05.13.11 ***МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ МАШИН, КОМПЛЕКСОВ КОМПЬЮТЕРНЫХ СЕТЕЙ***

MATHEMATICAL AND SOFTWARE OF COMPUTERS AND COMPUTER NETWORKS

DOI: 10.33693/2313-223X-2021-8-1-38-45

Метод прогнозирования времени выполнения программ для графических процессоров

А.А. Клейменов[a] ©, Н.Н. Попова[b] ©

Московский государственный университет имени М.В. Ломоносова, г. Москва, Российская Федерация

[a] E-mail: andreykleimenov@mail.ru

[b] E-mail: popova@cs.msu.ru

Аннотация. Ведущей тенденцией развития архитектур высокопроизводительных вычислительных систем в последние годы является использование сопроцессоров – графических ускорителей (GPU) и ПЛИС (FPGA) – программируемых логических интегральных схем. В связи с этим растет число приложений из самых разных предметных областей, модифицированных для использования на GPU и успешно использованных на практике. В работе предлагается подход к прогнозированию времени выполнения CUDA-ядер, основанный на статическом анализе исходного кода программы. Подход основан на построении модели CUDA-ядра и модели графического ускорителя. Разработанный метод оценки времени выполнения CUDA-ядер применен к реализации алгоритмов матричного умножения, преобразованию Фурье и методу обратного распространения ошибки для обучения нейросетей. В результате верификации подход показал хорошую точность предсказания, особенно при небольшой загрузке графического процессора.

Ключевые слова: анализ производительности, CUDA-ядро, статический анализ, модель графического процессора

Благодарности. Работа выполнена при финансовой поддержке РФФИ (проект № 20-07-01053)

ССЫЛКА НА СТАТЬЮ: Клейменов А.А., Попова Н.Н. Метод прогнозирования времени выполнения программ для графических процессоров // Computational nanotechnology. 2021. Т. 8. № 1. С. 38–45. DOI: 10.33693/2313-223X-2021-8-1-38-45

DOI: 10.33693/2313-223X-2021-8-1-38-45

A method for prediction execution time of GPU programs

A.A. Kleimenov[a] ©, N.N. Popova[b] ©

Lomonosov Moscow State University, Moscow, Russian Federation

[a] E-mail: andreykleimenov@mail.ru

[b] E-mail: popova@cs.msu.ru

Abstract. The use of coprocessors such as GPU and FPGA is a leading trend in HPC. Therefore a lot of applications from a wide variety of domains were modified for GPUs and successfully used. In this paper, we propose an approach for prediction execution time of CUDA kernels, based on a static analysis of a program source code. The approach is based on building a CUDA

core model and a graphics accelerator model. The developed method for estimating the execution time of CUDA kernels is applied to the implementation of matrix multiplication, the Fourier transform and the backpropagation method for training neural networks. As a result of verification, the approach showed good prediction accuracy, especially on low GPU loads.

Key words: performance analysis, CUDA-kernel, static analysis, GPU model

Acknowledgments. This work was financially supported by the Russian Foundation for Basic Research (Grant No. 20-07-01053)

FOR CITATION: Kleimenov A.A., Popova N.N. A method for prediction execution time of GPU programs. *Computational Nanotechnology*. 2021. Vol. 8. No. 1. Pp. 38–45. (In Russ.) DOI: 10.33693/2313-223X-2021-8-1-38-45

ВВЕДЕНИЕ

Статья посвящена проблеме оценки производительности параллельных программ. В статье предлагается метод прогнозирования времени выполнения программ на графических процессорах.

Ведущей тенденцией развития архитектур высокопроизводительных вычислительных систем в последние годы является использование сопроцессоров – графических ускорителей (GPU) и ПЛИС (FPGA) – программируемых логических интегральных схем. Примером этого может быть значительное увеличение числа суперкомпьютеров в списке Top500, использующих графические ускорители. В связи с этим растет число приложений из самых разных предметных областей, модифицированных для использования на GPU и успешно использованных на практике. Прогнозирование времени выполнения программ на графических ускорителях направлено на разработку более эффективных программ, адаптирование существующих программ под новые графические ускорители, оптимизацию выбора графических ускорителей для решения конкретных задач. Прогноз времени выполнения программ на графических ускорителях может использоваться для более эффективного планирования потока задач в условиях их выполнения на многопроцессорных системах.

В статье предлагается метод оценки выполнения программ, предназначенных для графических ускорителей NVIDIA, и реализованных с использованием технологии CUDA. Согласно принятой терминологии функции, реализованные с использованием этой технологии, называются CUDA-ядрами.

Проблема оценки времени выполнения CUDA-ядер нетривиальна по причине сложности архитектуры современных графических ускорителей, сложности исследуемых программ и часто нежелания производителей делиться аспектами реализации, которые могут оказывать существенное влияние на время выполнения программы.

Предлагаемый в статье подход к прогнозированию времени выполнения CUDA-ядер основан на статическом анализе текста программы, реализующей ядро, и является продолжением ранее предложенного подхода для оценки времени выполнения параллельных программ [Клейменов, Попова, 2021]. Подход основан на построении двух моделей: модели CUDA-ядра и модели графического ускорителя. Модель ядра в значительной степени формируется исходя из автоматизированного статического анализа исходного кода. Модель графического ускорителя строится на основе данных, полученных с помощью микробенчмарков, и информации, предоставляемой в официальной документации устройства.

