

Средства и системы параллельного программирования

сентябрь – декабрь 2016 г.

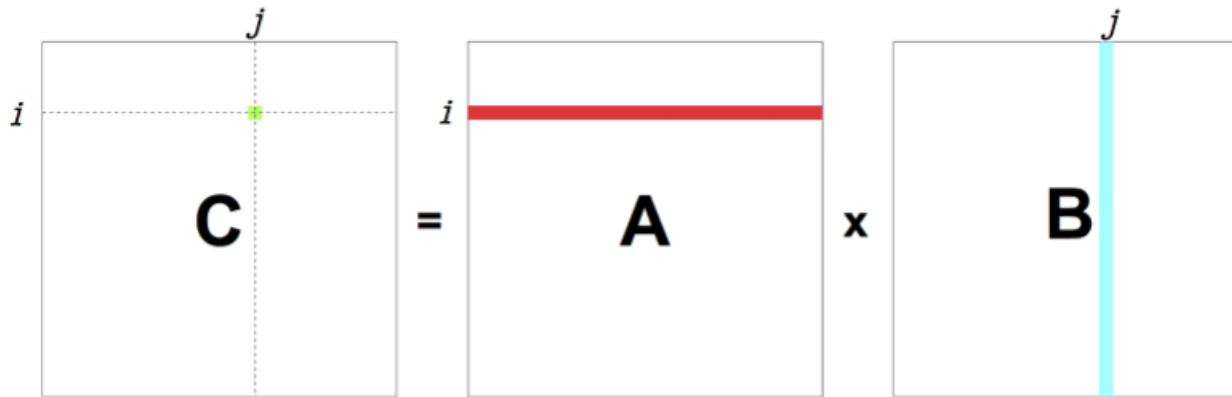
Лектор доцент Н.Н.Попова

Лекция 2
20 сентября 2016 г.

Тема

- Исследование производительности матричного умножения
- PAPI

Матричное умножение $C=A \times B$



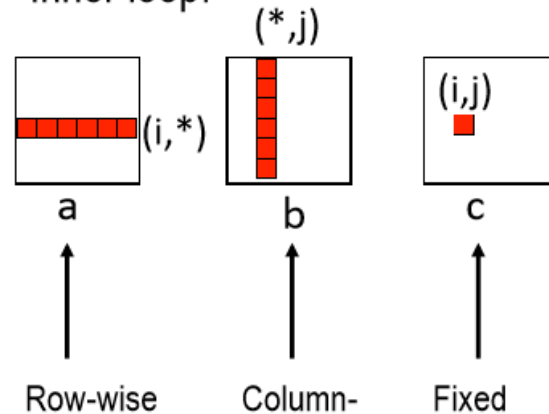
- A, B и C – квадратные матрицы размера $n \times n$
- n достаточно большое, т.е. $n > L$

Матричное умножение

- Матричное умножение – классический пример алгоритма с плохой производительностью кэша
 - время выполнения - $O(N^3)$
 - $c[i][j]$ вычисляется как скалярное произведение строки i на столбец j
 - предположим, что строка кэша равна 32 Bytes и размер матрицы достаточно большой ($N > 1000$)
 - в строке кэша могут размещаться 4 вещественных числа двойной точности

```
for (int i=0; i<N; i++) {  
    for (int j=0; j<N; j++) {  
        double sum = 0.0;  
        for (int k=0; k<N; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

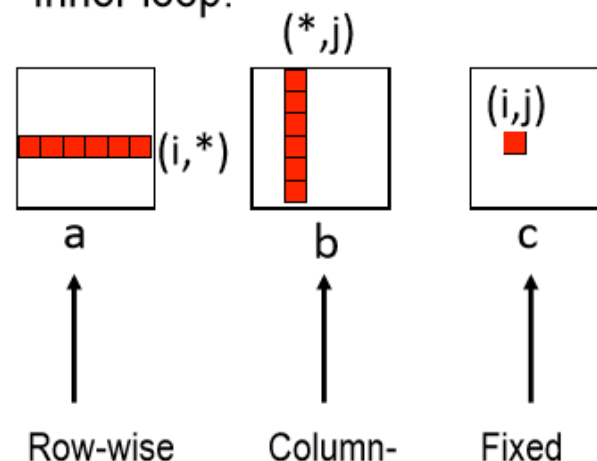
Inner loop:



Матричное умножение

```
for (int i=0; i<N; i++) {  
    for (int j=0; j<N; j++) {  
        double sum = 0.0;  
        for (int k=0; k<N; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

Inner loop:



Исследуем доступ к памяти во вложенном цикле.

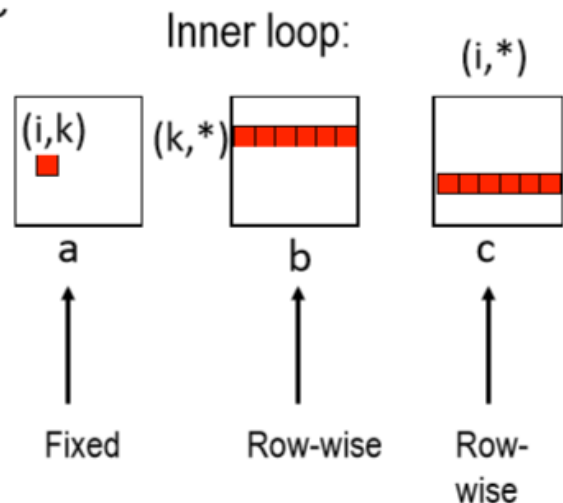
- Элементы матрицы a доступны с шагом 1
 - при чтении очередного элемента в строку кэша записываются 4 элемента., т.е. каждый 4-ый доступ к элементу будет происходить промах кэша (тип промаха – compulsory)
- Доступ к элементам матрицы b происходит с шагом N
 - при каждом чтении элемента матрицы b происходит промах кэша
- Значение sum будет находиться на регистре

Прوماхи кэша

a	b	c
0,25	1,0	0,0

Альтернативная реализация: (i,k,j)

- Что меняется, если меняем порядок вложенных циклов
 - умножаем $a[i][k]$ на все элементы строки k матрицы b и получаем частичную сумму строки i матрицы c
 - доступ к элементам матриц b и c строчный
 - переменная r будет находиться на регистре



Прوماхи кэша:

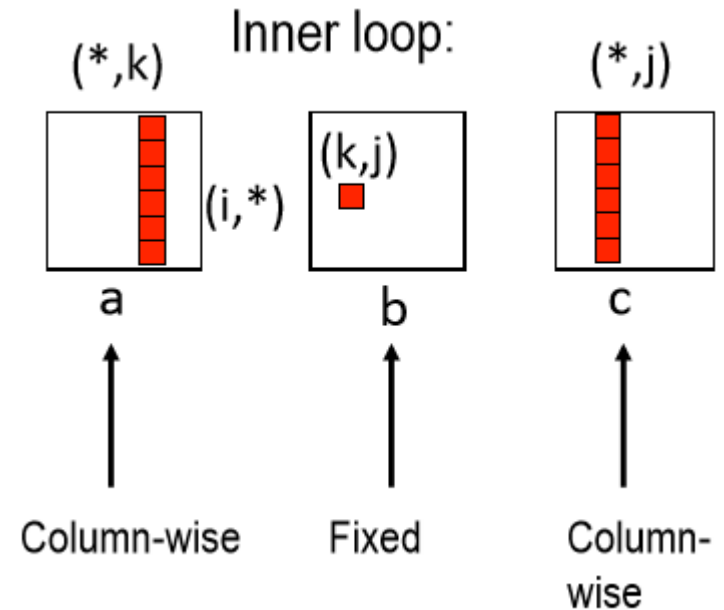
a	b	c
0,0	0,25	0,25

```
for (int i=0; i<N; i++) {  
    for (int k=0; k<N; k++) {  
        double r = a[i][k];  
        for (int j=0; j<N; j++)  
            c[i][j] += r * b[k][j];  
    }  
}
```

Альтернативная реализация : (j,k,i)

```
for (int j=0; j<N; j++) {  
    for (int k=0; k<N; k++) {  
        double r = b[k][j];  
        for (int i=0; i<N; i++)  
            c[i][j] += a[i][k] * r;  
    }  
}
```

