УДК 519.61; 519.684.6

ПРИВЕДЕНИЕ ПЛОТНЫХ МАТРИЦ С ЭЛЕМЕНТАМИ ИЗ GF(2) К СТУПЕНЧАТОМУ ВИДУ НА ПЛАТФОРМЕ NVIDIA CUDA.

П. А. Лебедев

МИЭМ НИУ ВШЭ, Москва
e-mail: cygnus@michiru.ru

Описан подход к реализации на программно-аппаратной платформе NVIDIA CUDA метода "четырех русских" приведения плотных матриц с элементами из GF(2) к ступенчатому виду. Получены оценки времени работы алгоритма и рекомендации по выбору параметров алгоритма. Показано, что разработанная реализация алгоритма является самой эффективной по сравнению с существующими решениями для матриц размера $2^{17} \times 2^{17}$.

Ключевые слова: приведение матрицы к ступенчатому виду, метод "четырех русских", NVIDIA CUDA.

REDUCING DENSE MATRICES OVER GF(2) TO ROW ECHELON FORM ON NVIDIA CUDA PLATFORM

P.A. Lebedev

Moscow Institute of Electronics and Mathematics of National Research University "Higher School of Economics", Moscow
e-mail: cygnus@michiru.ru

An approach is described to implementation of the Method of Four Russians for reducing the dense matrices over GF(2) to row echelon form using the NVIDIA CUDA platform. Estimates of the algorithm running time and recommendations on choosing the algorithm parameters are given. It is shown that the developed implementation is most effective in comparison with the existing solutions for matrices of size $2^{17} \times 2^{17}$.

Keywords: reducing a matrix to row echelon form, method of four russians, NVIDIA CUDA.

Введение. Дадим определение двум особым классам матриц (прямоугольных таблиц) с элементами из некоторого поля.

Матрица имеет *ступенчатый вид по строкам*, если

1. все строки, имеющие хотя бы один ненулевой элемент (ненулевые строки), расположены над строками, состоящими из одних нулевых элементов (нулевыми строками);
2. ведущий (первый ненулевой слева) элемент каждой ненулевой строки (кроме первой строки матрицы) располагается строго справа от ведущего элемента предудущей строки.

Матрица имеет *приведенный ступенчатый вид по строкам*, если она имеет ступенчатый вид по строкам и каждый ведущий элемент является единицей, а все остальные элементы в том же столбце — нули.

Приведение матрицы к ступенчатому виду операциями над строками и столбцами, сохраняющими определенные ее свойства, является основой численных методов вычисления рангов и определителей матриц, нахождения обратных матриц и решения систем линейных

уравнений. Линеаризация (в определенных рамках) позволяет использовать данные методы и для нелинейных систем. Программные решения задач линейной алгебры в поле действительных чисел разрабатываются очень давно (например, [4]). Свою реализацию стандарта BLAS имеет и платформа NVIDIA CUDA — cuBLAS [5].

Для матриц с элементами в конечных числовых полях задача приведения к ступенчатому виду встречается весьма часто, например в теории чисел при реализации современных алгоритмов факторизации — разложении больших чисел на множители (алгоритмов непрерывных дробей, квадратичного решета и решета числового поля). В криптографии подобные методы необходимы при определении начального заполнения линейного регистра сдвига и в других задачах. Для обработки матриц с элементами из GF(2) существует несколько общедоступных программных решений, рассмотренных, например, в работах [7, 8, 10, 11]. Все указанные решения используют для вычислений один или несколько центральных процессоров. В данной работе представлена авторская реализация на платформе NVIDIA CUDA метода "четырех русских" приведения матрицы с элементами из GF(2) к ступенчатому виду по строкам. Данная платформа хорошо зарекомендовала себя при решении подобных вычислительных задач, обладающих большим потенциалом для применения высокопараллельных алгоритмов.

Первая попытка реализации указанного метода на данной платформе была предпринята в [14], но результаты не были представлены. Автору известна единственная успешная работа [6], связанная с решением рассматриваемой задачи, однако реализация, рассматриваемая в данной статье, обладает значительно лучшей производительностью, позволяя обработать матрицу порядка 2^{17} за 138 с на видеокарте GeForce GTX 580.

Реализация метода. Платформа NVIDIA CUDA предоставляет набор инструментов для написания высокопараллельных программ для исполнения на современных видеокартах NVIDIA. Программы могут быть написаны как на C-подобном языке, так и на ассемблере. Кратко сформулируем основные понятия и принципы устройства данной системы; детальное описание платформы дано в [12].

