Нижегородский государственный университет
им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

Лекция №5
Элементы оптимизации прикладных программ для Intel Xeon Phi. Intel Compiler

При поддержке компании Intel

Горшков А.В.
Кафедра математического обеспечения ЭВМ

Содержание

Введение

- Intel Xeon Phi – новый сопроцессор от компании Intel, призванный существенно ускорить процесс вычислений для некоторого класса задач, алгоритмы решения которых допускают существенную степень параллелизма и векторизации
- Сопроцессор основан на архитектуре Intel Many Integrated Core (MIC), содержит несколько десятков x86 CPU ядер, поддерживает сотни потоков исполнения
- Основное достоинство – для создания программ для Intel Xeon Phi используются знакомые инструменты (C/C++ Compiler, OpenMP, MPI), не требуется существенной модификации кода
- **НО:** для эффективной работы требуется оптимизация

Методы работы с сопроцессором

- Явная схема работы с памятью:
  - Использование директив компилятора: **#pragma offload**
  - Явное управление обменами данных с сопроцессором
- Неявная схема работы с памятью:
  - Использование ключевых слов расширения **Intel Cilk Plus**
  - Использование участков памяти, разделяемых между процессором и сопроцессором

1. Явная схема работы с памятью

Явная схема работы с памятью…

```
__attribute__((target(mic))) void func(float* a,
    float* b, int count, float c, float d)
{
    #pragma omp parallel for
    for (int i = 0; i < count; ++i)
    {
        a[i] = b[i]*c + d;
    }
}

int main()
{
    const int count = 100;
    float a[count], b[count], c, d;
    …
    #pragma offload target(mic)
        func(a, b, count, c, d);
    …
}
```

Явная схема работы с памятью: C/C++

Описание	C/C++ синтаксис	Семантика
Директива offload	#pragma offload <clauses> <statement>	Запуск участка кода на сопроцессоре или CPU
Ключевое слово для указания MIC функции или переменной	__attribute__((target(mic)))	Компиляция функции или объявление переменной одновременно для CPU и сопроцессора
Указание MIC блока кода	#pragma offload_attribute(push, target(mic)) … #pragma offload_attribute(pop)	Компиляция блока кода одновременно для CPU и сопроцессора
Отдельная передача данных	#pragma offload_transfer target(mic)	Обеспечивает синхронную или асинхронную передачу данных между CPU и сопроцессором

Явная схема работы с памятью: Fortran

Описание	C/C++ синтаксис	Семантика
Директивы offload	!dir$ omp offload <clauses> <statement>	Запуск параллельного (OpenMP) участка кода на сопроцессоре или CPU
	dir$ offload <clauses> <statement>	Запуск участка кода (вызов функции) на сопроцессоре или CPU
Ключевое слово для указания MIC функции или переменной	!dir$ attributes offload:<mic> :: <ret-name> OR <var1,var2,…>	Компиляция функции или объявление переменной одновременно для CPU и сопроцессора
Отдельная передача данных	!dir$ offload_transfer target(mic)	Обеспечивает синхронную или асинхронную передачу данных между CPU и сопроцессором

Явная схема работы с памятью…

```
#pragma offload target(mic)
```

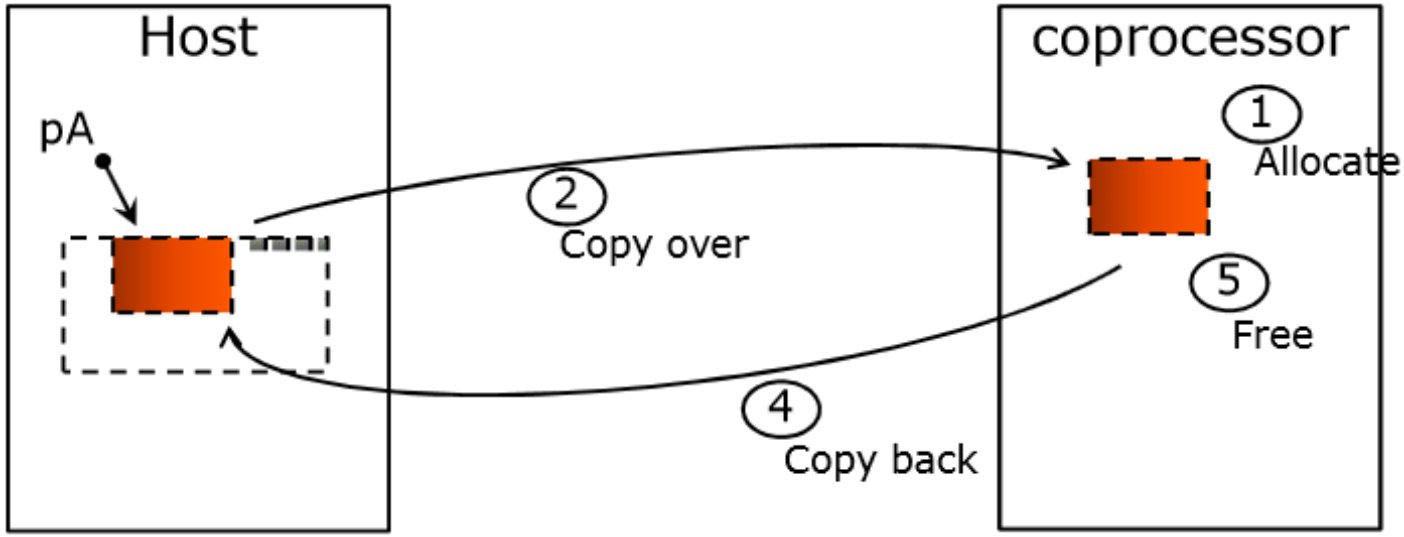
- Код будет выполнен на одном из доступных сопроцессоров
- Если сопроцессоров нет или они не доступны, код будет выполнен на центральном процессоре

```
#pragma offload target(mic:n)
```

- Код будет выполнен на сопроцессоре с номером (n % <ОбщееЧислоXeonPhi>)
- Если сопроцессор недоступен – возникнет ошибка времени исполнения

Явная схема работы с памятью: механизм вызова функции на сопроцессоре

- На сопроцессоре выделяется память под массивы **a[]** и **b[]**.
- Выполняется передача данных (всех массивов и переменных) на сопроцессор.
- Функция запускается на исполнение на Intel Xeon Phi.
- Выполняется передача данных (всех массивов) с сопроцессора в оперативную память.
- Удаляется память на сопроцессоре.



Явная схема работы с памятью…

Операторы (clauses)	C/C++ синтаксис	Семантика
Target спецификация	target(name[:card_number])	Явное указание того, где запускать код
Условный offload	if (condition)	Запуск кода, если условие истинно
Вход	in (var-list [modifiers])	Копирование с хоста на сопроцессор
Выход	out (var-list [modifiers])	Копирование с сопроцессора на хост
Вход и выход	inout (var-list [modifiers])	Копирование в обе стороны
Отмена копирования	nocopy (var-list [modifiers])	Локальные данные сопроцессора
Асинхронный offload	signal(signal-slot)	Режим асинхронного offload'а
Асинхронный offload	wait(signal-slot)	Ожидание завершения асинхронного offload'а

Явная работа с памятью…

Модификаторы (modifiers)	C/C++ синтаксис	Семантика
Размер памяти при копировании	length (element-count-expr)	Размер указывается в элементах, а не в байтах
Условное выделение	alloc_if (condition)	Выделить память на сопроцессоре, если условие истинно
Условное освобождение	free_if (condition)	Удалить память на сопроцессоре, если условие истинно
Выравнивание	align (expression)	Задание минимального выравнивания данных на сопроцессоре

