

进阶实验篇第3章

面向cache的优化

矩阵乘法中访存代价较大

程序	总指令数	访存指令	跳转指令	算术指令	控制指令	其他指令
mm_O0.s	81	33	6	29	5	8
mm_O1.s	55	18	10	14	3	10
mm_O2.s	46	16	6	12	3	9

1. 访存指令数占比大
2. 访存指令耗时更长
3. 必须先访存得到数据，才能计算



矩阵数据存于Cache

1

基础知识

2

分块矩阵乘法

3

计算顺序调整

4

性能对比

5

延伸阅读：矩阵分块方式



基础知识

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μ s	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

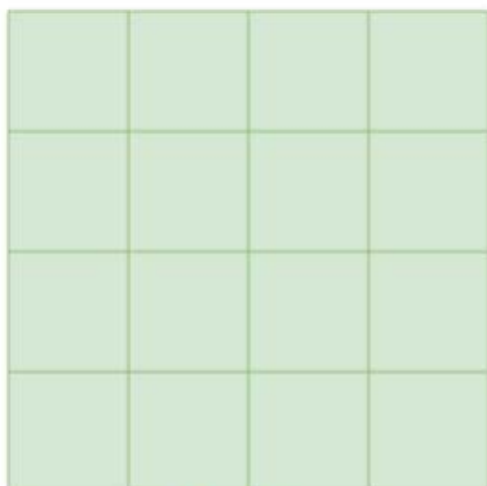
集成在CPU中

法的计算指令中，数
在访存速度最快的寄
。

相比于寄存器慢太多，
加Cache作为缓存，
数据存储其中。

缓存行 (Cacheline)

- cacheline是cache读取数据时的基本单位
- cacheline的大小是固定的，通常为64字节
- 缓存行是一组连续的字节



Cache

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{16}	A_{17}	A_{18}	A_{19}	A_{20}	A_{21}	A_{22}	A_{23}
A_{24}	A_{25}	A_{26}	A_{27}	A_{28}	A_{29}	A_{30}	A_{31}

A

A_0	A_1	A_2	A_3

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7

Cache

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{16}	A_{17}	A_{18}	A_{19}	A_{20}	A_{21}	A_{22}	A_{23}
A_{24}	A_{25}	A_{26}	A_{27}	A_{28}	A_{29}	A_{30}	A_{31}

A

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{16}	A_{17}	A_{18}	A_{19}	A_{20}	A_{21}	A_{22}	A_{23}
A_{24}	A_{25}	A_{26}	A_{27}	A_{28}	A_{29}	A_{30}	A_{31}

A

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

Cache

A_{16}	A_{17}	A_{18}	A_{19}
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

Cache

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{16}	A_{17}	A_{18}	A_{19}	A_{20}	A_{21}	A_{22}	A_{23}
A_{24}	A_{25}	A_{26}	A_{27}	A_{28}	A_{29}	A_{30}	A_{31}

A

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{16}	A_{17}	A_{18}	A_{19}	A_{20}	A_{21}	A_{22}	A_{23}
A_{24}	A_{25}	A_{26}	A_{27}	A_{28}	A_{29}	A_{30}	A_{31}

A



面向cache优化矩阵乘法

面向cache的矩阵乘法优化思路

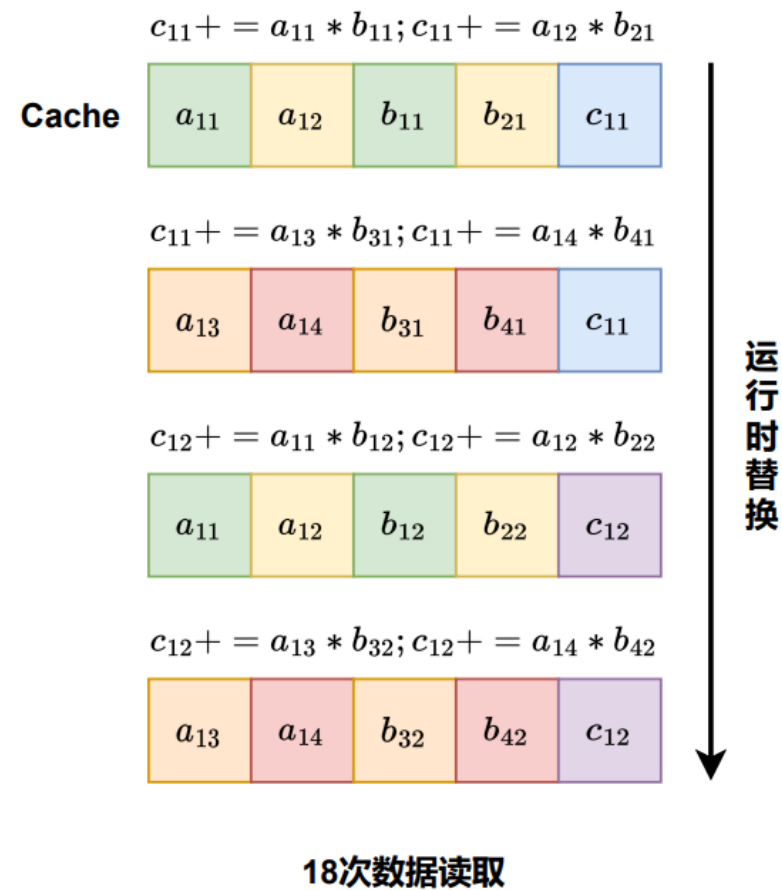
- ▶ 交换循环顺序
- ▶ 分块矩阵乘法

为了利用cache, 希望重复使用的部分能在cache中保持较长时间!