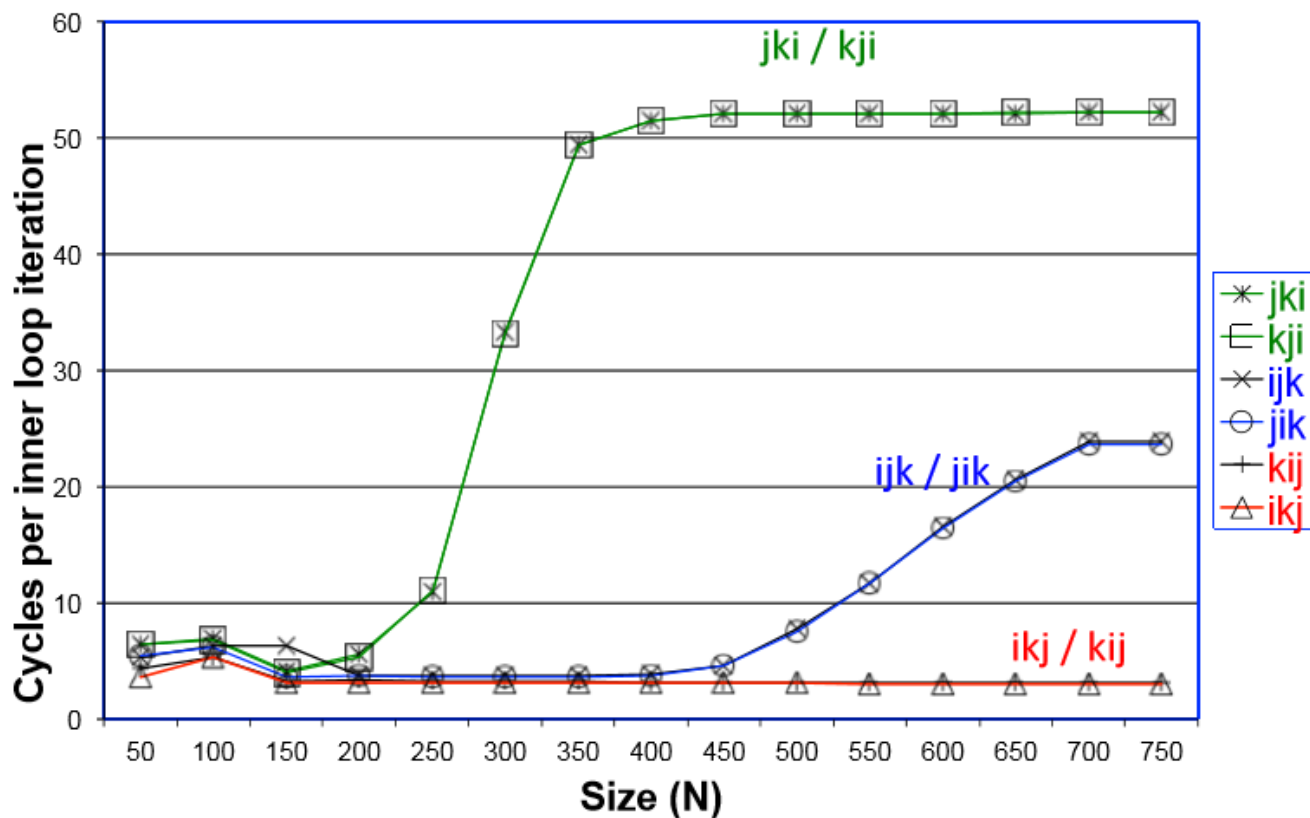
a	b	c
1,0	0,0	1,0



Выводы

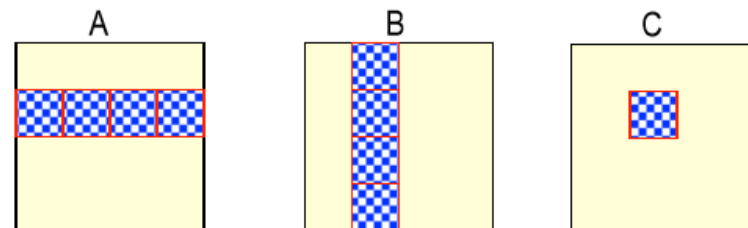
- Можно определить 6 вариантов перестановок 3-ех вложенных циклов перестановок индексов трех вложенных циклов при реализации алгоритма матричного умножения
 - варианты, при которых меняется порядок двух внешних циклов, имеют идентичное поведение промахов кэша
- ijk (и jik)
 - 2 loads, 0 stores
 - 1,25 cache misses/iteration
- ikj (и kij)
 - 2 loads, 1 store
 - 0,5 cache misses/iteration
- jki (и kji)
 - 2 loads, 1 store
 - 2,0 cache misses/iteration

Производительность матричного умножения на процессоре Core i7



Блочное матричное умножение

- Оптимизация для данных, которые не укладываются в кэше
- Разделение данных на меньшие блоки размера $b \times b$, которые размещаются в кэше
 - выполнение вычислений над блоками, находящимися в кэше
- Пример: матричное умножение $C = A \times B$
 - матрицы представляются в виде $N \times N$ матриц
 - вычисления над блоком выполняются без дополнительного обращения к памяти
- Размер блока выбирается таким образом, чтобы все данные, необходимые для вычисления одного блока, располагались в кэше
 - доступ к элементам B по-прежнему остается по столбцам, но блок при этом полностью располагается в кэше.



Блочное матричное умножение

```
for (int i = 0; i < N; i+=b) // b is the blocksize
    for (int j = 0; j < N; j+=b)
        for (int k = 0; k < N; k+=b)
            /* B x B mini matrix multiplications */
            for (i1 = i; i1 < i+b; i++)
                for (j1 = j; j1 < j+b; j++)
                    for (k1 = k; k1 < k+b; k++)
                        c[i1][j1] += a[i1][k1]*b[k1][j1];
```

Алгоритм с представлением матриц одним массивом

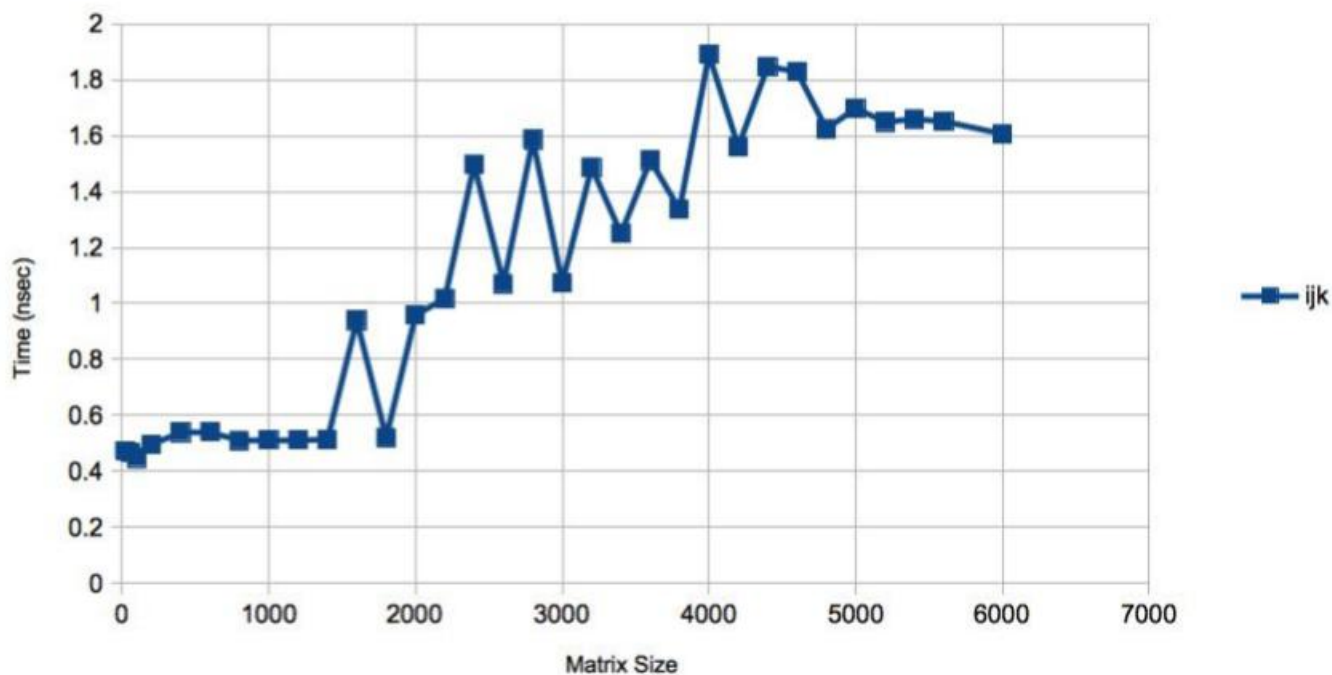
- Элементы матрицы располагаются одним массивом
- Формат представления – строчный
- $m \times n$ - m число строк, n – столбцов
- $A(i,j)$ элемент: i -ая строка, j -ый столбец, индекс в одномерном массиве - $(i \times n + j)$

