



Оптимизируем для процессоров Intel®: Векторизация, на что способен компилятор?

Дмитрий Сергеев
Intel



Software & Services Group, Developer Products Division

Copyright © 2010, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

- **Введение**
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

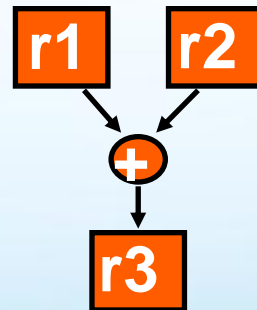
Введение

Векторные операции



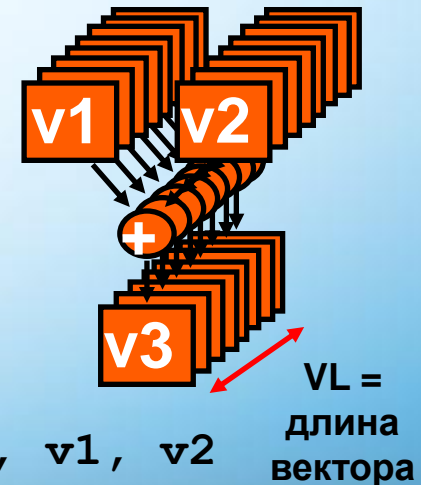
- Параллельность по данным
- Одна операция одновременно применяется к $N > 1$ элементам вектора – одномерного массива данных, состоящего из скалярных объектов - integers, floats, и т.д.
- Преимущество векторизации в том, что время выполнения векторной операции, такое же как и скалярной, но полезной работы выполняется больше.

Скалярная операция



`add.d r3, r1, r2`

Векторная операция

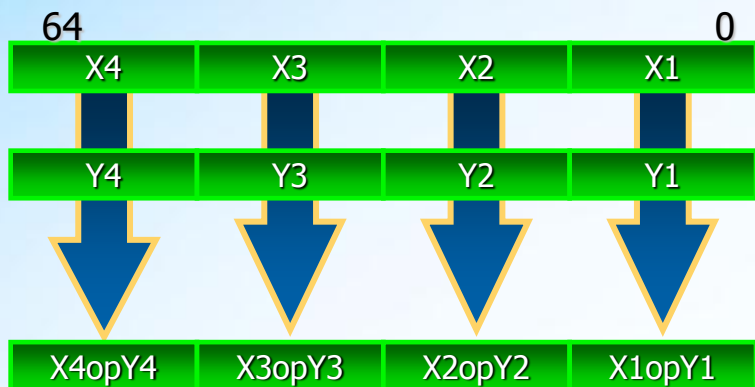


`addvec.d v3, v1, v2`

VL =
длина
вектора

- SIMD (Single Instruction Multiple Data) инструкции – это набор команд процессора для векторных операций, где:
 - Одна векторная операция – это одна инструкция процессора
 - Длина вектора фиксирована $VL=2,4,8,16$
 - Выполнение операции над всеми элементами вектора синхронно, т.е.
 - Все результаты доступны в одно и тоже время
- SIMD инструкции в процессорах Intel:
 - 64 bit Multi-Media Extension – MMX™
 - 128 bit Intel® Streaming SIMD Extension – Intel® SSE
 - 256 bit Intel® Advanced Vector Extensions – Intel® AVX
 - 512 bit vector instruction set extension of Intel® Many Integrated Core Architecture – Intel® MIC

SIMD: типы данных [1]

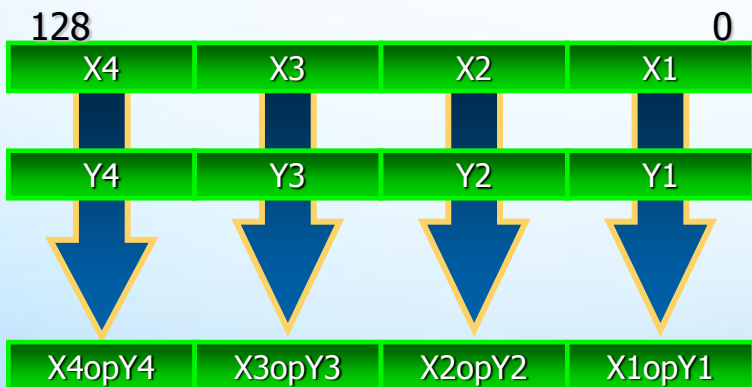


MMX™

Вектор: 64bit

Типы данных: 8, 16 and 32 bit integers

VL: 2,4,8



Intel® SSE

Вектор: 128bit

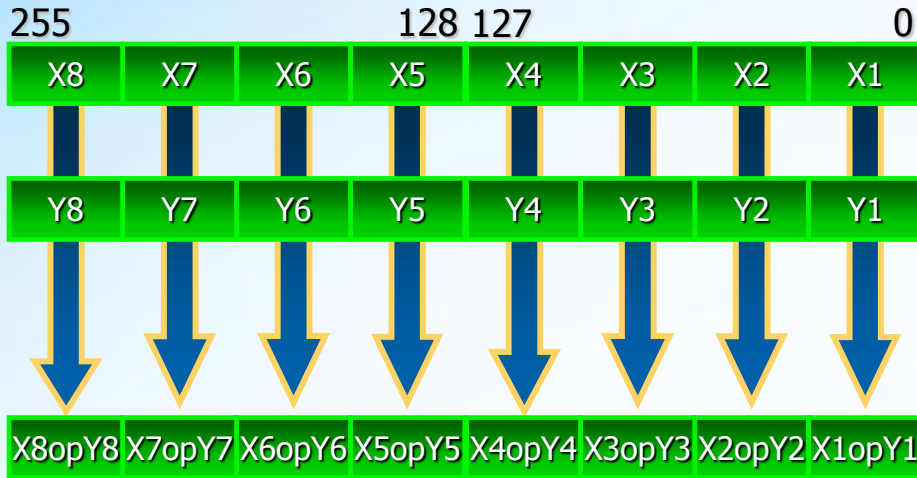
Типы данных:

8,16,32,64 bit integers

32 and 64bit floats

VL: 2,4,8,16

SIMD типы данных [2]



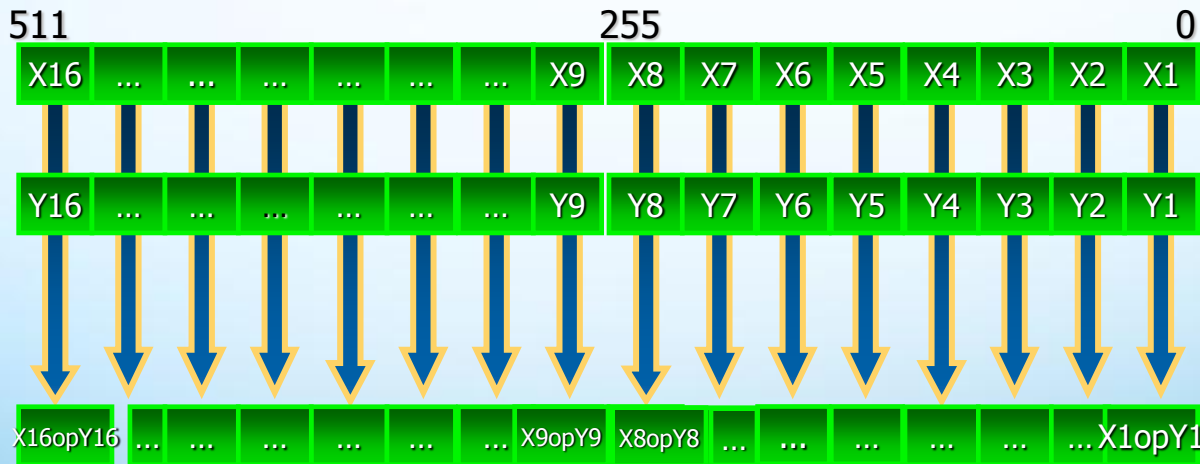
Intel® AVX

Вектор: 256bit

Типы данных: 32 and 64 bit floats

VL: 4, 8, 16

Нет арифметических операций над целыми числами – но есть логические: OR, XOR, AND ...



Intel® MIC

Вектор: 512bit

Типы данных:

32 and 64 bit integers

32 and 64bit floats

VL: 8,16



- Введение
- **Генерация векторного кода**
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

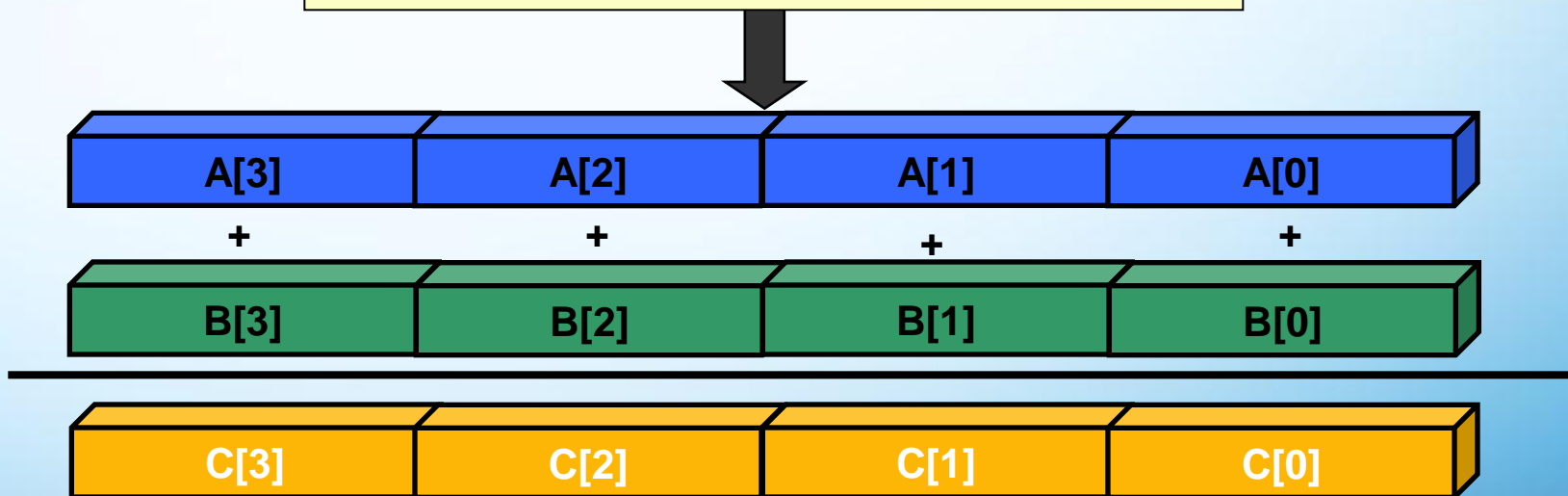
Что такое векторизация?



Использование векторных инструкций (SIMD, SSE) для обработки данных. Можно сделать:

- Вручную
- Автоматически (компилятор)

```
for (i=0; i<MAX; i++)  
  c[i]=a[i]+b[i];
```



Скалярные и векторные инструкции

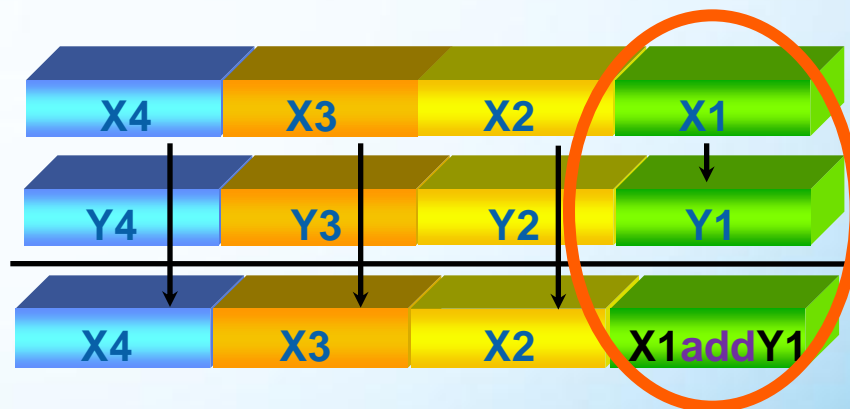


“Векторная” форма инструкции называется “упакованной” (packed) – а векторный код “упакованным кодом” 😊