Явная схема работы с памятью: статическая память

```
__attribute__((target(mic))) void func(float* a,
    float* b, int count, float c, float d)
{
    #pragma omp parallel for
    for (int i = 0; i < count; ++i)
    {
        a[i] = b[i]*c + d;
    }
}

int main()
{
    const int count = 100;
    float a[count], b[count], c, d;
    …
    #pragma offload target(mic) in(b) out(a)
        func(a, b, count, c, d);
    …
}
```

Явная схема работы с памятью: динамическая память…

```
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)

void f()
{
    int *p = (int *)malloc(100*sizeof(int));
    // Memory is allocated for p,
    // data is sent from CPU and retained
    #pragma offload target(mic:0) in(p[0:100] : ALLOC)
    { p[6] = 66; }
    …
    // Memory for p reused from previous offload
    // and retained once again
    // Fresh data is sent into the memory
    #pragma offload target(mic:0) in(p[0:100] : REUSE)
    { p[6] = 66; }
    …
    // Memory for p reused from previous offload,
    // freed after this offload.
    // Final data is pulled from coprocessor to CPU
    #pragma offload target(mic:0) out(p[0:100] : FREE)
    { p[7] = 77; }
    …
}
```

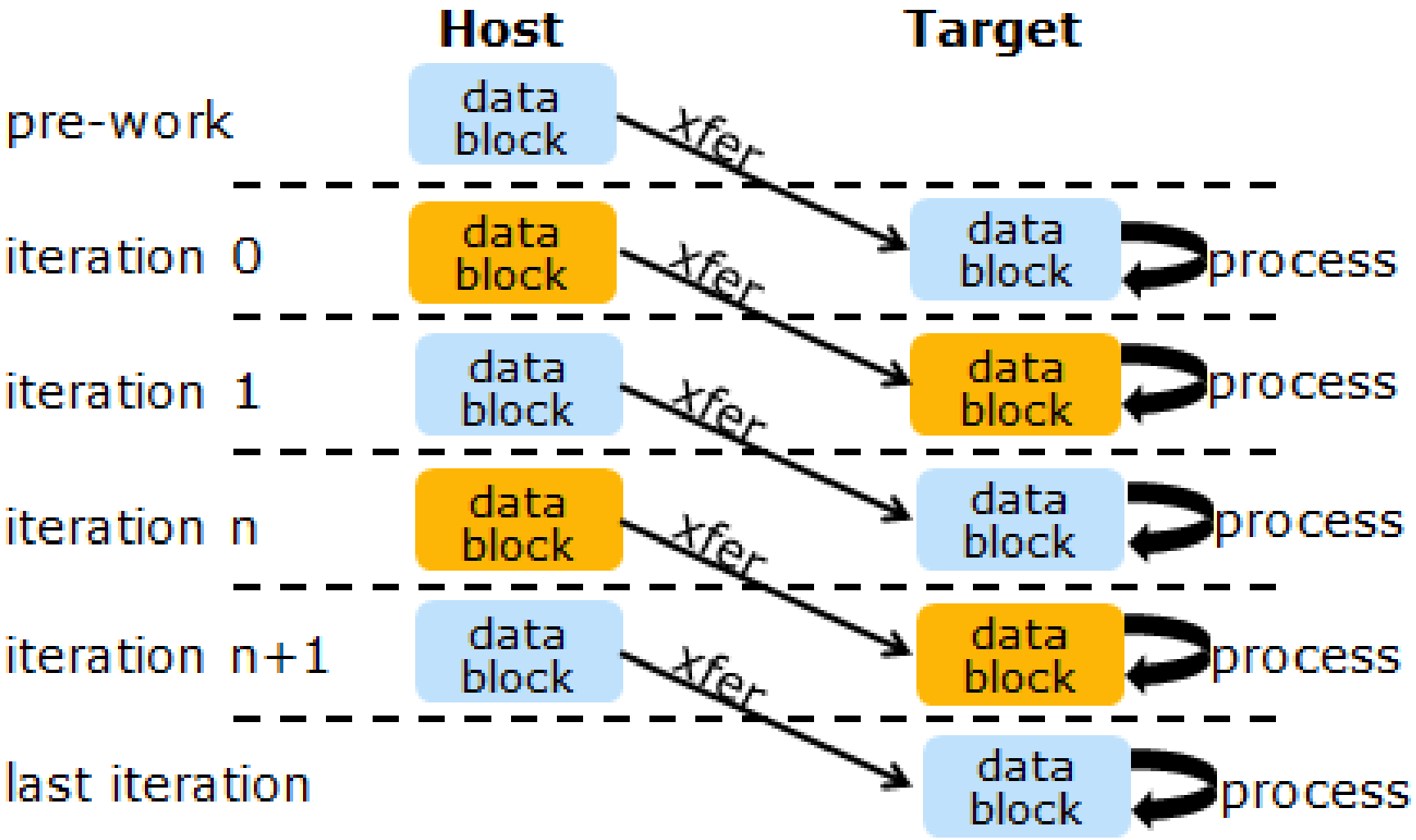
Явная схема работы с памятью: динамическая память

```
void f()
{
    int *p;
    …
    // The nocopy clause ensures CPU values pointed to by p
    // are not transferred to coprocessor
    #pragma offload target(mic:0) nocopy(p)
    {
        // Allocate dynamic memory for p on coprocessor
        p = (int *)malloc(100);
        p[0] = 77;
        …
    }
    ..
    // The nocopy clause ensures p is not altered
    // by the offload process
    #pragma offload target(mic:0) nocopy(p)
    {
        // Reuse dynamic memory pointed to by p
        … = p[0]; // Will be 77
    }

}
```

Явная работа с памятью: одновременное выполнение на процессоре и сопроцессоре

```
double __attribute__((target(mic)))
myworkload(double input)
{
    // do something useful here
    return result;
}

int main(void)
{
    //…. Initialize variables
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma offload target(mic)
                result1= myworkload(input1);
        }
        #pragma omp section
        {
            result2= myworkload(input2);
        }
    }
}
```

Явная схема работы с памятью: схема двойной буферизации…



Явная схема работы с памятью: схема двойной буферизации…

```
int main(int argc, char* argv[])
{
    // Allocate & initialize in1, res1,
    // in2, res2 on host
    #pragma offload_transfer target(mic:0) in(cnt)\
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(1) free_if(0))

    do_async_in();

    // Free MIC memory
    #pragma offload_transfer target(mic:0) \
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(0) free_if(1))

    return 0;
}
```

Явная схема работы с памятью: схема двойной буферизации…

```
void do_async_in()
{
    float lsum;
    int i;
    lsum = 0.0f;

    #pragma offload_transfer target(mic:0) \
        in(in1 : length(cnt) \
        alloc_if(0) free_if(0)) signal(in1)
```

Явная схема работы с памятью: схема двойной буферизации…

```
for (i = 0; i < iter; i++)
    {
        if (i % 2 == 0)
        {
            #pragma offload_transfer target(mic:0) \
                if(i !=iter - 1) in(in2 : length(cnt) \
                alloc_if(0) free_if(0)) signal(in2)

            #pragma offload target(mic:0) nocopy(in1) \
                wait(in1) out(res1 : length(cnt) \
                alloc_if(0) free_if(0))
            {
                compute(in1, res1);
            }

            lsum = lsum + sum_array(res1);
        }
```