Умножение для представления матриц одномерным массивом

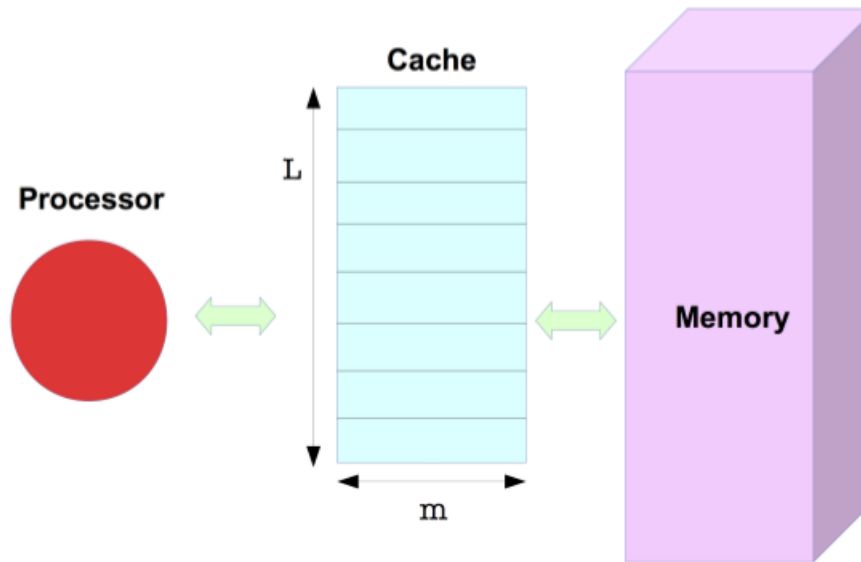
```
void square_dgemm (int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; ++i) {
        const int iOffset = i*n;
        for (int j = 0; j < n; ++j) {
            double cij = 0.0;
            for( int k = 0; k < n; k++ )
                cij += A[iOffset+k] * B[k*n+j];
            c[iOffset+j] += cij;
        }
    }
}
```

Общее число
умножений: n^3

Время умножения: ijk

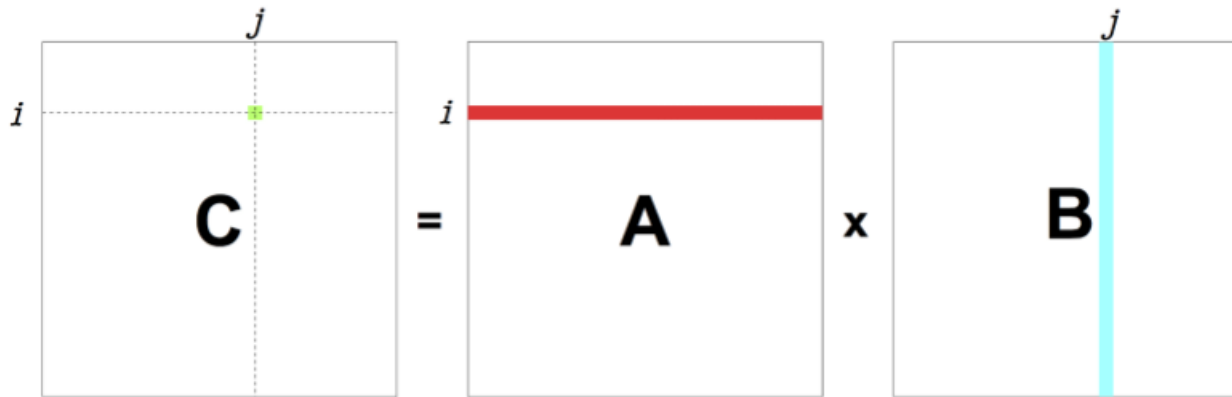


Пример: организация кэша



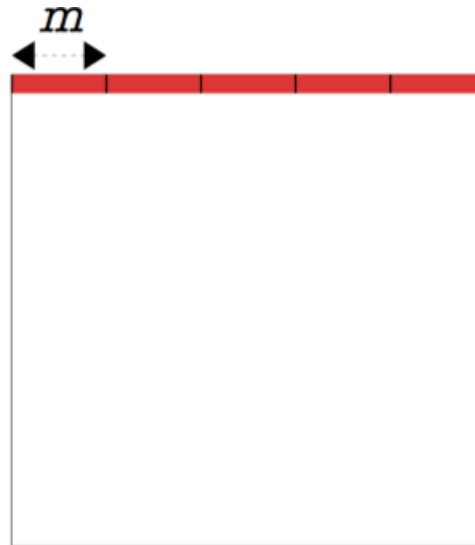
- L строк объемом m вещественных чисел с плав. точкой
- Предположим, что кэш «высокий» : $L > m$
- Алгоритм : Least Recently Used
- Нет аппаратной реализации предварительной загрузки (prefetching)

Матричное умножение $C=A \times B$



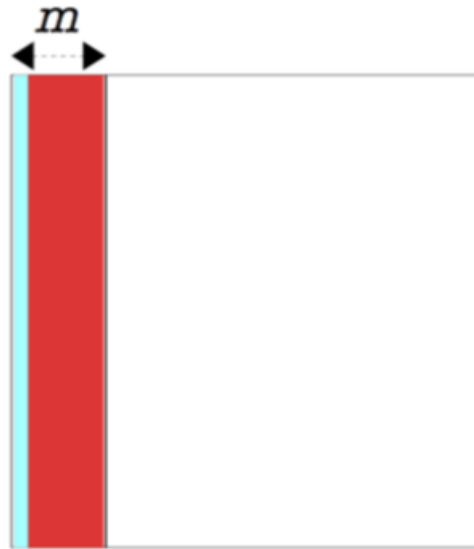
- A, B и C – квадратные матрицы размера $n \times n$
- n достаточно большое, т.е., $n > L$

Доступ к элементам строки в L1



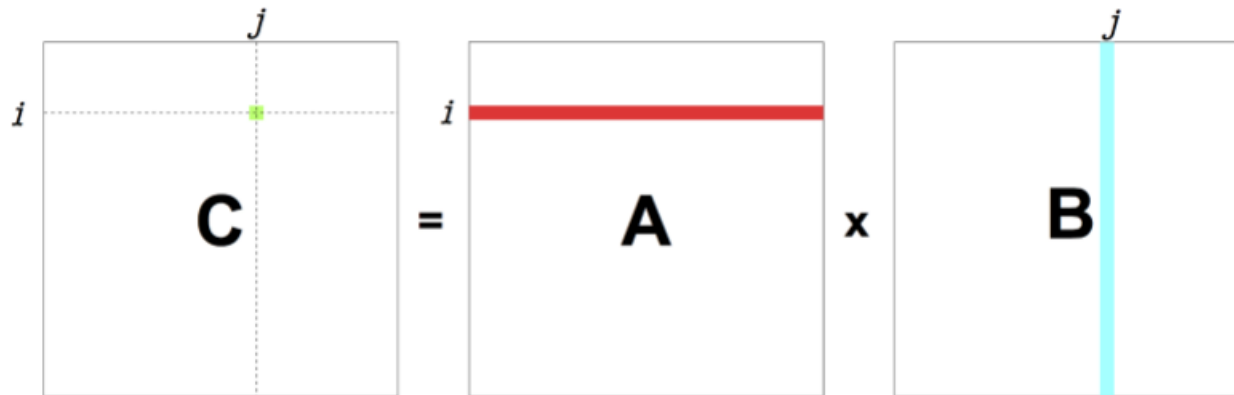
- Последовательный доступ : доступ к элементам строки требует $\frac{n}{m}$ промахов кэша

Доступ к элементам столбца в L1



- Доступ с шагом (Strided) : доступ к столбцу требует n промахов кэша

Оценка числа промахов кэша: ijk



- Для вычисления каждого $C[i][j]$ число промахов кэша: $1 + \frac{n}{m} + n$
- Если $n < m$, общее число промахов кэша - $\frac{2n^2}{m} + n^3$
- Если $n > m$, общее число промахов кэша - $\frac{n^2}{m} + \frac{n^3}{m} + n^3$

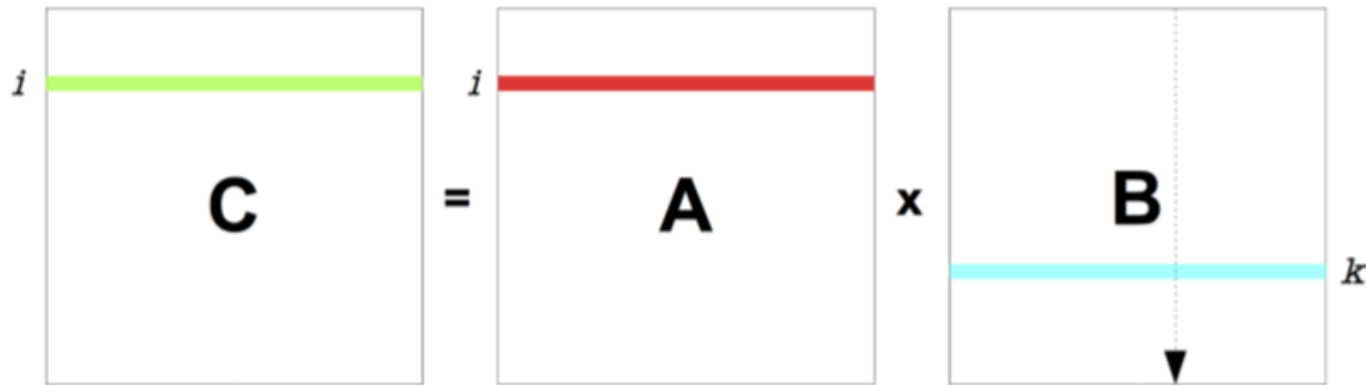
Оценка числа промахов кэша при $n < mL$

- Размер строки кэша 64 bytes означает $m = 8$
- 256 KB L2 кэш означает $mL = 32768$
- В реальных приложениях $n < 10 - 15\text{ k}$
- Таким образом, если $n < mL$, общее число промахов кэш: :

$$2n^2m + n^3 = O(n^3)$$

- Можно улучшить?