Графический процессор (ГП) обладает собственными вычислительными блоками, построенными по архитектуре SIMT (Single Instruction–Multiple Threads), и памятью, имеющей несколько областей и способов доступа. Его работа осуществляется параллельно с работой центрального процессора (ЦП). Подпрограмма для выполнения на ГП называется *ядром* и выполняется много раз каждым из потоков, число которых задается *конфигурацией выполнения*. Потоки

организуются в трехмерные *блоки*, а блоки — в трехмерную *сетку*. Потоки внутри одного блока могут использовать быструю общую память и примитивы синхронизации. Планировщик выполнения потоков ГП оперирует группами по 32 потока — *ворпами*. В ворпе все потоки выполняют одновременно одну и ту же инструкцию. Если из-за условий, зависящих от данных, пути выполнения потоков в ворпе расходятся, образовавшиеся ветви выполняются последовательно, снижая производительность. Половина ворпа также является единицей формирования транзакций доступа к памяти. Перечислим основные типы памяти с точки зрения ядра:

- регистры, выделяемые на каждый поток;
- общая память — небольшая быстрая память, доступная для чтения и записи всеми потоками блока, у каждого блока своя;
- константная память — небольшая кэшируемая память только для чтения;
- глобальная память — основной массив памяти ГП, доступна для чтения и записи;
- локальная память — часть глобальной памяти, используемая при нехватке регистров;
- текстурная память — доступ к области основной памяти ГП через блоки выборки текстур с их специфическими режимами адресации и интерполяции, только для чтения;
- поверхности — вариант текстурной памяти с возможностью записи, поддерживается только новыми моделями ГП.

Взаимодействие программы ЦП с ГП осуществляется через потоки команд, содержащие вызовы ядер и операции копирования данных между оперативной памятью и/или памятью ГП.

Рассмотрим метод Гаусса, являющийся основой метода “четырех русских” для обращения.

Для системы с элементами из GF(2) алгоритм Гаусса [2] не содержит шага деления строки на значение ее ведущего элемента, поэтому применение алгоритма состоит из чередования двух шагов:

— поиск строки с ненулевым элементом в текущем столбце и обмен местами строк, если это необходимо;

— вычитание текущей строки из всех нижележащих с ненулевым элементом в текущем столбце.

Этим шагам соответствуют два ядра CUDA, вызываемые контролирующей программой хост-системы. Выделение этих двух шагов в отдельные ядра связано с необходимостью постоянного изменения конфигурации вычислительной сетки во избежание простоя больших групп потоков.

Очевидным является хранение матрицы в построчном формате с использованием одного бита на элемент. Это позволяет совершать многие операции с числом элементов строки, равным числу бит в слове, за одну инструкцию. Для архитектуры CUDA оптимальным является выбор 32-битного слова.

Поиск ведущей строки. Выбранный формат хранения имеет свой недостаток: неэффективный доступ к памяти при поиске ведущей строки, так как элементы одного столбца матрицы хранятся в битах далеких друг от друга слов. Для преодоления этого недостатка используется параллельный поиск: n потоков считывают элементы последовательных строк и каждый поток, считавший ненулевой элемент, обновляет ячейку общей памяти атомарной операцией записи. Если ненулевых элементов не было найдено, процесс повторяется для следующих n строк. Строки между ведущей и первой, содержащей ненулевой элемент, исключаются из операции вычитания строк. Поскольку алгоритм использует барьеры, число потоков параллельного поиска ограничено сверху максимальным числом потоков на блок, что для современных ГП составляет 512 или 1024 [12]. Этого оказывается достаточно (увеличение числа потоков свыше примерно 300 не дает дополнительного ускорения). Применение параллельного поиска уменьшает общее время вычислений для сильно разреженных матриц (плотностью меньше $1/1000$), где эта операция является преобладающей по времени (на два порядка). Для плотных матриц параллельный поиск замедляет работу (до 10 %), и более удачным является ограничение числа потоков поиска размером ворпа как минимальной исполняемой единицы. Использование атомарных операций над общей памятью требует устройств, поддерживающих CUDA capability 1.2 [12], а для устройств с меньшими возможностями используется однопоточный поиск. Использование глобальной памяти для размещения общей ячейки делает работу алгоритма неприемлемо медленной.

