MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Autotuning Parallel Programs by Model Checking

N. O. Garanina[1,2], S. P. Gorlatch[3]

DOI: 10.18255/1818-1015-2021-4-338-355

[1]A.P. Ershov Institute of Informatics Systems (IIS), Siberian Branch of the Russian Academy of Sciences, 6 Acad. Lavrentjev pr., Novosibirsk 630090, Russia.
[2]Institute of Automation and Electrometry SB RAS, 1, Academician Koptyug ave., Novosibirsk 630090, Russia.
[3]University of Munster, 62 Einsteinstr, Muenster 48149, Germany.

MSC2020: 68W10
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

The paper presents a new approach to autotuning data-parallel programs. Autotuning is a search for optimal program settings which maximize its performance. The novelty of the approach lies in the use of the model checking method to find the optimal tuning parameters by the method of counterexamples. In our work, we abstract from specific programs and specific processors by defining their representative abstract patterns. Our method of counterexamples implements the following four steps. At the first step, an execution model of an abstract program on an abstract processor is described in the language of a model checking tool. At the second step, in the language of the model checking tool, we formulate the optimality property that depends on the constructed model. At the third step, we find the optimal values of the tuning parameters by using a counterexample constructed during the verification of the optimality property. In the fourth step, we extract the information about the tuning parameters from the counter-example for the optimal parameters. We apply this approach to autotuning parallel programs written in OpenCL, a popular modern language that extends the C language for programming both standard multi-core processors (CPUs) and massively parallel graphics processing units (GPUs). As a verification tool, we use the SPIN verifier and its model representation language Promela, whose formal semantics is good for modelling the execution of parallel programs on processors with different architectures.

Keywords: optimization problem; auto-tuning of parallel programs; parallel programs; GPU programming; model checking; counterexamples; OpenCL; SPIN; Promela

INFORMATION ABOUT THE AUTHORS

Natalia Olegovna Garanina
correspondence author
orcid.org/0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
Ph.D. in Mathematics, senior research fellow.

Sergei Petrovich Gorlatch
orcid.org/0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de
Prof. Dr. Habil.

Funding: This work was supported by the DAAD research scholarship of Dr. Natalia Garanina #91735805, by State assignment #AAAA-A19-119120290056-0, and by the DFG project PPP-DL at the University of Muenster, Germany.

For citation: N. O. Garanina and S. P. Gorlatch, "Autotuning Parallel Programs by Model Checking", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 338-355, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Подход к автонастройке параллельных программ методом проверки моделей

Н. О. Гаранина[1,2], С. П. Горлач[3]

DOI: 10.18255/1818-1015-2021-4-338-355

[1]Институт систем информатики имени А. П. Ершова СО РАН, пр. Лаврентьева, д. 6, г. Новосибирск, 630090 Россия.
[2]Институт автоматики и электрометрии СО РАН, пр. Акад. Коптюга, д. 1, г. Новосибирск, 630090 Россия.
[3]Университет Мюнстера, Эйнштейн-штрассе, д. 62, Мюнстер, 48149 Германия.

УДК 004.822, 681.51
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

В этой статье представлен новый подход к автонастройке программ, параллельных по данным. Автонастройка – это поиск оптимальных параметров настройки программы, при которых её производительность оказывается максимальной. Новизна подхода состоит в использовании метода проверки моделей для поиска оптимальных параметров настройки методом контрпримеров. В нашей работе мы абстрагируемся от конкретных программ и конкретных процессоров, задавая их представительные абстрактные шаблоны. Наш метод контрпримеров состоит в реализации следующих четырёх шагов. На первом шаге на языке инструмента проверки моделей задаётся модель исполнения абстрактной программы на абстрактном процессоре. На втором шаге на языке инструмента проверки моделей формулируем свойство оптимальности, зависящее от построенной модели. На третьем шаге подбираем оптимальные значения параметров настройки посредством использования контрпримеров, построенных в ходе верификации свойства оптимальности. На четвёртом шаге извлекаем информацию о параметрах настройки из контрпримера для оптимальных параметров. Мы применяем этот подход к автонастройке параллельных программ, написанных на языке OpenCL – современном популярном языке, который расширяет язык C для программирования как обычных многоядерных процессоров (CPU), так и массивно-параллельных графических процессоров (GPU). В качестве инструмента верификации мы используем верификатор SPIN и его язык представления моделей Promela, формальная семантика которого позволяет моделировать исполнение параллельных программ на процессорах с различной архитектурой.

Ключевые слова: задача оптимизации; автонастройка параллельных программ; параллельные программы; программирование GPU; проверка моделей; контрпримеры; OpenCL; SPIN; Promela

ИНФОРМАЦИЯ ОБ АВТОРАХ

Наталья Олеговна Гаранина
автор для корреспонденции
orcid.org/0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
канд. физ.-мат. наук, с.н.с.

Сергей Петрович Горлач
orcid.org/0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de
профессор, канд. физ.-мат. наук.

Финансирование: Представленное исследование выполнено в рамках гранта DAAD № 91735805, государственного задания № АААА-А-19-119120290056-0, и проекта DFG #PPP-DL в Университете Мюнстера, Германия.

Для цитирования: N. O. Garanina and S. P. Gorlatch, “Autotuning Parallel Programs by Model Checking”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 338-355, 2021.

Введение

Наша работа посвящена развитию методов автонастройки программ для современных архитектур параллельных процессоров. *Автонастройка* автоматически находит оптимальные значения так называемых *параметров настройки*, являющихся критичными для производительности исполнения программ на многоядерных процессорах (CPU) и многоядерных графических процессорах (GPU): оптимальные значения параметров настройки позволяют достичь максимальной производительности и/или минимального энергопотребления данной параллельной программы на конкретной архитектуре. Типичными примерами параметров настройки являются: количество параллельных потоков, количество потоков в одной рабочей группе, размер данных в локальной памяти процессора и т. п. Система автонастройки должна автоматически генерировать пространство поиска параметров и исследовать его, чтобы найти оптимальную (или достаточно хорошую) комбинацию значений параметров настройки. Подбор оптимальных параметров в системе автонастройки основан на выборе конфигурации параметров из пространства поиска и запуске программы с этими параметрами на реальном процессоре. Стадия перебора параметров и запуска является самой затратной по времени – как правило, поиск достаточно качественного решения занимает несколько часов, а иногда и сутки машинного времени.

