



doi 10.26089/NumMet.v23r102

УДК 519.686.4;
519.177

Модель параллельной программы для оценки времени ее исполнения

В. А. Антонюк

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

ORCID: 0000-0003-3069-3365, e-mail: antonjuk@physics.msu.ru

Н. Г. Михеев

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

ORCID: 0000-0002-4927-9394, e-mail: ng.mikheev@physics.msu.ru

Аннотация: Рассматриваются программы, выполняемые на видеокартах общего назначения и представленные в виде “ядер”, не содержащих циклов с неопределенной продолжительностью. Такие ядра могут быть реализованы, например, с помощью технологий CUDA или OpenCL. Для оценки времени работы подобных программ предложены модели их работы: от совсем “наивной” до более реалистичных. Все они формулируются как матричные выражения в max-plus-алгебре.

Ключевые слова: параллельные программы, CUDA, OpenCL, max-plus-алгебра.

Благодарности: Работа выполнена при частичной поддержке гранта РФФИ № 19-29-09044.

Для цитирования: Антонюк В.А., Михеев Н.Г. Модель параллельной программы для оценки времени ее исполнения // Вычислительные методы и программирование. 2022. **23**, № 1. 13–28. doi 10.26089/NumMet.v23r102.

A parallel program model for execution time estimation

Valery A. Antonyuk

Lomonosov Moscow State University, Faculty of Physics, Moscow, Russia

ORCID: 0000-0003-3069-3365, e-mail: antonjuk@physics.msu.ru

Nikita G. Mikheev

Lomonosov Moscow State University, Faculty of Physics, Moscow, Russia

ORCID: 0000-0002-4927-9394, e-mail: ng.mikheev@physics.msu.ru

Abstract: Programs for general-purpose graphics processing units represented as kernels without indefinite loops are considered in this paper. Such kernels can be implemented by CUDA or OpenCL technologies, for example. For execution time estimation, various models of program execution are introduced: from very “naive” to more reliable. All models are presented in the form of matrix expressions in max-plus algebra.

Keywords: parallel programs, CUDA, OpenCL, max-plus algebra.

Acknowledgements: The work was partially supported by the RFBR grant No. 19-29-09044

For citation: V. A. Antonyuk, N. G. Mikheev, “A parallel program model for execution time estimation,” Numerical Methods and Programming. **23** (1), 13–28 (2022). doi 10.26089/NumMet.v23r102.



1. Введение. В настоящее время заметное количество вычислительных задач решается с использованием видеокарт. Поддержка графических процессоров (GPU) появляется не только в специализированных библиотеках для решения узкого круга задач, но и в различных библиотеках общего назначения, таких как Boost [1], OpenCV [2], Octave [3], FFmpeg [4], TensorFlow [5]. Это говорит о том, что видеокарты являются действительно эффективным инструментом для решения широкого круга вычислительных задач, поэтому количество задач, решаемых с использованием подобных устройств, в ближайшее время будет только расти. Однако не все алгоритмы могут получить значительное ускорение за счет использования видеокарт [6]. По этой причине вопрос оценки переносимости и эффективности работы параллельных программ является достаточно актуальным. В настоящей работе будет рассмотрен метод моделирования параллельных программ, который может быть использован для анализа общего времени и временной структуры работы программ, выполняемых на GPU. Как известно, две основные технологии, позволяющие применять видеокарты для решения вычислительных задач общего назначения, — это CUDA [7] и OpenCL [8]; программа, исполняемая на GPU, именуется ядром.

2. Обзор существующих работ. Литература по прогнозированию времени исполнения различных программ на графических процессорах и сопутствующим вопросам достаточно многообразна, поэтому ограничимся здесь только упоминанием сравнительно недавних либо представляющихся авторам ключевыми работ.

В [9] разработан подход к прогнозированию динамических характеристик (под которыми понимаются, например, время выполнения, потребляемая электроэнергия, количество операций с плавающей точкой, объем используемой памяти и др.) параллельных программ без необходимости их запуска на целевой системе. Там же предложена и классификация существующих подходов к подобному прогнозированию: аналитические (когда используется вычисление некоторого аналитического выражения), симуляционные (когда предсказание получается моделированием программы) и гибридные, комбинирующие оба этих подхода. В работе [10] аналитические подходы расширены за счет использования методов машинного обучения на данных, собираемых по результатам запусков программы, и названы там эмпирически детерминированными подходами.

Один из первых подходов предложен в [11] и является комбинацией модели BSP [12], модели PRAM [13] и модели QRQW [14]. Отмечается, что ни одна из этих моделей сама по себе не может объяснить поведение GPU; для более точного моделирования требуются модификации моделей. Использование предлагаемой в работе модели демонстрируется на трех параллельных алгоритмах: умножение матриц, ранжирование списка и построение гистограммы. Эти алгоритмы выбраны из-за того, что один из них является вычислительно затратным, другой интенсивно использует (глобальную) память, а еще один основан на использовании разделяемой памяти, поэтому такой набор полнее раскрывает предложенную модель.

В работе [15] рассматривается аналитическая модель; вводится граф потока работ (work flow graph, WFG) как абстрактная модель ядра и на его основе оценивается время исполнения ядра. Узлами в этом графе являются либо операции с памятью, барьерная синхронизация, блоки непрерывных вычислительных инструкций, либо синтетические узлы входа и выхода. Дуги в нем означают потоки управления либо зависимости от данных из глобальной памяти. Для оценки производительности дугам сопоставляется вес, показывающий среднее число циклов, необходимых для исполнения инструкций узла-источника.

Простая аналитическая модель предложена в [16] для оценивания времени исполнения параллельных программ. Ключевая ее компонента — оценка количества одновременных запросов к памяти (называемого *memory warp parallelism*, MWP) на основе числа исполняемых потоков и потребляемой пропускной способности памяти. Вводится также понятие *CWP* (*computation warp parallelism*), представляющее количество вычислений, осуществляемых другими варпами, пока какой-то из них ожидает данные из памяти. На основе MWP и CWP в модели оценивается затратность обращений к памяти, на основе которой оценивается и общее время исполнения программы.

В [17] разработана (на основе результатов работы микротестов) модель производительности для трех главных компонент исполняемой на GPU программы: конвейера инструкций, доступа в разделяемую память и доступа в глобальную память. Отмечается, что впервые используется набор “родных” инструкций GPU, что критично для точности модели. В отличие от более ранних подходов [15, 16], когда сначала создается аналитическая модель, а затем она верифицируется с помощью набора тестов, здесь авторы предложили действовать в обратном порядке.



Для оценки производительности GPU (конкретно — NVIDIA GT200) в [18] предлагается и анализируется (на примере умножения матриц) разрабатываемый аналитический инструмент TEG (Timing Estimation tool for GPU). Входной информацией для него является дизассемблированный бинарный код ядра CUDA и последовательность исполнения инструкций, получаемая из функциональных симуляторов GPU. Код, однако, не исполняется, а используется лишь информация о времени, затрачиваемом на его исполнение.

В [19] представлен фреймворк анализа производительности, названный GPUPerf. Он использует улучшенную авторами версию модели MWP-CWP [16] для GPU. Предложены также несколько новых метрик для предсказания потенциальных преимуществ в производительности (т. е. возможности уменьшения расхождения между текущей и идеальной производительностью) рассматриваемой системы NVIDIA C2050 (“Fermi”).

В работе [20] предложена техника моделирования производительности GPU-системы, названная GPUMech, на основе интервального анализа, использованного ранее лишь для CPU-систем. Авторы улучшили этот подход моделированием многопоточности и конкуренции за ресурсы памяти.

В [21] проводилось сравнение аналитической модели на основе BSP-модели [12] с тремя различными подходами на основе машинного обучения; было обнаружено, что аналитическая модель обеспечивает лучшую точность, хотя и требует знания характеристик аппаратуры.

В работе [22] предложена абстрактная модель GPU, учитывающая также перенос данных между CPU и GPU. Ожидается, что это поможет усовершенствовать анализ разнородных вычислительных систем, состоящих из CPU и GPU.

Для предсказания времени исполнения программ GPU в [23] предложена аналитическая модель определения времени исполнения CUDA-ядра (но без учета влияния кэширования), использующая статический анализ PTX-кода, экспериментальное тестирование инструкций на пропускную способность, регрессионную модель для латентности глобальной памяти, учет накладных расходов на запуск ядра.

В [24] демонстрируется аналитическая модель оценивания производительности GPU с масштабированием частоты работы ядер и памяти (поскольку современные GPU поддерживают DVFS, Dynamic Voltage and Frequency Scaling). Сначала оцениваются затраты времени на множественные запросы к памяти при разных частотах. Затем используются данные профилирования конкретного ядра на базовой частоте для оценки его производительности на других частотах работы ядер и памяти.