Альтернативный доступ к элементам массивов: ikj

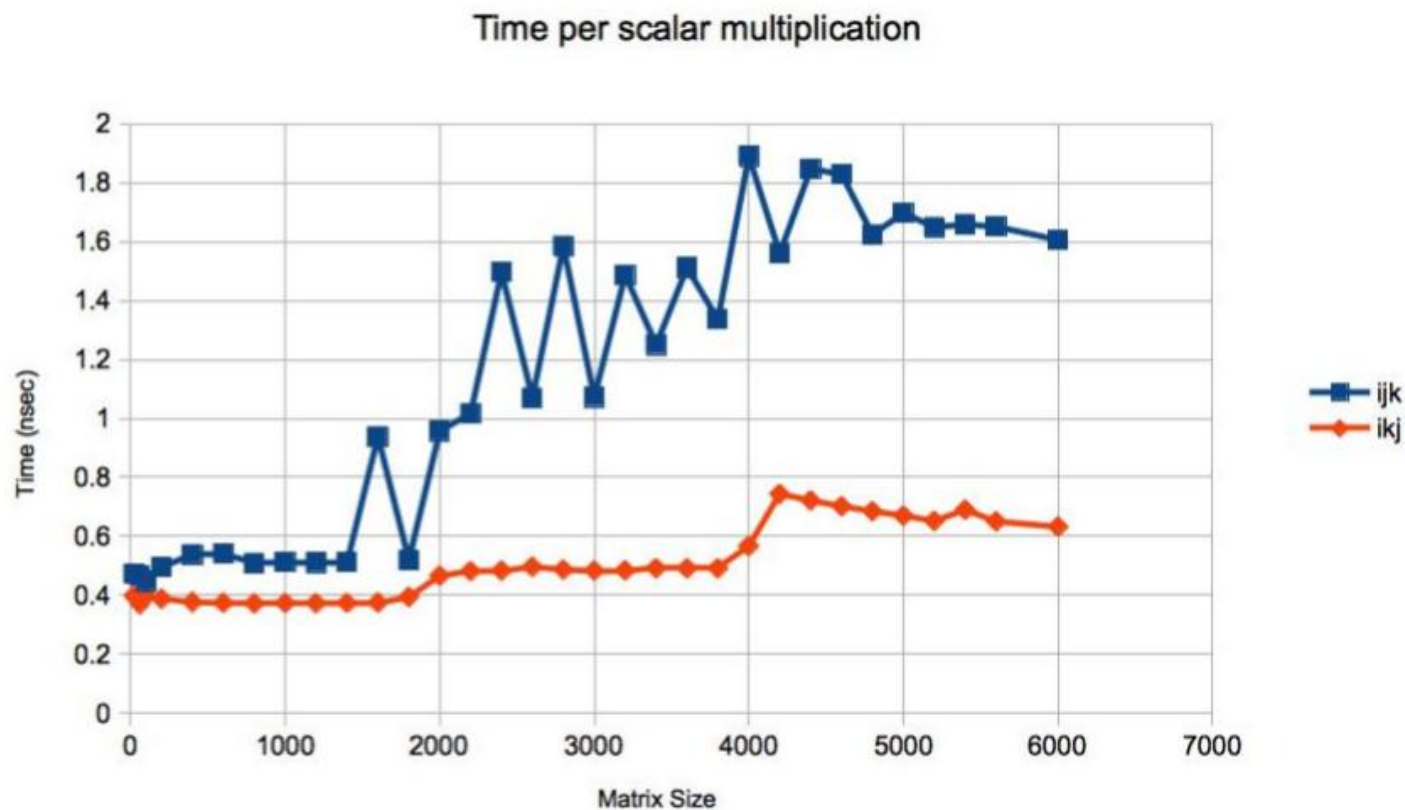


- Вычисление строки $C(i,:)$ промахи кэша $\frac{2n}{m} + \frac{n^2}{m}$
- Общее число промахов : $\frac{2n^2}{m} + \frac{n^3}{m} = O(\frac{n^3}{m})$

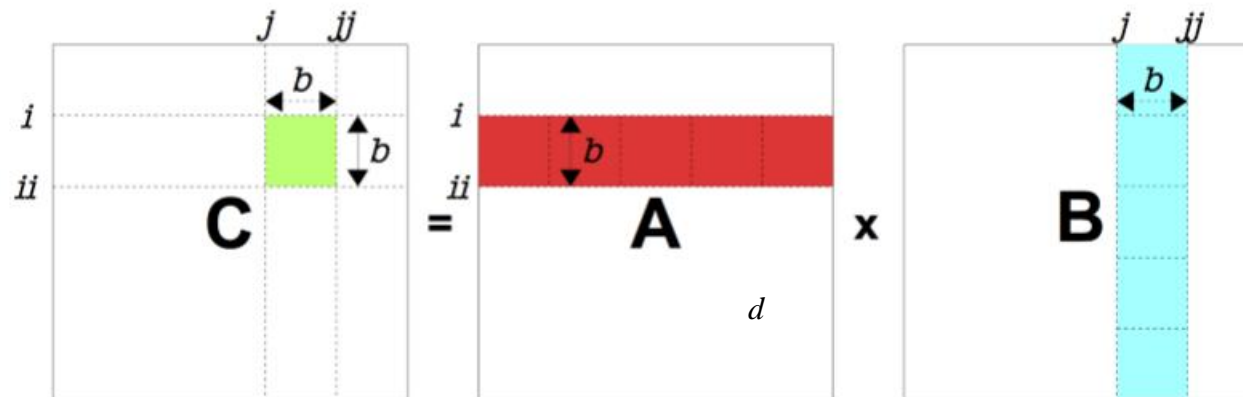
ikj реализация

```
void square_dgemm (int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; ++i)
    {
        const int iOffset = i*n;
        for (int k = 0; k < n; ++k) {
            double cij = 0.0;
            for( int j = 0; j < n; j++ )
                const int kOffset = k*n;
            C[iOffset+j] += A[iOffset+j] * B[kOffset+j]
            cij += A[iOffset+k] * B[k*n+j];
            c[iOffset+j] += cij;
        }
    }
}
```

Сравнение времени ikj и ijk



Блочный вариант



- Предположения для анализа: $n \% b = 0$ $m \% b = 0$
- Промехи кэша при загрузке кэша: $\frac{b^2}{m}$
- Промехи кэша при вычислении блока C: $\frac{b^2}{m} + \frac{b^2}{m} \frac{2n}{b} = \frac{b^2}{m} + \frac{2nb}{m}$
- Общее число промахов кэша: $\frac{n^2}{b^2} \left(\frac{b^2}{m} + \frac{2nb}{m} \right) = \frac{n^2}{m} + \frac{2n^3}{mb} = \Theta\left(\frac{n^3}{mb}\right)$

Оценка размера блока

- 3 блока A, B и C должны разместиться в кэше
- $3b^2 = mL$, i.e., $b = \sqrt{\frac{mL}{3}}$
- Для L1 объемом 32 KB, $mL = 4096$ и $b = 36.95$
- Значение b можно выбрать равным 32
- Общее число промахов кэша: $\frac{2n^3}{mb} = \frac{2\sqrt{3}n^3}{m\sqrt{mL}} = \Theta\left(\frac{n^3}{m\sqrt{\text{Cache Size}}}\right)$

