



**Нижегородский государственный университет
им. Н.И.Лобачевского**

Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

Лабораторная работа №2

**Оптимизация прикладных программ для Intel Xeon
Phi с использованием Intel C/C++ Compiler.**

Векторизация

При поддержке компании Intel

Бастраков С.И.

Кафедра математического обеспечения ЭВМ

Содержание

- ❑ Векторизация циклов компилятором. Использование директив
 - Использование ключевого слова `restrict`
 - Использование `#pragma ivdep`
 - Использование `#pragma simd`
- ❑ Использование `Array notation`
- ❑ Использование элементарных функций
- ❑ Векторизация циклов с вызовами математических функций
- ❑ Более сложные примеры векторизации
- ❑ Дополнительные задания и литература



Цели

- ❑ Изучение средств векторизации кода на Intel Xeon Phi.
- ❑ Освоение средств диагностики и различных механизмов векторизации на простом примере.
- ❑ Изучение способов векторизации вызовов математических функций.
- ❑ Рассмотрение более сложных случаев векторизации, возникающих в прикладной задаче.



Пример

□ Рассмотрим следующий код:

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

□ Итерации цикла независимы? Векторизация возможна?



Пример

```
int main() {  
    int n = 10000;  
    float* a = new float[n]; float* b = new float[n];  
    float* c = new float[n]; float* d = new float[n];  
    for (int i = 0; i < n; ++i)  
        a[i] = b[i] = c[i] = d[i] = (float)i;  
    #pragma noline  
    vectorization_simple(a, b, c, d, n);  
    delete[] a;    delete[] b;  
    delete[] c;    delete[] d;  
    return 0;  
}
```



Диагностика векторизации

- ❑ В компиляторе Intel есть возможность вывода отчета о векторизации.
- ❑ Компиляция с ключом **-vec-report[n]**
 - Вместо [n] число, определяющее степень подробности отчета: чем больше число, тем более подробный отчет.
 - В данной лабораторной работе будет использоваться 3-й уровень подробности, обычно дающий достаточное количество информации (**-vec-report3**).
- ❑ Строка компиляции с выводом отчета в файл report.txt:

```
icc -O2 *.cpp -mmic -o vectorization_simple  
-vec-report3 &> report.txt
```



Отчет о векторизации

main.cpp(10): (col. 5) remark: LOOP WAS VECTORIZED

main.cpp(10): (col. 5) remark: PEEL LOOP WAS VECTORIZED

main.cpp(10): (col. 5) remark: REMAINDER LOOP WAS VECTORIZED

vectorization_simple.cpp(5): (col. 5) remark: loop was not vectorized: existence of vector dependence

vectorization_simple.cpp(8): (col. 1) remark: vector dependence: assumed FLOW dependence between c line 8 and b line 7

...

vectorization_simple.cpp(8): (col. 1) remark: vector dependence: assumed OUTPUT dependence between c line 8 and a line 7



Выводы из отчета о векторизации

- ❑ Был векторизован только цикл с инициализацией массивов (**LOOP WAS VECTORIZED**).
- ❑ Основной цикл в функции **vectorization_simple** не был векторизован из-за наличия зависимости по данным между итерациями (**remark: loop was not vectorized: existence of vector dependence**).
- ❑ В отчете о векторизации приведено перечисление обнаруженных зависимостей.



Анализ возможности векторизации

- ❑ В чем причина обнаруженных компилятором зависимостей?
- ❑ На первый взгляд результат кажется неожиданным:
 - цикл является очень простым,
 - итерации «очевидно» независимы,
 - тем не менее, векторизация не была произведена компилятором.
- ❑ Удивление может вызвать и огромный список обнаруженных зависимостей в отчете о векторизации.
- ❑ Разберемся, в чем причина обнаруженных зависимостей и действительно ли они существуют.



Анализ возможности векторизации

- ❑ Компилятор не имеет права произвести некорректную векторизацию цикла.
- ❑ При векторизации делаются лишь преобразования, для которых доказана эквивалентность.
- ❑ В связи с этим компилятор вынужден быть очень консервативным и любая возможная зависимость является препятствием для векторизации.
- ❑ В рассматриваемой функции нет информации о том, как соотносятся указатели **a**, **b**, **c**, **d**.
- ❑ Например, если **b[1]** и **a[0]** расположены по одному и тому же адресу в памяти, то при векторном исполнении итераций 0 и 1 результат будет некорректен (отличаться от результата не векторизованного исполнения цикла).



