ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 76 за 2011 г.

Сивкова Т.А., Довгилович Л.Е., Воскобойникова О.И., Софронов И.Л.

Алгоритмы QD арифметики высокой точности на CUDA для работы с матрицами

Рекомендуемая форма библиографической ссылки: Алгоритмы QD арифметики высокой точности на CUDA для работы с матрицами / Т.А.Сивкова [и др.] // Препринты ИПМ им. М.В.Келдыша. 2011. № 76. 28 с. URL: http://library.keldysh.ru/preprint.asp?id=2011-76

ИНТСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М. В. КЕЛДЫША
РОССИЙСКОЙ АКАДЕМИИ НАУК

Т. А. СИВКОВА, Л. Е. ДОВГИЛОВИЧ,
О. И. ВОСКОБОЙНИКОВА, И.Л. СОФРОНОВ

АЛГОРИТМЫ QD АРИФМЕТИКИ ВЫСОКОЙ ТОЧНОСТИ НА CUDA ДЛЯ РАБОТЫ С МАТРИЦАМИ

Москва

Т. А. Сивкова, Л. Е. Довгилович, О. И. Воскобойникова, И.Л. Софронов

АЛГОРИТМЫ QD АРИФМЕТИКИ ВЫСОКОЙ ТОЧНОСТИ НА CUDA ДЛЯ РАБОТЫ С МАТРИЦАМИ

Аннотация.

Исследована эффективность библиотеки QD для работы с числами с длинной мантиссой на графических ускорителях Tesla C1060, GeForce GTX480, Tesla C2050. К библиотеке добавлены алгоритмы на CUDA для основных операций линейной алгебры. Алгоритмы оптимизированы для получения максимальной производительности. Получено, что Tesla C2050 может ускорять вычисления с матрицами в арифметике высокой точности до 100 раз; результат близок к предельному, поскольку почти такое же ускорение имеют и сами арифметические операции с quad-double.

T.A. Sivkova, L.E. Dovgilovich, O.I. Voskoboynikova, I.L. Sofronov

CUDA ALGORITHMS OF SOME LINEAR ALGEBRA OPERATIONS FOR THE EXTENDED PRECISION ARITHMETIC LIBRARY QD

Abstract.

We investigated efficiency of the extended precision library QD on the GPU Tesla C1060, GeForce GTX480, and Tesla C2050. We added CUDA algorithms of main linear algebra operations to the library. The algorithms are optimized for maximal performance. In particular, matrix computations with extended precision on Tesla C2050 have about 100x speed-up; this is close to the limit performance since the elementary arithmetic operation have the same rate.

Т. А. Сивкова, Л. Е. Довгилович (Московский физико-технический институт (государственный университет) (МФТИ)).

Работа выполнена при финансовой поддержке РФФИ, №10-01-00567.

Введение

Основная масса инженерных и научных расчетов выполняется при использовании двойной точности (реже – одинарной). Однако, существует ряд вычислительных задач, где бывает недостаточно даже двойной точности вычислений для достижения результата. Например, в задачах вычисления физических и математических постоянных. В этом случае нужно использовать числа более высокой точности, т.е. числа с более длинной мантиссой. Учитывая то, что алгоритмы операций с числами высокой точности имеют весьма высокую вычислительную сложность, их использование достаточно трудоемко и требует большого времени выполнения на обычных компьютерах. Современные параллельные вычислительные системы позволяют значительно ускорить выполнение программ, но доступ к ним ограничен. Прорывом в ускорении вычислений оказалось использование графических карт. При правильном использовании они могут дать значительный выигрыш в производительности, а также они более удобны в эксплуатации.

Проблемой арифметики высокой точности занимались уже давно. Например, Кнут в 80-ые годы прошлого столетия в своей книге «Искусство программирования» рассматривал несколько подходов к реализациям операций с такими числами [1]. В настоящее время разработано несколько библиотек для работы с числами произвольной и фиксированной точности. В нашей работе мы рассмотрим библиотеки DD (Double-Double) and QD (Quad-Double) Package [2]; как для обычных процессоров, так и для графических процессоров с использованием технологии CUDA.

В библиотеках DD и QD Package числа представлены 31и 62 цифрами мантиссы, соответственно. Обе библиотеки выполнены на C++, а также имеют аналоги с использованием технологии CUDA. В них реализованы основные алгебраические операции (+, −, *, /), операции сравнения, а также некоторые трансцендентные функции – *sqrt, exp, sin* и другие. Мы расширяем их возможности и добавляем основные операции с матрицами и векторами.

В дальнейшем эти библиотеки будут использованы для вычислений матриц оператора прозрачных граничных условий [3], [4]. По своей природе эти вычисления требуют арифметику высокой точности (из-за необходимости численного выполнения обратного преобразования Лапласа). Существующая реализация алгоритма на FORTRAN использует числа real*8 и real*16. Последние реализуются программно при компиляции, вследствие чего выполнение программы резко замедляется. Кроме того, иногда и этой точности не хватает.

Анализ эффективности полученных алгоритмов проводится на различных GPU для выяснения целесообразности применения той или иной графической карты для работы с числами высокой точности.

1. Рассмотренные вычислительные возможности

В данной работе были использованы три различные графических карты: Tesla C1050, GeForce GTX480, Tesla C2050. Они обладают различными вычислительными возможностями, см. [5]. Основные характеристики этих графических процессоров представлены в таблице 1 – 1.

Таблица 1 – 1. Технические характеристики

Характеристики	Tesla C1050	GeForce GTX480	Tesla C2050
Число потоковых ядер процессора	240	480	448
Объем памяти	4 GB	1.5 GB	3 GB
Частота работы ядер	1.3 GHz	1.4 GHz	1.15 GHz
Тактовая частота памяти	0.8 GHz	1.85 GHz	1.5 GHz
Производительность операций с плавающей запятой двойной точности (пиковая)	78 GFlops	168 GFlops	515 GFlops
Производительность операций с плавающей запятой одинарной точности (пиковая)	933 GFlops	1345 GFlops	1030 GFlops
Пропускная способность глобальной памяти (coalesced)	69.7 GB/s	151.8 GB/s	90.2 GB/s
Пропускная способность разделяемой памяти (coalesced)	100.9 GB/s	270.2 GB/s	221.1 GB/s

2. Основные функции-операторы работы с числами двойной точности (double), используемые для реализации алгоритмов арифметических операций с числами высокой точности

В библиотеках используется одинаковая идея реализации вычислений на основе чисел двойной точности. Суть в том, что каждое double число (64 бита) имеет мантиссу в 53 бита, при умножении двух double чисел получается также double число, т.е. половина чисел мантиссы просто теряются. Чтобы этого не происходило, переопределяются операции с double числами.

