW.2018.00091
2. Sanzharov V.V., Gorbonosov A.I., Frolov V.A., Voloboy A.G. Examination of the Nvidia RTX. *CEUR Workshop Proceedings*. 2019;2485:7–12. http://dx.doi.org/10.30987/graphicon-2019-2-7-12
3. Flynn M.J. Very high speed computing systems. *Proceedings of the IEEE*. 1966;54(12):1901–1909. https://doi.org/10.1109/PROC.1966.5273
4. Fisher J.A., Faraboschi P., Young C. *Embedded computing: A VLIW approach to architecture, compilers, and tools*. Elsevier; 2004. ISBN: 978-1-55860-766-8. URL: https://www.researchgate.net/publication/220690439_Embedded_computing_a_VLIW_approach_to_architecture_compilers_and_tools
5. Knoop J., Rüthing O., Steffen B. Partial dead code elimination. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*. 1994, p. 147–158. https://doi.org/10.1145/178243.178256
6. Fink S., Knobe K., Sarkar V. Unified analysis of array and object references in strongly typed languages. In: Palsberg J. (Ed.). *Static Analysis. SAS 2000. Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 2000. V. 1824. P. 155–174. https://doi.org/10.1007/978-3-540-45099-3_9
7. Runeson J., Nyström S.-O. Retargetable graph-coloring register allocation for irregular architectures. In: Krall A. (Ed.). *Software and Compilers for Embedded Systems (SCOPES 2003). Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 2003. V. 2826. P. 240–254. https://doi.org/10.1007/978-3-540-39920-9_17
8. Blindell G.H. *Instruction Selection: Principles, Methods, & Applications*. Springer; 2016. 171 p. ISBN 978-3-319-34017-3. http://dx.doi.org/10.1007/978-3-319-34019-7
9. Gibbons P.B., Muchnick S.S. Efficient instruction scheduling for a pipelined architecture. *ACM SIGPLAN Notices*. 1986;21(7):11–16. https://doi.org/10.1145/13310.13312
10. Su Ch.-L., Tsui Ch.-Y., Despain A.M. Low power architecture design and compilation techniques for high-performance processors. In: *Proceedings of COMPCON '94*. 1994, p. 489–498. https://doi.org/10.1109/CMPCON.1994.282878
11. Aycock J. A brief history of just-in-time. *ACM Comput. Surv.* 2003;35(2):97–113. https://doi.org/10.1145/857076.857077
12. Ogihara M. *Fundamentals of Java Programming*. Springer; 2018. 532 p.
13. Sage K. The Origins of Programming. In: *Concise Guide to Object-Oriented Programming. Undergraduate Topics in Computer Science*. Springer, Cham.; 2019, p. 1–9. https://doi.org/10.1007/978-3-030-13304-7_1
14. Saabith A.S., Fareez M.M.M., Vinothraj T. Python current trend applications-an overview. *IJAERD*. 2019;6(10):6–12. URL: http://ijaerd.com/papers/finished_papers/IJAERDV06I1085481.pdf
15. McFarling S. *Combining Branch Predictors*. Digital Western Research Lab (WRL). Technical Report, TN-36. 1993. 29 p. URL: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf
16. Skadron K., Martonosi M., Clark D.W. A Taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In: *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*. Philadelphia. 2000. https://doi.org/10.1109/PACT.2000.888344
17. Vintan L.N., Iridon M. Towards a high performance neural branch predictor. In: *IJCNN'99. International Joint Conference on Neural Networks Proceedings*. 1999. https://doi.org/10.1109/IJCNN.1999.831066
18. Kocher P., Horn J., Fogh A., Genkin D., *et al*. Spectre attacks: Exploiting speculative execution. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. 19 p. https://doi.org/10.1109/SP.2019.00002
19. Bhattacharyya A., Sandulescu A., Neugschwandtner M., Sorniotti A., *et al*. SMoTherSpectre: exploiting speculative execution through port contention. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 2019, p. 785–800. https://doi.org/10.1145/3319535.3363194
20. Chen G., Chen S., Xiao Y., Zhang Y., Lin Z., Lai T.H. SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, p. 142–157. https://doi.org/10.1109/EuroSP.2019.00020
21. Arnold M., Fink S., Grove D., Hind M., Sweeney P.F. Adaptive optimization in the Jalapeno JVM. In: *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2000, p. 47–65. https://doi.org/10.1145/353171.353175
22. Riazanov A. *Implementing an Efficient Theorem Prover*. PhD thesis. The University of Manchester; 2003. 210 p. URL: https://www.researchgate.net/publication/2906405_Implementing_an_Efficient_Theorem_Prover
23. Grant B., Mock M., Philipose M., Chambers C., Eggers S.J. Annotation-directed run-time specialization in C. *ACM SIGPLAN Not.* 1997;32(12):163–178. https://doi.org/10.1145/258994.259016
24. Pettis K., Hansen R.C. Profile guided code positioning. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90)*. 1990, p. 16–27. https://doi.org/10.1145/93542.93550
25. Wicht B., Vitillo R.A, Chen D., Levinthal D. *Hardware Counted Profile-Guided Optimization*. 24 November 2014. URL: https://arxiv.org/pdf/1411.6361.pdf

Об авторах

Сесин Игорь Юрьевич, аспирант, кафедра инструментального и прикладного программного обеспечения Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: isesin@protonmail.com. https://orcid.org/0000-0002-7323-9595

Болбаков Роман Геннадьевич, к.т.н., доцент, заведующий кафедрой инструментального и прикладного программного обеспечения Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: bolbakov@mirea.ru. https://orcid.org/0000-0002-4922-7260

About the authors

Igor Yu. Sesin, Postgraduate Student, Department of the Tool and Applied Software, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: isesin@protonmail.com. https://orcid.org/0000-0002-7323-9595

Roman G. Bolbakov, Cand. Sci. (Eng.), Associate Professor, Head of the Department of the Tool and Applied Software, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: bolbakov@mirea.ru. https://orcid.org/0000-0002-4922-7260