УДК 004 056 55

ВЛИЯНИЕ СВОЙСТВ СОВРЕМЕННЫХ ПРОЦЕССОРОВ НА ПРОИЗВОДИТЕЛЬНОСТЬ ПРОГРАММ

Е.Г. КАЧКО, Д.С. БАЛАГУРА, К.А. ПОГРЕБНЯК

Проводится анализ и сравнение способов оптимизации производительности программ. Приводятся теоретические и экспериментальные результаты исследований влияния на производительность конвейерной обработки, кеш памяти и распараллеливания вычислений. Даётся оценка эффективности использования разных технологий для повышения производительности и рекомендации по их применению.

Ключевые слова: конвейер, кеш-память, криптографические преобразования, параллельные вычисления,TBB, OpenMP.

ВВЕДЕНИЕ

На программы для криптографических преобразований очень часто накладываются жесткие ограничения на временные и пространственные характеристики. Эти требования связаны с массовостью использования этих преобразований и техническими ограничениями. В данной работе рассматриваются методы уменьшения временной сложности программ.

При оценке вычислительной сложности программ обычно учитывается количество операций, которые надо выполнить. Если алгоритм требует меньшее количество операций или одинаковое количество, но более простых операций, он считается более эффективным. Такой критерий оценки был справедлив для скалярных процессоров без конвейеров и других технических решений для увеличения быстродействия.

В работе рассмотрено влияние конвейера, кеша и параллелизма на временные характеристики программ. В конце показан учет приведенных факторов на примере оптимизации кода для умножения полиномов для алгоритма NTRU.

1. ИСПОЛЬЗОВАНИЕ КОНВЕЙЕРА

Для архитектуры X86 (i486) в 1989 году был добавлен простейший конвейер, содержащий 5 стадий: выборка команды, ее декодирование (определение кода и операндов), определение базы и смещения для адреса из операнда (только один операнд мог быть адресом памяти), выполнение инструкции, и, наконец, запись результата в регистр или память. Использование конвейера позволяет существенно уменьшить время выполнения последовательности команд. Действительно, пусть надо выполнить n команд. Пусть время выполнения каждой команды без конвейера равно 1. Тогда для выполнения этих команд потребуется T1 = n единиц времени. Пусть эта же последовательность команд выполняется с помощью конвейера, содержащего m блоков. В идеале каждый блок будет выполнять свою операцию за 1/m единиц времени (накладными расходами, связанными с использованием конвейера, пренебрегаем). Тогда время выполнения n команд равно T2 = 1 + n/m единиц времени. При большом n получаем ускорение приблизительно в m раз. Вот почему современные процессоры содержат не менее 10 блоков конвейера, а иногда и больше 20 [1]. Наращивание числа блоков конвейера ограничено только технологическими возможностями. Конвейер наиболее эффективный, если в программе нет инструкций перехода, которые нарушают естественный порядок выполнения команд. Инструкции перехода, нарушающие этот порядок, приводят к необходимости останова конвейера, связанного с необходимостью обработки другой последовательности команд, или встраиванию дублирующей ветви (так называемого спекулятивного выполнения), когда фактически выполняются команды для нескольких ветвей и «правильная» ветвь определяется после завершения выполнения команды перехода. Понятно, что оба решения приводят к потерям. Причем, эти потери тем существенней, чем больше команд перехода в программе. Заметим, что использование блоков прогнозирования переходов, которые являются составной частью всех современных процессоров, позволяют уменьшить эти потери только в случае успешного прогнозирования, которое гарантированно только в случае, если направление перехода не является случайным. Рассмотрим несколько примеров, которые позволят определить количественно потери, связанные с использование команд перехода.

Пример 1. Пусть необходимо определить количество отрицательных чисел последовательности. Очевидно, что для подсчета можно использовать код[1]

```
int count = 0;
for (int i = 0; i < n; ++i){
    if (src [i] < 0)
    count ++;
}
```

[1] Для задания кода используется синтаксис языка С. Процессор Intel (R) Core (TM) i5 – 4440 CPU @ 3.10 GHz

Определим количество необходимых операций для различных вариантов исходных данных (все числа не отрицательные, все числа отрицательные, отрицательные и неотрицательные числа равновероятны). При определении этого количества операции, связанные с организацией цикла, не учитываются, т. к. они необходимы для всех вариантов исходных данных.

