УДК 004.021:519.683:519.684

РЕАЛИЗАЦИЯ АЛГОРИТМА РЕШЕНИЯ НЕСИММЕТРИЧНЫХ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

С. Н. Чадов[1]

Рассматриваются вопросы численного решения разреженных систем линейных алгебраических уравнений на графических процессорах общего назначения. Системы решаются на основе варианта алгоритма BiCG-STAB. Приводится описание алгоритма, излагаются несколько форматов представления разреженных матриц с учетом особенностей архитектуры графических процессоров NVIDIA. Производительность предложенной реализации на трех различных графических процессорах сравнивается с производительностью аналогичного алгоритма на центральных процессорах. Обсуждается зависимость производительности от разных факторов. Предлагаются направления дальнейшего совершенствования алгоритмов.

Ключевые слова: параллельные вычисления, разреженные СЛАУ, GPGPU, CUDA, BiCG-STAB, графические процессоры.

1. Введение. Задача решения систем линейных уравнений находит широкое применение во многих инженерных и научных исследованиях в самых разных областях, от химической кинетики до компьютерной графики. Вследствие этого быстрая программная реализация алгоритма, решающего данную задачу, является весьма полезной. Этой тематике посвящено большое количество работ. В последнее время возрастает интерес к возможности переноса части вычислений на графические процессоры общего назначения (далее ГПУ — графические процессорные устройства, GPU — Graphics Processing Units), такие как NVIDIA GeForce, Tesla и AMD Radeon. Производители этих устройств предоставляют API (Application Programming Interfaces) для их использования в качестве процессоров общего назначения (в частности, в настоящей статье используется технология NVIDIA CUDA — Compute Unified Device Architecture) и заявляют крайне высокие значения пиковой производительности своих устройств (до 1 TFlops на GeForce GTX 285), тогда как современные центральные процессоры (ЦП) обеспечивают производительность менее 100 GFlops. Таким образом, реализация алгоритмов решения систем линейных уравнений на ГПУ представляется весьма выгодной.

Этой тематике посвящено немало работ, например [1, 2]. Однако в этих работах авторы были вынуждены, ввиду отсутствия на тот момент специализированных API, использовать стандартные средства графического программирования, такие как API OpenGL и языки программирования шейдеров, не очень хорошо подходящие для решения подобных задач и имеющие поэтому ряд ограничений. К тому же большинство подобных реализаций либо ограничивается решением систем с матрицами какого-либо специального вида, например ленточных, либо не приспособлены для решения разреженных систем. Однако в реальных задачах (например в энергетике) часто приходится иметь дело с разреженными системами очень большой размерности (десятки и сотни тысяч элементов и более), не сводящихся к одному из специальных видов. Одной из первых работ, посвященных решению систем линейных уравнений на ГПУ с использованием API общего назначения, является работа [3], которая реализует метод сопряженных градиентов (описание можно найти в [3, 4]). Однако предложенную в [3] реализацию также нельзя признать универсальной, поскольку метод сопряженных градиентов не позволяет решать несимметричные системы.

Настоящая статья призвана устранить этот недостаток. Предлагаемая реализация алгоритма решения систем линейных уравнений основана на реализации, изложенной в [3], однако вместо метода сопряженных градиентов использует стабилизированный метод бисопряженных градиентов (BiCG-STAB), позволяющий решать несимметричные системы.

2. Метод BICG-STAB. Метод BiCG-STAB является модификацией метода бисопряженных градиентов, обеспечивающей лучшую сходимость. Подробное описание метода можно найти в [4], здесь приведем лишь его краткую схему.

[1] Ивановский государственный энергетический университет, факультет информатики и вычислительной техники, ул. Рабфаковская, д. 34, 153003, г. Иваново; аспирант, e-mail: sergei.chadov@gmail.com

В алгоритме используются следующие обозначения: $Ax = b$ — решаемая система; $x^{(0)}$ — начальное приближение; ϵ — некоторая константа, характеризующая точность вычислений; M — матрица предобусловливания. В качестве способа предобусловливания воспользуемся методом Якоби [4], который хотя и гораздо менее эффективен в плане ускорения сходимости, чем, например, метод неполного LU-разложения [4] или метод SPAI (Sparse Approximate Inverse) [5], однако в отличие от них тривиально распараллеливается на большое количество потоков.