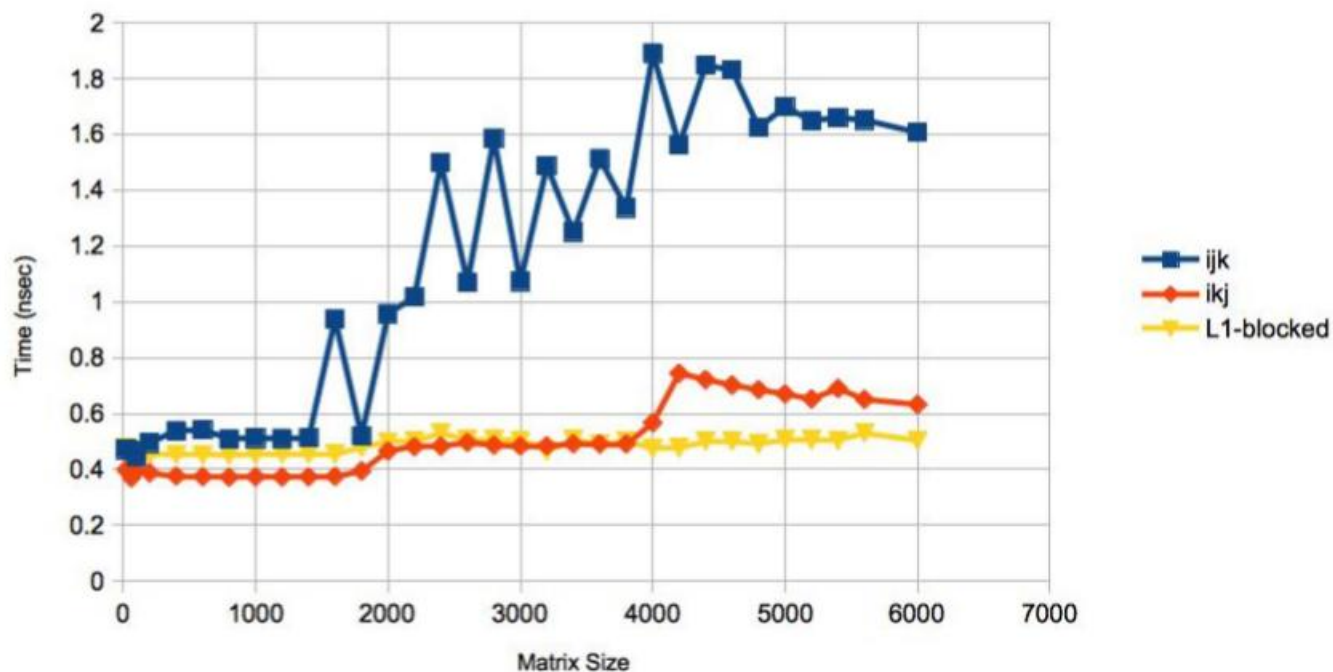
Блочный алгоритм (1)

```
void square_dgemm (int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; i += BLOCK_SIZE)
    {
        const int iOffset = i * n;
        for (int j = 0; j < n; j += BLOCK_SIZE)
            for (int k = 0; k < n; k += BLOCK_SIZE)
            {
                /* Correct block dimensions if block "goes off
                   edge of" the matrix */
                int M = min (BLOCK_SIZE, n-i);
                int N = min (BLOCK_SIZE, n-j);
                int K = min (BLOCK_SIZE, n-k);
                /* Perform individual block dgemm */
                do_block(n, M, N, K, A + iOffset + k,
                        B + k*n + j, C + iOffset + j);
            }
    }
}
```

Блочный алгоритм (2)

```
static void do_block (int n, int M, int N, int K, double* A,
                     double* B, double* C)
{
    for (int i = 0; i < M; ++i)
    {
        const int iOffset = i*n;
        for (int j = 0; j < N; ++j)
        {
            double cij = 0.0;
            for (int k = 0; k < K; ++k)
                cij += A[iOffset+k] * B[k*n+j];
            C[iOffset+j] += cij;
        }
    }
}
```

Сравнение вариантов ijk , ikj и L1-блочный

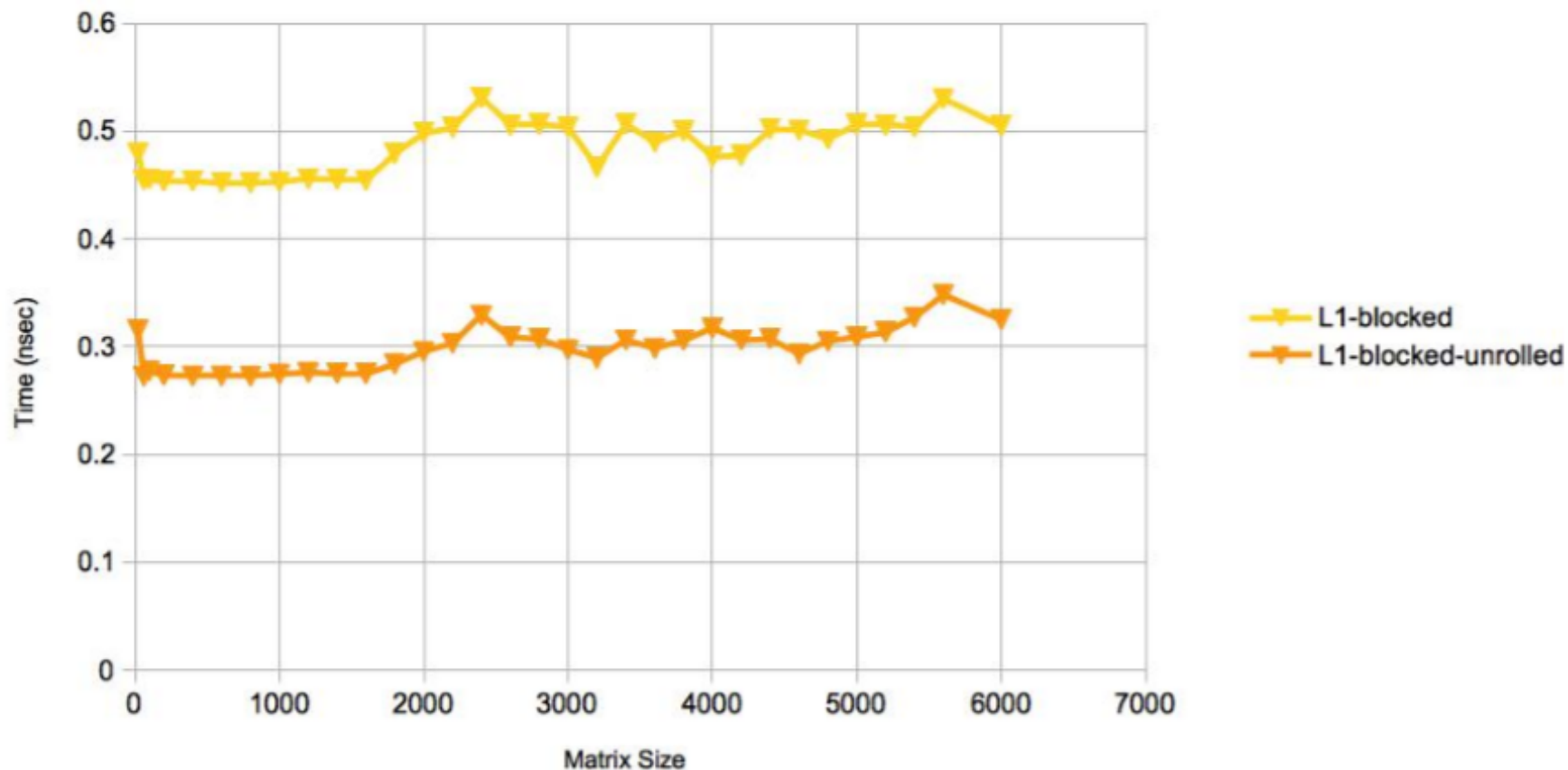


Оптимизация блочного алгоритма

```
for (int k = 0; k < K; ++k)
    cij += A[iOffset+k] * B[k*n+j];
```

```
for (int k = 0; k < K; k += 8)
{
    const double d0 = A[iOffset+k] * B[k*n+j];
    const double d1 = A[iOffset+k+1] * B[(k+1)*n+j];
    const double d2 = A[iOffset+k+2] * B[(k+2)*n+j];
    const double d3 = A[iOffset+k+3] * B[(k+3)*n+j];
    const double d4 = A[iOffset+k+4] * B[(k+4)*n+j];
    const double d5 = A[iOffset+k+5] * B[(k+5)*n+j];
    const double d6 = A[iOffset+k+6] * B[(k+6)*n+j];
    const double d7 = A[iOffset+k+7] * B[(k+7)*n+j];
    cij += (d0 + d1 + d2 + d3 + d4 + d5 + d6 + d7);
}
```

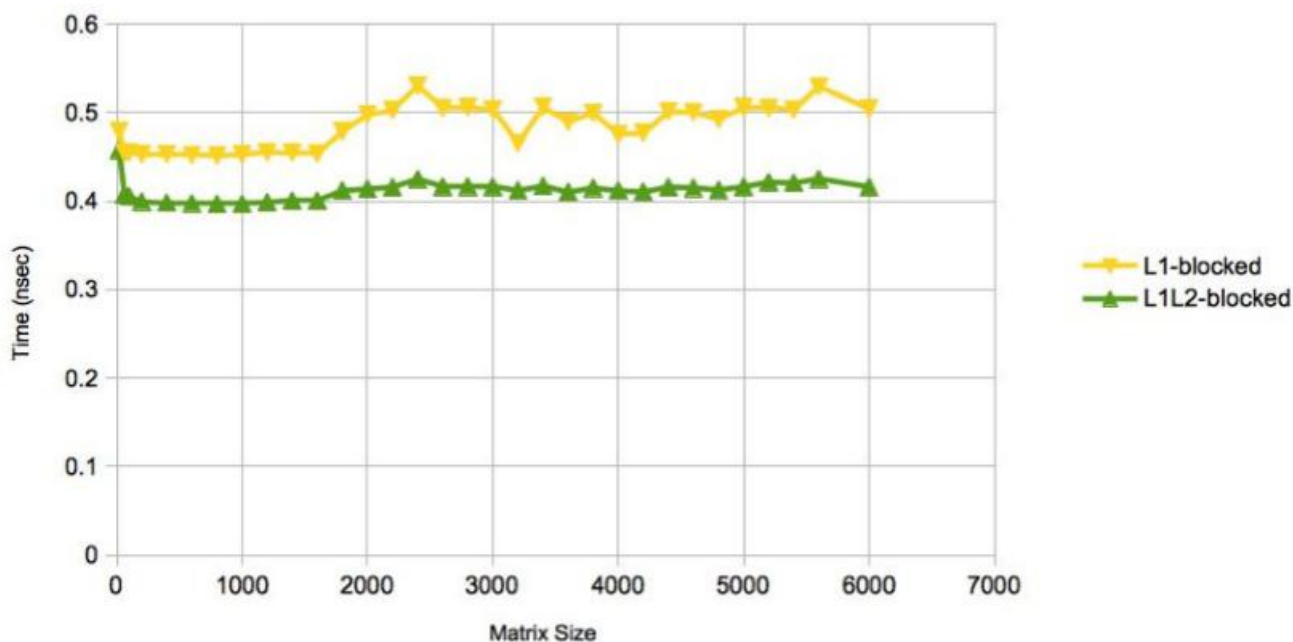
Сравнение: L1-блочный и L1-блочный с оптимизацией



