Векторизация, зачем она нужна, реализация в процессорах и компиляторах Intel

Воробцов Игорь, Intel

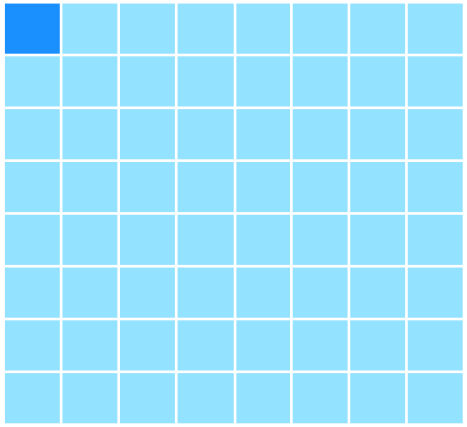
Содержание

- **Введение**
- Операция «Векторизация»
  - Генерация векторного кода
  - Ключи компилятора
  - Проверка успешности векторизации
  - Когда векторизация не работает
    - Зависимость по данным
    - Выравнивание
    - Другое: непоследовательный доступ к данным, вызовы функций и т.д. …
  - HLO преобразования циклов
- Резюме, ссылки
- Q&A

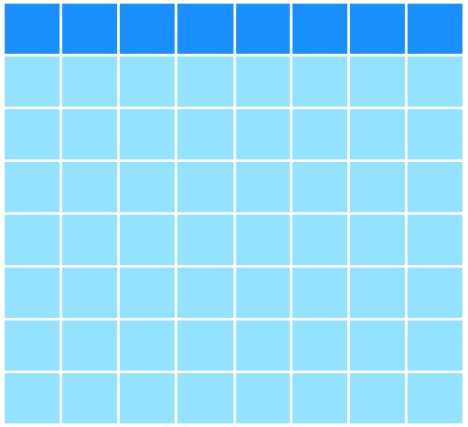
Максимум производительности

Где он?

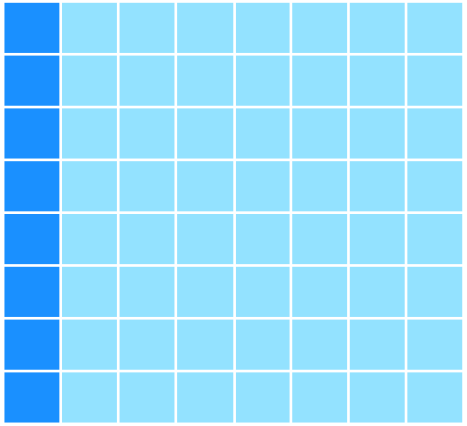
Скалярный код



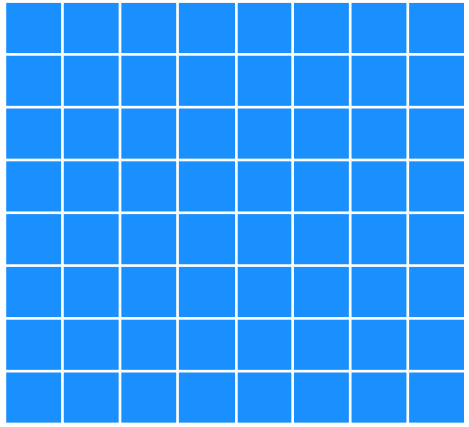
Векторный код



Параллельный код



Вместе - лучше



Максимум производительности
Как достичь?

- Гонка мегагерц
  Производительность «даром»

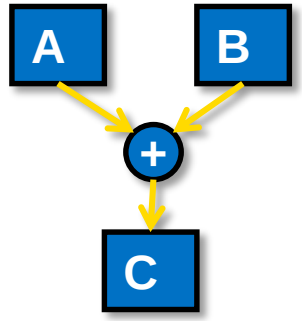
- Ширина команды SIMD **по данным (DLP)**
  - Перекомпиляция
  - Возможно, локальная переработка кода
  - Изредка, перепроектирование
  - Перекомпиляция для каждого нового поколения
- Многоядерность **по задачам (TLP)**
  - Перепроектирование
  - Возможно, глобальная переработка кода
  - Перекомпиляция
  - В дальнейшем «бесплатная» масштабируемость

Векторизация

Single **I**nstruction **M**ultiple **D**ata (SIMD):

- работа с вектором за одну операцию
- параллелизм по данным (Data Level Parallelism)
- более эффективная работа

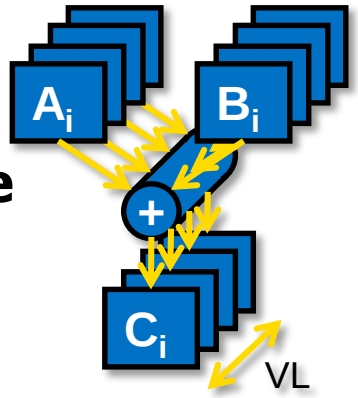
Скалярное выполнение



Вектор:

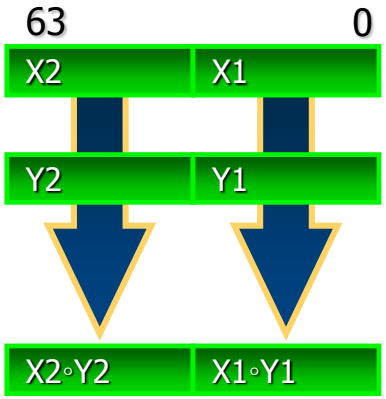
- содержит более одного элемента
- тип элемента совпадает с типами скаляров (например: float, integer, …)

Векторное выполнение



Длина вектора (Vector length - VL): число элементов в векторе

Векторизация реализуется через использование SIMD инструкций & «железа»



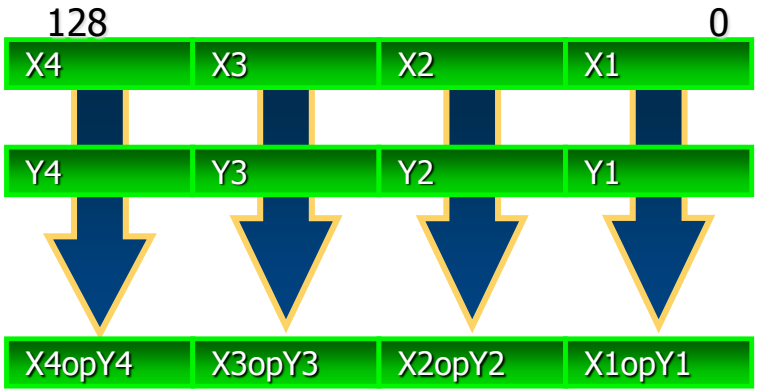
Intel® MMX™

Размер вектора : 64 bit

Типы данных:

8,16,32 битные целые

Длина вектора: 2, 4, 8



Intel® SSE

Размер вектора: 128 бит

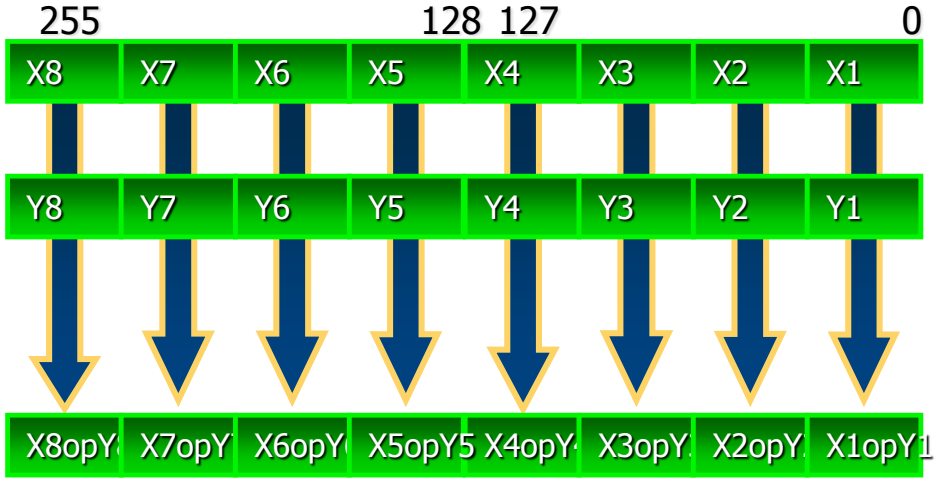
Типы данных:

8,16,32,64 битные целые

32 и 64 битные числа с плавающей точкой

Длина вектора: 2,4,8,16

Векторизация реализуется через использование SIMD инструкций & «железа»



