Средства и системы параллельного программирования

сентябрь – декабрь 2021 г.

Лектор доцент Попова Нина Николаевна

Лекция 2

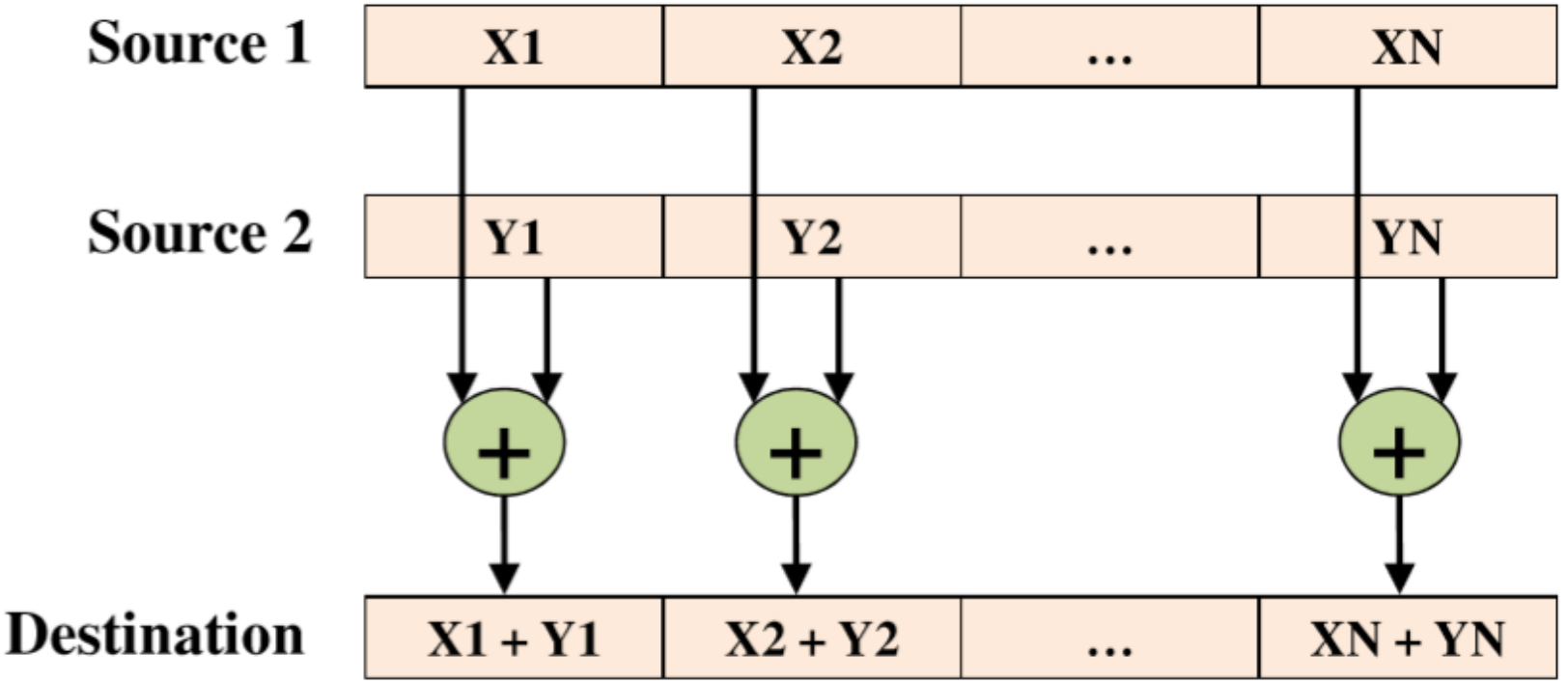
13 сентября 2021 г.

План лекции

- Параллелизм уровня данных. Векторные инструкции.
- Векторизация программ

Параллелизм уровня данных (Data Parallelism –DP)

Векторные (SIMD) инструкции



SIMD –Single Instrucrion Stream, Multiple Data Stream

Современные процессоры

-Single Instruction Multiple Data(SIMD) операции позволяют одновременное выполнение одной и той же инструкции с использование «широких» регистров.