С увеличением количества и типов доступных GPU выбирать для приложений наиболее пригодный ускоритель становится все труднее, поскольку слишком затратно исполнять каждое приложение на каждой системе с GPU. В работе [25] показано, что примененная авторами ранее к процессорам общего назначения (CPU) модель CF (collaborative filtering) может быть использована и для оценки производительности систем с GPU. Вводится некоторая архитектура нейронной сети, обучаемой на векторах характеристик для разных приложений и систем; далее она используется для предсказания производительности какого-либо приложения на конкретной системе.

В работе [26] предлагается метод и модель для предсказания времени исполнения ядра CUDA и его энергопотребления тоже на основе техники машинного обучения. Проверка модели произведена для ядер из различных тестовых наборов и для пяти типов GPU: NVIDIA K20, GTX1650, Titan Xp, P100, V100.

Следует заметить, что в большинстве работ (в том числе и тех, что упомянуты выше) прогнозируются характеристики только CUDA-ядер. Среди работ, рассматривающих также OpenCL-ядра, можно отметить статьи [27–32].

В работе [27] представлены две модели для предсказания производительности ядер OpenCL, исполняемых на GPU: линейная и основанная на машинном обучении. Поскольку часто нет четкого понимания, как в точности функционируют конкретные графические процессоры, их свойства предлагается анализировать специальными микротестами [34].

В [28] OpenCL (как переносимое решение) используется для построения модели производительности GPU от NVIDIA. Для этого собирается информация о том, как ядро использует ресурсы GPU (с помощью аппаратных счетчиков производительности). Далее применяется регрессия и PCA (метод главных компонент), чтобы найти наиболее существенные параметры и сконструировать регрессионную модель производительности. Затем значения наиболее важных компонент замеряются для нужного приложения — чтобы предсказать его производительность.

В работе [29] исследуются (среди прочих задач) времена исполнения ядер CUDA и OpenCL (из NVIDIA CUDA SDK) на архитектурах Fermi и Kepler от Nvidia и показывается, что неподвинутое сравне-

ние платформ CUDA и OpenCL дает сравнимые результаты по производительности; различия во временах исполнения (как в одну, так и в другую сторону) могут быть вызваны поведением компиляторов, создающих РТХ-код.

Поскольку лидирующие изготовители FPGA создали средства разработки для реализации OpenCL-решений применительно к используемой ими архитектуре, в работе [30] разрабатывается аналитическая модель для оценки производительности подобных решений, имеющая определенное сходство с предлагаемым в настоящей статье подходом — в части характера получаемых выражений (имеется в виду частое применение в них операции \max).

Исходя из привлекательности программной модели OpenCL для разнородных высокопроизводительных вычислительных систем, в работе [32] для исследования производительности OpenCL-ядер применяется инструмент AIWC (Architecture Independent Workload Characterization), представляющий собой плагин к Oclgrind [33] (симулятору OpenCL-устройства), созданный авторами. Он осуществляет такую симуляцию вместе со сбором характеристик, не зависящих от конкретной целевой архитектуры, и на их основе строится регрессионная модель, позволяющая предсказать времена исполнения ядер.

В [31] предлагается фреймворк для оценивания производительности OpenCL-ядер на основе статического анализа кода, преобразованного из исходного в промежуточное представление LLVM IR, и последующей генерации последовательности исполнения — без профилирования. Существующие способы оценки производительности ядер разделены на четыре категории: аналитические методы (где строится какая-либо абстрактная модель аппаратуры и ее загрузки, а затем используются дополнительные тесты для получения реальных характеристик аппаратуры); методы, использующие машинное обучение; измерительные методы; симуляционные методы.

Если придерживаться предложенной в [9, 10] классификации, то рассматриваемый в настоящей работе метод можно отнести либо к аналитическому подходу (в том случае, когда используемые в предлагаемых ниже выражениях времена заранее известны, скажем, из информационных материалов производителя), либо к гибриднему, поскольку в случае отсутствия данных о временах их придется определять “экспериментально”, т.е. путем надлежащим образом созданных тестовых программ (аналогично [34]).

Вводимые в настоящей работе графы похожи на предложенные в статье [15], однако отличаются тем, что узлы в них соответствуют состояниям данных; для работы с графами формируются матрицы (почти аналогичные взвешенным матрицам смежности этих графов) и далее — нужные степени таких матриц, причем с использованием при этом специальных операций. Описываемый подход может применяться также и к ядрам OpenCL.

3. Предлагаемая модель. Программа, исполняемая на GPU (как ядро CUDA или ядро OpenCL), в настоящей работе рассматривается с точки зрения перемещения и преобразования обрабатываемых данных и может быть условно представлена ориентированным графом, иллюстрирующим этот процесс. Узлами графа являются результаты промежуточных операций над данными, а ребрами (в случае орграфа их обычно называют дугами) — возможные направления переноса данных. С каждым ребром (дугой) могут быть также связаны конкретные времена, необходимые для таких действий.

Подобный вариант графа похож на граф алгоритма [35, с. 192], но характеризует не алгоритм, а его конкретную программную реализацию. Также он имеет сходство с информационным графом [35, с. 329], но является взвешенным: каждой его дуге сопоставлен некоторый промежуток времени (время, необходимое для перемещения или преобразования информации). Аналитическим представлением подобного графа будет удобно считать матрицу, аналогичную традиционной матрице смежности (но определяемую не самым распространенным способом: элемент a_{ij} задает свойства дуги, идущей из узла j в узел i) и содержащую связанные с дугами графа значения соответствующих времен; отсутствие каких-либо дуг обозначается в ней специальными (“невозможными”) временными значениями, о которых речь идет ниже.

Подходящим формализмом для работы с этими матрицами является так называемая \max -plus-алгебра [36, 37] — поскольку именно ее операции естественным образом возникают при анализе ситуации.

В настоящей работе предлагаются несколько вариантов модели ядер как параллельно исполняемых программ: “наивная”, учитывающая лишь взаимодействие с глобальной памятью, но без учета времени выполнения вычислительных операций; эта же модель с учетом времени выполнения всех вычислений; модель, дополнительно учитывающая последовательный характер обращений к памяти.

Разумеется, это возможно лишь в случае, если ядра не содержат никаких циклов неопределенной продолжительности, исполняемых непредсказуемое количество раз, — по причине зависимости количества итераций в этих циклах от входных данных. Если же такие циклы в ядрах имеются, то можно



оценивать некоторые средние значения времен исполнения ядер при определенных предположениях о статистических распределениях обрабатываемых ядрами данных, однако в настоящей работе этого не делается.

4. Max-plus-алгебра. Max-plus-алгебра \mathbb{R}_{\max} (иногда употребляется название (max, +)-алгебра) — это множество $\mathbb{R} \cup \{-\infty\}$, состоящее из вещественных чисел с присоединенным значением $-\infty$, и две операции, называемые условно “сложением” (обозначается знаком \oplus) и “умножением” (оно далее будет обозначаться знаком \odot , хотя чаще используется знак \otimes ¹). Эти операции определены для любых $a, b \in \mathbb{R}_{\max}$ так:

$$a \oplus b \stackrel{\text{def}}{=} \max(a, b), \quad a \odot b \stackrel{\text{def}}{=} a + b.$$

Нейтральным элементом операции “сложения” является $-\infty$; часто используется также альтернативное обозначение ε (т. е. $x \oplus \varepsilon = \varepsilon \oplus x = x$). Для “умножения” нейтральным элементом будет число 0 (так как $x \odot 0 = 0 \odot x = x$), а значение ε — “поглощающим” (т. е. $x \odot \varepsilon = \varepsilon \odot x = \varepsilon$ для любого $x \in \mathbb{R}_{\max}$).

Данная пара операций может быть распространена далее на матрицы и векторы обычным образом: “суммой” (совместимых по размеру) матриц $A = \{a_{ij}\}$ и $B = \{b_{ij}\}$, записываемой как $C = A \oplus B$, является матрица C с элементами $c_{ij} = a_{ij} \oplus b_{ij}$, а “произведением” $C = A \odot B$ — матрица C с элементами

$$c_{ij} = \bigoplus_{k=1}^n a_{ik} \odot b_{kj} \stackrel{\text{def}}{=} \max_k \{a_{ik} + b_{kj}\}.$$

Привычно определяется и “умножение” матрицы на константу: если $\alpha \in \mathbb{R}_{\max}$, то $\alpha \odot A \stackrel{\text{def}}{=} (\alpha \odot a_{ij})$.

5. “Наивная” модель — степень матрицы ядра. Код ядра получает данные через передаваемые параметры: каждому местоположению этих данных сопоставляется узел графа (можно считать такие узлы входными, поскольку данные из них только извлекаются). Аналогичным же образом возвращаемым из различных копий ядер результатам (с помощью указателей) сопоставляются узлы графа, которые можно считать выходными, поскольку данные туда только записываются². Ребрами предлагаемого графа (реально — дугами, поскольку граф предполагается ориентированным) будут пути перемещения данных, остальными узлами — места хранения и преобразования информации в коде ядра.