Следуя [2] введем следующие обозначения: обычные знаки арифметических операций с double числами будем заключать в круглые скобки, а переопределенные операции – не будем. Для любой бинарной операции $(\cdot) \in (+,-,*,/)$ введем

$$fl(a \cdot b) = a(\cdot)b$$

для обозначения ближайшего к верному результату числа с мантиссой в 53 бита, и

$$err(a \cdot b) = a \cdot b - fl(a \cdot b)$$

для обозначения разности между точным значением и $fl(a \cdot b)$. При помощи этих обозначений можно описать некоторые операции, обоснование корректности алгоритмов для которых дано в [6].

Основные функции-операторы:

1. Split (a): разделяет double число с 53 битной точностью на 2 части, по 26 бит точности каждое. Таким образом, что $a = a_{hi} + a_{lo}$. a_{hi} содержит первые 26 бит мантиссы, a_{lo} – оставшиеся 26 бит. (Теоретическая оценка числа операций с double – 4 flops)

$$\begin{aligned} &Split(a):\\ &t = (2^{27}+1)(*)a;\\ &a_{hi} = t(-)(t(-)a);\\ &a_{lo} = a(-)a_{hi};\\ &return\ (a_{hi}, a_{lo});\end{aligned}$$

2. Quick-Two-Sum(a,b): вычисляет $s = fl(a+b)$ и $e = err(a+b)$, учитывая, что $|a| \geq |b|$. (3 flops)

$Quick - Two - Sum(a,b)$:
$s = a(+)b;$
$e = b(-)(s(-)a);$
$return\ (s,e);$

3. Two-Sum (a, b): вычисляет $s = fl(a+b)$ и $e = err(a+b)$. (6 flops)

$Two\text{ - }Sum\ (a,\ b)$:
$s = a(+)b;$
$v = s(\text{-})a;$
$e = (a(\text{-})(s(\text{-})v))(+)(b(\text{-})v);$
$return\ (s,e);$

4. Three-Sum (a, b, c): вычисляет $s = fl(a+b)$, $e_1 = err_1(a+b)$ и $e_2 = err_2(a+b)$. (18 flops)

$Three - Sum\ (a,\ b,\ c)$:
$(t_1,t_2) = Two - Sum(a,b);$
$(s,t_3) = Two - Sum(c,t_1);$ Three-Sum (a, b, c):
$(e_1,e_2) = Two - Sum(t_2,t_3);$
$return\ (s,e_1,e_2);$

5. Two-Prod (a, b): вычисляет $p = fl(a*b)$ и $e = err(a*b)$. (17 flops)

$Two - \Pr od(a,b)$:
$p = a(*)b;$
$(a_{hi},a_{lo}) = Split(a);$
$(b_{hi},b_{lo}) = Split(b);$
$e = \left(\left(a_{hi}(*)b_{hi}(-)p\right)(+)a_{hi}(*)b_{lo}(+)a_{lo}(*)b_{hi}\right)(+)a_{lo}(*)b_{lo};$
$return(p,e);$

3. Операции в алгоритмах Quad-double и double-double

Данная библиотека предназначена для работы с числами определенной точности: 31 цифра точности (double-double) и 62 цифры точности(quad-double), реализована на C++.

3.1. Представление чисел

Double-double и quad-double представляются в памяти в виде массивов из 2 и 4 double слов 53 – битной точности соответственно. Пусть double-double массив $-\{a[n]\}$, quad-double массив – $\{b[n]\}$. Тогда числа высокой точности (A – double-double, B – quad-double) образуются следующим образом:

$$A = a[0] + a[1]$$
$$B = b[0] + b[1] + b[2] + b[3]$$

Мы рассмотрим алгоритмы основных арифметических операций для quad-double. Для double-double алгоритмы строятся аналогично.

3.2. Нормальная форма чисел

Поскольку любое число можно представить в виде суммы четырех чисел не единственным образом, то нужно потребовать какое-нибудь условие на слагаемые. Пусть

$$| a_{i+1} | \le \frac{1}{2} ulp(a_i), \qquad i = 0,1,2;$$

где $ulp(x)$ – это разница между x и ближайшим числом (double), большим чем x. Такое представление quad-double числа назовем нормализованным. Утверждается, что нормализованное представление числа единственно [2].

Алгоритм нормализации основан на методе нормализации Приста [7]. На вход подается 5 double чисел $(a_0, a_1, a_2, a_3, a_4)$, на выходе имеем нормализованное quad-double число.

Начиная с последнего double числа, поданного на вход, при помощи Quick-Two-Sum(a,b) определяем главную часть суммы и ошибку. Главная часть далее складывается с предыдущим числом, и определяются главная часть и ошибка этой суммы и так далее. Таким образом, мы имеем 5ти double слов $(t_0, t_1, t_2, t_3, t_4)$, которыми являются ошибки сложений, которые мы проделывали, а t_0 – главная часть последней суммы.

$$\sum_{i=0}^{4} a_i = \sum_{i=0}^{4} t_i$$

Но в результате нам надо только 4 слова (b_0, b_1, b_2, b_3) , таких, что

$$\sum_{i=0}^{4} a_i = \sum_{i=0}^{3} b_i$$

Поэтому проделываем похожую операцию, но в другом порядке, начиная с t_0. Более строго:

Renorm $(a_0, a_1, a_2, a_3, a_4)$

$(s, t_4) = Quick - Two - Sum(a_3, a_4)$

$(s, t_3) = Quick - Two - Sum(a_2, s)$

$(s, t_2) = Quick - Two - Sum(a_1, s)$

$(t_0, t_1) = Quick - Two - Sum(a_0, s)$

$s = t_0$

$k = 0$

Цикл for по $i = 1,2,3,4$

$(s, e) = Quick - Two - Sum(s, t_i)$

if $(e \neq 0)$

$b_k = s$

$s = e$

$k = k + 1$

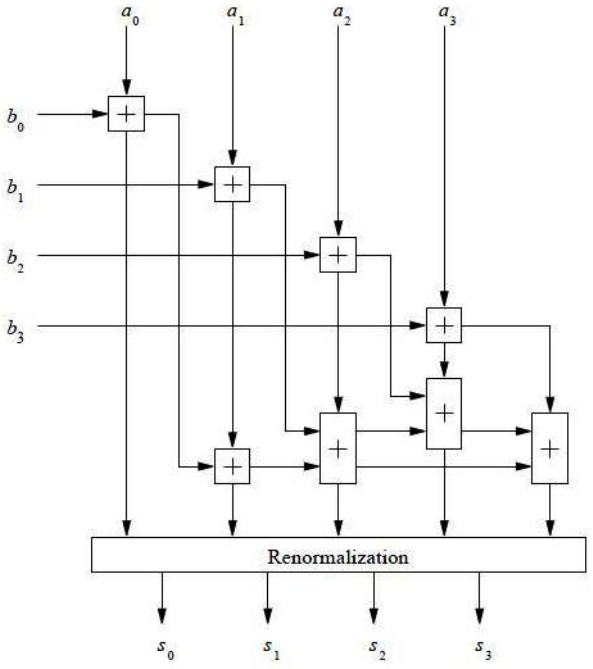
Конец if

Конец for

Return (b_0, b_1, b_2, b_3)

3.3. Сложение (вычитание)

Лучше всего алгоритм сложения двух quad-double чисел (a_0, a_1, a_2, a_3) и (b_0, b_1, b_2, b_3) иллюстрирует рисунок из [2]. Знак + в квадрате означает, что к входящим данным применяется функция Two-Sum(a, b). Знак + в прямоугольнике – что к входящим данным применяется функция Tree – Sum (a, b, c). Стрелка вниз – главная часть суммы, стрелки вправо – ошибки.



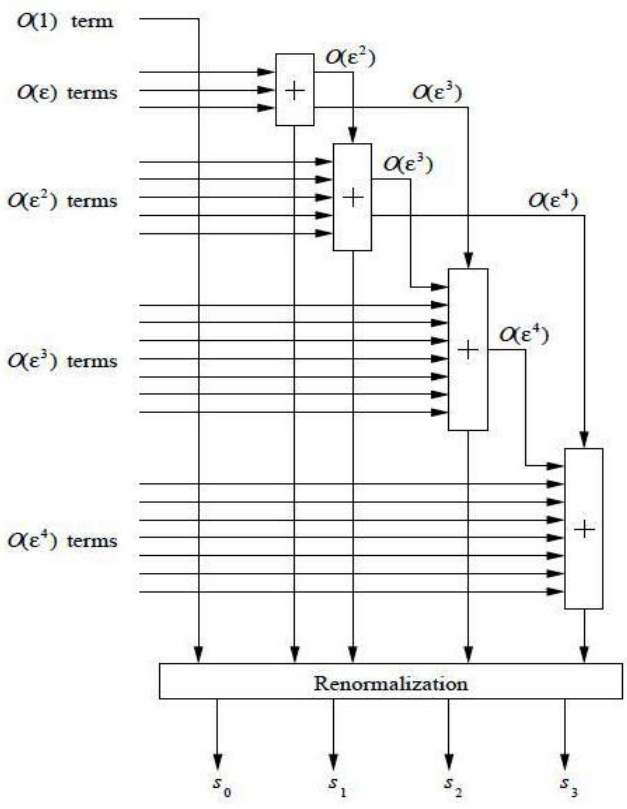
3.4. Умножение

Идея алгоритма умножения двух quad-double чисел $A=(a_0,a_1,a_2,a_3)$ и $B=(b_0,b_1,b_2,b_3)$ изображена на рисунке из [2]. Положим без ограничения общности, что наши числа порядка 1.

$$
\begin{aligned}
A * B \approx\ & a_0b_0 + \\
& a_0b_1 + a_1b_0 + \\
& a_0b_2 + a_1b_1 + a_2b_0 + \\
& a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + \\
& a_1b_3 + a_2b_2 + a_3b_1
\end{aligned}
$$

Слагаемые имеют порядок $O(1),O(\varepsilon),O(\varepsilon^2),O(\varepsilon^3),O(\varepsilon^4)$. Члены более высокого порядка нам не нужны для достижения точности в 212 бит. Для $i+j\le 3$ пусть $(p_{ij},q_{ij})=\text{Two-Prod}(a_i,b_j)$. Тогда $p_{ij}=O(\varepsilon^{i+j})$ и $q_{ij}=O(\varepsilon^{i+j+1})$. Мы имеем один член p_{00} порядка $O(1)$, три члена p_{00},p_{10},q_{00} порядка $O(\varepsilon)$, пять членов $p_{02},p_{11},p_{20},q_{01},,q_{10}$ порядка $O(\varepsilon^2)$, семь членов порядка $O(\varepsilon^3)$, и семь членов порядка $O(\varepsilon^4)$.

На рисунке встречаются 4 различных операции сложения: Tree-Sum, Six-Three-Sum (суммирует 6 double чисел и возвращает первые 3 компоненты суммы), Nine-Two-Sum (суммирует 9 double и возвращает 2 компоненты суммы), Nine-One-Sum (просто суммирует 9 double в нормальной арифметике).



3.5. Деление

Алгоритм деления $A=(a_0,a_1,a_2,a_3)$ и $B=(b_0,b_1,b_2,b_3)$ основан на алгоритме деления чисел произвольной точности. Сначала мы вычисляем отношение $q_0=a_0/b_0$. Можем вычислить остаток $R=A-q_0*B$, где $R=(r_0,r_1,r_2,r_3)$, и вычислить корректирующий член $q_1=r_0/b_0$. Мы будем продолжать этот процесс, пока не получим 5 членов (q_0,q_1,q_2,q_3,q_4). Затем, нормализовав их, получим верный результат деления.

