



Новосибирский государственный университет
Факультет информационных технологий
Кафедра параллельных вычислений

Эффективное программирование современных микропроцессоров и мультипроцессоров

Векторизация вычислений

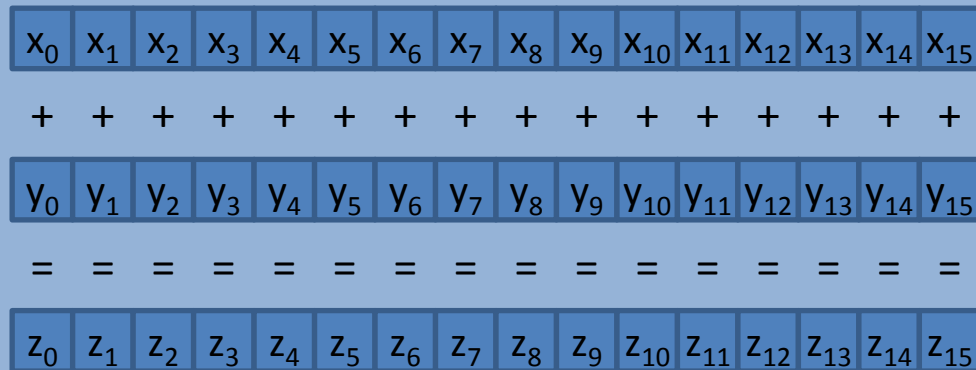
Преподаватели:
Киреев С.Е.
Калгин К.В.

План лекции

- **Введение**
- Обзор векторных расширений современных x86-микропроцессоров
- Проблемы векторизации
- Средства векторизации
- Автоматическая векторизация программ компилятором
- Полуавтоматическая векторизация с помощью OpenMP 4.0

Введение

- **Векторные вычисления** – это вид параллельных вычислений с параллелизмом на уровне данных (SIMD – Single Instruction Multiple Data)



Введение

- **Скалярная программа** – программа, оперирующая отдельными числами
- **Векторная программа** – программа, оперирующая векторами
- **Векторизация** (вид распараллеливания) – преобразование скалярной программы в векторную

Введение

- Цели векторизации
 1. Ускорить работу программы
 2. Уменьшить объем кода
- Предпосылки
 - Одна векторная команда распознаётся, декодируется и выполняется быстрее нескольких скалярных, выполняющих то же действие
 - Одна векторная команда занимает меньше места в программе и в различных очередях/таблицах/буферах в процессоре

Введение

- В современных скалярных микропроцессорах общего назначения векторные вычисления поддерживаются с помощью **векторных расширений** архитектуры
 - Примеры векторных расширений: MMX, SSE, AVX, ...
- Векторные расширения включают:
 - Векторные регистры – хранят множества скалярных значений
 - Векторные команды – для работы с векторными регистрами

План лекции

- Введение
- **Обзор векторных расширений современных x86-микропроцессоров**
- Проблемы векторизации
- Средства векторизации
- Автоматическая векторизация программ компилятором
- Полуавтоматическая векторизация с помощью OpenMP 4.0

Векторные расширения современных микропроцессоров

Основные параметры

- Размер регистра:
 - 8 байт, 16 байт, 32 байта, 64 байта
- Поддерживаемые типы данных:
 - Целочисленные: 1,2,4,8 байт, знаковые/беззнаковые
 - Вещественные: 4 байта (float), 8 байт (double)
- Поддерживаемые операции:
 - Чтение, запись
 - Арифметические, логические
 - Перестановки, копирование элементов вектора
 - Сравнение
 - Преобразования типов
 - Специальные операции

Векторные расширения

современных микропроцессоров

- 1997 – MMX (Pentium MMX)
 - 8 байт, целочисленные операции

- 8 регистров: mm0-mm7 (поверх FPU)
- Двухоперандные операции
- Арифметика с насыщением

Векторные расширения

современных микропроцессоров

- 1997 – [MMX](#) (Pentium MMX)
 - 8 байт, целочисленные операции
- 1998 – [3DNow!](#) (K6-2)
 - 8 байт, float

- 8 регистров (поверх MMX и FPU)

Векторные расширения

современных микропроцессоров

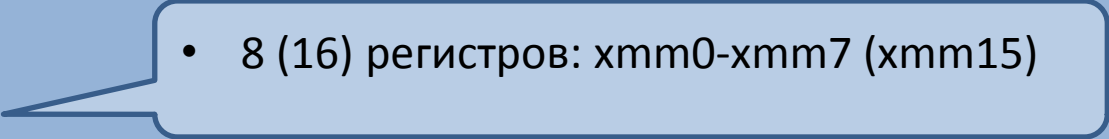
- 1997 – [MMX](#) (Pentium MMX)
 - 8 байт, целочисленные операции
- 1998 – [3DNow!](#) (K6-2)
 - 8 байт, float
- 1999 – [SSE](#) (Pentium III)
 - 16 байт, float

- 8 (16) регистров: xmm0-xmm7 (xmm15)

Векторные расширения

современных микропроцессоров

- 1997 – [MMX](#) (Pentium MMX)
 - 8 байт, целочисленные операции
- 1998 – [3DNow!](#) (K6-2)
 - 8 байт, float
- 1999 – [SSE](#) (Pentium III)
 - 16 байт, float
- 2001 – [SSE2](#) (Pentium 4)
 - 16 байт, double

- 
- 8 (16) регистров: xmm0-xmm7 (xmm15)

Векторные расширения

современных микропроцессоров

- 1997 – [MMX](#) (Pentium MMX)
 - 8 байт, целочисленные операции
- 1998 – [3DNow!](#) (K6-2)
 - 8 байт, float
- 1999 – [SSE](#) (Pentium III)
 - 16 байт, float
- 2001 – [SSE2](#) (Pentium 4)
 - 16 байт, double
- 2004 – [SSE3](#) (Pentium 4 Prescott)
 - Горизонтальные операции
- 2006 – [SSSE3](#) (Core 2, Atom)
 - Расширение набора целочисленных команд
- 2007 – [SSE4.1](#) (Penryn)
- 2008 – [SSE4.2](#) (Nehalem)
 - Специальные команды: обработка видео, обработка строк, криптография

Векторные расширения

современных микропроцессоров

- 1997 – [MMX](#) (Pentium MMX)
 - 16 регистров `ymm0-ymm15`
 - регистры `xmm` – часть регистров `ymm`
 - Отсутствуют требования к выравниванию обращений к памяти
- 2001 – [SSE2](#) (Pentium 4)
 - 16 байт, double
- 2004 – [SSE3](#) (Pentium 4 Prescott)
 - Горизонтальные операции
- 2006 – [SSSE3](#) (Core 2, Atom)
 - Расширение набора целочисленных команд
- 2007 – [SSE4.1](#) (Penryn)
- 2008 – [SSE4.2](#) (Nehalem)
 - Специальные команды: обработка видео, обработка строк, криптография
- 2011 – [AVX](#) (Sandy Bridge)
 - 32 байта, float, double
 - Трёхоперандные команды
- 2013 – [AVX2](#) (Haswell)
 - 32 байта, целочисленные операции
 - FMA: $r = a * b + c$

