Лекция №2
Приемы оптимизации последовательных программ

С учетом особенностей архитектуры современных ЭВМ

Особенности современных архитектур

- CISC и RISC

Время на решение задачи t = C*T*I

C – число тактов процессора на инструкцию

T = 1/F – время такта, F – тактовая частота

I – число инструкций на задачу

CISC: уменьшаем фактор I, проблемы с C и T

RISC: уменьшаем C и T, проблемы с I

Что сейчас? Внутри RISC, снаружи CISC

CISC и RISC

- Что мы можем сделать?
- Использовать оптимизированные библиотеки и компиляторы
- Минимизировать количество “дорогих” инструкций

Пример: a/(b*c) лучше чем a/b/c

Конвейеризация

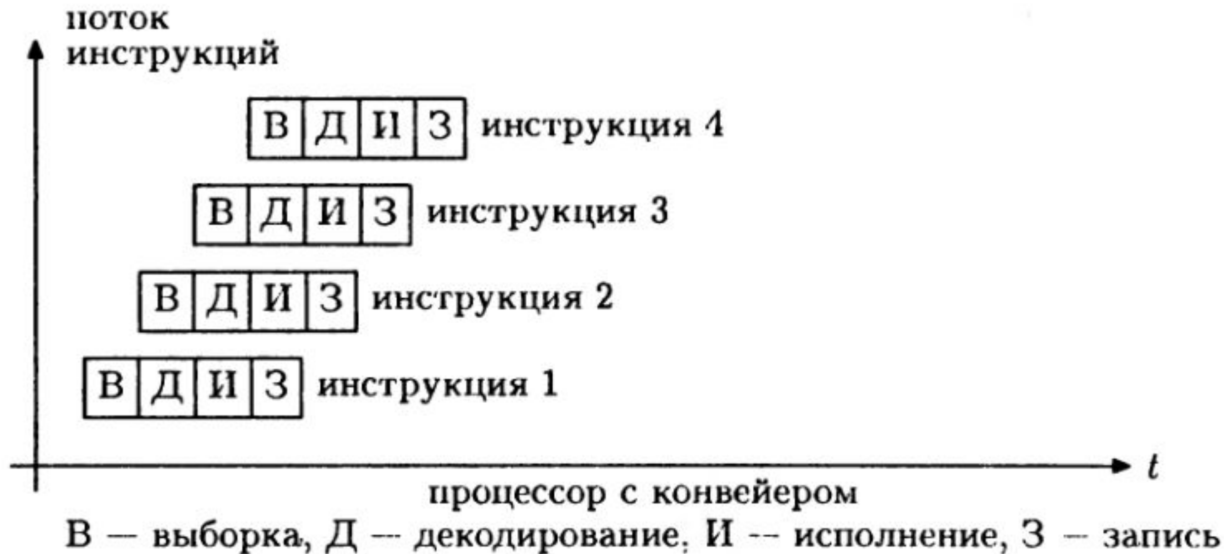
- Генри Форд был прав …

Разбиваем операцию на стадии

Например – исполнение инструкции состоит из Выборки, Декодирования, Исполнения, Записи результатов

При правильной реализации n-стадийный конвейер может одновременно исполнять n последовательных инструкций

Конвейеризация



Конвейеризация

- В идеале n-стадийный конвейер дает прирост производительности в n раз
- На практике: переходы, исключения, прерывания, задержки подсистемы памяти, взаимозависимые инструкции могут разрушить поток инструкций и остановить конвейер
- Пример: “кукурузные” гигагерцы

Конвейеризация

- Что мы можем сделать?

А) Положиться на компилятор

Б) Развернуть циклы

В) Выполнить потактовую ассемблерную оптимизацию

Развертка циклов (loop unroll)

Скалярное произведение векторов

```
double dot_prod(int n, double *a, double * b);
```

Стандартная реализация

```
double s=0.0; for (int i=0; i<n; i++) s += a[i] * b[i]; return s;
```

Развертка в 2 раза

```
double s1=0.0, s2=0.0; int n2 = (n/2) * 2;
for (int i=0; i<n2; i+=2){ s1 += a[i] * b[i]; s2 += a[i+1] * b[i+1];}
for (int i=n2; i<n; i++) s1 += a[i] * b[i];
return s1+s2;
```

Развертка цикла в 8 раз

```
double s1=0.0, s2=0.0, s3=0.0, s4=0.0;
int n8 = (n/8) * 8;
for (int i=0; i<n8; i+=8){
        s1 += a[i] * b[i] + a[i+1] * b[i+1];
        s2 += a[i+2] * b[i+2] + a[i+3] * b[i+3];
        s3 += a[i+4] * b[i+4] + a[i+5] * b[i+5];
        s4 += a[i+6] * b[i+6] + a[i+7] * b[i+7];
}
for (i=n8; i<n; i++) s4 += a[i] * b[i];
return (s4+s3)+(s2+s1);
```

Бонус: внутренний параллелизм

Вычисления внутри цикла могут быть выполнены одновременно на суперскалярном процессоре

Кэш-память

Cache != Cash ;)

Факт: пропускной способности памяти не хватает процессору (докажите!)

