Есть ли у вас вопросы?

Краткое содержание этой серии

- Фокусы
- Оптимизация компилятором

Фокус №1

1. Создаю глобальный двухмерный массив
2. Заполняю его случайными числами
3. Вычисляю сумму всех элементов:
   a) `sum += array[i][j]`
   b) `sum += array[j][i]`

На ПК вариант а быстрее почти в 5 раз!

На МК никакой разницы нет.

ПОЧЕМУ?

Фокус №1

А как лежит в памяти двумерный массив?

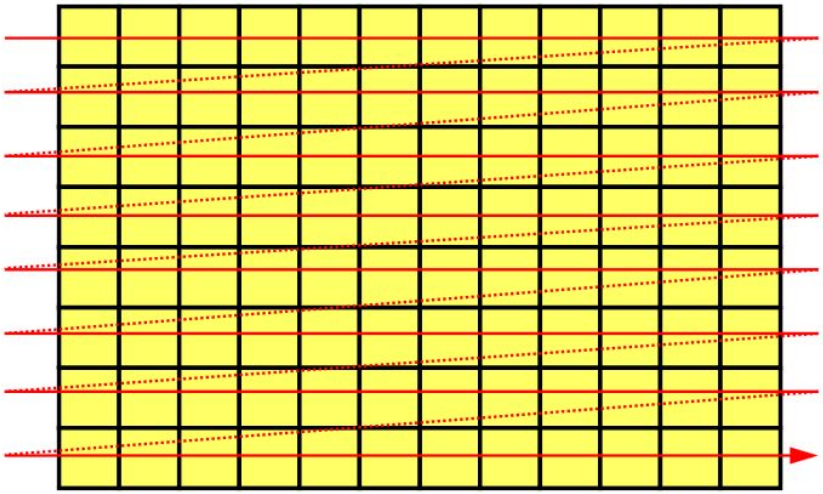
```
uint8_t array[2][4] = { {1,2,3,4}, {5,6,7,8}};
```

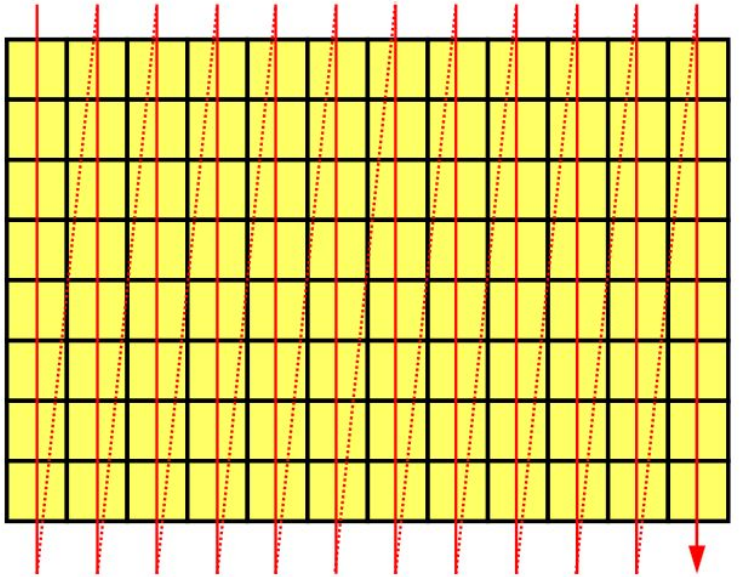
1	2	3	4
5	6	7	8

Точно так же, как одномерный array[8].

1	2	3	4	5	6	7	8

Но почему доступ вдоль строк быстрее, чем вдоль столбцов?





Все дело в кэш-памяти

Зачем нужен кэш?

Чтобы ускорить доступ к часто используемым данным, т.к. оперативная память слишком медленная.

На МК кэш-памяти нет – поэтому нет никакой разницы между вариантами а и б.

А как работает кэш?

Кэш состоит из «линий» (cache lines) -
при каждом обращении в память кэшируется *несколько последовательных байт* (64-128).

Если при обращении в память нужный элемент уже есть в кэше, то все хорошо (кэш-попадание).

Если нужного элемента в кэше нет – нужно пойти в память и считать линию (кэш-промах).

