© 2012 г. Б.Я. ШТЕЙНБЕРГ, д-р техн. наук
(Южный федеральный университет)

ЗАВИСИМОСТЬ ОПТИМАЛЬНОГО РАСПРЕДЕЛЕНИЯ ПЛОЩАДИ КРИСТАЛЛА ПРОЦЕССОРА МЕЖДУ ПАМЯТЬЮ И ВЫЧИСЛИТЕЛЬНЫМИ ЯДРАМИ ОТ АЛГОРИТМА

Работа поддержана ФЦП «Научные и научно-педагогические кадры инновационной России», Государственный контракт № 14.740.11.0006 от 1 сентября 2010.

Работа поддержана ОАО «Ангстрем»

Аннотация. В данной работе рассматривается проблема оптимизации площади на кристалле высокопроизводительных многоядерных процессоров. Суть проблемы в оптимальном разделении кристалла на память и на вычислительные устройства (ядра). Для различных вычислительных задач такое оптимальное разделение оказывается различным – в данной работе приводятся такие примеры. Из этого вытекает, что для эффективного решения различных классов задач требуются различные типы процессоров.

ABOUT THE CHIP DISTRIBUTION BETWEEN MEMORY AND COMPUTATION CORES. Boris J. Steinberg (Southern Federal University, Milchakova st. 8a, Rostov-on-Don, 344090, Russia, e-mail: borsteinb@mail.ru). The chip square optimum division between processor cores and memory is discussed in this paper. There are many examples illustrated different optimal chip divisions between processor cores and memory for different calculated tasks in this paper. Different classes' tasks need in different processor chips for effectively solving – it is follow from these examples.

1. Введение

Площадь на кристалле процессора распределяется между памятью и вычислительными ядрами. И увеличение объема кэш памяти и увеличение количества вычислительных ядер могут повышать быстродействие процессора. Чем меньшую площадь кристалла будут занимать вычислительные ядра, тем больше может быть кэш память. Уже появляются многоядерные процессоры с количеством ядер порядка 100 [4].

В работе рассмотрены несколько типов алгоритмов:

- алгоритмы, для которых увеличение кэш памяти и количества ядер не приводят к увеличению быстродействия,
- алгоритмы, для которых существенно увеличение кэш памяти,
- алгоритмы, для которых существенно увеличение количества ядер,

- алгоритмы, быстродействие которых зависит от баланса между объемом кэш памяти и количеством вычислительных ядер.

Приводимые примеры говорят о еще одной характеристике сложности алгоритмов – о их разбиваемости. Помимо классической сложности алгоритма, которая описывается количеством операций, как функцией количества входных данных, в научной литературе используются сложности алгоритмов по обращениям к памяти (см., например, [8]) и сложность межпроцессорных пересылок (см., например, [3]). Большой спектр характеристик вычислительных систем, влияющих на время работы программы, описан в [6].

В последнем параграфе обсуждается влияние размещений данных на эффективность отображения алгоритма на процессор со сложной архитектурой.

2. Соотношение объема кэш памяти и количества вычислительных ядер.

Площадь на кристалле S разбивается на 2 части: площадь M, отведенная под память, и площадь P, отведенная под множество вычислительных устройств. Память M условно будем называть кэш памятью, хотя во многих современных вычислительных архитектурах память на одном кристалле с процессором может быть адресуемой. Приводимые ниже рассуждения распространяются и на кристаллы с размещенной на них адресуемой памятью. В качестве вычислительных устройств на кристалле рассматриваются процессорные ядра, но все рассуждения справедливы для тех случаев, в которых эти устройства могут быть и процессорными элементами (иметь свою локальную адресуемую память) или простыми устройствами типа сумматор или умножитель (например, в конвейерных ускорителях).

Будем рассматривать вопрос об оптимальном соотношении M и P, при условии M+P=S.

Пусть имеется p процессоров, каждый из которых занимает площадь P_0 . Тогда $P = p * P_0$.

Пусть на вычислительной системе требуется вычислять алгоритм с количеством данных N, которых значительно больше, чем может вместить память, размещенная на кристалле с площадью M. Будем считать, что этот алгоритм сводится к алгоритмам обработки его частей, которые используют множество данных, занимающих площадь не более M . Требуется разбить площадь кристалла на части M и P так, чтобы время T работы алгоритма достигало минимального значения, при условии

$M+ p * P_0=S.$

Это условие в различных случаях может быть использовано в разных видах, в зависимости от того, что принимается в качестве параметра, память или количество вычислительных устройств:

$M = S – p * P_0$ или $p = (M – S)/ P_0$.

