Параллельное программирование и суперкомпьютерный кодизайн

Смирнов А.В. asmirnov@srcc.msu.ru

Раздел 4. Параллельное программирование на одном вычислительном ядре

Раздел 4. План.

Параллельное программирование на одном вычислительном ядре

- Ускорение суммирование вектора при помощи SSE инструкций
- Выравнивание в памяти и способы его достижения
- Ускорение суммирование вектора при помощи AVX инструкций и задержки памяти
- Кэш-ориентированный код

Раздел 4. План.

Параллельное программирование на одном вычислительном ядре

- **Ускорение суммирование вектора при помощи SSE инструкций**
- Выравнивание в памяти и способы его достижения
- Ускорение суммирование вектора при помощи AVX инструкций и задержки памяти
- Кэш-ориентированный код

Рассматриваемая задача

- Изначально планировалось продемонстрировать все на задаче сложения двух векторов

Рассматриваемая задача

- Изначально планировалось продемонстрировать все на задаче сложения двух векторов
- Но выяснилось в ходе тестов, что не ней не получается показать преимущество ручного подхода

Рассматриваемая задача

- Изначально планировалось продемонстрировать все на задаче сложения двух векторов
- Но выяснилось в ходе тестов, что не ней не получается показать преимущество ручного подхода
- Поэтому мы рассмотрим задачу суммирования вектора – сложить числа с 1 по n

Простая реализация

```
double sum_vector(std::vector<double>& a,
  size_t n) {
  double res = 0.;
  for (size_t i = 0; i!=n; ++i) {res += a[i];}
}
void main() {
  constexpr size_t n = 256 * 1024 * 1024;
  std::vector a(n);
  std::mt19937 gen(0);
  std::uniform_real_distribution<double>
      uid(-1., 1.);
  std::generate(a.begin(),a.end(),
      [&uid,&gen]()->double {return uid(gen);}
  );
  sum_vector(a, n);
}
```

Простая программа сложения векторов

Мы будем компилировать программу и смотреть на получаемый код на ассемблере

- Команда компиляции: `g++ -g -O3 -S -fverbose-asm program.cpp`

Простая программа сложения векторов

Мы будем компилировать программу и смотреть на получаемый код на ассемблере

- Команда компиляции: `g++ -g -O3 -S -fverbose-asm program.cpp`
- На выходе получается файл program.s в диалекте AT&T

Простая программа сложения векторов

Мы будем компилировать программу и смотреть на получаемый код на ассемблере

- Команда компиляции: `g++ -g -O3 -S -fverbose-asm program.cpp`
- На выходе получается файл program.s в диалекте AT&T
- Если же убрать `-S`, то это обычная компиляция

Простая реализация с -O1

```
        pxor    %xmm0, %xmm0    # <retval>
.L18:
        addsd   (%rax), %xmm0
        addq    $8, %rax
        cmpq    %rdx, %rax
        jne     .L18    #,
        ret
```

Простая реализация с -O1

```
        pxor    %xmm0, %xmm0    # <retval>
.L18:
        addsd   (%rax), %xmm0
        addq    $8, %rax
        cmpq    %rdx, %rax
        jne     .L18    #,
        ret
```

- rax это регистр, хранит a + i

Простая реализация с -O1

```
        pxor    %xmm0, %xmm0    # <retval>
.L18:
        addsd   (%rax), %xmm0
        addq    $8, %rax
        cmpq    %rdx, %rax
        jne     .L18    #,
        ret
```

- rax это регистр, хранит a + i
- (%rax) это переход по адресу

Простая реализация с -O1

```
        pxor    %xmm0, %xmm0    # <retval>
.L18:
        addsd   (%rax), %xmm0
        addq    $8, %rax
        cmpq    %rdx, %rax
        jne     .L18    #,
        ret
```

- rax это регистр, хранит a + i
- (%rax) это переход по адресу
- регистр xmm0 накапливает результат

Простая реализация с -O1

```
        pxor    %xmm0, %xmm0    # <retval>
.L18:
        addsd   (%rax), %xmm0
        addq    $8, %rax
        cmpq    %rdx, %rax
        jne     .L18    #,
        ret
```

- rax это регистр, хранит a + i
- (%rax) это переход по адресу
- регистр xmm0 накапливает результат
- время 236 ms

Простая реализация с -O3

```
        pxor    %xmm0, %xmm0    # <retval>
.L28:
        movsd   (%rax), %xmm1
        addq    $16, %rax
        addsd   %xmm1, %xmm0
        movsd   -8(%rax), %xmm1
        addsd   %xmm1, %xmm0
        cmpq    %rax, %rdx
        jne     .L28    #,
```

Простая реализация с -O3

```
        pxor    %xmm0, %xmm0    # <retval>
.L28:
        movsd   (%rax), %xmm1
        addq    $16, %rax
        addsd   %xmm1, %xmm0
        movsd   -8(%rax), %xmm1
        addsd   %xmm1, %xmm0
        cmpq    %rax, %rdx
        jne     .L28    #,
```

- компилятор сгенерировал код, который складывает последовательно пары чисел, сначала они загружаются в память, затем прибавляются к регистру накопления. Попытка скрыть задержки обращения к памяти

Простая реализация с -O3

```
        pxor    %xmm0, %xmm0    # <retval>
.L28:
        movsd   (%rax), %xmm1
        addq    $16, %rax
        addsd   %xmm1, %xmm0
        movsd   -8(%rax), %xmm1
        addsd   %xmm1, %xmm0
        cmpq    %rax, %rdx
        jne     .L28    #,
```

- компилятор сгенерировал код, который складывает последовательно пары чисел, сначала они загружаются в память, затем прибавляются к регистру накопления. Попытка скрыть задержки обращения к памяти
- время 222 ms

Реализация сложения с SSE

```
auto av = reinterpret_cast<__m128d *>(
    a.data() + start);
__m128d resv = _mm_setzero_pd();
for (size_t i = start;i != end; i+=2) {
    resv = _mm_add_pd(resv, *av);
    ++av;
}
double res[2];
_mm_store_pd(res, resv);
return res[0]+res[1];
```

Реализация сложения с SSE

```
auto av = reinterpret_cast<__m128d *>(
        a.data() + start);
__m128d resv = _mm_setzero_pd();
for (size_t i = start;i != end; i+=2) {
        resv = _mm_add_pd(resv, *av);
        ++av;
}
double res[2];
_mm_store_pd(res, resv);
return res[0]+res[1];
```

