ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

УДК 004.02

Р.М. Дмитриенко[1], А.А. Емельянов[2], С.А. Золотов[2], В.Ю. Климашов[1], С.А. Савихин[1], А.Б. Терентьев[2]

ПРОБЛЕМА ТОЧНОСТИ ВЫЧИСЛЕНИЙ В МЕТОДЕ ВАРИАЦИИ ФАЗЫ

ФГНУ «Научно-исследовательский радиофизический институт»[1],
ООО «Научно-исследовательский центр специальных вычислительных технологий»[2]

Описаны методы повышения точности и уменьшения погрешности вычислений в применении к задаче поиска корней уравнения при помощи вариации фазы, также оценивается эффективность применения графических процессоров при использовании данного метода.

Ключевые слова: численные методы, вариация, логарифмический вычет, мантисса, точность.

Введение

В последнее время, в связи с постоянным ростом производительности компьютеров, все чаще поднимаются вопросы о достоверности и надежности компьютерных вычислений. Основной недостаток арифметики с плавающей точкой состоит в том, что фиксируется только относительная ошибка округления. Этот недостаток может привести к потере точности и получению заведомо ложных результатов, например, при суммировании рядов. В случаях, когда ошибки в исходных данных и ошибки округления оказывают сильное влияние на результат вычислений, например, при решении плохо обусловленных систем уравнений, целесообразно вести расчеты с повышенной точностью.

Постановка задачи поиска комплексного корня уравнения

Одним из показательных примеров задач, для которых требуется повышенная точность вычислений и минимизация погрешности, является численное решение уравнений. Рассмотрим задачу нахождения комплексных корней произвольного уравнения

$$f(z) = 0$$

с помощью метода вариации фазы.

Будем считать, что функция $f(z)$ является аналитической внутри некоторой замкнутой области S за исключением конечного числа изолированных особых точек и непрерывна вплоть до границы этой области. Из теории комплексного анализа известно, что разность между полным числом нулей N и полюсов P функции $f(z)$ в области S вычисляется с помощью логарифмического вычета:

$$N - P = \frac{1}{2\pi i}\oint_L \frac{f'(\zeta)}{f(\zeta)}d\zeta,$$

где интегрирование ведется в положительном направлении (против часовой стрелки) по за-

мкнутому спрямляемому контуру L, который ограничивает область S. С другой стороны, известно, что логарифмический вычет функции равен вариации аргумента данной функции, подсчитанной вдоль контура L, то есть

$$\oint_L \frac{f'(\zeta)}{f(\zeta)} d\zeta = i \operatorname{var} (\arg f(z))\Big|_L .$$

На практике применима именно вторая формула. Предполагаем, что нам известны полюса функции $f(z)$ с их кратностями, тогда по числу полных оборотов годографа исходного контура L вокруг начала координат можно найти число нулей внутри контура L. Основная проблема заключается как раз в определении числа полных оборотов, так как годограф в малой окрестности нуля может вести себя довольно сложным образом.