Векторные расширения

современных микропроцессоров

- 1997 – [MMX](#) (Pentium MMX)
 - 8 байт, целочисленные операции
- 1998 – [3DNow!](#) (K6-2)
 - 8 байт, float
- 1999 – [SSE](#) (Pentium III)
 - 16 байт, float
- 2001 – [SSE2](#) (Pentium 4)
 - 16 байт, double
- 2004 – [SSE3](#) (Pentium 4 Prescott)
 - 16 байт, float, double
- 2011 – [AVX](#) (Sandy Bridge)
 - 32 байта, float, double
 - Трёхоперандные команды
- 2013 – [AVX2](#) (Haswell)
 - 32 байта, целочисленные операции
 - FMA: $r = a * b + c$
- 2016 – [AVX-512](#) (KNL, Skylake)
 - 64 байта

- 32 регистра zmm0-zmm31
 - регистры ymm – часть регистров zmm
- 8 регистров масок: k0-k7
- Команды scatter, gather
- Операции по маске

• Специальные команды. обработка видео, обработка строк, криптография

Векторные расширения

современных микропроцессоров

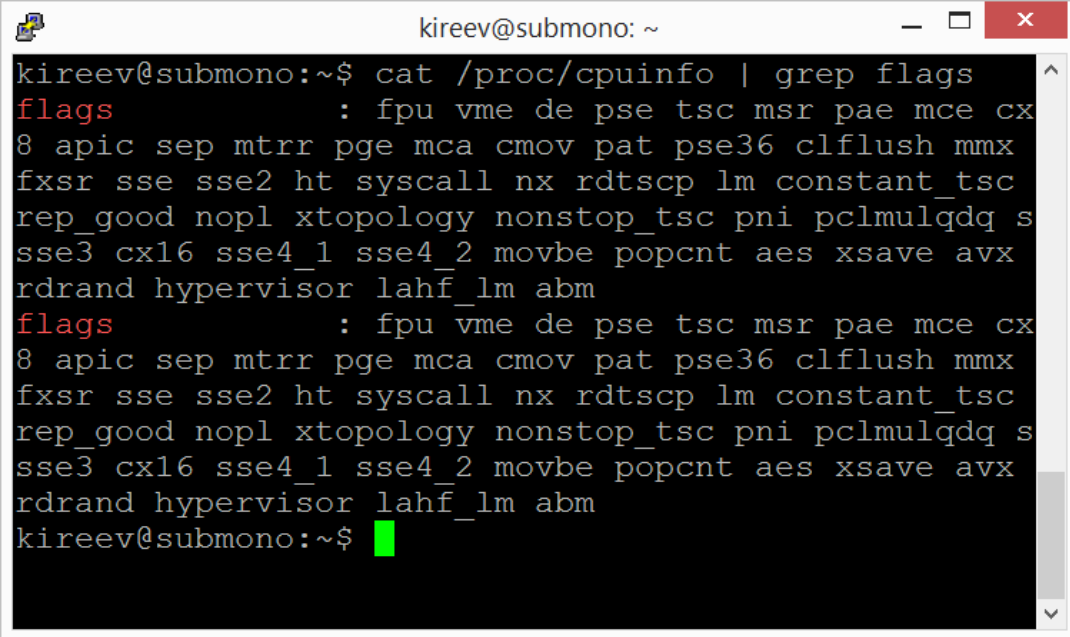
- 1997 – [MMX](#) (Pentium MMX)
 - 8 байт, целочисленные операции
- 1998 – [3DNow!](#) (K6-2)
 - 8 байт, float
- 1999 – [SSE](#) (Pentium III)
 - 16 байт, float
- 2001 – [SSE2](#) (Pentium 4)
 - 16 байт, double
- 2004 – [SSE3](#) (Pentium 4 Prescott)
 - Горизонтальные операции
- 2006 – [SSSE3](#) (Core 2, Atom)
 - Расширение набора целочисленных команд
- 2007 – [SSE4.1](#) (Penryn)
- 2008 – [SSE4.2](#) (Nehalem)
 - Специальные команды: обработка видео, обработка строк, криптография
- 2011 – [AVX](#) (Sandy Bridge)
 - 32 байта, float, double
 - Трёхоперандные команды
- 2013 – [AVX2](#) (Haswell)
 - 32 байта, целочисленные операции
 - FMA: $r = a * b + c$
- 2016 – [AVX-512](#) (KNL, Skylake)
 - 64 байта

Векторные расширения: общий взгляд

- **SSE3** (SSE, SSE2, SSE3)
 - Где: все современные микропроцессоры
 - Что: размер регистра: 16 байт
- **AVX**
 - Где: относительно старые микропроцессоры (после 2011 г.)
 - Что: размер регистра: 32 байта, float, double
- **AVX2**
 - Где: новые микропроцессоры
 - Что: размер регистра: 32 байта, FMA
- **AVX-512**
 - Где: некоторые новые микропроцессоры
 - Что: размер регистра: 64 байта

Векторные расширения: как проверить наличие

- Linux
 - `cat /proc/cpuinfo`
- Windows
 - [CPU-Z](#)

A terminal window titled 'kireev@submono: ~' showing the output of the command 'cat /proc/cpuinfo | grep flags'. The output lists various CPU features and flags, including fpu, vme, de, pse, tsc, msr, pae, mce, cx8, apic, sep, mtrr, pge, mca, cmov, pat, pse36, clflush, mmx, fxsr, sse, sse2, ht, syscall, nx, rdtscp, lm, constant_tsc, rep_good, nopl, xtopology, nonstop_tsc, pni, pclmulqdq, sse3, cx16, sse4_1, sse4_2, movbe, popcnt, aes, xsave, avx, rdrand, hypervisor, lahf_lm, and abm. The word 'flags' is highlighted in red in the original image.

```
kireev@submono:~$ cat /proc/cpuinfo | grep flags
flags           : fpu vme de pse tsc msr pae mce cx
8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc
rep_good nopl xtopology nonstop_tsc pni pclmulqdq s
sse3 cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx
rdrand hypervisor lahf_lm abm
flags           : fpu vme de pse tsc msr pae mce cx
8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc
rep_good nopl xtopology nonstop_tsc pni pclmulqdq s
sse3 cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx
rdrand hypervisor lahf_lm abm
kireev@submono:~$
```

