Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

Лекция №4
Векторные расширения Intel Xeon Phi

При поддержке компании Intel

Мееров И.Б.
Кафедра математического обеспечения ЭВМ

Содержание

- Введение
- Векторные расширения. Краткий обзор
  - Типы данных и регистровый пул
  - Обзор основных типов операций
  - Расширенная поддержка математических функций
- Векторизация в программах на языке высокого уровня
- Векторизация и математические функции
- Заключение
- Литература

Введение

Введение…

- Начиная с середины XX века, программирование проделало большой путь и постоянно продолжает развиваться
  - программирование в машинных кодах,
  - программирование на ассемблере,
  - программирование на языках высокого уровня.
- Оптимизирующие компиляторы (Intel C/C++ Comiler, Intel Fortran Compiler и др.) умеют
  - *использовать новые наборы команд*;
  - генерировать код, ***ориентированный на современные центральные процессоры***.

Введение

- Парадигма **SIMD** – single instruction multiple data
- MMX, SSE, SSE2, SSE3…, SSE4, AVX, векторные расширения Intel Xeon Phi

1.25	2.25	2.0	1.0
+			
2.0	1.15	1.25	2.0
=			
3.25	3.4	3.25	3.0

- Специальные регистры, специальные векторные инструкции (арифметика, работа с памятью…).

Как этим воспользоваться?

Векторные расширения. Краткий обзор

Введение

- Векторные расширения присутствуют в наборах команд процессоров различных производителей и архитектур.
  - Intel: MMX – SSE – SSE2 – SSE3 – SSE4 – AVX – расширения в XeonPhi.
  - AMD: 3DNow!
  - ARM: NEON (во встраиваемых системах)
- Рассмотрим кратко векторные расширения, реализованные в Intel Xeon Phi, чтобы составить общее впечатление о сути вопроса для лучшего понимания того, как соотносится расчетный код на C/C++ и машинный/ассемблерный коды, порождаемые компилятором.

Типы данных и регистровый пул

- В каждом ядре Xeon Phi сконструирован специальный модуль векторной обработки (VPU, vector processor unit)
  - 32 512-разрядных zmm-регистра; двукратное увеличение размера по сравнению с AVX
  - векторный FMA, fused multiply–add: ***a = a + b * c*** с однократным округлением.
- Одновременные действия над 16 32-битными целыми числами или 8 64-битными целыми числами или 16 числами с плавающей запятой одинарной точности или 8 числами с плавающей запятой двойной точности. Поддерживаются операции с комплексными данными.
- Большинство операций – тернарные (2 аргумента и один результат). По некоторым данным это приводит к 20% приросту производительности.

Обзор основных типов операций

- ***Арифметические операции***: сложение, вычитание, умножение, деление, FMA (для вычислений с плавающей запятой).
- ***Операции преобразования типов***, позволяющие выполнять повышающие и понижающие преобразования согласно определенным правилам (см. [1, 3]).
- ***Логические операции***, позволяющие выполнять векторные сравнения, находить минимум и максимум и т.д.
- ***Операции доступа к данным*** (загрузка/выгрузка память/регистр; scatter/gather, предвыборка, streaming stores). Могут применяться *маскирование*, *swizzle*, *shuffle*.

Расширенная поддержка математических функций

- В рамках Intel Xeon Phi была реализована расширенная поддержка некоторых математических функций:
  Команды для вычисления в одинарной точности:

Функция	Латентность	Пропускная способность
1/x	4	1
1/sqrt(x)	4	1
$\log_2(x)$	4	1
2^x	8	2

- **sqrt(x)**, **a^x** и **деление** могут быть вычислены при помощи указанных функций.

Векторизация в программах на языке высокого уровня

Введение

- Вычислительные ядра Intel Xeon Phi могут выполнять однотипные вычисления,
  - оперируя векторами из целых чисел или чисел с плавающей запятой,
  - обладают широким спектром
    - математических операций,
    - логических операций,
    - операций с битами,
    - операций работы с памятью.