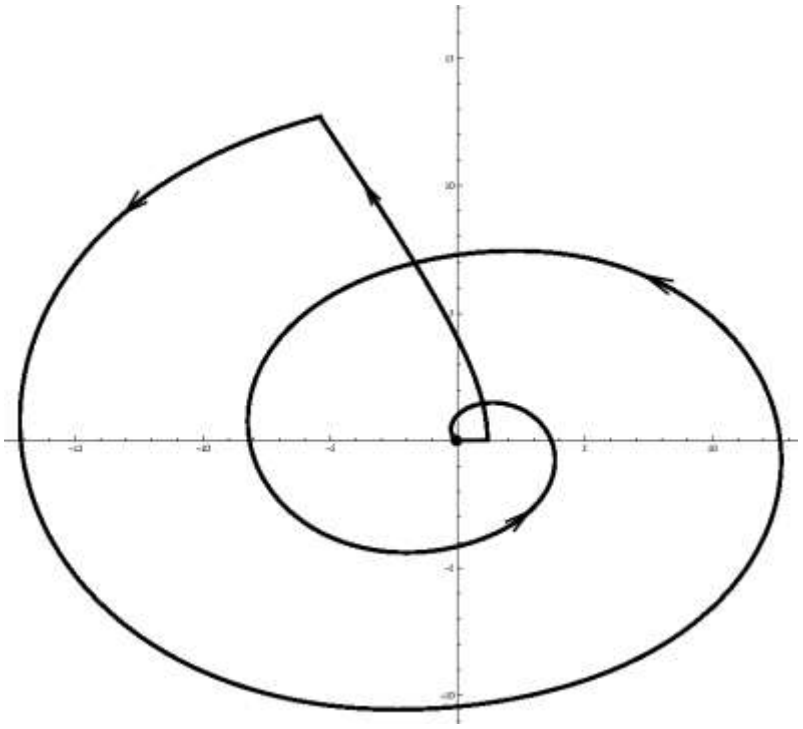

Рис. 1. Пример годографа прямоугольного контура

На рис. 1 показан пример кривой, в которую может перейти прямоугольный контур. По внешнему виду трудно подсчитать все ветки полученной кривой в окрестности нуля: изначально приближаясь к началу координат по спирали, она затем резко уходит в сторону.

Проблема детального исследования поведения годографа может быть решена увеличением шага разбиения исходного контура и повышением точности вычислений.

Авторами предложено два способа повышения точности и уменьшения погрешности, которые были применены к данной задаче. Эксперименты проводились с использованием графических процессоров, и далее будет представлено сравнение эффективности использования CPU и GPU. Но прежде чем переходить к описанию названных методов, целесообразно указать основы представления чисел с плавающей точкой.

Числа с плавающей точкой

В памяти компьютера числа с плавающей точкой представляются последовательным набором битов в следующем формате:

| знак | показатель степени | мантисса |

Под знак отводится один бит (0 — число положительное, 1 — число отрицательное).

Показатель степени и мантисса имеют фиксированный размер, поэтому диапазон представимых в данном формате чисел ограничен. Следовательно, существует наименьшее и наибольшее представимое число. Для формата double эти числа 10−308 и 10308 соответственно. Так как мантисса и показатель степени содержат только определенное количество значащих цифр, то, например, при сложении двух чисел, порядки которых сильно отличаются, можно получить неверный результат.

Рассмотрим пример. Пусть мантисса содержит три значащих цифры, а показатель степени – одну. Тогда при сложении двух чисел 1.23 * 103 и 3.45 * 10−2 получим

$$1.23 * 103 + 3.45 * 10{-}2 = 1230 + 0.0345 = 1230.0345 = 1.23 * 103,$$

что неверно. К настоящему моменту существует несколько библиотек для вычислений с повышенной точностью, например, GNU Multi-Precision Library (C++), ARPREC (C++/Fortran). Но все они не получили широкого распространения, так как время, затраченное на расчет с их помощью, в несколько раз превышает время расчета в формате double. Но сегодня этот недостаток можно компенсировать, если вести вычисления на графических процессорах, так как они существенно превосходят по производительности центральные процессоры, при большом объеме данных.

Переходим к описанию способов повышения точности и уменьшения погрешности. Первый способ заключается в эмуляции повышенной точности путем увеличения разрядности мантиссы и показателя степени. Смысл второго способа состоит в том, чтобы уменьшить погрешности вычислений стандартных форматов с плавающей точкой.

Эмуляция повышенной точности

Увеличим объем памяти, выделяемый под мантиссу, и показатель степени и переопределим основные операции арифметики: +, -, *, /. Это можно сделать следующим образом. Создадим структуру, которая будет содержать 32 бита под мантиссу, 32 бита под показатель степени и 1 бит под знак. По точности наш формат будет будет превосходить формат float на две десятичных значащих цифры. Для дальнейшего увеличения точности и определения мантиссы можно использовать массив чисел формата unsigned int.

```
struct myFloat
{
unsigned int mantissa;
 unsigned int exp;
 bool sign;
 myFloat operator+ ( myFloat &);
};
```