Явная схема работы с памятью: схема двойной буферизации

```
        else
        {
            #pragma offload_transfer target(mic:0) \
                if(i != iter - 1) in(in1 : length(cnt) \
                alloc_if(0) free_if(0))  signal(in1)

            #pragma offload target(mic:0) nocopy(in2) \
                wait(in2) out(res2 : length(cnt) \
                alloc_if(0) free_if(0))
            {
                compute(in2, res2);
            }

            lsum = lsum + sum_array(res2);
        }
    } // for
} // do_async_in()
```

Явная схема работы с памятью: работа со структурами…

```
typedef struct {
    int m1;
    char *m2;
} nbwcs;

void sample11()
{
    nbwcs struct1;
    struct1.m1 = 10;
    struct1.m2 = malloc(11);
    int m1;
    char *m2;
```

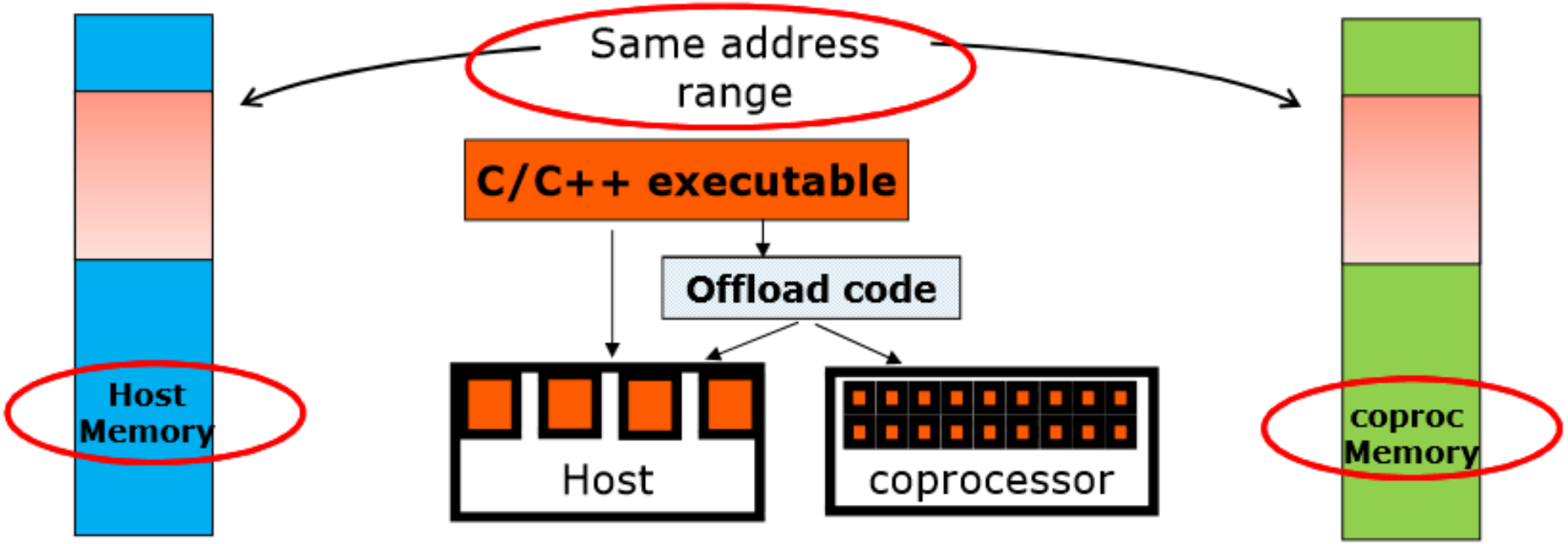
Явная схема работы с памятью: работа со структурами

```
    // Disassemble the struct for transfer to target
    m1 = struct1.m1;
    m2 = struct1.m2;

    #pragma offload target(mic) inout(m1) inout(m2 :
length(11)
    {
        nbwcs struct2;
        // Reassemble the struct on the target
        struct2.m1 = m1;
        struct2.m2 = m2;
        ...
    }
    ...
}
```

2. Неявная схема работы с памятью

Неявная схема работы с памятью…

- Основана на использовании ключевых слов
- Позволяет работать со сложными типами данных
- Поддерживается только в языках C/C++



Неявная схема работы с памятью…

- Для выделения участка разделяемой памяти динамически необходимо воспользоваться функциями:

```
void *_Offload_shared_malloc(size_t size);
void *_Offload_shared_aligned_malloc(size_t
size, size_t alignment);
```

- Удаление памяти выполняется соответственно с помощью функций:

```
void _Offload_shared_free(void *p);
void _Offload_shared_aligned_free(void *p);
```

Неявная схема работы с памятью: _Cilk_shared

Что	Синтаксис	Семантика
Функции	int _Cilk_shared f(int x) { return x + 1; }	Компиляция для CPU и MIC, функция может быть вызвана на любой стороне
Глобальные переменные	_Cilk_shared int x = 0;	Переменная доступна на обеих сторонах
Статические переменные	static _Cilk_shared int x;	Переменная доступна на обеих сторонах, видна только в рамках файла/функции
Классы	class _Cilk_shared x {…};	Поля, методы и операторы класса доступны на обеих сторонах
Указатели на разделяемые данные	int _Cilk_shared *p;	Локальный указатель на разделяемые данные
Разделяемые указатели	int* _Cilk_shared p;	Разделяемый указатель, должен указывать на разделяемые данные
Блоки кода	#pragma offload_attribute(push, target(mic)) … #pragma offload_attribute(pop)	Аналог _Cilk_shared для целого блока кода

Неявная схема работы с памятью: _Cilk_offload

Функциональность	Пример	Описание
Вызов функции на сопроцессоре	x = _Cilk_offload func(y);	Функция выполняется на сопроцессоре, если это возможно
	x = _Cilk_offload_to(card_num) func(y);	Функция должна выполниться на указанном сопроцессоре
Асинхронный вызов на сопроцессоре	x = _Cilk_spawn _Cilk_offload func(y);	Неблокирующее выполнение на сопроцессоре
Параллельный цикл for на сопроцессоре	_Cilk_offload _Cilk_for(i=0; i<N; ++i) { a[i] = b[i] + c[i]; }	Цикл выполняется параллельно на сопроцессоре

Неявная схема работы с памятью

```
void findpi()
{
  int count = 10000;

  // Initialize shared global
  // variables
  pi = 0.0f;

  //  Compute pi on target
  _Cilk_offload
     compute_pi(count);

  pi /= count;
}
```

```
// Shared variable declaration for pi
_Cilk_shared float pi;

// Shared function declaration for
// compute
_Cilk_shared void compute_pi(int count)
{
    int i;

    #pragma omp parallel for \
                reduction(+:pi)
    for (i=0; i<count; i++)
    {
        float t = (float)((i+0.5f)/count);
        pi += 4.0f/(1.0f+t*t);
    }
}
```