Вычислить $r^0 = b - Ax^{(0)}$ для некоторого начального значения $x^{(0)}$
$\widehat{r} = r^{(0)}$
for $i = 1, 2, \ldots$
  $\rho_{i-1} = \widehat{r}\, r^{(i-1)}$
  if $\rho_{i-1} = 0$, **return** <<Ошибка>>
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1}\nu^{(i-1)})$
  Решить $Mp' = p^{(i)}$
  $\nu^{(i)} = Ap'$
  $\alpha_i = \rho_{i-1}/\widehat{r}^{\,\mathrm{T}}\nu^{(i)}$
  $s = r^{(i-1)} - \alpha_i\nu^{(i)}$
  if $||s|| < \epsilon$
    $x^{(i)} = x^{(i-1)} + \alpha_i p'$
    return
  Решить $Ms' = s$
  $t = As'$
  $\omega_i = t^{\mathrm{T}}s/t^{\mathrm{T}}t$
  $x^{(i)} = x^{(i-1)} + \alpha_i p + \omega_i s'$
  $r^{(i)} = s - \omega_i t$
  Проверить сходимость, и если она не достигнута
  и $\omega_i \neq 0$, перейти к следующей итерации

3. Реализация. CUDA предоставляет реализацию процедур пакета BLAS (Basic Linear Algebra Subprograms) для плотных матриц, поэтому из примитивов линейной алгебры необходимо дополнительно реализовать только операцию умножения разреженной матрицы на вектор (SpMV). Рассмотрим эту операцию подробнее, поскольку, как будет видно далее, именно эта операция занимает больше всего времени при работе алгоритма.

В общем виде эту операцию можно выразить формулой $y_i = \sum_j a_{ij}x_j$. Способ вычисления по данной формуле в общем случае зависит от формата представления разреженной матрицы. Следуя работе [3], мы используем формат BCRS (Blocked Compressed Row Storage) с размерами блоков 1×1, 2×2 или 4×4. При размере блока, равном одному, формат BCRS становится эквивалентен более распространенному формату CRS. Итак, матрица в формате BCRS представляет собой:

— массив data ненулевых элементов в случае CRS-формата или массив data $(n \times n)$-блоков, содержащих хотя бы один ненулевой элемент и окрестности, в случае BCRS-формата;

— массив индексов ci, указывающих, в каком столбце матрицы находится соответствующий ненулевой элемент или блок;

— массив rp индексов первых ненулевых элементов каждой строки матрицы.

Например, в табл. 1 и 2 приведены CRS-представления и BCRS (2×2)-представления для конкретной матрицы A.

Несмотря на то что общий объем данных при размере блока, большем единицы, может увеличиваться, блочное представление значительно снижает количество необходимых косвенных обращений к памяти, что может существенно увеличить производительность. Реализация описанных вариантов представления разреженной матрицы на ГПУ различна для каждого размера блоков. Для блока размера 1×1 каждый из трех массивов просто копируется на ГПУ в неизменном виде. Для блоков 2×2 представление аналогично — с той разницей, что блоки ненулевых элементов запаковываются в тип данных float4. В случае размера блока 4×4 массив ненулевых элементов разбивается на четыре подмассива, содержащих блоки размера 2×2, аналогично реализации в [3].

Рассмотрим некоторые приемы, позволяющие увеличить производительность алгоритма умножения

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Таблица 1

CRS-представление матрицы A

data	1	5	7	3	9	1	5	4	2	3
ci	0	1	2	4	3	4	5	0	4	3
rp	0	1		3	4		6	7	8	9

Таблица 2

BCRS-представление матрицы A

data	1	0	0	0	0	0	3	0	0	0	0	5	0	0	2	0
	0	5	7	0	0	9	1	0	4	0	0	0	0	3	0	0
ci	0		1		1		2		0		2		1		2	
rp	0				2				4				6			

разреженной матрицы на вектор на ГПУ NVIDIA. Для этих устройств очевидным узким местом является доступ к глобальной памяти (в [6] утверждается, что обращение к глобальной памяти имеет латентность в 400–600 тактов). Однако, используя некоторые особенности процессоров NVIDIA, можно добиться значительного прироста производительности.

Во-первых, в документации NVIDIA [6] утверждается, что при выполнении определенных условий в одну операцию обращения к памяти могут быть объединены (coalesced) 16 запросов к памяти от различных потоков. Условия, при которых может произойти такое объединение, достаточно сложны и отличаются для различных поколений графических процессоров, однако минимизация количества необъединенных обращений к глобальной памяти приводит в большинстве случаев к значительному приросту производительности.