- Мы заводим переменную типа `__m128d *` – указатель на пары даблов. Подобные переменные годятся для использования SSE инструкций

Реализация сложения с SSE

```
auto av = reinterpret_cast<__m128d *>(
        a.data() + start);
__m128d resv = _mm_setzero_pd();
for (size_t i = start;i != end; i+=2) {
        resv = _mm_add_pd(resv, *av);
        ++av;
}
double res[2];
_mm_store_pd(res, resv);
return res[0]+res[1];
```

- Мы заводим переменную типа `__m128d *` – указатель на пары даблов. Подобные переменные годятся для использования SSE инструкций
- Продвигаемся по циклу парами (n четное), в результате первая часть resv содержит сумму нечетных элементов, а вторая часть – сумму четных

Реализация сложения с SSE

```
        pxor    %xmm0, %xmm0    # resv
.L26
        addpd   (%rax,%rcx,8), %xmm0
        addq    $2, %rcx        # i
        cmpq    %rcx, %r8       # n
        jne     .L26
```

Реализация сложения с SSE

```
        pxor    %xmm0, %xmm0    # resv
.L26
        addpd   (%rax,%rcx,8), %xmm0
        addq    $2, %rcx        # i
        cmpq    %rcx, %r8       # n
        jne     .L26
```

- Получившийся код использует инструкцию addpd – add pair double. Она складывает пары чисел, хранящихся в регистре xmm. Хотя в прошлом примере и использовался этот же регистр, на деле использовалась его младшая часть. Этот регистр 128-битный

Реализация сложения с SSE

```
        pxor    %xmm0, %xmm0    # resv
.L26
        addpd   (%rax,%rcx,8), %xmm0
        addq    $2, %rcx        # i
        cmpq    %rcx, %r8       # n
        jne     .L26
```

- Получившийся код использует инструкцию addpd – add pair double. Она складывает пары чисел, хранящихся в регистре xmm. Хотя в прошлом примере и использовался этот же регистр, на деле использовалась его младшая часть. Этот регистр 128-битный
- здесь rax это a, rcx это i

Реализация сложения с SSE

```
        pxor    %xmm0, %xmm0    # resv
.L26
        addpd   (%rax,%rcx,8), %xmm0
        addq    $2, %rcx        # i
        cmpq    %rcx, %r8       # n
        jne     .L26
```

- Получившийся код использует инструкцию addpd – add pair double. Она складывает пары чисел, хранящихся в регистре xmm. Хотя в прошлом примере и использовался этот же регистр, на деле использовалась его младшая часть. Этот регистр 128-битный
- здесь rax это a, rcx это i
- время 135 ms – мы существенно ускорились

Почему компилятор это не сделал?

Почему компилятор это не сделал?

- Фактически мы исказили порядок суммирования. Мы отдельно складываем нечетные элементы, отдельно четные и берем сумму. Но разве сложение не ассоциативно?

Почему компилятор это не сделал?

- Фактически мы исказили порядок суммирования. Мы отдельно складываем нечетные элементы, отдельно четные и берем сумму. Но разве сложение не ассоциативно?
- Чему равна сумма 1 + 1/2 + 1/3 + 1/4 + ...

Почему компилятор это не сделал?

- Фактически мы исказили порядок суммирования. Мы отдельно складываем нечетные элементы, отдельно четные и берем сумму. Но разве сложение не ассоциативно?
- Чему равна сумма 1 + 1/2 + 1/3 + 1/4 + ...
- Прибавление маленького числа к большому может не изменять большое

Почему компилятор это не сделал?

- В нашем примере переполнения не происходит

```
constexpr size_t n = 256 * 1024 * 1024;
std::vector a(n);
std::mt19937 gen(0);
std::uniform_real_distribution<double>
    uid(-1., 1.);
std::generate(a.begin(),a.end(),
    [&uid,&gen]()->double {return uid(gen);}
);
```

А можно работать больше, чем с парами чисел?

- Здесь мы использовали SSE инструкции. А можно больше?

А можно работать больше, чем с парами чисел?

- Здесь мы использовали SSE инструкции. А можно больше?
- Да. Большинство современных процессоров поддерживает также набор инструкций AVX, работающий с четверками пар чисел. А некоторые процессоры и AVX2, работающие с восьмерками пар чисел.

А можно работать больше, чем с парами чисел?

- Здесь мы использовали SSE инструкции. А можно больше?
- Да. Большинство современных процессоров поддерживает также набор инструкций AVX, работающий с четверками пар чисел. А некоторые процессоры и AVX2, работающие с восьмерками пар чисел.
- Но здесь возникает осложнение – наборы чисел для этого должны быть правильно выровнены в памяти. Этому посвящен следующий блок

Вопросы?

Раздел 4. План.

Параллельное программирование на одном вычислительном ядре

- Ускорение суммирование вектора при помощи SSE инструкций
- **Выравнивание в памяти и способы его достижения**
- Ускорение суммирование вектора при помощи AVX инструкций и задержки памяти
- Кэш-ориентированный код

Выравнивание памяти

- Каждый объект в памяти имеет адрес, и этот адрес – тоже число. Выравнивание в памяти определяется кратностью этого числа.

Выравнивание памяти

- Каждый объект в памяти имеет адрес, и этот адрес – тоже число. Выравнивание в памяти определяется кратностью этого числа.

```
int num = 21;
size_t addr =
    reinterpret_cast<size_t>(&num);
printf("%zx\n", addr);
```

Выравнивание памяти

- Каждый объект в памяти имеет адрес, и этот адрес – тоже число. Выравнивание в памяти определяется кратностью этого числа.

```
int num = 21;
size_t addr =
    reinterpret_cast<size_t>(&num);
printf("%zx\n", addr);
```

- 7fff61eee60c – число будет отличаться, но ключевое здесь то, на что оно заканчивается. c = 12, что означает кратность в 4 байта для int, но не 4 (могло и оказаться кратным 2 или 8, но это случайно).

Выравнивание памяти

- Каждый объект в памяти имеет адрес, и этот адрес – тоже число. Выравнивание в памяти определяется кратностью этого числа.

```
int num = 21;
size_t addr =
    reinterpret_cast<size_t>(&num);
printf("%zx\n", addr);
```

- 7fff61eee60c – число будет отличаться, но ключевое здесь то, на что оно заканчивается. c = 12, что означает кратность в 4 байта для int, но не 4 (могло и оказаться кратным 2 или 8, но это случайно).
- Аналогично можно проверить другие типы. Размер double – 8 байт = 64 бита, и кратность выравнивания тоже.