数据太多而Cache太小

$$\begin{array}{|c|c|c|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} \\ \hline a_{21} & a_{22} & a_{23} & a_{24} \\ \hline \end{array}
 \times
 \begin{array}{|c|c|} \hline b_{11} & b_{12} \\ \hline b_{21} & b_{22} \\ \hline b_{31} & b_{32} \\ \hline b_{41} & b_{42} \\ \hline \end{array}
 =
 \begin{array}{|c|c|} \hline c_{11} & c_{12} \\ \hline c_{21} & c_{22} \\ \hline \end{array}$$

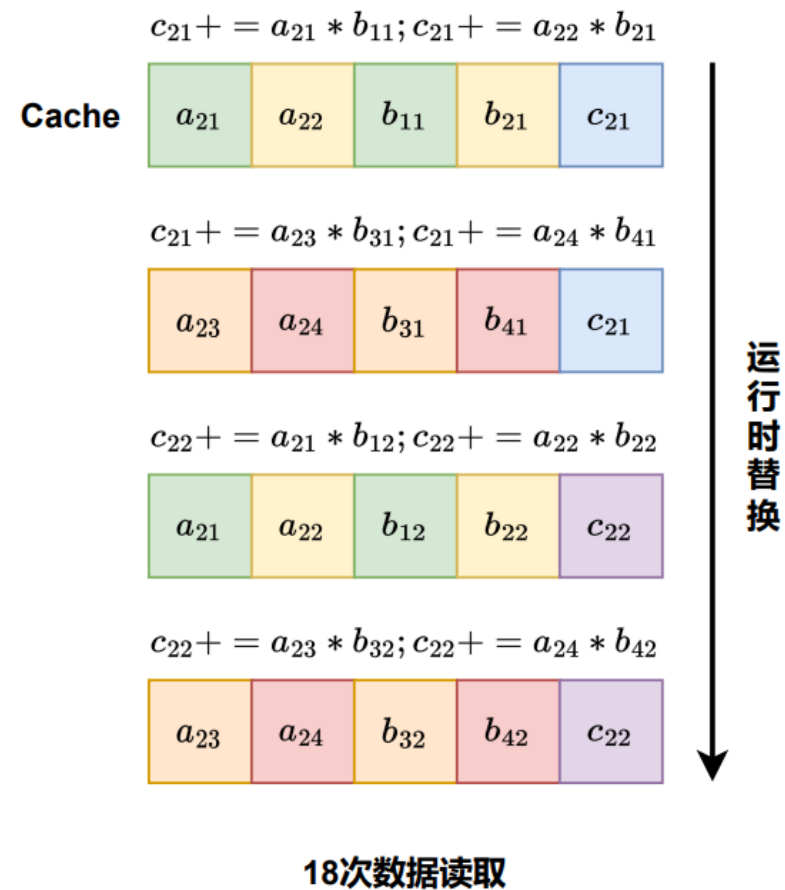
Cache容量限制导致频繁地从内存中读取数据。



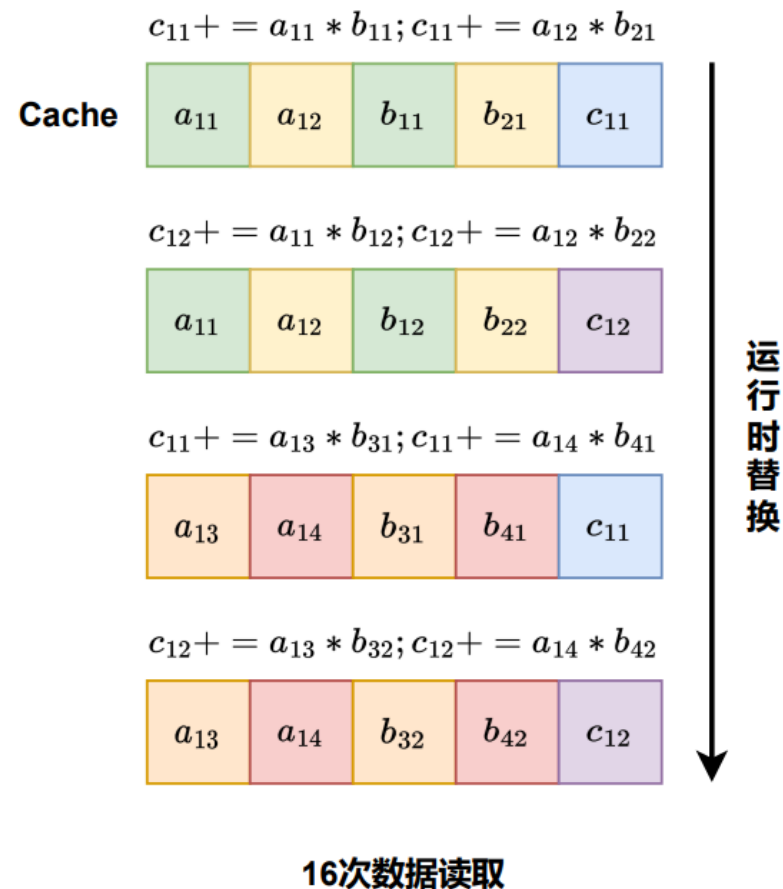
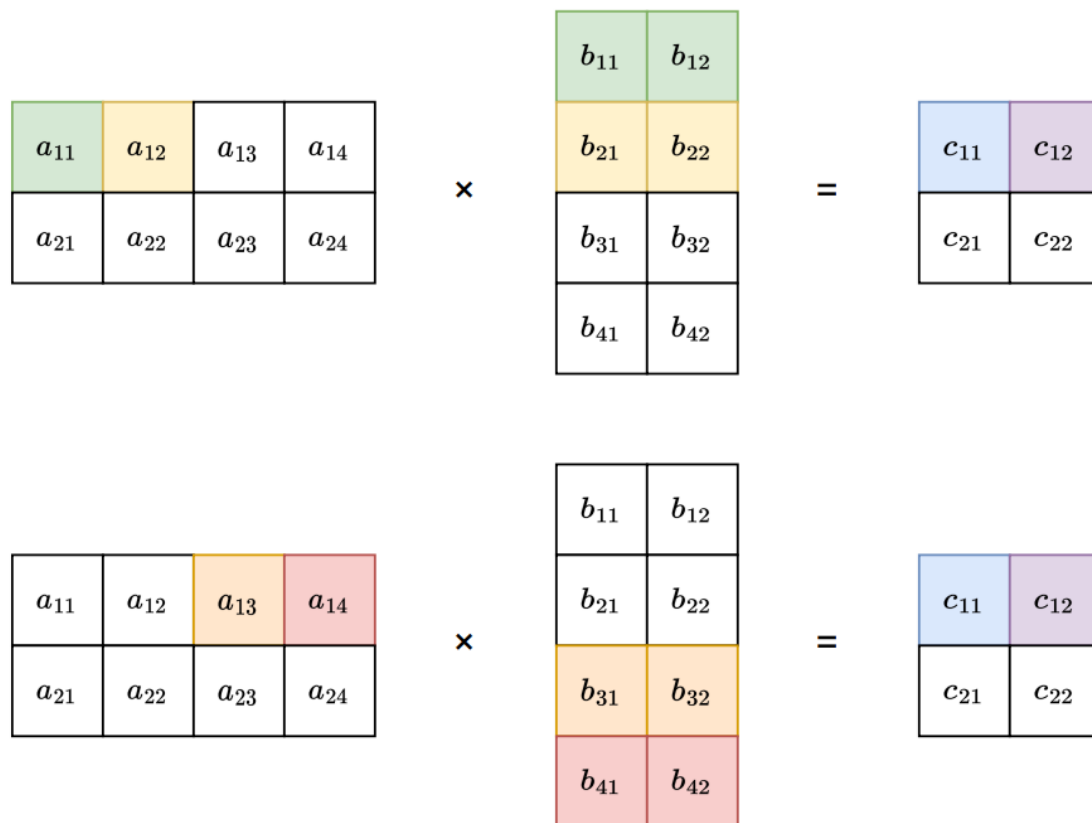
数据太多而Cache太小