Во-вторых, производительность можно повысить на основе следующего наблюдения: в алгоритме SpMV каждый элемент вектора x используется несколько раз. Поэтому для того, чтобы избежать чтения из глобальной памяти уже считанного значения, можно применять кэширование. Простой и эффективный способ реализации кэширования на ГПУ NVIDIA состоит в том, чтобы хранить вектор x в текстуре, обращения к которой кэшируются аппаратно. Каждый элемент матрицы A используется при умножении лишь однажды, однако для нее также можно применить кэширование в случае, когда матрица содержит относительно плотные подматрицы, которые могут быть загружены в более быструю память при первом обращении к одному из элементов. В нашем подходе такое кэширование может частично выполняться за счет использования формата BCRS, однако существуют и другие подходы. Например, в работе [7] предлагается осуществлять анализ матрицы с целью нахождения плотных блоков, которые затем, на этапе непосредственного умножения, кэшируются в разделяемой (shared) памяти блока потоков.

В-третьих, реализация алгоритма SpMV на ГПУ может быть выполнена различным образом по отношению к распределению элементов матрицы A между потоками исполнения. Простейший вариант — использовать один поток на строку матрицы. Недостатком такого подхода является то, что обращение к элементам матрицы A выполняется неоптимальным для устройства образом (подробнее см. [8]). В качестве альтернативного варианта в [8] предлагается так называемая "векторная" реализация, использующая несколько потоков для обработки каждой строки матрицы. Такая реализация обладает как рядом достоинств, так и рядом недостатков. В частности, сами авторы [8] отмечают, что "скалярная" реализация эффективнее векторной, если среднее количество ненулевых элементов в строке матрицы мало. Еще одна особенность "векторной" реализации заключается в том, что она меняет порядок вычислений, вследствие чего могут возникнуть проблемы устойчивости. В наших экспериментах реализация из [8] показала себя неэффективной, более эффективной для платы GeForce GTX оказалась модификация "векторной" реализации, описанная в [7], но использующая 8 потоков на строку матрицы вместо 16 в [7]; для платы GeForce 8800 GT более эффективной оказалась скалярная реализация.

4. Результаты. Было проведено сравнение реализации алгоритма на нескольких различных ГПУ и ЦПУ. Результаты приведены на рис. 1. На нем изображено среднее время, за которое выполняется 200 итераций алгоритма BiCG-STAB на различных устройствах. Для каждого устройства приведены результаты для блоков размером 1×1, 2×2 и 4×4. Поскольку здесь наибольший интерес представляет эффективность реализации метода решения линейных систем на ГПУ, а не сравнение производительности ГПУ между собой, на графике приведены только результаты наиболее быстрого для данного устрой-