Если две или более дуг сходятся в некотором узле, то это значит, что этот набор данных участвует в какой-то операции, осуществляемой в узле. Результат операции будет правилен лишь тогда, когда все данные для нее “готовы”, т. е. имеют “правильные” значения (что определяется временем “прибытия” максимально задержавшегося входного значения). Введением подобного типа узла можно учесть любые преобразования некоторого набора величин в какую-то другую величину, причем со своим временем, необходимым для такого преобразования (в “наивной” модели этим временем будем пренебрегать).

Подобный граф можно назвать графом эволюции информации в ядре; далее он будет именоваться просто графом ядра. Для реальной работы с таким графом удобно сформировать матрицу, содержащую времена переходов между различными узлами графа (будем называть ее матрицей ядра). Столбцы матрицы соответствуют узлам-источникам данных, строки матрицы соответствуют узлам-получателям данных. Если в ходе вычислений данные передаются между двумя узлами, то в элемент, находящийся на пересечении соответствующих столбца и строки, записывается время перехода данных из одного состояния в другое. Если пара узлов не соединена дугой (нет переноса данных между этими состояниями), в соответствующий элемент можно формально записать значение $-\infty$ (роль его станет ясна далее).

Для простых графов (например, таких, как в приводимых далее примерах) кажется, что определить время выполнения кода программы ядра (т. е. перехода данных через весь граф — от входных и до выходных узлов) весьма несложно, однако имеет смысл получить общее выражение для этого времени, чтобы связь рассматриваемой задачи с операциями max-plus-алгебры проступила наиболее отчетливо. Будем следовать соображениям из [37, с. 48–49], где формулируется сходная задача сетевого планирования. Удобно формально дополнить анализируемый граф до полного (напомним, что с отсутствующими дугами при этом будет связано специальное значение $-\infty$); для конкретности будем говорить о полном графе с четырьмя узлами³, хотя аналогичные формулы можно получить и для полных графов с любым

¹Такой выбор сделан потому, что знак \otimes используется также в другом смысле (как произведение Кронекера [38, с. 35]).

²Будем пока предполагать, что не существует узлов, используемых одновременно и для чтения, и для записи, хотя подобные программы тоже бывают; как быть с ними, обсуждается далее.

³Поскольку рассматриваемые далее иллюстративные примеры тоже используют графы с таким же количеством узлов.

количеством узлов. Предполагая, что взаимовлияние узлов в рассматриваемом графе распространяется с каждого узла на каждый, запишем все соотношения в самом общем виде. Обозначим времена появления (формирования) данных на выходах узлов — x_1, x_2, x_3, x_4 , а времена возникновения “правильных” данных на их входах — y_1, y_2, y_3, y_4 ; величинами z_{ij} будем обозначать времена передачи данных с выхода j на вход i . Тогда момент y_1 наступает не ранее, чем пройдет время $z_{11} + x_1$, а также не ранее наступления времен $z_{12} + x_2, z_{13} + x_3, z_{14} + x_4$, т. е. справедливы неравенства $y_1 \geq z_{11} + x_1, y_1 \geq z_{12} + x_2, y_1 \geq z_{13} + x_3, y_1 \geq z_{14} + x_4$ для возможных значений времени y_1 . Интересно иметь его минимально возможным, а это значит, что $y_1 = \max(z_{11} + x_1, z_{12} + x_2, z_{13} + x_3, z_{14} + x_4)$, или, записывая это выражение с помощью других обозначений: $y_1 = z_{11} \odot x_1 \oplus z_{12} \odot x_2 \oplus z_{13} \odot x_3 \oplus z_{14} \odot x_4$. Аналогичные соотношения можно получить и для величин y_2, y_3, y_4 .

Видно, что набор этих равенств — это формулы “умножения” матрицы $Z = \{z_{ij}\}, i, j = 1, \dots, 4$, на вектор $X = \{x_j\}, j = 1, \dots, 4$, — в терминах операций max-plus-алгебры — поэтому общий вариант выражения, описывающего “переход” по дугам графа на один “шаг”, выглядит в новой записи так: $Y = Z \odot X, Y = \{y_j\}, j = 1, \dots, 4$. Матрица Z является взвешенным аналогом матрицы смежности — в довольно редко используемом варианте определения: элемент z_{ij} описывает свойства дуги из узла j в узел i (а не из узла i в узел j , как чаще всего считается). Здесь это существенно, так как позволяет использовать такую матрицу для преобразования, а не только для формирования ее степеней (как это обычно происходит с матрицей смежности): степени матриц для двух упомянутых вариантов определений связаны между собой операцией транспонирования⁴.

По аналогии с тем, как в матрице смежности ее элемент (i, j) характеризует связь из узла j в узел i (подразумевается не совсем традиционное ее определение...), в матрице Z на месте такого же элемента находится время передачи информации из узла j в узел i . Отсутствие связи между узлами в матрице Z обозначается специальным значением $-\infty$, поскольку, как говорилось выше, для операции “умножения” оно является “поглощающим”, а для операции “сложения” — нейтральным. Часто в матрицах вместо него используется специальный символ ε ; в матрицах настоящей работы это значение будет заменяться точкой.

Поскольку матрица Z реализует лишь один этап перехода по графу: от каждого узла по дугам к непосредственно связанным с ним узлам, — необходимо рассматривать также и последовательности из нескольких таких этапов, когда “готовность” входных значений трансформируется в заведомую правильность результирующих. Какая при этом понадобится “степень” матрицы — зависит от длины пути от входных узлов графа до выходных (подобная величина в [35, с. 194] называется высотой графа).

Аналогично тому, как каждая новая степень матрицы смежности некоторого графа позволяет “шаг за шагом” фиксировать в нем все более длинные пути перехода из некоторого узла в какой-либо другой узел, различные степени рассматриваемых здесь матриц в алгебре $(\max, +)$ показывают задержки распространения преобразуемой информации при работе отдельного экземпляра ядра.

В принципе, отдельных элементов “степени” матрицы было бы достаточно для оценки времени прохождения информации через рассматриваемый набор узлов — от входов до выходов, — если считать поступление входных данных одновременным. Конечно, это не так, но пока удобнее будет считать совокупность всех независимых исполнителей находящимися в совершенно равных условиях, — чтобы описать эту параллельную конфигурацию наиболее простым образом (см. раздел об учете числа “исполнителей”).

И, кстати, интересно, что с “физической” точки зрения как операции “сложения”, так и операции “умножения” в $(\max, +)$ -алгебре “формируют” только “однотипные” величины, т. е. одной и той же размерности (что связано с тем, что при “умножении” — являющемся реально сложением — однотипные величины порождают однотипную, а при “сложении” — выборе одной из имеющихся — тем более не может возникнуть величина иного типа); тем самым получается, что в max-plus алгебре и вектор, и преобразующая его матрица, а также любые степени подобной матрицы — все состоят из величин, аналогичных величинам самой матрицы (в данном случае это — времена задержки).

6. Пример кода: ядро суммирования двух векторов. Будем пояснять суть моделей на простом примере кода ядер для сложения двух векторов размерности N (вариант такого ядра для CUDA приведен

⁴Транспонированная степень $(A^n)^\top$ некоторой матрицы A есть просто та же степень ее транспонированной матрицы:

$$(A^n)^\top = \underbrace{(A \cdot A \cdot \dots \cdot A)}_n^\top = \underbrace{A^\top \cdot A^\top \cdot \dots \cdot A^\top}_n = (A^\top)^n.$$

Листинг 1. Код ядра для сложения двух векторов (CUDA)
Listing 1. Vector addition kernel code (CUDA)

```

1  __global__ void vadd(float* a,
2                          float* b,
3                          float* c, int N)
4  {
5      int i = threadIdx.x;
6      if (i < N)
7          c[i] = a[i] + b[i];
8  }
    
```

Листинг 2. Код ядра для сложения двух векторов (OpenCL)
Listing 2. Vector addition kernel code (OpenCL)

```

1  __kernel void vadd(__global float* a,
2                      __global float* b,
3                      __global float* c, int N)
4  {
5      int i = get_global_id(0);
6      if (i < N)
7          c[i] = a[i] + b[i];
8  }
    
```

в листинге 1, для OpenCL — в листинге 2), а потом сформулируем процедуру построения реальных моделей и рассмотрим возникающие при этом затруднения.