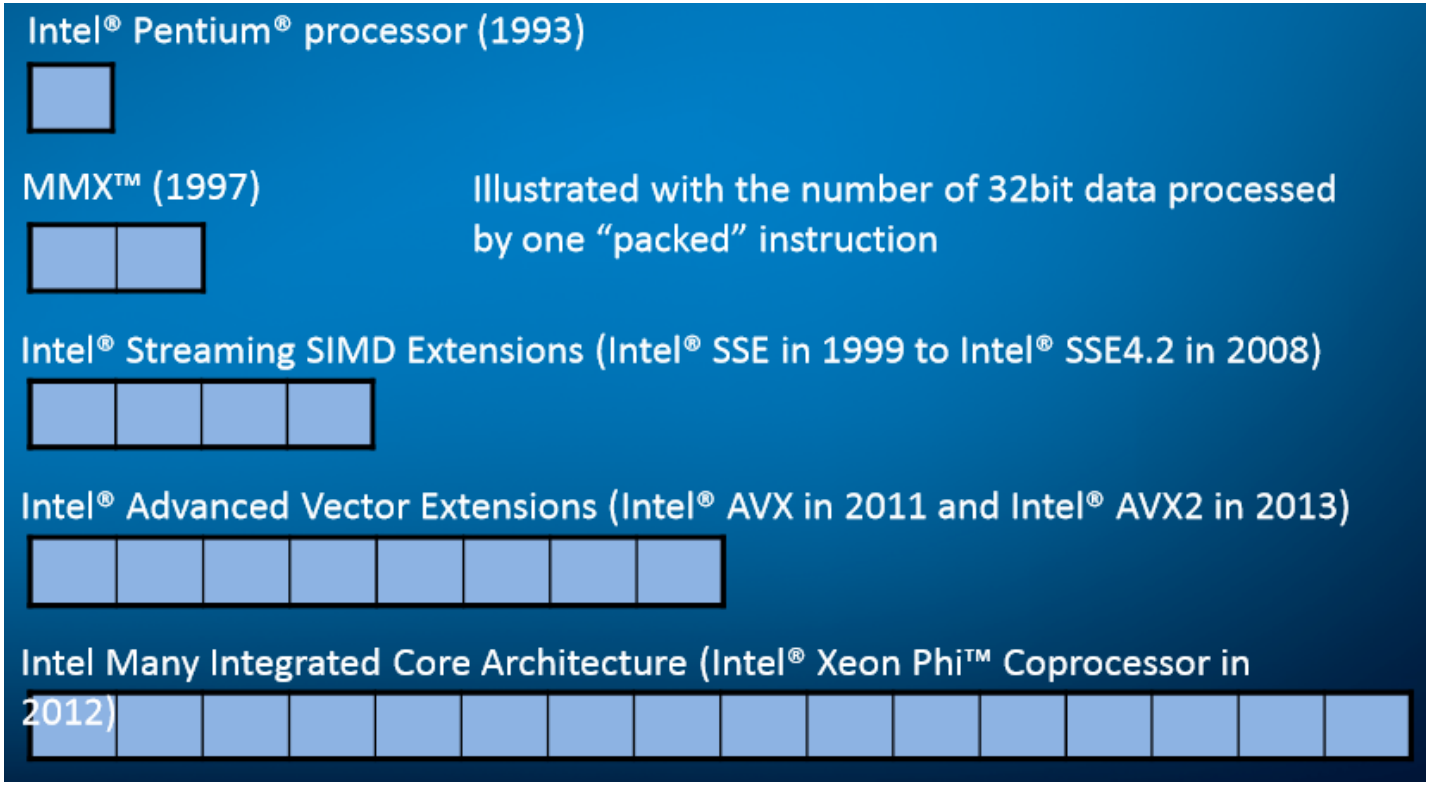
Сравнение явной и неявной схемы

	Явная схема	Неявная схема
Типы данных, для которых возможно автоматическое копирование	Скаляры, массивы, структуры с возможностью побитового копирования	Все типы данных
Когда происходит передача данных	Пользователь может явно контролировать передачу данных для каждой Offload директивы	Разделяемые данные синхронизируются в начале и в конце операторов _Cilk_offload
Когда Offload код копируется на сопроцессор	При первом вызове #pragma offload	В начале работы программы
Поддержка языков программирования	Fortran, C, C++ (без возможности передачи объектов класса)	C, C++
Синтаксис	Директивы #pragma offload (C/C++) и !dir$ offload (Fortran)	Ключевые слова _Cilk_shared и _Cilk_offload
Используется для…	Передачи непрерывных блоков данных	Передачи сложных структур данных или многих маленьких участков данных

3. Векторизация

Векторизация…

```
float *restrict A, *B, *C;
for (int i = 0; i < n; ++i)
{
    A[i] = B[i] + C[i];
}
```



Векторизация…

- В некоторых простых случаях компилятор может сам векторизовать ваш код
- Можно использовать возможности параллельного расширения Intel Cilk Plus для самостоятельной векторизации кода
- Можно воспользоваться библиотеками с уже векторизованным кодом, например, Intel MKL
- Можно использовать язык ассемблера с векторными инструкциями для оптимизации критичных участков кода, либо, что более удобно, оболочки этих инструкций в виде функций языка Си (intrinsics). Существуют также библиотеки классов SIMD

Автоматическая векторизация…

```
for(i=0;i<*p;i++)
{
     A[i] = B[i]*C[i];
     sum = sum + A[i];
}
```

- *p является инвариантом цикла
- A, B и C являются инвариантами цикла
- A[] не является другим именем для B[], C[] и/или sum (нет перекрытия по памяти между этими данными)
- sum не является другим именем для B[] и/или C[] (нет перекрытия по памяти между этими данными)
- операция “+” является ассоциативной
- ожидается ускорение векторной версии данного кода по отношению к скалярной

Автоматическая векторизация

- Для компилятора часто сложным является ответ на вопрос о том, что массив A[] не перекрываются с массивами B[] и C[]. Для того чтобы отразить это в синтаксисе языка, можно объявить указатель A с ключевым словом restrict:

```
float* restrict A;
```

- Для того чтобы сказать компилятору об отсутствии зависимостей в цикле, используется директива **#pragma ivdep** перед телом цикла:

```
#pragma ivdep
for(i=0; i < *p; i++)
{
    A[i] = B[i]*C[i];
    sum = sum + A[i];
}
```

Использование директивы SIMD…

```
void add_floats(float *a, float *b, float *c, float *d,
float *e, int n){
    int i;

    #pragma simd
    for (i=0; i<n; i++){
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
```

Использование директивы SIMD…

- ***vectorlength(n)*** – данный параметр определяет количество итераций цикла, которые могут быть выполнены независимо за одну векторную операцию

```
#pragma simd vectorlength(4)
for (i = 0; i < n; i++) {
    a[i] = a[i] + b[i] + c[i];
}
```

- ***linear(var1:step1 [,var2:step2]...)*** – этот параметр сообщает компилятору, что переменные *var* инкрементируются с шагом *step* на каждой итерации цикла

```
#pragma simd linear(k:j)
for (i = 0; i < n; i += step) {
    k += j;
    a[i] = a[i] + b[n - k + 1];
}
```

Использование директивы SIMD…

- ***reduction(oper:var1 [,var2]…)*** – параметр аналогичен соответствующему параметру директивы OMP, обеспечивает выполнение операции редукции для заданного списка переменных по окончании выполнения операций цикла

```
int x = 0;
#pragma simd reduction(+:x)
for (i = 0; i < n; ++i)
    x = x + A[i];
```

- ***private(var1[, var2]...)*** – параметр аналогичен соответствующему параметру директивы OMP, сообщает компилятору о необходимости создания отдельного экземпляра переменной для каждой итерации цикла. Определены также параметры ***firstprivate*** и ***lastprivate***, позволяющие задать начальное и конечное значение переменной в рамках каждой итерации цикла