- Большинство векторных инструкций имеют скалярную форму, оперирующую с 1 элементом вектора

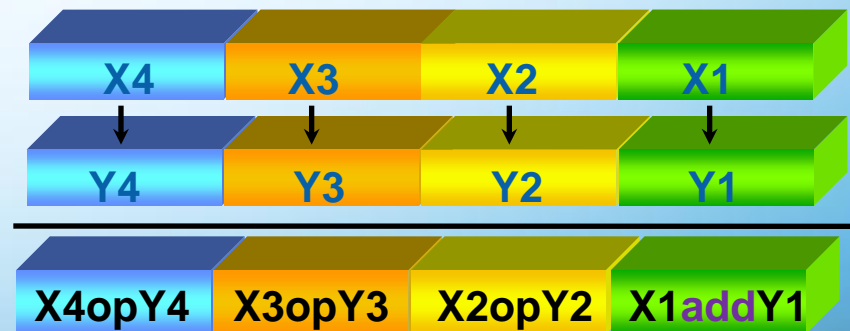
add_{ss} Scalar Single-FP Add

↑↑
single precision FP data
scalar execution mode



add_{ps} Packed Single-FP Add

↑↑
single precision FP data
packed execution mode

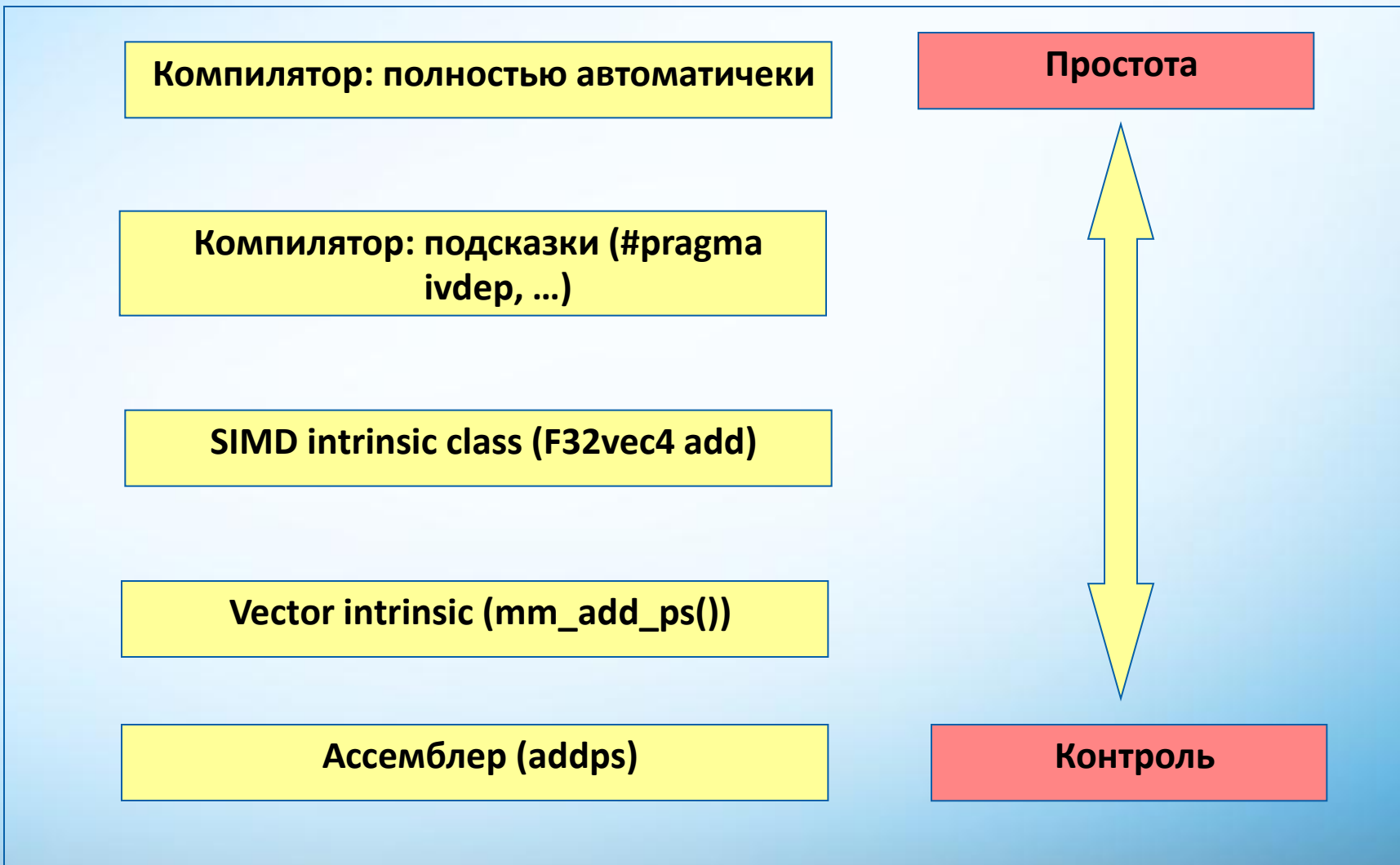


Вопросы:



1. Как векторизовать код?
 - Наша цель – **автоматическая векторизация** с помощью компилятора, хотя есть и другие способы ...
2. Что делать с различными наборами SIMD команд: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX ...?
 - Каждый набор инструкций включает в себя все предыдущие, поэтому желательно использовать последний доступный набор команд.
 - Если мы чётко знаем где программа будет выполняться – тут все просто, используем то что поддерживает ваш процессор
 - Если нет, используем опцию **выбора процессора в реальном времени**, которая использует соответствующую ветку кода в зависимости от процессора на котором выполняется