1. Если все числа не отрицательные, то требуется n операций сравнения.

2. Если все числа отрицательные, то требуется n операций сравнения и . n операций увеличения на 1 счетчика чисел.

3. Если отрицательные и неотрицательные числа равновероятны, то требуется n операций сравнения и . n/2 операций увеличения на 1 счетчика чисел.

Таким образом, наибольшее количество операций требуется в случае, когда все числа отрицательные. Результаты вычислительного эксперимента приведены в табл. 1. При задании числа операций не учитываются команды сравнения, количество которых одинаково для всех вариантов.

Таблица 1

Определение количества отрицательных чисел (n = 1024 * 1024)

	1	2	3
Количество операций	0	n	n/2
Время выполнения (мс)	0.47	0.37	4.3

Как следует из табл. 1, ожидания не подтвердились, вариант, который ожидался как самый быстрый, только на втором месте (уступает второму варианту 27%). Вариант 3, который должен быть на втором месте, оказался последним и требует более чем в 11 раз больше времени, чем второй, по прогнозам, самый медленный вариант. Такой результат связан с потерями в конвейере, которые пропорциональны количеству ошибок прогнозирования. Этих ошибок совсем нет для 1 варианта, и максимальное количество для варианта 3. Для рассмотренного выше примера легко написать код без команды перехода, который основывается на внутреннем представлении положительных и отрицательных чисел.

```
int count = 0;
for (int i = 0; i < n; ++i)
    count -= src [i]>>31;
```

Заметим, что в данном случае нет команд перехода, а на каждой итерации надо выполнить две операции. Общее время выполнения в этом случае не зависит от типа исходных иданных и составляет 0.5 мс, что практически совпадает с первым вариантом.

Пример 2. Алгоритм LLL, который используется в одной из атак для криптоалгоритма NTRU[2], необходимо для заданного числа с плавающей точкой находить ближайшее целое (округлять это число). Для выполнения этой операции можно использовать код (для одной итерации):

```
dest [i] = (int)(src [i]+ (src [i]<0? -0.5 : 0.5));
```

Фактически в данном случае используется одна операция сравнения src [i]<0 и одна операция сложения. Если предположить, что вероятность отрицательного и неотрицательного чисел одинакова, то естественный порядок будет нарушаться в половине случаев, и прогнозирование здесь не поможет. Код без команды перехода (для одной итерации):

```
double RoundConst [] = {0.5, -0.5};
...
dest [i] = (int)(src [i] + RoundConst [*((unsigned*)&src[i] + 1)>>31]);
```

В последнем случае операции сравнения нет, но для каждой итерации используются операции вычисления адреса и разадресации, сдвига, определение элемента массива с заданным индексом, т.е. больше операций, чем в первом случае.

В табл. 2 приведены результаты для вычислительного эксперимента с использованием (Вариант 1) и без использования (Вариант 2) команды перехода

Таблица 2

Округление чисел (n = 1024 * 1024)

	Вариант 1	Вариант 2
Время (мс)	17	2.4

Таким образом, второй вариант позволяет получить ускорение в 7 раз, хотя требует больше операций для каждой итерации.

Примеры, приведенные выше, показывают, что наличие команд перехода существенно влияет на вычислительную сложность кода, необходимо минимизировать количество этих команд.

2. КЕШ-ПАМЯТЬ

Современные процессоры имеют частично ассоциативный кеш для хранения команд и данных. Частичная ассоциативность означает, что данные помещаются не в любое место кеша, а в место, которое определяется его адресом. Пусть размер внутреннего кеша n байтов (как правило, это 32 КБ или 64 КБ), элемент кеша содержит k байтов (k обычно равно 64), кеш m-ассоциативный, т.е. в строке кеша помещается m элементов по k байтов. Это означает, что если расстояние между адресами данных, которые используются последовательно, больше 64, для каждого данного потребуется новая операция загрузки его в кеш. Эти операции очень дорогие. А если расстояние между адресами данных, которые используются последовательно, кратно n/m, то эти данные используют только одну строку кеша. Таким образом, каждый m+1 элемент массива располагается в кеше вместо первого элемента, хотя весь остальной кеш может быть свободным.