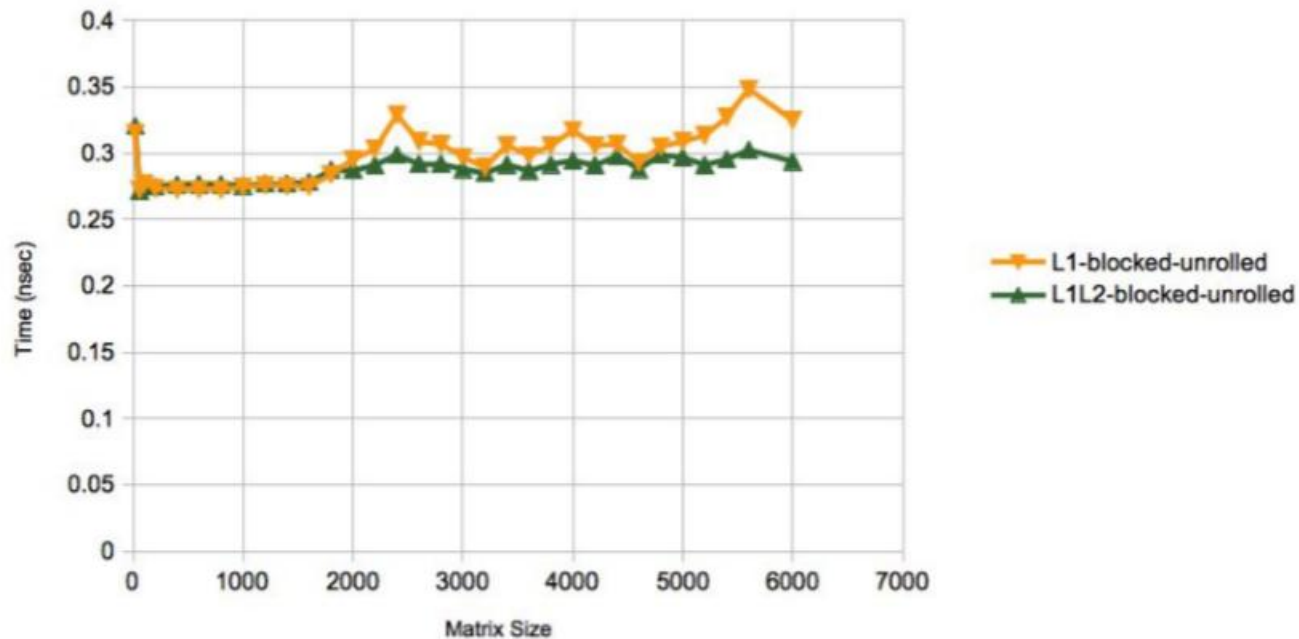
L2

- В дополнение к L1 блочность может быть и для L2
- Для L2 объемом 256 KB, $mL = 4096$ и $b = 105$
- Определение лучшего размера L2 является нетривиальной задачей и требует проведения экспериментов с различными размерами
- В приводимом примере выбрано значение 88 для L2

Сравнение: L1-блочный & L1L2-блочный



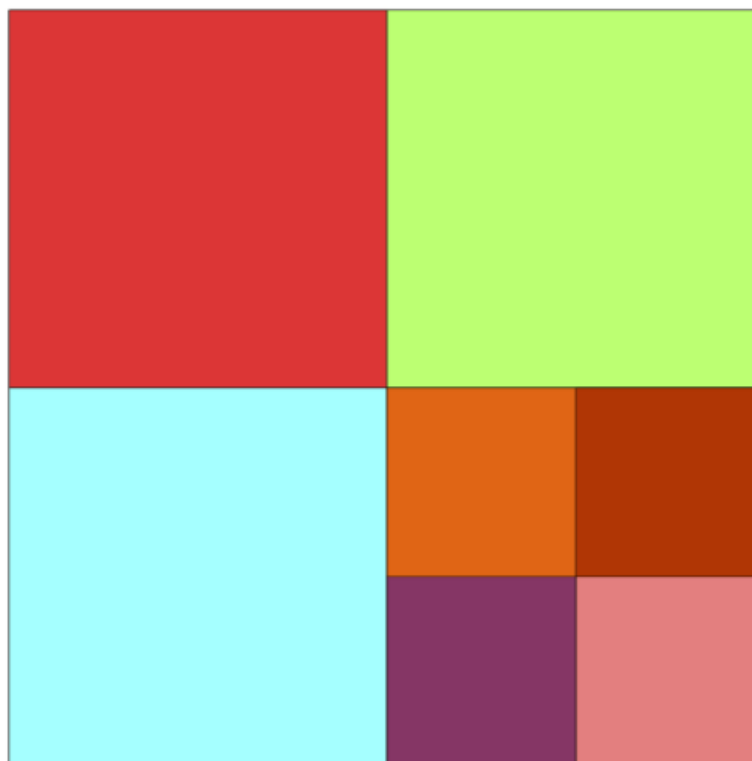
Сравнение оптимизированных вариантов: L1-блочный & L1L2-блочный



Рекурсивный алгоритм умножения

$$\begin{aligned}C &= A \times B \\ \Rightarrow \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\ \Rightarrow C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}\end{aligned}$$