Выравнивание памяти

- Если мы просто берем `std::vector<double>(n)`, то внутри начиная с поля `data()` лежит массив их double, но его выравнивание совпадает с выравниванием double

Выравнивание памяти

- Если мы просто берем `std::vector<double>(n)`, то внутри начиная с поля `data()` лежит массив их double, но его выравнивание совпадает с выравниванием double
- В то же время avx-инструкции требуют выравнивания по 256 бит или 32 байта

Выравнивание памяти

- Если мы просто берем `std::vector<double>(n)`, то внутри начиная с поля `data()` лежит массив их double, но его выравнивание совпадает с выравниванием double
- В то же время avx-инструкции требуют выравнивания по 256 бит или 32 байта
- Как добиться повышенного выравнивания в массиве? Существуют различные способы

Способы повышенного выравнивания в массиве

Существуют различные способы добиться повышенного выравнивания в массиве, все со своими преимуществами и недостатками

- Взятие большего размера и отсчет не с нуля
- aligned_alloc
- aligned new
- Кастомные аллокаторы для контейнеров

Способы повышенного выравнивания в массиве

Существуют различные способы добиться повышенного выравнивания в массиве, все со своими преимуществами и недостатками

- **Взятие большего размера и отсчет не с нуля**
- aligned_alloc
- aligned new
- Кастомные аллокаторы для контейнеров

Взятие большего размера и отсчет не с нуля

- Возьмем увеличенный вектор

Взятие большего размера и отсчет не с нуля

- Возьмем увеличенный вектор

```
std::vector<double> vec(n+3);
size_t addr =
   reinterpret_cast<size_t>(&vec[0]);
size_t start = (addr % 32) / 8;
if (start > 0) start = 4 - start;
std::cout<<start<<std::endl;
```

Взятие большего размера и отсчет не с нуля

- Возьмем увеличенный вектор

```
std::vector<double> vec(n+3);
size_t addr =
  reinterpret_cast<size_t>(&vec[0]);
size_t start = (addr % 32) / 8;
if (start > 0) start = 4 - start;
std::cout<<start<<std::endl;
```

- Далее обращаемся как `vec[start + i]` вместо `vec[i]`

Взятие большего размера и отсчет не с нуля

- Плюсы: не требуется никаких дополнительных конструкций, работает и в старых стандартах C++

Взятие большего размера и отсчет не с нуля

- Плюсы: не требуется никаких дополнительных конструкций, работает и в старых стандартах C++
- Минусы: нужно где-то хранить этот стартовый отступ и не забывать его передавать во все функции, работающие с вектором, меняется семантика

Взятие большего размера и отсчет не с нуля

- Плюсы: не требуется никаких дополнительных конструкций, работает и в старых стандартах C++
- Минусы: нужно где-то хранить этот стартовый отступ и не забывать его передавать во все функции, работающие с вектором, меняется семантика
- Нейтрально: может быть потрачено 3 лишних байта, в большинстве случаев это не страшно

Способы повышенного выравнивания в массиве

Существуют различные способы добиться повышенного выравнивания в массиве, все со своими преимуществами и недостатками

- Взятие большего размера и отсчет не с нуля
- **aligned_alloc**
- aligned new
- Кастомные аллокаторы для контейнеров

aligned_alloc

```
#include <cstdlib>

double* vec = static_cast<double*>
      (std::aligned_alloc(32,
            n * sizeof(double)));
...
free(vec);
```

aligned_alloc

```
#include <cstdlib>

double* vec = static_cast<double*>
      (std::aligned_alloc(32,
            n * sizeof(double)));
...
free(vec);
```

- Минусы: приходится использовать ручное динамическое выделение памяти в стиле Си, а не контейнер vector

aligned_alloc

```
#include <cstdlib>

double* vec = static_cast<double*>
      (std::aligned_alloc(32,
            n * sizeof(double)));
...
free(vec);
```

- Минусы: приходится использовать ручное динамическое выделение памяти в стиле Си, а не контейнер vector
- Минусы: путаница со стандартами. Функция aligned_alloc есть в стандарте C11, но C++17. При этом, не совсем ясно, которая вызывается, а на mac os x не работает совсем

aligned_alloc

```
#include <cstdlib>

double* vec = static_cast<double*>
      (std::aligned_alloc(32,
            n * sizeof(double)));
...
free(vec);
```

- Минусы: приходится использовать ручное динамическое выделение памяти в стиле Си, а не контейнер vector
- Минусы: путаница со стандартами. Функция aligned_alloc есть в стандарте C11, но C++17. При этом, не совсем ясно, которая вызывается, а на mac os x не работает совсем
- Минусы: нужно не забыть освободить память

aligned_alloc + unique_ptr

- Сделаем версию с автоматическим освобождением памяти

aligned_alloc + unique_ptr

- Сделаем версию с автоматическим освобождением памяти

```
struct free_delete {
    void operator()(void* x) { free(x); }
};
auto vec =
    std::unique_ptr<double,free_delete>(
        static_cast<double*>(
            aligned_alloc(32,
                n * sizeof(double))
        )
    );
```

aligned_alloc + unique_ptr

- Сделаем версию с автоматическим освобождением памяти

```
struct free_delete {
    void operator()(void* x) { free(x); }
};
auto vec =
    std::unique_ptr<double, free_delete>(
        static_cast<double*>(
            aligned_alloc(32,
                n * sizeof(double))
        )
    );
```

- При выходе из области видимости vec вызовется деструктор, который приведет к вызову free.

aligned_alloc + unique_ptr

- Сделаем версию с автоматическим освобождением памяти

```
struct free_delete {
    void operator()(void* x) { free(x); }
};
auto vec =
    std::unique_ptr<double, free_delete>(
        static_cast<double*>(
            aligned_alloc(32,
                n * sizeof(double))
        )
    );
```

- При выходе из области видимости vec вызовется деструктор, который приведет к вызову free.
- Проблема освобождения памяти ушла, но это не решает другие проблемы

Способы повышенного выравнивания в массиве

Существуют различные способы добиться повышенного выравнивания в массиве, все со своими преимуществами и недостатками