- **Как написать программу, чтобы задействовать эти возможности?**

Способы векторизации

1. Использовать высокопроизводительные специализированные библиотеки, эффективно использующие векторные инструкции.
2. Написать программу на C/C++ или Fortran и откомпилировать ее тем транслятором, который «знает» про соответствующие наборы команд.
3. Использовать специальные ключи и директивы компилятора (подсказки).
4. Использовать возможности Array Notation и Elemental Function в рамках технологии Intel Cilk Plus.
5. Использовать классы интринсиков для SIMD.
6. Использовать векторные функции-интринсики.
7. Написать реализацию на ассемблере.

Векторизация. Используем векторизованные библиотеки

- **MKL** и не только.

- **Проблемы**:
  - Не все, что нам нужно, есть в библиотеке.
  - Реализация, присутствующая в библиотеке, не всегда оптимальна для нашей конкретной задачи.
  - Сложности интеграции, поддержки, миграции на другие платформы.

Векторизация. Используем C/C++ или Fortran в сочетании с оптимизирующим компилятором

Intel Compliers (предпочтительно), **gcc**

- Крайне желательно минимизировать всевозможные зависимости по данным.
- Нежелательно вызывать функции в вычислительно трудоемких циклах. Однако вызов функции тоже не приговор. Компилятор может встроить код функции и успешно векторизовать цикл.
- Необходимо следить за выравниванием данных в памяти, то есть за их размещением с «правильных» адресов, кратным определенному числу байт (размеру кеш-линии, размеру векторного регистра).

  SSE: по 16, AVX: по 32, Xeon Phi: по 64

  __declspec(aligned), __mm_malloc()…

Векторизация. Используем C/C++ или Fortran в сочетании с оптимизирующим компилятором

- Необходимо стараться обеспечить однородный доступ к памяти, когда мы загружаем/выгружаем данные, лежащие последовательно (крайне желательно) либо с одинаковым шагом (допустимо).
- Необходимо сводить к минимуму смешивание объектов разных типов данных в выражениях.
- Необходимо по возможности избавляться от условных операторов в теле внутреннего цикла. Раньше компилятор в принципе отказывался векторизовывать такие циклы. Сейчас в ряде случаев ему удается это сделать.

Примеры:

Векторизация. Используем C/C++ или Fortran в сочетании с оптимизирующим компилятором

Пример 1:

```
#pragma simd
#pragma vector aligned
for (int i = 0; i < n; i++) {  s = s + max(a[i],0); }
```

Пример 2:

```
#pragma simd
#pragma vector aligned
for (int i = 0; i < n; i++)
  if (a[i] == key) {
    Index = i; break;
  }
```

Векторизация. Используем C/C++ или Fortran в сочетании с оптимизирующим компилятором

- по возможности не использовать объектно-ориентированные конструкции в расчетах; существуют способы умелого сочетания высокоуровневой объектно-ориентированной архитектуры и низкоуровневого программирования вычислительно трудоемких участков кода;
- стремиться к достаточно существенному объему работы во внутренних циклах; заметим, что компилятор преимущественно векторизует именно внутренние циклы, поэтому необходим простор для его деятельности.

Векторизация. Используем C/C++ или Fortran в сочетании с оптимизирующим компилятором

Пример кода, автоматически векторизуемого компилятором:

```
void test(float * restrict a, float * restrict b, float * restrict c, int n)
{
  for (int i = 0; i < n; i++)
    c[i] = a[i] * b[i] + a[i];
}
```

- **restrict**: мы говорим компилятору, что доступ в указанную память будет осуществляться только через указатель, использованный при объявлении. Не забываем ключ **–restrict**.
- Это необходимо, что компилятор зафиксировал факт отсутствия зависимостей по данным между массивами a, b и c и успешно векторизовал цикл.

Векторизация. Используем ключи и директивы компилятора

- -O2, -O3
- #pragma ivdep
- #pragma vector
  - always
  - aligned
  - nontemporal
- #pragma simd (с параметрами)
  - самый мощный на сегодняшний день инструмент;
  - необходимо пользоваться с осторожностью.