Intel® AVX

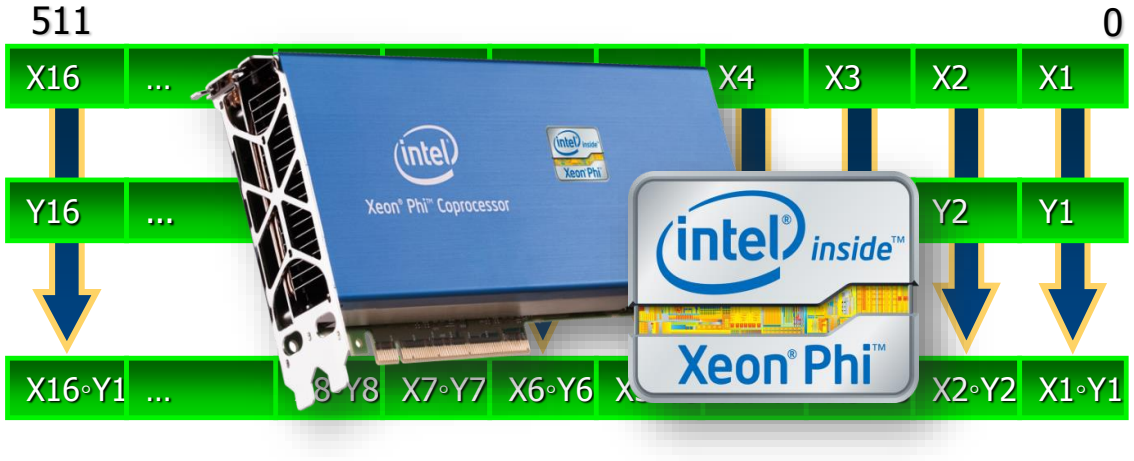
Размер вектора : 256bit

Типы данных:

- 32 и 64 битные числа с плавающей точкой

Длина вектора: 4, 8, 16

Впервые появилась в 2011



Intel® MIC

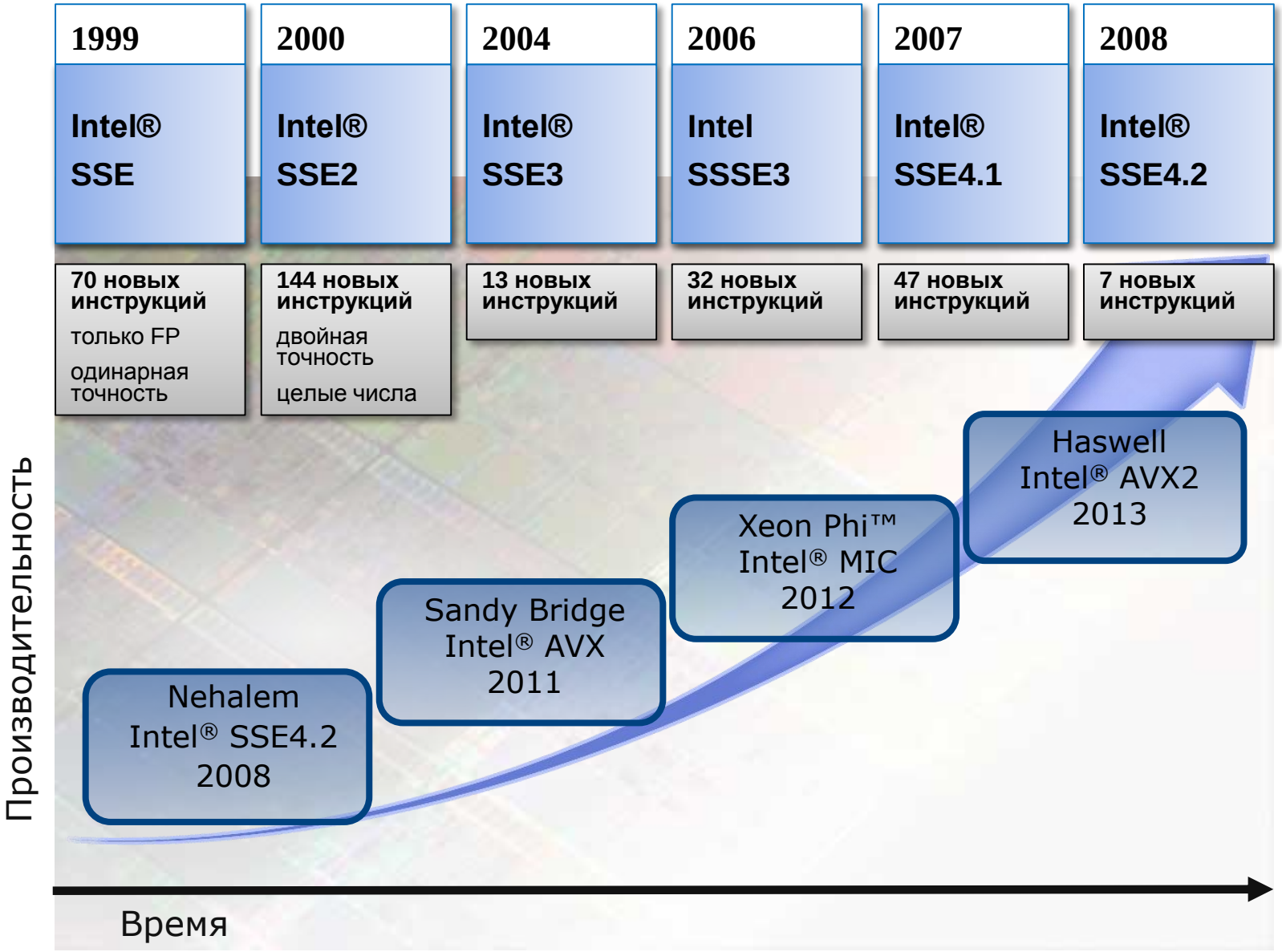
Размер вектора : 512bit

Типы данных:

- 32 битные целые
- 32 и 64 битные числа с плавающей точкой

Длина вектора: 8, 16

Поколения SSE



Содержание

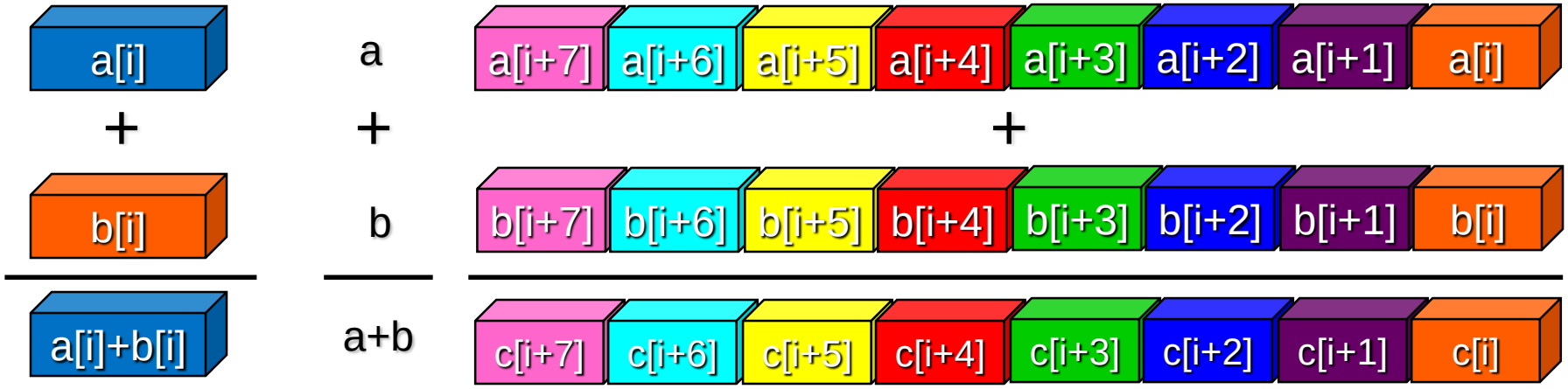
- Введение
- Операция «Векторизация»
  - **Генерация векторного кода**
  - Ключи компилятора
  - Проверка успешности векторизации
  - Когда векторизация не работает
    - Зависимость по данным
    - Выравнивание
    - Другое: непоследовательный доступ к данным, вызовы функций и т.д. …
  - HLO преобразования циклов
- Резюме, ссылки
- Q&A

Векторизация

SIMD – одна инструкция для вектора данных

- Скалярное выполнение
  - x86/x87/SSE
  - одна инструкция дает один результат
- Векторное выполнение
  - SSE или AVX инструкции
  - одна инструкция дает вектор результатов

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



Различные способы векторизации

Компилятор:
Авто-векторизация (без изменения кода)