- Взятие большего размера и отсчет не с нуля
- aligned_alloc
- **aligned new**
- Кастомные аллокаторы для контейнеров

aligned_new

```
#include <new>

double* vec = new(DoubleAligment) double[n];

operator delete[] (vec, DoubleAligment);
```

aligned_new

```
#include <new>

double* vec = new(DoubleAligment) double[n];

operator delete[] (vec, DoubleAligment);
```

- Плюсы: современный код из C++17

aligned_new

```
#include <new>

double* vec = new(DoubleAligment) double[n];

operator delete[] (vec, DoubleAligment);
```

- Плюсы: современный код из C++17
- Минусы: требуется ручное освобождение памяти (будем решать)

aligned_new + unique_ptr

```
constexpr std::align_val_t
    DoubleAligment {32};
struct aligned_delete {
    void operator()(void* x) {
        operator delete[] (x, DoubleAligment);
    }
};
auto vec =
    std::unique_ptr<double,aligned_delete>(
        new(DoubleAligment) double[n]
    );
```

aligned_new + unique_ptr

```
constexpr std::align_val_t
    DoubleAligment {32};
struct aligned_delete {
    void operator()(void* x) {
        operator delete[] (x, DoubleAligment);
    }
};
auto vec =
    std::unique_ptr<double, aligned_delete>(
        new(DoubleAligment) double[n]
    );
```

- Теперь мы не забудем освободить память, но конструкция довольно серьезно усложнилась, и у нас все равно массивы, а не вектора

Способы повышенного выравнивания в массиве

Существуют различные способы добиться повышенного выравнивания в массиве, все со своими преимуществами и недостатками

- Взятие большего размера и отсчет не с нуля
- aligned_alloc
- aligned new
- **Кастомные аллокаторы для контейнеров**

Кастомные аллокаторы для контейнеров

- Последний подход основан на том, что шаблон вектора (да и прочих контейнеров) содержит дополнительный аргумент, обычно не использующийся, – аллокатор памяти

Кастомные аллокаторы для контейнеров

- Последний подход основан на том, что шаблон вектора (да и прочих контейнеров) содержит дополнительный аргумент, обычно не использующийся, – аллокатор памяти

```
std::vector<T, Allocator>
```

Кастомные аллокаторы для контейнеров

- Последний подход основан на том, что шаблон вектора (да и прочих контейнеров) содержит дополнительный аргумент, обычно не использующийся, – аллокатор памяти

```
std::vector<T, Allocator>
```

- Но что же писать в поле аллокатор? Нужен специальный класс, удовлетворяющий целому ряду свойств. Но вы же не будете писать каждый раз подобный аллокатор, когда собираетесь где-то выровнять память!

Кастомные аллокаторы для контейнеров

- Последний подход основан на том, что шаблон вектора (да и прочих контейнеров) содержит дополнительный аргумент, обычно не использующийся, – аллокатор памяти

```
std::vector<T, Allocator>
```

- Но что же писать в поле аллокатор? Нужен специальный класс, удовлетворяющий целому ряду свойств. Но вы же не будете писать каждый раз подобный аллокатор, когда собираетесь где-то выровнять память!
- К счастью, есть кочующее по интернету решения аллокатора для контейнеров std, основанное на описанных выше методах

Кастомные аллокаторы для контейнеров

- Берем код из https://johanmabille.github.io/blog/2014/12/06/aligned-memory-allocator/
- или https://godbolt.org/z/PG5Ph7936

Кастомные аллокаторы для контейнеров

- Берем код из https://johanmabille.github.io/blog/2014/12/06/aligned-memory-allocator/
- или https://godbolt.org/z/PG5Ph7936

```
#include "aligned_mem.h"
constexpr size_t align = 32;
std::vector<double,
     AlignedAllocator<double, align>> vec;
```

Кастомные аллокаторы для контейнеров

- Берем код из https://johanmabille.github.io/blog/2014/12/06/aligned-memory-allocator/
- или https://godbolt.org/z/PG5Ph7936

```
#include "aligned_mem.h"
constexpr size_t align = 32;
std::vector<double,
     AlignedAllocator<double, align>> vec;
```

- Плюсы – минимальное изменение семантики, мы работаем именно с вектором. Достаточно универсальное решение

Кастомные аллокаторы для контейнеров

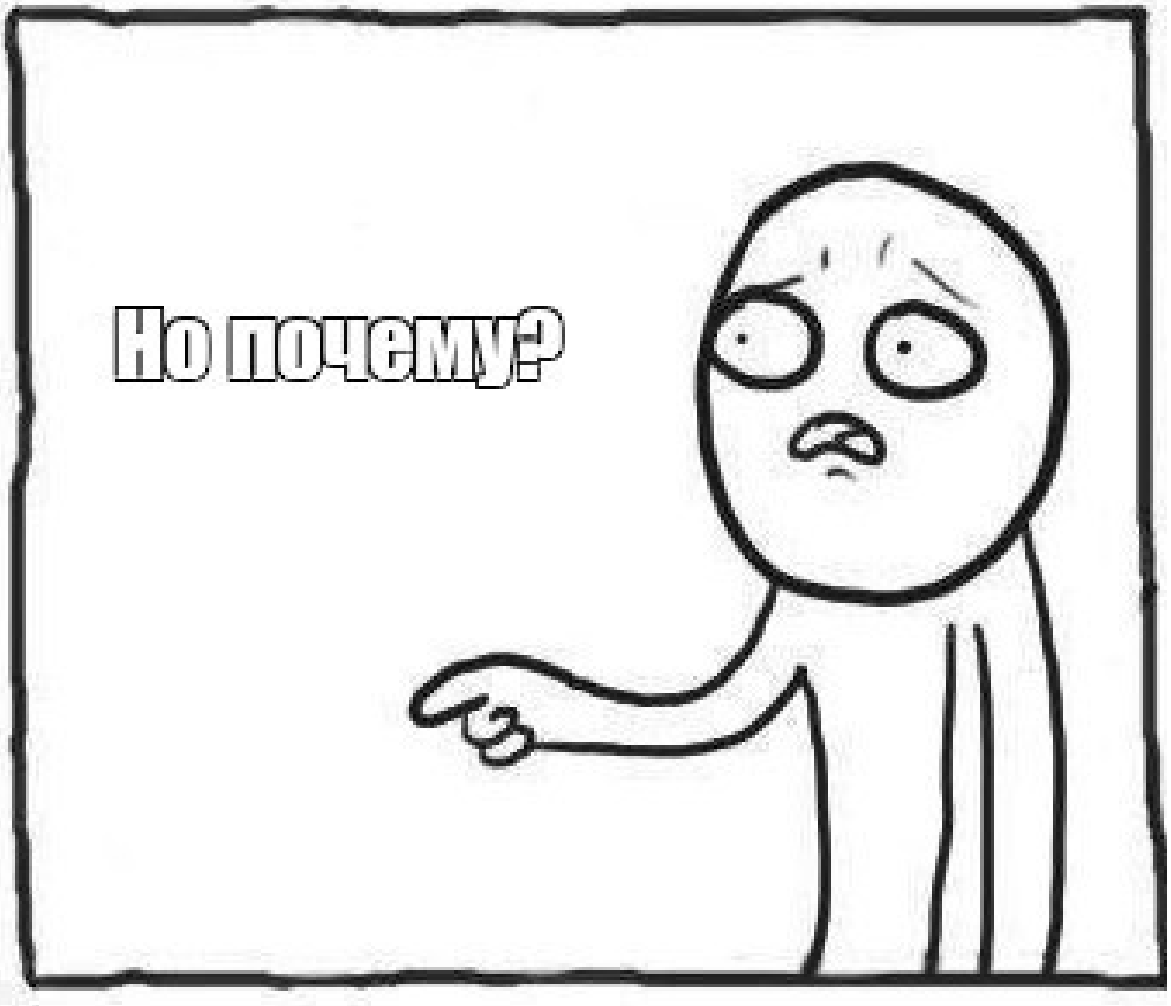
- Берем код из https://johanmabille.github.io/blog/2014/12/06/aligned-memory-allocator/
- или https://godbolt.org/z/PG5Ph7936

```
#include "aligned_mem.h"
constexpr size_t align = 32;
std::vector<double,
     AlignedAllocator<double, align>> vec;
```