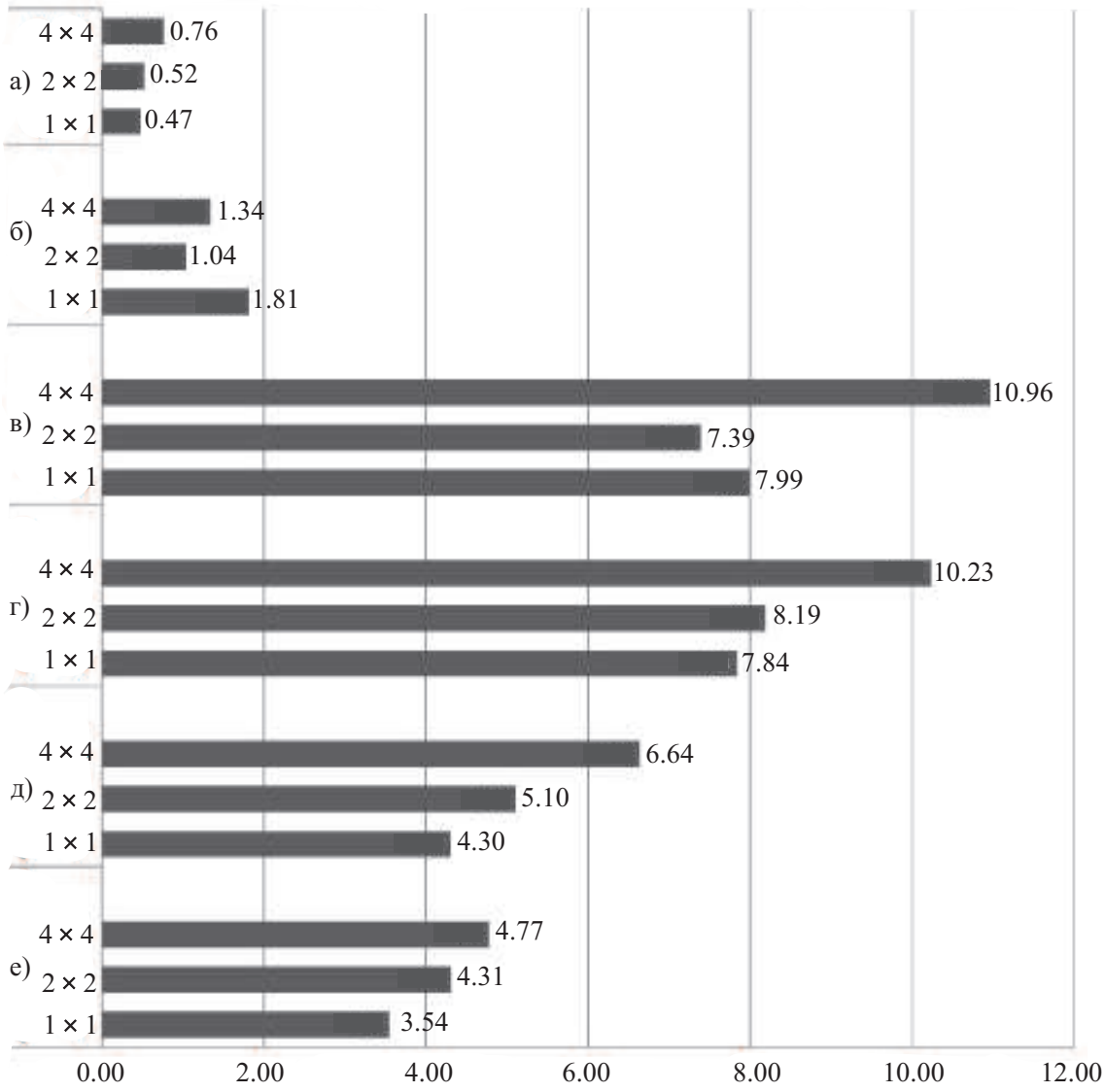

Рис. 1. Результаты выполнения алгоритма на различных устройствах: а) GeForce GTX260, б) GeForce 8800 GT, в) GeForce 8500 GT, г) Athlon X2 4600 + @2.4 GHZ, д) Core 2 Duo E6600 @2.4 GHZ, е) Core 2 Quad Q6600 @2.4 GHZ

ства алгоритма. Так, для GeForce GTX260 использовался "векторный" алгоритм SpMV с 8 потоками на блок, для GeForce 8800 GT и 8500 GT — скалярный алгоритм. В обоих случаях было использовано текстурное кэширование. Для центральных процессоров использовался вариант скалярного алгоритма с применением Intel Math Kernel Library, дополнительно распараллеленной на необходимое количество потоков при помощи библиотеки Intel TBB (Threading Building Blocks), а также собственной реализации некоторых операций с использованием инструкций SSE (Streaming SIMD Extensions). Во всех реализациях вычисления велись с 32-битной точностью.

Из рисунка видно, что на ГПУ удалось добиться значительно большей производительности (примерно в 7.5 раз для GTX260 и в 3.4 раза для 8800 GT), чем на четырехъядерном ЦПУ. Даже младшая модель платы GeForce (имеющая всего 16 процессоров и 128-битную шину памяти) показала производительность, сравнимую с производительностью двухъядерного ЦПУ.

Среди форматов представления матриц заметно, что формат 4×4 обеспечивает меньшую эффективность в большинстве случаев, поэтому его использование, вероятно, будет эффективно только для матриц особого вида. Формат 2×2, напротив, практически не уступает (на GeForce GTX260) или превосходит (на GeForce 8800 GT и 8500 GT) формат CRS и поэтому, наряду с ним, может быть рекомендован в качестве формата для представления разреженных матриц общего вида на ГПУ.

Попытаемся проанализировать, каким образом производительность предложенной реализации может быть повышена далее. Наиболее очевидное решение — увеличение количества потоковых процессоров. Как видно из графика, плата GeForce GTX260, имеющая 216 процессоров, опережает плату GeForce 8800 GT, имеющую 112 процессоров. Перспективным является совместное использование нескольких ГПУ, что теоретически позволяет использовать до 1000 потоковых процессоров даже в относительно недорогих системах.