Как векторизовать код



Разные наборы команд – разный код!



```
static double A[1000], B[1000],
              C[1000];

void add() {
    int i;
    for (i=0; i<1000; i++)
        if (A[i]>0)
            A[i] += B[i];
        else
            A[i] += C[i];
}
```



```
.B1.2::
    movaps    xmm2, A[rdx*8]
    xorps     xmm0, xmm0
    cmpltpd   xmm0, xmm2
    movaps    xmm1, B[rdx*8]
    andps     xmm1, xmm0
    andnps    xmm0, C[rdx*8]
    orps      xmm1, xmm0
    addpd     xmm2, xmm1
    movaps    A[rdx*8], xmm2
    add       rdx, 2
    cmp       rdx, 1000
    jl        .B1.2
```

SSE2

```
.B1.2::
    vmovaps   ymm3, A[rdx*8]
    vmovaps   ymm1, C[rdx*8]
    vcmpgtpd  ymm2, ymm3, ymm0
    vblendvpd ymm4, ymm1, B[rdx*8], ymm2
    vaddpd    ymm5, ymm3, ymm4
    vmovaps   A[rdx*8], ymm5
    add       rdx, 4
    cmp       rdx, 1000
    jl        .B1.2
```

AVX

```
.B1.2::
    movaps    xmm2, A[rdx*8]
    xorps     xmm0, xmm0
    cmpltpd   xmm0, xmm2
    movaps    xmm1, C[rdx*8]
    blendvpd  xmm1, B[rdx*8], xmm0
    addpd     xmm2, xmm1
    movaps    A[rdx*8], xmm2
    add       rdx, 2
    cmp       rdx, 1000
    jl        .B1.2
```


SSE4.1

Software & Services Group, Developer Products Division



Copyright © 2011, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

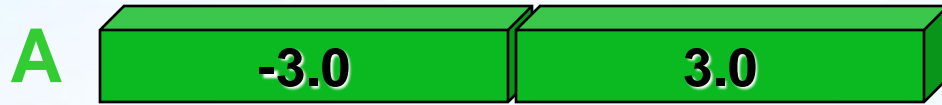
Optimization
Notice 

Пример использования Intrinsic

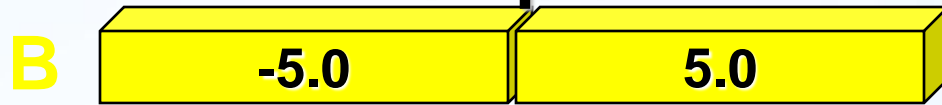
Условия без переходов



```
for (i=0,...) R[i] = (A[i]<B[i])? C[i]:D[i];
```



cmplt



and

nand



or



Result R



Пример использования Intrinsic



```
// R[i] = (A[i] < B[i])? C[i] : D[i]

__m128d mask = _mm_cmplt_pd(a, b);
r = _mm_or_pd(
    _mm_and_pd(mask, c),
    _mm_nand_pd(mask, d)
);
```

- Лучше использовать intrinsic или SIMD векторные классы чем прямое кодирование на ассемблере
 - Практически одинаковая производительность
 - Скрывают такие детали как прямое обращение к регистрам
 - Более портабельны – поддерживаются многими компиляторами!

Ручной выбор процессора



- В Intel® C++ Compiler есть API для явной реализации определенных функций под конкретный процессор
- Идентификация процессора по ключевому слову `cpuid`, например `core_i7_sse4_2` для Intel® Core™ i7
- Два расширения для определения функций:
 - Под определённый список архитектур:
`__declspec(cpu_dispatch(cpuid-list)) func(..)`
 - Под конкретную архитектуру:
`__declspec(cpu_specific(cpuid)) func(..)`
- Дополнительно – software.intel.com



- Введение
- Генерация векторного кода
- **Ключи компилятора для автоматической векторизации**
- Проверка успешности векторизации
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

Ключи компилятора для автоматической векторизации



Набор команд	Расширение
Intel® Streaming SIMD Extensions 2 (Intel® SSE2) для Pentium® 4 или совместимых не-Intel процессоров	SSE2
Intel® Streaming SIMD Extensions 3 (Intel® SSE3) для Pentium® 4 или совместимых не-Intel процессоров	SSE3
Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) для Intel® Core™ 2 Duo процессоров	SSSE3
Intel® SSE4.1 для Intel Core™ micro-architecture	SSE4.1
Intel® SSE4.2 Accelerated String and Text Processing instructions для Intel® Core™ i7 процессоров	SSE4.2
Расширения для Intel® ATOM™ процессоров : Intel® SSSE3 and MOVBE instruction	SSE3_ATOM
Intel® Advanced Vector Extensions (Intel® AVX) для 2-ого поколения Intel® Core™ i7 процессоров	AVX

Ключи векторизации [1]



{L&M} -x<extension> {W}: /Qx<extension>

- Только для Intel® процессоров (специальные оптимизации)
- Добавляется проверка процессора в программу (код не будет работать на не-Intel процессорах)

{L&M}: -m<extension> {W}: /arch:<extension>

- Нет проверки на Intel/не-Intel процессоры
- Нет специальных оптимизаций
- Код работает на всех процессорах поддерживающих данное расширение

{L&M}: -ax<extension> {W}: /Qax<extension>

- Двойной код - 'универсальный' и 'оптимизированный'
- Специальные оптимизации для Intel процессоров
- 'Универсальный' код, тоже самое что -msse2 (Windows: /arch:SSE2)
 - 'Универсальный' тип кода можно изменить, если специфицировать ключи -m или -x (/Qx or /arch)



Ключи векторизации [2]



По умолчанию **-msse2** (Windows: **/arch:SSE2**)

- Активируется при **-O2** или выше
- Необходим процессор поддерживающий Intel® SSE2
- Нужно использовать **-mia32** (Windows **/arch:IA32**) в случае если процессор не поддерживает SSE2 (Intel® Pentium™ 3)

Специальный ключ **-xHost** (Windows: **/QxHost**)