- Плюсы – минимальное изменение семантики, мы работаем именно с вектором. Достаточно универсальное решение
- Минусы – нам понадобился сторонний код. Но почему его нет в стандарте?

2024 год на дворе...

- Минусы – нам понадобился сторонний код. Но почему его нет в стандарте?



Вопросы?

Раздел 4. План.

Параллельное программирование на одном вычислительном ядре

- Ускорение суммирование вектора при помощи SSE инструкций
- Выравнивание в памяти и способы его достижения
- **Ускорение суммирование вектора при помощи AVX инструкций и задержки памяти**
- Кэш-ориентированный код

Реализация сложения с AVX

```
auto av = reinterpret_cast<__m256d *>(
      a.data() + start);
__m256d resv = _mm256_setzero_pd();
for (size_t i = start;i != end; i+=4) {
      resv = _mm256_add_pd(resv, *av);
      ++av;
}
double res[4];
_mm256_store_pd(res, resv);
return res[0]+res[1]+res[2]+res[3];
```

Реализация сложения с AVX

```
auto av = reinterpret_cast<__m256d *>(
      a.data() + start);
__m256d resv = _mm256_setzero_pd();
for (size_t i = start;i != end; i+=4) {
      resv = _mm256_add_pd(resv, *av);
      ++av;
}
double res[4];
_mm256_store_pd(res, resv);
return res[0]+res[1]+res[2]+res[3];
```

- Мы заводим переменную типа `__m256d *` – указатель на четверки даблов. Подобные переменные годятся для использования AVX инструкций

Реализация сложения с AVX

```
auto av = reinterpret_cast<__m256d *>(
    a.data() + start);
__m256d resv = _mm256_setzero_pd();
for (size_t i = start;i != end; i+=4) {
    resv = _mm256_add_pd(resv, *av);
    ++av;
}
double res[4];
_mm256_store_pd(res, resv);
return res[0]+res[1]+res[2]+res[3];
```

- Мы заводим переменную типа `__m256d *` – указатель на четверки даблов. Подобные переменные годятся для использования AVX инструкций
- Продвигаемся по циклу четверками (n делится), потом нужно сложить все части resv

Реализация сложения с AVX

```
        vxorpd  %xmm0, %xmm0, %xmm0     # resv
.L30:
        vaddpd  (%rax,%rcx,8), %ymm0, %ymm0
        addq    $4, %rcx
        cmpq    %rcx, %r8
        jne     .L30    #,
```

Реализация сложения с AVX

```
        vxorpd   %xmm0, %xmm0, %xmm0      # resv
.L30:
        vaddpd   (%rax,%rcx,8), %ymm0, %ymm0
        addq     $4, %rcx
        cmpq     %rcx, %r8
        jne      .L30    #,
```

- Для компиляции потребуется флаг -mavx, разрешающий использовать avx инструкции (все-таки не у каждого процессора, что вы найдете, они есть)

Реализация сложения с AVX

```
        vxorpd  %xmm0, %xmm0, %xmm0     # resv
.L30:
        vaddpd  (%rax,%rcx,8), %ymm0, %ymm0
        addq    $4, %rcx
        cmpq    %rcx, %r8
        jne     .L30    #,
```

- Для компиляции потребуется флаг -mavx, разрешающий использовать avx инструкции (все-таки не у каждого процессора, что вы найдете, они есть)
- Получившийся код использует инструкцию vaddpd, работающая с четверками чисел. Также используется 256-битный регистр ymm, младшей частью которого является xmm

Реализация сложения с AVX

```
        vxorpd  %xmm0, %xmm0, %xmm0     # resv
.L30:
        vaddpd  (%rax,%rcx,8), %ymm0, %ymm0
        addq    $4, %rcx
        cmpq    %rcx, %r8
        jne     .L30    #,
```

- Для компиляции потребуется флаг -mavx, разрешающий использовать avx инструкции (все-таки не у каждого процессора, что вы найдете, они есть)
- Получившийся код использует инструкцию vaddpd, работающая с четверками чисел. Также используется 256-битный регистр ymm, младшей частью которого является xmm
- Здесь rax это a, rcx это i

Реализация сложения с AVX

```
        vxorpd  %xmm0, %xmm0, %xmm0     # resv
.L30:
        vaddpd  (%rax,%rcx,8), %ymm0, %ymm0
        addq    $4, %rcx
        cmpq    %rcx, %r8
        jne     .L30    #,
```

- Для компиляции потребуется флаг -mavx, разрешающий использовать avx инструкции (все-таки не у каждого процессора, что вы найдете, они есть)
- Получившийся код использует инструкцию vaddpd, работающая с четверками чисел. Также используется 256-битный регистр ymm, младшей частью которого является xmm
- Здесь rax это a, rcx это i
- Время 109 ms, при том, что на SSE было 135 – почему?