Компилятор :
Директивы для векторизации

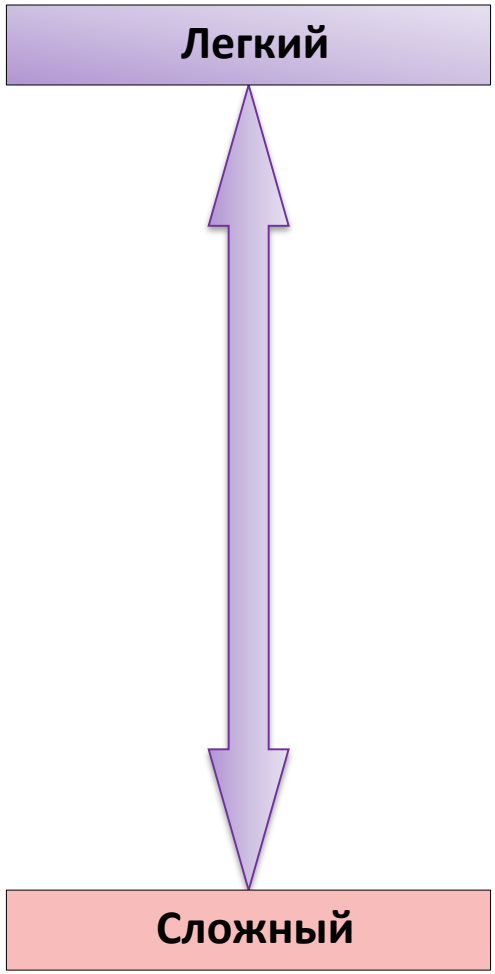
Компилятор:
#pragma simd / omp simd

Компилятор :
Intel® Cilk™ Plus Array Notation

Классы SIMD intrinsic
(например: `F32vec`, `F64vec`, …)

Векторные intrinsic функции
(например : `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

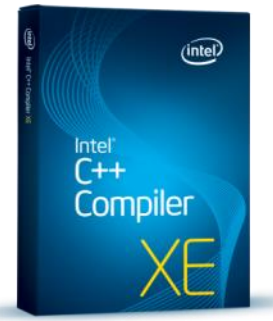
Чистый ассемблер
(например : `[v]addps`, `[v]addss`, …)



habrahabr.ru

“Разработчик на распутье: как векторизовать?! ”

Автоматическая векторизация




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

```
subroutine add(A, B, C)
  real*8 A(1000), B(1000), C(1000)
  do i = 1, 1000
      C(i) = A(i) + B(i)
  end do
end
```

Intel® AVX

```
..B1.2:
  vmovupd   (%rsp,%rax,8),  %ymm0
  vmov
  vmo
  vmo
  vac
  vac
  vac
  vac
  vmo
  vmo
  vmo
  vmo
  add
  cmpq
  jb        ..B1.2
...
```

Intel® SSE4.2

```
..B1.2:
  movaps    (%rsp,%rax,8),  %xmm0
  jb        ..B1.2
...
```

Компилятор Intel сам векторизует циклы

- по умолчанию
  - с уровнем оптимизации O2
- можно контролировать набор инструкций
  - -x<…>, -m<…>, -ax<…> (-xAVX)
- все ли циклы векторизованы?

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

Содержание

- Введение
- Операция «Векторизация»
  - Генерация векторного кода
  - **Ключи компилятора**
  - Проверка успешности векторизации
  - Когда векторизация не работает
    - Зависимость по данным
    - Выравнивание
    - Другое: непоследовательный доступ к данным, вызовы функций и т.д. …
  - HLO преобразования циклов
- Резюме, ссылки
- Q&A

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™ 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™ процессоров	AVX
Intel® Advanced Vector Extensions (Intel® AVX) для 3-го поколения Intel® Core™ процессоров	CORE-AVX-I
Intel® Advanced Vector Extensions 2 (Intel® AVX2)	CORE-AVX2

Ключи векторизации

{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

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)

Содержание

- Введение
- Операция «Векторизация»
  - Генерация векторного кода
  - Ключи компилятора
  - **Проверка успешности векторизации**
  - Когда векторизация не работает
    - Зависимость по данным
    - Выравнивание
    - Другое: непоследовательный доступ к данным, вызовы функций и т.д. …
  - HLO преобразования циклов
- Резюме, ссылки
- Q&A

Как понять что код векторизован?

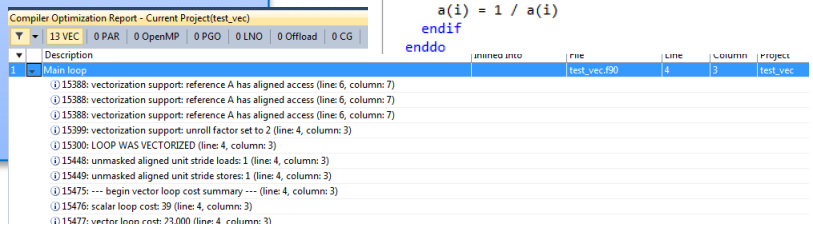
- Единая опция для всех отчетов **-opt-report**

(объединение -vec-report, -par-report, -openmp-report и др.)

- Файл с отчётом для каждого объектника.

```
int size();
 void foo(double *restrict
a, double *b){
  int i;
  for (i=0;i<size();i++){
    a[i] += b[i];
  }
}
```

```
program test
  real a(1000)
  1 optimization note
  Main loop
    15388: vectorization support: reference A has aligned access (line: 6, column: 7)
    15388: vectorization support: reference A has aligned access (line: 6, column: 7)
    15388: vectorization support: reference A has aligned access (line: 6, column: 7)
    15399: vectorization support: unroll factor set to 2 (line: 4, column: 3)
    15300: LOOP WAS VECTORIZED (line: 4, column: 3)
    15448: unmasked aligned unit stride loads: 1 (line: 4, column: 3)
    15449: unmasked aligned unit stride stores: 1 (line: 4, column: 3)
    15475: --- begin vector loop cost summary --- (line: 4, column: 3)
    15476: scalar loop cost: 39 (line: 4, column: 3)
    15477: vector loop cost: 23.000 (line: 4, column: 3)
  do i=1,1000
    if (a(i) .gt. 0) then
      a(i) = 1 / a(i)
    endif
  enddo
```



icpc –c -O3 –restrict –opt-report x.cpp

15.0 compiler:

LOOP BEGIN at x.cpp(6,15)

remark #15523: loop was not vectorized: cannot compute loop iteration count before executing the loop.

LOOP END

Содержание

- Введение
- Операция «Векторизация»
  - Генерация векторного кода
  - Ключи компилятора
  - Проверка успешности векторизации
  - **Когда векторизация не работает**
    - Зависимость по данным
    - Выравнивание
    - Другое: непоследовательный доступ к данным, вызовы функций и т.д. …
  - HLO преобразования циклов
- Резюме, ссылки
- Q&A

Когда векторизация не работает …

Наиболее часто: зависимость

- Итерации цикла должны быть независимы

Другие случаи

- Выравнивание данных
- Вызов функции в цикле
- Сложный цикл / условные переходы
- Цикл “не считаемый”
  - верхняя граница цикла вычисляется в ран-тайме
- Смешанные типы данных (многие случае могут быть векторизованы)
- Неединичный шаг элементов массива
- Векторизация не эффективна
- Другое …

Зависимость по данным

Определение

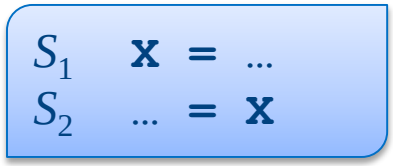
- Зависимость по данным между строками S_1 и S_2 (пишется $S_1 \ \delta \ S_2$) **тогда и только тогда :**
  - Когда выполнение идёт от S_1 к S_2
  - S1 и S2 используют один и тот же блок памяти и либо S_1 , либо S_2 пишет в него данные
- Примечание: S_1 и S_2 могут быть одной и той же строкой

Классификация зависимостей:

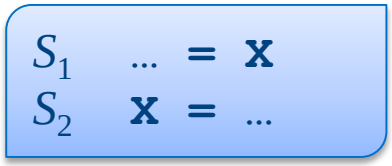
$S_1 \ \delta^F \ S_2$: S_1 пишет, S_2 читает : **“Flow”** зависимость

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

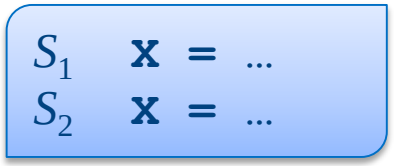
$S_1 \ \delta^O \ S_2$: S_1 пишет, S_2 пишет: **“Output”** зависимость