$$\begin{array}{|c|c|c|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} \\ \hline a_{21} & a_{22} & a_{23} & a_{24} \\ \hline \end{array}
 \times
 \begin{array}{|c|c|} \hline b_{11} & b_{12} \\ \hline b_{21} & b_{22} \\ \hline b_{31} & b_{32} \\ \hline b_{41} & b_{42} \\ \hline \end{array}
 =
 \begin{array}{|c|c|} \hline c_{11} & c_{12} \\ \hline c_{21} & c_{22} \\ \hline \end{array}$$

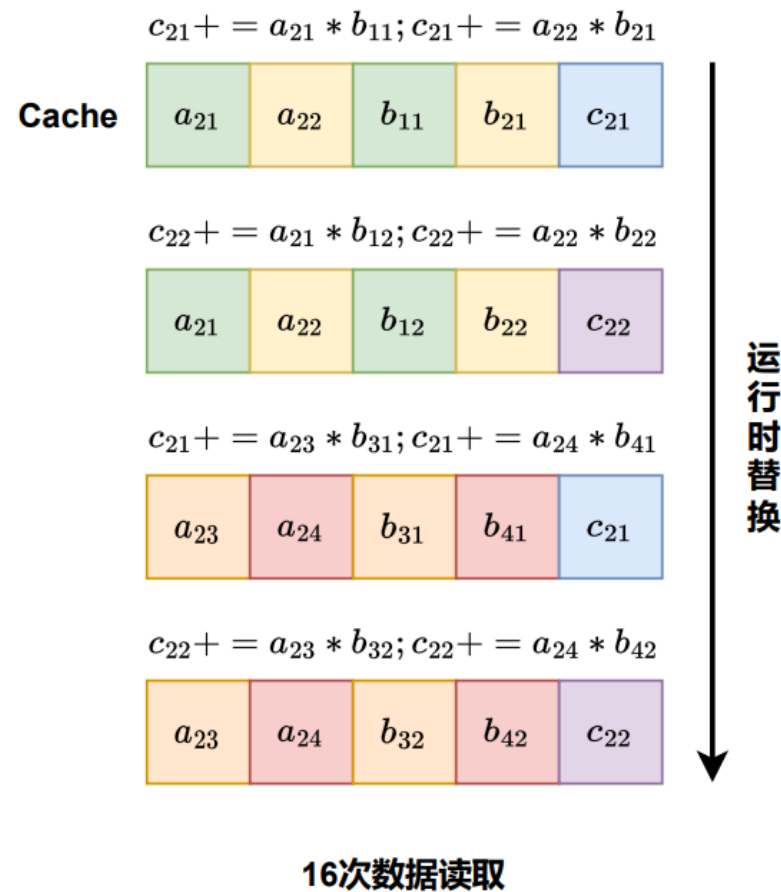
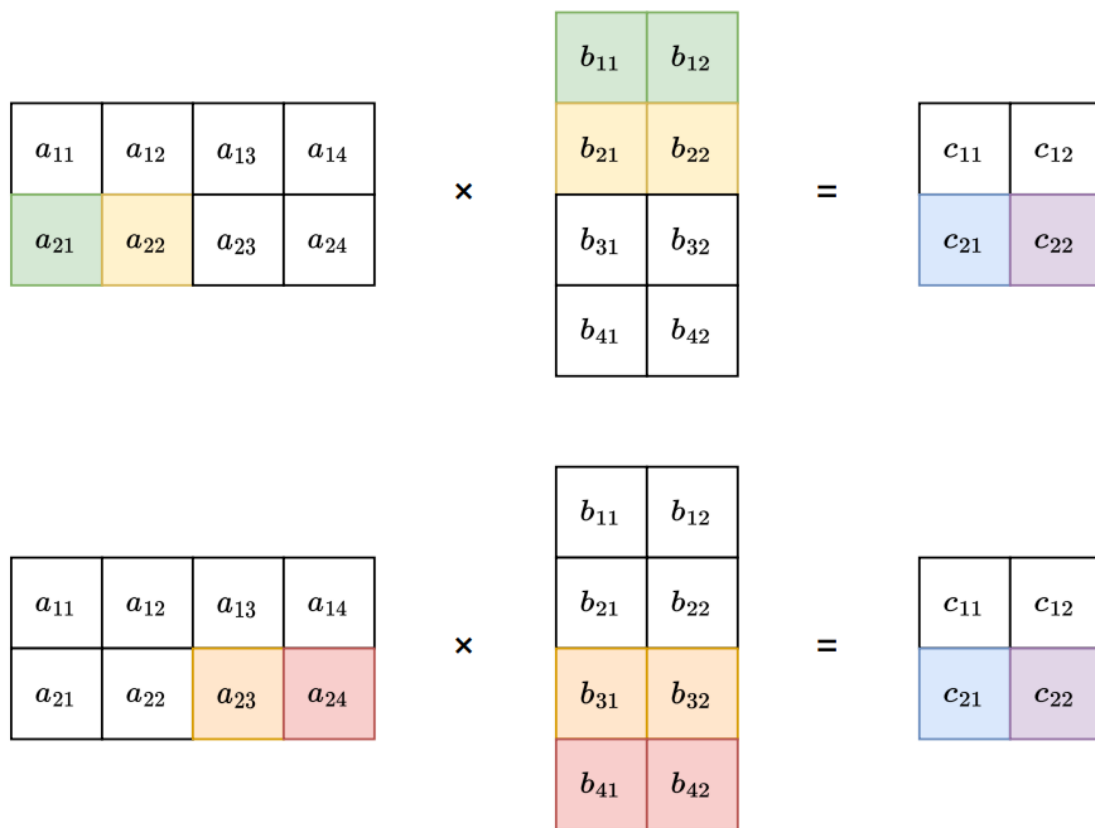
Cache容量限制导致频繁地从内存中读取数据。