Граф ядра формирования суммы двух значений, которые извлекаются из памяти за время t , и дальнейшего размещения этой суммы в памяти за время T , может выглядеть так, как показано на рис. 1. Действия по получению глобального индекса копии ядра и время проверки здесь не учитываются, хотя это легко сделать. Матрица ядра для этого случая (отсутствие переноса данных между узлами графа показано точками) следующая:

$$Z_1 = \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \cdot & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Здесь узлы размещения/формирования данных уже (вполне произвольно) занумерованы: сумма $c[i]$ располагается в узле 1, формирование ее ($c[i] = a[i] + b[i]$) производится в узле 2, а в узлах 3 и 4 располагаются, соответственно, значения $a[i]$ и $b[i]$. Временем формирования суммы в узле 2 пока пренебрегаем (тем более, что на фоне обращений к глобальной памяти оно, как правило, малó: немного утрируя, можно сказать, что часто — особенно для простых ядер — время их исполнения определяется

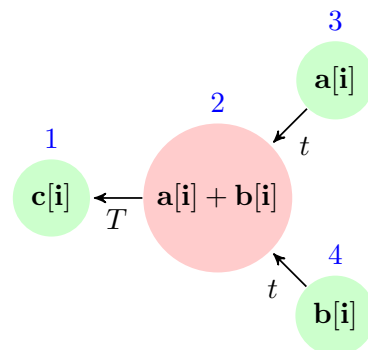


Рис. 1. Граф ядра сложения векторов
Fig. 1. Vector addition kernel graph

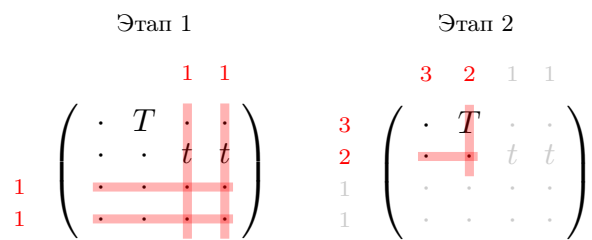


Рис. 2. Определение высоты графа ядра (рис. 1) по матрице ядра
Fig. 2. The kernel graph (Fig. 1) height calculation using the kernel matrix

почти исключительно временами чтения входных данных из глобальной памяти и записи выходных данных обратно).

Хотя для рассматриваемого примера определить в графе максимальную длину пути от входных узлов до выходных несложно, имеет смысл напомнить процедуру получения данной величины в общем случае (она излагается в [35, с. 194] и там эта величина именуется “высотой параллельной формы графа”). Выбираются узлы, не имеющие входящих ребер, помечаются текущим значением индекса (первоначально его значение равно единице) и удаляются из графа — вместе с исходящими из них ребрами, если таковые имеются, после чего текущий индекс увеличивается на единицу. Далее процедура снова повторяется для оставшегося графа — вплоть до исчерпания всех его узлов. На завершающем этапе — когда удаляются последние узлы, а ребер уже не осталось, — значение высоты графа уже сформировано, поэтому количество этапов будет на единицу больше высоты графа.

Применительно к матрице ядра эта процедура будет выглядеть так, как показано на рис. 2: из первоначальной (или потом — из получившейся после предыдущего этапа) матрицы поэтапно “вычеркиваются” (но не удаляются — для возможности указания при них значений индексов, — хотя для подсчета высоты это не нужно) полностью “пустые” строки (т. е. целиком состоящие из “поглощающих” элементов) вместе с соответствующими им столбцами — с присвоением им текущего значения индекса. В рассматриваемом примере в результате получится высота 2 (строки/столбцы 3 и 4 отброшены на первом этапе, строка/столбец 2 — на втором), что означает необходимость получения квадрата матрицы — в смысле умножения в max-plus-алгебре:

$$Z_1^{\odot 2} = Z_1 \odot Z_1 = \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \cdot & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \odot \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \cdot & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & T+t & T+t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Задержки прохождения информации от узлов 3 и 4 до узла 1 в данном случае одинаковы и равны $T + t$. Максимальное значение времени в первой строке матрицы дает гарантированное время формирования результата вычисления для одной копии ядра — если считать, что входная информация в узлах 3 и 4 доступна одновременно.

В принципе, содержимого надлежащей max-plus-степени матрицы ядра Z_1 (в данном случае — $Z_1^{\odot 2}$) достаточно для определения этого времени, однако удобно записать и явное выражение для искомой величины — в общем виде. Вводя обозначения: $\mathbb{1}_i$ — для вектора входов (это вектор-столбец с нулевыми значениями для входных узлов и значениями ε — для всех остальных) и $\mathbb{1}_o$ — для вектора выходов (содержащего нулевые значения для выходных узлов), можно записать время, необходимое для формирования “правильных” выходных величин, — если все входные величины являются “готовыми” в “нулевой” момент времени — и вычислить его: $\langle \mathbb{1}_o, Z_1^{\odot 2} \odot \mathbb{1}_i \rangle = T + t$. Здесь угловыми скобками обозначено скалярное произведение (вычисляемое в max-plus-операциях) двух векторов, приводимых в этих скобках через запятую: первый — это вектор выходов, а второй — результат воздействия квадрата матрицы Z_1 на вектор входов; эти векторы позволяют выделить из матрицы лишь времена распространения информации от входов до выходов, никак не изменяя значений самих времен.

7. Учет конечного числа исполнительных устройств. Если бы все копии ядра (в количестве N) исполнялись параллельно, то это же значение времени было бы длительностью вычисления суммы двух векторов в “наивной” модели, так как описание N копий ядра с помощью N одинаковых компонент графа привело бы к получению матрицы всех N ядер (назовем ее “полной”) и все относящиеся к разным ядрам ее фрагменты были бы одинаковыми.

Однако реальные возможности существующих устройств параллельной обработки часто гораздо скромнее: количество одновременно исполняемых копий ядра в каждый момент времени (т. е. без учета неявного цикла перезагрузки ядер и наличия различных потоков исполнения) будет равно некоторому (другому) значению n , а значит, приводимый выше результат применим лишь к матрице n ядер (ее будем называть “частичной”) и полное время исполнения — вместе с последовательными перезагрузками ядер и перебором потоков исполнения — будет больше приводимого в $\left\lceil \frac{N}{n} \right\rceil$ раз, где $\lceil x \rceil$ обозначает ближайшее к x сверху целое значение.

И “полная”, и “частичная” матрицы будут иметь регулярную структуру, которую удобно описывать при помощи операции, сходной с произведением Кронекера [38, с. 35], — с той лишь разницей, что вместо

обычного умножения в ней будет использоваться max-plus-умножение (обозначение такого “модифицированного” произведения Кронекера оставим стандартным), — $I_n \otimes K$, где K — некоторая матрица ядра, I_n — квадратная матрица $n \times n$ с нулевыми значениями на главной диагонали и равными ε — всеми вне-диагональными; она является “единичной” матрицей в max-plus-алгебре: $X \odot I_n = I_n \odot X = X$, где X — любая матрица размера $n \times n$.

Как известно, для “смешанного” произведения (где используются как традиционное произведение матриц, так и произведение Кронекера) справедливо соотношение $(A \otimes B)(C \otimes D) = AC \otimes BD$, если размеры матриц A, B, C и D позволяют сформировать произведения AC и BD . Это можно объяснить тем, что традиционное произведение матриц отвечает за изменение значений элементов, а произведение Кронекера — за позиционирование самих элементов в результирующей матрице.

Аналогично, для смешанного “модифицированного” произведения Кронекера и произведения матриц в (max, +)-алгебре будет справедливым подобное же соотношение: $(A \otimes B) \odot (C \otimes D) = (A \odot C) \otimes (B \odot D)$, поскольку по сравнению с предыдущим равенством здесь изменился только способ получения значений элементов в матрицах.

Это позволяет доказать (по индукции) справедливость соотношения $(I_n \otimes K)^{\odot m} = I_n \otimes K^{\odot m}$, дающего возможность от вычислений (max, +)-степеней “составной” матрицы из одинаковых матриц ядра K вдоль главной диагонали просто перейти к аналогичным степеням самой матрицы ядра.

Действительно, соотношение очевидно выполняется при $m = 1$, а если предположить, что оно будет справедливо для некоторого m , легко получается, что оно справедливо и для $m + 1$:

$$(I_n \otimes K)^{\odot(m+1)} = (I_n \otimes K)^{\odot m} \odot (I_n \otimes K) = (I_n \otimes K^{\odot m}) \odot (I_n \otimes K) = (I_n \odot I_n) \otimes (K^{\odot m} \odot K) = I_n \otimes K^{\odot(m+1)}.$$

В этих четырех последовательных равенствах первое и четвертое показывают просто свойства “степени” (max-plus-степени); второе использует справедливость соотношения для m (по предположению); третье основано на упомянутом выше соотношении для смешанного произведения Кронекера (“модифицированного”) и произведения матриц в max-plus-алгебре.

Таким образом, время работы программы суммирования двух векторов размерности N , реализованной с помощью ядер (листинги 1 и 2) и исполняемой одновременно n независимыми исполнителями, в “наивной” модели будет равно $\left\lceil \frac{N}{n} \right\rceil \langle \mathbb{1}_0, Z_1^{\odot 2} \odot \mathbb{1}_1 \rangle = \left\lceil \frac{N}{n} \right\rceil (T + t)$, где $\left\lceil \frac{N}{n} \right\rceil$ — ближайшее к $\frac{N}{n}$ сверху целое значение.