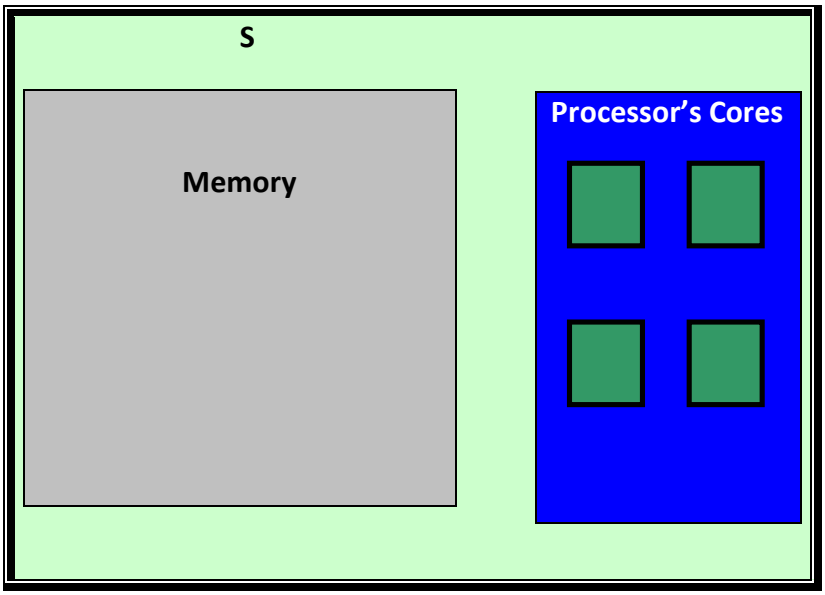

Рис. 1. Площадь на кристалле делится две части: часть, отводимая под память и часть, отводимая под вычислительные ядра.

3. Алгоритмы, для которых увеличение кэш памяти и количества вычислительных ядер не дают ускорения

Пример 1. Скалярное произведение векторов.

For I = 1 to N do

X(i) = X(i)+A(i)*B(i)

Вычисление содержит 2*N чтений из памяти и 2*N арифметических операций (сложений и умножений). Время счета можно оценить следующей формулой

$T = (N/M) * (C_1 * M + C_2 * M/p) = N * (C_1 + C_2/p)$

M – объем кэш памяти

n = M/2 длина частей векторов, которые размещаются в кэш памяти.

$C_2 * M/p = C_2 * 2*n/p$ время выполнения скалярного произведения частей векторов в кэш памяти на p процессорных ядрах.

N/M – количество частей, на которые делятся данные для закачивания в кэш память.

$C_1 * M$ – время закачивания частей векторов из оперативной памяти в кэш.

Увеличение количества процессорных ядер ведет к увеличению быстродействия, но незначительному, поскольку в современных процессорах константа C_1 значительно больше C_2 (более чем в 10 раз). Объем кэш памяти должен быть достаточен для загрузки процессорных ядер, а большее увеличение объема кэш-памяти бесполезно. Этим объясняется то, что формула времени не зависит от переменной M.

Конвейерное вычисление (одновременное выполнение операций чтения данных и арифметических операций).

Рассмотрим модель вычислений, при которой, как только в кэш память попадают первые части векторов, они сразу же перемножаются, а в это же время закачиваются следующие части векторов. В этом случае, поскольку на современных процессорах $C_1 >> C_2$, время работы равно

$T = N * C_1$

Время выполнения программы не зависит от объема кэш памяти, и лучшее быстродействие достигается на одном процессоре.

Пример 1а. Стандартное перемножение квадратных матриц размера n.

For i = 1 to N do

For j = 1 to N do

For k = 1 to N do

C(i,j) = C(i,j)+A(i,k)*B(k,j)

Стандартный алгоритм сводится к n^2 алгоритмам вычисления скалярного произведения векторов длины n. Если L – длина кэш линейки, то обращений к памяти порядка n^3 / L.

Время выполнения программы не зависит от объема кэш памяти и от количества процессоров.

В данном примере в кэш память попадают строка (или часть строки) матрицы A и столбец (часть столбца) матрицы B – 2*n чисел. С ними выполняется n умножений и n-1 сложений – всего тоже 2*n операций. Следовательно, если выполнение арифметической операции в 15 раз быстрее считывания данных из оперативной памяти в кэш память, то процессор загружен на 1/15 часть своей мощности.