В первом параграфе статьи рассматриваются основные аспекты построения архитектур графических процессоров и дается краткий обзор программной модели CUDA. Во втором параграфе приводится обзор существующих подходов к предсказанию времени работы CUDA-ядер. В третьем параграфе описываются предлагаемые модели CUDA-ядра и графического ускорителя. С использованием предложенных моделей дается описание метода прогнозирования времени работы CUDA-ядер. В четвертом параграфе обсуждаются результаты верификации предложенного подхода.

1. МОДЕЛЬ GPU-ПРОГРАММ И АРХИТЕКТУРА GPU

В контексте программно-аппаратной архитектуры CUDA код, выполняющийся на GPU, называется ядром. С точки зрения программной реализации ядро – это функция, содержащая последовательный код, который может обращаться к ряду специальных переменных. Примером таких переменных являются размер сетки, идентификатор блока, размер блока и идентификатор нити в блоке. Вызов функции, реализующей CUDA-ядро, может проводиться в программе, выполняющейся на CPU, или из другого CUDA-ядра. CPU в данном контексте называют хостом, а GPU устройством. Таким образом, можно рассматривать два типа ядер: ядра, которые быть вызваны только из GPU и ядра, которые могут быть вызваны как на хосте, так и на устройстве.

В основе программной модели GPU лежит SIMT (Single Instruction Multiple Threads) подход к параллелизму, предполагающий выполнение одной и той же последовательности инструкций большим количеством нитей. Нити организованы в двухуровневую иерархию. Первый уровень иерархии – это сетка, состоящая из блоков. На втором уровне иерархии находится блок, который, в свою очередь, состоит из множества нитей. Иногда такие блоки также называют массивом скооперированных нитей (CTA – Cooperative Thread Array). У каждого блока в рамках сетки и у каждой нити в рамках блока есть свой идентификатор, состоящий из одного, двух или трех чисел (такие идентификаторы удобно рассматривать как координаты в пространстве). Размер сетки и блоков указывается при запуске ядра.

Переменные, к которым происходит обращение при исполнении ядра, находятся в памяти GPU, которая разделяется на несколько типов: локальная память, разделяемая память, глобальная память и константная память. Пространство локальной памяти индивидуально для каждой нити и не доступно другим нитям. Пространство разделяемой памяти общее для всех нитей в блоке, но не доступно из других блоков. Глобальная память доступна для всех нитей ядра. Константная память также доступна всем нитям ядра, но она не может изменяться.

Для синхронизации нитей в рамках CUDA в основном используются примитивы двух типов: барьерная синхронизация и атомарные операции. Барьерная синхронизация используется для синхронизации нитей в рамках блока. Нити блокируются при вызове барьера до тех пора, пока все нити блока не достигнут барьера и обращения к глобальной и разделяемой памяти завершатся. Атомарные операции гарантируют эксклюзивный доступ нити к участку памяти.

С точки зрения архитектуры можно выделить несколько основных отличий GPU от CPU: степень параллелизма GPU во много раз больше (в 100–1000 раз), чем у CPU, тактовая частота GPU обычно ниже тактовой частоты CPU (в 2–3 раза), пропускная способность памяти у GPU намного больше (приблизительно на порядок), а латентность обращения к памяти больше.

В дальнейшем обсуждении главным образом будут рассматриваться архитектуры графических ускорителей Nvidia и, соответственно, будет использоваться характерная для этих архитектур терминология.

GPU состоит из одного или нескольких вычислительных кластеров (GPC – Graphic Processing Cluster). Все вычислительные кластеры имеют доступ к общему L2 кэшу и памяти. Каждый GPC состоит из одного или нескольких кластеров текстурной обработки (TPC – Texture Processor Cluster), а также может включать дополнительные устройства, например, блок растеризации (Raster Engine). Кластеры текстурной обработки в свою очередь состоят из одного или нескольких потоковых мультипроцессоров (SM – Streaming Multiprocessor) и блоков наложения текстур. Схема GPU представлена на рис. 1.

Потоковый мультипроцессор включает в свой состав один или несколько блоков обработки (Processing Block), L1 кэш, блок разделяемой памяти, кэш инструкций и ряд других устройств. Каждый блок обработки содержит планировщик варпов (Warp Scheduler), устройство доставки (Dispatch Unit), ядра для операций с числами с плавающей точкой, целыми числами и т.д. Схема потокового мультипроцессора представлена на рис. 2.

При запуске ядра блоки нитей распределяются по потоковым мультипроцессорам, и остаются там до завершения выполнения. Количество блоков, которые могут выполняться одновременно на одном потоковом мультипроцессоре, ограничено и зависит от множества факторов, таких как размер блоков, количество регистров, используемых нитями, количество используемой разделяемой памяти. Если количество блоков в сетке больше, чем количество блоков, которые могут суммарно обработать все потоковые мультипроцессоры, то выполнение ядра разбивается на несколько этапов (волн). Нити блока разбиваются на варпы (варп обычно состоит из 32 нитей) и распределяются по блокам обработки. В процессе выполнения варпы не перемещаются в другие блоки обработки.