3.6. Проверка эффективности реализации и некоторые выводы

Был проведен ряд тестов по исследованию эффективности алгоритмов библиотек DD и QD на процессоре с использованием одного ядра. Эти тесты заключаются в последовательном применении арифметической операции к числам определенного типа. Программа выполняется в режиме release без распараллеливания (одно ядро). Результаты приведены в таблице 3.6 – 1.

Таблица 3.6 – 1. Количество операций в миллисекундах (мс) для различных типов данных

Операция	double	double-double	quad-double
+	1.27e+006	78500	8760
–	1.27e+006	73600	8190
*	7.75e+005	14100	2230
/	6.30e+004	8160	810

Можно заметить, что операции выполняются достаточно быстро, и только для quad-double происходит значительное замедление, примерно в 8 – 10 раз по сравнению с double-double.

Проверим точность теоретических оценок. Предположим, что за 1 мс выполняется 10^6 операций с double, оценим теоретическое число операций с double-double и quad-double в соответствии с разобранными выше алгоритмами и сравним с практическими данными.

Рассчитаем число операций с double приходящихся на одну операцию с double-double и quad-double по данным полученным на практике:

$$\frac{\text{практ. число опер. с } double \text{ в мс}}{\text{практ. число опер. с } quad-double \text{ в мс}} = \text{Практ. число опер. с } double$$

Далее рассчитаем отношение теоретического и практического числа операций с double:

$$\text{Коэффициент} = \frac{\text{теоретич. число опер. с } double}{\text{практич. число опер. с } double}$$

Результаты приведены в таблице 3.6 – 2.

Таблица 3.6 – 2. Теоретическое и практическое число операций с double

Опер.	Double-double		Quad-double	
	Теор. число опер. с double	Коэф.	Теор. число опер. с double	Коэф.
+	11	0.9	95	0.8
–	11	0.8	95	0.8
*	24	0.3	216	0.5
/	37	0.3	667	0.5

Коэффициенты для сложения и умножения очень близки к единице. Это говорит о точности теоретических оценок и о том, что почти все время выполнения операций с double-double и quad-double идет на операции с double.

Посмотрим, во сколько раз медленнее выполняются операции с double-double и с quad-double, чем с double. Результаты в таблице 3.6 – 3.

Таблица 3.6 – 3. Отношение скоростей операций с double к скоростям соответствующих операций с double-double и quad-double

Опер.	Double-double	Quad-double
+	16.2	145.0
–	17.3	155.1
*	55.1	347.3
/	7.7	77.8

4. Реализация QD на CUDA

4.1. Представление чисел

Qd-double (Quda-double-double) представляется в памяти в виде специального типа double2, встроенного в CUDA, а qq-double (Quda-quad-double) – в виде структуры из двух double2. В процессе разработки программ, при переходе от CPU к GPU нужно обязательно преобразовать типы double-double и quad-double к qd-double и qq-double соответственно.

Все арифметические операции с числами реализованы точно так же, как и в случае программы на CPU, т.е. улучшение быстродействия программ заключается в оптимизации их алгоритмов, а не алгоритмов операций с типами данных в CUDA. Далее, для простоты типы qd-double и qq-double будут обозначаться по-прежнему, т.е. как double-double и quad-double.

5. Операции с векторами

5.1. Операции поэлементного сложения, вычитания, умножения и деления

В пакете gqd_linux_1_1 были реализованы алгоритмы поэлементного сложения, вычитания, умножения и деления двух векторов [8] на CUDA. Мы проанализировали этот алгоритм на предмет быстродействия для различных GPU. Обозначим количество элементов вектора через len. В программе используется грид из фиксированного числа блоков, состоящих из фиксированного числа нитей, а именно 128 блоков по 128 нитей. Получается, что если $len > 128*128$, то каждая нить выполняет несколько операций чтения из глобальной памяти и несколько арифметических операций. При аналогичном расчете на CPU используются возможности распараллеливания OpenMP на 4 ядра. Результаты выполнения теста в таблице 5.1 – 1:

Таблица 5.1 – 1. Число операций в миллисекунду с соответствующими типами данных

a) Tesla C1060

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	3.41e+006	1.78e+005	1.02e+006	4.44e+004	2.43e+005
*	3.59e+005	3.41e+006	1.31e+005	9.75e+005	2.51e+004	1.45e+005
/	2.87e+005	2.71e+006	7.60e+004	4.74e+005	7.52e+003	4.20e+004

b) GeForce GTX480

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	5.71e+006	1.78e+005	2.47e+006	4.44e+004	7.55e+005
*	3.59e+005	5.71e+006	1.31e+005	2.42e+006	2.51e+004	3.62e+005
/	2.87e+005	5.21e+006	7.60e+004	1.33e+006	7.52e+003	1.06e+005

c) Tesla C2050

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	3.54e+006	1.78e+005	1.54e+006	4.44e+004	6.75e+005
*	3.59e+005	3.53e+006	1.31e+005	1.52e+006	2.51e+004	4.95e+005
/	2.87e+005	3.50e+006	7.60e+004	1.67e+006	7.52e+003	2.24e+005

Таблица 5.1 – 2. Отношение скорости операций на GPU к скорости операций на CPU

Опер.	double			double-double			quad-double		
	C1060	GTX480	C2050	C1060	GTX480	C2050	C1060	GTX480	C2050
+	9.70	16.22	10.06	5.73	13.87	8.65	5.47	17.01	15.20
*	9.50	15.90	9.83	7.44	18.47	11.60	5.77	14.42	19.72
/	9.44	18.15	12.20	6.23	17.5	21.97	5.58	14.09	29.78

Видно, что, так как операции с double на GeForce GTX480 выполняются быстрее и эта карточка обладает самой быстрой памятью, то ее более высокая скорость означает, что в программе много операций связано с обращением к памяти. Аналогичная ситуация наблюдается и с double-double. И только в программе с quad-double можно заметить, что арифметические операции начинают играть некоторую роль, потому что выигрыш от использования Tesla C2050 становится сравним с выигрышем от GeForce GTX480.