Подобная “наивная” модель, конечно, предельно упрощает реальную ситуацию. Во-первых, здесь никак не учитывается время собственно вычислений, а лишь время перемещения данных. В какой-то степени это допустимо, особенно если вычислений не так много. Дело в том, что обращения к глобальной памяти могут быть “расточительны” на фоне затрат на вычисления, хотя наличие кэширования читаемых из нее данных несколько сглаживает ситуацию. Во-вторых, не принимается во внимание тот факт, что иметь “готовыми” (одновременно) все входные данные нереально, потребуется время на их последовательное считывание. В-третьих, игнорируется и принципиальная невозможность одновременного размещения в (глобальной) памяти выходных данных; определенное время необходимо и для этого. Варианты улучшения модели рассмотрены далее.

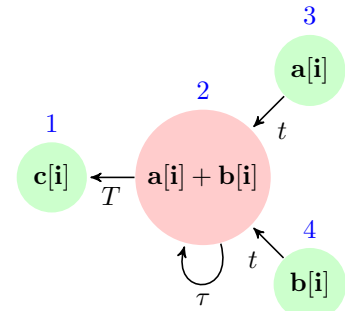


Рис. 3. Граф ядра с учетом операций

Fig. 3. Kernel graph with operations

8. Учет времени выполнения операций. Для того чтобы учесть время какого-либо преобразования, можно добавить петлю в узел (рис. 3), соответствующий этому преобразованию, подразумевая, что результат преобразования (операции) становится “правильным” не сразу, а по истечении времени τ после поступления самой “запоздавшей” из входных величин. Тогда в матрице появится диагональный элемент с нужным значением временного промежутка (здесь это — величина τ):

$$Z_2 = \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \tau & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Однако подобные действия с графом изменяют его характер: первоначально он был ациклическим, теперь же он превращается в так называемый псевдограф, поскольку в нем появляются петли, т.е. дуги из узла в тот же узел. Основная неприятность при работе с псевдографом — при многошаговых переходах по дугам теперь возможны пути, включающие в себя отдельные дуги неоднократно. Таким образом, процедура нумерации узлов графа для получения величины, соответствующей длине пути распространения данных (и определения необходимой степени матрицы ядра), должна быть модифицирована, причем так, чтобы петли в путях от входных до выходных узлов были учтены лишь один раз. Она может быть, например, вот такой.

На каждом этапе выбираются узлы (если таковые есть): либо вообще не имеющие входящих ребер, либо имеющие только петлю без других входящих ребер. Если узлы только с петлями есть, то они помечаются текущим значением индекса, сами петли удаляются, а значение индекса увеличивается на единицу. Если же таких узлов не было, то ситуация сходна с упомянутой в [35, с. 194]: так как петли на узлах без других входящих ребер отсутствуют, можно выбрать узлы без входящих ребер, пометить их текущим значением индекса (возможно, изменяя при этом уже имеющееся у них значение) и затем удалить эти узлы и исходящие из них ребра, после чего снова увеличить индекс на единицу. Далее весь процесс повторяется — вплоть до полного исчерпания узлов.

В результате таких операций инкремента индекса получается целочисленная величина, аналогичная используемой в [35, с. 194] высоте параллельной формы графа, характеризующей число переходов по дугам графа от входных узлов до выходных, а при наличии в узлах графа петель получаемая величина будет учитывать и петли — ровно по одному разу.

Действия применительно к матрице ядра, получаемой из графа ядра, поясняются на рис. 4.

Таким образом, при учете времени вычислений в рассматриваемом примере понадобится уже третья степень матрицы ядра Z_2 , вычисляемая в max-plus-алгебре:

$$Z_2^{\odot 2} = Z_2 \odot Z_2 = \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \tau & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \odot \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \tau & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & T + \tau & T + t & T + t \\ \cdot & 2\tau & \tau + t & \tau + t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$Z_2^{\odot 3} = Z_2^{\odot 2} \odot Z_2 = \begin{pmatrix} \cdot & T + \tau & T + t & T + t \\ \cdot & 2\tau & \tau + t & \tau + t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \odot \begin{pmatrix} \cdot & T & \cdot & \cdot \\ \cdot & \tau & t & t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & T + 2\tau & T + \tau + t & T + \tau + t \\ \cdot & 3\tau & 2\tau + t & 2\tau + t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Время работы программы (с учетом вычислений) будет равно

$$\left\lceil \frac{N}{n} \right\rceil \langle \mathbb{1}_0, Z_2^{\odot 3} \odot \mathbb{1}_1 \rangle = \left\lceil \frac{N}{n} \right\rceil (T + \tau + t),$$

все используемые здесь обозначения были пояснены ранее.

Можно также учесть время выполнения вычислительных операций введением дополнительных дуг, соответствующих этим операциям. Вместо вычислительных узлов в графе будут узлы, соответствующие

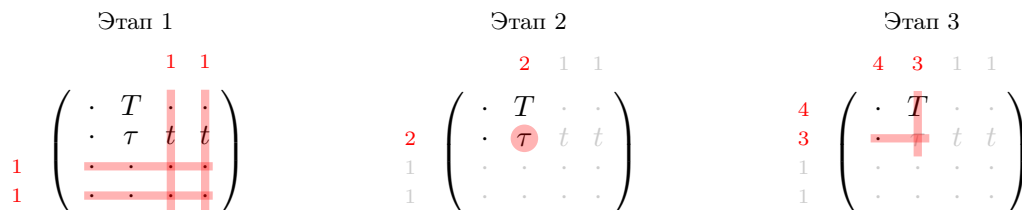


Рис. 4. Построение нумерации узлов графа ядра при наличии диагональных элементов в матрице ядра

Fig. 4. Kernel graph nodes enumeration in the presence of diagonal elements in the kernel matrix



наборам обрабатываемых операциями данных, и по одной дополнительно соединяющей дуге — от наборов данных к результатам операций. Размер матрицы ядра при этом станет больше, зато сам граф будет свободен от циклов, а необходимая степень этой матрицы — от возникающих (из-за наличия диагональных значений матрицы) “посторонних” элементов. Правда, стоит заметить, что при выделении в степени матрицы ядра времен “распространения” информации от входных узлов до выходных все “постороннее” (например, элементы, отличные от $T + \tau + t$ в матрице $Z_2^{\odot 3}$ выше) все равно будет проигнорировано. Какой из двух вариантов учета времени операций предпочтительнее — пока трудно сказать.

9. Учет обращений к глобальной памяти. “Наивный” вариант модели ((max, +)-алгебраическая степень матрицы ядра: $Z_1^{\odot 2}$ или $Z_2^{\odot 3}$ из приводимых примеров) никак не учитывает, что из памяти нельзя читать “параллельно” и в произвольные моменты времени: чтение на самом деле осуществляется последовательно и в лучшем случае небольшими параллельными порциями.

Исполнители различных копий кода ядра могут конкурировать друг с другом за разделяемый ресурс (доступ к глобальной памяти) и получать этот доступ в некотором случайном порядке; можно предположить, что в том же порядке, в каком осуществлялось чтение из памяти, будет происходить и запись в нее результатов разными копиями ядра. В отношении матрицы ядра все это будет означать, что можно модифицировать элементы, соответствующие временам считывания и временам записи, последовательными значениями в некотором согласованном порядке, например по возрастанию их условных номеров.

Однако в наборе одновременно исполняемых копий кода ядра всегда найдется такая, где доступ ко всем данным будет предоставлен с наибольшей задержкой; логично предположить, что формируемые именно этим экземпляром ядра выходные данные будут “готовы” в последнюю очередь.

Представляется, таким образом, что длительность обработки данных набором из n параллельно работающих исполнителей будет всецело определяться работой самого “отстающего” экземпляра, информация из (глобальной) памяти которому будет доступна через время $t + (n - 1)\Delta t$, где t определяет “стандартное” время считывания, а $(n - 1)\Delta t$ — задержку по отношению к “лидирующему” экземпляру ядра⁵; выходная информация попадет в (глобальную) память, соответственно, не через “стандартное” время записи T , а через время $T + (n - 1)\Delta T$, т. е. с задержкой $(n - 1)\Delta T$ (опять же, при условии однократного обращения к памяти для записи).

Следовательно, длительность параллельной обработки (фрагмента) вектора с размером n будет определяться только одним из блоков “частичной” матрицы (т. е. самым “отстающим” экземпляром ядра) с такими величинами:

$$Z_3 = \begin{pmatrix} \cdot & T + (n - 1)\Delta T & \cdot & \cdot \\ \cdot & \tau & t + (2n - 2)\Delta t & t + (2n - 1)\Delta t \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Время работы всей программы (с учетом вычислений и сериализации обращений к глобальной памяти):

$$\left\lceil \frac{N}{n} \right\rceil \langle \mathbb{1}_0, Z_3^{\odot 3} \odot \mathbb{1}_1 \rangle = \left\lceil \frac{N}{n} \right\rceil (T + (n - 1)\Delta T + \tau + t + (2n - 1)\Delta t),$$

где $\left\lceil \frac{N}{n} \right\rceil$ — ближайшее к $\frac{N}{n}$ сверху целое значение ($Z_3^{\odot 3}$ выглядит аналогично $Z_2^{\odot 3}$, полученной ранее).