Если найденная строка не является первой, то ее необходимо обменять местами с первой. Для ускорения этой операции часто используется индексный массив строк, но для реализации на CUDA оказывается быстрее обменять само содержимое строк, отказавшись от использования дополнительной индексации. Это связано с относительно редкой необходимостью в этой операции для плотных матриц и невозможностью сделать полностью совмещенным доступ к индексам из ядра вычитания строк.

Вычитание ведущей строки. Сам по себе алгоритм вычитания ведущей строки при выбранном формате хранения матрицы тривиален, однако его реализация критически важна, так как для плотных матриц его время работы составляет подавляющую часть всего времени вычислений.

Данные вычитаемой строки используются на этом шаге многократно, поэтому важно выбрать оптимальный способ их хранения. Из всех видов памяти и способов доступа, предоставляемых платформой CUDA, глобальная память имеет слишком большую латентность, поэтому для рассмотрения остаются общий, константный и текстурный разделы памяти. Автором были реализованы все три подхода и было установлено, что они не имеют существенных отличий по скорости выполнения. Было выбрано использование текстуры, поскольку такая реализация проще всего расширяется до метода "четырех русских", который будет рассмотрен далее.

Другим важным параметром, влияющим на производительность, является конфигурация запуска ядра вычитания строк. Очевидно, что данное ядро является ограниченным по памяти. Для выбора оптимальных конфигураций используется предварительный тест производительности на случайных данных, перебирающий все возможные конфигурации запуска с числом потоков, кратным половине ворпа для матриц разного размера. Результаты калибровки согласуются с результатами теоретических расчетов занятости блоков ГП по [13]. Данный тест также определяет оптимальную форму блока для необходимого числа потоков в нем и учитывает особенности формы самой обрабатываемой матрицы. Это позволяет увеличить производительность в 1,5–2 раза по сравнению с ручным выбором конфигурации. Время работы калибровки в результаты работы авторской реализации не включено, поскольку ее необходимо провести всего один раз для конкретной видеокарты.

Метод "четырех русских" для обращения (*англ.* M4RI) является относительно простым расширением метода Гаусса. Алгоритм был представлен в [1] и детально рассмотрен в [3]. Его основную идею можно пояснить с помощью рис. 1.

После того как был найден и установлен на место ведущий элемент (шаг 1), делается попытка привести к каноническому виду по строкам подматрицу из $l \cdot k$ строк, начиная с текущей, где l и k — параметры алгоритма (шаг 2). При этом формируется единичная матрица размера $k \times k$ с использованием обычного метода Гаусса. Если данная операция успешна, то формируется полный набор из всех 2^k линейных комбинаций данных строк, что можно эффективно сделать с использованием кодов Грея (шаг 3). После этого проводится обнуление сразу k столбцов оставшейся части матрицы за один проход (шаг 4).

В [3] предлагаются значения параметров $l = 3$ и $k = \log_2 m$ для матрицы размера $m \times n$; следовательно, вероятность успеха на шаге

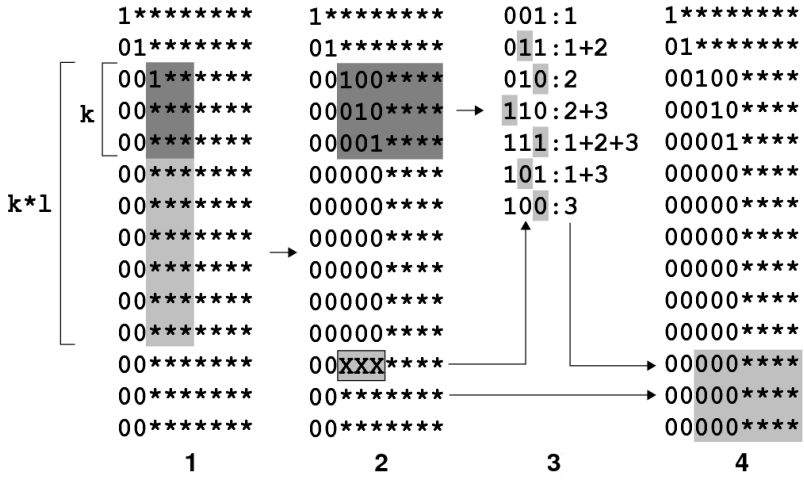

Рис. 1. Метод “четырех русских”

приведения подматрицы к каноническому виду для матрицы с равновероятным распределением нулевых и единичных элементов есть $1 - m^{-2}$.