Далее определим операцию суммирования. Чтобы сложить два числа, сначала выравниваем показатели степеней. Для этого сдвинем мантиссу числа с меньшим показателем степени вправо на количество бит, равное разнице между показателями степени, после чего складываем мантиссы. При сложении мантисс может появится единица в старшем бите, поэтому надо проверить результат операции. Если в результате сложения получилось число меньше какого-либо слагаемого, то есть произошло переполнение, то необходимо сдвинуть мантиссу результата на один бит вправо и записать единицу в нулевой бит. Далее приведен код программы на C++, реализующий операцию сложения.

```
myFloat myFloat :: operator+ ( myFloat & ff2 )
{
 myFloat res;
 if ( exp >= ff2 . exp )
 {
 unsigned int diff = ff2.mantissa >> ( exp - ff2.exp );
 res.exp = exp;
 res.sign = sign;
```

```
 res.mantissa = mantissa + diff;
 if (res.mantissa < mantissa || res.mantissa < ff2.mantissa)
 {
 res.mantissa = res.mantissa >> 1;
 res.mantissa = res.mantissa | 0x80000000 ;
 res.exp++;
 }
 return res;
}
 else
 {
 unsigned int diff = mantissa >> ( ff2.exp - exp );
 res.exp = ff2.exp;
 res.sign = ff2.sign;
 res.mantissa = ff2.mantissa + diff;
 if (res.mantissa < mantissa || res.mantissa < ff2.mantissa)
 {
 res.mantissa = res.mantissa >> 1;
 res.mantissa = res.mantissa | 0x80000000;
 res.exp++;
 }
 return res;
 }
}
```

Теперь определим операцию умножения. При умножении двух чисел нашего формата необходимо перемножить мантиссы, сложить показатели степени и определить знак результата. Перемножение мантисс делается следующим образом. Если в мантиссе второго сомножителя на *i*-м месте стоит единица, то необходимо сдвинуть мантиссу первого множителя на *i* бит вправо и прибавить к результату. Знак результата операции будет положительным, если знаки сомножителей совпадают, и отрицательным, – если не совпадают.

```
myFloat myFloat :: operator* ( myFloat & ff2 )
{
 myFloat res;
 res.mantissa = mantissa;
 res.exp = 0;
 uint tmp = mantissa;
 for ( uint i = 0x40000000; i != 0; i >> 1)
 {
 tmp = tmp >> 1;
 if (( ff2.mantissa & i) == i)
 {
 uint oldmantissa = res.mantissa;
 res.mantissa = res.mantissa + tmp;
 if ( res.mantissa < oldmantissa )
 {
 res.mantissa = res.mantissa >> 1;
 res.matice = res.mantissa | 0x80000000;
 res.exp = res.exp + 1;
 tmp = tmp >> 1;
 }
 }
 }
```

```
 res.exp = res.exp + exp + ff2.exp;
 if ( sign == ff2.sign )
 res.sign = 0;
 else
 res.sign = 1;
}
```

Подобным образом можно определить операции вычитания и деления. При применении рассматриваемого метода к решению исходной задачи мы получили, что время работы алгоритма с применением графического процессора с двойной точностью сопоставимо со временем работы центрального процессора с типом данных float. Тем самым, использование тира double с применением графического процессора перестает быть затратным по времени.

Отсутствие большего выигрыша по скорости можно объяснить тем, что при вычислениях по данному методу относительно редко встречаются операции, которые значительно быстрее выполняются на GPU (например, перемножение элементов массива), а выигрыш от использования графического процессора при поэлементном сложении двух массивов незначителен.

Снижение погрешностей в вычислениях с плавающей точкой одинарной и двойной точности

Смысл этого метода состоит в том, чтобы складывать или вычитать числа, близкие по показателям степени. Рассмотрим пример. Пусть имеется число f формата float, равное 100.000.000. Будем прибавлять к нему миллион раз единицу. В результате получим тоже число, равное 100.000.000, что неверно, поскольку формат float вмещает только 7-8 значащих цифр. Рассмотрим подробнее операцию сложения. Для этого определим наш формат как структуру, состоящую из вектора чисел формата float.

```
struct newFloat
{
 std :: vector < float > ff;
 newFloat operator+ ( newFloat & );
};
```

Пусть есть два экземпляра структуры newFloat: ff1, ff2. Чтобы сложить их, будем переписывать числа из вектора ff2::ff в ff1::ff следующим образом: берем число из ff2::ff и ищем в векторе ff1::ff такое число, чтобы модуль разности показателей степени этих чисел был меньше определенного числа. Выбор этого числа зависит от формата чисел в векторе. Если в векторе ff1::ff, такое число нашлось, то мы их суммируем, если нет, то дописываем число из ff2::ff в конец ff1::ff. После того, как все необходимые сложения в программе выполнятся, мы получим вектор чисел. Далее, чтобы получить ответ, необходимо упорядочить этот вектор по возрастанию и просуммировать все его элементы начиная с нулевого. В результате всех этих операций мы значительно уменьшим количество сложений чисел с сильно различающимися показателями степени.