Кэш не бесконечен! Поэтому чтобы записать в него новую линию, нужно стереть старую.

Кэш

Вывод?

Последовательный доступ к памяти гораздо быстрее, чем случайный.

С точки зрения железа самая быстрая структура данных – обычный массив (на не слишком большом количестве данных).

Кэш

В современных процессорах есть:

- кэш данных (D-cache)
- кэш инструкций (I-cache)
- буфер ассоциативной трансляции (TLB)

Как правило, существует несколько уровней кэша.

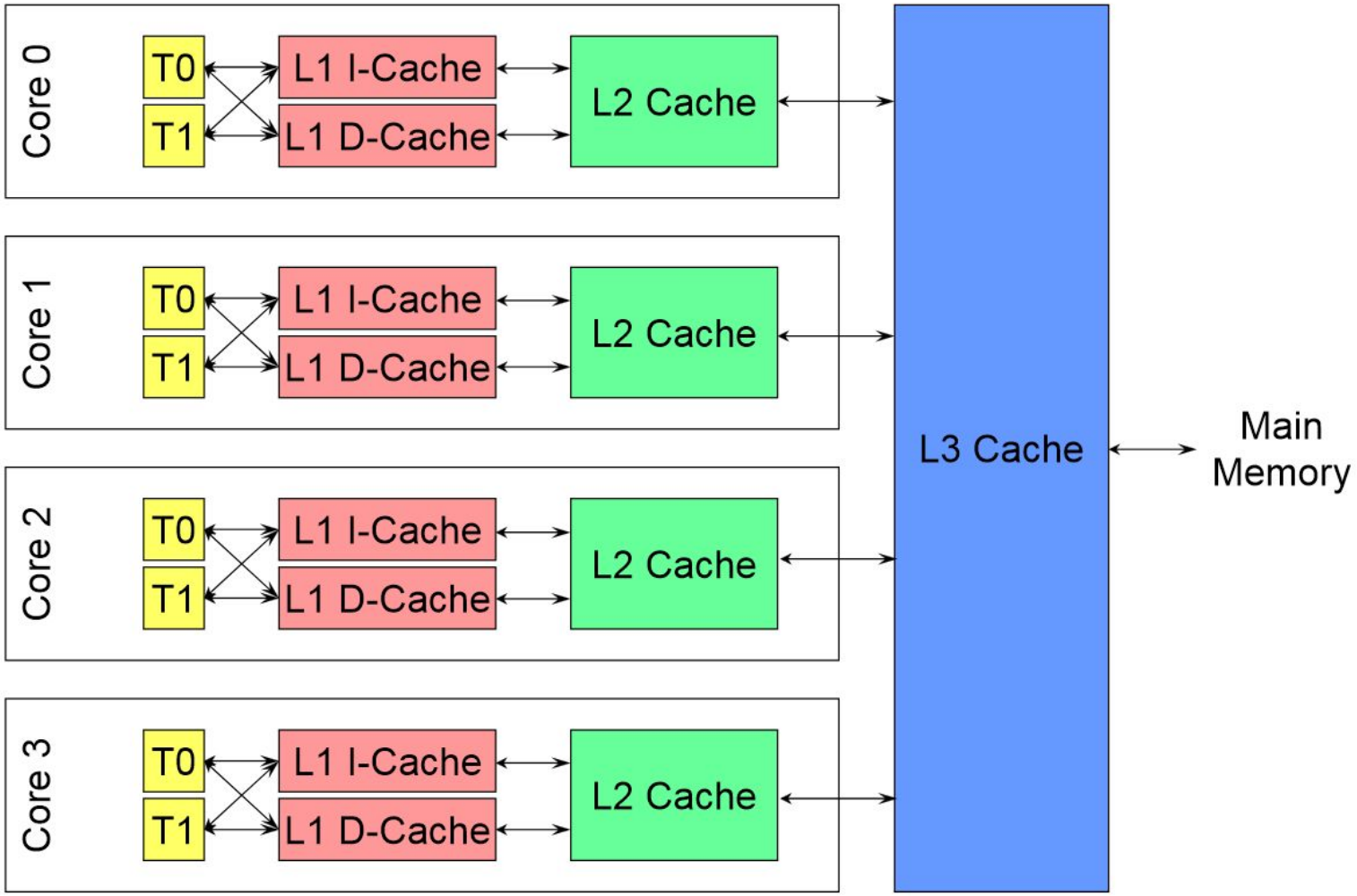
Кэш в современном процессоре

Intel Core i7-9xx:

L1 cache: 64 KiB

L2 cache: 256 KiB

L3 cache: 8 MiB



Кэш в современном процессоре

Время чтения из памяти для Core i7-9xx:

- L1 - 4 такта.
- L2 - 11 тактов.
- L3 - 39 тактов.
- Основная ОЗУ – 107 тактов.

Кэш

Допустим, что два ядра процессора обращаются к одной и той же переменной.

Тогда соответствующий кусок памяти будет закэширован дважды в двух кэшах L1.

А что будет, если одно ядро что-нибудь в эту переменную запишет?

Что тогда прочитает другое ядро?

Если доступ к переменной организован *правильно,* то все будет в порядке. Для программиста кэш в этом смысле «прозрачен».

Но за это придется платить скоростью работы..

Кэш

Допустим, что два ядра процессора обращаются к двум разным переменным, которые расположены в памяти *рядом.*

Одна и та же кэш-линия опять-таки будет находится в двух кэшах.

Прозрачность кэша гарантирует, что значения переменных будут корректными.

Но для этого при каждой записи эта линия будет записываться в основную память и читаться опять! И скорость работы программы упадет.

Это называется «false sharing» (ложное разделение памяти).

Фокус №1.5

Возьмем неудачный способ сложения элементов массива (по столбцам).

Логично предположить, что чем больше массив – тем больше времени занимает его обход.

Массив 4100x4100 обходится быстрее чем 4096x4096.

Степени двойки – это плохо?

Ассоциативность кэша

А как узнать, закэширована переменная или нет?

- Кэш прямого отображения - каждый адрес памяти может быть закэширован в одно, заранее определенное место в кэше.
  - Легко подвергается конфликтам.
- Полностью ассоциативный кэш – любая переменная может быть закэширована в любой участок кэша.
  - Очень сложная реализация.
- Частично ассоциативный кэш – каждая переменная может находится в нескольких, заранее определенных участках кэша.
  - Компромисс, используется на практике.

Частично ассоциативный кэш

Например, 16-входовой частично ассоциативный кэш – линии кэша делятся на 16 групп.

Каждая переменная входит в одну группу и может входить только в линии кэша из этой группы.

Номер группы, как правило, определяется *адресом переменной.*

Переменные с адресами, кратными определенному числу, будут входить в одну группу и соревноваться за одни и те же линии кэша!

Кэш для инструкций

- Линейный код (без переходов) выполняется быстрее
- Маленькие программы (которые целиком помещаются в кэш) выполняются быстрее

Выводы

- При оценке быстродействия алгоритма нужно помнить про кэш.
- Писать быстродействующие программы – это сложно.
- Тестировать быстродействие – это сложно (разные процессоры, разные входные данные, «прогрев» кэша..).

Фокус №2

Вариант А:

- Заполним одномерный массив случайными элементами.
- Много раз найдем сумму всех элементов больше 128.

Вариант Б:

- Заполним одномерный массив случайными элементами.
- Отсортируем массив
- Много раз найдем сумму всех элементов больше 128.

На МК вариант Б занимает больше времени.

На ПК вариант Б занимает существенно меньше

Предсказание переходов

Ключевой момент:

```
if (data[c] >= 128)
                sum += data[c];
```

Если массив отсортирован – то переходы очень предсказуемы, предсказатель редко ошибается.

Если массив не отсортирован – предсказатель ошибается постоянно!

Оптимизация

Критерии оптимизации:

- по объему кода (бинарного файла)
- по скорости исполнения

Иногда можно (и хочется) оптимизировать сразу по двум критериям, но не всегда.

Оптимизация «на пальцах»

У компилятора есть некая «область просмотра» (scope), в пределах которой он оптимизирует код:

- одна строка
- несколько строк, цикл
- функция
- файл
- весь проект

Грубо говоря, чем больше эта область, тем лучше оптимизация.