Мы слегка изменили алгоритм, чтобы оптимизировать работу с памятью. Число нитей в блоке так и остается фиксированным и равным 128, а число блоков в гриде рассчитывается исходя из длины вектора, так чтобы каждая нить делала только по 2 операции с глобальной памятью (чтение из памяти и запись в нее) и одну арифметическую операцию. Результаты проведения такого теста:

Таблица 5.1 – 3. Число операций в миллисекунду с соответствующими типами данных. Когда число блоков – максимально

a) Tesla C1060

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	3.48e+006	1.78e+005	1.06e+006	4.44e+004	2.73e+005
*	3.59e+005	3.48e+006	1.31e+005	1.04e+006	2.51e+004	1.54e+005
/	2.87e+005	2.32e+006	7.60e+004	5.21e+005	7.52e+003	4.93e+004

b) GeForce GTX480

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	6.66e+006	1.78e+005	3.07e+006	4.44e+004	6.50e+005
*	3.59e+005	6.66e+006	1.31e+005	2.96e+006	2.51e+004	3.83e+005
/	2.87e+005	6.15e+006	7.60e+004	1.33e+006	7.52e+003	1.12e+005

c) Tesla C2050

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	4.44e+006	1.78e+005	2.05e+006	4.44e+004	7.70e+005
*	3.59e+005	4.44e+006	1.31e+005	2.10e+006	2.51e+004	5.88e+005
/	2.87e+005	4.21e+006	7.60e+004	1.70e+006	7.52e+003	2.54e+005

Таблица 5.1 – 4. Отношение скорости операций на GPU к скорости операций на CPU

Опер.	double			double-double			quad-double		
	C1060	GTX480	C2050	C1060	GTX480	C2050	C1060	GTX480	C2050
+	9.88	18.92	12.61	5.95	17.24	11.51	6.14	14.63	17.34
*	9.70	18.55	12.36	7.93	22.59	16.03	6.13	15.26	23.42
/	8.10	21.42	15.14	6.85	17.50	22.36	6.55	14.90	33.77

Получилась аналогичная ситуация: в операциях в double и double-double все еще большую роль играют операции с памятью, и только для quad-double видна роль вычислений. Но в сравнении с прошлым тестом производительность выполнения программы возросла в среднем на 12 процентов.

Для получения оценки максимальной производительности операций с числами высокой точности мы провели тесты, в которых время работы с памятью незначительно. Для этого была выбрана сравнительно небольшая длина массивов (16384 элемента), а вычисления проводились над каждым элементом массива большое число раз (порядка 10000). Полученные данные:

Таблица 5.1 – 5. Число операций в миллисекунду с соответствующими типами данных при малом обращении к памяти

a) Tesla C1060

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	2.91e+007	1.78e+005	2.50e+006	4.44e+004	3.41e+005
*	3.59e+005	2.91e+007	1.31e+005	1.70e+006	2.51e+004	2.34e+005
/	2.87e+005	1.96e+006	7.60e+004	8.13e+005	7.52e+003	4.58e+004

b) GeForce GTX480

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	7.80e+007	1.78e+005	6.10e+006	4.44e+004	8.14e+005
*	3.59e+005	7.80e+007	1.31e+005	4.16e+006	2.51e+004	5.56e+005
/	2.87e+005	1.20e+007	7.60e+004	2.24e+006	7.52e+003	1.12e+005

c) Tesla C2050

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
+	3.52e+005	6.76e+007	1.78e+005	1.27e+007	4.44e+004	2.01e+006
*	3.59e+005	6.83e+007	1.31e+005	8.14e+006	2.51e+004	1.07e+006
/	2.87e+005	1.15e+007	7.60e+004	3.60e+006	7.52e+003	2.60e+005

Таблица 5.1 – 6. Отношение числа операций на GPU к числу операций на CPU

Опер.	double			double-double			quad-double		
	C1060	GTX480	C2050	C1060	GTX480	C2050	C1060	GTX480	C2050
+	82.67	221.59	192.04	14.04	34.27	71.35	7.68	18.33	45.27
*	81.05	217.27	190.25	12.98	31.76	62.14	9.32	22.15	42.63
/	6.83	41.81	40.07	10.70	29.47	47.37	6.10	14.89	34.57

Как мы и ожидали, основную роль теперь играют вычисления, потому как наибольшая производительность достигается на карточке Tesla C2050 (для quad-double это особенно заметно).

5.2. Поэлементное применение других операций

В библиотеке QD помимо основных арифметических операций реализованы некоторые трансцендентные. Рассмотрим извлечение квадратного корня, *exp, log* и *sin*, следуя изложенному в [2].

5.2.1. Извлечение квадратного корня

Алгоритм извлечения квадратного корня из QD или DD числа A выполнен на основе итераций Ньютона для функции $f(x) = \frac{1}{x^2} - A$, которая имеет корни $\pm A^{-1/2}$. Это приводит к итерациям $x_{i+1} = x_i + x_i(1 - Ax_i^2)/2$. Заметим, что итерации не требуют операций деления чисел высокой точности, а умножение происходит покомпонентно. Так как итерации Ньютона имеют квадратичный порядок сходимости, то только 2 итерации требуется, если начать с double аппроксимации $x_0 = A_0^{-1/2}$ (в программе реализовано 3 итерации). После того как $x = \pm A^{-1/2}$ вычислено, выполняем умножение $\sqrt{A} = Ax$ и получаем нужную нам величину.

5.2.2. Экспонента

Для оценки e^x используется разложение в ряд Тейлора – Маклорена. Но, прежде чем проводить разложение, заметим, что $e^{kr+m\log 2} = 2^m(e^r)^k$, где целое число m выбирается из условия, что $m\log 2$ – ближайшее к e^x. Таким образом, мы можем получить $|kr| \le \frac{1}{2}\log 2 \approx 0.34657$. Используя $k = 256$, мы имеем

$|r| \le \frac{\log 2}{512} \approx 0.001354$. Поэтому вычисление e^r может быть эффективно выполнено при помощи разложения в ряд Тейлора. В разложении используем только первые 18 членов.

