Новосибирский государственный университет
Факультет информационных технологий
Кафедра параллельных вычислений

Ускоритель Intel Xeon Phi

Векторизация вычислений

Преподаватель:
Киреев С.Е.

Цель векторизации

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

Цель векторизации

- Цели
  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

Проблемы векторизации

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

СРЕДСТВА ВЕКТОРИЗАЦИИ

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

Xeon Phi

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

Xeon Phi

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 / micvec.h

Xeon Phi

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

~~Xeon Phi~~

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

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

Xeon Phi

- 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

Векторизуемые операции с массивами

- 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

Xeon Phi

Пример:

Fortran 90	Расширение Intel Cilk Plus для C/C++

`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`
	`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;`

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)

Xeon Phi

- 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

Xeon Phi

```
#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/

АВТОМАТИЧЕСКАЯ ВЕКТОРИЗАЦИЯ В INTEL C/C++ COMPILER

Автоматическая векторизация в Intel C/C++ Compiler

- Ограничения на автоматическую векторизацию
  - Рассматриваются только самые внутренние циклы
  - Цикл должен правильной структуры: for (i=0;i<N;i++)
    - Границы и шаг цикла – инварианты (не меняются внутри или снаружи)
    - Тело цикла не должно иметь других точек входа и выхода (return, break, continue, goto, …)
  - Тело цикла не должно быть слишком сложным
  - Итерации цикла должны быть независимыми на дистанции размера вектора (компилятор может генерировать несколько версий кода)
  - Типы данных должны быть векторизуемыми
  - Вызываемые функции должны иметь векторизованные варианты
  - Векторизация цикла должна быть выгодна

Зависимости по данным

- Flow (True) dependence – Read After Write
  - X := 10
  - Y := X + C

  нельзя устранить
- Anti dependence – Write After Read
  - X := Y + C
  - Y := 10

  можно устранить
- Output dependence – Write After Write
  - X := 10
  - X := 20

  можно устранить
- Input dependence – Read After Read
  - Y := X + 3
  - Z := X + 5

  не нужно устранять

Автоматическая векторизация в Intel C/C++ Compiler

- Распараллеливание возможно, если нет цикловых зависимостей между итерациями
- Векторизация возможна, если нет цикловых зависимостей между каждым выражением в теле цикла с глубиной меньшей или равной длине вектора

Пример 1:

```
for (i=0; i<N; i++)
{
  y[i] = a*x[i] + b;
}
```

Возможна векторизация и распараллеливание

Пример 2:

```
for (i=5; i<N; i++)
{
  y[i] = a*x[i] + b*y[i-5];
}
```

Возможна векторизация с длиной вектора не больше 4

Автоматическая векторизация в Intel C/C++ Compiler

- Когда код 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 в Intel C/C++ Compiler

- Когда код 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.~~

`#pragma vector always`

Считать, что векторизация всегда выгодна

Автоматическая векторизация в Intel C/C++ Compiler

- Когда код 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.

`#pragma ivdep`

Считать, что скрытых зависимостей нет

Автоматическая векторизация в Intel C/C++ Compiler

- Когда код 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.~~

`#pragma simd / #pragma omp simd`

Векторизовать в любом случае,

даже если это испортит программу

Автоматическая векторизация в Intel C/C++ Compiler

- Как посмотреть:
  - векторизовался ли код?
  - почему не векторизовался?

- Сообщения компилятора:
  - Ключ: icc –vec-report=3

- Посмотреть ассемблерный листинг
  - Ключ: icc –S

Автоматическая векторизация в Intel C/C++ Compiler

- Пример:

```
void vsum(int n, double *x, double *y, double *z)
{ int i;

  for (i=0;i<n;i++) z[i] = x[i] + y[i];
}
```

- Компилятор сгенерирует несколько версий кода:
  - Скалярную для случая, если указатели пересекаются
  - Векторную для случая, если указатели не пересекаются

Автоматическая векторизация в Intel C/C++ Compiler

- Пример:

```
void vsum(int n, double *x, double *y, double *z)
{ int i;
  #pragma ivdep
  for (i=0;i<n;i++) z[i] = x[i] + y[i];
}
```

- Компилятор сгенерирует векторный код

Автоматическая векторизация в Intel C/C++ Compiler

- Пример:

```
double add(double a, double b) { return a+b; }

void vsum(int n, double *x, double *y, double *z)
{
  int i;
  for (i=0;i<n;i++) z[i] = add(x[i],y[i]);
}
```

Автоматическая векторизация в Intel C/C++ Compiler

- Пример:

```
__attribute__((vector))
double add(double a, double b) { return a+b; }

void vsum(int n, double *x, double *y, double *z)
{ int i;
  #pragma ivdep
  for (i=0;i<n;i++) z[i] = add(x[i],y[i]);
}
```

Автоматическая векторизация в Intel C/C++ Compiler

- Пример: в нотации Intel C/C++ Compiler

```
__attribute__((vector))
double add(double a, double b) { return a+b; }

void vsum(int n, double *x, double *y, double *z)
{ int i;
  #pragma simd
  for (i=0;i<n;i++) z[i] = add(x[i],y[i]);
}
```

- Пример: в нотации OpenMP 4.0

```
#pragma omp declare simd
double add(double a, double b) { return a+b; }

void vsum(int n, double *x, double *y, double *z)
{ int i;
  #pragma omp simd
  for (i=0;i<n;i++) z[i] = add(x[i],y[i]);
}
```

Автоматическая векторизация в Intel C/C++ Compiler

- Пример: в нотации OpenMP 4.0

```
#pragma omp declare simd uniform(c)
double add(double a, double b, double c)
{ return a + b + c; }

void vsum(int n, double *x, double *y, double *z)
{ int i;
  #pragma omp simd
  for (i=0;i<n;i++) z[i] = add(x[i],y[i],0.5);
}
```

ОБЗОР СРЕДСТВ УПРАВЛЕНИЯ ВЕКТОРИЗАЦИЕЙ В INTEL C/C++ COMPILER

Автоматическая векторизация в Intel C/C++ Compiler

Ключи компилятора для управления векторизацией

- -guide-vec[=#] – включить подсказки по векторизации, #=1..4
- -vec-report[=#] – выдавать сообщения автовекторизатора, #=0..7
- -no-vec – отключить автовекторизацию
- -no-simd – отключить использование pragma simd
- -[no-]openmp-simd – включить/отключить использование pragma omp simd

Вспомогательные ключи компилятора

- -opt-report[=#] – выдавать сообщения оптимизатора, #=0,1,2,3
- -restrict – допускать использование ключевого слова restrict
- -ipo – анализировать программу в целом, а не по функциям
- -O3 – включить высокоуровневые оптимизации
- -fno-alias / -fargument-noalias / -ansi-alias – указание, что в программе нет перекрытия указателей

Автоматическая векторизация в Intel C/C++ Compiler

Директивы для управления векторизацией циклов

- #pragma novector – не векторизовать цикл
- #pragma vector always – векторизовать цикл независимо от оценки производительности
- #pragma vector [no]vecremainder – векторизовать ли остаток цикла
- #pragma simd – векторизовать цикл в любом случае
  - vectorlength(#1[,#2]…) – задание допустимой длины вектора в элементах
  - vectorlength(type) – задание допустимой длины вектора как size_of_vector_register/sizeof(type)
  - private (v1[,v2]…) – как в OpenMP
  - firstprivate (v1[,v2]…) – как в OpenMP
  - lastprivate (v1[,v2]…) – как в OpenMP
  - reduction (op:v1[,v2]…) – как в OpenMP
  - linear (v1:step1[,v2:step2]…) – линейное изменение значения переменной: v1+=step1, …
  - assert – генерировать «assertion failed» когда векторизация не удалась
  - [no]vecremainder – векторизовать ли остаток цикла

Вспомогательные директивы

- #pragma ivdep – игнорировать потенциальные зависимости между итерациями цикла
- #pragma loop count(#) – задать оценку числа итераций цикла

Автоматическая векторизация в Intel C/C++ Compiler

Атрибуты функций: __attribute__((param1[,param2,]…))

- vector – сгенерировать векторный аналог функции
- vector(clause1[,clause2]…)
  - processor(cpuid) – сгенерировать код для конкретного процессора (pentium4, …, core_4th_gen_avx, mic)
  - vectorlength(#) – задание допустимой длины вектора в элементах
  - linear(param1:step1[,param2:step2]…) – линейное изменение значения переменной в векторе с заданным шагом
  - uniform(param1[,param2]…) – считать, что заданный параметр имеет одно и то же значение во всём векторе
  - mask – сгенерировать маскированную версию подпрограммы

Литература

- **User and Reference Guide for the Intel® C++ Compiler 14.0**
  - Google: «intel compiler 14 reference manual» 1-я ссылка
  - https://software.intel.com/en-us/compiler_14.0_ug_c


NEC
SX9