В связи с высокой актуальностью проблемы автонастройки, к настоящему времени предложен и реализован ряд конкретных систем автонастройки: ATLAS, PATUS, FFTW, MILEPOST, CHiLL, OSKI, ActiveHarmony, Apollo, OpenTuner, CLTune, ATF [1—11]. Эти системы используют различные методы поиска оптимального решения, включая моделирование отжига, случайный поиск, статистические методы экспериментального проектирования, генетические алгоритмы, поиск с использованием машинного обучения и динамическое программирование. Но лишь некоторые из них предлагают исчерпывающий поиск, который находит гарантированно оптимальный результат. При этом, этот поиск использует прямолинейный алгоритм перебора всех возможных вариантов и поэтому, как правило, используется редко из-за высокой ресурсоёмкости и временных затрат.

В этой работе мы предлагаем новую технику автонастройки, основанную на исчерпывающем поиске с использованием инструментов проверки модели (model checking) и моделировании абстрактных процессоров. Обычно инструменты проверки модели исследуют каждый элемент пространства поиска, используя различные сокращения явного перебора этого пространства, такие как символьное кодирование, редукция частичных порядков и абстракция [12]. Эти сокращения позволяют исследовать бóльшие пространства поиска за более короткое время. Поэтому мы предполагаем, что инструменты проверки моделей будут находить гарантированно оптимальные значения параметров настройки за меньшее время, чем традиционные программы автонастройки. Кроме того, моделирование абстрактных процессоров позволяет не зависеть от наличия конкретных процессоров при поиске оптимальных параметров параллельной программы.

Проверка модели является формальным методом верификации программ. Этот метод проверяет выполнимость свойства параллельной системы для всех сценариев её поведения. Если проверяемое свойство не выполняется в некотором сценарии, то этот сценарий является *контрпримером*: последовательностью действий системы и условий, влекущих невыполнимость свойства. Контрпримеры являются основой подхода к решению задач оптимизации и, в частности, составления оптимальных расписаний для реагирующих систем [13—17]. В этой статье мы адаптируем технику контрпримеров для поиска оптимальных конфигураций параметров настройки.

Мы будем решать задачу автонастройки программ, написанных на популярном языке параллельных вычислений OpenCL [18], предложенного как новый стандарт для программирования как CPU, так и GPU. Основные компоненты нашего абстрактного процессора соответствуют схеме графического процессора, т.к. автонастройка наиболее часто используется именно для GPU-программ, поскольку архитектура GPU особенно сложна для предсказания конечной производительности

программ на этой архитектуре. В качестве инструмента проверки моделей мы будем использовать верификатор SPIN [19] – надежную и удобную программу проверки моделей. Его язык моделирования Promela имеет C-подобный синтаксис и темпоральную логику LTL для формулирования свойств программы. Семантика языка объединяет модели параллелизма CSP [20] и акторную модель [21], что идеально подходит для моделирования параллельного исполнения программ на различных типах графических процессоров. Также немаловажно, что в описании моделей на языке Promela допускаются вставки кода на языке C.

Оставшаяся часть работы организована следующим образом. Раздел 1 посвящён описанию техники контрпримеров в задачах планирования/оптимизации и нашей адаптации этой техники для задачи поиска оптимальных параметров автонастройки. В разделе 2 рассмотрены основные понятия языка OpenCL и нашей абстрактной модели графического процессора, необходимые для решения задачи автонастройки в контексте проверки моделей, а также идеи их моделирования на языке Promela. Раздел 3 содержит описание предлагаемой нами техники поиска оптимальных параметров автонастройки с использованием инструмента SPIN. В заключительном разделе 4 изложены выводы и представлен план будущих исследований.

1. Проверка моделей в задачах планирования, оптимизации и автонастройки

Проверка моделей – это формальный метод верификации модели системы относительно заданной спецификации её свойств. Этот метод проверяет свойства модели системы для всех сценариев её поведения. Модель системы и свойство задаются с помощью формальных языков. Например, модель системы можно задать как структуру Крипке, а свойство – формулой темпоральной логики LTL. Если свойство не выполняется в модели, то метод позволяет найти контрпример, т.е. условия и последовательность действий модели, влекущих невыполнимость свойства. Контрпримеры являются основой техники получения оптимальных расписаний для параллельных систем [13—17]. Кроме того, известный Дагштуль-семинар №14482 [22] был посвящён сочетанию методов автоматического планирования и проверки моделей.

Кратко изложим нашу идею использования проверки моделей в задаче планирования (оптимизации), которую мы предлагаем применять для автонастройки. *Задача планирования* состоит в поиске *плана*, т.е. поиска последовательности действий в некоторой среде, которая приводит к заданной цели. При этом *среду* мы представляем как дискретное множество состояний и переходов между ними. Мы считаем, что состояния среды дискретны, каждое состояние полностью наблюдаемо и множество состояний конечно. Определено *начальное состояние* и определена *цель* как множество заключительных состояний. Известны возможные действия по изменению состояний, и при этом каждое действие приводит к единственному новому состоянию, а время выполнения действий не учитывается. *Решением* задачи планирования будет являться путь от начального состояния до целевого состояния, который, вообще говоря, может удовлетворять некоторому набору ограничений. Если оценивать качество решения, то есть решать *задачу оптимизации*, то необходимо определить *функцию стоимости пути*. В зависимости от критерия оптимальности, это может быть наибольшая или наименьшая стоимость среди всех возможных решений задачи планирования. Таким образом, для алгоритма планирования необходимы методы представления среды и переходов между состояниями, а также метод управления перебором в пространстве состояний.

Очевидно, что метод проверки моделей полностью соответствует вышеизложенной постановке задач планирования и оптимизации. В этой постановке средой является пространство состояний модели системы, в которых выделяется множество начальных состояний. Состояния цели можно задать с помощью булевой формулы относительно переменных системы. Тогда, чтобы найти решение задачи планирования с помощью метода проверки моделей, нужно задать свойство *невозможности достижения цели*. В случае, если цель всё-таки достижима, то метод проверки моделей построит контрпример, демонстрирующий возможность достижения цели. Поскольку такой контр-

пример является последовательностью действий модели, приводящей к состоянию, в котором цель достижима, то он и будет планом, ведущим к цели. Если целевые состояния заданы булевой формулой φ, тогда свойство невозможности достижения цели на языке логики LTL выглядит как $\mathbf{G}\neg\varphi$. Если имеются ограничения на пути к цели φ, которые можно выразить булевой формулой ψ, то LTL-формула для свойства невозможности достижения цели при заданных ограничениях выглядит как $\neg(\psi\mathbf{U}\varphi)$.