В [6] показано, что несколько более эффективным является обработка подматрицы из $l \cdot k$ строк на ЦП, даже с учетом потерь на пересылку данных в оперативную память и обратно. Это утверждение было проверено и в авторской реализации. В первом варианте использовалось копирование подматрицы в оперативную память и обработка ее простой реализацией метода Гаусса на ЦП. Область адресного пространства, используемая для хранения подматрицы, была зафиксирована в оперативной памяти и зарегистрирована как таковая в CUDA. Во втором варианте использовалось единое ядро, содержащее оба шага метода Гаусса. Это удалось сделать за счет того, что рабочая область памяти невелика и имеет фиксированный размер в течение всего времени выполнения этого шага. Для кэширования текущей вычитаемой строки использовалась общая память, поскольку это единственный вид памяти, доступный из ядра на запись[1]. Получено, что реализация с обработкой подматрицы на ЦП оказалась более быстрой и в этой работе. Ускорение невелико и составляет около 3–5 % для больших матриц, но для матриц порядка до 2^{13} может достигать 30 %.

Данная реализация автоматически уменьшает эффективное k при отсутствии возможности формирования единичной матрицы полного размера. Поддержка групп столбцов, находящихся в двух смежных 32-битных словах, позволяет получить максимальный прирост производительности при k, не делящих 32 нацело. Тем не менее, алгоритм

[1]Не считая глобальной памяти и требующих CUDA capability 2.0 или выше записываемых поверхностей, которые автором не рассматривались из-за ограничений на их размер.

Рис. 2. Схема реализации

стремится минимизировать число ситуаций, где используется такая поддержка, поскольку она более требовательна к и без того занятой полосе пропускания глобальной памяти.

Таблица кодов Грея вычисляется заранее на хосте для максимального поддерживаемого k и хранится в константной памяти. Используя рефлективное свойство кодов Грея, хранится только половина таблицы, а построение комбинаций осуществляется в два параллельных прохода. Такой метод оказывается на 10–40 % быстрее генерации в один полный проход. Комбинации хранятся (как и в реализации метода Гаусса) в общей памяти, обращение к которой на чтение осуществляется через текстуру. Хотя коды Грея могут быть относительно легко вычислены самим ядром, использование готовых таблиц оказывается быстрее. Для всех поддерживаемых k компилируются варианты ядра с полностью развернутым циклом.

Итоговая схема реализации алгоритма приведена на рис. 2.

Анализ результатов. Тестирование проводилось на машине с процессором Core i7 920 и видеокартой GeForce GTX 580 под управлением ОС Linux. Видеокарта на данной системе не использовалась для вывода изображения, что значительно уменьшило разброс значений между несколькими прогонами. Для сравнения была использована библиотека M4RI [8] версии 20110715, собранная с поддержкой многопоточности, которая широко применяется и обладает максимальной производительностью на ЦП для большинства практических случаев [9]. Тестировались две реализации алгоритма приведения матрицы к ступенчатому виду: метод “четырех русских” (m4ri) с автоматическим

выбором значения k и $PLUQ$-разложение[2] (pluq). Для авторской реализации M4RI на CUDA получены результаты для $k = 1$ (метода Гаусса), $k = 4, 8$ и 14. Результаты для квадратных матриц различного порядка приведены в таблице. Для исследования влияния производительности видеокарты на работу алгоритма также использованы данные, полученные на видеокарте GeForce GTX 550 на машине с ОС Windows.

Затраты времени на обработку матриц размера $2^N \times 2^N$

N	Время, с					
	M4RI		CUDA			
	m4ri	pluq	Гаусс	$k = 4$	$k = 8$	$k = 14$
10	0,00223	0,00254	0,0251	0,0138	0,0107	0,0576
11	0,00601	0,0109	0,0506	0,0262	0,0175	0,115
12	0,0212	0,0525	0,126	0,0604	0,0399	0,238
13	0,113	0,279	0,397	0,182	0,119	0,571
14	1	1,81	1,77	0,797	0,537	1,83
15	7,71	12,5	11	4,89	3,39	4,6
16	85,7	115	77,4	35,1	25,9	20,7
17	1097	928	611	286	209	138

При малых размерах матриц реализация M4RI обладает значительно меньшей латентностью, поскольку содержит соответствующие оптимизации, а показатели CUDA включают в себя время на пересылку матрицы на ГП и обратно и другие постоянные издержки использования графического адаптера. Для матриц, начиная с размера $2^{14} \times 2^{14}$, вперед выходит реализация на CUDA, причем бóльшие k показывают себя лучше на бóльших матрицах (рис. 3).