Сложность (по количеству арифметических операций) всего алгоритма равна $O(n^3)$, где n – размерность матрицы, или, $O((N/2)^{3/2})$, где N – количество входных данных ($N = n^2$ – количество элементов двух матриц размера n^2). А сложность части алгоритма, которая выполняется в кэш памяти, равна O(N).

Таким образом, исходный алгоритм неэффективно разрезан на части.

4. Алгоритмы, увеличение быстродействия которых зависит от объема кэш памяти и не зависит от количества процессоров

Пример 2. Сортировка.

Имеет смысл использовать следующий алгоритм. Сортируемый массив нарезается на части, которые могут поместиться в кэш памяти. Для сортировки этих частей используется алгоритм, требующий мало дополнительной памяти (квиксорт), чтобы не выходить за пределы кэш памяти. Затем, отсортированные части массива сливаются (сортировка слиянием).

$T = (n/M) * (C_1 * M + C_2 * M*\log(M)/p) + C_3 *(n/M)*\log(n/M) = n * (C_1 + C_2 * \log(M)/p) + C_3 *(n/M)*\log(n/M) =$

$= n * (C_1 + C_2 * P_0 *\log M / (S – M)) + C_3 *(n/M)*\log(n/M)$

M – часть исходного массива, которая сортируется в кэш памяти.

(n/M) – количество частей исходного массива, которые сортируются в кэш памяти.

C_1 *M – время закачки части массива объема M в кэш память.

C_2 * M*log(M)/p – время сортировки порции (части) данных в кэш памяти.

C_3 * (n/M)*log(n/M) – время слияния частей.

Получить формулу оптимального соотношения величины объема M кэш памяти и количества процессоров затруднительно. Такое оптимальное соотношение легко вычислить на компьютере перебором всех возможных значений количества процессорных ядер p. При больших n второе слагаемое становится существенно больше первого и определяет время работы всего алгоритма. Второе слагаемое убывает с ростом M . Поэтому, для задачи сортировки больших массивов, чем больше кэш память, тем быстрее эта задача выполняется и следует на кристалле иметь только одно процессорное ядро.

5. Алгоритмы, увеличение быстродействия которых зависит от количества процессорных ядер и не зависит от объема кэш памяти.

Пример 3. Вычисления суммы ряда.

for i = 1 to n do

$S = S + x^i/i$

В каждом ядре с номером k = 0, …, (p-1) вычисляется часть ряда

for i = k*n/p +1 to (k+1)*n/p do

$S = S + x^i/i$

Затем вычисляется сумма результатов этих вычислений.

Данный алгоритм может обходиться регистровой памятью и в кэш памяти не нуждается. Увеличение количества процессоров приводит к увеличению производительности.

Следует отметить, что рассматриваемый параллельный алгоритм изменяет результат погрешности вычислений и предполагается, что такое изменение точности устраивает пользователя.

6. Алгоритмы, предполагающие баланс между кэш памятью и количеством процессорных ядер

Пример 4. Блочное перемножение квадратных матриц.

```
For i1 = 1 to N/m do
For j1 = 1 to N/m do
For k1 = 1 to N/m do
d1 = i1*(m-1)
d2 = j1*(m-1)
d3 = k1*(m-1)
For i2 = 1 to m do
For j2 = 1 to m do
For k2 = 1 to m do
i = d1+i2
j = d2+j2
k = d3+k2
C(i,j) = C(i,j)+A(i,k)*B(k,j)
```

Блочное перемножение квадратных матриц размера n сводится к перемножению $(n/m)^3$ блоков m*m (и соответствующих сложений блоков).

Считывание блока из памяти требует m^2/L чтений кэш линеек. Всего обращений к памяти $(n/m)^3 * (m^2/L) = n^3 / (m*L)$

Время выполнения алгоритма

$T = (n/m)^3 * (C_1 * M + C_2 * m^3/p) = n^3 * (C_1 * M/m^3 + C_2 /p) = n^3 * (C_1 * (3*r)^{3/2}/M^{1/2} + C_2 /p) = n^3 * (C_1 * (3*r)^{3/2} /M^{1/2} + C_2 /p) = n^3 * (C_1 * (3*r)^{3/2} /(S – p * P_0)^{1/2} + C_2 /p)$

$M = r*3*m^2$; $m = (M/(3*r))^{1/2}$ – соотношение между размером блоков и размером кэш памяти.

r – коэффициент, зависящий от того, какую площадь на кристалле занимает один бит памяти.