Векторные расширения: как проверить наличие

- Linux
 - `cat /proc/cpuinfo`

```
kireev@submono: ~  
kireev@submono:~$ cat /proc/cpuinfo | grep flags  
flags           : fpu vme de pse tsc msr pae mce cx  
8 apic sep mtrr pge mca cmov pat pse36 clflush mmx  
fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc  
rep_good nopl xtopology nonstop_tsc pni pclmulqdq s  
sse3 cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx  
rdrand hypervisor lahf_lm abm  
flags           : fpu vme de pse tsc msr pae mce cx  
mov pat pse36 clflush mmx  
nx rdtscp lm constant_tsc  
onstop_tsc pni pclmulqdq s  
ovbe popcnt aes xsave avx  
abm
```

- Windows
 - [CPU-Z](#)

The screenshot shows the CPU-Z application window. The 'CPU' tab is selected, displaying the following information:

- Processor Name:** Intel Core i7 4510U
- Code Name:** Haswell ULT, Max TDP: 15.0 W
- Package:** Socket 1168 BGA
- Technology:** 22 nm, Core Voltage: 0.933 V
- Specification:** Intel® Core™ i7-4510U CPU @ 2.00GHz
- Family:** 6, **Model:** 5, **Stepping:** 1
- Ext. Family:** 6, **Ext. Model:** 45, **Revision:** C0
- Instructions:** MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3

Additional details shown include:

- Clocks (Core #0):** Core Speed: 2794.34 MHz, Multiplier: x 28.0 (8 - 31), Bus Speed: 99.80 MHz, Rated FSB: (empty)
- Cache:** L1 Data: 2 x 32 KBytes (8-way), L1 Inst.: 2 x 32 KBytes (8-way), Level 2: 2 x 256 KBytes (8-way), Level 3: 4 MBytes (16-way)

At the bottom, it shows 'Selection: Processor #1', 'Cores: 2', and 'Threads: 4'. The version is 'CPU-Z Ver. 1.77.0.x64'.

Скалярные или векторные, в чём разница?

- Скалярные инструкции: $1 \times \text{double}$
 - addsd $a + b$ latency: 3, 4 throughput: 1, 0.8
 - mulsd $a * b$ latency: 3, 5 throughput: 1, 0.5
 - divsd a / b latency: <14-22 throughput: 4-22
- Векторные SSE2: $2 \times \text{double}$
 - addpd $a + b$ latency: 3, 4 throughput: 1, 0.8
 - mulpd $a * b$ latency: 3, 5 throughput: 1, 0.5
 - divpd a / b latency: <14-22 throughput: 4-22
- Векторные AVX: $4 \times \text{double}$
 - vaddpd $a + b$ latency: 3, 4 throughput: 1
 - vmulpd $a * b$ latency: 3,4,5 throughput: 1, 0.5
 - vdivpd a / b latency: 14-35 throughput: 8-28
- Векторные AVX2: $4 \times \text{double}$
 - vfmadd132pd $a*b+c$ latency: 5 throughput: 0.5

Скалярные или векторные, в чём разница?

- Скалярные инструкции: $1 \times \text{double}$
 - addsd $a + b$ latency: 3, 4 throughput: 1, 0.8
 - mulsd $a * b$ latency: 3, 5 throughput: 1, 0.5
 - divsd a / b latency: <14-22 throughput: 4-22
- Векторные SSE2: $2 \times \text{double}$
 - addpd $a + b$ latency: 3, 4 throughput: 1, 0.8
 - mulpd $a * b$ latency: 3, 5 throughput: 1, 0.5 ×2
 - divpd a / b latency: <14-22 throughput: 4-22
- Векторные AVX: $4 \times \text{double}$
 - vaddpd $a + b$ latency: 3, 4 throughput: 1
 - vmulpd $a * b$ latency: 3,4,5 throughput: 1, 0.5 ×4
 - vdivpd a / b latency: 14-35 throughput: 8-28
- Векторные AVX2: $4 \times \text{double}$
 - vfmadd132pd $a*b+c$ latency: 5 throughput: 0.5 ×4+

План лекции

- Введение
- Обзор векторных расширений современных x86-микропроцессоров
- **Проблемы векторизации**
- Средства векторизации
- Автоматическая векторизация программ компилятором
- Полуавтоматическая векторизация с помощью OpenMP 4.0

Проблемы векторизации

- Поиск в программе однотипных операций над различными данными
(приведение к однотипным операциям)
 - Проще для операций с векторами и массивами
- Доказательство независимости операций
- Выровненный доступ к данным
- Оценка затрат на сборку-разборку векторов
 - Выигрыш должен быть больше затрат
- Переносимость
 - Какое векторное расширение использовать?
 - Многоверсионный код

План лекции

- Введение
- Обзор векторных расширений современных x86-микропроцессоров
- Проблемы векторизации
- **Средства векторизации**
- Автоматическая векторизация программ компилятором
- Полуавтоматическая векторизация с помощью OpenMP 4.0

Средства векторизации

- Вставки на ассемблере (микрокодирование)
- Векторные операции и типы данных в языке
 - Встроенные в компилятор операции (intrinsics) и типы данных
 - Классы векторных типов данных в ICC
 - Встроенные атрибуты векторных типов в GCC
- Директивы компилятора
- Векторизуемые операции с массивами
- Векторизующий компилятор
- Библиотеки векторизованных подпрограмм



Средства векторизации

Вставки на ассемблере (микрокодирование)

Где работает:

- Работает на всех компиляторах, допускающих ассемблерные вставки
- Встроенный ассемблер должен знать используемые команды

Пример: сложение двух 4-элементных векторов с использованием расширения SSE

```
typedef struct{
    float x, y, z, w;
} Vector4;

void SSE_Add(Vector4 *res, Vector4 *a, Vector4 *b){
    asm volatile ("mov %0, %%eax"::"m"(a));
    asm volatile ("mov %0, %%ebx"::"m"(b));
    asm volatile ("movups (%eax), %xmm0");
    asm volatile ("movups (%ebx), %xmm1");
    asm volatile ("addps %xmm1, %xmm0");
    asm volatile ("mov %0, %%eax"::"m"(res));
    asm volatile ("movups %xmm0, (%eax)");
}
```

Средства векторизации

Векторные операции и типы данных в языке

Встроенные в компилятор операции (intrinsics) и типы данных

- Для каждого представления векторного регистра есть свой тип данных
- Для каждой векторной команды процессора есть своя встроенная функция

Где работает:

- На большинстве известных компиляторов (gcc, clang, icc, cl.exe, ...)
- Компилятор должен поддерживать используемое векторное расширение

Пример: скалярное произведение векторов длины n, кратной 4-м, с использованием расширения SSE

```
#include <xmmintrin.h>
float inner(int n, float* x, float* y){
    __m128 *xx = (__m128*)x;
    __m128 *yy = (__m128*)y;
    __m128 s = _mm_setzero_ps();
    for(int i=0; i<n/4; ++i){
        __m128 p = _mm_mul_ps(xx[i],yy[i]);
        s = _mm_add_ps(s,p);
    }
    __m128 p = _mm_movehl_ps(p,s);
    s = _mm_add_ps(s,p);
    p = _mm_shuffle_ps(s,s,1);
    s = _mm_add_ss(s,p);
    float sum;
    _mm_store_ss(&sum,s);
    return sum;
}
```

Intel Intrinsics Guide:

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

Средства векторизации

Векторные операции и типы данных в языке

Классы векторных типов данных в Intel C++ Compiler

- Для каждого представления векторного регистра есть свой класс
- Для каждой векторной команды процессора есть свой метод
 - обёртка над SIMD intrinsics
- Дополнительные методы и операторы для работы с векторами
 - `add_horizontal`, `mul_horizontal`, `flip_sign`, `length`, `length_sqr`, `dot`, `normalize`, `<<`, `[]`, ...