Реализация сложения с AVX

```
        vxorpd  %xmm0, %xmm0, %xmm0     # resv
.L30:
        vaddpd  (%rax,%rcx,8), %ymm0, %ymm0
        addq    $4, %rcx
        cmpq    %rcx, %r8
        jne     .L30    #,
```

- Для компиляции потребуется флаг -mavx, разрешающий использовать avx инструкции (все-таки не у каждого процессора, что вы найдете, они есть)
- Получившийся код использует инструкцию vaddpd, работающая с четверками чисел. Также используется 256-битный регистр ymm, младшей частью которого является xmm
- Здесь rax это a, rcx это i
- Время 109 ms, при том, что на SSE было 135 – почему?
- Мы уперлись в скорость работы памяти!

Попытаемся выжать еще немного

```
const double * av = a.data() + start;
__m256d resv1 = _mm256_setzero_pd();
__m256d resv2 = _mm256_setzero_pd();
__m256d val1, val2;
for (size_t i = start;i != end; i+=8) {
    val1 = _mm256_load_pd(av);
    resv1 = _mm256_add_pd(resv1, val1);
    val2 = _mm256_load_pd(av + 4);
    resv2 = _mm256_add_pd(resv2, val2);
    av += 8;
}
resv1 = _mm256_add_pd(resv1, resv2);
double res[4];
_mm256_store_pd(res, resv1);
return res[0]+res[1]+res[2]+res[3];
```

Попытаемся выжать еще немного

```
const double * av = a.data() + start;
__m256d resv1 = _mm256_setzero_pd();
__m256d resv2 = _mm256_setzero_pd();
__m256d val1, val2;
for (size_t i = start;i != end; i+=8) {
    val1 = _mm256_load_pd(av);
    resv1 = _mm256_add_pd(resv1, val1);
    val2 = _mm256_load_pd(av + 4);
    resv2 = _mm256_add_pd(resv2, val2);
    av += 8;
}
resv1 = _mm256_add_pd(resv1, resv2);
double res[4];
_mm256_store_pd(res, resv1);
return res[0]+res[1]+res[2]+res[3];
```

- Пытаемся разнести инструкции загрузки и сложения

Реализация сложения с AVX

```
        vxorpd   %xmm1, %xmm1, %xmm1 # resv2
        vmovapd %ymm1, %ymm0      #, resv1
.L30:
        vaddpd   (%rax,%rcx,8), %ymm0, %ymm0
        vaddpd   32(%rax,%rcx,8), %ymm1, %ymm1
        addq     $8, %rcx
        cmpq     %rcx, %r8
        jne      .L30
```

Реализация сложения с AVX

```
        vxorpd   %xmm1, %xmm1, %xmm1 # resv2
        vmovapd %ymm1, %ymm0      #, resv1
.L30:
        vaddpd   (%rax,%rcx,8), %ymm0, %ymm0
        vaddpd   32(%rax,%rcx,8), %ymm1, %ymm1
        addq     $8, %rcx
        cmpq     %rcx, %r8
        jne      .L30
```

- asm-код в итоге не сильно сложнее, у нас два накапливающих регистра

Реализация сложения с AVX

```
        vxorpd   %xmm1, %xmm1, %xmm1 # resv2
        vmovapd  %ymm1, %ymm0      #, resv1
.L30:
        vaddpd   (%rax,%rcx,8), %ymm0, %ymm0
        vaddpd   32(%rax,%rcx,8), %ymm1, %ymm1
        addq     $8, %rcx
        cmpq     %rcx, %r8
        jne      .L30
```

- asm-код в итоге не сильно сложнее, у нас два накапливающих регистра
- Время 108ms вместо 109 – и все-таки мы упираемся в память

Реализация сложения с AVX

```
        vxorpd  %xmm1, %xmm1, %xmm1 # resv2
        vmovapd %ymm1, %ymm0      #, resv1
.L30:
        vaddpd  (%rax,%rcx,8), %ymm0, %ymm0
        vaddpd  32(%rax,%rcx,8), %ymm1, %ymm1
        addq    $8, %rcx
        cmpq    %rcx, %r8
        jne     .L30
```

- asm-код в итоге не сильно сложнее, у нас два накапливающих регистра
- Время 108ms вместо 109 – и все-таки мы упираемся в память
- Тем не менее, это не означает, что AVX инструкции мало полезны, все зависит от специфики задачи. Но показать работающий пример, на котором они дадут выигрыш порядка 50% на лекции не выйдет. Однако у нас как раз не давно в работе было получено подобное ускорение.

Стартовый пример

А если складывать вектора, почему этот пример не походит?

```
for (i = 0; i!=n; ++i) {
    c[i] = a[i] + b[i]
}
```

Стартовый пример

А если складывать вектора, почему этот пример не походит?

```
for (i = 0; i!=n; ++i) {
    c[i] = a[i] + b[i]
}
```

- Здесь компилятор сам может сообразить использовать SSE и AVX инструкции в цикле, поэтому ручное написание интринсиков не обосновано

Стартовый пример

А если складывать вектора, почему этот пример не походит?

```
for (i = 0; i!=n; ++i) {
    c[i] = a[i] + b[i]
}
```

- Здесь компилятор сам может сообразить использовать SSE и AVX инструкции в цикле, поэтому ручное написание интринсиков не обосновано
- Здесь больше нагрузка на память, и запись, и чтение, и поэтому эффект от AVX вообще не наблюдается

Вопросы?

Раздел 4. План.