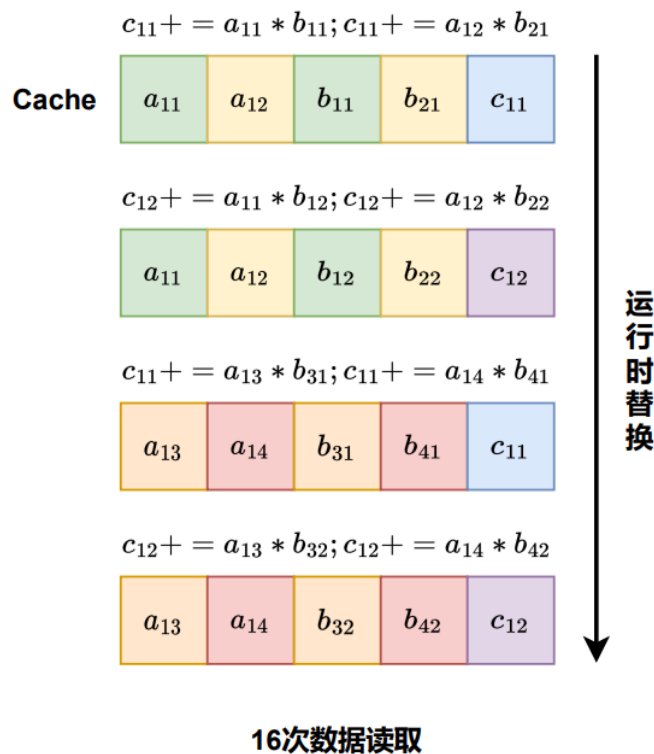
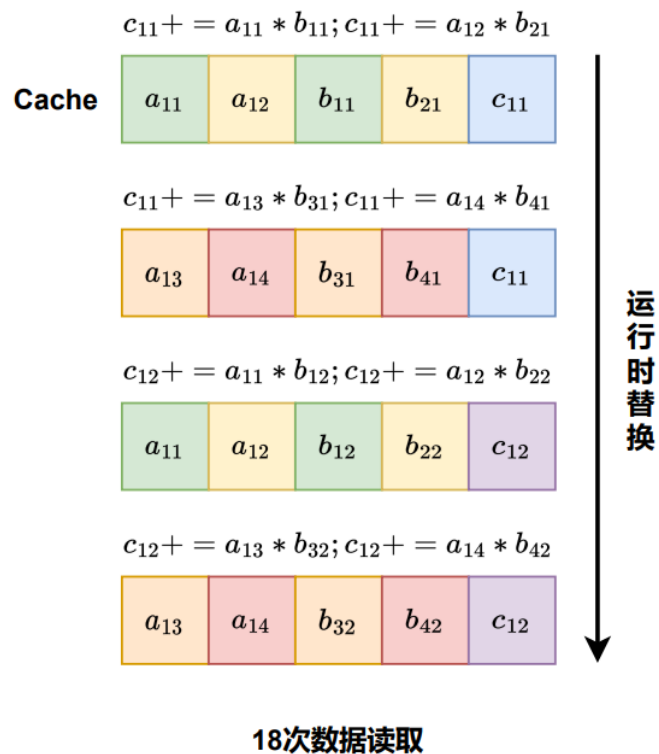
变换一种计算顺序



变换一种计算顺序



计算顺序带来的替换次数差异



调换运算顺序后，结果并未发生改变，但访问内存的次数减少。



矩阵分块



交换循环顺序

普通的矩阵乘法，对cache行为进行模拟

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

四次
cache miss

A_0	A_1	A_2	A_3
B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

A_0	A_1	A_2	A_3
B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

A_0	A_1	A_2	A_3
B_{12}	B_{13}	B_{14}	B_{15}
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

A_0	A_1	A_2	A_3
B_{12}	B_{13}	B_{14}	B_{15}
B_0	B_1	B_2	B_3
B_8	B_9	B_{10}	B_{11}

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

- 读取矩阵A时，每次读取一行发生一次cache miss
- 读取矩阵B的每个元素都会发生一次cache miss
- 每次读取矩阵A的一行时，就对应着要遍历一遍矩阵B
- 所以我们一共发生了 $4 + 16 \times 4 = 68$ 次cache miss

交换计算顺序

```
for (int i = 0; i < N; i++) {  
    for (int j = 0; j < N; j++) {  
        C[i][j] = 0;  
        for (int k = 0; k < N; k++) {  
            C[i][j] += A[i][k] * B[k][j];  
        }  
    }  
}
```

- 原来的计算顺序: i、j、k

```
for (int i = 0; i < N; i++) {  
    for (int k = 0; k < N; k++) {  
        for (int j = 0; j < N; j++) {  
            C[i][j] += A[i][k] * B[k][j];  
        }  
    }  
}
```