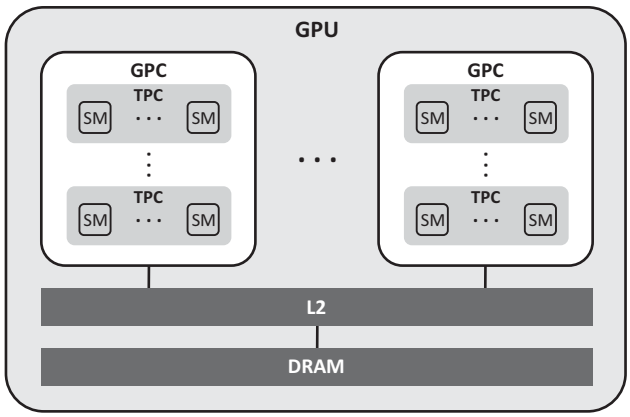


Рис. 1. Схема GPU

Fig. 1. GPU scheme

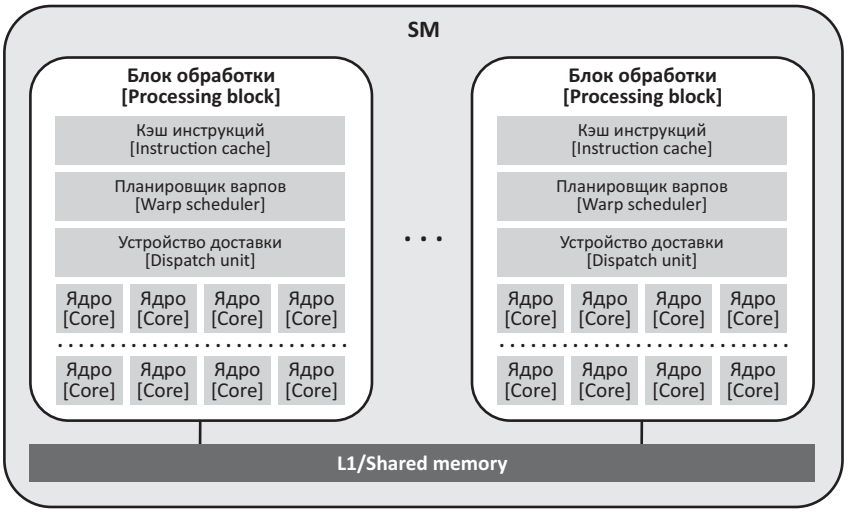


Рис. 2. Схема потокового мультипроцессора

Fig. 2. Scheme of a streaming multiprocessor

2. ОБЗОР ПОДХОДОВ К ПРОГНОЗИРОВАНИЮ ВРЕМЕНИ РАБОТЫ CUDA-ЯДЕР

Существующие подходы к прогнозированию времени работы CUDA-ядер можно разделить на три класса:

- аналитические, с использованием исходного кода ядра [Hong, Kim, 2009; Baghsorkhi и др., 2010; Zhang, Owens, 2011; Sim и др., 2012; Konstantinidis, Cotronis, 2017; Alavani, Varma, Sarkar, 2018];
- аналитические, с использованием методов машинного обучения на данных, собранных в результате запуска программы [Jia, Shaw, Martonosi, 2012; Wu и др., 2015], в дальнейшем будем называть их эмпирически детерминированными подходами;
- симуляционные [Bakhoda и др., 2009; Collange и др., 2010; Malhotra, Goel, Sarangi, 2014].

В аналитических подходах прогнозирование выполняется на основе вычисления аналитического выражения.

Аналитические подходы на основе анализа исходного кода обычно трудоемки в использовании, поскольку требуют хорошего понимания программы исследователем. Эмпирически детерминированные подходы, наоборот, не требуют глубоко изучения исследуемой программы и легко автоматизируются, но при этом они требуют многочисленных запусков программы для сбора данных для дальнейшего обучения. В симуляционных подходах прогнозирование осуществляется посредством симулирования программы или ее редуцированного представления. Преимуществами симуляционного подхода является возможность рассмотрения целевой программы как черного ящика и отсутствие необходимости наличия целевого устройства. Основным недостатком данной группы подходов является их времязатратность. Более подробный анализ подходов к предсказанию времени выполнения параллельных программ дается в статье [Клейменов, Попова, 2021].

Один из наиболее известных аналитических подходов к оценке времени выполнения CUDA-ядер представлен в статье [Hong, Kim, 2009]. В его основе лежит определение двух характеристик: MWP (Memory Warp Parallelism) и CWP (Computation Warp Parallelism). MWP отражает максимальное количество одновременно обрабатываемых запросов к памяти и зависит от характеристик целевого устройства. CWP отражает арифметическую интенсивность приложения. Если программа активно обращается к памяти, то значение CWP будет большим и программа будет ограничена пропускной способностью памяти. Если же программа редко обращается в память, то значение CWP будет небольшим и время работы программы будет определяться вычислительной производительностью устройства. К сожалению, данный подход не учитывает латентность арифметических инструкций, поэтому он не может дать хороших оценок для приложений с высокой арифметической интенсивностью.

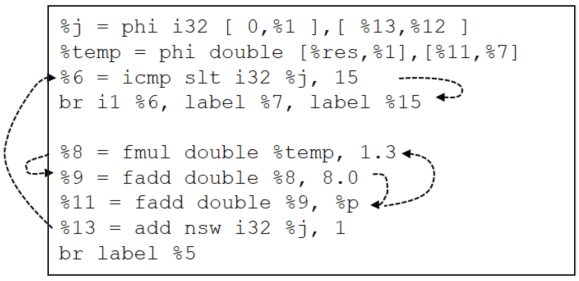
Подход, представленный в статье [Zhang, Owens, 2011], основан на Roofline модели и в отличие от рассмотренной выше работы имеет лучшую точность на приложениях с высокой арифметической интенсивностью, поскольку учитывает латентность инструкций. Однако, при средних значениях арифметической интенсивности данный подход дает значительную ошибку, поскольку Roofline модель не учитывает тот факт, что операции обращения к памяти и арифметические операции могут не пересекаться во времени.

В статье [Alavani, Varma, Sarkar, 2018] описывается подход к прогнозированию времени работы программы на основе анализа ее PTX кода. На основе анализа кода вычисляется доля времени, затрачиваемого на выполнение одного варпа. Время работы программы определяется посредством умножения числа активных варпов на вычисленную долю времени. Подход имеет плохую точность на приложениях с невысокой арифметической интенсивностью.