5.2.3. Логарифм

Так как ряд Тейлора для логарифма сходится медленнее, чем для экспоненты, то алгоритм нахождения логарифма числа A основан на итерациях Ньютона для функции $f(x) = e^x - A$. Итерации имеют вид: $x_{i+1} = x_i + A\exp(-x_i) - 1$. Требуется только 3 итерации.

5.2.4. Тригонометрические функции

Тригонометрические функции, такие как $\sin(\cdot)$ и $\cos(\cdot)$, вычисляются при помощи ряда Тейлора после уменьшения аргумента. Сначала аргумент уменьшается по модулю 2, так что $|x| \le \pi$. Далее заметим, что $\sin(y + k\pi/2)$ и $\cos(y + k\pi/2)$ являются формами записи для $\pm\sin y$ и $\pm\cos y$ для любых целых k, поэтому можно уменьшить аргумент по модулю $\pi/2$ так, что нужно будет вычислить только при $|y| \le \pi/4$. Окончательно, y представляется в виде $y = z + m\frac{\pi}{1024}$, где целое число m выбирается так, что $|z| \le \frac{\pi}{1024} \approx 0.001534$. Так как $|y| \le \pi/4$, то $|m| \le 256$. Используя заранее вычисленную таблицу , можно получить

$$\sin(z + \frac{m\pi}{1024}) = \sin z \cos\frac{m\pi}{1024} + \cos z \sin\frac{m\pi}{1024}$$

И аналогично для косинуса. В результате при разложении в ряд достаточно учитывать первые 10 членов.

5.2.5. Проверка эффективности реализации и некоторые выводы.

Для проверки эффективности алгоритмов трансцендентных операций мы использовали точно такие же тесты, как и для проверки арифметических операций. Далее представлены полученные результаты.

В таблице 5.2.5 – 1 результаты тестов усовершенствованного нами алгоритма поэлементного применения операции к векторам. А в таблице 5.2.5 – 3 мы вычисляем скорость выполнения операций, учитывая, что память играет маленькую роль в общем времени выполнения программы.

Таблица 5.2.5 – 1. Число операций в миллисекунду с соответствующими типами данных (с максимальным числом блоков)

a) Tesla C1060

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
$\sqrt{\cdot}$	9.74e+004	2.10e+006	3.56e+004	5.80e+005	1.78e+003	1.61e+004
exp(·)	4.21e+004	1.82e+006	3.93e+003	4.83e+004	4.97e+002	3.67e+003
log(·)	2.95e+004	1.08e+006	3.52e+003	4.40e+004	1.64e+002	1.12e+003
sin(·)	6.95e+004	9.30e+005	4.83e+003	2.64e+004	4.58e+002	3.19e+003

b) GeForce GTX480

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
$\sqrt{\cdot}$	9.74e+004	3.26e+006	3.56e+004	9.10e+005	1.78e+003	3.66e+004
exp(·)	4.21e+004	1.93e+006	3.93e+003	1.05e+005	4.97e+002	8.44e+003
log(·)	2.95e+004	1.18e+006	3.52e+003	9.45e+004	1.64e+002	2.67e+003
sin(·)	6.95e+004	1.67e+006	4.83e+003	5.86e+004	4.58e+002	8.66e+003

c) Tesla C2050

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
$\sqrt{\cdot}$	9.74e+004	4.68e+006	3.56e+004	2.34e+006	1.78e+003	8.84e+004
exp(·)	4.21e+004	4.67e+006	3.93e+003	2.52e+005	4.97e+002	2.16e+004
log(·)	2.95e+004	3.30e+006	3.52e+003	2.28e+005	1.64e+002	6.26e+003
sin(·)	6.95e+004	3.70e+006	4.83e+003	1.34e+005	4.58e+002	1.17e+004

Таблица 5.2.5 – 2. Отношение скорости операций на GPU к скорости операций на CPU

Опер.	double			double-double			quad-double		
	C1060	GTX480	C2050	C1060	GTX480	C2050	C1060	GTX480	C2050
$\sqrt{\cdot}$	2.16	33.47	48.05	16.29	25.56	65.73	9.04	20.56	49.66
exp(·)	43.23	45.84	110.93	12.29	26.72	64.12	7.38	16.98	43.46
log(·)	36.61	40.00	111.86	12.50	26.84	64.77	6.83	16.28	38.17
sin(·)	13.38	24.03	53.23	5.46	12.13	27.74	6.96	18.91	25.54

Можно сказать, что эти операции являются достаточно медленными и их использование быстрее идет на карточке Tesla C2050.

Таблица 5.2.5 – 3. Число операций в миллисекунду с соответствующими типами данных (На CPU и GPU). Вычислений существенно больше, чем обменов с памятью.

a) Tesla C1060

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
$\sqrt{\cdot}$	1.60e+005	5.06e+006	4.30e+004	1.69e+006	2.31e+003	3.95e+004
exp(·)	7.10e+004	4.80e+006	3.60e+003	1.19e+005	5.90e+002	3.70e+003
log(·)	3.40e+004	2.86e+006	3.81e+003	1.07e+005	1.99e+002	1.16e+003
sin(·)	1.07e+005	3.86e+006	5.43e+003	6.37e+004	5.52e+002	3.08e+003

b) GeForce GTX480

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
$\sqrt{\cdot}$	1.60e+005	7.70e+006	4.30e+004	1.60e+006	2.31e+003	3.69e+004
exp(·)	7.10e+004	4.66e+006	3.60e+003	1.10e+005	5.90e+002	8.44e+003
log(·)	3.40e+004	2.68e+006	3.81e+003	9.89e+004	1.99e+002	2.70e+003
sin(·)	1.07e+005	3.64e+006	5.43e+003	6.02e+004	5.52e+002	8.68e+003

c) Tesla C2050

Опер.	Double		double-double		quad-double	
	CPU	GPU	CPU	GPU	CPU	GPU
$\sqrt{\cdot}$	1.60e+005	4.99e+006	4.30e+004	3.15e+006	2.31e+003	8.88e+004
exp(·)	7.10e+004	7.28e+006	3.60e+003	2.62e+005	5.90e+002	2.14e+004
log(·)	3.40e+004	4.78e+006	3.81e+003	2.35e+005	1.99e+002	6.25e+003
sin(·)	1.07e+005	5.65e+006	5.43e+003	1.36e+005	5.52e+002	1.21e+004