- 交换后的计算顺序: i、k、j

交换计算顺序，对cache行为进行模拟

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

A_0	A_1	A_2	A_3
B_0	B_1	B_2	B_3

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

A_0	A_1	A_2	A_3
B_0	B_1	B_2	B_3

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

A_0	A_1	A_2	A_3
B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7

Cache

A_0	A_1	A_2	A_3
A_4	A_5	A_6	A_7
A_8	A_9	A_{10}	A_{11}
A_{12}	A_{13}	A_{14}	A_{15}

A

B_0	B_1	B_2	B_3
B_4	B_5	B_6	B_7
B_8	B_9	B_{10}	B_{11}
B_{12}	B_{13}	B_{14}	B_{15}

B

A ₀	A ₁	A ₂	A ₃
B ₀	B ₁	B ₂	B ₃
B ₄	B ₅	B ₆	B ₇

Cache

A ₀	A ₁	A ₂	A ₃
A ₄	A ₅	A ₆	A ₇
A ₈	A ₉	A ₁₀	A ₁₁
A ₁₂	A ₁₃	A ₁₄	A ₁₅

A

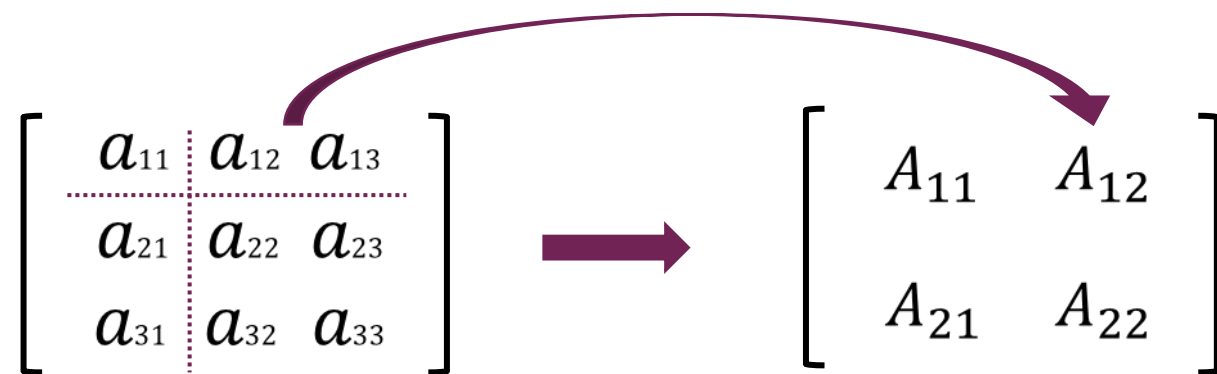
B ₀	B ₁	B ₂	B ₃
B ₄	B ₅	B ₆	B ₇
B ₈	B ₉	B ₁₀	B ₁₁
B ₁₂	B ₁₃	B ₁₄	B ₁₅

B

- 连续读矩阵B的4个元素才会发生一次cache miss
- 访问矩阵A时，每次读新的一行会发生一次cache miss，一共发生4次
- 矩阵B也是每次读取矩阵B的一行时会发生一次cache miss
- 每读一行矩阵A就需要遍历一次矩阵B，即读取矩阵B的4行，所以可以计算得出这种情况下的cache miss数为 $4 + 4 \times 4 = 20$ 次。

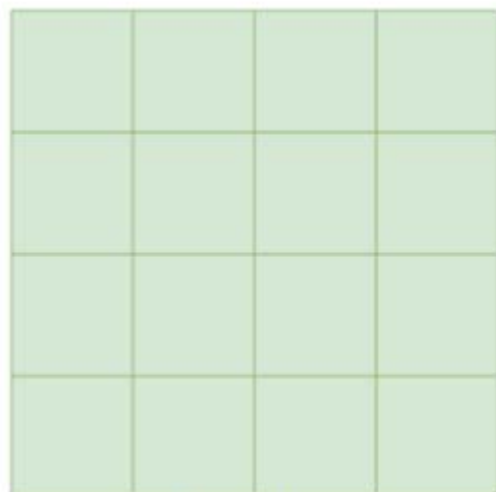
分块矩阵乘法

► 矩阵分块


$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

► 分块乘法

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix}$$



Cache



A



B

- 计算顺序的改变，本质就是对矩阵进行分块，计算小矩阵块之间的矩阵乘法。而加载到Cache中的小矩阵在完成所有可能需要的计算后被替换，避免未来可能的重复加载。



性能对比

计时方法：chrono库，精度可以到纳秒级别

C++11中的chrono库提供了一种方便的方法来处理时间和日期。它具有高度的可移植性和更好的精度。库中主要包含三个组件：duration, time_point和clock。

- duration：用于表示或获取一段时间
- time_point：用于表示或获取一个时刻
- clock：提供了一组时间度量标准，如C++11中提供了3种类型的时钟类，分别为system_clock（系统时钟）、steady_clock（稳定时钟）、high_resolution_clock（高精度时钟）

计时方法示例代码：

```
#include <chrono> // 添加性能测试所需的头文件
using namespace std::chrono; // 高精度时间库命名空间
auto start = high_resolution_clock::now(); // 记录开始时间

// 需要测量执行时间的代码块

auto end = high_resolution_clock::now(); // 记录结束时间
auto duration = duration_cast<nanoseconds>(end - start); // 计算执行时间

cout << "Time: " << duration.count() << " nanoseconds" << endl;
```

性能表示方法: Gops

ops指的是每秒进行的运算的数量

对于矩阵乘法而言, 核心操作是乘法和加法, 因此

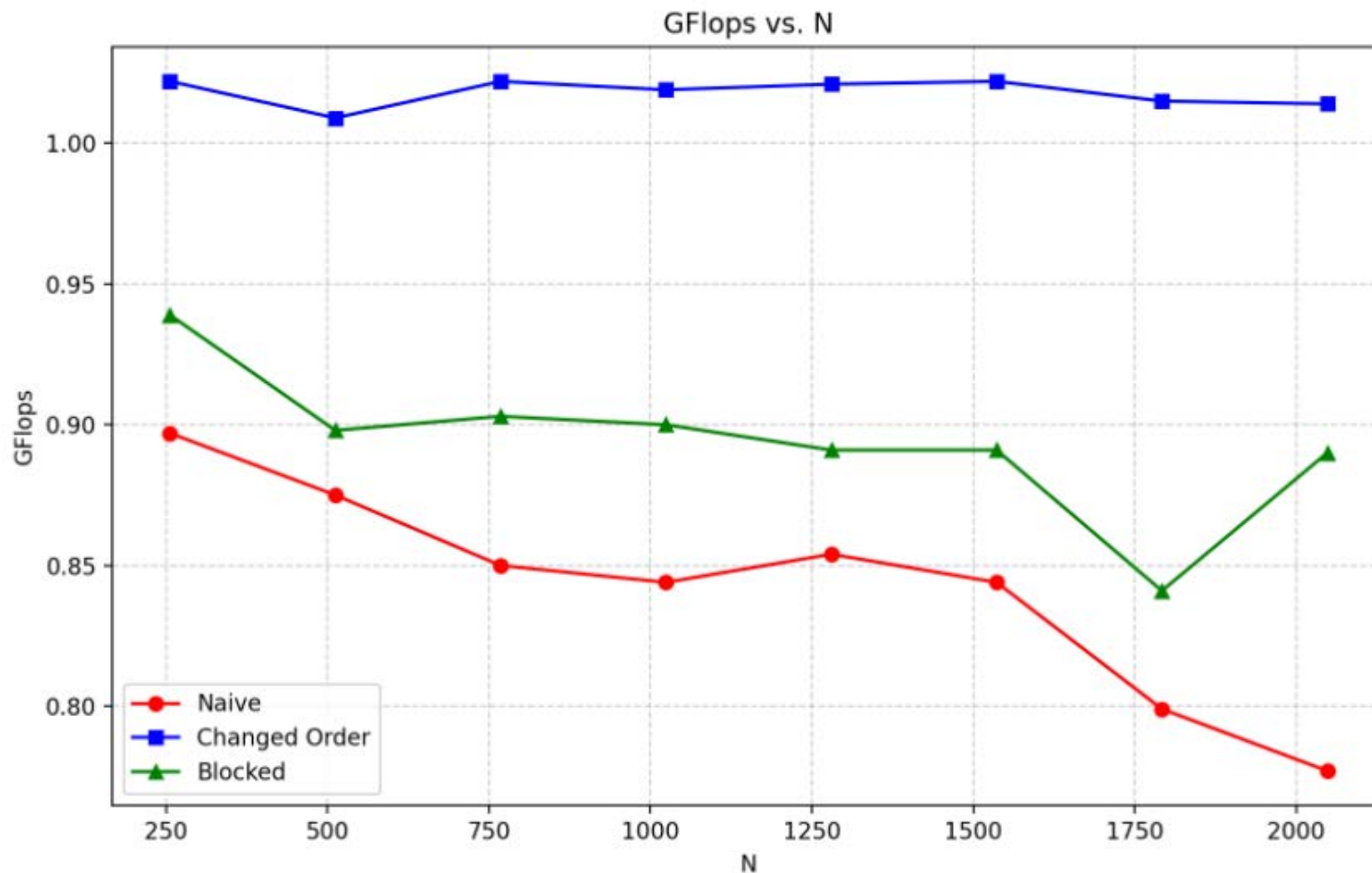
$$\text{ops} = 2 \cdot (m \cdot n \cdot k) / \text{time}$$