- Компилятор проверяет какие инструкции поддерживает процессор на котором происходит компиляция, и оптимизирует под него
- Нельзя использовать если необходим запуск приложения на разных платформах

Поддерживается комбинация ключей **-x<ext1>** и **-ax<ext2>** (Windows: **/Qx<ext1>** and **/Qax<ext2>**)

- В результате генерируется несколько версий кода
- Можно использовать **ext1 = ia32** в случае если 'универсальная' версия кода должна поддерживать старые процессоры, без SSE2 (Intel® Pentium™ 3)



Выключить векторизацию

- Глобально: {L&M}: **-no-vec** {W}: **/Qvec-**
- Для цикла: директива **novector**
 - Выключение векторизации означает запрет на “packed” инструкции SSE/AVX. Остальные инструкции входящие в расширение могут использоваться

Принудительная векторизация цикла:

#pragma vector always

- Цикл нужно векторизовать, несмотря на то что компилятор считает иначе (в случае не единичного шага доступа или не выровненных данных например)
- Цикл все равно не векторизуется, если компилятор не понимает его семантику
- Использование **#pragma vector always assert** даст ошибку в случае если цикл векторизовать не удалось

- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- **Проверка успешности векторизации**
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

Проверка успешности векторизации

- Исследование ассемблерного кода
 - Листинг: {L&M}: -S and {W}: /Fa
 - Проверить на наличие скалярных или векторных инструкций для определенного места кода (в листинге есть информация по строкам кода)
- Отчёт об оптимизации по фазе “High-Performance-Optimizer” (HPO)
 - {L&M}: `-opt-report<N>` `-opt-report-phase:hpo`
 - {W}: `/Qopt-report:<N>` `/Qopt-report-phase:hpo`
 - N=1,2,3 уровень детализации, N=2 по умолчанию
- Отчёт векторизатора
- Сбор профиля с помощью Intel® VTune Amplifier™, для подсчёта **векторных** SSE инструкций
 - Нужно собрать событие
`FP_COMP_OPS_EXE.SSE_FP_PACKED` для Intel® Core™ i7

Отчёт векторизатора



- Детальный отчет о векторизации:
 - L&M: `-vec-report<n>`, n=0,1,2,3,4,5
 - W: `/Qvec-report<n>`, n=0,1,2,3,4,5

```
35:      subroutine fd( y )
36:      integer :: i
37:      real, dimension(10), intent(inout) :: y
38:      do i=2,10
39:          y(i) = y(i-1) + 1
40:      end do
41:      end subroutine fd
```

```
novvec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novvec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novvec.f90(38:3-38:3):VEC:MAIN_: loop was not vectorized: existence of
vector dependence
```

Управление уровнем диагностики

L&M: -vec-report<N> W: /Qvec-report<N>



N	Сообщения
0	нет
1	Только векторизованные циклы (по умолчанию)
2	Векторизованные/не векторизованные циклы
3	Тоже что и N=2 + информация о предполагаемых и доказанных зависимостях по данным
4	Только не векторизованные
5	Тоже что и N=4 + объяснение почему

Примечание:

- Если используется (-ipro или /Qipro) нужно передавать этот ключ на этапе линковки



Software & Services Group, Developer Products Division

Copyright © 2011, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

Optimization
Notice 

- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- **Когда векторизация не работает**
 - **Зависимость по данным**
 - Выравнивание
 - Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

Когда векторизация не работает ...



- Наиболее часто: зависимость
 - Итерации цикла должны быть независимы
- Другие случаи
 - Выравнивание
 - Вызов функции в цикле
 - Сложный цикл / условные переходы
 - Цикл “не считаемый”
 - Т.е. верхняя граница цикла вычисляется в ран-тайм
 - Не внутренний цикл
 - Внешние циклы не векторизуются
 - Смешанные типы данных (многие случаи могут быть векторизованы)
 - Неединичный шаг элементов массива
 - Слишком сложный – не хватает регистров
 - Векторизация не эффективна
 - Другое ...



Зависимость по данным



Определение

- Зависимость по данным между строками S_1 и S_2 (пишется $S_1 \delta S_2$) **тогда и только тогда** :
 - Когда выполнение идёт от S_1 к S_2
 - S_1 и S_2 используют один и тот же блок памяти и либо S_1 , либо S_2 пишет в него данные
- Примечание: S_1 и S_2 могут быть одной и тоже строкой

Классификация зависимостей:

$S_1 \delta^F S_2$: S_1 пишет, S_2 читает : **"Flow"** зависимость

$S_1 \delta^A S_2$: S_1 читает, S_2 пишет : **"Anti"** зависимость

$S_1 \delta^O S_2$: S_1 читает, S_2 пишет: **"Output"** зависимость

$$\begin{array}{l} S_1 \quad \mathbf{x} = \dots \\ S_2 \quad \dots = \mathbf{x} \end{array}$$

$S_1 \delta^F S_2$

$$\begin{array}{l} S_1 \quad \dots = \mathbf{x} \\ S_2 \quad \mathbf{x} = \dots \end{array}$$

$S_1 \delta^A S_2$

$$\begin{array}{l} S_1 \quad \mathbf{x} = \dots \\ S_2 \quad \mathbf{x} = \dots \end{array}$$

$S_1 \delta^O S_2$

Зависимости в циклах



Зависимости в циклах наиболее интересны с точки зрения векторизации

- Чтобы понять есть ли зависимость в цикле – можно его “виртуально” развернуть:



```
DO I = 1, N
S1:   A(I+1) = A(I) + B(I)
ENDDO
```

```
S1 A(2) = A(1) + B(1)
S1 A(3) = A(2) + B(2)
S1 A(4) = A(3) + B(3)
S1 A(5) = A(4) + B(4)
...
```

В случае если для возникновения зависимости необходимо выполнить более чем 1 итерацию цикла – мы называем это “циклическая” зависимость. В противном случае – “цикло-независимая” зависимость

```
DO I = 1, 10000
S1   A(I) = B(I) * 17
S2   X(I+1) = X(I) + A(I)
ENDDO
```

($S_1 \delta^F S_2$) цикло-независимая

($S_2 \delta^F S_2$) циклическая



Зависимости и векторизация



Теорема: Цикл может быть векторизован тогда и только тогда, если в нем не существует циклической зависимости между операциями.