Таблица 5.2.5 – 4. Отношение скорости операций на GPU к скорости операций на CPU

Опер.	double			double-double			quad-double		
	C1060	GTX480	C2050	C1060	GTX480	C2050	C1060	GTX480	C2050
$\sqrt{\cdot}$	31.63	48.13	31.19	39.30	37.21	73.25	16.95	16.09	38.44
exp(·)	67.60	65.63	102.53	33.06	30.56	72.78	6.17	14.01	36.27
log(·)	84.12	78.82	140.59	28.16	26.05	61.68	6.01	13.50	31.40
sin(·)	36.07	34.02	52.80	11.85	11.11	25.05	5.63	15.82	21.92

Если сравнить таблицы 5.2.5 – 1.c) и 5.2.5 – 3.c), то можно заметить, что отношение, указанное в них, для карты GeForce GTX480 уменьшилось, а для Tesla C2050 почти не изменилось. Это говорит о том, что даже в тесте с поэлементным применением операции, время, затраченное на вычисления, уже играет основную роль.

6. Операции с матрицами

В этой части будут рассмотрены такие операции с матрицами, как произведение матриц, и решение систем линейных уравнений, где в качестве правой части и вектора неизвестных выступают матрицы.

6.1. Произведение матриц

Для произведения матриц был взят стандартный алгоритм из SDK (http://developer.nvidia.com/gpu-computing-sdk [9]), в котором работа с глобальной памятью оптимизирована при помощи использования разделяемой памяти.

6.1.1. Алгоритм

Приведем кратко алгоритм для перемножения квадратных матриц A и B. Будем использовать блоки размера 8x8, поскольку для использования блоков большего размера не хватает объема разделяемой памяти для QD, и будем считать, что размер матриц N кратен 8. Каждый блок будет вычислять одну 8x8 подматрицу C' искомого произведения. Для вычисления подматрицы нам приходится постоянно обращаться к двум полосам (подматрицам) исходных матриц A' и B'. Обе эти полосы имеют размер Nx8, и их элементы многократно используются в расчётах. Идеальным вариантом было бы разместить копии этих полос в разделяемой памяти, однако для нашей задачи это неприемлемо из-за ее небольшого объема. Однако, если каждую из этих полос мы разобьём на квадратные подматрицы 8x8, то становится видно, что результирующая матрица C' просто является суммой попарных произведений подматриц из этих двух полос: $C' = A_1' \times B_1' + A_2' \times B_2' + \ldots + A_{N/8}' \times B_{N/8}'$. За счет этого можно выполнить вычисление подматрицы C' всего за $N/8$ шагов. На каждом таком шаге в разделяемую память загружаются одна 8x8 подматрица A и одна подматрица B, при этом каждая нить блока загружает ровно по одному элементу из каждой из этих подматриц, то есть каждая нить делает всего два обращения к глобальной памяти на один шаг. После этого считается произведение подматриц, загруженных в разделяемую память, и суммируется нужный элемент произведения, далее идет переход к следующей паре подматриц.

Таким образом, при вычислении произведения матриц нам на каждый элемент произведения C нужно выполнить всего $2\text{x}N/8$ чтений из глобальной памяти. Количество арифметических операций составляет $2\text{x}N - 1$.

Данный алгоритм легко обобщается на произведение прямоугольных матриц.

6.1.2. Результаты тестовых расчетов эффективности

Для сравнения производительности используем последовательную программу перемножения матриц на CPU. Данные расчётов по данному алгоритму приведены в таблице 6.2.2 – 1.

По полученным данным можно сказать, что алгоритм достаточно оптимален. Время выполнения на GeForce GTX480 в два раза меньше, чем на Tesla C1060, т.е. память играет значительную роль в перемножении матриц, но время выполнения на Tesla C2050 еще в два раза меньше чем на GeForce GTX480, т.е. сами вычисления также играют немалую роль.

Таблица 6.2.2 – 1. Перемножение матриц: A (1600x3200), B (1600x3200). Размер блоков: 8x8 (число операций ≈ $1.64*10^{10}$)

Устр.	C1060	GTX 480	C2050	CPU	C1060	GTX 480	C2050	CPU
Тип	Время выполнения одного умножения, s				Производительность = число опер./ms			
Double	0.34	0.20	0.26	95.68	6.20e+7	8.25e+07	6.17e+07	1.71e+05
d-double	8.34	3.62	1.75	557.78	1.97e+6	4.52e+06	9.34e+06	2.93e+04
q-double	74.63	31.37	14.22	1785.50	2.20e+5	5.22e+05	1.15e+06	9.18e+03

Таблица 6.2.2 – 2. Отношение скорости операций на GPU к скорости операций на CPU

	C1060	GTX 480	C2050
Double	362.57	482.46	360.82
d-double	67.23	154.26	318.77
q-double	23.96	56.86	125.27

Можно ли улучшить данный алгоритм применительно к нашим типам данных? Обращение к глобальной памяти coalesced для всех наших типов. При обращении к разделяемой памяти возникают конфликты банков высокого порядка, но для их полного устранения придется кардинально изменять сам алгоритм произведения матриц, частичное устранение конфликтов банков возможно, но так как обращение к разделяемой памяти быстрая операция, то значительного прироста производительности не наблюдается.

6.2. Решение системы линейных уравнений $AX = B$

Для решения системы линейных уравнений воспользуемся методом Гаусса. Образуем одну матрицу из матриц A и B, просто располагая одну рядом с другой, так как в памяти матрицы хранятся по строкам, то матрица AB будет храниться так же. Будем работать с ней как с одной матрицей.