$S_1 \ \delta^F \ S_2$



$S_1 \ \delta^A \ S_2$

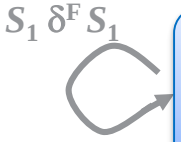


$S_1 \ \delta^O \ S_2$

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]);
```

Различные способы векторизации

Компилятор:
Авто-векторизация (без изменения кода)

Компилятор :
Директивы для векторизации

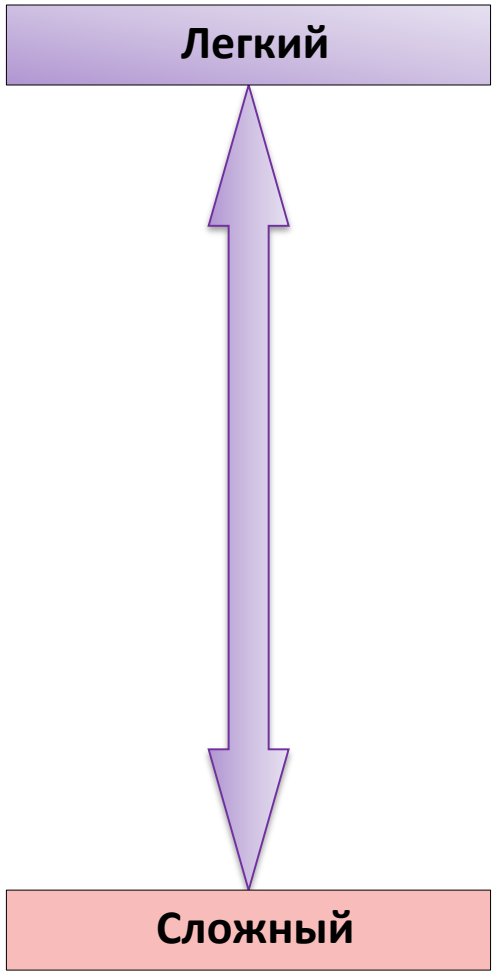
Компилятор:
#pragma simd / omp simd

Компилятор :
Intel® Cilk™ Plus Array Notation

Классы SIMD intrinsic
(например: `F32vec`, `F64vec`, …)

Векторные intrinsic функции
(например : `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Чистый ассемблер
(например : `[v]addps`, `[v]addss`, …)



habrahabr.ru

“Разработчик на распутье: как векторизовать?! ”

Подсказки [C/C++]

Некоторые директивы и ключи

#pragma ivdep

- “**I**gnore **V**ector **Dep**endencies” – компилятор будет игнорировать предполагаемые (но не доказанные) зависимости в цикле следующем за директивой

Нет алиасинга в программе

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

Алиасинг в программе соответсвует правилам ISO C

- {L&M}: -ansi-alias {W}: /Qansi-alias
- Указатель может быть разыменован только на объект того же типа

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

Что делать с зависимостями #2

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

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

    проверка делается для N=2, но 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

Выравнивание данных

- 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

“Что такое выравнивание, и как оно влияет на работу ваших программ”

Выравнивание данных [C/C++]

Выделение памяти:

```
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)`

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 Stride) доступ

Non-unit stride: Доступ к памяти 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
  }
```

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

```
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;
}
```

Различные способы векторизации

Компилятор:
Авто-векторизация (без изменения кода)

Компилятор :
Директивы для векторизации

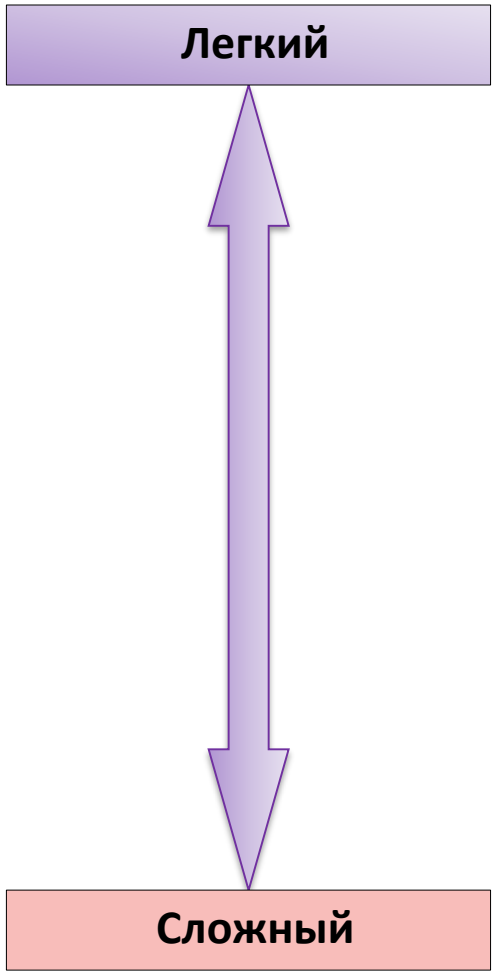
Компилятор:
#pragma simd / omp simd

Компилятор :
Intel® Cilk™ Plus Array Notation

Классы SIMD intrinsic
(например: `F32vec`, `F64vec`, …)

Векторные intrinsic функции
(например : `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Чистый ассемблер
(например : `[v]addps`, `[v]addss`, …)



habrahabr.ru

“Разработчик на распутье: как векторизовать?! ”

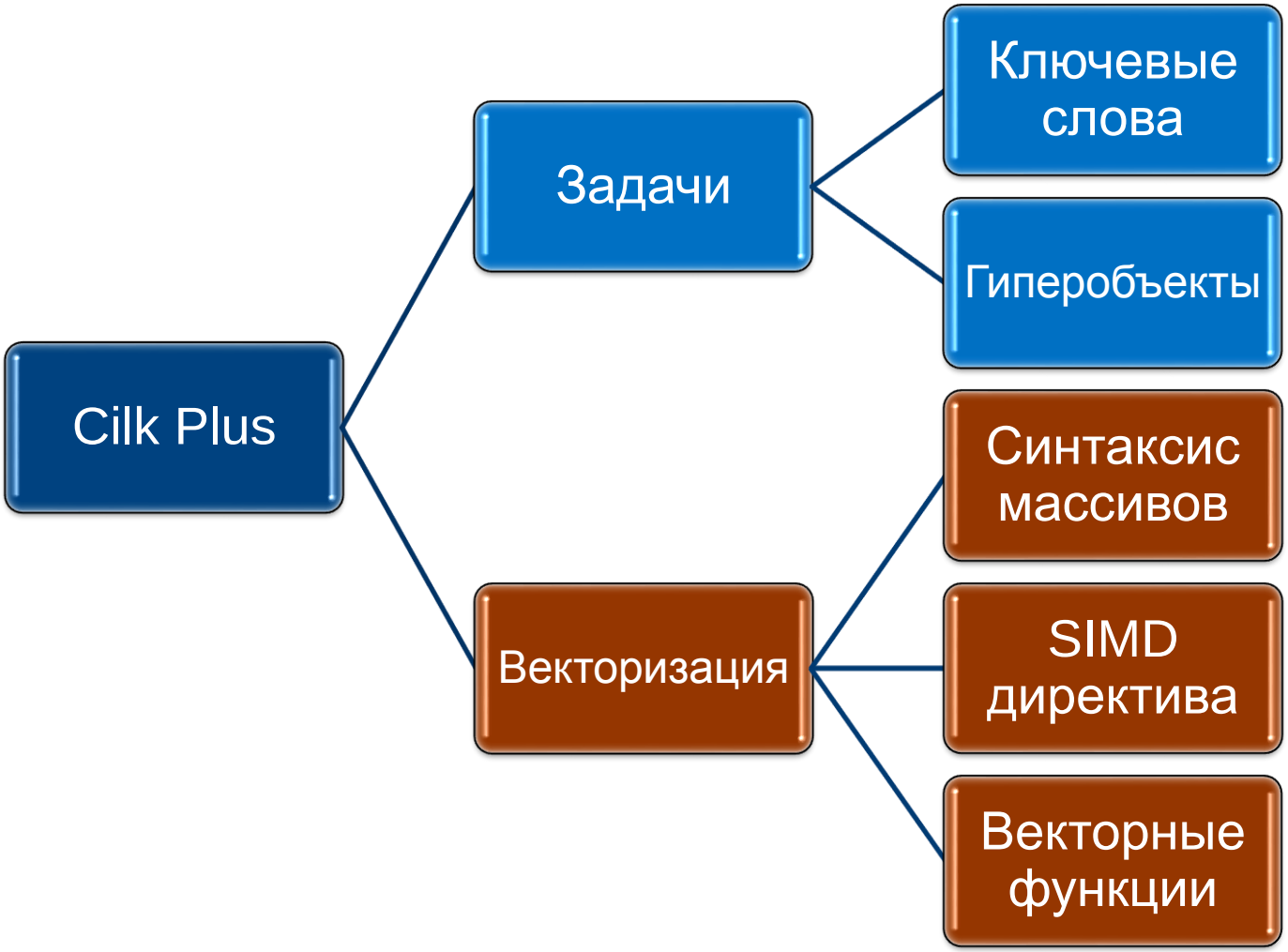
Лирическое отступление

Модели параллельного программирования

Intel® Cilk™ Plus	Intel® Threading Building Blocks	Специализированные библиотеки	Поддержка стандартов
Расширение C/C++	Библиотека шаблонов C++	Intel® Integrated Performance Primitives	Message Passing Interface (MPI)
Простой параллелизм	Открытый код	Intel® Math Kernel Library	OpenMP*
Открытый код	Продукт Интел		Coarray Fortran
Продукт Интел			OpenCL*

Используйте различные модели в зависимости от задачи!

Intel® Cilk Plus



habrahabr.ru

“Поговорим про Intel® Cilk™ Plus”

Всего три слова
для параллельности

- Параллельный и масштабируемый код с малыми накладными расходами
- Внутри – современный, лёгкий и эффективный планировщик задач
- С захватом работы для балансировки загрузки

```
cilk_for (int i=0; i<n; ++i) {
  Foo(a[i]);
}
```

Параллельные циклы – легко!