Стоит отметить, что в некоторых системах при обращении к глобальной памяти может применяться механизм кэширования. В этом случае время повторного считывания данных отличается от времени первого считывания. Однако кэш-память имеет конечный размер, поэтому время переноса данных будет зависеть и от их объема. Все это затрудняет построение адекватной модели, учитывающей использование кэширования. В простейшем варианте учета наличия кэширования памяти можно операциям считывания из нее сопоставлять два значения для соответствующего времени: для самого первого считывания по каждому конкретному адресу — одно, а для последующих считываний — другое (меньшее значение). Подобный вариант неявно предполагает, вообще говоря, неограниченный объем кэширующей памяти, однако представляется вполне допустимым. Кэширование также может быть учтено на этапе построения

⁵При условии однократного обращения к памяти; если же обращений в ядре несколько, задержка будет равна $(2n - 1)\Delta t$ для двух обращений, $(3n - 1)\Delta t$ — для трех обращений и т. д., где Δt — время между последовательными обращениями.

графа путем, например, явного разделения узлов на те, что хранятся в глобальной памяти, и те, что хранятся в кэш-памяти, и присвоения соответствующих времен дугам, выходящим из подобных узлов.

10. Процедура построения графа реального ядра. Разумеется, в общем случае строить такой граф вручную слишком трудозатратно; оптимальным вариантом был бы специальный анализатор кода ядра, но его еще только предстоит реализовать. Перечислим основные этапы построения графа.

Для его создания используется код ядра: либо исходный (на языке C), либо уже откомпилированный машинный. Все циклы (как уже говорилось, они предполагаются циклами с фиксированным числом итераций) полностью разворачиваются. Другие управляющие конструкции (переключатели или ветвления) замещаются либо наиболее долгим по исполнению фрагментом, либо суммой каких-то фрагментов — это зависит от конкретного варианта реализации одновременного исполнения ветвлений аппаратурой. В отношении остальных действий следует знать (или оценить экспериментально) время их исполнения. Необходимо также выбрать наборы состояний данных — сообразно тем преобразованиям, которым они подвергаются; эти наборы данных будут соответствовать узлам графа (и строкам/столбцам матрицы ядра). Далее к узлам добавляются дуги, которым должны сопоставляться времена переносов/преобразований данных (применительно к матрице ядра — это заполнение некоторых ее элементов соответствующими временами, остальные будут иметь “невозможные” значения $-\infty$).

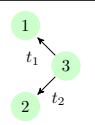
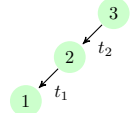
После упомянутых действий, конечно, и программа ядра, и его граф, и матрица могут стать весьма большими, что дополнительно затрудняет ручную обработку. Поэтому можно действовать также и в направлении уменьшения сложности описания кода ядра.

11. Упрощение графа путем понижения детализации. Графы реальных ядер могут иметь весьма высокий уровень детализации и быть довольно сложными, поэтому объемными будут и матрицы ядер. В подобных случаях граф можно анализировать по частям. Для этого имеет смысл выделять в нем подграфы и анализировать их по отдельности, заменяя в исходном графе группы узлов и ребер эквивалентными составными узлами и ребрами⁶. Таким образом, детализация общего графа понизится, а процедура его анализа упростится. В частности, можно объединять последовательно или параллельно соединенные узлы в эквивалентные составные узлы. В последнем столбце табл. 1 приведены выражения для определения времен выполнения составных операций, соответствующих таким узлам.

Стоит также заметить, что операции max-plus-алгебры таковы, что результат практически любых их комбинаций над используемыми величинами (а это времена или наборы времен) будет давать либо сумму некоторого числа промежутков времени, либо набор таких сумм.

Таблица 1. Два простейших графа для ядер-фрагментов: k — необходимая степень матрицы ядра, Z и $Z^{\odot k}$ — матрица ядра и ее степень k , $\mathbb{1}_i$ и $\mathbb{1}_o$ — векторы входных и выходных узлов, $\langle \mathbb{1}_o, Z^{\odot k} \odot \mathbb{1}_i \rangle$ — результирующее время

Table 1. Two simple graphs for fragmentary-kernels: k — the required power of the kernel matrix, Z and $Z^{\odot k}$ — kernel matrix and its k -th power, $\mathbb{1}_i$ and $\mathbb{1}_o$ — input and output node vectors, $\langle \mathbb{1}_o, Z^{\odot k} \odot \mathbb{1}_i \rangle$ — resulting time

| Граф Graph | k | Z | $Z^{\odot k}$ | $\mathbb{1}_i$ | $\mathbb{1}_o$ | $\langle \mathbb{1}_o, Z^{\odot k} \odot \mathbb{1}_i \rangle$ |
|-------------------------------------------------------------------------------------|-----|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------|----------------------------------------------------------------|
|  | 1 | $\begin{pmatrix} \cdot & \cdot & t_1 \\ \cdot & \cdot & t_2 \\ \cdot & \cdot & \cdot \end{pmatrix}$ | $\begin{pmatrix} \cdot & \cdot & t_1 \\ \cdot & \cdot & t_2 \\ \cdot & \cdot & \cdot \end{pmatrix}$ | $\begin{pmatrix} \cdot \\ \cdot \\ 0 \end{pmatrix}$ | $\begin{pmatrix} 0 \\ 0 \\ \cdot \end{pmatrix}$ | $\max\{t_1, t_2\}$ |
|  | 2 | $\begin{pmatrix} \cdot & t_1 & \cdot \\ \cdot & \cdot & t_2 \\ \cdot & \cdot & \cdot \end{pmatrix}$ | $\begin{pmatrix} \cdot & \cdot & t_1 + t_2 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}$ | $\begin{pmatrix} \cdot \\ \cdot \\ 0 \end{pmatrix}$ | $\begin{pmatrix} 0 \\ \cdot \\ \cdot \end{pmatrix}$ | $t_1 + t_2$ |

⁶Например, точки синхронизации исполнителей в ядрах (т.е. вызовы функций `__syncthreads()` в CUDA и `barrier()` в OpenCL) разбивают код ядра на фрагменты, исполнение которых происходит строго последовательно.



12. Заключение. В настоящей работе делаются попытки построения реалистичной модели подсчета времени исполнения параллельных программ в виде ядер CUDA или OpenCL. Предлагаются несколько простых вариантов модели ядер как параллельно исполняемых программ: “наивная”, учитывающая лишь взаимодействие с глобальной памятью, но без учета времени выполнения вычислительных операций; эта же модель с учетом времени выполнения всех вычислений; модель, дополнительно учитывающая последовательный характер обращений к памяти. Все они формулируются как матричные выражения в max-plus-алгебре.

Из очевидных недостатков предложенных вариантов модели можно указать, например, отсутствие учета того, что в реальных GPU вычислительные операции и операции с памятью для различных нитей часто могут перекрываться во времени; это будет приводить к завышенности оценок времени исполнения ядер. Улучшение модели в этом и других направлениях, а также верификация ее вариантов для реально исполняемых ядер и на многочисленных общепринятых тестах может служить направлением дальнейших исследований.