Векторизация. Используем ключи и директивы компилятора

```
#pragma ivdep
#pragma vector always
#pragma vector aligned
  for (int i = 0; i < n; i++)
    c[i] = a[i] * b[i] + a[i];
```

Или так:

```
  #pragma simd
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    c[i] = a[i] * b[i] + a[i];
```

Векторизация. Array Notation и Elemental Function в рамках технологии Intel Cilk Plus

- Технология Intel Cilk Plus позволяет разрабатывать эффективные параллельные программы для систем с общей памятью, по сравнению с OpenMP
  - упрощая обучение начинающих параллельному программированию,
  - предоставляя мощные, логичные и достаточно простые средства организации параллелизма с использованием механизма логических задач.
- Наряду с этим, в Cilk Plus добавлена так называемая Array Notation, что позволяет записывать вычисления в циклах как бы без самих циклов, явно показывая компилятору, что эти вычисления можно «положить» на векторную архитектуру.

Векторизация. Array Notation и Elemental Function в рамках технологии Intel Cilk Plus

```
void test(float * restrict a, float * restrict b,
          float * restrict c, int n)
{
    c[0:n] = a[0:n] * b[0:n] + a[0:n];
}
```

- Кроме того, в Intel Cilk Plus вводится специальный вид функций, **Elemental Function**. Эти функции, описанные специальным образом, могут выполнять операции над единицей данных. Такие функции могут быть использованы для векторизации и распараллеливания циклов.
- Подробнее Array Notation и Elemental Functions рассматриваются в лекции 5.

Векторизация. Интринсики и ассемблер

- **Большой контроль**: делаем ровно то, что хотим.
- **Сложность разработки**: нужно хорошо знать систему команд и особенности архитектуры, а также иметь опыт низкоуровневого программирования.
- **Трудности с переносимостью кода** на другие программно-аппаратные платформы.

Изучение выходит за рамки данного курса.

Векторизация
и математические функции

Введение

- Отдельно необходимо обсудить важный вопрос о сочетании векторизации и математических функций, вызываемых в циклах, так как именно на вычисление этих функций приходится основное время работы значительного числа прикладных программ.

- Ранее мы установили факт наличия в наборе команд Intel Xeon Phi специальных инструкций для вычисления четырех математических функций.

Как быть с остальными функциями?

Пример

```
void test(float * a, float * b,
          float * c, int n)
{
  #pragma simd
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    c[i] = a[i] * b[i] + sinf(a[i]);
}
```

Отчет: цикл векторизован.

Вопрос: как векторизуется синус?

Реализации математических функций

- **LibM** – модуль компилятора.
  ICC содержит быстрый LibM, оптимизированный под современные архитектуры.
- **SVML** (short vector math library) – модуль компилятора ICC.

  Используется, если цикл векторизован.

  Мат. функции реализованы с использованием SIMD, вычисляются для короткого вектора аргументов. Длина вектора соответствует длине xmm, ymm, zmm регистра.
- **VML** (vector math library) – часть библиотеки MKL.

  Используется при явном вызове функций (vsSin, vdSin…).

  Вычисляет значение функции в N точках.

Реализации математических функций

- **LibM** (см. math.h). Перекомпиляция ICC в программах, активно использующих мат. функции, часто приводит к ускорению расчетов.
- **SVML vs. VML**

  VML может выигрывать у SVML на больших длинах, но не всегда (эффект существенно зависит от архитектуры).

```
for (int i = 0; i < n; i++)
    c[i] = a[i] * b[i] + sinf(a[i]);
```

Не векторизован: **LibM**

Векторизован: **SVML**

VML: vsSin(n, a, c);

Реализации математических функций. Точность

Настройки, влияющие на точность (ICC):

- -fp-model
- -fimf-domain-exclusion
- -fimf-precision

Настройки, влияющие на точность (MKL/VML):

- Режимы High Accuracy (HA), Low Accuracy (LA), Enhanced Performance (EP).
- Могут быть настроены для конкретного вызова.

Авторский коллектив

- Мееров Иосиф Борисович,
  к.т.н., доцент, зам. зав. кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ

  meerov@vmk.unn.ru