- Доказательство – ссылка [3]
- Теорема предполагает “бесконечную” длину вектора (VL). В случае если VL константа – 2,4,8, ... что справедливо для реальных случаев (SSE/AVX), циклические зависимости, для возникновения которых требуется VL+1 или более итераций, могут быть проигнорированы.
- Таким образом, в определённых случаях, векторизация для **SSE/AVX** может быть осуществима, тогда как теорема говорит – нет.

Пример:

```
DO I=1,N
  A(I) = A(I+3) + C
END DO
```

Есть циклическая зависимость, но цикл может быть векторизован для SSE, в случае если тип данных **double** precision float **но не** для **single** precision float

Что делать с зависимостями #1



Подсказки компилятору

- Многие зависимости только предполагаются компилятором, но не существуют на самом деле, например пересечение указателей
 - Компилятор консервативен и всегда предполагает самый плохой случай

```
// Sample: Without additional information (like inter-procedural
// knowledge) compiler has to assume 'a' and 'b' to alias
void scale(int *a, int *b)
{
    for (int i=0; i<10000; i++) b[i] = z*a[i];
}
```



Подсказки

Ключевое слово "restrict" для указателей



{L&M}: -restrict

{W}: /Qrestrict

{L&M}: -std=c99

{W}: /Qstd=c99

- Указывает, что только сам указатель или значение основанное на указателе, такое как (pointer+1) – будет использовано для доступа к объекту
- Только для C, не для C++

```
void scale(int *a, int * restrict b)
{
    for (int i=0; i<10000; i++) b[i] = z*a[i];
}

// two-dimension example:
void mult(int a[][NUM],int b[restrict][NUM]);
```

Подсказки [C/C++]

Некоторые директивы и ключи



#pragma ivdep

- “Ignore Vector Dependencies” – компилятор будет игнорировать предполагаемые (но не доказанные) зависимости в цикле следующем за директивой
- В случае использования с ключом `-ivdep-parallel (/Qivdep-parallel)`, игнорируются только циклические зависимости

Нет алиасинга в программе

- {L&M}: `-fno-alias` {W}: `/Oa`

Алиасинг в программе соответствует правилам ISO C

- {L&M}: `-ansi-alias` {W}: `/Qansi-alias`
- A pointer can be de-referenced only to an object of the same type or compatible type

Нет алиасинга для аргументов функций

- {L&M}: `-fargument-noalias` {W}: `/Qalias-args-`
- Для каждой отдельно взятой функции, аргументы этой функции не ссылаются на общий объект

- **Динамический анализ зависимостей**

- Компилятор может (!) в реальном времени определять пересекаются ли массивы данных
- В зависимости от результата выполняется скалярная или векторная версия цикла ("Loop Versioning")
- Эвристика компилятора настроена на баланс между оверхедом от проверки и получаемом выигрыше в производительности
 - Например для присвоения

$$A[...] = B_1[...] + B_2[...] + \dots + B_N[...]$$

проверка делается для N=2 не для N=5

- Ключ `-opt-multi-version-aggressive` (/Qopt-multi-version-aggressive для Windows) позволяет управлять этой эвристикой
- **Межпроцедурный анализ зависимостей**
 - Активируется как "inter-procedural optimization": `-ipo` (/Qipo для Windows)
 - Для `-O2` и `-O3`, IPO внутри файла включено по умолчанию
 - Позволяет компилятору видеть определения и создание аргументов функций во всей программе, векторизация делается после анализа всех файлов программы

- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- **Когда векторизация не работает**
 - Зависимость по данным
 - **Выравнивание**
 - Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

Выравнивание



- Для векторных SSE инструкций необходимо чтобы данные были выравнены по 16 байт
- Для векторных AVX – по 32 байта
- Не выровненные данные загружаются с помощью команд не выровненной загрузки, которые работают медленнее.
- Компилятор может делать 'versioning' в случае если выравнивание не понятно на этапе компиляции



Подсказки для выравнивания [C/C++]



- Выравненная аллокация

```
void* _mm_malloc (int size, int base)
```

Linux & Mac OS X:

```
int posix_memaligned(void **p, size_t base, size_t size)
```

- Директивы

```
#pragma vector aligned | unaligned
```

- Атрибуты для переменных

```
{W,L,M} : __declspec(align(base)) <array_decl>
```

```
{L&M} : <array_decl> __attribute__((aligned(base)))
```

- Расширения для C/C++

```
__assume_aligned(<variable>, base)
```



- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- **Когда векторизация не работает**
 - Зависимость по данным
 - Выравнивание
 - **Другое: неединичный шаг доступа к данным, вызовы функций и т.д. ...**
- Векторизация специальных конструкций
 - Определение идиом
 - Сложные типы данных
- HLO преобразования циклов
- Резюме, ссылки

Неподдерживаемая структура цикла



- “Unsupported loop structure” чаще всего означает что компилятор просто не может вычислить количество итераций в цикле
 - Например while цикл, где количество итераций определяется в процессе выполнения
 - Верхняя/нижняя граница for цикла не является циклически-независимой
- В некоторых случаях это можно легко исправить:

```
struct _x { int d; int bound;};

doit1(int *a, struct _x *x)
{
    for (int i=0; I < x->bound; i++)
        a[i] = 0;
}
```

```
struct _x { int d; int bound;};

doit1(int *a, struct _x *x)
{
    int local_ub = x->bound;
    for (int i=0; I < local_ub; i++)
        a[i] = 0;
}
```

Непоследовательный (Non-Unit) доступ



Non-unit stride: Доступ в память в цикле происходит
непоследовательно

- Векторизация в некоторых случаях возможна (в случае если доступ детерминированный/линейный), но “дорогие”
непоследовательные операции с памятью могут перекрыть все
плюсы от векторизации
 - Отчёт векторизатора: “Loop was not vectorized:
vectorization possible but seems inefficient”


Пример:

```
for (I=0;I<=MAX;I++)
  for (J=0;J<=MAX;J++)
  {
    D[I][J]+=1;           // Unit Stride
    D[J][I]+=1;           // Non-Unit but linear
    A[J*J]+=1;            // Non-unit
    A[B[J]]+=1;           // Non-Unit
    if (A[MAX-J]==1) last=J; // Non-Unit
  }
```

Software & Services Group, Developer Products Division

Copyright © 2011, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

Optimization
Notice 



Как избежать?