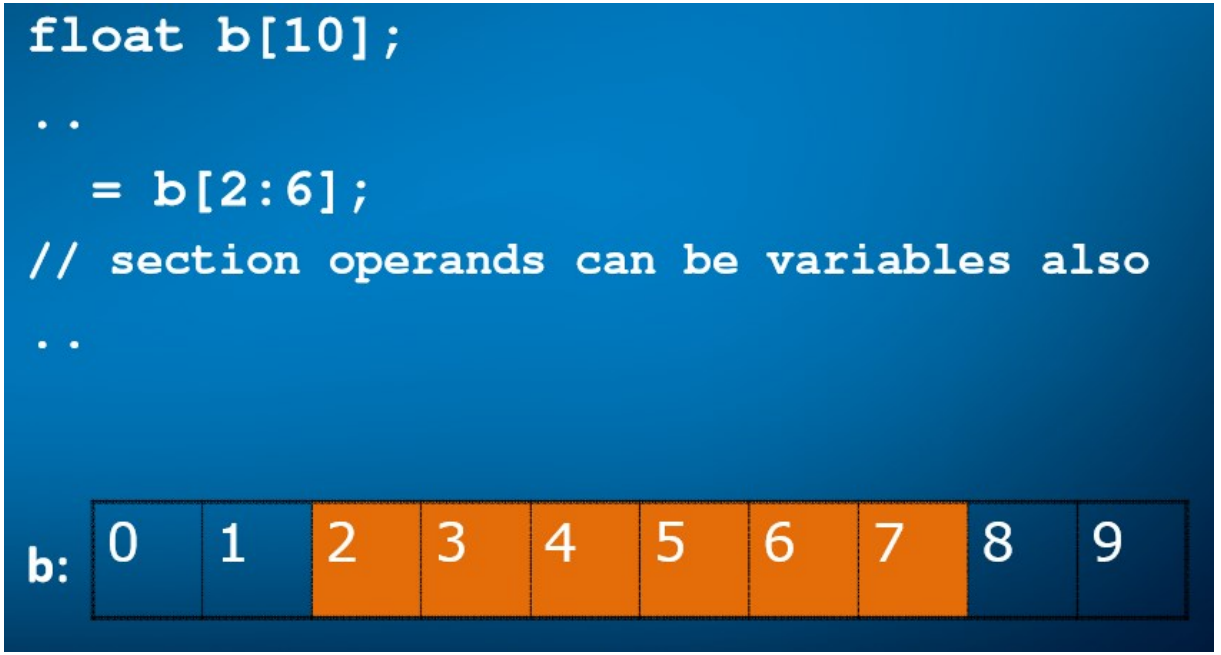
Использование директивы SIMD

```
double pi(int count)
{
    int i;
    double pi = 0.0;
    double t;
    #pragma simd private(t) reduction(+:pi)
    for (i=0; i<count; i++) {
        t = (double)((i+0.5)/count);
        pi += 4.0/(1.0+t*t);
    }
    pi /= count;
    return pi;
}
```

Технология Array Notation…

- С помощью выражения ***A[:]*** задается весь массив A (размер массива определяется на этапе компиляции, а значит должен быть константным)
- Выражение ***A[start_index : length]*** задает отрезок массива, начиная со *start_index* длиной *length*

```
float b[10];
..
  = b[2:6];
// section operands can be variables also
..
```



Технология Array Notation…

- Выражение ***A[start_index : length : stride]*** говорит о том, что мы хотим использовать каждый *stride* элемент массива, начиная со *start_index*. Количество таких элементов должно быть равно *length*

```
float d[10];
..
  = d[0:3:2];
..
```

- Поддерживаются многомерные массивы

Технология Array Notation: поддерживаемые операции…

- Возможно использование операторов языков C/C++:

```
d[:] = a[:] + (b[:]*c[:]);
```

- Возможна передача массивов в качестве аргументов функции. При этом вызов функции осуществляется для каждого заданного элемента массива:

```
b[:] = func(a[:]);
```

- Поддерживается операция редукции для сложения, минимума, максимума и т.п.:

```
sum = __sec_reduce_add(a[:]);
```

Технология Array Notation: поддерживаемые операции…

- Поддерживаются условные операторы if-then-else:

```
if (mask[:])
{
    a[:] = b[:];
}
```

- Поддерживаются операции типа scatter/gather, с помощью которых можно собрать определенные элементы одного массива в другой (собрать разрозненные элементы в один непрерывный массив), и наоборот:

```
c[:] = a[b[:]]; //gather
a[b[:]] = c[:]; //scatter
```

Технология Array Notation: поддерживаемые операции

- Поддерживаются операции сдвига. Операция ***shift*** сдвигает элементы массива на *shift_val* позиций влево/вправо, освободившиеся элементы заполняются значением *fill_val*. Операция ***rotate*** обеспечивает циклический сдвиг элементов влево/вправо на ***rotate_val*** позиций. Результат работы этих функций записывается в новый массив:

```
b[:] = __sec_shift_right(a[:], shift_val, fill_val);
b[:] = __sec_shift_left(a[:], shift_val, fill_val);
b[:] = __sec_rotate_right(a[:], rotate_val);
b[:] = __sec_rotate_left(a[:], rotate_val);
```

Технология Array Notation: принципы работы

- Размер массива должен быть известен на стадии компиляции. Если используется динамический массив, то при использовании данной нотации необходимо явно указывать начальную позицию (*start_index*) и длину (*length*) отрезка массива.
- В случае если векторизация кода невозможна, будет сгенерирован обычный цикл.
- Оптимальный векторный код будет получен только в случае работы с выровненными данными.
- Требуется соответствие рангов и длин массивов в рамках одной операции.

Технология Array Notation: скалярное произведение

- Скалярный код:

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

    return dp;
}
```

- Векторный код:

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

Технология Array Notation: динамические массивы

```
#include <cilk\cilk.h>

void saxpy_vec(int m, float a, float x[m], float y[m])
{
    y[0:m] += a*x[0:m];
}

void main(void)
{
    int n = 2048;
    const int m = 256;
    float* a = new float[n];
    float* b = new float[n];
    cilk_for (int i = 0; i < n; i += m)
    {
        saxpy_vec(m, 2.0, &a[i], &b[i]);
    }
}
```

Элементарные функции…

- Элементарная функция – это функция, выполняющая вычисления над скалярными элементами данных.
- Строка **__declspec(vector)** указывает на необходимость векторизации кода этой функции:

```
__declspec(vector) float foo(float *B, float *C, int i)
{
    return B[i] + C[i];
}
…
for(i=0; i<N; i++)
{
    A[i] = foo(B, C, i);
}
…
```

Элементарные функции: способы вызова

- В цикле для элементов массива:

```
for (i=0; i<n; i++)
{
    A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

- С использованием технологии Array Notation:

```
A[:] = foo(B[:], C[:], D[:], E[:]);
```

- Из другой элементарной функции:

```
__declspec(vector) float bar(float a, float b,
    float c, float d)
{
    return sinf(foo(a,b,c,d));
}
```

- В скалярном коде:

```
e = foo(a, b, c, d);
```

Элементарные функции: ограничения

- Непрямые вызовы запрещены;
- Запрещена передача структур по значению (по ссылке допустимо);
- Запрещена синхронизация;
- Запрещено использование многопоточных конструкций (_Cilk_spawn/_Cilk_for).

Использование отчетов компилятора

- Опция компилятора, отвечающая за отчеты по векторизации, имеет вид ***-vec-report[n]*** для Linux и ***/Qvec-report[n]*** для Windows версии компилятора.
- *–vec-report3:*

```
loop was not vectorized: unsupported data type
```

- *–vec-report6:*

```
vectorization support: type TTT is not supported for operation OOO
```

- *–vec-report7* позволяет получить дополнительную информацию о векторизуемом коде, например, ожидаемое ускорение от векторизации, используемый здесь шаблон доступа к памяти и др.

Выравнивание данных…

- При работе с Intel Xeon Phi необходимо выравнивать данные по границе в 64 Б.
- Выравнивание статических массивов:

```
__declspec(align(64)) float A[1000];          //Windows
float A[1000] __attribute__((aligned(64)));   //Linux, Mac
```

- Выравнивание динамических массивов:

```
buf = (char*) _mm_malloc(bufsize, 64);
```

- Помимо выделения выровненных данных в программе, для эффектной векторизации вашего кода необходимо сообщить компилятору о выравнивании в том месте кода, где эти данные непосредственно используются.