Эмпирически детерминированный подход, представленный в статье [Jia, Shaw, Martonosi, 2012], основан на пошаговой регрессии. Подход предполагает выполнение следующих действий. Сначала случайным образом сэмплируется параметрическое пространство графического ускорителя. Затем приложение выполняется или симулируется при выбранных значениях параметров. После этого осуществляется пошаговая регрессия для нахождения наиболее значимых параметров, влияющих на производительность приложения. Затем с использованием сплайнов производится интерполяция результатов, полученных на этапе сэмплирования. В результате вычисляется время работы программы при заданных параметрах графического ускорителя.

3. ПРЕДЛАГАЕМЫЙ ПОДХОД К ОЦЕНКЕ ВРЕМЕНИ РАБОТЫ CUDA-ЯДЕР

Основная идея предлагаемого подхода к оценке времени выполнения CUDA-ядер состоит в построении аналитических выражений на основе автоматизированного анализа исходного кода, отражающих характеристики нитей ядра (такие, как время работы нити, такты задержки, генерируемые нитью, объем памяти, используемый нитью) и их использовании для расчета времени выполнения всего ядра. Для определения количества тактов задержки производится оценка количества исполнений инструкций, которая умножается на количество тактов задержки, генерируемых данными инструкциями при исполнении на целевом графическом процессоре. Для построения аналитического выражения времени работы нити дополнительно учитываются зависимости операций по данным и их латентности на целевом графическом процессоре. На рис. 3 приведен фрагмент кода ядра, представляющий цикл, записанный на языке промежуточного представления кода LLVM и соответствующее ему аналитическое выражение времени его выполнения нитью. Стрелочками на рисунке обозначены зависимости по данным. Переменные, начинающиеся с *lat*, представляют латентности инструкций, а переменные, начинающиеся с *del*, представляют такты задержки инструкций. Переменная *trip* – это количество итераций цикла.



$$t_{loop} = (lat_{\%6} + del_{br} + lat_{\%8} + lat_{\%9} + max(max(del_{\%13} + del_{br}, lat_{\%13}) + lat_{\%6} + br_{del}, lat_{\%11}) + lat_{\%8} + lat_{\%9} + del_{\%11} + del_{\%13})/2*trip$$

Рис. 3. Фрагмент кода CUDA-ядра на языке LLVM и аналитическое выражение времени его выполнения нитью

Fig. 3. Code fragment of the CUDA-kernel in LLVM IR and the analytical expression for its execution time by a thread

Реализацию предлагаемого подхода можно описать следующим образом.

1. Построение модели графического ускорителя исходя из данных, полученных в результате запуска микробенчмарков на целевом графическом ускорителе и/или полученных из официальной документации и публикаций.
2. Построение модели программы ядра на основе статического анализа ее кода и предоставленной пользователем информации о программе.
3. Прогнозирование времени выполнения ядра, основанное на итеративном пересчете средней латентности обращений в память с учетом ограниченности пропускной способности памяти графического ускорителя.

Модель графического ускорителя задается кортежем:

$m_{gpu} = <smCount, lat_{inst}, delay_{inst}, lat_{mem}, band_{mem}, limits>$

где *smCount* – количество потоковых мультипроцессоров; lat_{inst} – латентности PTX инструкций; $delay_{inst}$ – количество тактов между стартом выполнения инструкции и стартом выполнения последующей инструкции (в дальнейшем будем называть эту характеристику тактами задержки инструкций); lat_{mem} – латентности доступа к памяти: L1 кэшу, L2 кэшу, DRAM, а также латентность плохо структурированных запросов в память; $band_{mem}$ – пропускные способности различных уровней памяти; *limits* – лимиты, ограничивающие количество блоков нитей на потоковом мультипроцессоре: размер разделяемой памяти на потоковом мультипроцессоре, количество регистров на потоковом мультипроцессоре.

Латентности инструкций могут быть получены с помощью микробенчмарков или найдены в публикациях, например, в [Alavani, Varma, Sarkar, 2018; Jia и др., 2018; Arafa и др., 2019]. Для определения количества тактов задержки инструкций можно использовать два способа. Первый способ предполагает вычисление количества тактов задержки $dela\ y_{inst} = warpSize/throughput_{inst}$, где $throughput_{inst}$ – пропускная способность, выраженная в количестве операций, выполняемых за такт. Пропускную способность многих инструкций также можно определить используя микробенчмарки или официальную документацию [CUDA C++ Programming Guide, 2021] и другие публикации [Alavani, Varma, Sarkar, 2018]. Второй способ определения тактов задержки инструкций можно определить как

$$dela\ y_{inst} = \max\left(\frac{warpSize}{unitCount_{inst}},\ 1\right),$$

где $unitCount_{inst}$ – количество устройств в блоке обработки данного типа инструкций. Данный подход основывается на наблюдении того факта, что часто узким местом является работа устройства доставки, которое вынуждено тратить дополнительные такты на доставку инструкции варпа (вместо доставки следующей), если не смогло разом доставить инструкции для всего варпа из-за недостатка исполняющих устройств. Так, если у блока обработки есть всего лишь одно исполняющее устройство для операций с числами двойной точности, то устройству доставки понадобится 32 такта, чтобы доставить инструкции всего варпа прежде, чем он сможет начать доставлять следующую инструкцию.

Количество устройств, занимающихся обработкой инструкций, латентности доступов и пропускные способности различных уровней памяти могут быть определены с помощью микробенчмарков [Mei, Chu, 2017] или найдены в официальной документации устройств.