Где работает:

- Intel C++ Compiler, необходимо подключить `ivec.h` / `fvec.h` / `dvec.h`

Пример: скалярное произведение векторов длины n , кратной 4-м, с использованием расширения SSE

```
#include<fvec.h>

float inner(int n, float* x, float* y) {
    F32vec4 *xx = (F32vec4*)x;
    F32vec4 *yy = (F32vec4*)y;
    F32vec4 s; s.set_zero();
    for(int i=0; i<n/4; ++i)
        s += xx[i] * yy[i];
    return add_horizontal(s);
}
```

Средства векторизации

Векторные операции и типы данных в языке

Встроенные атрибуты векторных типов в GCC

- Векторные типы данных: `__attribute__((vector_size(16)))`
- Перегруженные обычные операции: `+`, `*`, `>=`, `>>`, ...
- Встроенные операции: `__builtin_shuffle(a,b,mask)`

Где работает:

- gcc, clang

Пример: вычисление квадрата разности двух 4-элементных векторов

```
typedef float v4f __attribute__((vector_size (16)));

float inner(int n, float* x, float* y){
    v4f *xx = (v4f*)x;
    v4f *yy = (v4f*)y;
    v4f s = {0.0f, 0.0f, 0.0f, 0.0f};
    for(int i=0; i<n/4; ++i)
        s += xx[i] * yy[i];
    return s[0] + s[1] + s[2] + s[3];
}
```

GCC vector extension:

[https://gcc.gnu.org/onlinedocs/
gcc/Vector-Extensions.html](https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html)

Средства векторизации

Директивы компилятора

- Директивы распараллеливания циклов на основе OpenMP
- Программист сам следит за корректностью применения директив

Где работает:

- Intel C/C++ Compiler (#pragma simd)
- Компиляторы, поддерживающие OpenMP 4.0
 - icc, gcc-4.9 -fopenmp-simd
- Пример: скалярное произведение векторов длины n:

```
float inner(int n, float* x, float* y){  
    float s = 0.0f;  
    #pragma omp simd reduction(+:s)  
    for(int i=0; i<n; ++i)  
        s += x[i] * y[i];  
    return s;  
}
```

Intel SIMD vectorization:

<https://software.intel.com/ru-ru/node/512635>

OpenMP 4.0:

<http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>

Средства векторизации

Расширение языка

- Правила выделения секций массивов
- Скалярные операции и функции определены для массивов
- Функции для редукции по элементам массива

Где работает Cilk Plus:

- icc, gcc-4.9 -fcilkplus

Пример:

Fortran 90	Расширение Intel Cilk Plus для C/C++
<pre>real*8 a(N,N), b(N,N), c(N,N) a(1:N/2,:) = -1.0 a(N/2+1:N,:) = 1.0 b = 2.0 c = sin(a) + b*5.0</pre>	<pre>double a[N][N], b[N][N], c[N][N]; a[0:N/2][:] = -1.0; a[N/2:N/2][:] = 1.0; b[:][] = 2.0; c[:][] = sin(a[:][]) + b[:][]*5.0;</pre>

Пример: скалярное произведение векторов длины n:

```
float inner(int n, float x[n], float y[n]){
    return __sec_reduce_add(x[:] * y[:]);
}
```

Intel Cilk Plus:

<https://www.cilkplus.org/>

Средства векторизации

Векторизующий компилятор

- Компилятор распознаёт циклы, которые могут быть векторизованы, и векторизует их
- Пользователь может сообщать компилятору дополнительную информацию и пожелания с помощью директив
- Где работает:
 - gcc (циклы попроще), icc (циклы посложнее)
- Пример: скалярное произведение векторов длины n:

```
float inner(int n, float* x, float* y){
    float s = 0.0f;
    for(int i=0; i<n; ++i)
        s += x[i] * y[i];
    return s;
}
```

```
$icc -vec-report=3 test.c
```

```
...
```

```
test.c(21): (col. 3) remark: LOOP WAS VECTORIZED
```

```
...
```

Intel automatic vectorization:
<https://software.intel.com/ru-ru/node/512629>

Средства векторизации

Библиотеки векторизованных подпрограмм

- Библиотека подпрограмм, которые уже реализованы с использованием векторных расширений
- Пример: операция вычисления скалярного произведения векторов из библиотеки BLAS MKL

Где работает: везде

```
#include<mkl_blas.h>
```

```
float inner(int n, float* x, float* y) {  
    int inc = 1;  
    return SDOT(&n, x, &inc, y, &inc);  
}
```

ATLAS: <http://math-atlas.sourceforge.net/>

Intel MKL: <https://software.intel.com/en-us/intel-mkl>

AMD ACML: <http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/>

План лекции

- Введение
- Обзор векторных расширений современных x86-микропроцессоров
- Проблемы векторизации
- Средства векторизации
- **Автоматическая векторизация программ компилятором**
- Полуавтоматическая векторизация с помощью OpenMP 4.0

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы
- Цикл должен быть правильной структуры
 - `for (i=a1; i<a2; i+=a3) ...`
 - `a1, a2, a3` – целочисленные, инварианты цикла
 - Нет других точек входа и выхода (`return`, `break`, `continue`, `goto`)

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы
- Цикл должен быть правильной структуры
 - `for (i=a1; i<a2; i+=a3) ...`
 - `a1, a2, a3` – целочисленные, инварианты цикла
 - Нет других точек входа и выхода (`return`, `break`, `continue`, `goto`)
- Тело цикла должно быть простым
 - Без циклов, без сложных условных конструкций

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы
- Цикл должен быть правильной структуры
 - `for (i=a1; i<a2; i+=a3) ...`
 - a_1, a_2, a_3 – целочисленные, инварианты цикла
 - Нет других точек входа и выхода (`return`, `break`, `continue`, `goto`)
- Тело цикла должно быть простым
 - Без циклов, без сложных условных конструкций
- Итерации цикла должны быть независимыми на дистанции размера вектора