```
int fib(int n)
{
    if (n <= 2)
        return n;
    else {
        int x,y;
        x = fib(n-1);
        y = fib(n-2);
        return x+y;
    }
}
```

...код превращается...

```
int fib(int n)
{
    if (n <= 2)
        return n;
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return x+y;
    }
}
```

Лёгким движением руки....

...в параллельный код

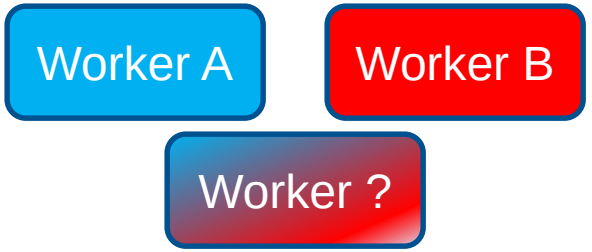
Открытая спецификация на cilkplus.org

Intel® Cilk™ Plus

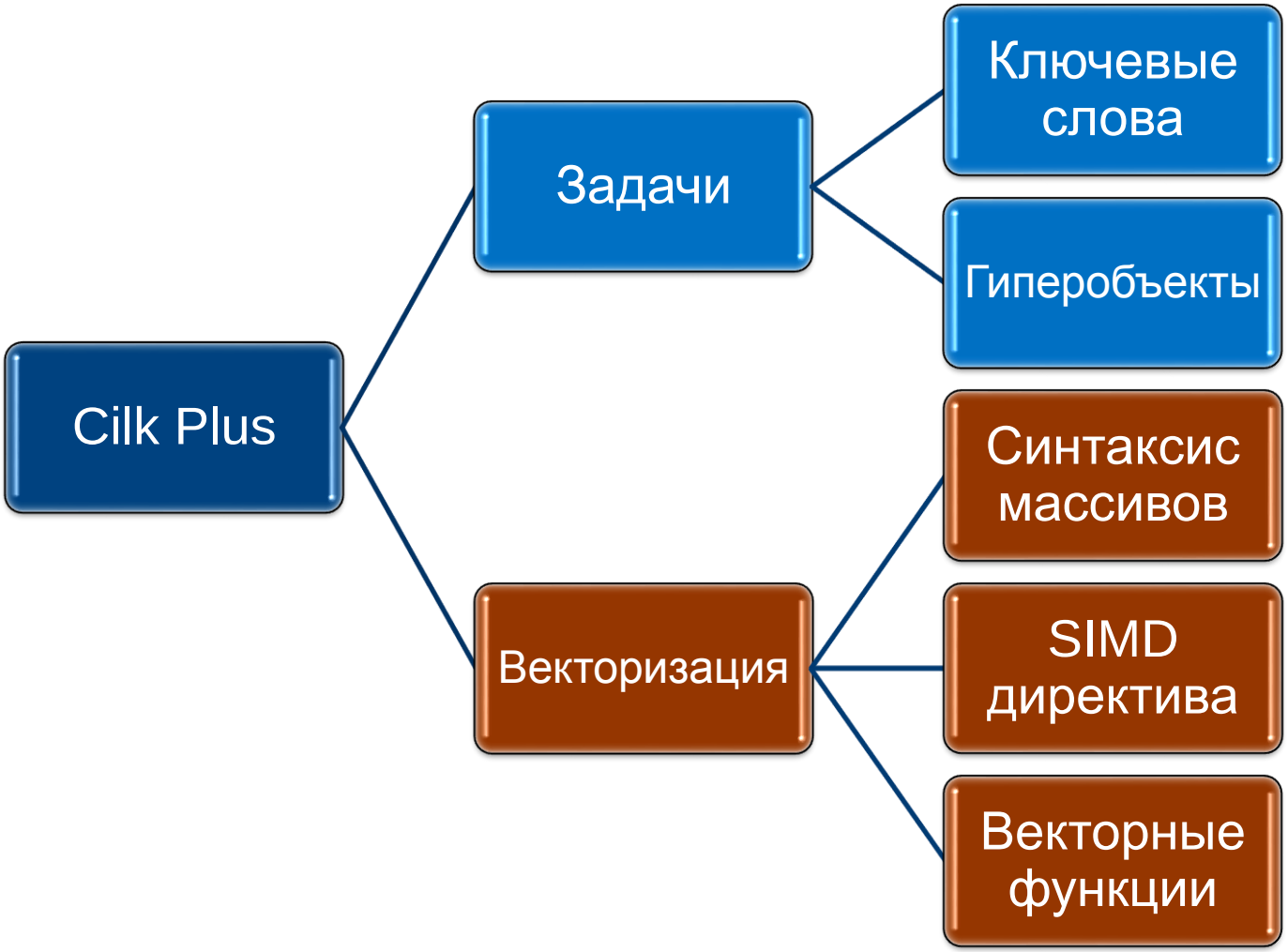
Параллелизм по задачам

```
void f()
{
  cilk_spawn g();
  work
  work
  work
  cilk_sync;
  work
}
```

```
void g()
{
  work
  work
  work
}
```



Intel® Cilk Plus



#pragma simd

C/C++: `#pragma simd [clause [,clause]…]`

Fortran: `!DIR$ SIMD [clause [,clause]…]`

Без дополнительных условий (clause), директива “заставляет” компилятор векторизовать цикл, игнорируя все зависимости (даже доказанные!)

Пример:

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

icl /Qvec-report6 test.cpp

remark: loop was not vectorized: existence of vector dependence.

#pragma simd

```
void addfl(float *a, float *b, float *c, float *d, float *e, int n)
{
#pragma simd
for(int i = 0; i < n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

remark: SIMD LOOP WAS VECTORIZED.

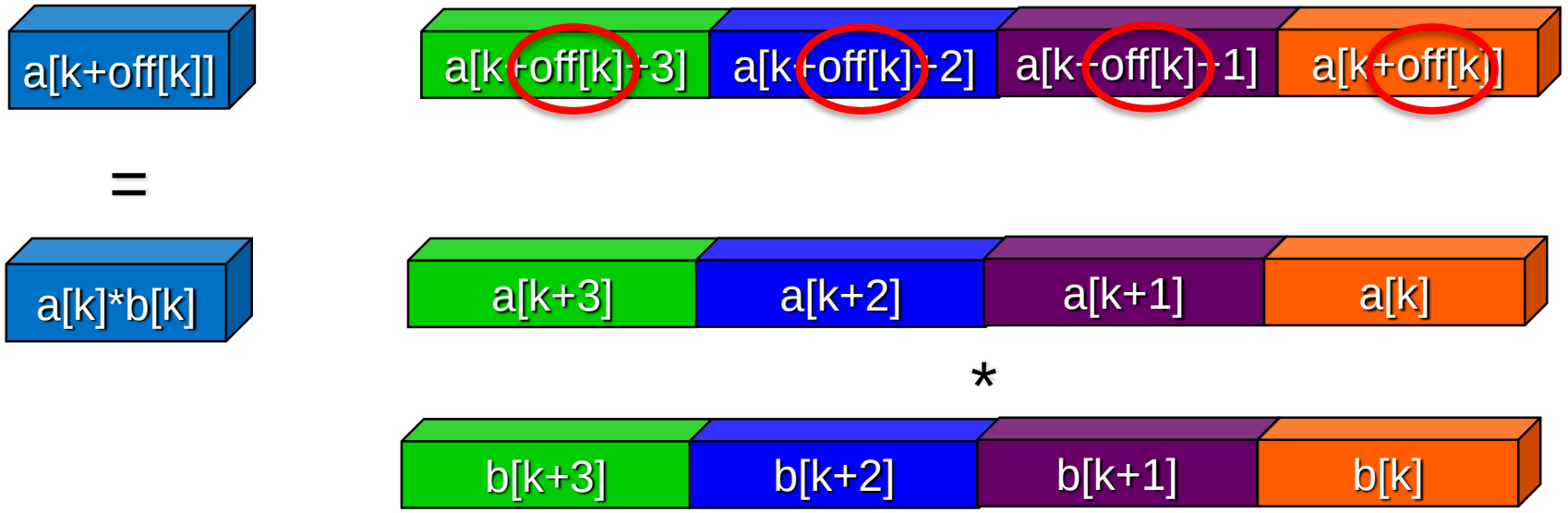
Можно задавать дополнительные условия (контекст), как в OpenMP директивах:

- vectorlength(VL)/vectorlengthfor(TP)
- private/firstprivate/lastprivate(v1[, v2, …])
- reduction(op1:v1[, …][, op2:v2[,…]])
- linear(v1[:st1][, v2[:st2], …])
- [no]assert, [no]vecremainder

Примерчик

```
void foo(float *restrict a, float *restrict b, int offmax, int n, int off[n])
{
  for(int k = 0; k < n - offmax; k++) a[k + off[k]] = a[k] * b[k];
}
```

Вспомним, что происходит при векторизации:



Примерчик

```
void foo(float *restrict a, float *restrict b, int offmax, int n, int off[n])
{
  for(int k = 0; k < n - offmax; k++) a[k + off[k]] = a[k] * b[k];
}
```

Компилятор не может векторизовать этот цикл, даже если нет «пересечения» a и b (restrict не помогает).
#pragma ivdep так же не работает из-за неэффективного доступа к **off[]**;
кроме того, это просто опасно:
С AVX компилятор может использовать вектор длиной 8 элементов.
Если хотя бы одно смещение **off** меньше, то возникает зависимость!