$C_1 * M$ – время пересылки порции данных в кэш память и обратно.

$C_2 * m^3/p$ – время обработки данных в кэш памяти.

n/M – количество порций данных, передаваемых в кэш память.

$(n/m)^2$ – количество блоков, на которое разбита каждая матрица.

$(n/m)^3$ – количество перемножений блоков

Для нахождения оптимального количества процессорных ядер приравняем к нулю производную функцию времени по количеству процессорных ядер

$T' = n^3 * (1/2 * P_0 * C_1 * (3*r)^{3/2} /(S – p * P_0)^{3/2} – C_2 /p^2) = 0$

и придем к уравнению

$C_0 * p^4 = (S/P_0 – p)^3$

Здесь $C_0 = (1/2 * (3*r)^{3/2} * C_1 / C_2)$

Из полученного уравнения можно найти оптимальное значение количества вычислительных ядер p. Решать уравнение можно перебором (компьютерным) всех возможных значений количества вычислительных ядер.

Если полагать отношение времени чтения данных ко времени обработки (для современных процессоров) примерно равным $C_1/C_2 = 20$, то получим значение константы $C_0 = 50$. Полагая $S/P_0 = 1000$, получим, что оптимальное количество процессорных ядер p = 64 близко к оптимальному.

7. Использование кэш памяти и топологии соединения процессорных ядер

Размещение данных алгоритма для оптимизации использования кэш памяти описано, например, в [2], [5]. Одновременное разбиение операций и данных алгоритма на части для оптимизации использования кэш памяти описано, например, в [1], [10]. В данной работе приведены примеры, иллюстрирующие границы возможностей методов [1], [10]. В качестве описания размещений данных, которые подстраиваются и под кэш, и под распределенную память, можно использовать блочно-аффинные размещения данных [7], [9].

На сайте [11] приводятся первые примеры программ, использующие такие блочно-аффинные размещения данных, которые удобны и для автоматизации размещения данных в распределенной памяти при автоматическом распараллеливании и могут давать до 40% ускорения при использовании кэш-памяти (программы написаны программистами Е. Кравченко и М. Юрушкиным с использованием системы ДВОР www.ops.rsu.ru).

Для размещения данных в иерархической распределенной памяти предполагается использовать рекурсивные описания блочно-аффинных размещений,

СПИСОК ЛИТЕРАТУРЫ

1. *Арыков С.Б., Малышкин В.Э.* Система асинхронного параллельного программирования «Аспект»// Вычислительные методы и программирование. 2008. Т. 9, №1, с. 205-209.

2. *Касперский К.* Техника оптимизации программ. Эффективное использование памяти. - СПб.: БХВ-Петербург, 2003 г. – 456 с.

3. *Климов А.В.* Умножение плотных матриц на неоднородных высокопараллельных вычислительных системах (анализ коммуникационной нагрузки). «Информационные технологии», №3, 2008, с.24-31.

4. *Корнеев В.В.* Проблемы программирования суперкомпьютеров на базе многоядерных мультитредовых кристаллов. Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность: Труды Всероссийской суперкомпьютерной конференции (21-26 сентября 2009 г., г. Новороссийск).– М.: Изд-во МГУ, 2009.

5. *Лиходед Н.А.* Распределение операций и массивов данных между процессорами.// Программирование, 2003, №3, с. 73-80.

6. *Шон Пипер, Джоан Пол, Майкл Скот.* Новая эра в оценке производительности компьютерных систем//«Открытые системы», 2007, № 9, с. 52-58.

7. *Штейнберг Б.Я.* Оптимизация размещения данных в параллельной памяти. Ростов-на-Дону, изд-во Южного федерального университета, 2010, 255 с.

8. *Штейнберг Б. Я.* Блочно рекурсивное параллельное перемножение матриц. Известия ВУЗов. Приборостроение, т. 52, №10, 2009 г. , с. 33-41.

9. *Штейнберг Б.Я.* Блочно-аффинные размещения данных в параллельной памяти. Информационные технологии. М.: из-во «Новые технологии», 2010, №6, с. 36-41.

10. *Lim Amy W., Lam Monika S*. Cache Optimizing with Affine Partitioning. 14 p.

11. www.ops.rsu.ru