В задаче оптимизации предполагается, что цель достижима и есть численная оценка пути достижения цели. Поэтому, для решения этой задачи с помощью проверки моделей, в модель системы, представляющую среду, нужно ввести оценочную переменную, значения которой в состоянии модели зависят от пути, ведущего в это состояние. Тогда для получения оптимальных значений этой переменной, нужно задать свойство *невозможности достижения цели при заданной оценке*. Пусть целевые состояния заданы булевой формулой φ и, например, (минимальная) оценка задана неравенством $x < N$, где x – оценочная переменная, а N – число. Тогда это свойство выражается следующей формулой логики LTL: $\mathbf{G}(\varphi \longrightarrow x \geq N)$. Если это свойство не выполняется в модели, это означает, что цель достижима при значениях оценки x, меньших заданного порога N. Значит, если уменьшить порог, то можно получить лучший план, который приводит к цели. Таким образом, для получения оптимального значения оценочной функции необходимо последовательно выполнять проверку моделей с изменяющимся значением порога до тех пор, пока метод не перестанет конструировать контрпримеры. Порог, для которого минимальное его изменение ведёт к отсутствию контрпримера, будет оптимальным значением численной оценки, а контрпример, который порождается для этого порога, является оптимальным планом. Для автоматического составления оптимальных расписаний оценочной переменной может быть переменная, хранящая глобальное время работы системы. В этом случае, как правило, необходимо найти минимальное время работы системы. Вышеописанную технику можно обобщить для численных оценок нескольких параметров.

В этой статье мы адаптируем технику контрпримеров проверки модели для задачи поиска оптимальных параметров автонастройки исполнения параллельных программ на языке OpenCL на абстрактном графическом процессоре следующим образом.

В задаче параллельных вычислений целью является вычисление результата для заданных входных данных. Эта цель может быть достигнута различными путями в зависимости от распределения параллельной обработки данных по вычислительным элементам и использования локальной памяти процессора. Задача оптимизации состоит в поиске последовательности действий, которая за кратчайшее время позволяет получить результат вычисления. Поэтому свойство Φ_o невозможности достижения цели при заданной оптимальной оценке формулируется как: “Параллельная программа *не может* завершиться через T единиц времени”. Свойство Φ_o нужно верифицировать в модели, которая представляет исполнение параллельной программы вычислений в заданном процессоре, с помощью инструмента проверки модели. Если это свойство не выполняется, это означает, что вычисления действительно завершаются в течение времени T. Инструмент проверки модели конструирует контрпример, который описывает условия завершения работы программы за время T, включая конфигурацию параметров настройки, которые отвечают за распределение параллельной обработки данных по вычислительным элементам и интенсивность использования локальной памяти. Далее нужно уменьшить время завершения и проверять это свойство снова и снова, пока не найдётся *минимальное* время выполнения программы MT: если мы уменьшим это время на одну единицу времени, программа проверки модели докажет, свойство Φ_o выполняется, т.е. программа действительно не может завершиться за время T. Начальное значение времени завершения может быть задано с помощью симуляции модели программы. Как правило, инструменты проверки моделей позволяют симулировать сценарии выполнения модели и можно выявить значение вре-

мени исполнения параллельной программы в каком-либо из сценариев. Уменьшать полученное значение можно, например, методом бисекции.

Итак, предлагаемый нами *метод контрпримеров для поиска оптимальной конфигурации параметров настройки* состоит из следующих шагов.

1. Представление на языке инструмента проверки моделей параметров настройки и исполнения программы на выбранном процессоре.
2. Формулирование свойства Φ_o невозможности достижения цели на языке программы проверки моделей.
3. Поиск минимального времени завершения программы MT.
4. Извлечение информации об оптимальной конфигурации параметров из контрпримера для минимального времени MT.

В следующих разделах мы решаем задачу поиска параметров настройки для программ OpenCL, исполняющихся на различных графических процессорах. Для решения этой задачи методом контрпримеров мы выбрали в качестве средства проверки моделей инструмент SPIN [19], поскольку язык Promela, как и OpenCL, тоже близок к языку C, и при этом он допускает встраивание кода C в проверяемые модели. Кроме того, формальная семантика Promela, основанная на исчислении взаимодействующих процессов CSP [20] и алгебре акторов [21], допускает формализацию абстрактного графического процессора. Вышеперечисленные 4 шага поиска параметров настройки OpenCL с помощью SPIN мы подробно опишем в разделе 3.

В следующем разделе мы описываем наш подход к абстрагированию понятий языка OpenCL [18] и графического процессора, необходимые для построения Promela-модели исполнения абстрактной программы OpenCL на абстрактном процессоре.

2. Абстрактный графический процессор и язык OpenCL

Язык *OpenCL* используется для реализации параллельных вычислений на т.н. *OpenCL-устройствах*, которыми могут быть как многоядерные процессоры (CPU), так и графические процессоры (GPU). В нашей работе мы рассматриваем только графические процессоры (далее – *процессоры*), поскольку OpenCL разрабатывался в первую очередь для таких устройств, и как правило, реализация программ OpenCL на них более эффективна. Однако наш подход достаточно общий и может быть адаптирован для CPU.

2.1. Абстрактный графический процессор

В этой работе мы рассматриваем элементы архитектуры нашего абстрактного графического процессора `Abstract Processor`, ориентируясь на архитектуру Ферми корпорации Nvidia, которая является ведущим производителем GPU-процессоров [23]. Другие графические процессоры устроены похожим образом.

Процессор связан с CPU, откуда загружаются данные в *глобальную память* процессора GPU. Наш `Abstract Processor` включает m *мультипроцессоров SM* (Streaming Multiprocessor). Каждый мультипроцессор содержит ряд процессорных элементов (обычно их число составляет 2^n). В частности, мультипроцессор с архитектурой Fermi включает 32 универсальных *вычислителя* (cores), 16 элементов для работы с данными *LSU* (load/store units) и 4 элемента для работы со специальными функциями *SFU* (Special Function Units). В дальнейшем мы предполагаем, что абстрактный мультипроцессор содержит 2^n процессорных элемента, которые, для простоты моделирования, являются только универсальными вычислителями.

Процессорные элементы работают одновременно. Их вычислениями управляют встроенные *диспетчеры*. Они запускают вычислители группами – так называемыми *варпами* (warp). Будем считать,

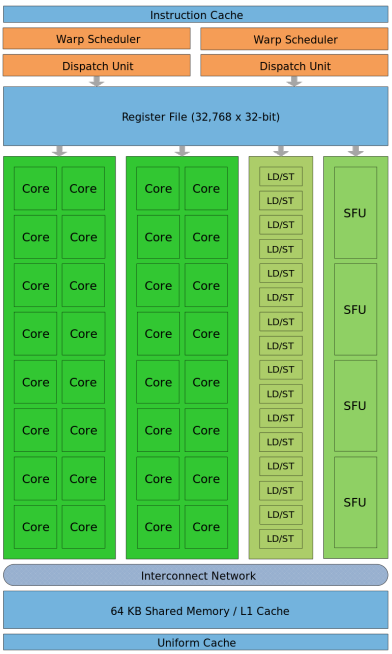

Fig. 1. Fermi architecture of multiprocessor

Рис. 1. Мультипроцессор архитектуры Ферми

что в абстрактный мультипроцессор встроены 2^k диспетчеров и размер варпа составляет 2^{n-k}[1]. Диспетчеры для работы с варпами используют список инструкций обработки данных. Процессорные элементы выбранного диспетчером варпа одновременно выполняют свою очередную инструкцию. Когда все вычислители варпа завершают свою работу, диспетчер выбирает следующий варп.