Выход: кэш память – быстрое статическое ОЗУ, вставленное между процессором и системной памятью, предназначенное для сохранения последних использованных инструкций и данных

Cache hit – хорошо, cache miss – плохо.

Кэш-память

- Единый vs. разделяемый (гарвардский) кэш
- Уровни: TLB, L1D, L1I, L2, L3 …
- Кэш с прямой записью (write-through) vs. кэш с обратной записью (write-back)

Характеристики, важные для программиста: длина строки, обьем кэш-памяти

Кэш-память

Как можно оптимизировать?

- Правильное размещение данных
- Локализация
- Аппаратная или программная предвыборка (prefetch)

Базовый пример: матричное умножение

По определению из линейной алгебры:

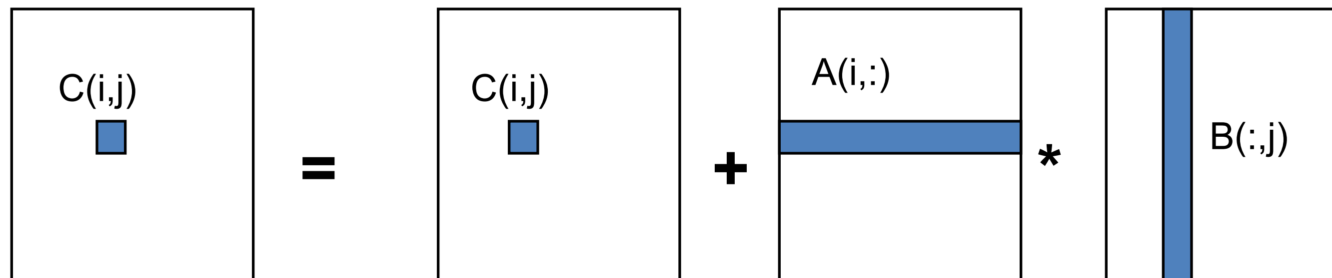
$$c_{ij} = \sum_{k=1}^{n_2} a_{ik} c_{kj}, \; i = \overline{1, n_1}, \; j = \overline{1, n_3}$$

Обобщенное матричное умножение
(например, *dgemm BLAS3*)
C = C + α A B

В дальнейшем считаем матрицы квадратными размерности *n*

Рассмотрим систему с двумя уровнями памяти: быстрой и медленной
Алгоритм 1. По определению

```
for i = 1 to n
    {считываем строку i матрицы A в быструю память}        for j = 1
to n
        {считываем C[i][j] в быструю память}
        for k = 1 to n
            {считываем B[k][j] в быструю память}
        C[i][j] = C[i][j] + A[i][k] * B[k][j]
        {записываем C[i][j] обратно в медленную память}
```



Анализ обращений к памяти

```
for i = 1 to n
    {считываем строку i матрицы A в быструю память}        for j = 1
to n
        {считываем C[i][j] в быструю память}
        for k = 1 to n
            {считываем B[k][j] в быструю память}
        C[i][j] = C[i][j] + A[i][k] * B[k][j]
        {записываем C[i][j] обратно в медленную память}
```

Подсчитаем число обращений к медленной памяти

m = n^3 для чтения столбцов B n-раз

\+ n^2 для чтения строк A один раз для каждого i

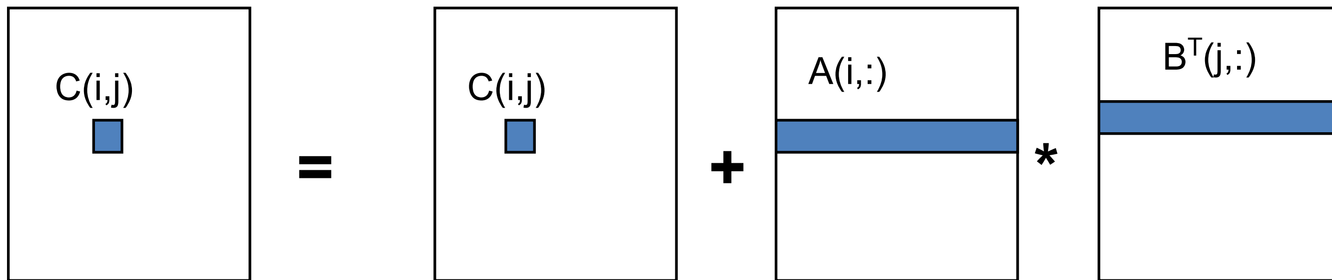
\+ 2n^2 для чтения и записи каждого элемента C