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы
- Цикл должен быть правильной структуры
 - `for (i=a1; i<a2; i+=a3) ...`
 - a_1, a_2, a_3 – целочисленные, инварианты цикла
 - Нет других точек входа и выхода (`return`, `break`, `continue`, `goto`)
- Тело цикла должно быть простым
 - Без циклов, без сложных условных конструкций
- Итерации цикла должны быть независимыми на дистанции размера вектора
- Типы данных должны быть векторизуемыми

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы
- Цикл должен быть правильной структуры
 - `for (i=a1; i<a2; i+=a3) ...`
 - `a1, a2, a3` – целочисленные, инварианты цикла
 - Нет других точек входа и выхода (`return`, `break`, `continue`, `goto`)
- Тело цикла должно быть простым
 - Без циклов, без сложных условных конструкций
- Итерации цикла должны быть независимыми на дистанции размера вектора
- Типы данных должны быть векторизуемыми
- Вызываемые функции должны иметь векторизованные варианты (Intel C/C++ Compiler)

Автоматическая векторизация программ компилятором

- Рассматриваются внутренние циклы
- Цикл должен быть правильной структуры
 - `for (i=a1; i<a2; i+=a3) ...`
 - `a1, a2, a3` – целочисленные, инварианты цикла
 - Нет других точек входа и выхода (`return`, `break`, `continue`, `goto`)
- Тело цикла должно быть простым
 - Без циклов, без сложных условных конструкций
- Итерации цикла должны быть независимыми на дистанции размера вектора
- Типы данных должны быть векторизуемыми
- Вызываемые функции должны иметь векторизованные варианты (Intel C/C++ Compiler)
- Векторизация должна быть выгодна

Автоматическая векторизация программ компилятором

- Пример 1: первый взгляд на векторизацию
 - Включение векторизации:
 - в GCC: ключи `-ftree-vectorize`, `-O3`
 - в ICC: ключи `-simd`, `-O2`

Автоматическая векторизация программ компилятором

- Пример 1: первый взгляд на векторизацию
 - Включение векторизации:
 - в GCC: ключи `-ftree-vectorize`, `-O3`
 - в ICC: ключи `-simd`, `-O2`
 - Проверка векторизации:
 - в GCC: ключи `-fopt-info-vec-optimized`, `-fopt-info-vec-missed`
 - в ICC: ключ `-vec-report=3`
 - Посмотреть команды ассемблера

Автоматическая векторизация программ компилятором

- Пример 1: первый взгляд на векторизацию
 - Включение векторизации:
 - в GCC: ключи `-ftree-vectorize`, `-O3`
 - в ICC: ключи `-simd`, `-O2`
 - Проверка векторизации:
 - в GCC: ключи `-fopt-info-vec-optimized`, `-fopt-info-vec-missed`
 - в ICC: ключ `-vec-report=3`
 - Посмотреть команды ассемблера
 - Разные векторные расширения
 - `-msse4`, `-mavx`, `-mavx2`, `-march=haswell`

Автоматическая векторизация программ компилятором

- Пример 1: первый взгляд на векторизацию
 - Включение векторизации:
 - в GCC: ключи `-ftree-vectorize`, `-O3`
 - в ICC: ключи `-simd`, `-O2`
 - Проверка векторизации:
 - в GCC: ключи `-fopt-info-vec-optimized`, `-fopt-info-vec-missed`
 - в ICC: ключ `-vec-report=3`
 - Посмотреть команды ассемблера
 - Разные векторные расширения
 - `-msse4`, `-mavx`, `-mavx2`, `-march=haswell`
 - Есть ли эффект от векторизации?

Автоматическая векторизация программ компилятором

- Пример 1: первый взгляд на векторизацию
 - Включение векторизации:
 - в GCC: ключи `-ftree-vectorize`, `-O3`
 - в ICC: ключи `-simd`, `-O2`
 - Проверка векторизации:
 - в GCC: ключи `-fopt-info-vec-optimized`, `-fopt-info-vec-missed`
 - в ICC: ключ `-vec-report=3`
 - Посмотреть команды ассемблера
 - Разные векторные расширения
 - `-msse4`, `-mavx`, `-mavx2`, `-march=haswell`
 - Есть ли эффект от векторизации?
 - Зависит от соотношения операций и обращений в память

Автоматическая векторизация программ компилятором

- Пример 1: первый взгляд на векторизацию
 - Включение векторизации:
 - в GCC: ключи `-ftree-vectorize`, `-O3`
 - в ICC: ключи `-simd`, `-O2`
 - Проверка векторизации:
 - в GCC: ключи `-fopt-info-vec-optimized`, `-fopt-info-vec-missed`
 - в ICC: ключ `-vec-report=3`
 - Посмотреть команды ассемблера
 - Разные векторные расширения
 - `-msse4`, `-mavx`, `-mavx2`, `-march=haswell`
 - Есть ли эффект от векторизации?
 - Зависит от соотношения операций и обращений в память
 - Зависит от векторизуемых операций

Автоматическая векторизация программ компилятором

- Пример 2: выравнивание данных

– Хорошее: $y[i] = 2.5 * x[i] + y[i];$

Автоматическая векторизация программ компилятором

- Пример 2: выравнивание данных
 - Хорошее: $y[i] = 2.5 * x[i] + y[i];$
 - Не очень хорошее: $y[i] = 2.5 * x[i+1] + y[i];$

Автоматическая векторизация программ компилятором

- Пример 2: выравнивание данных
 - Хорошее: $y[i] = 2.5 * x[i] + y[i];$
 - Не очень хорошее: $y[i] = 2.5 * x[i+1] + y[i];$
 - Не очень хорошее: $y[i] = 2.5 * x[2 * i] + y[i];$

Автоматическая векторизация программ компилятором

- Пример 3: функции в цикле

Автоматическая векторизация программ компилятором

- Пример 3: функции в цикле

– Нет векторных аналогов в GCC: $\sin(x)$

Автоматическая векторизация программ компилятором

- Пример 3: функции в цикле
 - Нет векторных аналогов в GCC: $\sin(x)$
 - Есть векторные, но не используются: \sqrt{x}

Автоматическая векторизация программ компилятором

- Пример 3: функции в цикле
 - Нет векторных аналогов в GCC: $\sin(x)$
 - Есть векторные, но не используются: \sqrt{x}
 - Используются, если понизить требования к вещественной арифметике: ключи `-ffast-math`, `-Ofast`

Автоматическая векторизация программ компилятором

- Пример 3: функции в цикле
 - Нет векторных аналогов в GCC: `sin(x)`
 - Есть векторные, но не используются: `sqrt(x)`
 - Используются, если понизить требования к вещественной арифметике: ключи `-ffast-math`, `-Ofast`
 - ICC автоматически использует библиотеку векторизованных операций `SVML`