- “Вывернуть” цикл, (loop interchange), в случае если доступ линейный
- Часто компилятор может сделать это автоматически, например цикл умножения матриц:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        for (k=0; k<N; k++)  
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Но не всегда, в случае ниже, нужно делать это руками:

```
// Non-unit access  
for (j = 0; j < N; j++)  
    for (i = 0; i <= j; i++)  
        c[i][j] = a[i][j]+b[i][j];
```

```
// Unit access  
for (i = 0; i < N; i++)  
    for (j = i; i <= N; j++)  
        c[i][j] = a[i][j]+b[i][j];
```


Вызовы функций / In-lining

- Вообще, вызов функции в цикле не даёт компилятору его векторизовать
 - Исключение #1 : “intrinsic” функции, типа математических
 - Исключение #2 : Если функция инлайнится

Intel Compiler:
15 times
faster by
using `-ipo`
(/Qipo on
Windows) !*


```
for (i=1;i<nx;i++) {  
    x = x0 + i*h;  
    sumx = sumx + func(x,y, xp, yp) ;  
}  
  
float func(float x, float y, float xp, float yp)  
{  
    float denom;  
    denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) ;  
    denom = 1./sqrt(denom) ;  
    return denom;  
}
```

*: Intel® C++ Compiler 12.0 U1 for Linux, Redhat Enterprise Linux 64bit 6.0, Intel XEON® X5560 processor, 2.8GHz

Software & Services Group, Developer Products Division

Copyright © 2011, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

Optimization
Notice 



Большинство математических функций “векторизуется” путем вызова их векторных версий из “Short Vector Math Library” – libsvml

Функции для которых
есть векторные версии
libsvml
(Intel® Composer XE
2011)

acos	ceil	fabs	round
acosh	cos	floor	sin
asin	cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	erfc	log	tan
atan2	erfinv	log10	tanh
atanh	exp	log2	trunc
cbrt	exp2	pow	



Условные зависимости



- Сложные условия в цикле не дают компилятору его векторизовать
- Но тем не менее, “определённые условия” можно векторизовать с помощью бит-масок
 - Идея следующая:

```
for (i=1; i<=U; i++)  
    if ( R1[i] > R2[i] )  
        L1[i] = R1[i];  
    else  
        L2[i] = R2[i];
```

```
MASK[1:U] = (R1[1:U] > R2[1:U]);  
L1[1:U] = (MASK[1:U] & R1[1:U]) |  
          (!MASK[1:U] & L1[1:U]);  
L2[1:U] = (MASK[1:U] & L2[1:U]) |  
          (!MASK[1:U] & R2[1:U]);
```

- Т.е. сначала вычисляем маску – а потом по ней делаем присвоение
- В SSE инструкциях есть команды (blend) для эффективной реализации таких алгоритмов
 - Пример на слайде 12



- Такое преобразование, приводит к тому что обе ветки условия вычисляются для каждой итерации. Компилятор делает это только в том случае, если он уверен что не возникнет никаких ошибок, которые были скрыты в оригинальном коде условием
 - Отчёт векторизатора: "... condition may protect exception"
 - В нашем случае все ок, так как выражение справа и так вычисляется на каждой итерации
 - В некоторых случаях компилятор может добавить проверки на корректность
 - Директивы типа "#pragma vector always" говорят компилятору что такая трансформация корректна в любом случае



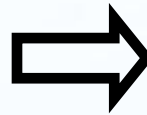
- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: непоследовательный доступ к данным, вызовы функций и т.д. ...
- **Векторизация специальных конструкций**
 - **Определение идиом**
 - **Комплексный тип данных**
- HLO преобразования циклов
- Резюме, ссылки

Определение идиом

Компилятор определяет программные конструкции (идиомы) и конвертирует их в более эффективные конструкции

Пример:

```
unsigned char a[N], b[N];
void swap32(int n)
{
    int i;
    for (i = 0; i < n; i+=4)
    {
        a[i+0] = b[i+3];
        a[i+1] = b[i+2];
        a[i+2] = b[i+1];
        a[i+3] = b[i+0];
    }
}
```



```
L:  movdqa    xmm1, b[ecx*4-4]
    pshufb   xmm1, xmm0
    movdqa   a[ecx*4-4], xmm1
    add     ecx, 4
    cmp     ecx, eax
    jle    L
```

Перестановка байт определена тут как идиома.
PSHUFB входит в состав SSSE-3

Idiom Recognition – Saturation




- To enable idiom recognition, the source code needs to express exactly the conditions required to use the corresponding SSE instruction
- In the sample below, the compiler will use PADDSSB (“Add packed signed bytes with saturation”) because the source code limits both, the upper and lower bound, of the add operation
 - Frequently the lower bound check is missing which would be ok here only for unsigned char !

```
define N 1000
void sat_signed_char(char va[N], char vb[N], char vc[N])
{
    int i;
    for (i = 0; i < N; i++)
        vc[i] = ( (vb[i] + va[i] > 127) ? 127 :
                  ( (vb[i] + va[i] < -128) ? -128 :
                    vb[i] + va[i] ) );
}
```

Software & Services Group, Developer Products Division

Copyright © 2011, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

Optimization
Notice 



Векторизация комплексной арифметики



- C (C99) и Fortran поддерживают тип данных COMPLEX
- Компилятор может векторизовать такие конструкции с использованием SSE3 инструкций
- Лучше не использовать “самописные” классы/структуры для работы с комплексными числами