Пример 3. В алгоритме LLL необходимо вычислять скалярное произведение двух векторов. В матрицах, которые используют этот метод, вектора являются векторами – столбцами. В языках высокого уровня обычно элементы матрицы располагаются по строкам.

Код для вычисления скалярного произведения:

```
double SMCol (double *b, double *bz, int k, int
l, int n){
      int n2 = n * n;
      double *pbk = b + k, *pbzl = bz + l, s = 0;
      for (int i = 0; i < n2; i+=n){
      s+=pbk [i ] * pbzl[i ];
      }
      return s;
   }
```

Здесь b, bz – матрицы, из которых используются векторы – столбцы (задаются как одномерные массивы размерностью n*n); k, l – номера столбцов для векторов.

В этом случае для каждого элемента вектора фактически выбирается 64 байта (8 чисел типа double), но используется только одно число. Так как n обычно большое (для NTRU минимальное значение n равно 802), то для формирования скалярного произведения будет 2n промахов кеша. Если исходные матрицы расположить так, что вектор будет строкой матрицы, то количество промахов кеша уменьшится в 8 раз. А так как промах кеша «стоит» значительно больше, чем вычисление s+=pbk [i] * pbzl[i]; то производительность в этом случае должна увеличиться больше, чем в 8 раз. Результаты вычислительного эксперимента приведены в табл. 3

Таблица 3

Скалярное умножение
(n = 1499 * 2[2], данные типа double)

	Вектор-столбец	Вектор-строка
Время (мс)	0.226	0.0033

Получаем ускорение более, чем в 68 раз.

3. ПАРАЛЛЕЛИЗМ

В современных процессорах реализован параллелизм на уровне команд, данных и потоков.

Параллелизм команд означает, что отдельные команды, если между ними нет зависимостей, выполняются параллельно. Это свойство процессора называется суперскалярностью, и обеспечивается самим процессором и компиляторами.

Параллелизм данных означает параллельную обработку блока данных. Обеспечивается за счет использования SIMD команд.

Параллелельное выполнение потоков означает одновременное выполнение потоков разными ядрами процессора, обеспечивается с помощью функций операционной системы для управления потоками, специальными средами для разработками паралелльных программ, например, Open MP или специальными библиотеками, например, TBB.

В настоящее время наиболее применимыми из класса SIMD команд являются SSE и AVX[3] SSE команды одновременно обрабатывают блок длиной 128 бит, компонентами блока могут быть числа с плавающей точкой с обычной и двойной точностью (типы данных __m128, __m128d соответственно) и целые числа длиной от 1 до 8 байтов (тип данных __m128i). AVX команды обрабатывают блок данных длиной 256 бит, в перспективе до 1024 бит, компонентами блока могут быть числа с плавающей точкой с обычной и двойной точностью (типы данных __m256, __m256d соответственно), целые данные длиной 1–8 байтов (тип данных __m256i)[1].

Пример 4. Вычисление скалярного произведения для векторов – строк. Так как количество элементов в блоке зависит от размера элемента, рассмотрим влияние типа данных на временные характеристики кода.

Код для вычисления скалярного произведения, не зависящий от типа:

```
template <typename T>
T SM (T *x, T *y, int n){
    T s = 0;
    for (int i = 0; i < n; ++i){
            s+=x[i] * y [i];
    }
    return s;
 }
```

Пример кода для вычисления скалярного произведения для данных типа float :

```
float SSE_SM_float (float *x, float *y, int n){
      __m128 *x128 = (__m128*)x, *y128 = (__
m128*)y, s128 = _mm_setzero_ps ();
      float *s = (float *)&s128;
      for (int i = 0; i < n/4; ++i)
            s128 = _mm_add_ps (_mm_mul_
ps(x128 [i], y128 [i]), s128);
      return s[0] + s [1] + s[2] + s [3];
}
```

Временные характеристики кода для различных типов данных приведены в табл. 4

Таблица 4

Скалярное умножение (n = 1024*1024).
Использование SSE операций

	short	int	float	double
Без SSE	0.6	0.85	3.4	1.6
С SSE	0.16	0.77	0.93	1.5

Таким образом, использование SSE эффективно для всех типов данных, эффективность тем больше, чем больше элементов данного типа помещается в блоке.

[2] Такое значение n выбрано с учетом максимальной размерности многочлена для NTRU

[3] Необходимо проверять возможность использования!