因为ops的值通常较大, 因此会使用Gops来表示算力

对于不同尺寸的矩阵, 单纯计量其时间无法直接体现出变化的趋势

因此将所有矩阵的尺寸做归一化处理, 得到其等效的算力, 可以更加方便的对比

在AMD Ryzen 9 7950X CPU上的测试结果



L1 cache: 512K

L2 cache: 16M

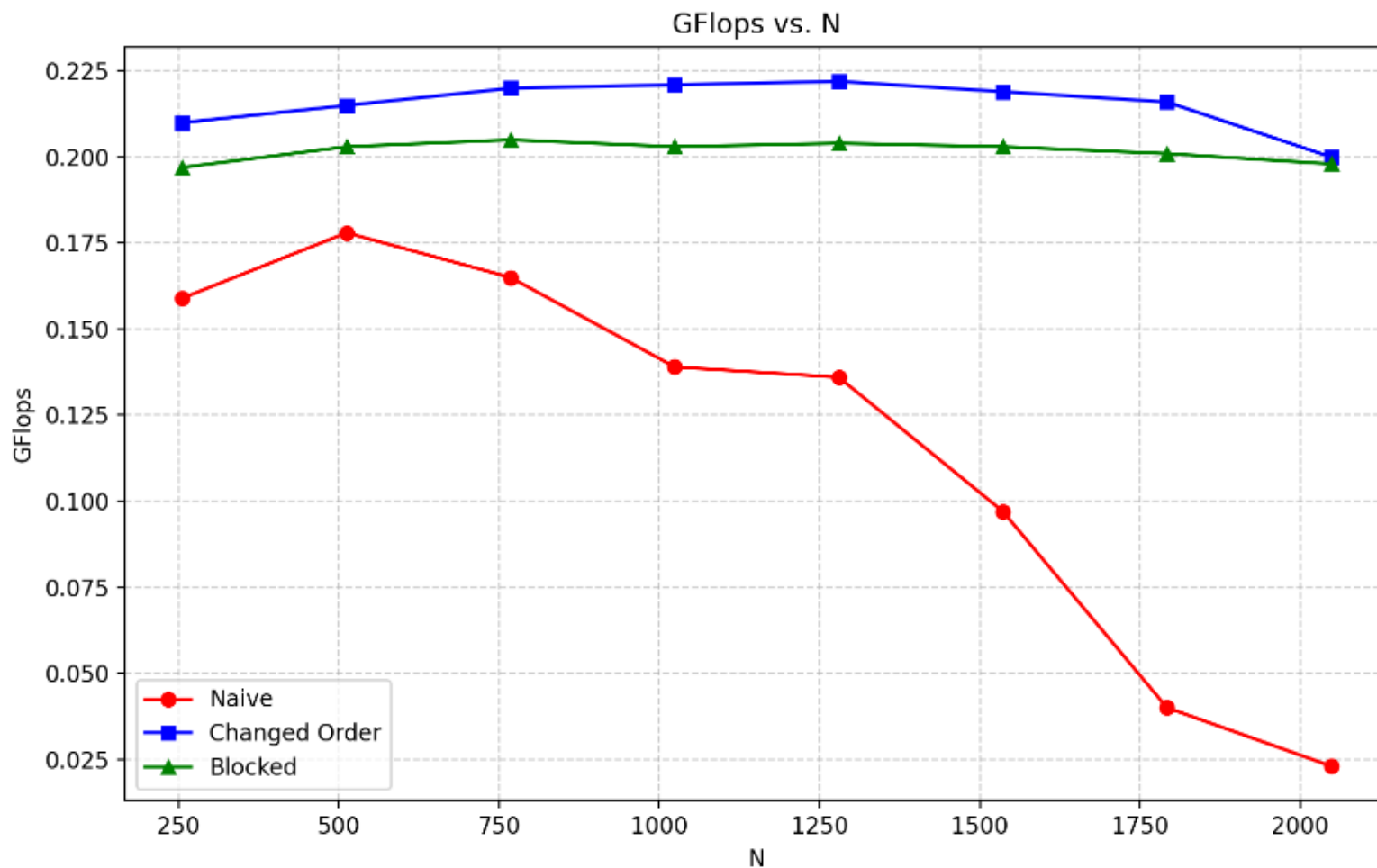
L3 cache: 64M

红线为普通矩阵乘法

蓝线为交换顺序的矩阵乘法

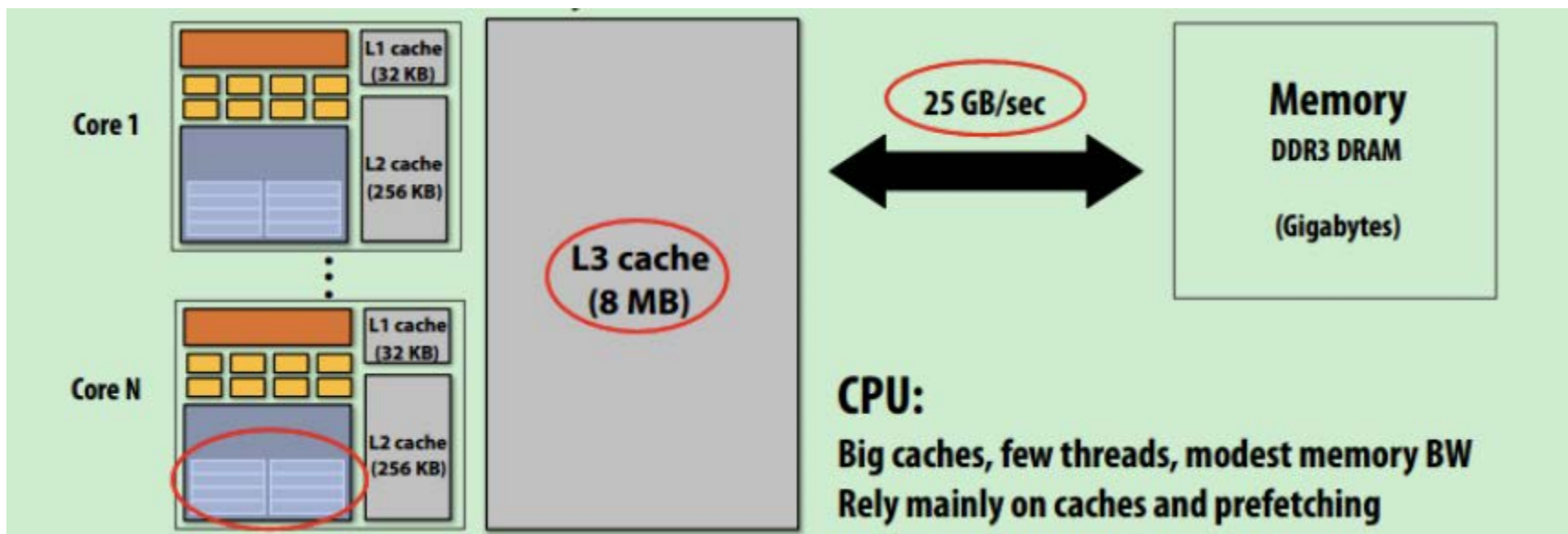
绿线为分块矩阵乘法

在龙芯（Loongson）实验平台上的测试结果



L1 cache: 64K
L2 cache: 256K
L3 cache: 16M

红线为普通矩阵乘法
蓝线为交换顺序的矩阵乘法
绿线为分块矩阵乘法



处理器为什么会停下来不执行指令了？

- 处理器的内存带宽有限、无法满足当前指令的处理器需求
- 处理器的cache容量有限，无法容纳所有的IO请求
- CPU和存储器的速度带差，使得CPU不得不停下来等待数据到达

Perf工具：是Linux系统提供的性能分析工具集，包含多种子工具，能够监测多种硬件及软件性能指标，这些可监测指标称为event，例如：

- cache-references: LLC缓存引用的总次数
 - cache-misses: 访问LLC的未命中次数
 - L1-dcache-load-misses: 从L1 cache中加载数据时的未命中次数
 - L1-dcache-loads: 从L1 cache中加载数据时的总加载次数
 - L1-dcache-stores: 对L1 cache中存储数据时的操作次数