Параллельное программирование на одном вычислительном ядре

- Ускорение суммирование вектора при помощи SSE инструкций
- Выравнивание в памяти и способы его достижения
- Ускорение суммирование вектора при помощи AVX инструкций и задержки памяти
- **Кэш-ориентированный код**

Скорость работы памяти и процессора

- sudo lshw -short -C memory

Скорость работы памяти и процессора

- sudo lshw -short -C memory
- `/0/a/0 memory 8GiB SO-DIMM DDR4 Синхронная Unbuffered (Unregistered) 3200 MHz (0,3 ns) /0/a/1 memory 16GiB SO-DIMM DDR4 Синхронная Unbuffered (Unregistered) 2400 MHz (0,4 ns)`

Скорость работы памяти и процессора

- sudo lshw -short -C memory
- `/0/a/0 memory 8GiB SO-DIMM DDR4 Синхронная Unbuffered (Unregistered) 3200 MHz (0,3 ns) /0/a/1 memory 16GiB SO-DIMM DDR4 Синхронная Unbuffered (Unregistered) 2400 MHz (0,4 ns)`
- AMD Ryzen 7 3750H - 2.3GHz – числа сопоставимые, однако на операции работы с памятью требуется больше тактов

Скорость работы памяти и процессора

- sudo lshw -short -C memory
- `/0/a/0 memory 8GiB SO-DIMM DDR4 Синхронная Unbuffered (Unregistered) 3200 MHz (0,3 ns) /0/a/1 memory 16GiB SO-DIMM DDR4 Синхронная Unbuffered (Unregistered) 2400 MHz (0,4 ns)`
- AMD Ryzen 7 3750H - 2.3GHz – числа сопоставимые, однако на операции работы с памятью требуется больше тактов
- Презентация про avx: https://indico.cern.ch/event/327306/contributions/760669/attachments/635800/875267/HaswellConundrum.pdf

Уровни иерархии памяти

Где же кэширование?

- В задаче, что мы реализовывали, большого толку от кэширования нет. Да, процессор в любом случае загружает блоки вектора из оперативной памяти в кэш. Но к каждому элементу вектора обращение происходит один раз, а это фактически снимает пользу кэширования. Нормальная польза будет, если к каждому элементу возможно многократное обращение (например, перемножение матриц)

Где же кэширование?

- В задаче, что мы реализовывали, большого толку от кэширования нет. Да, процессор в любом случае загружает блоки вектора из оперативной памяти в кэш. Но к каждому элементу вектора обращение происходит один раз, а это фактически снимает пользу кэширования. Нормальная польза будет, если к каждому элементу возможно многократное обращение (например, перемножение матриц)
- Тем не менее, написанный у нас код, был cache-friendly, поскольку последовательные элементы вектора лежат последовательно в памяти, и поэтому процессор загружает данные в кэш блоками. Замени мы vector на list, было бы существенно хуже
- В задачах, где возможно кэширование, стоит использовать технологию block caching, когда реиспользуемые блоки делаются такого размера, чтобы попадать в кэш. Ниже продемонстрируем это, но начнем с более простого.

Умножение матриц

Напишем сначала обычное умножение матриц:

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[i][k] * b[k][j];
    }
    c[i][j] = res;
  }
}
```

Умножение матриц

Напишем сначала обычное умножение матриц:

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[i][k] * b[k][j];
    }
    c[i][j] = res;
  }
}
```

- Что плохого в этом коде?

Умножение матриц

Напишем сначала обычное умножение матриц:

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[i][k] * b[k][j];
    }
    c[i][j] = res;
  }
}
```

- Что плохого в этом коде?
- А я не уточнил, что такое a, b, c.

Умножение матриц

Варианты хранения матрицы

- std::vector<std::vector<double> >

Умножение матриц

Варианты хранения матрицы

- std::vector<std::vector<double> >
- Здесь плохо, что происходит двойной переход по памяти, это совершенно не cache friendly

Умножение матриц

Варианты хранения матрицы

- std::vector<std::vector<double> >
- Здесь плохо, что происходит двойной переход по памяти, это совершенно не cache friendly
- double[N][N]

Умножение матриц

Варианты хранения матрицы

- std::vector<std::vector<double> >
- Здесь плохо, что происходит двойной переход по памяти, это совершенно не cache friendly
- double[N][N]
- Такой двумерный массив из си. Позволяет адресацию парами квадратных скобок. Но неудобно выделять динамически. Поэтому перепишем код через адресацию с умножением, чтобы данные хранились последовательно. Я даже не буду делать замер с двойной адресацией, это очень неэффективно.

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[j+k*size];
    }
    c[j+size*i] = res;
  }
}
```

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[j+k*size];
    }
    c[j+size*i] = res;
  }
}
```

- Что плохого в этом коде?

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[j+k*size];
    }
    c[j+size*i] = res;
  }
}
```

- Что плохого в этом коде?
- По второй матрице происходит передвижение скачками. Это ни разу не cache-friendly!

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[j+k*size];
    }
    c[j+size*i] = res;
  }
}
```

- Что плохого в этом коде?
- По второй матрице происходит передвижение скачками. Это ни разу не cache-friendly!
- Матрицу b надо предварительно транспонировать. Перепишем код с учетом того, что это уже сделано.

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[k+j*size];
    }
    c[j+size*i] = res;
  }
}
```

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[k+j*size];
    }
    c[j+size*i] = res;
  }
}
```

- Сравним время (ms)

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[k+j*size];
    }
    c[j+size*i] = res;
  }
}
```

- И все-таки что плохо в этом коде?

Умножение матриц

```
for (size_t i = 0; i < size; ++i) {
  for (size_t j = 0; j < size; ++j) {
    double res = 0.;
    for (size_t k = 0; k < size; ++k) {
      res += a[k+i*size] * b[k+j*size];
    }
    c[j+size*i] = res;
  }
}
```

- И все-таки что плохо в этом коде?
- Помимо того, что мы не использовали векторизацию (но я сейчас делаю акцент на другом), недостаточно хорошо используются кэши

Block caching

- Предположим, у нас есть условный код

```
for (i = 0; i < n; ++i) {
    for (j = 0; j < n; j++) {
        a[i] += compute(b[i], c[j]);
    }
}
```

Block caching

- Предположим, у нас есть условный код

```
for (i = 0; i < n; ++i) {
    for (j = 0; j < n; j++) {
        a[i] += compute(b[i], c[j]);
    }
}
```

- Если n достаточно большое, то вектор c не попадает в кэш целиком, поэтому при переходе к следующему i начнет считываться из оперативной памяти заново

Block caching

- Предположим, у нас есть условный код

```
for (i = 0; i < n; ++i) {
    for (j = 0; j < n; j++) {
        a[i] += compute(b[i], c[j]);
    }
}
```

- Если n достаточно большое, то вектор c не попадает в кэш целиком, поэтому при переходе к следующему i начнет считываться из оперативной памяти заново
- Решение – разбить на блоки, подходящие для кэширования

Block caching

```
for (jj = 0; jj < n; jj += BLOCK) {
    for (i = 0; i < n; ++i) {
        for (j = jj; j < jj + BLOCK; j++) {
            a[i] += compute(b[i], c[j]);
        }
    }
}
```

Block caching

```
for (jj = 0; jj < n; jj += BLOCK) {
    for (i = 0; i < n; ++i) {
        for (j = jj; j < jj + BLOCK; j++) {
            a[i] += compute(b[i], c[j]);
        }
    }
}
```

- В таком случае блок их вектора c сначала считывается в кэш и будет переиспользован для всех i. Аналогично можно разбить и вторую часть.