Список литературы

1. Szuppe J. Boost.Compute: a parallel computing library for C++ based on OpenCL // IWOCCL'16: Proceedings of the 4th International Workshop on OpenCL. New York: ACM Press, 2016. doi 10.1145/2909437.2909454.
2. OpenCV API Reference. GPU-accelerated Computer Vision. <https://docs.opencv.org/2.4.13.7/modules/gpu/doc/gpu.html>. Cited January 5, 2022.
3. Bosi L.B., Mariotti M., Santocchia A. GPU linear algebra extensions for GNU/Octave // J. Phys. Conf. Ser. 2012. **368**, N 1. doi 10.1088/1742-6596/368/1/012062.
4. FFmpeg Hardware Acceleration. <https://trac.ffmpeg.org/wiki/HWAccelIntro>. Cited January 5, 2022.
5. Abadi M., Barham P., Chen J., et al. TensorFlow: a system for large-scale machine learning // ArXiv preprint: 1605.08695 [cs.DC]. Ithaca: Cornell Univ. Library, 2016. <https://arxiv.org/abs/1605.08695>. Cited January 5, 2022.
6. Vuduc R., Chandramowlishwaran A., Choi J., et al. On the limits of GPU acceleration // Proc. of the 2nd USENIX Conference on Hot Topics in Parallelism. Berkeley: USENIX Association, 2010. doi 10.5555/1863086.1863099.
7. CUDA C++ Programming Guide PG-02829-001_v11.5. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Cited January 5, 2022.
8. The OpenCL Specification. Khronos OpenCL Working Group. Version V3.0.10., 19 Nov 2021. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html. Cited January 5, 2022.
9. Клейменов А.А., Попова Н.Н. Статически-детерминированный метод прогнозирования динамических характеристик параллельных программ // Вестн. ЮУрГУ. Серия: Вычислительная математика и информатика. 2021. **10**, № 1. 20–31. doi 10.14529/cmse210102.
10. Клейменов А.А., Попова Н.Н. Метод прогнозирования времени выполнения программ для графических процессоров // Computational Nanotechnology. 2021. **8**, N 1. 38–45. doi 10.33693/2313-223X-2021-8-1-38-45.
11. Kothapalli K., Mukherjee R., Rehman M.S., Patidar S., Narayanan P.J., Srinathan K. A performance prediction model for the CUDA GPGPU platform // Proc. 16th International Conference on High Performance Computing. New York: IEEE Press, 2009. 463–472. doi 10.1109/HIPC.2009.5433179.
12. Valiant L.G. A bridging model for parallel computation // Commun. ACM. 1990. **33**, N 8. 103–111. doi 10.1145/79173.79181.
13. Fortune S., Wyllie J. Parallelism in random access machines // Proc. 10th ACM Symposium on Theory of Computing. New York: ACM Press, 1978. 114–118. doi 10.1145/800133.804339.
14. Gibbons P.B., Matias Y., Ramachandran V. The queue-read queue-write PRAM model: accounting for contention in parallel algorithms // SIAM J. Comput. 1998. **28**, N 2. 733–769. doi 10.1137/S009753979427491.
15. Bagsorkhi S.S., Delahaye M., Patel S.J., Gropp W.D., Hwu W.M. An adaptive performance modeling tool for GPU architectures // ACM SIGPLAN Not. 2010. **45**, N 5. 105–114. doi 10.1145/1837853.1693470.
16. 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. **37**, N 3. 152–163. doi 10.1145/1555754.1555775.
17. Zhang Y., Owens J.D. A quantitative performance analysis model for GPU architectures // Proc. IEEE 17th International Symposium on High Performance Computer Architecture. New York: IEEE Press, 2011. 382–393. doi 10.1109/HPCA.2011.5749745.
18. Lai J., Seznec A. Break down GPU execution time with an analytical method // Proc. 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools. New York: ACM Press, 2012. 33–39. doi 10.1145/2162131.2162136.
19. Sim J., Dasgupta A., Kim H., Vuduc R. A performance analysis framework for identifying potential benefits in GPGPU applications // ACM SIGPLAN Not. 2012. **47**, N 8. 11–22. doi 10.1145/2145816.2145819.

20. Huang J.-C., Lee J.H., Kim H., Lee H.-H.S. GPUMech: GPU performance modeling technique based on interval analysis // Proc. 47th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC: IEEE Press, 2014. 268–279. doi 10.1109/MICRO.2014.59.
21. Amarís M., Cordeiro D, Goldman A., Camargo de R.Y. A simple BSP-based model to predict execution time in GPU applications // Proc. IEEE 22nd International Conference on High Performance Computing. Washington, DC: IEEE Press, 2015. 285–294. doi 10.1109/HiPC.2015.34.
22. Carroll T.C., Wong P.W.H. An improved abstract GPU model with data transfer // Proc 46th International Conference on Parallel Processing Workshops. New York: IEEE Press, 2017. 113–120. doi 10.1109/ICPPW.2017.28.
23. Alavani G., Varma K., Sarkar S. Predicting execution time of CUDA kernel using static analysis // Proc. IEEE Intl. Conf. on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications. New York: IEEE Press, 2018. 948–955. doi 10.1109/BDCLOUD.2018.00139.
24. Wang Q., Chu X. GPGPU performance estimation with core and memory frequency scaling // ArXiv preprint: 1701.05308v2 [cs.PF]. Ithaca: Cornell Univ. Library, 2018. <https://arxiv.org/abs/1701.05308>. Cited January 5, 2022.
25. Salaria S., Drozd A., Podobas A., Matsuoka S. Learning Neural representations for predicting GPU performance // Lecture Notes in Computer Science. Vol. 11501. Cham: Springer, 2019. 40–58. doi 10.1007/978-3-030-20656-7_3.
26. Braun L., Nikas S., Song C., Heuveline V., Fröning H. A simple model for portable and fast prediction of execution time and power consumption of GPU kernels // ArXiv preprint: 2001.07104v3 [cs.DC]. Ithaca: Cornell Univ. Library, 2020. <https://arxiv.org/abs/2001.07104>. Cited January 6, 2022.
27. Dao T.T., Kim J., Seo S., Egger B., Lee J. A performance model for GPUs with caches // IEEE Transactions on Parallel and Distributed Systems. 2015. 26, N 7. 1800–1813. doi 10.1109/TPDS.2014.2333526.
28. Karami A., Khanjush F., Mirsoleimani S.A. A statistical performance analyzer framework for OpenCL kernels on Nvidia GPUs // J. Supercomput. 2015. 71, N 8. 2900–2921. doi 10.1007/s11227-014-1338-z.
29. Memarzia P., Khanjush F. An in-depth study on the performance impact of CUDA, OpenCL, and PTX code // Journal of Information and Computing Science. 2015. 10, N 2. 124–136.
30. Wang Z., He B., Zhang W., Jiang S. A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs // HPCA'16: IEEE International Symposium on High Performance Computer Architecture, 2016. doi 10.1109/HPCA.2016.7446058.
31. Wang X., Huang K., Knoll A., Qian X. A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation // Proc. IEEE International Symposium on High Performance Computer Architecture. New York: IEEE Press, 2019. 506–518. doi 10.1109/HPCA.2019.00062.
32. Johnston B., Falzon G., Milthorpe J. OpenCL performance prediction using architecture-independent features // Proc. of International Workshop on High Performance Computing & Simulation. New York: IEEE Press, 2018. 561–569. doi 10.1109/HPCS.2018.00095.
33. Price J. An OpenCL device simulator and debugger. <https://github.com/jrprice/Oclgrind>. Cited January 8, 2022.
34. Wong H., Papadopoulou M., Sadooghi-Alvandi M., Moshovos A. Demystifying GPU microarchitecture through microbenchmarking // Proc. IEEE International Symposium on Performance Analysis of Systems & Software. New York: IEEE Press, 2010. 235–246. doi 10.1109/ISPASS.2010.5452013.
35. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
36. Cuninghame-Green R. Minimax algebra. Berlin: Spinger, 1979. doi 10.1007/978-3-642-48708-8.
37. Кривулин Н.К. Методы идемпотентной алгебры в задачах моделирования и анализа сложных систем. СПб.: Изд-во СПбГУ, 2009.
38. Тьртышников Е.Е. Основы алгебры. М.: Физматлит, 2017.

Поступила в редакцию
19 сентября 2021 г.

Принята к публикации
25 декабря 2021 г.

Информация об авторах

Валерий Алексеевич Антонюк — к.ф.-м.н., доцент, Московский государственный университет имени М.В. Ломоносова, физический факультет, Ленинские горы, 1, стр. 2, 119991, Москва, Российская Федерация.

Никита Глебович Михеев — аспирант, Московский государственный университет имени М.В. Ломоносова, физический факультет, Ленинские горы, 1, стр. 2, 119991, Москва, Российская Федерация.