Пример:

```
float _Complex zc[10];
```

```
float _Complex za=4 + __I__*2;  
    //Real Part           = 4  
    //Imaginary part     = 2
```

```
void zscale() {  
    for (int i=0; i<10; i++)  
        zc[i] = za*csin(zc[i]);  
}
```

```
//Compile (W): icl complex.c /Qstd=c99 /QxSSE3
```

```
//Compile (L & M): icl complex.c -std=c99 -xSSE3
```



- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: непоследовательный доступ к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Комплексный тип данных
- **NLO преобразования циклов**
- Резюме, ссылки

Трансформация циклов

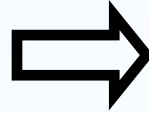
- Часто векторизация (эффективная) возможна только после определённых преобразований циклов
- В компиляторе за это отвечает HLO – High Level Optimization
 - HLO работает для O2 и O3, но только на O3 используется полный набор трансформаций
- Отчёт HLO:
 - {L&M}: `-opt-report -opt-report-phase:hlo`
 - {W}: `/Qopt-report /Qopt-report-phase:hlo`

```
...  
LOOP INTERCHANGE in loops at line: 7 8 9  
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )  
  
...  
Loop at line 7 unrolled and jammed by 4  
Loop at line 8 unrolled and jammed by 4  
  
...
```

Преобразования цикла



```
14: for (i=0; i<100; i++)
15: {
16:     a[i] = 0;
17:     for (j=0; j<100; j++)
18:         a[i] += b[j][i];
19: }
```



```
a[0:99] = 0;
for (j=0; j<100; j++)
    a[0:99] += b[j][0:99];
```

Отчёт векторизатора:

file.c(16) : (col. 8) remark: PARTIAL LOOP WAS VECTORIZED.

file.c(14) : (col. 8) remark: loop was not vectorized: not inner loop.

file.c(18) : (col. 10) remark: PERMUTED LOOP WAS VECTORIZED.

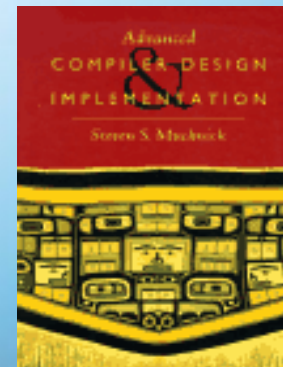
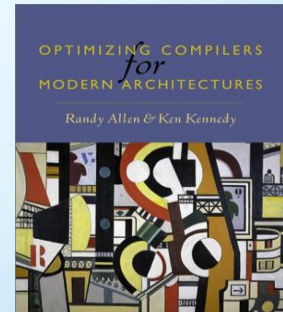
Преобразования сделанные компилятором:

- 1) i-цикл **distributed** (разбит) на 2 цикла: отдельный цикл и вложенный цикл
- 2) Вложенный цикл **interchanged** (“вывернут”) для последовательного доступа к элементам b[j][i]
- 3) Первый цикл **векторизован**. (1-е VECTORIZED)
- 4) “Вывернутый” цикл **векторизован** (2-е VECTORIZED)

- Введение
- Генерация векторного кода
- Ключи компилятора для автоматической векторизации
- Проверка успешности векторизации
- Когда векторизация не работает
 - Зависимость по данным
 - Выравнивание
 - Другое: непоследовательный доступ к данным, вызовы функций и т.д. ...
- Векторизация специальных конструкций
 - Определение идиом
 - Комплексный тип данных
- HLO преобразования циклов
- **Резюме, ссылки**

- Компилятор Intel® C++ and Intel® Fortran поддерживают генерацию векторного кода для всех процессоров Intel®
- Даже для случая “автоматической” векторизации, можно улучшить производительность программы, если подсказать компилятору что делать
 - Компилятор даёт полный отчёт по генерации кода
 - Директивы и ключи позволяют полностью контролировать процесс генерации
- Понимание что такое зависимости, выравнивание – необходимые знания для эффективного использования расширений SSE/AVX и для получения наилучшей производительности вашего кода

- [1] Aart Bik: "The Software Vectorization Handbook"
 - http://www.intel.com/intelpress/sum_vmmx.htm
- [2] Intel® 64 and IA-32 Architectures Software Developer's Manuals
 - <http://www.intel.com/products/processor/manuals/index.htm>
- [3] Randy Allen, Ken Kennedy: "Optimizing Compilers for Modern Architectures: A Dependence-based Approach"
- [4] Intel Software Forums, Knowledge Base, White Papers, Tools Support etc
 - <http://software.intel.com>
- [5] Steven S. Muchnik, "Advanced Compiler Design and Implementation"



Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Legal Disclaimer



INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2010. Intel Corporation.


<http://intel.com/software/products>



Software & Services Group, Developer Products Division

Copyright © 2011, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

Optimization
Notice 

Student Exercise # 1

Which Loops will Vectorize ?



```
#01: for (j=1; j<MAX; j++)  a[j]=a[j-n]+b[j];
```

```
#02: for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];
```

```
#03: for (int j=0; j<SIZE; j++)  
      for (int i=0; i<SIZE; i++)  
          b[i] += a[i][j] * x[j];
```

```
#04: for (int i=0; i<SIZE; i++)  
      b[i] += a[i] * x[index[i]];
```

```
#05: for (j=1; j<MAX; j++)  sum = sum + a[j]*b[j]
```

```
#06: for (int i=0; i<length; i++)  
      if ( s >= 0 )  
          x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
```

Student Exercise # 2

Find (if any) all Dependencies in these Samples



Dependencies? Type ?

```
for (i=0;i<MAX-2,i++)  
S:   A[i+2]=A[i] + 1;
```

```
for (i=1;i<MAX,i++)  
{  
S1:  A[i]=A[i-1] * 2;  
S2:  B = A[i-1];  
}
```

```
for (i=0;i<MAX,i++)  
S:  A[i+1,j] = A[i,k] + B;
```

```
for (i=0;i<MAX-1,i+=2)  
S:  A[i+1]=A[i] + 1;
```