Наиболее простым способом перехода от последовательного к параллельному выполнению потоков является использование OpenMP[3]. Эта технология позволяет создавать один и тот же код для последовательного и параллельного выполнения, поддерживается большинством современных ОС общего назначения и языков программирования C, C++, FORTRAN.

Технология создания параллельных программ предусматривает предварительную разработку последовательного кода с максимальной эффективностью. Этот код используется в качестве меры при сравнении и в качестве источника для тестирования. Для вычислительного эксперимента будем использовать скалярное произведение без и с использованием SSE.

Пример кода для параллельного выполнения с SSE :

```
float OPENMP_SSE_SM_float_ (float *x, float *y, int n){
        __m128 S [SM_ThreadNum];
        #pragma omp parallel for
        for (int i = 0; i < SM_ThreadNum; ++i){
                int n4 = n/4, h = n4 /SM_ThreadNum;
        __m128 *px128 = (__m128 *)x, *py128 = (__m128 *)y, s;
                int begin = i * h, end = begin + h;
                if (i == SM_ThreadNum - 1) end = n4;
                s= _mm_setzero_ps();
                for (int j = begin; j < end; ++j)
                        s = _mm_add_ps (s, _mm_mul_ps (px128 [j], py128 [j]));
                S [i] = s;
        }
        for (int i = 1; i < SM_ThreadNum; ++i) S [0] = _mm_add_ps (S[0], S [i]);
        float *s = (float *)&S [0];
        return s[0] + s [1] + s [2] + s [3];
    }
```

Здесь SM_ThreadNum – количество потоков, которое определяется числом ядер процессора.

Временные характеристики параллельного кода для различных типов данных приведены в табл. 5.

Таблица 5

Скалярное умножение (n = 1024*1024).
Использование OpenMP. SM_ThreadNum] = 4

	short	Int	float	double
Лучший без OpenMP	0.16	0.77	0.93	1.5
Лучший с OpenMP	1.3	0.48	0.51	1.4

В данном случае небольшой эффект достигнут для всех типов данных кроме данных типа short. Незначительность эффекта объясняется тем, что время выполнения параллельной ветви небольшое. В этом случае на производительность существенно влияют накладные расходы, связанные с использованием потоков.

Далее рассмотрим учет рассмотренных выше факторов при разработке функции для умножения полиномов в алгоритме NTRU. Данная функция используется при генерации открытого ключа, шифровании и расшифровании. Операция умножения полиномов по сравнению с другими операциями, которые используются в алгоритме, имеет максимальную временную сложность.

Оптимизация алгоритма умножения полиномов для алгоритма NTRU. В алгоритме NTRU необходимо найти произведение полиномов, один из которых имеет коэффициенты 0, 1, -1 (вероятность каждого значения одинакова), а второй – коэффициенты в диапазоне [-1024, 1023]. Степени полиномов одинаковы. Результатом умножения полиномов является полином той же степени, его коэффициенты приводятся к интервалу [-1024, 1023].

Код для умножения полиномов (первый вариант)

```
void Mul0 (int *x, int *y, int *z, size_t n){
    size_t i, j;
    for (i = 0; i < n; ++i)     z [i] = 0;
    for ( i = 0; i < n; ++i)
            for (j = 0; j < n; ++j)     {
                    z [(i + j)%n] += x [i] * y [j];
                    z [(i + j)%n]%=2048;
            }
}
```

Для оптимизации кода необходимо:

1 Оптимизировать последовательный вариант за счет

1.1 Минимизации длины данных с учетом их значений (более эффективное использование Кеша)

1.2 Уменьшения числа операций вычисления модуля (взятие по модулю возможно после выполнения умножения, а не на каждом шаге)

1.3 Переход от операции умножения на операции сложения – вычитания (с учетом значений коэффициентов первого полинома 1, -1, 0). Минимизация числа команд перехода, связанных с анализом этих значений.

1.4 Уменьшение числа команд перехода за счет уменьшения числа итераций, связанных с использованием блоков

2 Использовать параллельные вычисления

2.1 Использование команд SSE для вычисления промежуточных сумм и модулей

2.2 Использование Open MP для вычисления промежуточных сумм.

В табл. 6 приведены временные характеристики функции умножения полиномов в результате предложенных оптимизаций.