Мультипроцессор имеет быструю *локальную память*, к которой есть доступ у всех процессорных элементов. Туда могут загружаться данные из глобальной памяти процессора. Отношение скоростей доступа к локальной памяти и глобальной составляет примерно 1/100.

На рисунке 1 представлен мультипроцессор архитектуры Ферми. Здесь оранжевым цветом обозначены компоненты, отвечающие за диспетчеризацию, оттенками зелёного цвета – процессорные элементы, а синим – модули локальной памяти.

Итак, наш `Abstract Processor` имеет следующие компоненты: процессорные элементы, диспетчеры и мультипроцессоры, а также CPU-процесс, который распределяет вычисления по мультипроцессорам. Мы будем представлять их с помощью отдельных Promela-процессов. Все эти процессы иерархически связаны отношениями подчинения: CPU-процесс запускает мультипроцессоры, мультипроцессоры активируют своих диспетчеров, а диспетчеры, в свою очередь, регулируют работу своего варпа, т.е. набора процессорных элементов. Для синхронизации своей работы они используют каналы сообщений ёмкостью 1, по которым пересылаются команды начала и окончания работы, а также сообщения о завершении вычислений. Различие в скоростях доступа к локальной и глобальной памяти моделируется с помощью констант, задающих объём времени обращения к памяти. Для моделирования глобального времени в системе мы используем специальный процесс, использующий число активных процессорных элементов для определения момента, когда счётчик времени увеличивается. В этой работе мы не учитываем время коммуникации между компонентами процессора, которое на практике существенно меньше, чем другие временные задержки. Мы изложим детали нашей реализации на Promela абстрактного процессора в разделе 3.

[1]Мультипроцессор с архитектурой Fermi включает два диспетчера и в одном его варпе может быть 16 вычислителей или LSU, или 4 SFU.

2.2. Ключевые понятия программы на языке OpenCL

Программа на OpenCL состоит из двух логических частей: *хост-программа* и *ядро*. Хост-программа исполняется на CPU, соединенном с графическим процессором (GPU), а ядро – параллельно всеми процессорными элементами этого GPU. Хост-программа обеспечивает исполнение ядра: компилирует ядро, работает с данными для него: резервирует глобальную память процессора, копирует в неё данные, и выгружает данные результата из этой памяти в память CPU. Ядро производит обработку данных и вычисления.

Типичная программа на OpenCL обрабатывает массивы и выдаёт результат обработки. Поэтому, как правило, множество ядер параллельно вычисляют выражения на основе доступных данных, зависящих от индекса массива, и присваивают результат вычисления отдельной переменной или элементу массива. Отметим, что массив может быть многомерным, и тогда его элементы будут иметь соответствующий многомерный индекс. Каждое вычисление (ядро) выполняет один *рабочий элемент*. Рабочий элемент имеет идентификатор, соответствующий индексу массива. Таким образом, рабочие элементы выполняют одни и те же вычисления, но для разных данных, т.е. элементов массива. Этот метод распараллеливания известен в литературе как параллелизм по данным (data parallelism). Если вычисления одной итерации цикла зависят от вычислений другой итерации, то есть от индекса массива, то в OpenCL предусмотрены средства синхронизации вычислений: *барьеры* (barrier). Множество рабочих элементов разбивается на *рабочие группы* либо явно программистом, либо по умолчанию конкретной реализацией OpenCL. Элементы одной группы должны исполняться на одной вычислительной единице (например, мультипроцессоре).

В OpenCL память разделяется на 4 вида: *глобальная, постоянная, локальная и приватная*. К глобальной памяти имеют доступ все рабочие элементы и хост-программа. Постоянная память – это неизменяемая часть глобальной памяти. К локальной памяти имеют доступ элементы одной рабочей группы. Приватной памятью может пользоваться один рабочий элемент. Поскольку предполагается, что доступ к локальной памяти на устройстве, для которого компилируется программа, существенно быстрее доступа к глобальной памяти, то ядро часто задаёт, какие данные и в каком объёме будут загружаться в локальную память. Эти данные называются *плитками* (tiles).

Язык C служит основой языка OpenCL. Основные ограничения OpenCL по сравнению с C:

- отсутствуют многие стандартные функции, например, `printf` или `malloc`;
- нет массивов переменной длины;
- рекурсия не допускается;
- нет указателей на функции.

Схема абстрактной программы на OpenCL выглядит следующим образом:

```
int main ( void ) {
 1. Initialize OpenCL
 2. Compile kernel source code
 3. Reserve global memory on the device and copy data to the device
 4. Execute kernel (instances)
 5. Copy data from the device
}
```

Листинг 1: Абстрактная программа на OpenCL

В дальнейшем мы будем рассматривать следующую схему типичного ядра программы на OpenCL:

```
__kernel void abstract_kernel (__global float* N_g, __global float* R_g,
    int size){
1. __local float N_l[ TS ];
2. int idx_g = get_global_id (0);
3. int idx_l = get_local_id (0);
4. float result = 0;
5. for (int i = 0; i < size / TS; ++i) {
```

```
    // access to global memory
6.      N_l[idx_l] = f(i, idx_l, size, tile, N_g);
    // waiting for local co-workers
7.      barrier (CLK_LOCAL_MEM_FENCE);
8.      if b(idx_l) // access to local memory
9.           for (int k=0; k < TS; ++k) result = g1(k, idx_l, N_l, result);
10.     else for (int k=0; k < TS; ++k) result = g2(k, idx_l, N_l, result);
    // waiting for local co-workers
11.     barrier (CLK_LOCAL_MEM_FENCE);}
 // copy the result of this working item to global memory
12. R_g[h(idx_g, size)] = result;
```