Выравнивание данных…

- Допустим, используется цикл, который обращается к массиву **A** как **A[i]**, и к массиву **B** как **B[i+n1]**. Здесь **i** – это счетчик цикла.
- Для того чтобы компилятор использовал команды работы с выровненными данными, ему необходимо сообщить:
  - Адрес начала массивов **A** и **B** кратен 64 байтам. Используется конструкция ***__assume_aligned(A, 64)***. В случае если массивы выделены статически, ничего дополнительно делать не надо.
  - Величина **n1** кратна 16 (при размере типа данных в 4 байта). Эта информация может быть указана с помощью конструкции ***__assume(n1%16==0)***.

Выравнивание данных…

```
__declspec(align(64)) float X[1000], X2[1000];

void foo(float * restrict a, int n, int n1, int n2) {
    int i;
    __assume_aligned(a, 64);
    __assume(n1%16==0);
    __assume(n2%16==0);

    for(i=0;i<n;i++) {
        X[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2]
            + a[i-n2];
    }

    for(i=0;i<n;i++) {
        X2[i] += X[i]*a[i];
    }
}
```

Выравнивание данных…

- В качестве альтернативы предложенным конструкциям можно использовать директиву ***#pragma vector align*** перед телом векторизуемого цикла:

```
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] + a[i];
}
```

- **Обратите внимание**, что данная директива относится ко всем массивам, используемым в рамках цикла.

Выравнивание данных: использование параллелизма…

- Вариант кода без векторизации:

```
static double * a;
a = _mm_malloc((N+OFFSET) * sizeof(double),64);

#pragma omp parallel for
for (j = 0; j < N; j++)
    a[j] = 2.0E0 * a[j];
```

Выравнивание данных: использование параллелизма

- С использованием векторизации:

```
static double * a;
a = _mm_malloc((N+OFFSET) * sizeof(double),64);

#pragma omp parallel
{
    #pragma omp master
    {
        num_threads = omp_get_num_threads();
        N1 = ((N / num_threads)/8) * num_threads * 8;
    }
}

#pragma omp parallel for
#pragma vector aligned
for (j = 0; j < N1; j++)
    a[j] = 2.0E0 * a[j];

for (j = N1; j < N; j++)
    a[j] = 2.0E0 * a[j];
```

Векторизация внешних циклов…

- По умолчанию компилятор пытается векторизовать только внутренние циклы.
- Если число итераций внутренних циклов
- Способы векторизации внешних циклов:
  - Посредством добавления директивы *#pragma simd* перед внешним циклом.
  - С помощью директивы *#pragma simd* и элементарных функций.
  - С использованием технологии Array Notation.

Векторизация внешних циклов…

- Исходный код:

```
for(LocalOrdinalType row=ib*BLOCKSIZE; row<top; ++row) {
    local_y[row]=0.0;

    for(LocalOrdinalType i=Arowoffsets[row];
            i<Arowoffsets[row+1]; ++i) {
        local_y[row] += Acoefs[i]*local_x[Acols[i]];
    }
}
```

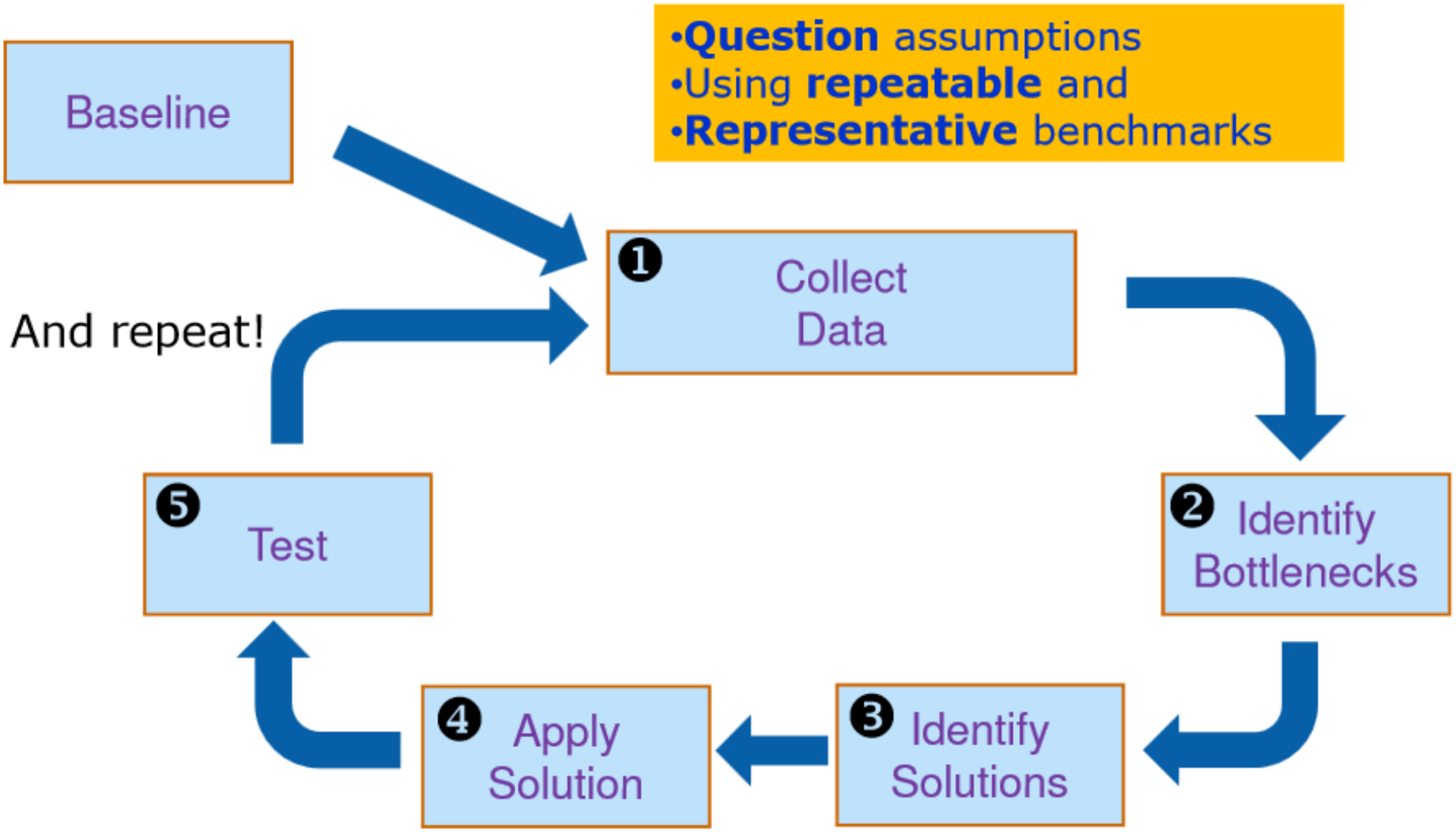
Векторизация внешних циклов

- Код с векторизацией:

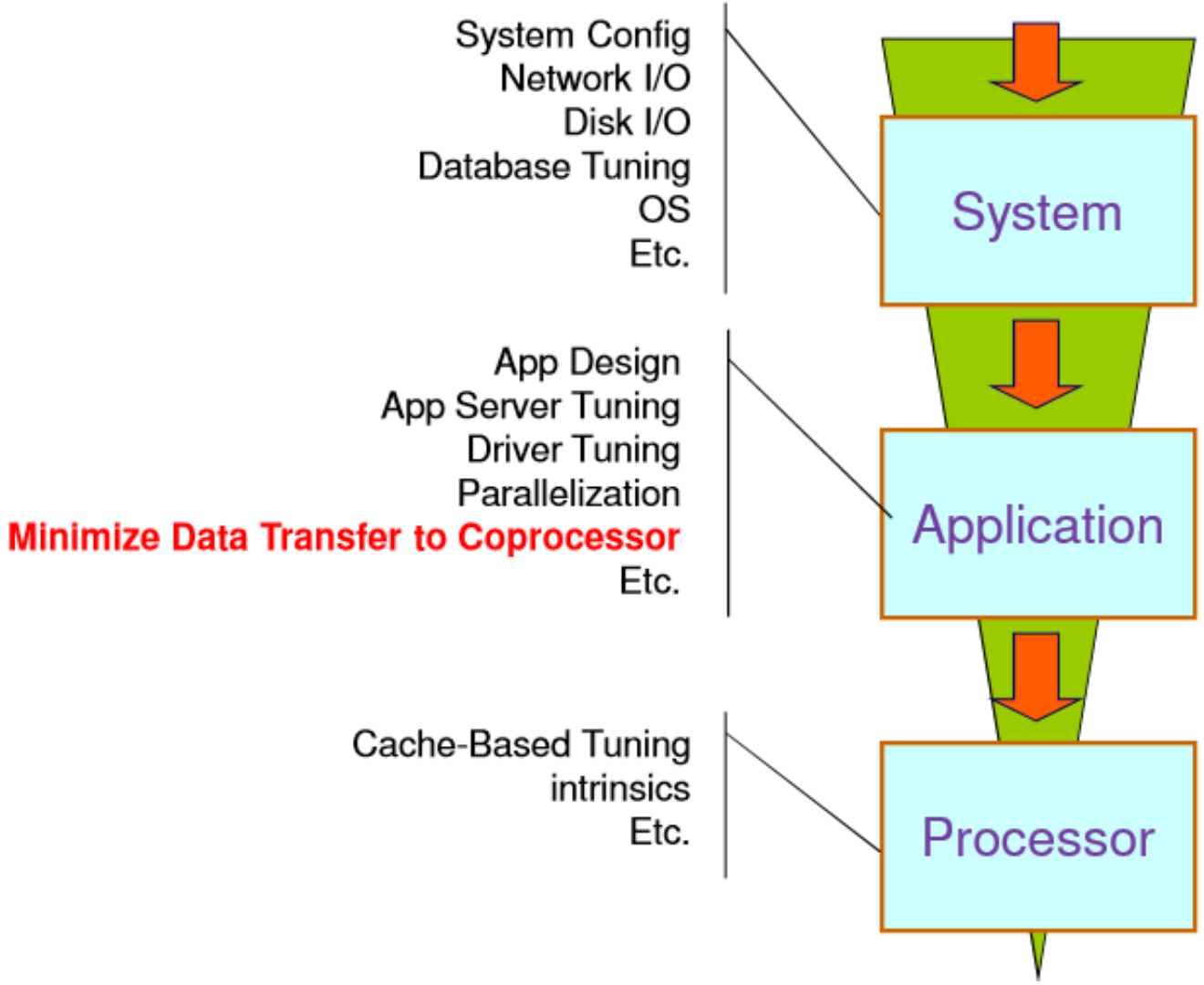
```
#pragma simd
for(LocalOrdinalType row=ib*BLOCKSIZE; row<top; ++row) {
    local_y[row]=0.0;
    Inner_loop_elem_function(local_y, row, Acoefs, local_x,
        Acols, Arowoffsets);
}

__declspec(vector(uniform(Arowoffsets, Acoefs, local_x,
        Acols, local_y), linear(row))))
    Inner_loop_elem_function(float *local_y, int row,
        float *Acoefs, float *local_x, int *Acols,
        int *Arowoffsets)
{
    for(LocalOrdinalType i=Arowoffsets[row];
            i<Arowoffsets[row+1]; ++i) {
        local_y[row] += Acoefs[i]*local_x[Acols[i]];
    }
}
```

4. Подходы к оптимизации прикладных программ для Intel Xeon Phi

Итеративный процесс оптимизации



Уровни оптимизации приложений



Использование встроенного профилировщика циклов…

- Для профилировки приложения необходимо включить инструментацию кода в процессе компиляции:

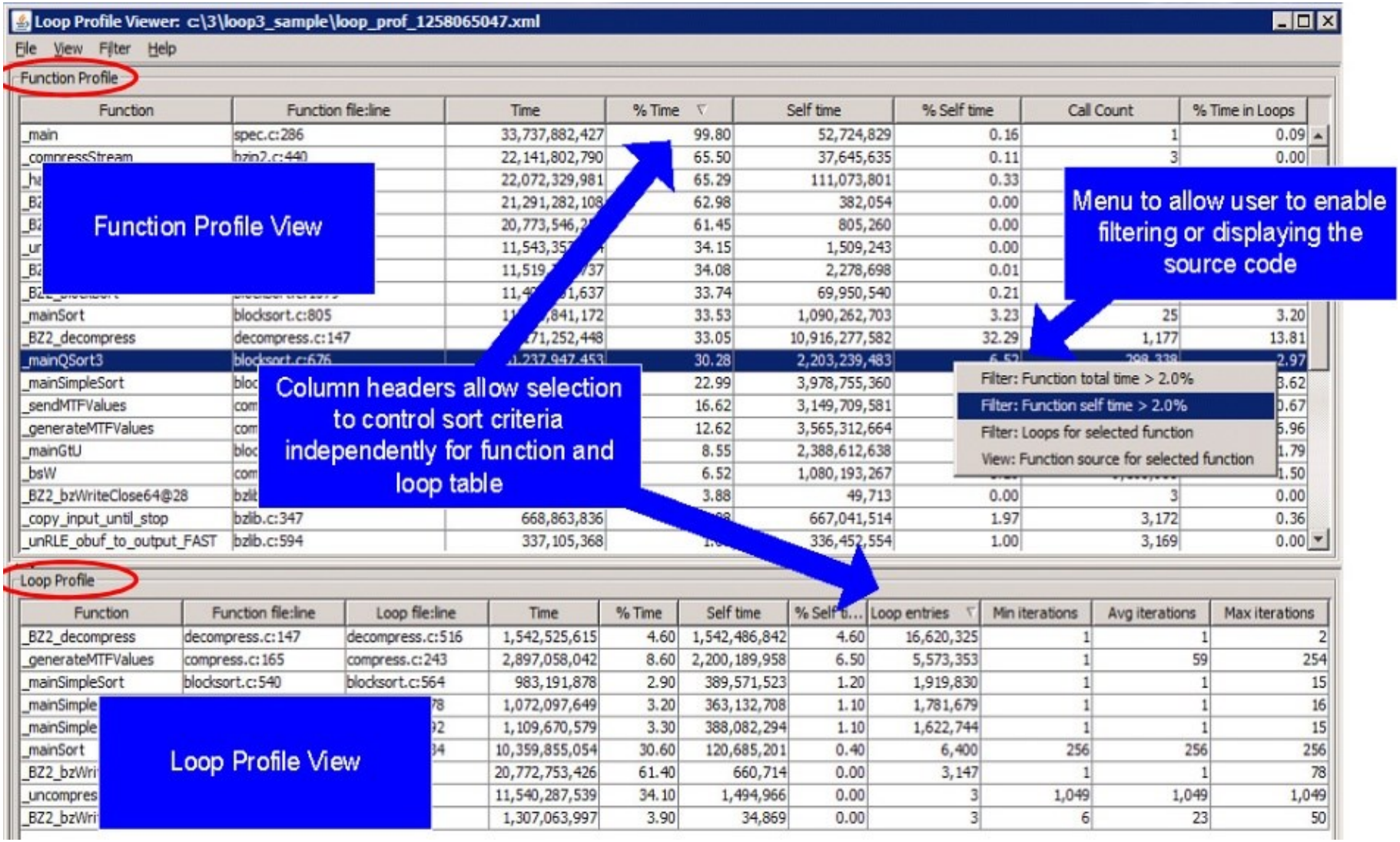
```
icc -O1 -profile-functions -profile-loops=all
-profile-loops-report=2 …
```

- Далее нужно запустить приложение на интересующем вас тесте.
- Программа соберет статистику о своей работе и запишет ее в формате таблицы (понятном для пользователя) и в формате xml (который можно проанализировать с использованием специального GUI приложения – Loop Profile Viewer).