- Linux性能计数器是一个基于内核的子系统，它提供一个性能分析框架，比如硬件（CPU、PMU（Performance Monitoring Unit））功能和软件（软件计数器、tracepoint）功能。
 - 通过perf，应用程序可以利用PMU、tracepoint和内核中的计数器来进行性能统计。使用如下命令可以获取cache miss rate来检测代码是否面向cache得到优化。

```
sudo perf stat -e cache-misses,cache-references ./matrix_mul
```

使用perf命令，得到如下输出：

```
(base) young@Young-Host:~/zhx/MM-main$ sudo perf stat -B -e L1-dcache-load-misses,L1-dcache-loads ./mm_naive
N=1024, blockSize=256
Matrix multiplication without blocking: 2.723768 seconds

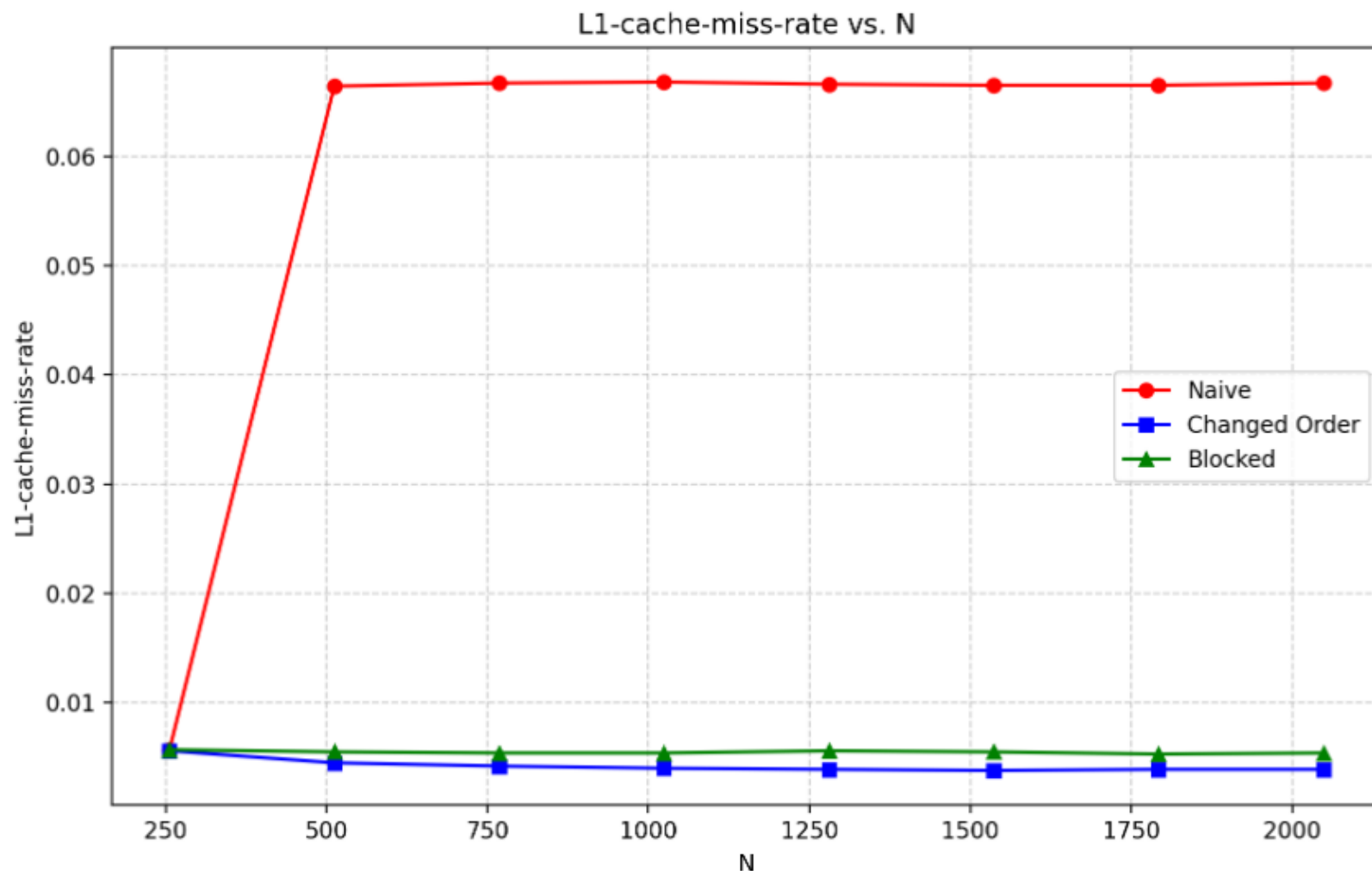
Performance counter stats for './mm_naive':

    1,294,399,379      L1-dcache-load-misses      #    6.68% of all L1-dcache accesses
    19,383,237,341      L1-dcache-loads
    2.741457076 seconds time elapsed

    2.736712000 seconds user
    0.004001000 seconds sys
```