References

1. J. Szuppe, “Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL,” in *Proc. 4th Int. Workshop on OpenCL, Vienna, Austria, April 19–21, 2016* (ACM Press, New York, 2016), doi 10.1145/2909437.2909454.
2. OpenCV API Reference. GPU-accelerated Computer Vision. <https://docs.opencv.org/2.4.13.7/modules/gpu/doc/gpu.html>. Cited January 5, 2022.
3. L. B. Bosi, M. Mariotti, and A. Santocchia, “GPU Linear Algebra Extensions for GNU/Octave,” *J. Phys. Conf. Ser.* **368** (1) (2012), doi 10.1088/1742-6596/368/1/012062.
4. FFmpeg Hardware Acceleration. <https://trac.ffmpeg.org/wiki/HWAccelIntro>. Cited January 5, 2022.
5. M. Abadi, P. Barham, J. Chen, et al., *TensorFlow: A System for Large-Scale Machine Learning*. ArXiv preprint: 1605.08695 [cs.DC] (Cornell Univ. Library, Ithaca, 2016). <https://arxiv.org/abs/1605.08695>. Cited January 5, 2022.
6. R. Vuduc, A. Chandramowlishwaran, J. Choi, et al., “On the Limits of GPU Acceleration,” in *Proc. 2nd USENIX Conf. on Hot Topics in Parallelism, Berkeley, USA, June 14–15, 2010* (USENIX Association, Berkeley, 2010), doi 10.5555/1863086.1863099.
7. CUDA C++ Programming Guide PG-02829-001_v11.5. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Cited January 5, 2022.
8. The OpenCL Specification. Khronos OpenCL Working Group. Version V3.0.10, 19 Nov 2021. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html. Cited January 5, 2022.
9. A. A. Kleymenov and N. N. Popova, “A Method for Prediction Dynamic Characteristics of Parallel Programs Based on Static Analysis,” *Vestn. Yuzhn. Ural. Gos. Univ. Ser. Vychisl. Mat. Inf.*, **10** (1), 20–31 (2021). doi 10.14529/cmse210102.
10. A. A. Kleimenov and N. N. Popova, “A Method for Prediction Execution Time of GPU Programs,” *Comp. Nanotechnol.* **8** (1), 38–45 (2021). doi 10.33693/2313-223X-2021-8-1-38-45.
11. K. Kothapalli, R. Mukherjee, M. S. Rehman, et al., “A Performance Prediction Model for the CUDA GPGPU Platform,” in *Proc. 2009 Int. Conf. on High Performance Computing, Kochi, India, December 16–19, 2009* (IEEE Press, New York, 2009), pp. 463–472, doi 10.1109/HIPC.2009.5433179.
12. L. G. Valiant, “A Bridging Model for Parallel Computation,” *Commun. ACM* **33** (8), 103–111 (1990). doi 10.1145/79173.79181.
13. S. Fortune and J. Wyllie, “Parallelism in Random Access Machines,” in *Proc. 10th ACM Symposium on Theory of Computing, San Diego, USA, May 1–3, 1978* (ACM Press, New York, 1978), pp. 114–118. doi 10.1145/800133.804339.
14. P. B. Gibbons, Y. Matias, and V. Ramachandran, “The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms,” *SIAM J. Comput.* **28** (2), 733–769 (1998). doi 10.1137/S009753979427491.
15. S. S. Bagsorkhi, M. Delahaye, S. J. Patel, et al., “An Adaptive Performance Modeling Tool for GPU Architectures,” *ACM SIGPLAN Not.* **45** (5), 105–114 (2010). doi 10.1145/1837853.1693470.
16. S. Hong and H. Kim, “An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness,” *ACM SIGARCH Comput. Archit. News* **37** (3), 152–163 (2009). doi 10.1145/1555754.1555775.
17. Y. Zhang and J. D. Owens, “A Quantitative Performance Analysis Model for GPU Architectures,” in *Proc. IEEE 17th Int. Symposium on High Performance Computer Architecture, San Antonio, USA, February 12–16, 2011* (IEEE Press, New York, 2011), pp. 382–393, doi 10.1109/HPCA.2011.5749745.
18. J. Lai and A. Seznec, “Break Down GPU Execution Time with an Analytical Method,” in *Proc. 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Paris, France, January 23, 2012* (ACM Press, New York, 2012), pp. 33–39, doi 10.1145/2162131.2162136.
19. J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, “A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications,” *ACM SIGPLAN Not.* **47** (8), 11–22 (2012). doi 10.1145/2145816.2145819.
20. J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, “GPUMech: GPU Performance Modeling Technique Based on Interval Analysis,” in *Proc. 47th Annual IEEE/ACM Int. Symposium on Microarchitecture, Cambridge, United Kingdom, December 13–17, 2014* (IEEE Press, Washington, DC, 2014), pp. 268–279, doi 10.1109/MICRO.2014.59.
21. M. Amaris, D. Cordeiro, A. Goldman, and R. Y. De Camargo, “A Simple BSP-based Model to Predict Execution Time in GPU Applications,” in *Proc. IEEE 22nd Int. Conf. on High Performance Computing, Bengaluru, India, December 16–19, 2015* (IEEE Press, Washington, DC, 2015), pp. 285–294, doi 10.1109/HiPC.2015.34.
22. T. C. Carroll and P. W. H. Wong, “An Improved Abstract GPU Model with Data Transfer,” in *Proc. 46th Int. Conf. on Parallel Processing Workshops, Bristol, United Kingdom, August 14–17, 2017* (IEEE Press, New York, 2017), pp. 113–120, doi 10.1109/ICPPW.2017.28.

23. G. Alavani, K. Varma, and S. Sarkar, “Predicting Execution Time of CUDA Kernel Using Static Analysis,” in *IEEE Int. Conf. on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/I-UCC/BDCLOUD/SocialCom/SustainCom)*, Melbourne, Australia, December 11–13, 2018 (IEEE Press, New York, 2018), pp. 948–955, doi [10.1109/BDCLOUD.2018.00139](https://doi.org/10.1109/BDCLOUD.2018.00139).
24. Q. Wang and X. Chu, *GPGPU Performance Estimation with Core and Memory Frequency Scaling*, ArXiv preprint: 1701.05308v2 [cs.PF] (Cornell Univ. Library, Ithaca, 2018). <https://arxiv.org/abs/1701.05308>. Cited January 6, 2022.
25. S. Salaria, A. Drozd, A. Podobas, and S. Matsuoka, “Learning Neural Representations for Predicting GPU Performance,” in *Lecture Notes in Computer Science* (Springer, Cham, 2019), Vol. 11501, pp. 40–58. doi [10.1007/978-3-030-20656-7_3](https://doi.org/10.1007/978-3-030-20656-7_3).
26. L. Braun, S. Nikas, C. Song., et al., *A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels* ArXiv preprint: 2001.07104v3 [cs.DC] (Cornell Univ. Library, Ithaca, 2020). <https://arxiv.org/abs/2001.07104>. Cited January 6, 2022.
27. T. T. Dao, J. Kim, S. Seo, et al., “A Performance Model for GPUs with Caches,” *IEEE Trans. Parallel Distrib. Syst.* **26** (7), 1800–1813 (2015). doi [10.1109/TPDS.2014.2333526](https://doi.org/10.1109/TPDS.2014.2333526).
28. A. Karami, F. Khunjush, and S. A. Mirsoleimani, “A Statistical Performance Analyzer Framework for OpenCL Kernels on Nvidia GPUs,” *J. Supercomput.* **71** (8), 2900–2921 (2015). doi [10.1007/s11227-014-1338-z](https://doi.org/10.1007/s11227-014-1338-z).
29. P. Memarzia and F. Khunjush, “An In-depth Study on the Performance Impact of CUDA, OpenCL, and PTX Code,” *J. Inf. Comput. Sci.* **10** (2), 124–136 (2015).
30. Z. Wang, B. He, W. Zhang, and S. Jiang, “A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs,” in *Proc. 2016 IEEE Int. Symposium on High Performance Computer Architecture, Barcelona, Spain, March 12–16, 2016* (IEEE Press, New York, 2016), pp. 114–125, doi [10.1109/HPCA.2016.7446058](https://doi.org/10.1109/HPCA.2016.7446058).
31. X. Wang, K. Huang, A. Knoll, and X. Qian, “A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation,” in *Proc. IEEE Int. Symposium on High Performance Computer Architecture, Washington, DC, USA, February 16–20, 2019* (IEEE Press, New York, 2019), pp. 506–518, doi [10.1109/HPCA.2019.00062](https://doi.org/10.1109/HPCA.2019.00062).
32. B. Johnston, G. Falzon, and J. Milthorpe, “OpenCL Performance Prediction Using Architecture-Independent Features,” in *Proc. Int. Workshop on High Performance Computing & Simulation Orleans, France, July 16–20, 2018* (IEEE Press, New York, 2018), pp. 561–569, doi [10.1109/HPCS.2018.00095](https://doi.org/10.1109/HPCS.2018.00095).
33. J. Price, “An OpenCL Device Simulator and Debugger,” <https://github.com/jrprice/Oclgrind>. Cited January 8, 2022.
34. H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying GPU Microarchitecture through Microbenchmarking,” in *Proc. IEEE Int. Symposium on Performance Analysis of Systems & Software, White Plains, USA, March 28–30, 2010* (IEEE Press, New York, 2010), pp. 235–246, doi [10.1109/ISPASS.2010.5452013](https://doi.org/10.1109/ISPASS.2010.5452013).
35. V. V. Voevodin and V. V. Voevodin, *Parallel Computing* (BHV-Petersburg, St. Petersburg, 2002) [in Russian].
36. R. Cuninghame-Green, *Minimax Algebra* (Springer, Berlin, 1979). doi [10.1007/978-3-642-48708-8](https://doi.org/10.1007/978-3-642-48708-8).
37. N. K. Krivulin, *Methods of Idempotent Algebra for Problems in Modeling and Analysis of Complex Systems* (St. Petersburg Univ. Press, St. Petersburg, 2009) [in Russian].
38. E. E. Tyrtysnikov, *Fundamentals of Algebra* (Fizmatlit, Moscow, 2017) [in Russian].

Received
September 19, 2021

Accepted for publication
December 25, 2021

Information about the authors

Valery A. Antonyuk — Ph.D., Associate Professor, Lomonosov Moscow State University, Faculty of Physics, Leninskie Gory, 1, building 2, 119991, Moscow, Russia.

Nikita G. Mikheev — PhD Student, Lomonosov Moscow State University, Faculty of Physics, Leninskie Gory, 1, building 2, 119991, Moscow, Russia.