Автоматическая векторизация программ компилятором

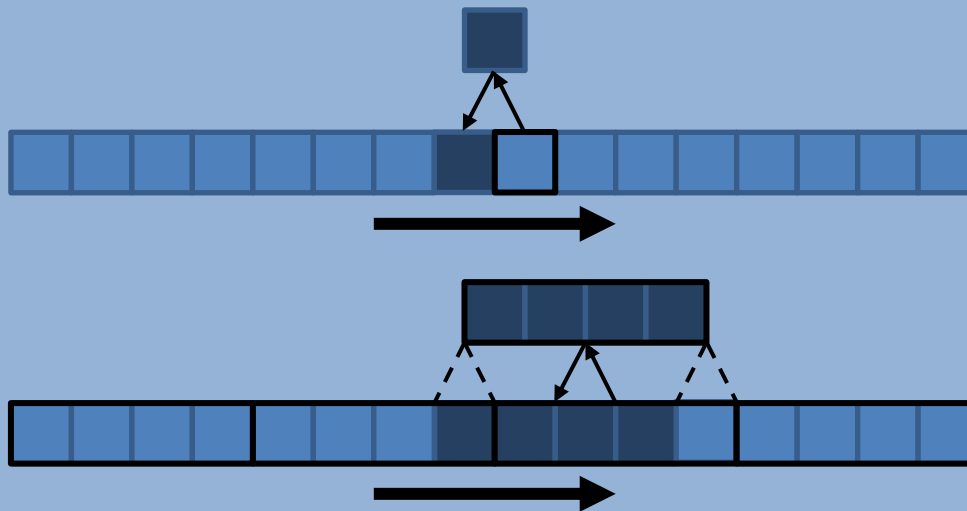
- Пример 3: функции в цикле
 - Нет векторных аналогов в GCC: `sin(x)`
 - Есть векторные, но не используются: `sqrt(x)`
 - Используются, если понизить требования к вещественной арифметике: ключи `-ffast-math`, `-Ofast`
 - ICC автоматически использует библиотеку векторизованных операций `SVML`
 - Пользовательские функции – векторизуются, если простые и подставленные вместо вызова (`inline`)

Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями
 - $y[i] = 2.5 * x[i] + y[i+1];$

Автоматическая векторизация программ компилятором

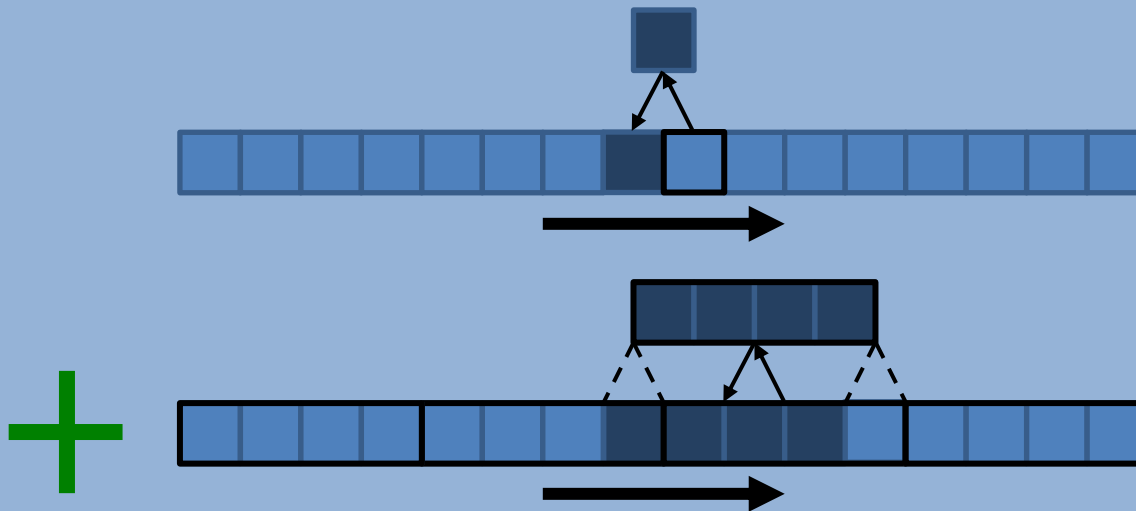
- Пример 4: зависимости между итерациями
 - $y[i] = 2.5 * x[i] + y[i+1];$



Автоматическая векторизация программ компилятором

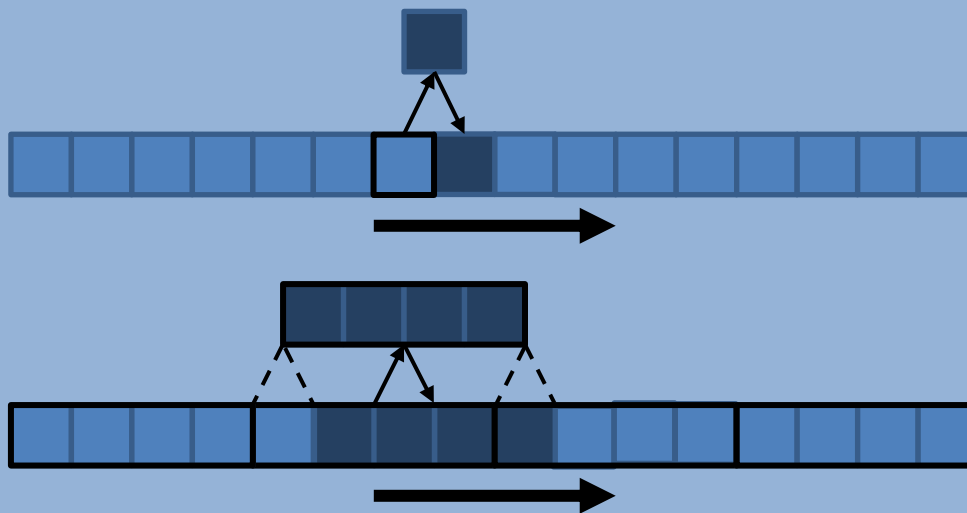
- Пример 4: зависимости между итерациями

– $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости



Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями
 - $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости
 - $y[i] = 2.5 * x[i] + y[i-1];$