计算 $L1\text{-cache-load-misses}/L1\text{-cache-misses}$ 可以得到我们程序的L1 cache未命中率

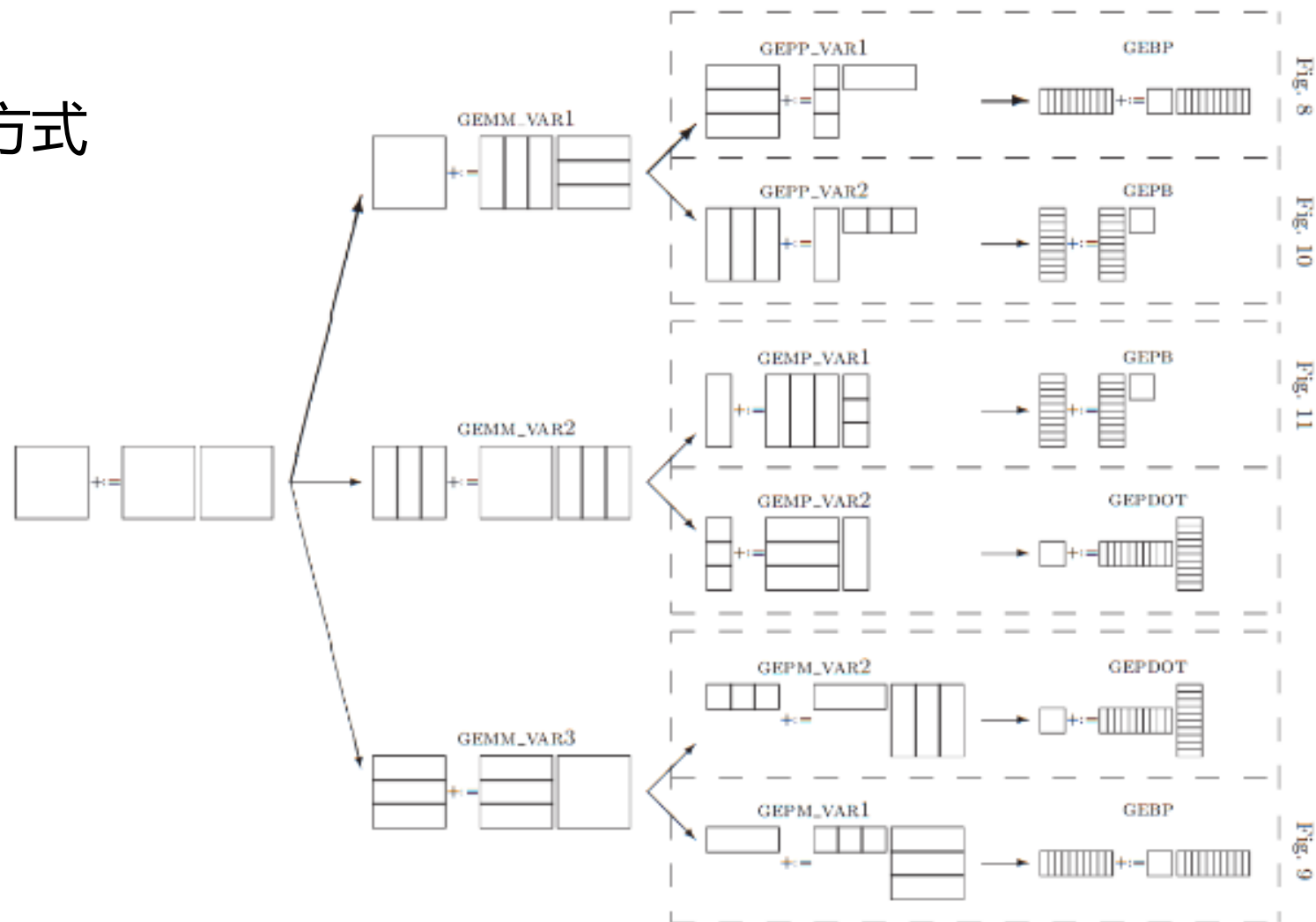
优化后矩阵，cache miss rate明显降低



红线为普通矩阵乘法
蓝线为交换顺序的矩阵乘法
绿线为分块矩阵乘法

延伸阅读：矩阵分块方式

对矩阵乘法的不同分块方式进行了介绍与讨论





本章作业

程序设计

利用C语言编程实现矩阵乘法，并针对cache进行优化

1. 针对不同的矩阵规模，设计适合cache和数据复用的矩阵分块方法
2. 测量数据复用方法对的矩阵乘法运算时间的影响，并使用GOPS来呈现

观察并完成实验报告

1. 使用perf观察不同计算方法引起的cache miss事件



感谢阅读