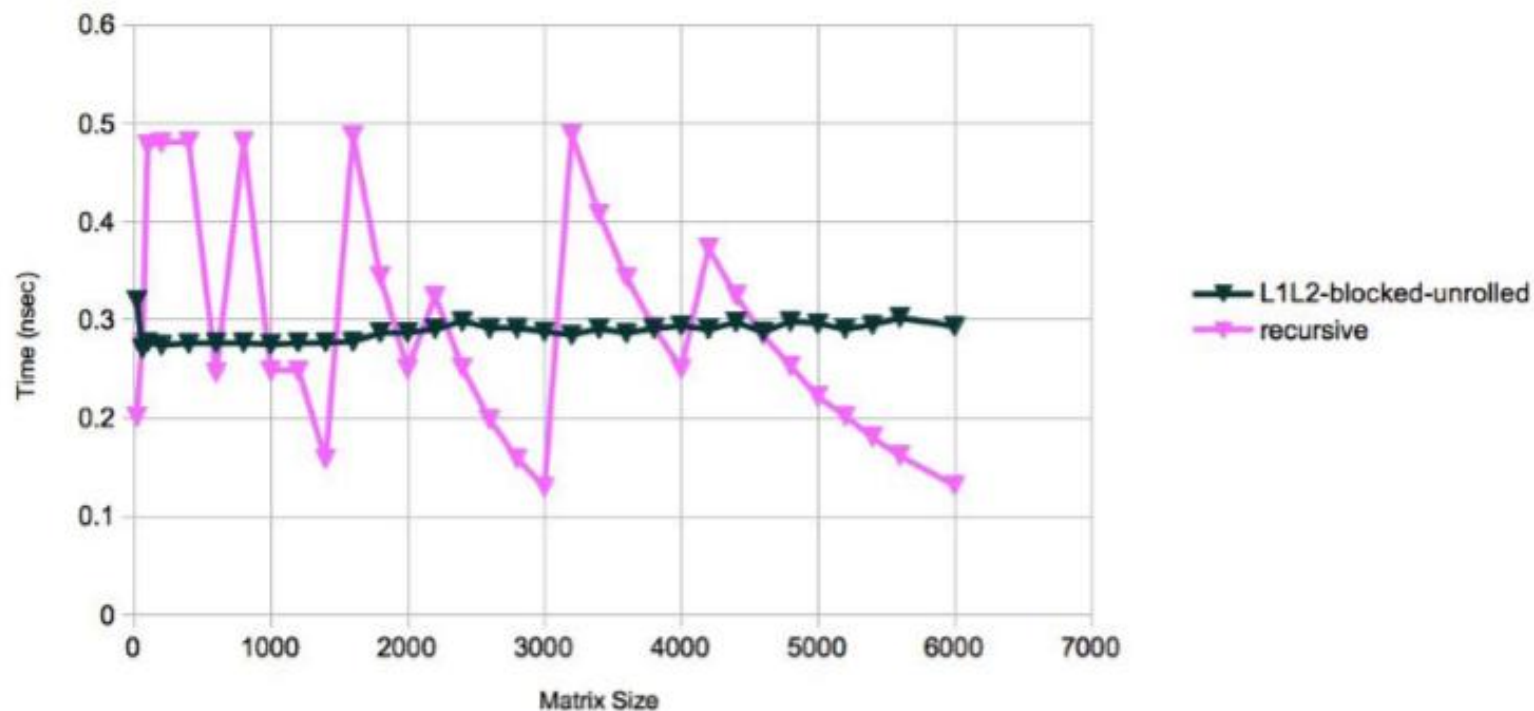
Рекурсивный алгоритм: разбиение матриц



Рекурсивный алгоритм

```
void multiply(const int n, const int m, double * A, double * B, double * C)
{
    //Base cases
    if(m == 2)
    {
        baseCase2(n, m, A, B, C);
        return;
    }
    else if(m == 1)
    {
        C[0] += A[0] * B[0];
        return;
    }
    //Recursive multiply
    const int offset12 = m/2;
    const int offset21 = n*m/2;
    const int offset22 = offset21 + offset12;
    multiply(n, m/2, A, B, C);
    multiply(n, m/2, A+offset12, B+offset21, C);
    multiply(n, m/2, A, B+offset12, C+offset12);
    multiply(n, m/2, A+offset12, B+offset22, C+offset12);
    multiply(n, m/2, A+offset21, B, C+offset21);
    multiply(n, m/2, A+offset22, B+offset21, C+offset21);
    multiply(n, m/2, A+offset21, B+offset12, C+offset22);
    multiply(n, m/2, A+offset22, B+offset22, C+offset22);
    return;
}
```

Сравнение: рекурсивный алгоритм и L1L2 оптимизированный блочный



Литература

- Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms, Agner Fog
(<http://www.agner.org/optimize/>)
- Cache: a place for concealment and safekeeping, Gustavo Duarte
(<http://duartes.org/gustavo/blog/post/intel-cpu-caches/>)
- Drepper, U. What Every Programmer Should Know About Memory (2007). <http://people.redhat.com/drepper/cpumemory.pdf>.

Архитектура процессоров

- RISC или суперскалярные архитектуры
 - Конвейерные функциональные устройства (ФУ)
 - Несколько ФУ на процессоре
 - Спекулятивное исполнение операций
 - Многоуровневый кэш
 - Блоки (Cache lines) могут разделяться между несколькими процессорами

Аппаратные счетчики

- Небольшое множество регистров, который накапливают число *событий* – специфических сигналов, связанных с функционированием процессора
- Контроль за этими событиями позволяет наблюдать за тем, насколько эффективно исполняемая на процессоре программа использует ресурсы процессора

Конвейерные ФУ

- Схема из логических элементов на процессоре, которая выполняет определенные операции называется **функциональным устройством**
- Большинство ФУ, выполняющих целочисленную арифметику и операции с плавающей точкой являются **конвейерными**
 - Каждая стадия конвейерного ФУ работает одновременно (со своими операндами)
 - Цель: после начальной инициализации конвейера генерировать результат за один такт работы процессора

Суперскалярные процессоры

- Процессоры, имеющие несколько ФУ называются ***суперскалярными***
- Пример:
 - IBM Power 3
 - 2 floating point units (multiply-add)
 - 3 fixed point units
 - 2 load/store units
 - 1 branch/dispatch unit

“Out of Order” выполнение операций

- CPU динамически выполняет инструкции по мере их готовности, в случае необходимости, не в последовательном, предусмотренном программой порядке
 - Любой результат, полученный в результате «out of order» является временным, пока все предшествующие операции не завершатся успешно.
 - Для выбора инструкций на выполнение на ФУ используются очереди
 - Соответствующие аппаратные метрики: выданные инструкции, завершённые инструкции

Спекулятивное выполнение

- ЦПУ пытается предсказать порядок условных переходов и выполняет инструкции спекулятивно по предсказанному пути
 - Если предсказание неверное, выполненные инструкции отменяются
 - На многих процессорах аппаратные счетчики накапливают число правильных и неправильных предсказаний (branch prediction).

Счетчики инструкций (instruction counts) и статуса ФУ

- Аппаратные счетчики, хранящие данные об
 - общем числе тактов
 - общем числе инструкций
 - операциях с плавающей точкой
 - Load/store инструкциях
 - тактах, во время которых ФУ простаивали
 - заблокированных тактах, связанных с
 - ожиданием доступа к операндам в памяти
 - ожиданием ресурсов
 - Инструкциях условного перехода
 - выполненных
 - неправильно предсказанных (mispredicted)

Аппаратные счетчики для работы с кэшем

- Аппаратные счетчики
 - Cache misses and hit ratios
 - Cache line invalidations (рассогласование кэша)

TLB и виртуальная память

- Страничная организация памяти .
- ОС транслирует виртуальные адреса в физические.
 - Последние используемые адреса кэшируются в *translation lookaside buffer (TLB)*.
 - Если в программе происходит доступ к виртуальным адресам, отсутствующим в TLB, происходит *TLB miss* .
- Соответствующий аппаратный счетчик: TLB misses

Задержки по доступу к памяти (latency)

- CPU register: 0 cycles
- L1 cache hit: 2-3 cycles
- L1 cache miss satisfied by L2 cache hit: 8-12 cycles
- L2 cache miss satisfied from main memory, no TLB miss: 75-250 cycles
- TLB miss requiring only reload of the TLB: ~2000 cycles
- TLB miss requiring reload of virtual page – *page fault*: hundreds of millions of cycles

Измерение времени выполнения программы

■ Типы времени

wall-clock time

- базируется на реальном времени (постоянно меняется)
- включает все активности

virtual process time (CPU time)

- время, когда выполняется процесс (CPU активно)

user time and system time (могут означать разные вещи)

- не включает время ожидания, в котором находится процесс

Timer: gettimeofday()

- UNIX функция
 - возвращает wall-clock time в секундах и микросекундах
 - точность зависит от аппаратуры
 - базовое отсчет от 00:00 UTC, January 1, 1970
 - некоторые реализации возвращают и временную зону

Пример

```
#include <sys/time.h>
struct timeval tv;
double walltime; /* seconds */
gettimeofday(&tv, NULL);
walltime = tv.tv_sec + tv.tv_usec * 1.0e-6;
```

Timer: *clock_gettime()*

POSIX функция

- wall-clock время в секундах и наносекундах
- Разрешение зависит от аппаратуры

```
#include <time.h>  
struct timespec tv;  
double walltime; /* seconds */  
clock_gettime(CLOCK_REALTIME, &tv);  
walltime = tv.tv_sec + tv.tv_nsec * 1.0e-9;
```

Timer: *getrusage()*

- UNIX функция
 - обеспечивает различную информацию, включая время, системное время, использование памяти и т.п.
 - зависит от реализации

```
#include <sys/resource.h>
struct rusage ru;
double usertime; /* seconds */
int memused;
getrusage(RUSAGE_SELF, &ru);
usertime = ru.ru_utime.tv_sec +
ru.ru_utime.tv_usec * 1.0e-6;
memused = ru.ru_maxrss;
```

Аппаратные метрики

- Cycles / Instructions IPC
- Floating point instructions FLOPS
- Integer instructions computation intensity
- Load/stores instructions per load/store
- Cache misses load/stores per cache miss
- Cache misses cache hit rate
- Cache misses loads per load miss
- TLB misses loads per TLB miss

Overview of PAPI

(<http://icl.cs.utk.edu/papi/>)