Листинг 2: Схема типичного ядра OpenCL

Параллельные по данным вычисления – это всегда вычисления для массивов (возможно, многомерных). Поэтому для каждого индекса массива отдельный рабочий элемент вычисляет локальный результат исполнения кода ядра. В нашем примере программы в листинге 2 входными данными является массив `N_g` размера `size`. Выходными данными (результатом) здесь являются элементы массива `R_g`. Для уменьшения обращений к глобальной памяти в строке 1 объявляется локальный массив `N_l` размера `TS`, элементы которого зависят от входных данных. Каждый из элементов рабочей группы, который получил значение своего индекса `idx_l` в строке 2, с помощью функции `f` параллельно вычисляет (или копирует) своё значение локального массива `N_l[idx_l]` в строке 6, но пользоваться этим значением могут все элементы его группы, поэтому в строке 7 необходимо дождаться завершения работы с глобальными данными всех элементов данной рабочей группы с помощью оператора синхронизации `barrier`. Далее в строках 9-10 итеративные вычисления результата `result` зависят только от локальных данных и индекса элемента группы. Кроме того, в зависимости от индекса, эти вычисления могут проводиться по-разному: в строке 8 булева функция `b(idx_l)` регулирует варианты вычисления (функция `g1` или `g2`). В строке 11 рабочий элемент дожидается завершения вычислений всех одногруппников с текущими локальными данными, и на следующей итерации цикла строки 5, локальные данные снова вырабатываются на основе очередной порции глобальных данных. Когда глобальные данные будут обработаны полностью, результат работы элемента группы сохраняется в глобальной памяти в элементе массива `R_g`, индекс которого зависит от размера входных данных и глобального индекса `idx_g`, полученного элементом группы `idx_l` в строке 2. Список *Возможных деталей* программ на OpenCL, которые не показаны в примере схемы ядра `abstract_kernel` включает следующие пункты:

1) массивы могут быть многомерными;
2) может быть несколько входных массивов, и кроме массивов могут быть другие входные переменные и константы, размещённые в глобальной памяти;
3) выходными данными может быть несколько массивов или чисел;
4) нет изменений в глобальной памяти, которые требуют синхронизации всех процессов всех рабочих групп.

Однако пример листинга 2 легко дополнить этими деталями, не выходя за рамки его функций и операторов.

Наш подход к использованию языка Promela использует тот факт, что поскольку язык Promela, как и язык OpenCL, близок к языку C, то программы на OpenCL можно транслировать в язык Promela с учётом следующих ограничений. Так как SPIN предполагает абстрагирование от вычислительных аспектов программ и предназначен для проверки взаимодействия, синхронизации и координации параллельных процессов, все данные Promela-модели должны иметь конечный тип. Более того, при задании на Promela абстрактного ядра, мы абстрагируемся от конкретных вычислений, и для решения нашей задачи поиска оптимальных параметров мы учитываем лишь время этих вычислений, которое зависит от количества и соотношения обращений к глобальной и локальной памяти.

Поэтому строки вычислений 6, 9, 10 и 12 листинга 2 в Promela-модели абстрактного ядра заменяются на код, реализующий течение времени, необходимого для этих вычислений. По этой же причине мы игнорируем параметры и локальные переменные ядра, задающие содержание данных для вычислений (строки 1-4). Однако количество этих вычислений, т. е. `size` должно быть учтено. В силу абстрагирования от вычислений в Promela-модели ядра, пункты 1-3 списка Возможных деталей также не учитываются при моделировании, а значит, пример ядра `abstract_kernel` является достаточно представительным с точностью до количества циклов и структур управления программы, так как для моделирования вычислений имеет значение только размер входных данных.

Кроме того, важным аспектом, который мы учитываем при моделировании, является синхронизация элементов рабочих групп относительно изменений в локальной и глобальной памяти. Для обеспечения локальной синхронизации в модель вводятся процессы барьера, отвечающие за отдельную рабочую группу, а для глобальной синхронизации – процесс барьера для всех рабочих элементов. В нашем примере ядра глобальный барьер отсутствует, но его реализация на Promela аналогична реализации локального барьера.

Роль хост-программы в нашем моделировании на Promela сводится к процессу “компиляции” ядра, т. е. распределению вычислений в заданном графическом процессоре. Детали реализации на Promela исполнения абстрактного ядра `abstract_kernel` и его хост-программы на абстрактном процессоре `Abstract Processor` изложены в следующем разделе, где описывается применение метода проверки моделей для решения задачи оптимизации производительности программ OpenCL.

3. Поиск оптимальных параметров программ OpenCL с помощью SPIN

Для реализации первого шага поиска оптимальной конфигурации параметров настройки (раздел 1), т. е. моделирования на языке инструмента проверки моделей исполнения программы на выбранном процессоре, будем опираться на следующую общепринятую семантику исполнения программ OpenCL [18].

Согласно этой семантике, компилятор выделяет для исполнения хост-программы один хост-процессор (CPU), соединенный с графическим процессором (GPU). Последний объединяет m мультипроцессоров SM, а элементы одной рабочей группы исполняются на одном мультипроцессоре. Каждый рабочий элемент группы последовательно исполняется на одном вычислителе мультипроцессора. Следовательно, одновременно могут исполняться 2^n рабочих элемента, по 2^{n-k} на каждый варп, где 2^k – число диспетчеров. Если размер рабочей группы больше 2^n, то множество рабочих элементов разбивается на несколько варпов и два диспетчера мультипроцессора выбирают поочерёдно по одному варпу для исполнения очередной инструкции рабочих элементов группы. За один такт выполняется ровно одна инструкция элемента. Поскольку в коде ядра могут встречаться условные операторы, зависящие от номера рабочего элемента, то в этом случае варп может выполниться только частично, а инструкции оставшихся элементов варпа выполняются в следующих тактах, пока все элементы варпа не исполнят свою очередную инструкцию. После этого диспетчер выбирает следующий варп. Рабочие элементы могут использовать данные как из локальной памяти мультипроцессора, так и из глобальной памяти. При этом данные локальной памяти не доступны элементам других рабочих групп, исполняемых на других мультипроцессорах. Поскольку доступ к локальной памяти значительно быстрее, чем к глобальной, разумно ожидать, что разбиение на группы максимизирует использование локальной памяти, то есть в одной группе оказываются элементы, которые в основном используют данные, вырабатываемые внутри группы.

Таким образом, мы выделяем следующие параметры, влияющие на производительность параллельных вычислений для программы на OpenCL. Эти параметры зависят от размера входных данных `size`, обрабатываемых программой. Количество рабочих элементов зависит от размера данных.

- *Размер рабочей группы* WG (определяется в хост-программе). При оптимально выбранном размере рабочей группы все вычислители всех мультипроцессоров GPU загружены полностью и равномерно, что приводит к сокращению общего времени вычислений.
- *Размер плиток* TS (определяется в ядре). При оптимальном выборе размера данных, периодически загружаемых в быструю локальную память, достигается минимизация числа обращений к медленной глобальной памяти, что также приводит к сокращению времени вычислений.

Итак, мы решаем задачу поиска оптимальных размеров рабочих групп WG и размеров плиток TS для абстрактного ядра abstract_kernel (листинг 2) и его хост-программы, исполняемых на абстрактном процессоре Abstract Processor (раздел 2.1).