Использование встроенного профилировщика циклов…

- Файлы содержат такую информацию, как:
  - имена и количество вызовов функций, количество циклов в них;
  - общее время работы функций и циклов;
  - время работы функций и циклов без учета внутренних вызовов;
  - среднее, минимальное и максимальное количество итераций циклов.

Использование встроенного профилировщика циклов



Использование отчетов компилятора

- Отчет о векторизации: **-vec-report[n]**
- Руководство по векторизации: **-guide-vec[=n]**
  - Данная опция позволяет получить рекомендации по изменению кода, которые позволят компилятору найти больше возможностей для векторизации.
- Руководство по дополнительной оптимизации: **-guide**
  - Позволяет получить ряд советов касательно модификации вашего кода с тем, чтобы компилятор мог лучше его оптимизировать.
- Отчет об оптимизации: **-opt-report [n]**
  - Данная опция информирует программиста о том, как компилятор модифицирует ваш код, пытаясь сгенерировать наиболее оптимальную его версию.

Балансировка нагрузки…

- Сопроцессор Intel Xeon Phi позволяет запускать одновременно 4 логических потока на ядро. Однако часто бывает эффективнее запускать меньшее их количество, т.к.:
  - это позволяет минимизировать нагрузку на кэши разных уровней (L1, L2, TLB), т.к. если потоков на ядре много, они начинают соперничать за доступ в кэш;
  - меньше соревнований между потоками за единственный векторный модуль ядра;
  - уменьшаются запросы к основной памяти.

Балансировка нагрузки…

- Существуют доводы и за использование 4-х потоков на ядро:
  - можно получить преимущества от локальности данных при обращении в основную память и в кэш (когда все потоки используют одни и те же данные, которые удается загрузить в кэш);
  - хорошо подходит для задач, требующих больших вычислительных ресурсом и не требующих много памяти.

Балансировка нагрузки…

- В силу специфики архитектуры ядра сопроцессора ***при использовании 1 потока на ядро сопроцессор будет простаивать минимум половину времени***, соответственно в большинстве случаем использование 2-х потоков на ядро будет эффективнее, нежели использование 1-го потока;
- Для большинства приложений подойдет использование 3 потоков на ядро (экспериментальные данные специалистов компании Intel);
- Для приложений с хорошей локальностью данных и большими требованиями к вычислительным ресурсам лучше использовать 4 потока на ядро.

Балансировка нагрузки

- KMP_AFFINITY=”compact”
  - В этом случае на ядро будет приходиться максимально возможное число потоков, часть ядер будет свободна. Подходит приложениям с хорошей локальностью данных. Для остальных приложений может вызвать снижение производительности.
- KMP_AFFINITY=”scatter”
  - Потоки равномерно распределяются по ядрам. Подходит для максимального использования системных ресурсов.
- KMP_AFFINITY=”balanced”
  - Потоки равномерно распределяются по ядрам, но с условием, что на каждом ядре лежат потоки с соседними номерами. Это позволяет использовать локальность данных приложения вместе с эффективным использованием системных ресурсов.

Дополнительные рекомендации…

- Следует позаботиться о выравнивании данных в памяти (полезно не только в случае векторизации).
- Следует использовать такие структуры данных, которые лучше всего соотносятся с шаблонами доступа к ним во время вычислений. Например, часто лучше использовать структуры массивов вместо привычных в языках C/C++ массивов структур. Применение правильных структур данных позволяет существенно повысить эффективность работы с кэш памятью.
- Следует обратить внимание на эффективное использование кэшей L1 и L2, т.к. стоимость доступа в память более чем в 10 раз медленнее, чем в кэш L2.

Дополнительные рекомендации

- Сопроцессор Intel Xeon Phi имеет аппаратное устройство предвыборки. Компилятор также генерирует указания для предвыборки автоматически. Однако для некоторых задач с нерегулярным доступом к данным лучше делать предвыборку самостоятельно. Для этого применяется функция _mm_prefetch(char* addr, int hint). Заметим, что предвыборку лучше не делать для L1 кэша, т.к. обычно это не эффективно.
- Следует использовать страницы памяти размеров 2 МБ в приложениях с большими структурами и частыми обращениями к памяти для минимизации количества промахов TLB кэша.

Литература

- Intel Corporation. Beginning Intel Xeon Phi Coprocessor Workshop, Offload Compilation, September 2012.
- Fourestey G. Intel Xeon Phi Programming Models [https://hpcforge.org/plugins/mediawiki/wiki/pracewp8/images/6/68/XeonPhi.pdf]
- Intel Corporation. Intel® C++ Compiler XE 13.1 User and Reference Guides: User-mandated or SIMD Vectorization [http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-42986DEF-8710-453A-9DAC-2086EE55F1F5.htm]

Литература

- Intel Corporation. Intel® C++ Compiler XE 13.1 User and Reference Guides: SIMD [http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/cref_cls/common/cppref_pragma_simd.htm]
- Intel Corporation. Advanced Intel Xeon Phi Coprocessor Workshop, Extracting Vector Performance with Intel Compilers, September 2012.
- Green R.W. Effective Use of the Intel Compiler's Offload Features: [http://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features]

Литература

- Green R.W. Overview of Vectorization Reports and new vec-report6: [http://software.intel.com/en-us/articles/overview-of-vectorization-reports-and-new-vec-report6]
- Krishnai R. Data Alignment to Assist Vectorization: [http://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization]
- Green R.W. Outer Loop Vectorization via Intel Cilk Plus Array Notations: [http://software.intel.com/en-us/articles/outer-loop-vectorization-via-intel-cilk-plus-array-notations]
- Green R.W. Outer Loop Vectorization: [http://software.intel.com/en-us/articles/outer-loop-vectorization]

Литература

- Intel Corporation. Beginning Intel Xeon Phi Coprocessor Workshop, Optimization, September 2012.
- Intel Corporation. A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel® Xeon® processors and Intel® Xeon Phi™ product family coprocessors: [http://software.intel.com/en-us/articles/a-case-study-comparing-aos-arrays-of-structures-and-soa-structures-of-arrays-data-layouts]
- J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. -Morgan Kaufmann, 2013. -432 p.

Литература

- Руководство по использованию Intel® Cilk Plus: [http://software.intel.com/sites/products/evaluation-guides/docs/cilk-plus-evaluation-guide.pdf]
- Спецификация синтаксических конструкций Intel® Cilk Plus: [http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf]
- Intel Developer Zone [http://software.intel.com/en-us/mic-developer]

Авторский коллектив

- Горшков Антон Валерьевич,
  ассистент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.
  anton.v.gorshkov@gmail.com