Второй очевидный путь повышения производительности — увеличение частоты работы каждого конкретного процессора и памяти. Рассмотрим отдельно зависимость производительности алгоритма от частоты потоковых процессоров и частоты памяти (рис. 2 и 3).

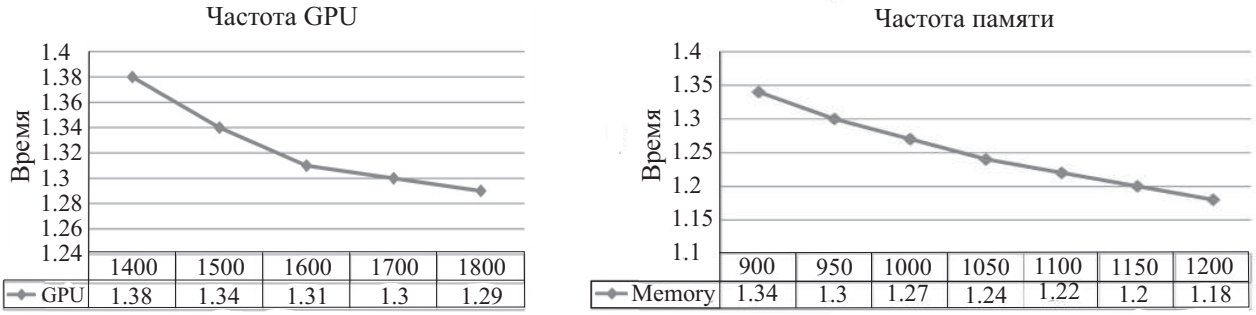

Рис. 2. Зависимость времени работы алгоритма от частоты памяти и ГПУ для GeForce 8800 GT

Как видно из приведенных графиков, для GeForce 8800 в обоих случаях зависимость близка к линейной. Однако в абсолютных значениях средний прирост производительности при увеличении частоты памяти на мегагерц более чем в два раза превышает прирост производительности при увеличении частоты ГПУ на мегагерц. Следует также отметить, что прирост производительности на мегагерц частоты ГПУ замедляется после отметки в 1600 мегагерц, что говорит о том, что частота ГПУ не является существенно ограничивающим фактором для производительности. Для GeForce GTX260 зависимость от частоты ГПУ носит достаточно нерегулярный характер, что скорее всего объясняется какими-то внутренними особенностями реализации, зависимость же от частоты памяти аналогична этой зависимости для GeForce 8800. Эти наблюдения подтверждают выводы, сделанные нами в предыдущем разделе из теоретических соображений, о первостепенной важности производительности подсистемы памяти для реализации этого алгоритма.

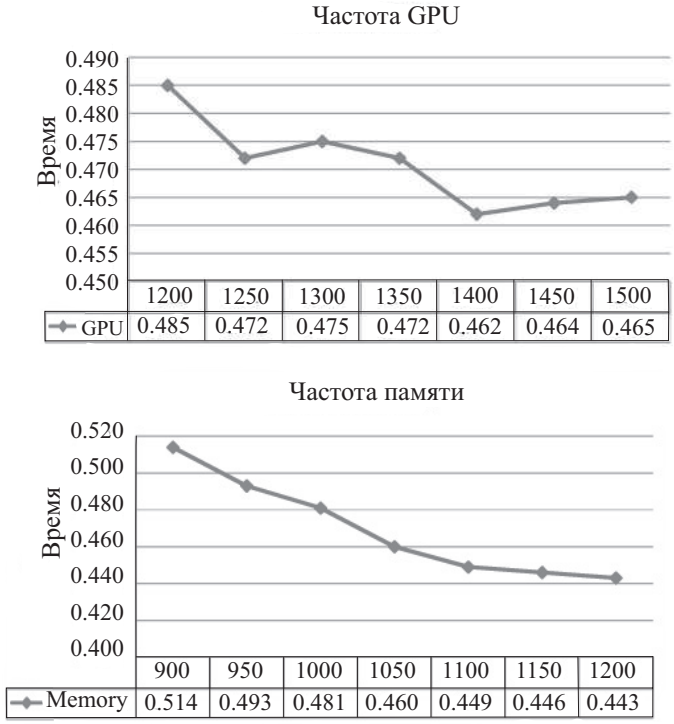

Рис. 3. Зависимость времени работы алгоритма от частоты памяти и ГПУ для GeForce GTX260

В разделе 3 основное внимание было уделено оптимизации функции произведения разреженной матрицы на вектор. Для того чтобы убедиться в оправданности такого решения, рассмотрим распределение времени выполнения алгоритма BiCG-STAB по функциям, вызываемым в процессе выполнения (рис. 4). Как следует из описания алгоритма (раздел 2), каждая итерация главного цикла требует 6 операций AXPY ($ax + y$), 4 скалярных произведения, 2 вычисления нормы, 2 вычисления предобусловливателя (которые в данном случае превращаются в простую операцию покомпонентного перемножения векторов, обозначенного на рис. 4 “vecvec_mul”).