= n^3 + 3n^2

Модификация 1: транспонирование B

Для улучшения скорости доступа транспонируем B

$$C = C + AB^T$$



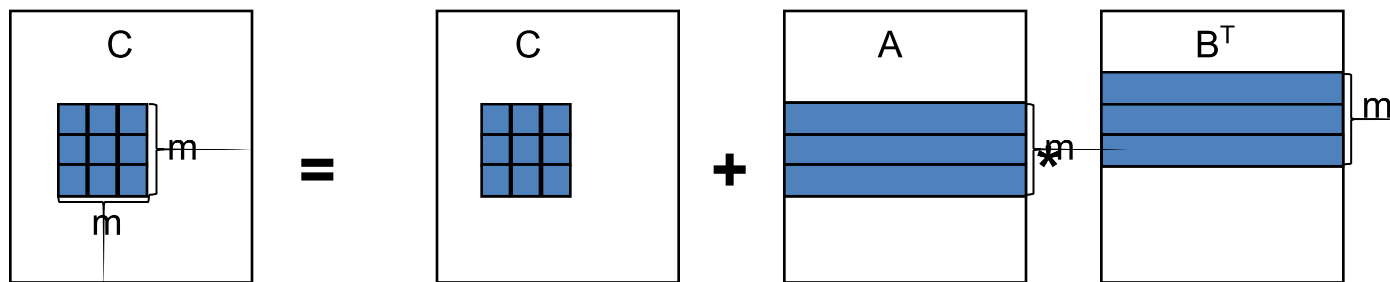
Доступ по строке соответствует алгоритму работы кэш-памяти

Возможность использования оптимизированной функции скалярного произведения

```
double dot_prod(const int n, double *a, double * b);
C[i][j] = C[i][j] + dot_prod(n, A[i], BT[j])
```

Модификация 2: переупорядочивание вычислений

Будем вычислять значения c[i][j] не построчно, а в пределах блоков размерности mxm



Объем данных для заполнения одного блока

$2m^2 + 2mn <= M$,

M – объем быстрой памяти (число вещественных чисел)

$m <= 0,5(\sqrt{n^2 + 2M} - n)$

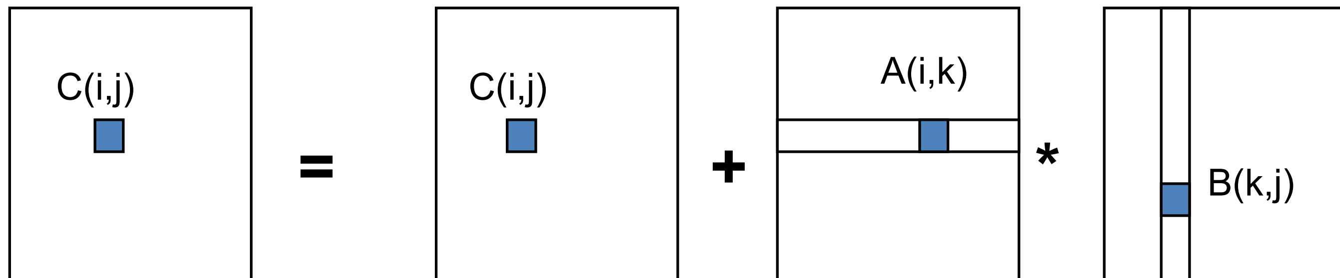
Если пренебречь необходимостью чтения и записи блока матрицы C, то $m <= 0,5M/n$

Пример: cache = 1Мб, M = 1024^2/8, n = 1000,

m_opt1 = 61, m_opt2 = 65

Модификация 3: Блочный алгоритм

- Разобьем матрицы A,B,C на блоки размерности m на m. m=n / N – размер блока, N – число блоков в строке и столбце матрицы
- for i = 1 to N
- for j = 1 to N
- {считываем блок C(i,j) в быструю память}
- for k = 1 to N
- {считываем блок A(i,k) в быструю память}
- {считываем блок B(k,j) в быструю память}
- C(i,j) = C(i,j) + A(i,k) * B(k,j) {выполняем матричное умножение для блоков}
- {записываем блок C(i,j) в медленную память}



Анализ

Number of slow memory references on blocked matrix multiply

- **m = N*n2 to read each block of B N3 times (N3 * n/N * n/N)**
- **+ N*n2 to read each block of A N3 times**
- **+ 2n2 to read and write each block of C once**
- **= (2N + 2) * n2**
- **~ (2/b) * n3**
- **b = n/N times fewer slow memory references than untiled algorithm**
  - **Assumes all three bxb blocks from A,B,C must fit in fast memory**
  - **3b2 <= M = fast memory size**
  - **Decrease in slow memory references limited to a factor of O(sqrt(M))**
- **Theorem (Hong & Kung, 1981): “Any” reorganization of this algorithm (using only associativity) has at least O(n3/sqrt(M)) slow memory refs**
- **Apply tiling recursively for multiple levels of memory hierarchy**