- “SIMD width” – число операндов, которые могут быть размещены в регистре

- Максимальное ускорение равно количеству элементов в векторном регистре

Векторные расширения

В современных скалярных микропроцессорах общего назначения векторные вычисления поддержаны с помощью **векторных расширений архитектуры**

- Примеры векторных расширений: MMX, SSE, AVX, …
- Векторные расширения включают:
  - Векторные регистры – хранят множества скалярных значений
  - Векторные команды (инструкции) – для работы с векторными регистрами

Векторные расширения

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

SIMD-инструкции в составе наборов команд

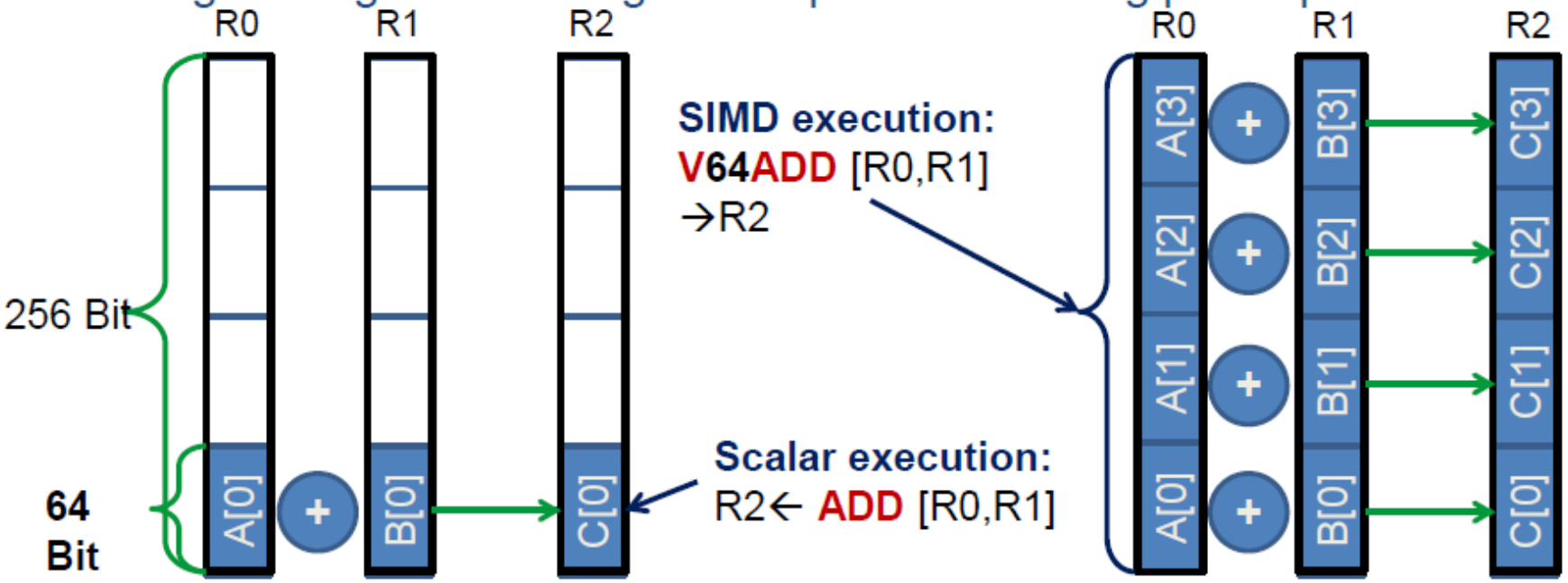
- Intel MMX (1996)
- IBM POWER AltiVec(1999)
- AMD 3DNow! (1998)
- Intel SSE (Intel Pentium III, 1999)
- Intel SSE2, SSE3, SSE4
- AVX (Advanced Vector Extension, Intel & AMD, 2008)

AVX2 (Haswell, 2013)

- AVX-512 (2015)
- ARM Advanced SIMD (NEON) –Cortex-A8, 2011

Пример выполнения векторных инструкций

- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX(/2): register width = 256 Bit → 4 double precision floating point operands
  - AXV512: you get it.
- Adding two registers holding double precision floating point operands