Модель ядра можно описать кортежем:

$$m_{kernel} = \langle time, delay, mem, uncoalMem, l1pct, l2pct, syncs, registers, sharedMem \rangle$$

где *time* – функция, возвращающая время выполнения нити в тактах, как если бы только она одна выполнялась на ускорителе; *delay* – функция, возвращающая суммарное число тактов задержки, генерируемое нитью; *mem* – функция, возвращающая объем памяти, используемой нитью в процессе выполнения; *uncoalMem* – функция, возвращающая объем памяти, используемой нитью в рамках неструктурированных запросов. Под неструктурированными запросами понимаются такие запросы варпа, при которых контроллер памяти вынужден запросить больше сегментов памяти, чем при последовательном доступе к памяти нитями варпа; *l*1*pct* – функция, возвращающая долю обращений нити в память, попадающих в L1 кэш; *l*2*pct* – функция, возвращающая долю обращений нити в память, попадающих в L2 кэш; *syncs* – функция, возвращающая количество вызовов операций барьерной синхронизации; *registers* – количество регистров, используемых нитями; *sharedMem* – количество разделяемой памяти, используемой блоками.

Функции *time* и *delay* определяются ранее описанной моделью GPU. Они могут зависеть от идентификатора нити и входных параметров ядра. Функция *time* также зависит от латентности доступа к памяти. Функции *mem*, *uncoalMem*, *syncs* могут зависеть от идентификатора нити, а также от входных параметров ядра.

Автоматизацию получения функций модели ядра предлагается проводить с использованием фреймворка LLVM [Lattner, Adve, 2004]. С этой целью код ядра на языке LLVM IR анализируется с помощью средств, предоставляемых LLVM. При этом инструкции этого представления отображаются на аналогичные PTX-инструкции. Функции *l*1*pct* и *l*2*pct* задаются пользователем.

Время работы ядра задается формулой:

$$time_{kernel} = time_{launch} + time_{exec},$$

где $time_{launch}$ – время запуска ядра; $time_{exec}$ – время выполнения инструкций ядра.

Определение времени запуска ядра проводится с использованием линейной регрессии. Сначала формируется набор данных о времени выполнения запуском пустого, не содержащего кода ядра с различными размерами сетки и блоков. Наблюдения показали, что при одном и том же размере блока время запуска линейно зависит от размера сетки. Поэтому для каждого размера блока, выраженного в количестве варпов в блоке, проводится линейная регрессия, в результате чего создаются [*maxBlockSize*/*warpSize*] линейных моделей вида $gridSize*x_1 + x_2$, где *maxBlockSize* – максимальный размер блока в нитях (обычно 1024 нитей или 32 варпа).

Для получения оценки времени выполнения ядра суммируются времена выполнения волн:

$$t_{exec} = \sum_{w \in waves} t_{wave}(w).$$

Для расчета времени выполнения волны сначала вычисляется время работы потоковых мультипроцессоров с учетом пропускной способности L1 кэша. Затем рассчитывается время выполнения волны на GPU с учетом пропускной способности L2 кэша, DRAM памяти и пропускной способности для неструктурированных запросов.

Введем несколько функций, необходимых для вычисления времени выполнения волны, в которых для краткости будет опущена зависимость от входных параметров ядра и модели графического процессора:

- объем данных, загружаемых нитями варпа из L1 кэша

$$mem\_l1_{WP}(wp) = \sum_{th \in threads(wp)} mem(th) * l1pct(th),$$

  где *threads* – функция, возвращающая идентификаторы нитей, принадлежащих варпу *wp*. Аналогичным образом мы вводим функции $mem\_l2_{WP}(wp)$, $mem\_dram_{WP}(wp)$, $mem\_uncoal_{WP}(wp)$;
- время, затрачиваемое на барьерную синхронизацию, рассчитываемое в соответствии со способом, предложенным в статье [Hong, Kim, 2009],

$$time\_syncs(wp) = syncs(getMainThread(wp)) * delay_{dep} * (blockSize - 1).$$

Функция *getMainThread* возвращает идентификатор самой продолжительной нити варпа, $delay_{dep}$ = ($gpum.lat_{uncoal}$ – $gpum.lat_{DRAM}$)/28;

- время выполнения варпа

$$time_{WP}(wp, lat) = \\ = time(getMainThread(wp), lat) + time\_sync(wp);$$

- время задержки, генерируемое варпом,

$$delay_{WP}(wp) = delay(getMainThread(wp));$$

- время выполнения блока обработки

$$time_{PB}(pb, lat) = \\ = \max\left(\max_{wp \in warps(pb)} time_{WP}(wp, lat), \sum_{wp \in warps(pb)} delay_{WP}(wp)\right);$$

- объем памяти, считываемой потоковым мультипроцессором,

$$mem_{SM}(sm) = \sum_{wp \in warps(sm)} mem_{WP}(wp).$$

Аналогичным образом определяются $mem\_l1_{SM}(sm)$, $mem\_l2_{SM}(sm)$, $mem\_dram_{SM}(sm)$, $mem\_uncoal_{SM}(sm)$;

- объем памяти, используемой приложением,

$$mem_{GPU}(gpu) = \sum_{sm \in sms(gpu)} mem_{SM}(sm).$$

Аналогичным образом определяются $mem\_l2_{GPU}(gpu)$, $mem\_dram_{GPU}(gpu)$ и $mem\_uncoal_{GPU}(gpu)$;

- доля памяти, загружаемой из L1 кэша нитями, находящимися на мультипроцессоре,

$$l1pct_{SM}(sm) = \frac{meml1_{SM}(sm)}{mem_{SM}(sm)}.$$

Аналогичным образом определяются $l2pct_{SM}$, $drampct_{SM}$;