Далее приведем исходный код основных операций для операции сложения. Суммирование двух элементов формата myFloat:

```
newFloat & newFloat::operator+ ( newFloat & ff2 )
{
 for ( uint i = 0; i < ff2.size(); i++)
 {
 uint* bb = ( uint*)( void*)(& ff2 [i]);
 uint exp2 = (bb << 1) >> 16;
 for ( uint j = 0; j < this.size (); j++)
 {
 uint* aa = ( uint*)( void*)(& this [j]);
 uint exp1 = (aa << 1) >> 16;
```

```
 if ( abs(exp1 - exp2 ) < 8)
 {
 this [j] = this [j] + ff2 [i];
 }
 }
 }
}
```

Сортировка вектора и суммирование вектора чисел:

```
float answer ( newFloat & ff)
{
 for ( uint i = 0; i < ff.size(); i ++)
 {
 for ( uint j = i; j < ff.size(); j ++)
 {
 if (ff[i] > ff[j])
 {
 float tmp;
 tmp = ff[i];
 ff[i] = ff[j];
 ff[j] = tmp;
 }
 }
 }
 float answ = 0;
 for ( uint i = 0; i < ff.size(); i++)
 answ = answ + ff[i];
 return answ;
}
```

Вновь рассмотрим предыдущий пример. В результате первого сложения чисел 100.000.000 и 1, мы получим вектор <100.000.000, 1>. При следующем сложении вектор будет иметь вид <100.000.000, 2>. В конечном итоге, нам придется складывать числа, близкие по показателю степени, что даст верный результат.

Эксперименты по использованию данного метода при решении описанной в начале статьи задачи показали, что с применением графического процессора скорость выполнения указанного алгоритма увеличивается в два раза. Этот показатель ниже заявленного производителями графических карт. Объяснение подобной ситуации заключается в том, что данный метод использует сортировку и прочие операции с данными, которые могут вызывать множественные коллизии обращения к памяти.

Выводы

Рассмотренные способы реализации вычислений повышенной точности требуют много времени на выполнение расчетов. Но при больших объемах данных и высокой сложности вычислений данные способы можно реализовать на архитектуре графических процессоров, что позволит существенно сократить время выполнения вычислений. Выигрыш во времени вычислений достигается благодаря тому, что графические процессоры содержат сотни потоковых ядер, что позволяет запустить вычислительный алгоритм, если его можно распараллелить, в большом количестве потоков выполнения. Для сравнения, в настоящий момент максимальное число ядер для центрального процессора равняется 12 (AMD Opteron 6174), для графических процессоров это число составляет 1600 (ATI Radeon HD 5870).

Представленные методы были применены к вычислительной задаче поиска решения уравнения. Результаты показали, что использование графических процессоров в сочетании с правильной организацией данных и действий над ними дают неплохой результат. Пред-

ложенный подход к решению поставленной задачи позволил увеличить размер разрядной сетки, не уменьшая скорости, а также понизить погрешность в вычислениях с плавающей точкой.

1. **Литвинов, Г.Л.** Приближенная рациональная арифметика с контролируемыми ошибками округления / Г.Л. Литвинов, А.Я. Радионов, А.В. Чуркин. – М.: Вычислительные технологии, 2001. Т. 6. С. 87–94.

Дата поступления
в редакцию 11.10.2011

R.M. Dmitrienko, A.A. Emelyanov, S.A. Zolotov, V.U. Klimashov, S.A. Savikhin, A.B. Terentev

THE PROBLEM OF THE COMPUTING PRECISION OF THE PHASE VARIATION METHOD

This paper describes methods of precision improvement and computing error reduction as applied to the problem of phase variation-based equation roots solving. Estimations for the performance of GPU-based implementations of this method are given in the paper as well.

Key words: numerical computing, variation, logarithmic residue, mantissa, precision.