Данные результаты в целом соотносятся с ожидаемыми в соответствии с асимптотической сложностью используемых алгоритмов: $O(n^3)$ для метода Гаусса и $O(n^3/\log n)$ для метода "четырех русских". Результаты для метода Гаусса несколько лучше ожидаемых для большинства матриц, кроме самых малых и самых больших. Это объясняется наличием резерва вычислительной мощности, не используемого в первом случае и полностью используемого в последнем, данный эффект наблюдается только на самых мощных видеокартах. Для метода "четырех русских" результаты соответствуют ожидаемым с минимальной погрешностью.

[2]Данные методы выбраны как документированные в соответствующем заголовочном файле библиотеки, детально их реализация не рассматривалась.

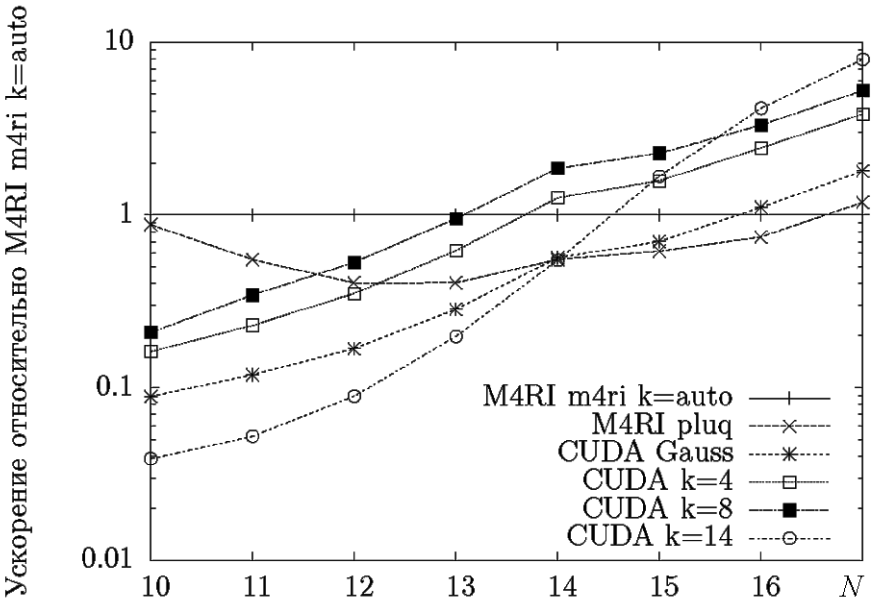

Рис. 3. Относительное время работы при обработке матриц размера $2^N \times 2^N$ разными методами

В [3] указано, что при незначительном отклонении от используемой оценки оптимального значения k скорость обработки мало отличается от оптимальной. Для малых матриц слишком велика доля временных затрат, не связанных напрямую с выполнением алгоритма, что не позволяет делать выводов по поводу этого утверждения. На больших матрицах видно, что оптимальное k оценке $\log_2 m$ не соответствует. Это связано с аппаратными особенностями графической карты и различных видов памяти, используемых в реализации алгоритма. Более того, эти значения различны для разных видеокарт.

Что касается сравнения с реализацией [6], выполненной с использованием той же технологии, провести его в полной мере не удалось из-за неполноты результатов, представленных в этой работе. Тем не менее, приведенный в ней результат — 71 с для матрицы 64000×65536, полученный на видеокарте GeForce GTX 480, во много раз хуже лучшего для данной реализации.

Отдельно отметим вопрос энергопотребления вычислительной системы во время работы алгоритма. Необходимые измерения проводились с помощью простого ваттметра, входящего в конструкцию имевшейся системы. Хотя он и не является профессиональным измерительным инструментом, но позволяет приближенно оценить изменение энергопотребления всего компьютера во время расчетов относительно состояния бездействия.

Реализация метода “четырех русских” на CUDA отличается стабильным энергопотреблением на протяжении всего времени расче-

тов, которое складывается из потребления видеокарты и одного ядра ЦП. Для $k = 1$ оно составляет 260 Вт и несколько уменьшается для бóльших k, составляя 220 Вт при $k = 14$. Это объясняется большей долей времени выполнения ядер, использующих только один блок вычислительной сетки и, следовательно, только один мультипроцессор ГП. В то же время объем работы, выполняемый ядром вычитания линейных комбинаций из строк матрицы, от самого k фактически не зависит.