Как проверить векторные расширения

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts nopl xtopology nonstop_tsc
aperfmperf pni pclmulqdq dtes64 ds_cpl vmx smx est tm2
ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt
tsc_deadline_timer xsave avx lahf_lm ida arat epb xsaveopt
pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
```

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

Использование инструкций SSE

Ассемблер

Встроенные функции компилятора (Intrinsic)

C++ классы

Автоматическая векторизация компилятора

Лучшая управляемость

Простота использования

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

SSE Intrinsics(builtin functions)

- **Заголовочные файлы:**
  - `#include <mmintrin.h>  /* MMX */`
  - `#include <xmmintrin.h>  /* SSE */`
  - `#include <emmintrin.h>  /* SSE2 */`
  - `#include <pmmintrin.h>  /* SSE3 */`
  - `#include <smmintrin.h>  /* SSE4 */`
  - `#include <immintrin.h>  /* AVX */`

SSE Intrinsics: типы данных

- `__m128  /* float[4] */`
- `__m128d  /* double[2] */`
- `__m128i  /* integer: byte[8], int[4] */`
- `__m64  /* MMX integer SIMD */`

SSE Intrinsics

```
#include <xmmintrin.h>    /* SSE */

void add(float *a, float *b, float *c)
{
    __m128 t0, t1;

    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

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.

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

GCC директивы

GCC прагмы векторизации

#pragma GCC ivdep

: - уведомление компилятору о том, что в цикле нет зависимостей по данным

Векторные операции и типы данных в языке. Пример.

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

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

- 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

–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

-march=native
- Использовать инструкции, поддерживаемые локальным CPU

–Есть ли эффект от векторизации?
- Зависит от соотношения операций и обращений в память
- Зависит от векторизуемых операций

Автоматическая векторизация программ компилятором

Что может ухудшить или не дать выполнить автоматическую векторизацию:

- –Плохое выравнивание данных
- –Вызов функций в цикле
- –Наличие зависимостей
- –Наличие условных конструкций
- –Редукция

Пример автовекторизации. Простой цикл.

```
1  #define SIZE     (1L << 16)
2  void simpleLoop(double * a, double * b)
3  {
4       for (int i = 0; i < SIZE; i++)
5       {
6            a[i] += b[i];
7       }
8  }
```

Результат

simpleLoop . c : 4 : 5 : note : l o op v e c t o r i z e d
simpleLoop . c : 4 : 5 : note : l o op v e r s i o n e d f o r v e c t o r i z a t i o n becaus e o f p o s s i b l e a l i a s i n g
simpleLoop . c : 4 : 5 : note : l o op p e e l e d f o r v e c t o r i z a t i o n to enhance a l i g nme n t

Пример автовекторизации. Простой цикл. Вариант 2.

```
1  #define SIZE     (1L << 16)
2  void improvedLoop(double * restrict a, double *
       restrict b)
3  {
4       for (int i = 0; i < SIZE; i++)
5       {
6            a[i] += b[i];
7       }
8  }
```

Результат 2

```
improvedLoop.c:4:5: note: loop vectorized
improvedLoop.c:4:5: note: loop peeled for
      vectorization to enhance alignment
```

Пример автовекторизации. Простой цикл. Вариант 3.Оптимизированный.

```
1  #define SIZE    (1L << 16)
2  #define GCC_ALN(var, alignment)
       __builtin_assume_aligned(var, alignment)
3  void optimizedLoop(double * restrict a, double *
       restrict b)
4  {
5      a = (double *) GCC_ALN(a, 32);
6      b = (double *) GCC_ALN(b, 32);
7      for (int i = 0; i < SIZE; i++)
8      {
9          a[i] += b[i];
10     }
11 }
```

Результат 3

```
optimizedLoop.c:7:5: note: loop vectorized
```

```
.L2:
    vmovapd ymm0, YMMWORD PTR [rdi+rax]
    vaddpd ymm0, ymm0, YMMWORD PTR [rsi+rax]
    vmovapd YMMWORD PTR [rdi+rax], ymm0
    add rax, 32
    cmp rax, 524288
    jne .L2
```

Пример автовекторизации. Простой цикл. Вариант 4. C11

C11 compatible solution

```
1  struct data{
2      alignas(32) double vec[SIZE];
3  };
4  void optimizedLoop(struct data * restrict a,
       struct data * restrict b)
5  {
6      for (int i = 0; i < SIZE; i++)
7          a->vec[i] += b->vec[i];
8  }
```

Сравнение вариантов

Simple Loop	106.442
Improved Loop	105.883
Optimized Loop	99.719
Optimized Loop C11	99.540
Non-vectorized Loop	444.142