- доля памяти, используемой в рамках структурированных запросов,

$$coalpct_{SM}(sm) = \frac{mem_{SM}(sm) - uncoalMem_{SM}(sm)}{mem_{SM}(sm)};$$

- доля памяти, используемой в рамках неструктурированных запросов,

$$uncoalpct_{SM}(sm) = \frac{uncoalMem_{SM}(sm)}{mem_{SM}(sm)};$$

- время работы потокового мультипроцессора

$$time_{SM}(sm, lat) = \max_{pb \in pbs(sm)} time_{PB}(pb, lat).$$

Среднее время обращения нитей в память на потоковом мультипроцессоре определяется следующим образом:

$$lat(sm, lat_{L1}, lat_{L2}, lat_{DRAM}, lat_{uncoal}) = \\ = coalpct_{SM}(sm)\left(l1pct_{SM}(sm)lat_{L1} + \right. \\ + l2pct_{SM}(sm)lat_{L2} + drampct_{SM}(sm)lat_{DRAM}\left.\right) + \\ + uncoalpct_{SM}(sm)lat_{uncoal}.$$

Расчет времени работы потокового мультипроцессора с учетом пропускной способности L1 кэша проводится путем итерационного пересчета латентности доступа в L1 кэш до тех пор, пока требуемая приложением пропускная способность L1 кэша превышает пропускную способность L1 кэша. В результате определяются времена работы потоковых мультипроцессоров и латентности доступов нитей в L1 кэш на этих мультипроцессорах. Алгоритм определения времени выполнения волны потоковым мультипроцессором с учетом пропускной способности L1 кэша представлен на рис. 4.

$old\_time = 0$
$time = \varepsilon + 1$
$lat_{L1} = gpum.lat_{L1}$
$lat = lat_{SM}(sm, lat_{L1}, gpum.lat_{L2}, gpum.lat_{DRAM}, gpum.lat_{uncoal})$
while $|old\_time - time| > \varepsilon$ **do**
  $old\_time = time$
  $time = time_{SM}(sm, lat)$
  $l1mul = (mem\_l1_{SM}(sm)/time)/gpum.band_{L1}$
  $lat_{L1} = lat_{L1} * l1mul$
  $lat = lat_{SM}(sm, lat_{L1}, gpum.lat_{L2}, gpum.lat_{DRAM}, gpum.lat_{uncoal})$
end while
return $time, lat_{L1}$

Рис. 4. Алгоритм определения времени выполнения волны потоковым мультипроцессором с учетом пропускной способности L1 кэша: *gpum* – структура содержащая модель графического ускорителя, ε – точность определения времени

Fig. 4. Algorithm for calculating wave execution time by a SM with respect to L1 cache bandwidth: *gpum* – structure containing model of GPU, ε – accuracy of timing

После расчета времени работы потоковых мультипроцессоров определяется время выполнения волны на всем графическом ускорителе с учетом пропускных способностей L2 кэша, DRAM и пропускной способности для неструктурированных запросов.

Определим время работы волны, используя данные латентностей доступа к памяти:

$$time_{GPU}(gpu, lat_{L2}, lat_{DRAM}, lat_{uncoal}) = \\ = \max_{sm \in sms(gpu)} time_{SM} \times \\ \times\left(sm, lat_{SM}(sm.lat_{L1}, lat_{L2}, lat_{DRAM}, lat_{uncoal})\right),$$

где $sm.lat_{L1}$ – полученная на предыдущем шаге латентность доступа в L1 кэш, специфическая для потокового мультипроцессора.

Алгоритм расчета времени выполнения волны графическим процессором с учетом пропускной способности L2 кэша представлен на рис. 5.

$lat_{L2} = gpum.lat_{L2}$
$old\_time = 0$
$time = \varepsilon + 1$
while $|old\_time - time| > \varepsilon$ **do**
  $old\_time = time$
  $time = time_{GPU}(gpu, lat_{L2}, gpum.lat_{DRAM}, gpum.lat_{uncoal})$
  $l2mul = (mem\_l2_{GPU}/time)/gpum.band_{L2}$
  $lat_{L2} = lat_{L2} * l2mul$
end while
return $time, lat_{L2}$

Рис. 5. Алгоритм расчета времени выполнения волны графическим процессором с учетом пропускной способности L2 кэша

Fig. 5. Algorithm for calculating wave execution time by a GPU with respect to L2 cache bandwidth

В результате определяется обновленное время выполнения волны с учетом пропускной способности L2 кэша и оценки латентности доступа к нему.

Аналогично с учетом пропускной способности DRAM и пропускной способности неструктурированных запросов определяется время выполнения волны. В итоге время выполнения ядра определяется суммированием времен выполнения волн.

4. ВЕРИФИКАЦИЯ ПОДХОДА

Верификация предложенного подхода к оценке времени выполнения CUDA-ядер проводилась на ядрах, реализующих следующие алгоритмы: матричного умножения, описанного в руководстве по программированию CUDA [CUDA C++ Programming Guide, 2021], преобразование Фурье, описанного в [Hlavac, 2017]) и обратное распространение ошибки, используемое для обучения нейросетей (представлено в набор бенчмарков Rodinia [Che и др., 2009]). Эксперименты проводились на графическом ускорителе NVIDIA GeForce GTX 1050 [Nvidia GeForce GTX 1050, 2021]. Необходимые для проведения экспериментов параметры GPU были найдены в публикации [Jia и др., 2018] и с помощью бенчмарка [Mei, Chu, 2017].