6.2.1. Алгоритм

1. Проверяем, не равен ли нулю диагональный элемент подматрицы, соответствующей матрице A. Если равен, то ищем строку, в которой элемент в соответствующем столбце не ноль, запоминаем номера обоих строк (первая – в которой 0 на диагонали; вторая – у которой в соответствующем столбце не 0). Вместо операции запоминания номеров строк, можно их менять местами, но это более трудоемкая операция при действиях с памятью.
2. Делим найденную строку, у которой на месте диагонального элемента матрицы A не 0, на диагональный элемент. Эта операция выполняется на GPU при помощи одномерного грида и одномерных блоков (грид соответствует строке матрицы), каждая нить делит свой элемент на диагональный, разделяемая память не используется.
3. Линейными преобразованиями строк обнуляем все элементы над и под диагональным элементом. В этом случае можно использовать как двумерный грид и двумерные блоки (каждой нити соответствует свой элемент матрицы) так и двумерный грид из линейных блоков (соответствие нитей элементам матрицы аналогичное). При реализации в разделяемую память помещается строка, содержащая очередной диагональный элемент. Каждая нить выполняет над своим элементом операции деления и вычитания, в соответствии с методом Гаусса.

Далее, эти пункты повторяются для каждой строки матрицы. В результате мы получим единичную подматрицу на месте матрицы A, а на месте матрицы B будет решение нашей системы.

6.2.2. Результаты тестовых расчетов эффективности

Все расчеты были проведены для следующей задачи: $AX = B$, где A(2400x2400), B(1600x2400). Примерное число операций с double составляет $1.54*10^{10}$.

Результаты тестов для квадратных блоков 8x8 приведены в таблице 6.3.2–1.

Таблица 6.3.2 – 1. Решение систем линейных уравнений с использованием квадратных блоков

Устрой-ство	C1060	GTX 480	C2050	CPU	C1060	GTX 480	C2050	CPU
	Время выполнения одного решения				Производительность = число опер./ms			
Double	18.89	5.49	5.26	41.33	8.14e+5	2.80e+6	2.92e+6	3.72e+5
d-double	33.63	10.04	8.69	378.33	4.57e+5	1.53e+6	1.77e+6	4.07e+4
q-double	181.94	50.73	33.44	3383.58	8.46e+4	3.03e+5	4.60e+5	4.55e+3

Таблица 6.3.2 – 2. Отношение скорости операций на GPU к скорости операций на CPU

	C1060	GTX 480	C2050
Double	2,19	7,52	7,85
d-double	11,24	37,68	43,53
q-double	18,5	66,7	101,2

В качестве линейных блоков были использованы блоки длиной 128. Результаты в таблице 6.3.2 – 3.

Таблица 6.3.2 – 3. Решение систем линейных уравнений с использованием квадратных блоков

Устрой-ство	C1060	GTX 480	C2050	CPU	C1060	GTX 480	C2050	CPU
	Время выполнения одного решения				Производительность = число опер./ms			
Double	18.43	3.94	3.26	41.33	8.35e+5	3.90e+6	4.72e+6	3.72e+5
d-double	36.06	9.42	6.98	378.33	4.27e+5	1.63e+6	2.20e+6	4.07e+4
q-double	251.35	64.70	40.52	3383.58	6.12e+4	2.38e+5	3.79e+5	4.55e+3

Таблица 6.3.2 – 4. Отношение скорости операций на GPU к скорости операций на CPU

	C1060	GTX 480	C2050
Double	2.24	10.5	12.68
d-double	10.5	40.16	54.2
q-double	13.46	52.3	83.5

Можно заметить, что большой разницы в использовании квадратных и линейных блоков нет. Подобный тест был проведен на матрицах меньшего размера, в нем эта разница еще незначительней. Отсюда можно сделать вывод, что при увеличении размера матриц выгоднее использовать квадратные блоки. В целом, алгоритм является неплохим, поскольку основные затраты идут на вычисления, а не на работу с памятью.

7. Выводы

1) В работе исследована эффективность библиотеки QD арифметических и трансцендентных операций с числами, у которых 62 значащие цифры мантиссы, на CPU и GPU.
2) Были разработаны и реализованы алгоритмы на CUDA для основных операций линейной алгебры в DD и QD арифметике, а именно поэлементное применение операций к векторам и матрицам, произведение матриц и решение систем линейных уравнений. Имеющаяся библиотека QD была дополнена этими операциями.
3) Был проведен анализ эффективности этих алгоритмов на различных GPU: Tesla C1060, GeForce GTX480, Tesla C2050. Алгоритмы были оптимизированы с целью уменьшения нагрузки на работу с памятью. В результате было получено, что Tesla C2050 может ускорять вычисления с матрицами в арифметике высокой точности до 100 раз. Этот результат близок к предельному, поскольку почти такое же ускорение имеют и сами арифметические операции с quad-double. Таким образом, полученная библиотека работы с числами высокой точности на CUDA имеет почти оптимальную производительность для основных операций с матрицами.

Список литературы.

1. *D.E. Knuth,* The Art of Computer Programming, Vol. 2: Seminumerical Algorithm, second edition. Addison Wesley, Reading, Mass., 1981.
2. *Yozo Hida, Xiaoye S. Li, David H. Bailey,* Library for Double-Double and Quad-Double Arithmetic. – 2008.
3. Н. А. Зайцев, И. Л. Софронов, “Применение прозрачных граничных условий для решения двумерных задач упругости с азимутальной анизотропией”, Матем. моделирование, **19**:8 (2007), 49–54
4. I.L. Sofronov, N.A. Zaitsev, Numerical generation of transparent boundary conditions on the side surface of a vertical transverse isotropic layer, J. Comp. Appl. Math., **234** (2010) 1732-1738
5. *Кривов М., Казеннов А.,* GeForce или Tesla? // Суперкомпьютеры Top 50 – 2011, N 4(10).
6. *J.R. Shewchuk*, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. – 1997.
7. *Douglas M. Priest.* On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations. – 1992.
8. *Mian Lu, Bingsheng He, Qiong Luo.* Supporting Extended Precision on Graphics Processors – 2010.
9. GPU-computing SDK (http://developer.nvidia.com/gpu-computing-sdk)

Содержание