Решение: Если нам известно, что все значения смещений не меньше 4, то нам поможет директива simd с заданным условием **vectorlenght(4)**

```
void foo(float *restrict a, float *restrict b, int offmax, int n, int off[n])
{
#pragma simd vectorlength(4)
  for(int k = 0; k < n - offmax; k++) a[k + off[k]] = a[k] * b[k];
}
```

OpenMP* 4.0
Директива OMP SIMD

- Директива, указывающая на выполнение цикла с использованием SIMD инструкций

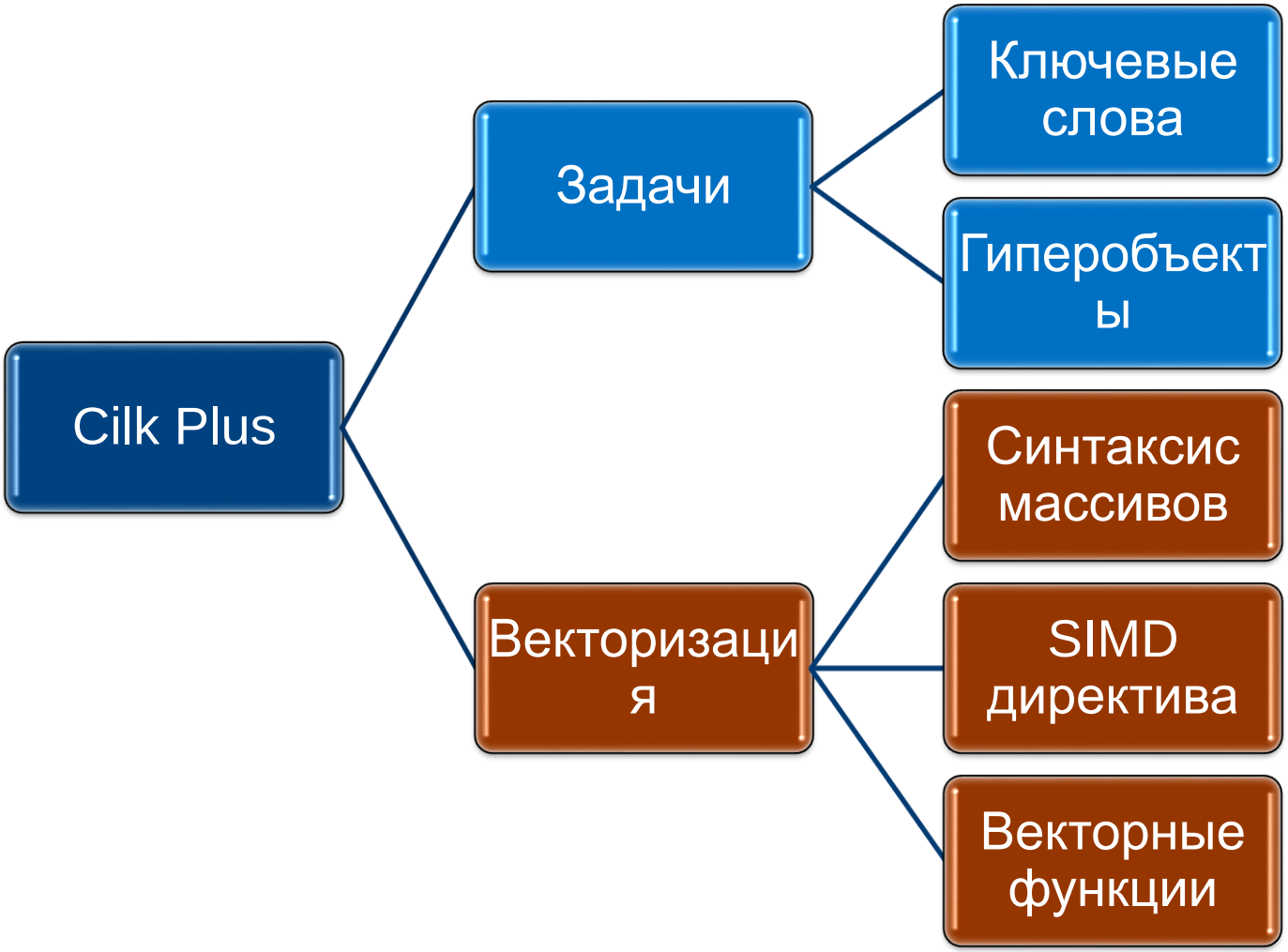
```
#pragma omp simd [clause[[,] clause] ...]
    <for-loop>
```

  - Аналог директивы SIMD из Intel Cilk Plus
  - Контекст схож (например, **linear**) с «интеловским», но есть различия (**aligned**)
  - Может быть совмещена с parallel-for:

```
#pragma omp parallel for simd [clause[[,] clause] ...] new-line
    <for-loop>
```

  - В начале, цикл делится на SIMD куски, которые выполняются различными потоками

Intel® Cilk Plus



Как векторизовать этот цикл?

- main.c

```
extern int my_add (int, int);
for (int i = 0; i < 10000; i++)
  {
     z[i] = my_add (x[i], y[i]);
  }
```

- add.c

```
int my_add (int x, int y)
{
   return x + y;
}
```

Векторные функции

- Способ векторизации циклов с вызовами функций
  - Функция обработки одного элемента
  - Применяется сразу к нескольким - к вектору элементов
- Добавляем __declspec(vector)
- Описываем параметры
  - uniform(p)
  - linear(i:1)

```
__declspec(vector(uniform(p),

linear(i:1)))
float add1(float *p, int i) {
  return p[i]+1;
}
```

OpenMP* 4.0
Директива OMP SIMD Declare

- Директива для функций, вызываемых в цикле

```
#pragma omp declare simd [clause[[,] clause] ...]
   <for-loop>
```

  - Аналог Векторных функций (Elemental Function) из Intel Cilk Plus
  - Контекст схож с Cilk Plus (но не идентичен)
  - OpenMP вводит директиву и в Фортране!!!

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)

Различные способы векторизации

Компилятор:
Авто-векторизация (без изменения кода)

Компилятор :
Директивы для векторизации

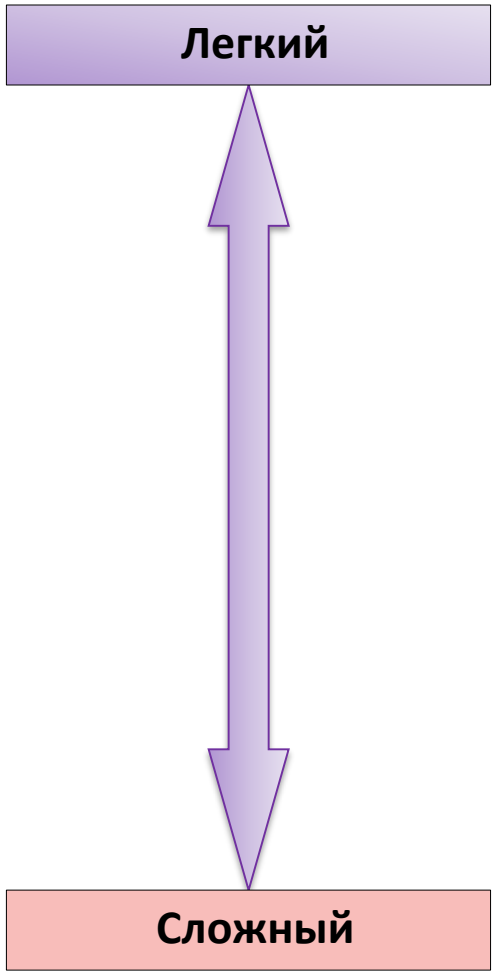
Компилятор:
#pragma simd / omp simd

Компилятор :
Intel® Cilk™ Plus Array Notation

Классы SIMD intrinsic
(например: `F32vec`, `F64vec`, …)

Векторные intrinsic функции
(например : `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