Шаг 1 метода контрпримеров

Этот шаг является самым трудоёмким этапом нашего метода, так как необходимо учесть все детали исполнения параллельных программ OpenCL на абстрактном графическом процессоре. На этом шаге мы определяем в Promela-модели *PM* этого исполнения следующие Promela-процессы.

Процесс рабочего элемента группы pex выполняет вычисления экземпляра ядра abstract_kernel. Общее число этих процессов, создающихся в Promela-модели, зависит от размера входных данных. Число одновременно работающих процессов pex зависит от количества мультипроцессоров и величины варпов, что значительно меньше общего числа процессов. Каждый из них запускается своим диспетчером, Promela-процесс которого описан ниже. Процесс pex связан каналами синхронизации pex_b и b_pex с локальным барьером группы, и pex_d и d_pex – со своим диспетчером. В эти каналы рабочий элемент получает команды запуска и останова, а также сообщает о завершении (этапа) вычислений. Мы абстрагируемся от конкретных вычислений, которые проводит ядро, поэтому в модели используем только время выполнения вычислений. При этом время вычислений, использующих локальную память, мы считаем равным одной условной единице времени, а время вычислений, использующих глобальную память, равным GMT условным единицам времени. Таким образом, в процессе pex смоделировано только количество шагов вычислений, совершаемых ядром (цикл for в строке 5), зависящее от размера входных данных и обращений к глобальной памяти (строки 6-11, 23-28) и к локальной памяти (строки 14-19). По завершении шага вычислений pex сообщает об этом событии процессу, реализующему глобальное время посредством увеличения счётчика работающих в данный момент процессов NRP_work. Отметим, что в силу блокирующей семантики языка Promela, процесс может перейти к следующему этапу своих вычислений только, когда глобальное время time увеличится на 1 (строки 10, 16, 19 и 27). Синхронизация по локальному барьеру происходит дважды, как и в исходном ядре: строка 7 ядра соответствует строке 13 модели, а строка 11 – строке 21. Процесс pex завершает работу (строка 29), когда выгружает результат своей работы в глобальную память.

```
proctype pex ( byte me; chan pex_b; chan b_pex; chan d_pex; chan pex_d) {
1.  do
2.  :: d_pex ? go, me ->
3.     start_time = time;
4.     cur_time = time;
5.     for ( i : 0 .. size/TS){
6.          do  // access to global memory
7.          :: time > start_time + GMT -> break;
8.          :: else -> atomic { cur_time = time;
9.                              NRP_work++;}
10.                    time == cur_time + 1;
11.         od;
12.         b_pex ! done;
          // waiting for local co-workers
13.         pex_b ? go, me;
          // 'if' access to local memory
14.         atomic { cur_time = time;
```

```
15.                   NRP_work++;}
16.          time == cur_time + 1;
         // 'else' access to local memory
17.          atomic { cur_time = time;
18.                   NRP_work++;}
19.          time == cur_time + 1;
20.          b_pex ! done;
         // waiting for local co-workers
21.          pex_b ? go, me;
       }
22.    start_time = time;
   // copy the result of this working item to global memory
23.    do
24.    :: time > start_time + GMT -> break;
25.    :: else -> atomic { cur_time = time;
26.                        NRP_work++;}
27.                 time == cur_time + 1;
28.    od;
29.    pex_d ! done;
30. :: d_pex ? stop, me -> break;
31. od;
}
```

Листинг 3: Promela-процесс для рабочего элемента группы pex

Процесс локальной синхронизации `barriere` синхронизирует процессорные элементы одного мультипроцессора, с которыми он связан каналами `pex_b` и `b_pex`. Он принимает от процессов `pex` сообщение о приостановке их работы и подсчитывает в переменной `i` число процессов, которые ожидают завершения работы с локальной памятью других элементов группы, выполняющих в данный момент вычисления. Когда оно окажется равным числу процессорных элементов `PE`, барьер считается пройденным и в строке 13 процесс барьера позволяет процессорным элементам продолжить вычисления.

```
proctype barriere (chan pex_b; chan b_pex) {
1. do
2. :: pex_b ? done ->
3.             i = 1;
4.             do
5.             :: i < PE -> atomic {
6.                             pex_b ? done;
7.                             NRP--;
8.                             i++;}
9.             :: else -> break;
10.            od;
11.            atomic {
12.              NRP = NRP + PE;
13.              for (i : 1..PE) { b_pex ! go, i;} }
14. :: pex_b ? stop -> break;
15. od;
}
```

Листинг 4: Promela-процесс для локальной синхронизации `barrier`

Процесс диспетчера `dispatcher` запускает в строке 5 один варп, т.е. `WR` процессов `pex`, исполняющих ровно одну операцию одновременно. При этом он обновляет количество `NRP` запущенных в данный момент процессоров (строка 4). На это число влияет число процессов диспетчеров `ND` в одном мультипроцессоре и число `NM` мультипроцессоров, которые зависят от выбранного процессора. В зависимости от размера рабочей группы `WG`, может быть несколько варпов, приписанных одному диспетчеру. В таком случае, диспетчер активирует их поочерёдно, пока все элементы рабочей группы не выполнят все свои вычисления (строки 6-16). Мы считаем, что диспетчер запускает варпы целое число раз (`WG % (WR*ND) = 0` в строке 6). Процесс `dispatcher` завершает работу, когда

все приписанные к нему процессы pex завершаются и сообщает об этом своему мультипроцессору (строка 19).

```
proctype dispatcher (byte me; chan pex_b; chan b_pex; chan mul_d; chan
    d_mul) {
 chan d_pex = [1] of {mtype : action, byte};
 chan pex_d = [1] of {mtype : action};
1. do
2. :: mul_d ? go, me ->
3.              atomic {
4.                NRP = NRP + WR;
5.                for (i : 1..WR){ run pex (i, pex_b, b_pex, d_pex, pex_d);}}
6.              for (j : 1..WG/(WR*ND)) {
7.                   atomic {
8.                     NRP = NRP + WR;
9.                     for (i : 1..WR) { d_pex ! go, i;} }
10.                  i = 0;
11.                  do
12.                  :: i < WR -> atomic {
13.                                  pex_d ? done;
14.                                  NRP--;
15.                                  i++; }
16.                  :: else -> break;
17.                  od; }
18.           for (i : 1..WR) { d_pex ! stop, i;}
19.           d_mul ! done;
20. :: mul_d ? stop, me -> break;
21. od;
}
```