Block caching

```
for (jj = 0; jj < n; jj += BLOCK) {
    for (i = 0; i < n; ++i) {
        for (j = jj; j < jj + BLOCK; j++) {
            a[i] += compute(b[i], c[j]);
        }
    }
}
```

- В таком случае блок их вектора c сначала считывается в кэш и будет переиспользован для всех i. Аналогично можно разбить и вторую часть.
- Попробуем продемонстрировать эту технологию на примере умножения матриц (код на слайдет будет упрощен с условием, что size кратно делится на выбранный размер блока bsize)

Кэш-ориентированное умножение матриц

```
for (size_t jj = 0; jj < size; jj+=bsize) {
  for (size_t i = 0; i < size; i++) {
    for (size_t j = jj; j < size; j++) {
      c[j+i*size] = 0.0;
    }
    for (size_t kk = 0; kk < size; kk+=bsize) {
      for (size_t i = 0; i < size; i++) {
        for (size_t j = jj; j < size; j++) {
          double res = 0.0;
          for (size_t k = kk; k < size; k++) {
            res += a[k+i*size] * b[k+j*size];
          }
          c[j+i*size] += res;
} } } } }
```

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334
block 16	992	9003	77566

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334
block 16	992	9003	77566
block 32	652	5500	51052

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334
block 16	992	9003	77566
block 32	652	5500	51052
block 64	689	5559	46889

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334
block 16	992	9003	77566
block 32	652	5500	51052
block 64	689	5559	46889
block 128	928	6502	56190

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334
block 16	992	9003	77566
block 32	652	5500	51052
block 64	689	5559	46889
block 128	928	6502	56190
block 256	910	7325	85243

Кэш-ориентированное умножение матриц

size	1024	2048	4096
direct	7840	63520	1537347
transpose	998	7719	60334
block 16	992	9003	77566
block 32	652	5500	51052
block 64	689	5559	46889
block 128	928	6502	56190
block 256	910	7325	85243

- Почему именно выбор 32-64 дает оптимальные результаты?

Кэш-ориентированное умножение матриц

```
lscpu

Caches (sum of all):
L1d:                    128 KiB (4 instances)
L1i:                    256 KiB (4 instances)
L2:                     2 MiB (4 instances)
L3:                     4 MiB (1 instance)
```

Кэш-ориентированное умножение матриц

```
lscpu

Caches (sum of all):
L1d:                    128 KiB (4 instances)
L1i:                    256 KiB (4 instances)
L2:                     2 MiB (4 instances)
L3:                     4 MiB (1 instance)
```

- Да, ускорение не столь впечатляющее по сравнение с тем, что дал переход на транспонирование, но оно есть.

Кэш-ориентированное умножение матриц

```
lscpu

Caches (sum of all):
L1d:                    128 KiB (4 instances)
L1i:                    256 KiB (4 instances)
L2:                     2 MiB (4 instances)
L3:                     4 MiB (1 instance)
```

- Да, ускорение не столь впечатляющее по сравнение с тем, что дал переход на транспонирование, но оно есть.
- Можно еще попробовать воспользоваться инструкцией __builtin_prefetch для просьбы закэшировать блок, но в данном случае она не дала никакого дополнительного эффекта, компилятор и так хорошо справляется.

Кэш-ориентированное умножение матриц

```
lscpu

Caches (sum of all):
L1d:                    128 KiB (4 instances)
L1i:                    256 KiB (4 instances)
L2:                     2 MiB (4 instances)
L3:                     4 MiB (1 instance)
```

- Да, ускорение не столь впечатляющее по сравнение с тем, что дал переход на транспонирование, но оно есть.
- Можно еще попробовать воспользоваться инструкцией __builtin_prefetch для просьбы закэшировать блок, но в данном случае она не дала никакого дополнительного эффекта, компилятор и так хорошо справляется.
- Более серьезное ускорение от block-caching на лекции сложно продемонстрировать. Как и с другими технологиями, нужно всегда понимать, что и зачем применяется.

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 2. Если 0, то берете в начале _mm, иначе _mm256
- Возьмите остаток деления своего номера на 4. Если 0 или 2, то берете в конце _pd, иначе _ps
- Возьмите частное от деления своего номера на 8 без остатка. В зависимости от значения выберите середину инструкции: 0 или 9: _add, 1 или 10: _and, 2 или 11: _load, 3: _store, 4: _set или _set1, 5: _cmp или _cmpge, 6: _mul, 7: _div, 8: _sub

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 2. Если 0, то берете в начале _mm, иначе _mm256
- Возьмите остаток деления своего номера на 4. Если 0 или 2, то берете в конце _pd, иначе _ps
- Возьмите частное от деления своего номера на 8 без остатка. В зависимости от значения выберите середину инструкции: 0 или 9: _add, 1 или 10: _and, 2 или 11: _load, 3: _store, 4: _set или _set1, 5: _cmp или _cmpge, 6: _mul, 7: _div, 8: _sub
- Реализуйте простейшую программу, демонстрирующую полезность именно этой конструкции.

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 2. Если 0, то берете в начале _mm, иначе _mm256
- Возьмите остаток деления своего номера на 4. Если 0 или 2, то берете в конце _pd, иначе _ps
- Возьмите частное от деления своего номера на 8 без остатка. В зависимости от значения выберите середину инструкции: 0 или 9: _add, 1 или 10: _and, 2 или 11: _load, 3: _store, 4: _set или _set1, 5: _cmp или _cmpge, 6: _mul, 7: _div, 8: _sub
- Реализуйте простейшую программу, демонстрирующую полезность именно этой конструкции.
- Результат сдается на сайте курса – нужен код программы (должна компилироваться) и описание, что вы этим демонстрируете (можно внутри кода в комментариях). Рекомендуется сдавать ее вместе с Makefile.

Вопросы?