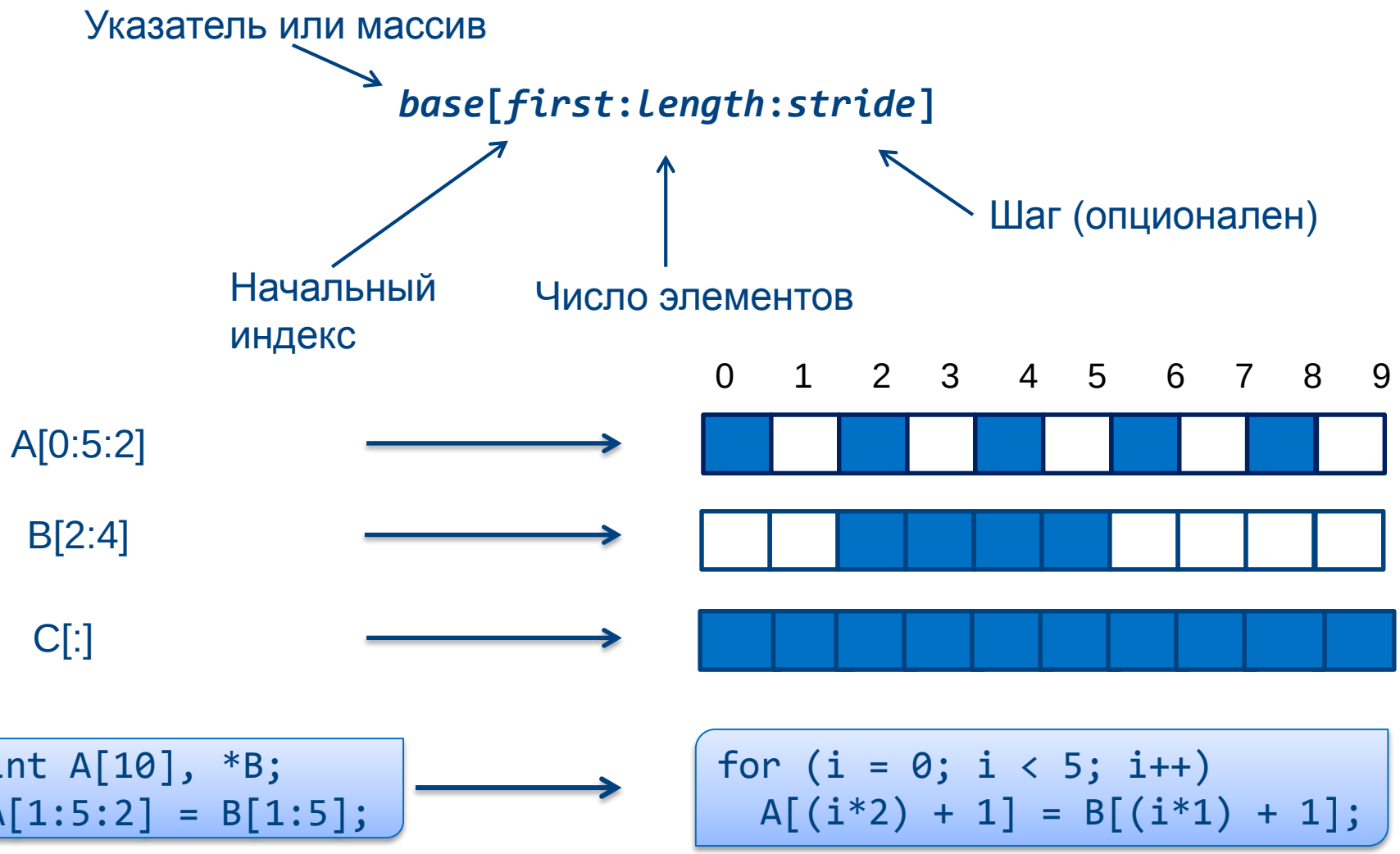
Чистый ассемблер
(например : `[v]addps`, `[v]addss`, …)



habrahabr.ru

“Разработчик на распутье: как векторизовать?! ”

Intel® Cilk™ Plus – Array notation



Ясный синтаксис для простой векторизации

Примерчик
скалярное произведение

Обычный код

```
float dot_product(int sz, float A[sz], float B[sz])
{
    int i;
    float dp = 0.0f;
    for (i = 0; i < sz; i++) {
        dp += A[i] * B[i];
    }
    return dp;
}
```

Векторизуемо?
Возможно…

Array notation

```
float dot_product(int sz, float A[sz], float B[s
{
    return __sec_reduce_add(A[:] * B[:]);
}
```

Векторизуемо?
Точно!

Различные способы векторизации

Компилятор:
Авто-векторизация (без изменения кода)

Компилятор :
Директивы для векторизации

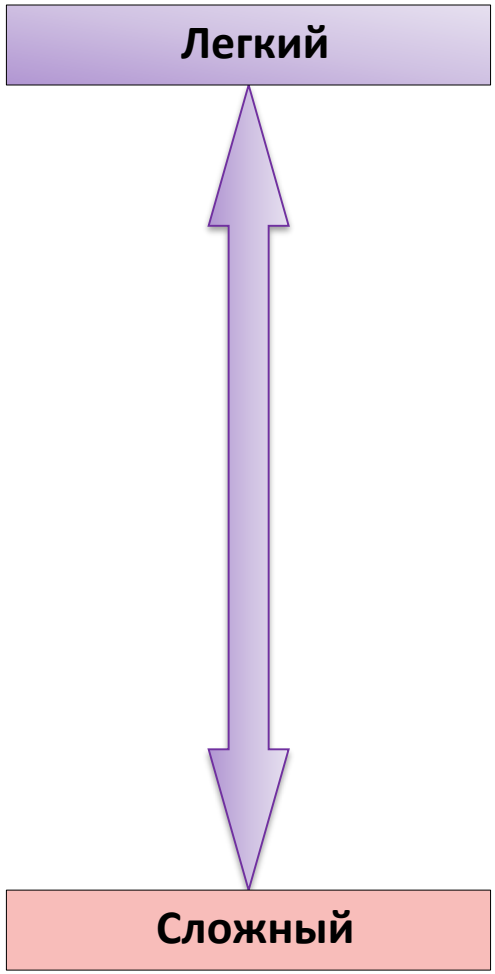
Компилятор:
#pragma simd / omp simd

Компилятор :
Intel® Cilk™ Plus Array Notation

Классы SIMD intrinsic
(например: `F32vec`, `F64vec`, …)

Векторные intrinsic функции
(например : `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Чистый ассемблер
(например : `[v]addps`, `[v]addss`, …)



habrahabr.ru

“Разработчик на распутье: как векторизовать?! ”

Интринсики

Заголовочные файлы:

- **`mmintrin.h`**: MMX™
- **`xmmintrin.h`**: Intel® SSE
- **`emmintrin.h`**: Intel® SSE2
- **`pmmintrin.h`**: Intel® SSE3
- **`tmmintrin.h`**: Intel SSSE3
- **`smmintrin.h`**: Intel® SSE4.1
- **`nmmintrin.h`**: Intel® SSE4.2
- **`wmmintrin.h`**: AESNI & PCLMULQDQ; no SIMD
- **`immintrin.h`**: Intel® AVX
- **`zmmintrin.h`**: Intel® MIC

Перечислены в хронологическом порядке

Нужно использовать только **`immintrin.h`**!

Остальные в качестве документации типов и функций.

Интринсики

Новые типы данных:

- **`__m512/__m256/__m128`**:
  16/8/4 single-precision FP

- **`__m512d/__m256d/__m128d`**:
  8/4/2 double-precision FP

- **`__m512i/__m256i/__m128i`**:
  Различные целочисленные значения (8, 16, 32 & 64 бит)

Перечислены в хронологическом порядке

Нужно использовать только **`immintrin.h`**!

Остальные в качестве документации типов и функций.

Интринсики

У SSE или AVX VEX.128 интрисиков есть префикс `_mm_` :

```
_mm_exp2_ps(__m128)
_mm_add_pd(__m128d, __m128d)
```

У AVX VEX.256 префикс `_mm256_`:

```
_mm256_exp2_ps(__m256)
_mm256_add_pd(__m256d, __m256d)
```