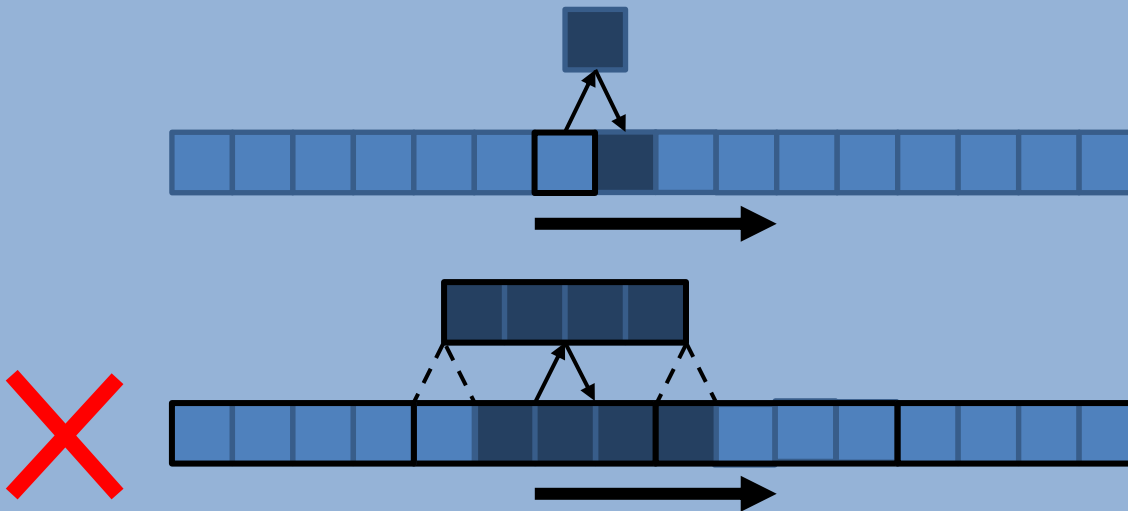


Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями

– $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-1];$ // Есть зависимость



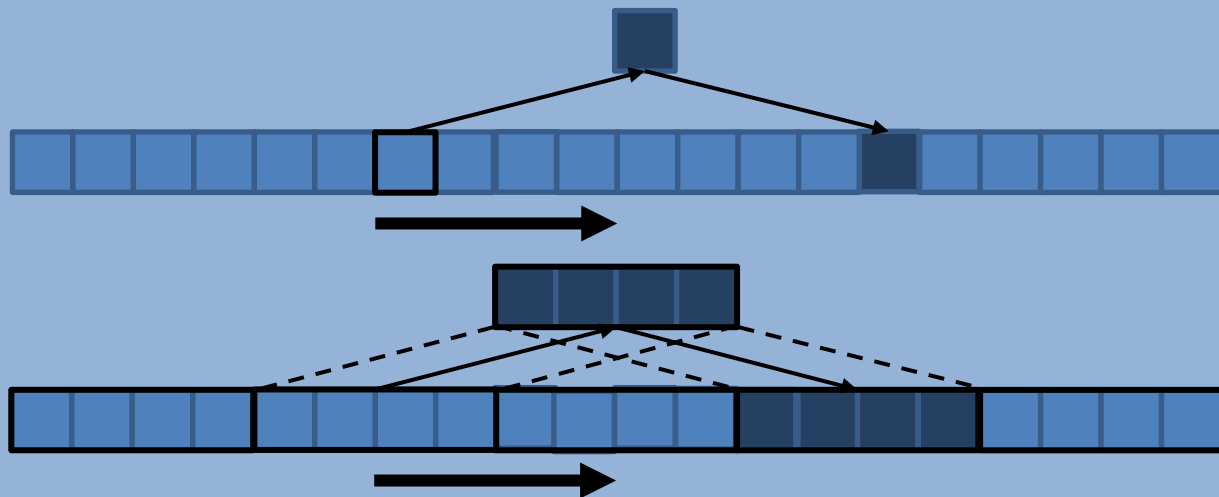
Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями

– $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-1];$ // Есть зависимость

– $y[i] = 2.5 * x[i] + y[i-8];$



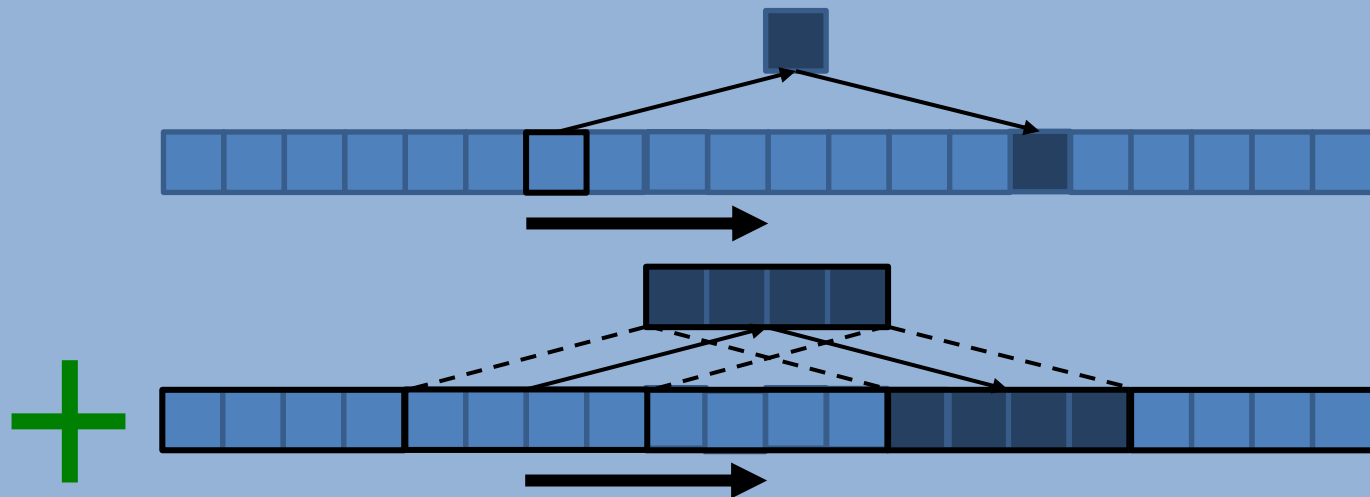
Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями

– $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-1];$ // Есть зависимость

– $y[i] = 2.5 * x[i] + y[i-8];$ // Нет зависимости



Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями

– $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-1];$ // Есть зависимость

– $y[i] = 2.5 * x[i] + y[i-8];$ // Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-3];$

Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями

– $y[i] = 2.5 * x[i] + y[i+1];$

// Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-1];$

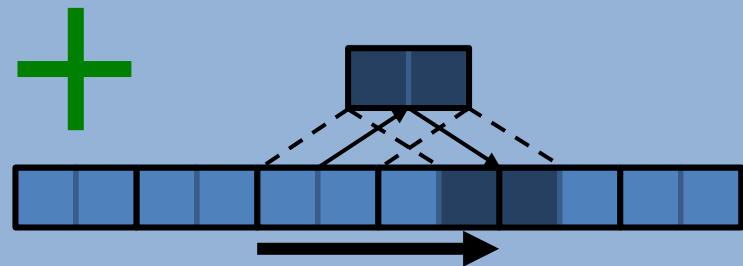
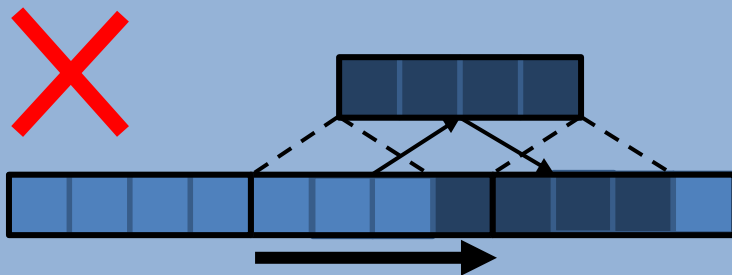
// Есть зависимость

– $y[i] = 2.5 * x[i] + y[i-8];$

// Нет зависимости

– $y[i] = 2.5 * x[i] + y[i-3];$

// Зависит от длины вектора



Автоматическая векторизация программ компилятором

- Пример 4: зависимости между итерациями
 - $y[i] = 2.5 * x[i] + y[i+1];$ // Нет зависимости
 - $y[i] = 2.5 * x[i] + y[i-1];$ // Есть зависимость
 - $y[i] = 2.5 * x[i] + y[i-8];$ // Нет зависимости
 - $y[i] = 2.5 * x[i] + y[i-3];$ // Зависит от длины вектора
 - Если у компилятора есть сомнения, то он может сделать несколько версий кода.