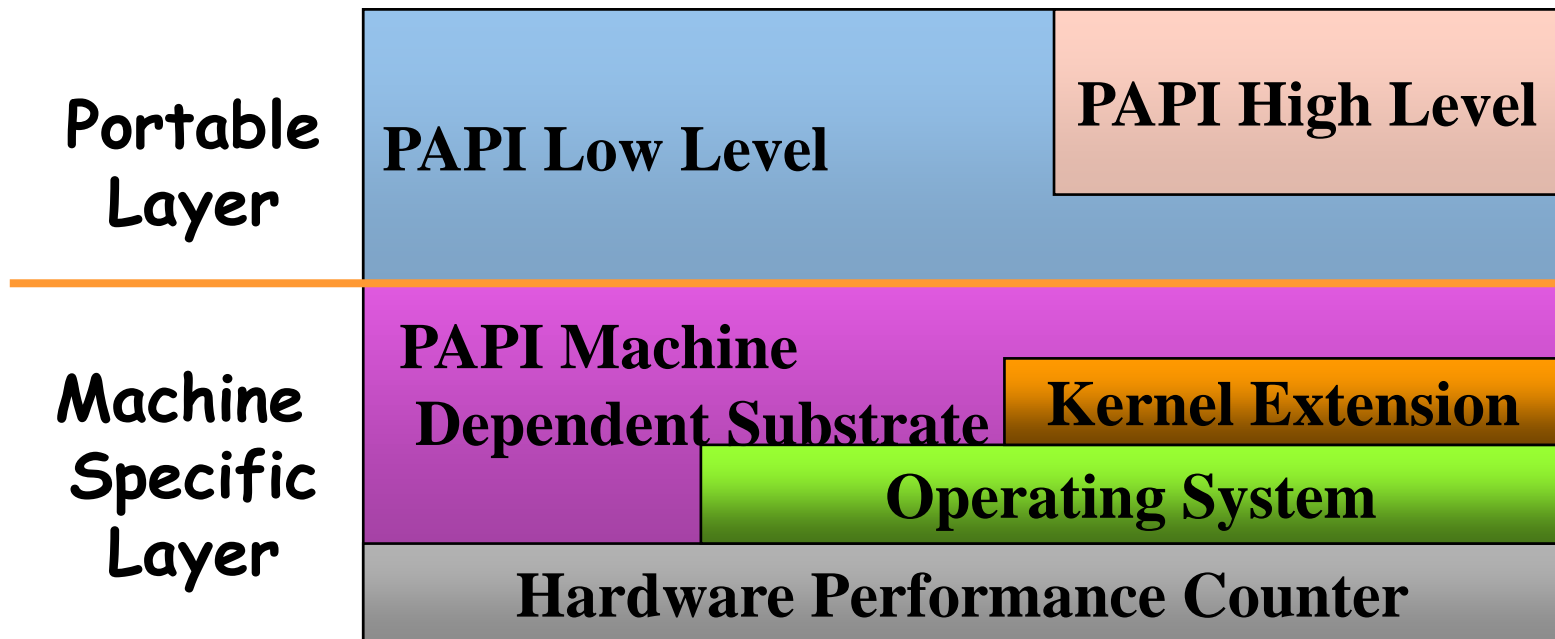
- **Performance Application Programming Interface**
- The purpose of the PAPI project is to design, standardize and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
- Parallel Tools Consortium project
<http://www.ptools.org/>

PAPI Counter Interfaces

- PAPI обеспечивает 3 типа интерфейса к аппаратным счетчикам:
 1. The low level interface manages hardware events in user defined groups called *EventSets*.
 2. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.
 3. Graphical tools to visualize information.

PAPI Implementation

Java Monitor GUI



Основные функции PAPI

- Восемь основных функций:

- *PAPI_num_counters*
- *PAPI_start_counters*,
- *PAPI_stop_counters*
- *PAPI_read_counters*
- *PAPI_accum_counters*
- *PAPI_flops*
- *PAPI_flips*, *PAPI_ipc*

Low Level API

- Increased efficiency and functionality over the high level PAPI interface
 - 54 functions
 - access to native events
 - obtain information about the executable, the hardware, and memory
 - set options for multiplexing and overflow handling

Event sets

- The event set contains key information
 - What low-level hardware counters to use
 - Most recently read counter values
 - The state of the event set (running/not running)
 - Option settings (e.g., domain, granularity, overflow, profiling)
- Event sets can overlap if they map to the same hardware counter set-up.
 - Allows inclusive/exclusive measurements

Event set Operations

- Event set management
PAPI_create_eventset, PAPI_add_event[s],
PAPI_rem_event[s], PAPI_destroy_eventset
- Event set control
PAPI_start, PAPI_stop, PAPI_read, PAPI_accum
- Event set inquiry
PAPI_query_event, PAPI_list_events,...

Initialize the PAPI library

PAPI_library_init()

```
PAPI_library_init()  
if (PAPI_VER_CURRENT !=  
    PAPI_library_init(PAPI_VER_CURRENT))  
    ehandler("PAPI_library_init error.");
```

PAPI_num_counters()

Check how many counters this CPU can monitor

```
const size_t EVENT_MAX = PAPI_num_counters();
```

PAPI_query_event()

```
if (PAPI_OK!=PAPI_query_event(PAPI_TOT_INS))
    ehandler("Cannot count PAPI_TOT_INS.");
if (PAPI_OK != PAPI_query_event(PAPI_L1_DCM))
    ehandler("Cannot count PAPI_L1_DCM.");
if (PAPI_OK != PAPI_query_event(PAPI_L2_DCM))
    ehandler("Cannot count PAPI_L2_DCM.");
```


PAPI_start_counters()

```
size_t EVENT_COUNT = 3;  
int events[] = { PAPI_TOT_INS, PAPI_L1_DCM, PAPI_L2_DCM };  
PAPI_start_counters(events, EVENT_COUNT);
```

PAPI_read_counters()

```
long long values[EVENT_COUNT];  
if (PAPI_OK != PAPI_read_counters(values, EVENT_COUNT))  
    ehandler("Problem reading counters 1.");  
C = matrix_prod(n, n, n, n, A, B);  
if (PAPI_OK != PAPI_read_counters(values, EVENT_COUNT))  
    ehandler("Problem reading counters 2.");  
printf("%d %lld %lld %lld\n", n, values[0], values[1], values[2])
```

PAPI_flops()

```
float rtime;  
float ptime;  
long long flpops;  
float mflops;  
if (PAPI_OK != PAPI_flops(&rtime, &ptime, &flpops, &mflops))  
    ehandler("Problem reading flops 1");  
C = matrix_prod(n, n, n, n, A, B);  
if (PAPI_OK != PAPI_flops(&rtime, &ptime, &flpops, &mflops))  
    ehandler("Problem reading flops 2");  
printf("%d %lld %f\n", n, flpops, mflops);
```

Simple Example

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC}, EventSet;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
```

High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Not thread safe
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (additional code) than low-level

High-level API

- C interface
 - PAPI_start_counters
 - PAPI_read_counters
 - PAPI_stop_counters
 - PAPI_accum_counters
 - PAPI_num_counters
 - PAPI_flops

PAPI High-level Example

```
long long values[NUM_EVENTS];
unsigned int
    Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC}
;
/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);
/* What we are monitoring? */
do_work();
/* Stop the counters and store the results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```

Setting up the High-level Interface

- *int PAPI_num_counters(void)*
 - Initializes PAPI (if needed)
 - Returns number of hardware counters
- *int PAPI_start_counters(int *events, int len)*
 - Initializes PAPI (if needed)
 - Sets up an event set with the given counters
 - Starts counting in the event set
- *int PAPI_library_init(int version)*
 - Low-level routine implicitly called by above

Controlling the Counters

- *PAPI_stop_counters(long_long *vals, int alen)*
Stop counters and put counter values in array
- *PAPI_accum_counters(long_long *vals, int alen)*
Accumulate counters into array and reset
- *PAPI_read_counters(long_long *vals, int alen)*
Copy counter values into array and reset counters
- *PAPI_flops(float *rtime, float *ptime,
 long_long *flpins, float *mflops)*
 - Wallclock time, process time, FP ins since start,
 - Mflop/s since last call

PAPI_flops

- ***int PAPI_flops(float *real_time, float *proc_time, long_long *flpins, float *mflops)***
 - only two calls needed, PAPI_flops before and after the code you want to monitor
 - real_time is the wall-clocktime between the two calls
 - proc_time is the “virtual” time or time the process was actually executing between the two calls (not as fine grained as real_time but better for longer measurements)
 - flpins is the total floating point instructions executed between the two calls
 - mflops is the Mflop/s rating between the two calls

Return codes

Name	Description
PAPI_OK	No error
PAPI_EINVAL	Invalid argument
PAPI_ENOMEM	Insufficient memory
PAPI_ESYS	A system/C library call failed. Check errno variable
PAPI_ESBSTR	Substrate returned an error. E.g. unimplemented feature
PAPI_ECLOST	Access to the counters was lost or interrupted
PAPI_EBUG	Internal error
PAPI_ENOEVNT	Hardware event does not exist
PAPI_ECNFLCT	Hardware event exists, but resources are exhausted
PAPI_ENOTRUN	Event or event set is currently counting
PAPI_EISRUN	Events or event set is currently running
PAPI_ENOEVST	No event set available
PAPI_ENOTPRESET	Argument is not a preset
PAPI_ENOCNTR	Hardware does not support counters
PAPI_EMISC	Any other error occurred

Пример (1)

```
#include <papi.h>
#define NUM_FLOPS 10000
#define NUM_EVENTS 1
main()
{
int Events[NUM_EVENTS] = {PAPI_TOT_INS};
long_long values[NUM_EVENTS];
/* Start counting events */
if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
handle_error(1);
/* Defined in tests/do_loops.c in the PAPI source distribution */
do_flops(NUM_FLOPS);
/* Read the counters */
```

Пример (2)

```
if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);
printf("After reading the counters: %lld\n", values[0]);
do_flops(NUM_FLOPS);
/* Add the counters */
if (PAPI_accum_counters(values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);
printf("After adding the counters: %lld\n", values[0]);
do_flops(NUM_FLOPS);
/* Stop counting events */
if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);
printf("After stopping the counters: %lld\n", values[0]);}
```

Пример

Вывод программы:

- After reading the counters: 441027
- After adding the counters: 891959
- After stopping the counters: 443994