Возможно использовать SSE и AVX интринсики вместе, но нежелательно

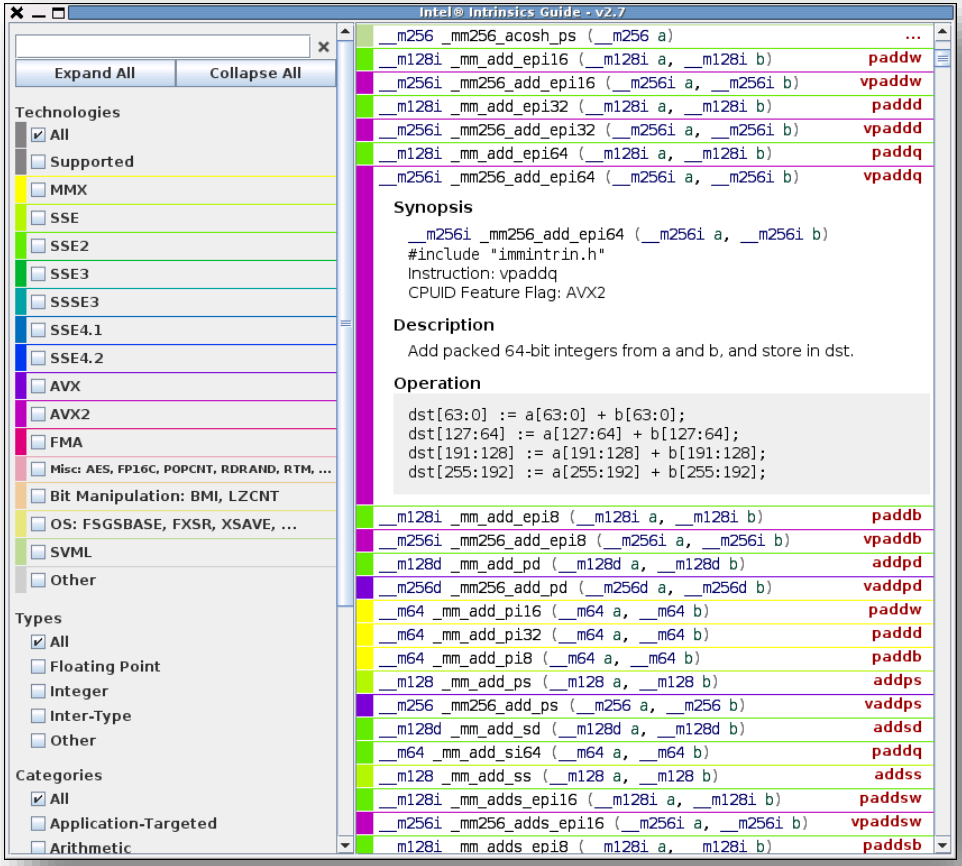
Пример использования AVX интринсиков:

```
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 4) {
        __m256d a = _mm256_load_pd(&A[i]);
        __m256d b = _mm256_load_pd(&B[i]);
        __m256d c = _mm256_add_pd(a, b);
        _mm256_store_pd(&C[i], c);
}
```

Intel® Intrinsics Guide

- Список всех поддерживаемых интринсиков
- Сортировка по версиям SIMD
- Быстрый поиск
- Подробное описание



Скачать здесь:
http://software.intel.com/en-us/articles/intel-intrinsics-guide

Различные способы векторизации

Компилятор:
Авто-векторизация (без изменения кода)

Компилятор :
Директивы для векторизации

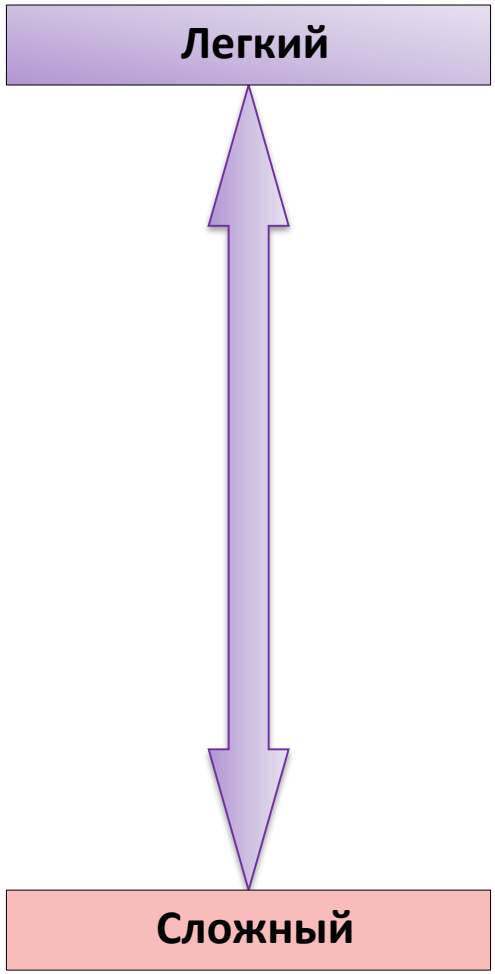
Компилятор:
#pragma simd / omp simd

Компилятор :
Intel® Cilk™ Plus Array Notation

Классы SIMD intrinsic
(например: `F32vec`, `F64vec`, …)

Векторные intrinsic функции
(например : `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Чистый ассемблер
(например : `[v]addps`, `[v]addss`, …)



habrahabr.ru

“Разработчик на распутье: как векторизовать?! ”

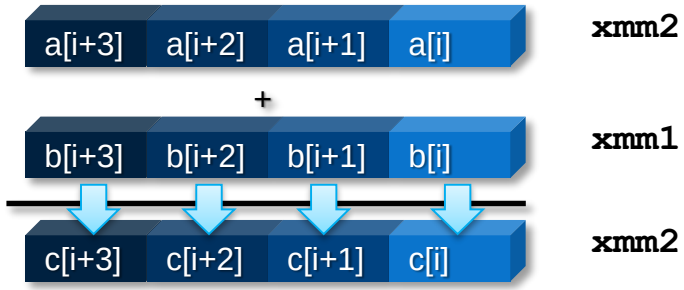
SSE: Packed vs. Scalar

- **“Упакованные” (packed)** инструкции оперируют со всеми элементами вектора
- Большинство таких инструкций имеют “**скалярную” (scalar)** версию, работающую только с одним элементом
- Для использования SIMD возможностей необходимо избегать использования скалярных версий инструкций

Packed single-precision FP Addition:

```
addps xmm2, xmm1
```

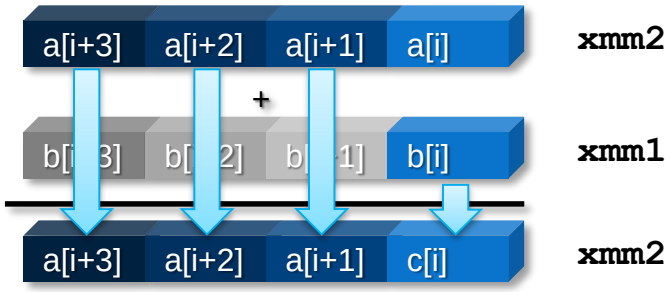
single-precision FP data type
packed execution mode



Scalar single-precision FP Addition:

```
addss xmm2, xmm1
```

single-precision FP data type
scalar execution mode



Переход от SSE к AVX

- 2 режима:
  - **VEX.256**:
    Операции с 256-битными векторам
  - **VEX.128**:
    Операции с младшими 128 битами вектора
    Старшие 128 бит обнуляются

VEX.256:
```
vaddps ymm1, ymm2, ymm3
```
VEX.128:
```
vaddps xmm1, xmm2, xmm3
vaddss xmm1, xmm2, xmm3
```
v = VEX encoding (AVX)
3 операнда в иснтрукциях!

- Переход от SSE:
  - Практически все SSE инструкции доступны как VEX.128 версии, например:
    addps/addss: vaddps/vaddss xmm1, xmm2, xmm3
  - Некторые из них работают в режиме VEX.256:
    vaddps ymm1, ymm2, ymm3
  - Скалярных версий SSE инструкций в VEX.256 режиме **нет**:
    ~~**vaddss ymm1, ymm2, ymm3**~~
  - Совместное использование SSE & AVX инструкция возможно, но влечёт потерю производительности (**state change penalty)**; избегайте такое использование!
- AVX использует трёх- и четырёх-операндные инструкции!

Резюме

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

Ссылки

[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

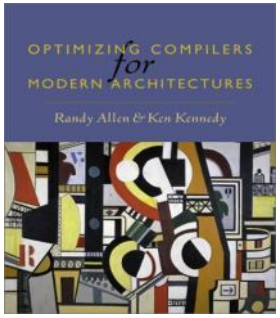
– http://www.intel.com/products/processor/manuals/index.htm

[3] Randy Allen, Ken Kennedy: “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”

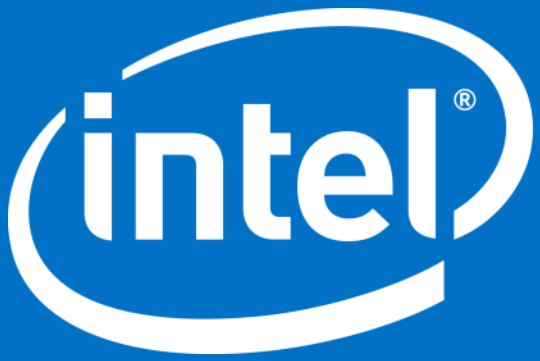
[4] Intel Software Forums, Knowledge Base, White Papers, Tools Support etc

– http://software.intel.com

[5] Steven S. Muchnik, “Advanced Compiler Design and Implementation”



Q&A


intel

Уведомление об оптимизации

Уведомление об оптимизации

Компиляторы Intel могут не обеспечивать для процессоров других производителей такой же уровень оптимизации для оптимизаций, которые не являются присущими только процессорам Intel. В число этих оптимизаций входят наборы команд SSE2, SSE3 и SSSE3, а также другие оптимизации. Корпорация Intel не гарантирует наличие, функциональность или эффективность оптимизаций микропроцессоров других производителей. Содержащиеся в данной продукции оптимизации, зависящие от микропроцессора, предназначены для использования с микропроцессорами Intel. Некоторые оптимизации, не характерные для микроархитектуры Intel, резервируются только для микропроцессоров Intel. Более подробную информацию о конкретных наборах команд, покрываемых настоящим уведомлением, можно получить в соответствующих руководствах пользователя и справочниках на продукт.

Уведомление, редакция № 20110804