Из графика видно, что более 65% времени вычисления занимает функция умножения разреженной матрицы на вектор и еще около 15% — операция AXPY, реализованная в CUBLAS (CUDA BLAS).

5. Выводы и направление дальнейших исследований. Предложенная реализация алгоритма решения разреженных систем линейных уравнений на ГПУ показывает значительно более высокую производительность по сравнению с реализацией аналогичного алгоритма на центральном процессоре и может оказаться достаточно полезной во многих научных и инженерных расчетах. Эффективность этой реализации обеспечивается во многом оптимизацией, учитывающей особенности оборудования (так, для GeForce 8800 GT наиболее оптимальной оказалась скалярная реализация SpMV и размер BCRS-блока 2×2, а для GTX260 — векторная реализация SpMV и CRS-формат). Существуют возможности увеличения быстродействия как алгоритмические (наиболее перспективным является, на наш взгляд, реализация более эффективного предобусловливания), так и путем перехода на одновременное использование нескольких ГПУ.

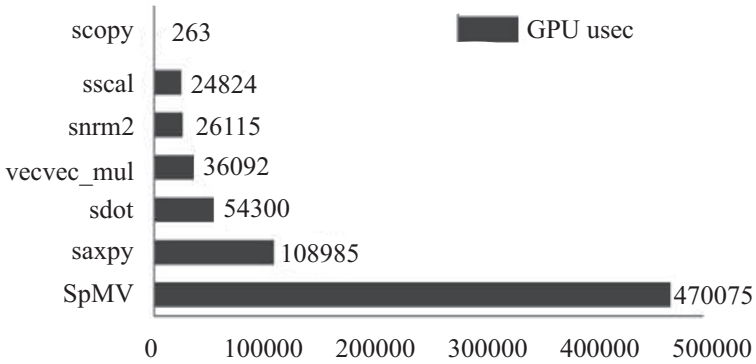

Рис. 4. Распределение времени выполнения алгоритма BiCG-STAB по функциям

Однако нельзя не отметить ряд присущих данной реализации особенностей, которые могут сузить область ее применения. Наиболее значительной, по нашему мнению, является ограниченность вычислениями с одинарной точностью, причем даже одинарная точность может в некоторых случаях не достигаться, поскольку CUDA-совместимые устройства не полностью реализуют стандарт IEEE-754 [6]. Несмотря на то что новые платы (compute capability 1.3 по терминологии NVIDIA) поддерживают вычисления с двойной точностью, эта поддержка реализована таким образом, что вычисления с двойной точностью выполняются в 8–10 раз медленнее вычислений с одинарной точностью.

СПИСОК ЛИТЕРАТУРЫ

1. *Krüger J., Westermann R.* Linear algebra operators for gpu implementation of numerical algorithms // ACM Transactions on Graphics. 2003. **22**, N 3. 908–916.
2. *Bolz J., Farmer I., Grinspun E., Schröder P.* Sparse matrix solvers on the gpu: conjugate gradients and multigrid // ACM Transactions on Graphics. 2003. **22**, N 3. 917–924.
3. *Buatois L. et al.* Concurrent number cruncher: a gpu implementation of a general sparse linear solver // International Journal of Parallel, Emergent and Distributed Systems (to appear).
4. *Barrett R. et al.* Templates for the solution of linear systems: building blocks for iterative methods. Philadelphia: SIAM, 1994.
5. *Grote M., Huckle T.* Parallel preconditioning with sparse approximate inverses // SIAM Journal on Scientic Computing. 1997. **18**, N 3. 838–853.
6. NVIDIA. CUDA Programming Guide, 2.1 edition. 2009.
7. *Baskaran M., Bordawekar R.* Optimizing sparse matrix-vector multiplication on GPUs. IBM Tech. Rep. 2009.
8. *Bell N., Garland M.* Efficient sparse matrix-vector multiplication on cuda. NVIDIA Tech. Rep. 2008.

Поступила в редакцию
26.04.2009