Результаты, полученные с использованием предложенного подхода, сравнивались с аналогичными оценками, приведенными в статье [Hong, Kim, 2009]. Для более объективного сравнения этот метод был модифицирован с учетом особенностей архитектуры современных графических процессоров. На рис. 6 представлены графики относительной ошибки методов в зависимости от количества варпов для каждого из рассматриваемых ядер. Относительная ошибка измеряется как $E_{rel} = |(t - \hat{t})/t|$, где t – реальное время выполнения ядра, а $\hat{t}$ – прогнозируемое время.

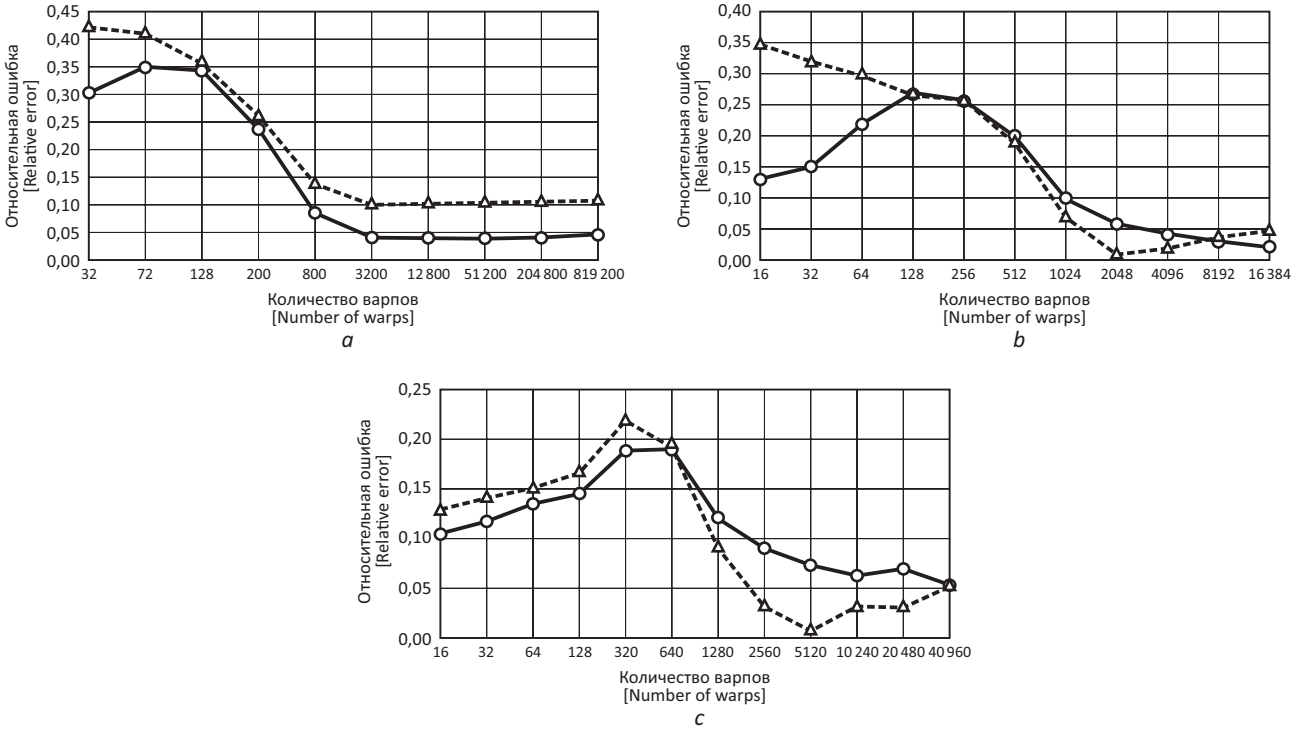


Рис. 6. Графики относительной ошибки прогнозирования времени выполнения CUDA-ядер:

a – матричного умножения; *b* – преобразования Фурье; *c* – обратного распространения ошибки. Сплошной кривой обозначены графики, полученные применением предложенного подхода, пунктирной – результаты, полученные на основе подхода описанного в статье [Hong, Kim, 2009]

Fig. 6. Plots of a relative error of kernels execution time prediction:

a – matrix multiplication; *b* – Fourier transform; *c* – backpropagation. Solid curve is a relative error of our approach, dotted curve is a relative error of the approach described in [Hong, Kim, 2009]

Из рисунков видно, что предлагаемый метод хорошо показал себя на всех рассматриваемых ядрах. Можно заметить, что с увеличением числа варпов относительная ошибка уменьшается, снижаясь до 5%. По сравнению с методом, описанным в [Hong, Kim, 2009] предложенный метод показывает себя лучше при малом количестве варпов, поскольку в этом случае латентности инструкций не компенсируются другими варпами, а в подходе [Hong, Kim, 2009] эти латентности не учитываются.

5. ЗАКЛЮЧЕНИЕ

В работе представлен разработанный метод прогнозирования времени работы CUDA-ядер. Предложены модель графического ускорителя, модель CUDA-ядра и метод оценки времени работы ядра на основе этих моделей. Для построения модели графического ускорителя могут потребоваться результаты запусков ряда микробенчмарков. Процесс построения модели ядра в значительной степени автоматизирован с помощью LLVM фреймворка.

Метод был протестирован на трех ядрах: матричном умножении, преобразовании Фурье и методе обратного распространения ошибки. В результате показано, что метод дает достаточно низкую относительную ошибку.

Из практики разработки CUDA-ядер известно, что время выполнения ядер зависит от параметров, определяющих размеры блоков и сеток. Предложенный метод может быть использован для оптимальной настройки этих параметров, не выполняя программу на GPU.