Реализация библиотеки M4RI обладает поддержкой многопоточности с использованием OpenMP и потенциально может использовать четыре ядра и восемь логических потоков используемого процессора. Ее энергопотребление в процессе работы претерпевало значительные изменения, особенно для *PLUQ*-разложения, что, вероятно, связано с особенностями используемого алгоритма и отсутствием возможности полностью загрузить все ядра ЦП. Среднее энергопотребление составляет около 70 Вт для обоих алгоритмов, что в три раза меньше, чем для реализации на CUDA. Учитывая более чем шестикратное преимущество реализации на CUDA по времени вычисления, можно считать, что энергетические и, следовательно, финансовые затраты на работу данных реализаций в худшем случае сравнимы.

Заключение. В данной работе была представлена реализация метода "четырех русских" на графических картах NVIDIA с использованием технологии CUDA. Показано, что она обладает лучшей или сравнимой производительностью по сравнению с наиболее быстрой из известных многопоточных реализаций на ЦП. Дополнительные задержки, связанные с пересылкой данных и инициализацией видеокарты, могут быть минимизированы в случае использования видеокарты как основного вычислительного процессора. Учитывая специфику используемой аппаратуры, вместо стандартных формул для расчета оптимальных параметров следует использовать динамическую калибровку. Для малых матриц существующие реализации на ЦП остаются более выгодными.

Автор полагает, что реализация асимптотически более быстрых методов приведения матрицы с элементами из GF(2) к ступенчатому виду не сможет показать своих преимуществ из-за ограниченного объема видеопамяти. Этот вопрос и использование нескольких видеокарт для обхода данного ограничения являются направлениями дальнейших исследований.

СПИСОК ЛИТЕРАТУРЫ

1. Арлазаров В. Л., Диниц Е. А., Кронрод М. А., Фараджев И. А. Об экономном построении транзитивного замыкания ориентированного графа. Докл. АН СССР. – 1970. – Т. 194, № 3. – С. 487–488.

2. В о л к о в Е. А. Численные методы. – М.: Наука, 1987. – С. 139–150.
3. G r e g o r y V. B a r d. Matrix Inversion (or LUP-Factorization) via the Method of Four Russians, in $\Theta(n^3 \log n)$ Time. http://www.math.umd.edu/ bardg/m4ri.new.pdf
4. BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/
5. T h e NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library. http://developer.nvidia.com/cublas
6. D e n i s e D e m i r e l. Effizientes Lösen linearer Gleichungssysteme über GF(2) mit GPUs — 27. September 2010. http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Denise_Demirel.diplom.pdf
7. GAP — Groups, Algorithms, Programming — a System for Computational Discrete Algebra. http://www.gap-system.org/
8. M4RI — Lienar Algebra over F_2. http://m4ri.sagemath.org
9. M4RI Performance over GF(2). http://m4ri.sagemath.org/performance.html
10. Magma Computational Algebra System. http://magma.maths.usyd.edu.au/magma/
11. NTL: A Library for doing Number Theory. http://www.shoup.net/ntl/
12. N V I D I A C o r p o r a t i o n NVIDIA CUDA C Programming Guide Version 4.1 11/18/2011. http://developer.download.nvidia.com/compute/DevZone/docs/html /C/doc/CUDA_C_Programming_Guide.pdf
13. N V I D I A C o r p o r a t i o n CUDA GPU Occupancy Calculator. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/ tools/CUDA_Occupancy_Calculator.xls
14. J é r é m i e T h a r a u d, R a f a ë l L a u r e n t. Linear algebra over the field with two elements using GPUs. http://moais.imag.fr/membres/ jeanlouis.roch/perso_html/transfert/2009-06-19-IntensiveProjects-M1-SCCI-Reports/ tharaud_laurent_article.pdf /

Статья поступила в редакцию 6.12.2011

Павел Александрович Лебедев — ведущий программист кафедры “Компьютерная безопасность” МИЭМ НИУ ВШЭ. Автор трех научных работ в области высокопараллельных вычислительных систем.

P.A. Lebedev — leading programmer of Computer Safety department of the Moscow Institute of Electronics and Mathematics of National Research University Higher School of Economics. Author of three publications in the field of highly parallel computing systems.