Заметим, что более эффективное использование кеша (1.1) за счет использования более коротких данных не существенно улучшило время

выполнения. Это связано с тем, что в обоих случаях полином и все промежуточные данные в кеше помещаются. Уменьшение числа операций вычисления модуля (1.2) существенно улучшило эффективность кода. Колонки табл. 6 1.3 и 1.4 показывают эффективность соответствующих преобразований. Суммарное ускорение за счет преобразований 1.1–1.4 составляет более 34 раз.

Таблица 6

Умножение усеченных полиномов (n = 1499[4])

	Mul0	1.1	1.2	1.3	1.4	2.1	2.2
Время (мс)	25.1	22.6	1.6	0.83	0.73	0.15	0.0466

Переход на параллельные вычисления (пункты 2.1 и 2.2) приводят также к увеличению эффективности кода. Суммарное ускорение от параллельных вычислений ускорило операцию умножения более, чем в 15 раз.

Общее ускорение, достигнутое для операции умножения полиномов, составляет более, чем 500 раз.

ЗАКЛЮЧЕНИЕ

Исследование, проведенное в данной работе, показывает, что для современных процессоров число операций не всегда является достаточным критерием для оценки эффективности алгоритмов. Так, при реализации пунктов 1.3, 1.4 было увеличено число операций для достижения минимизации переходов. Параллельные вычисления с помощью SIMD операций безусловно эффективны, но они применимы только для специфических задач. Эффективность параллельных вычислений с помощью потоков зависит от соотношения полезной работы и накладных расходов. В любом случае при разработке современных приложений необходимо обязательно учитывать рассмотренные особенности современных процессоров

Литература

[1] http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[2] A Chosen-Ciphertext Attack against NTRU. http://www.iacr.org/archive/crypto2000/18800021/18800021.pdf

[3] OpenMP Application Program Interface. Version 4.0 - July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[4] Efficiency Improvement for NTRU. *Johannes Buchmann Martin Döring, Richard Lindner.* https://www.cdc.informatik.tu-darmstadt.de/reports/reports/ntrusves.pdf

Поступила в редколлегию 26.06.2014

[4] Максимальная степень полинома в соответствии со стандартом 1499.

Качко Елена Григорьевна, кандидат технических наук, профессор кафедры ПО ЭВМ ХНУРЭ. Научные интересы: программные средства криптографических систем.

Балагура Дмитрий Сергеевич, кандидат технических наук, доцент кафедры безопасности информационных технологий ХНУРЭ. Научные интересы: защита информации, криптографические протоколы выработки и согласования ключей.

Погребняк Константин Анатолиевич, доцент каф. БИТ ХНУРЭ, начальник отдела КЗИ АО «ИИТ». Научные интересы: применение методов алгебраической геометрии в криптологии, асимметричный криптоанализ.

УДК 004 056 55

Вплив властивостей сучасних процесорів на продуктивність програм / О. Г. Качко, Д.С. Балагура, К.А. Погребняк // Прикладна радіоелектроніка: наук.-техн. журнал. — 2014. — Том 13. — № 3. — С. 281–285.

Здійснюється аналіз та порівняння способів оптимізації продуктивності програм. Наводяться теоретичні та експериментальні результати дослідженнь впливу на продуктивність конвеєрної обробки, кеш-пам'яті та розпаралелювання обчислень. Надається оцінка ефективності використання різних технологій для підвищення продуктивності та рекомендації щодо їх застосування.

Ключові слова: конвеєр, кеш-пам'ять, криптографічні перетворення, паралельні обчислення TBB, OpenMP.

Табл.: 6. Бібліогр.: 4 найм.

UDC 004 056 55

Influence of the properties of modern processors on program performance / O.G. Kachko, D.S. Balagura, K.A. Pogrebnyak // Applied Radio Electronics: Sci. Journ. — 2014. — Vol. 13. — № 3. — P. 281–285.

An analysis and comparison of ways to optimize programs performance are carried out. Theoretical and experimental results of researches of the effect on the performance of pipelining, a cache memory, and parallel computing are provided. Assesses of the effectiveness of using different technologies to improve performance and recommendations for their use are given.

Keywords: pipeline, cache memory, cryptographic transformations, parallel computing, TBB, OpenMP.

Tab.: 6.: Ref.: 4 items.