В качестве дальнейших направлений работы планируется исследовать применение предложенного метода к графическим ускорителям других моделей, разработать метод оценки энергопотребления ядра, основываясь на предложенных моделях. Подход будет расширен возможностью учета конфликтов при обращении к разным банкам разделяемой памяти.

Литература / References

1. *Alavani G.*, *Varma K.*, *Sarkar S*. Predicting Execution Time of CUDA Kernel Using Static Analysis. *IEEE Intl. Conf. on Parallel & Distributed Processing with Applications*, *Ubiquitous Computing & Communications*, *Big Data & Cloud Computing*, *Social Computing & Networking*, *Sustainable Computing & Communications.* 2018. Pp. 948–955. URL: ISPA/IUCC/BDCloud/SocialCom/SustainCom
2. *Arafa Y*. et al. Low Overhead Instruction Latency Characterization for NVIDIA GPGPUs. *IEEE High Performance Extreme Computing Conference* (*HPEC*). 2019. Pp. 1–8.
3. *Baghsorkhi S.S*. et al. An adaptive performance modeling tool for GPU architectures. *ACM SIGPLAN Not*. 2010. Vol. 45. No. 5. Pp. 105–114.
4. *Bakhoda A*. et al. Analyzing CUDA workloads using a detailed GPU simulator. *IEEE International Symposium on Performance Analysis of Systems and Software*. 2009. Pp. 163–174.
5. *Che S*. et al. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization* (*IISWC*). 2009. Pp. 44–54.
6. *Collange S*. et al. Barra: A Parallel Functional Simulator for GPGPU. *IEEE International Symposium on Modeling*, *Analysis and Simulation of Computer and Telecommunication Systems*. 2010. Pp. 351–360.
7. *Hlavac M*. FFT-cuda [Electronic resource]. URL: https://github.com/mmajko/FFT-cuda (дата обращения: 12.01.2021).
8. *Hong S.*, *Kim H*. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Comput. Archit. News*. 2009. Vol. 37. No. 3. C. 152–163.
9. *Jia W.*, *Shaw K.A.*, *Martonosi M*. Stargazer: Automated regression-based GPU design space exploration. *IEEE International Symposium on Performance Analysis of Systems & Software*. 2012. Pp. 2–13.
10. *Jia Z*. et al. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv*. 2018.
11. *Konstantinidis E.*, *Cotronis Y*. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J. Parallel Distrib. Comput*. 2017. Vol. 107. Pp. 37–56.
12. *Lattner C.*, *Adve V*. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization*, *2004*. 2004. Pp. 75–86.
13. *Malhotra G.*, *Goel S.*, *Sarangi S.R*. GpuTejas: A parallel simulator for GPU architectures. *21st International Conference on High Performance Computing*, *HiPC 2014*. 2014.
14. *Mei X.*, *Chu X*. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Trans. Parallel Distrib. Syst*. 2017. Vol. 28. No. 1. Pp. 72–86.
15. *Sim J*. et al. A performance analysis framework for identifying potential benefits in GPGPU applications. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming – PPoPP'12. New York, USA: ACM Press, 2012. P. 11.
16. *Wu G*. et al. GPGPU performance and power estimation using machine learning. *IEEE 21st International Symposium on High Performance Computer Architecture* (*HPCA*). 2015. Pp. 564–576.
17. *Zhang Y.*, *Owens J.D*. A quantitative performance analysis model for GPU architectures. *IEEE 17th International Symposium on High Performance Computer Architecture*. 2011. Pp. 382–393.
18. *Клейменов А.А.*, *Попова Н.Н*. Статически-детерминированный метод прогнозирования динамических характеристик параллельных программ // Вестн. ЮУрГУ. Сер.: Выч. матем. информ. 2021. Т. 10. № 1. С. 20–31. [Kleymenov A.A., Popova N.N. A method for prediction dynamic characteristics of parallel programs based on static analysis. *Bulletin of the South Ural State University. Series*: *Computational Mathematics and Software Engineering*. 2021. Vol. 10. No. 1. Pp. 20–31. (In Russ.)]
19. Nvidia GeForce GTX 1050 [Electronic resource]. URL: https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1050/ (access date: 12.01.2021).
20. CUDA C++ Programming Guide [Electronic resource]. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (access date: 12.01.2021).

Статья проверена программой Антиплагиат. Оригинальность – 95,89%

Статья поступила в редакцию 15.02.2021, принята к публикации 27.03.2021
The article was received on 15.02.2021, accepted for publication 27.03.2021

СВЕДЕНИЯ ОБ АВТОРАХ

Клейменов Анатолий Анатольевич, аспирант факультета вычислительной математики и кибернетики Московского государственного университета имени М.В Ломоносова. Москва, Российская Федерация. ORCID: 0000-0002-2060-1511; E-mail: andreykleimenov@mail.ru

Попова Нина Николаевна, кандидат физико-математических наук; доцент факультета вычислительной математики и кибернетики Московского государственного университета имени М.В. Ломоносова. Москва, Российская Федерация. ORCID: 0000-0002-3339-1365; E-mail: popova@cs.msu.ru

ABOUT THE AUTHORS

Kleimenov Andrey Anatolievich, PhD student at the Department of Computational Mathematics and Cybernetics of the Lomonosov Moscow State University. Moscow, Russian Federation. ORCID: 0000-0002-2060-1511; E-mail: andreykleimenov@mail.ru

Popova Nina Nikolaevna, Cand. Sci. (Eng.); associate professor at the Department of Computational Mathematics and Cybernetics of the Lomonosov Moscow State University. Moscow, Russian Federation. ORCID: 0000-0002-3339-1365; E-mail: popova@cs.msu.ru