Автоматическая векторизация программ компилятором

- Пример 5: условие в цикле

Автоматическая векторизация программ компилятором

- Пример 5: условие в цикле
 - GCC не векторизует
 - ICC векторизует в случае простых конструкций

Автоматическая векторизация программ компилятором

- Пример 5: условие в цикле
 - GCC не векторизует
 - ICC векторизует в случае простых конструкций
- Пример 6: редукция

Автоматическая векторизация программ компилятором

- Пример 5: условие в цикле
 - GCC не векторизует
 - ICC векторизует в случае простых конструкций
- Пример 6: редукция
 - GCC не распознаёт редукцию
 - ICC распознаёт редукцию

Автоматическая векторизация программ компилятором

- Итого: что может ухудшить или не дать выполнить автоматическую векторизацию
 - Плохое выравнивание данных
 - Вызов функций в цикле
 - Наличие зависимостей
 - Наличие условных конструкций
 - Редукция

Автоматическая векторизация программ компилятором

- Когда цикл векторизуется?
 1. Компилятор доказал отсутствие цикловых зависимостей, используя только доказанные факты о программе.
 2. Компилятор доказал отсутствие цикловых зависимостей, сделав худшие предположения там, где был недостаток доказанных фактов.
 3. Компилятор считает, что векторизация будет выгодной.

Автоматическая векторизация программ компилятором

- Когда цикл векторизуется?
 1. Компилятор доказал отсутствие цикловых зависимостей, используя только доказанные факты о программе.
 2. Компилятор доказал отсутствие цикловых зависимостей, сделав худшие предположения там, где был недостаток доказанных фактов.
 - ~~3. Компилятор считает, что векторизация будет выгодной.~~

Считать, что векторизация всегда выгодна:

ICC: `#pragma vector always`

GCC: `???`

Автоматическая векторизация программ компилятором

- Когда цикл векторизуется?
 1. Компилятор доказал отсутствие цикловых зависимостей, используя только доказанные факты о программе.
 - ~~2. Компилятор доказал отсутствие цикловых зависимостей, сделав худшие предположения там, где был недостаток доказанных фактов.~~
 3. Компилятор считает, что векторизация будет выгодной.

Считать, что скрытых зависимостей нет:

ICC: `#pragma ivdep`

GCC: `#pragma GCC ivdep`

Автоматическая векторизация программ компилятором

- Когда цикл векторизуется?
 - ~~1. Компилятор доказал отсутствие цикловых зависимостей, используя только доказанные факты о программе.~~
 - ~~2. Компилятор доказал отсутствие цикловых зависимостей, сделав худшие предположения там, где был недостаток доказанных фактов.~~
 - ~~3. Компилятор считает, что векторизация будет выгодной.~~

Векторизовать в любом случае, даже если это испортит программу:

ICC: `#pragma omp simd / #pragma simd`

GCC: `#pragma omp simd`

План лекции

- Введение
- Обзор векторных расширений современных x86-микропроцессоров
- Проблемы векторизации
- Средства векторизации
- Автоматическая векторизация программ компилятором
- **Полуавтоматическая векторизация с помощью OpenMP 4.0**

Полуавтоматическая векторизация с помощью OpenMP 4.0

- В стандарт OpenMP 4.0 включён набор директив по векторизации циклов
 - `#pragma omp simd ...`
 - `#pragma omp declare simd ...`
- OpenMP 4.0 поддерживают:
 - Intel C/C++ Compiler с версии 15.0
 - GCC с версии 4.9

Полуавтоматическая векторизация с помощью OpenMP 4.0

- Пример: Базовая директива `simd`
 - Есть сомнение, можно ли векторизовать:
for (i=1;i<N-1;i++)
y[i] = 1.1*x[i-1] + 2.5*x[i] + 3.7*x[i+1];

Полуавтоматическая векторизация с помощью OpenMP 4.0

- Пример: Базовая директива `simd`
 - Есть сомнение, можно ли векторизовать:

```
for (i=1;i<N-1;i++)  
  y[i] = 1.1*x[i-1] + 2.5*x[i] + 3.7*x[i+1];
```
 - Указание: векторизовать не смотря ни на что!
`#pragma omp simd`

```
for (i=1;i<N-1;i++)  
  y[i] = 1.1*x[i-1] + 2.5*x[i] + 3.7*x[i+1];
```

Полуавтоматическая векторизация с помощью OpenMP 4.0

- Пример: Редукция
 - GCC автоматически не векторизует

```
s = 0.0;
for (i=1;i<N-1;i++) s += y[i] + 2.5*x[i];
```


Полуавтоматическая векторизация с помощью OpenMP 4.0

- Пример: Редукция

- GCC автоматически не векторизует

- `s = 0.0;`

- `for (i=1;i<N-1;i++) s += y[i] + 2.5*x[i];`

- Указание, как это векторизовать

- `s = 0.0;`

- `#pragma omp simd reduction(+:s)`

- `for (i=1;i<N-1;i++) s += y[i] + 2.5*x[i];`

Полуавтоматическая векторизация с помощью OpenMP 4.0

- Пример: Пользовательские функции
 - Если неочевидно, как сделать векторный аналог:
double f(double u, double v, int k) { return 2.0*u + k*v; }
...
for (i=0;i<N;i++) y[i] = 2.5*x[i] + f(y[i], 4.0, i);

Полуавтоматическая векторизация с помощью OpenMP 4.0

- Пример: Пользовательские функции
 - Если неочевидно, как сделать векторный аналог:

```
double f(double u, double v, int k) { return 2.0*u + k*v; }  
...  
for (i=0;i<N;i++) y[i] = 2.5*x[i] + f(y[i], 4.0, i);
```
 - Указание, как сделать векторный аналог функции:

```
#pragma omp declare simd uniform(v) linear(k,1)  
double f(double u, double v, int k) { return 2.0*u + k*v; }  
...  
#pragma omp simd  
for (i=0;i<N;i++) y[i] = 2.5*x[i] + f(y[i], 4.0, i);
```