Листинг 5: Promela-процесс для диспетчера

Процесс мультипроцессора `muproc` запускает свои процессы `dispatcher` числом `ND` в строках 4-5 и фиксирует завершение их работы в строках 8-13, после чего сообщает процессу `host` о завершении вычислений в своей рабочей группе. Отметим, что мультипроцессор запускает единый для своей рабочей группы процесс локальной синхронизации `barriere` с каналами, в который есть доступ у всех процессорных элементов этой группы, поскольку они передаются как параметр.

```
proctype muproc (byte me; chan m_hst; chan hst_m) {
 chan pex_b = [1] of {mtype : action};
 chan b_pex = [1] of {mtype : action, byte};
 chan d_mul = [1] of {mtype : action};
 chan mul_d = [1] of {mtype : action, byte};
1.  do
2.  :: hst_m ? go, me ->
3.             run barriere (px_bar, bar_px);
4.             atomic { for (i : 1..ND) {
5.                        run dispatcher (i, pex_b, b_pex, mul_d, d_mul); }}
6.             for (i : 1..ND) { mul_d ! go, i;}
7.             i = 0;
8.             do
9.             :: i < WR -> atomic {
10.                           d_mul ? done;
11.                           i++;}
12.            :: else -> break;
13.            od;
14.            for (i : 1..ND) { mul_d ! stop, i;}
15.            px_bar ! stop;
16.            m_hst ! done, me;
17. :: hst_m ? stop, me -> break;
18. od;
}
```

Листинг 6: Promela-процесс для мультипроцессора

Процесс хост-программы `host` запускает мультипроцессоры `muproc`. Если число рабочих групп `WGs` оказалось больше числа мультипроцессоров в выбранном процессоре, то вычисления, разбитые на рабочие группы, запускаются последовательно (строки 13-31). Здесь мы не требуем, чтобы число рабочих групп делилось нацело на число мультипроцессоров `NM`. Процесс `host` фиксирует завершение работы мультипроцессоров присваиванием глобальной переменной `DONE` значение `true`. Это означает, что параллельные вычисления завершились.

```
proctype host () {
 chan m_hst = [1] of {mtype : action, byte};
 chan hst_m = [1] of {mtype : action, byte};
1.  DONE = false;
2.  if
3.  :: WGs <= NM ->
4.                atomic { for (i : 1..WGs) {run muproc (i, m_hst, hst_m);}}
5.                for (i : 1..WGs) {hst_m ! go, i};
6.                i = 0;
7.                do
8.                :: i < WGs -> atomic {
9.                                m_hst ? done;
10.                               i++; }
11.               :: else -> break;
12.               od;
13. :: else -> atomic { for (i : 1..NM) {run muproc (i, m_hst, hst_m);}}
14.             for (i : 1..NM) {hst_m ! go, i};
15.             i = 0;
16.             do
17.             :: i < WGs -> atomic {
18.                             m_hst ? done, j;
19.                             hst_m ! stop, j;
20.                             run muproc (j, m_hst, hst_m);
21.                             hst_m ! go, j;
22.                             i++; }
23.             :: else -> break;
24.             od;
25.             i = 0;
26.             do
27.             :: i < NM -> atomic {
28.                            m_hst ? done, j;
29.                            i++;}
30.             :: else -> break;
31.             od;
33. fi;
33. DONE = true;
}
```

Листинг 7: Promela-процесс для хост-программы

Основной процесс `main` выбирает значения параметров настройки `WG` и `TS` и запускает процессы `host` и `clock`. Количество мультипроцессоров `NM`, количество диспетчеров `ND` для одного мультипроцессора, размер варпа `WR` и количество вычислителей `PE` являются глобальными константами и объявляются в начале описания модели. Это даёт возможность настройки Promela-модели для различных архитектур реальных графических процессоров. В этой модели для простоты мы считаем, что размер данных `size` является степенью числа 2. Поэтому числа `WG` и `TS` также будут какой-либо случайной степенью 2. Эти степени выбираются в строках 3 и 6. Поскольку Promela не поддерживает операцию возведения в степень, выбранные случайным образом числа можно получить посредством соответствующего побитового сдвига `size`.

```
active proctype main() {
byte d;
 // let size = 2^n
1.  byte n = 10;
```

```
2.   size = 1024;
 // WG selection
3.   select ( d  : 1 .. n-1 );
4.   WG = size >> (n - d);
 // Number of Working Groups
5.   WGs = size/WG;
 // tile size selection
6.   select ( d  : 1 .. n/2 );
7.   TS = size >> (n - d);
8.   Topt = 100;
9.   atomic { run host(); run clock(); }
}
```

Листинг 8: Promela-процесс для выбора параметров настройки и запуска вычислений

Процесс часов `clock` реализует подсчёт глобального времени. Этот процесс увеличивает глобальную переменную счётчика времени `time`, когда все запущенные в настоящий момент процессы `pex` (их число `NRP`) сообщили посредством увеличения разделяемой переменной `NRP_work`, что они находятся в состоянии вычисления очередного значения. Процесс `clock` останавливается, когда глобальная переменная `DONE` принимает значение `true`. Значение `time` в этот момент является временем, затраченным на вычисления.

```
proctype clock () {
1. do
2. :: DONE -> break;
3. :: NRP != 0 && NRP_work == NRP -> atomic { NRP_work = 0; time++; }
4. od;
}
```

Листинг 9: Promela-процесс для подсчёта глобального времени

Таким образом, мы задали модель исполнения программы OpenCL на абстрактном графическом процессоре, т.е. выполнили первый шаг метода контрпримеров для поиска оптимальных параметров настройки (раздел 1).

Шаг 2 метода контрпримеров

На втором шаге метода при формулировании свойства невозможности достижения цели Φ_o^a мы будем использовать значение переменной `DONE`, фиксирующей окончание вычислений, и финальное значение `time`. Языком спецификации свойств в SPIN является темпоральная логика LTL и поэтому $\Phi_o^a = \mathbf{G}(\texttt{DONE} \longrightarrow (\texttt{time} \geq MT))$, что соответствует высказыванию "Всегда при завершении параллельных вычислений время работы программы больше минимального времени MT."

Шаг 3 метода контрпримеров

Третий шаг метода, который заключается в поиске минимального времени, за которое программа параллельных вычислений может завершиться, начинается с запуска верификатора SPIN с построенной моделью PM и формулой Φ_o^a для некоторого значения минимального времени MT. Затем мы уменьшаем MT до тех пор, пока SPIN не перестанет генерировать контрпримеры, т.е. пока не согласится с тем, что программу за время, меньшее заданного MT, выполнить нельзя. Начальное значение MT можно задать, используя возможность симуляции моделей в SPIN. При симуляции SPIN воспроизводит один из конечных сценариев работы системы, фиксируя значения используемых в модели переменных по завершении симуляции. Поэтому можно воспользоваться значением `time`, соответствующим концу работы программы в симулируемом сценарии. Для уменьшения значения MT в следующих запусках SPIN мы используем метод бисекции.

Шаг 4 метода контрпримеров

Последний шаг нашего подхода – анализ контрпримера для извлечения оптимальной конфигурации параметров настройки. Для анализа контрпримера SPIN предоставляет возможность

запуска симуляции, соответствующей переходам контрпримера. В задаче автонастройки не нужно осуществлять поиск оптимального пути вычислений, поэтому собственно переходы контрпримера анализировать нет необходимости. Для решения нашей задачи нужно извлечь лишь значения параметров настройки `WG` и `TS`, которые, как и значения других переменных, известны в конце симуляции.

Таким образом, мы показали, как можно использовать инструмент проверки моделей SPIN для решения задачи поиска оптимальных параметров программ на языке OpenCL, исполняемых на абстрактном графическом процессоре. Абстрактный графический процессор можно специализировать путём задания конкретных значений числа мультипроцессоров, диспетчеров и процессорных элементов, а также, возможно, изменением алгоритма взаимодействия диспетчера с процессорными элементами.

4. Заключение

Результаты нашей работы направлены на развитие методов автонастройки за счет разработки новой техники, исчерпывающего поиска оптимальных параметров настройки параллельных программ на основе проверки моделей. В представленной статье мы предложили подход к поиску оптимальных параметров исполнения программ на языке OpenCL на абстрактном графическом процессоре, который обобщает архитектуры GPU, используемые на практике. Наш подход использует метод контрпримеров, основанный на проверке моделей. Метод предполагает представление исполнения программ и свойства оптимальности на языке инструмента проверки моделей. В качестве такого инструмента мы выбрали верификатор SPIN с языком Promela для моделирования исполнения программ и темпоральной логикой LTL для представления свойства оптимальности.

Наше моделирование исполнения программ OpenCL в Promela абстрагируется от конкретных вычислений, сохраняя логику взаимодействия и синхронизации параллельных процессов программы в выбранном графическом процессоре. Отметим, что поскольку язык Promela имеет формальную семантику, то Promela-модель заданной OpenCL-программы можно считать формальной операционной семантикой взаимодействия и синхронизации параллельных процессов программы в выбранном графическом процессоре. Более того, варьирование параметров и алгоритмов взаимодействия компонентов абстрактного процессора позволяет задавать и исследовать конкретные графические процессоры. Это даёт возможность поиска оптимальных параметров настройки программ при физическом отсутствии реальных графических процессоров, в то время как традиционные системы автонастройки такой возможности предоставить не могут.

В будущем мы планируем добавить в модель время коммуникации между процессами, а также рассмотреть другие параметры настройки, в частности, количество рабочих элементов. Кроме того, мы адаптируем метод поиска оптимальных значений из работы [14], который позволяет избежать многократного запуска верификатора SPIN в методе контрпримеров. Существенным ограничением изложенной реализации нашего метода является малый размер Promela-модели, который допускает одновременную активность только 255 параллельных процессов, в то время как в реальных графических процессорах их могут быть тысячи. Поэтому мы планируем разработать подход к масштабированию результатов нашего метода контрпримеров, в котором может использоваться то, что размер реальных данных и количество процессорных элементов реальных графических процессоров является степенью числа 2. Развитием этой темы будет применение нашего подхода к реальным OpenCL-программам и конкретным графическим процессорам.

References

[1] J. Ansel, S. Kamil, K. Veeramachaneni, and et al., “OpenTuner: An extensible framework for program autotuning”, in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[2] J. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, “Apollo: Reusable models for fast, dynamic tuning of input-dependent code”, in *Proc. of 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2017.

[3] C. Chen, J. Chame, and M. Hall, “CHILL: A framework for composing high-level loop transformations”, *Technical Report 08-897. Los Angeles, CA*, pp. 136–150, 2008.

[4] M. Christen, O. Schenk, and H. Burkhart, “PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures”, in *Proc. of 2011 IEEE International Parallel Distributed Processing Symposium*, IEEE, 2011.

[5] R. Whaley and J. Dongarra, “Automatically tuned linear algebra software”, in *Proc. of the ACM/IEEE Conference on Supercomputing*, IEEE, 1998.

[6] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3”, *IEEE*, vol. 93(2), 2005.

[7] G. Fursin, Y. Kashnikov, A. Memon, and et al., “Milepost GCC: machine learning enabled self-tuning compiler”, *Int J Parallel Prog*, vol. 39(3), pp. 296–327, 2011.

[8] C. Nugteren and V. Codreanu, “CLTUne: A generic auto-tuner for OpenCL kernels”, in *Proc. of 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, IEEE, 2015.

[9] A. Rasch and S. Gorlatch, “ATF: A Generic, Directive-Based Auto-Tuning Framework”, *Concurrency and Computation: Practice and Experience*, vol. 31(5), 2018.

[10] C. Tapus, I. Chung, and J. Hollingsworth, “Active harmony: towards automated performance tuning”, in *Proc. of 2002 ACM/IEEE Conference on Supercomputing*, IEEE, 2002.

[11] R. Vuduc, J. W. Demmel, and K. A. Yelick, “OSKI: A library of automatically tuned sparse matrix kernels”, in *Proc. of Scientific Discovery through Advanced Computing Conference*, IEEE, 2005.

[12] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Springer International Publishing, 2018, ch. 1. Introduction to Model Checking, pp. 1–13.

[13] T. C. Ruys and E. Brinksma, “Experience with Literate Programming in the Modelling and Validation of Systems”, in *Proc. of the 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’98)*, Springer, 1998, pp. 393–408.

[14] T. Ruys, “Optimal Scheduling Using Branch and Bound with SPIN 4.0”, in *Proc. of Model Checking Software. SPIN 2003*, Springer, 2003.

[15] E. Brinksma, A. Mader, and A. Fehnker, “Verification and optimization of a PLC control schedule”, *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 21–33, 2002.

[16] A. Wijs, J. V. D. Pol, and E. M. Bortnik, “Solving scheduling problems by untimed model checking: The clinical chemical analyser case study.”, in *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, 2005, pp. 54–61.

[17] R. Malik and P. Pena, “Optimal Task Scheduling in a Flexible Manufacturing System using Model Checking”, *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 230–235, 2018.

[18] *The OpenCL Specification*. Khronos OpenCL working group, 2021.

[19] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[20] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, 1985.

[21] M. Gaspari and G. Zavattaro, “An Algebra of Actors”, in *Proc. of Formal Methods for Open Object-Based Distributed Systems. FMOODS 1999*, Springer, 1999.

[22] A. Cimatti, S. Edelkamp, M. Fox, D. Magazzeni, and E. Plaku, "Automated Planning and Model Checking (Dagstuhl Seminar 14482)", *Dagstuhl Reports*, vol. 4, no. 11, pp. 227–245, 2015, ISSN: 2192-5283. DOI: 10.4230/DagRep.4.11.227. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/4973.

[23] P. N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture*. NVIDIA Corporation, 2009.