Анализ возможности векторизации

- ❑ В то же время программист может обладать информацией о том, что массивы **a**, **b**, **c**, **d** никогда не пересекаются.
- ❑ В таком случае обнаруженная компилятором зависимость потенциально возможна, но на самом деле никогда не осуществляется.
- ❑ Существует несколько возможностей предоставить компилятору дополнительную информацию или гарантии отсутствия зависимости и, таким образом, способствовать векторизации.
- ❑ Рассмотрим данные возможности.





Векторизация циклов компилятором. Использование директив



Использование ключевого слова `restrict`

- Одним из способов предоставления гарантии того, что указатели не пересекаются, является использование ключевого слова **`restrict`** (добавлено в стандарте C99).
- Функция с использованием `restrict` имеет следующий вид:

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



Использование ключевого слова `restrict`

- ❑ Для компиляции с поддержкой ключевого слова `restrict` необходимо добавить ключ **`-restrict`**.
- ❑ В этом случае компилятор имеет гарантию отсутствия зависимостей, которые ранее препятствовали векторизации.
- ❑ Отчет о векторизации для данной версии подтверждает, что векторизация была успешно произведена:

```
vectorization_restrict.cpp(5): (col. 5) remark: LOOP WAS  
VECTORIZED
```

```
vectorization_restrict.cpp(5): (col. 5) remark: PEEL LOOP WAS  
VECTORIZED
```

```
vectorization_restrict.cpp(5): (col. 5) remark: REMAINDER  
LOOP WAS VECTORIZED
```



Использование `#pragma ivdep`

- Использование ключевого слова **restrict** дает гарантию, что данные указатели не могут пересекаться.
- Другим способом предоставления компилятору дополнительной информации является директива компилятора **#pragma ivdep** (`ivdep` – сокращение от `ignore vector dependencies`).
- Ее использование изменяет способ принятия решения о возможности векторизации данного цикла:
 - компилятор по-прежнему анализирует все возможные зависимости между данными,
 - недоказанные зависимости не принимаются во внимание (считаются несуществующими).



Использование #pragma ivdep

- Если существование зависимости доказано, то векторизация произведена не будет, но все недоказанные зависимости более не будут препятствием для векторизации.
- Функция с #pragma ivdep:

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



Использование `#pragma ivdep`

- Векторизация цикла производится успешно, что подтверждается отчетом:

```
vectorization_ivdep.cpp(6): (col. 5) remark: LOOP WAS  
VECTORIZED
```

```
vectorization_ivdep.cpp(6): (col. 5) remark: PEEL LOOP WAS  
VECTORIZED
```

```
vectorization_ivdep.cpp(6): (col. 5) remark: REMAINDER LOOP  
WAS VECTORIZED
```



Использование `#pragma simd`

- ❑ Директива `#pragma ivdep` является относительно традиционным способом помощи компилятору при векторизации.
- ❑ Его суть состоит в предоставлении дополнительной информации, на основе которой компилятор может принять более качественное решение о векторизации.
- ❑ В некотором смысле, это является косвенным, хотя и весьма мощным, средством векторизации.
- ❑ `#pragma simd` представляет другой, новый механизм векторизации. Он дает компилятору явное указание произвести векторизацию данного цикла.



Использование `#pragma simd`

- Использование для рассматриваемой функции:

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

- Цикл успешно векторизуется.



Использование Array notation



Array notation

- ❑ Иногда специфика алгоритма позволяет в явном виде записать его как последовательность операций над векторами.
- ❑ В этом случае использование расширения Intel Cilk Plus под названием **Array notation** позволяет получить высокую производительность и, в частности, автоматически векторизовать код.
- ❑ Расширение **Array notation** позволяет записывать выражения и операции, исполняемые над несколькими элементами вектора, без написания цикла.



Array notation. Операция :

- ❑ Вводится операция :, при ее использовании в индексе соответствующая операция применяется ко всем элементам указанного диапазона.
- ❑ В общем случае выражения имеют вид:
A[start_index : length], где первый параметр обозначает начальный индекс, а второй – количество элементов.
- ❑ Если оба параметра опущены, подразумевается весь массив.
- ❑ Пример: сложение векторов

```
float A[100], B[100], C[100];
```

```
A[:] = B[:] + C[:];
```



Использование Array notation

```
#include <cilk\cilk.h>
void vectorization_array(float* a, float* b,
                        float* c, float* d, int n)
{
    a[0:n] = b[0:n] * c[0:n];
    c[0:n] = a[0:n] + b[0:n] - d[0:n];
}
```

- Данный код будет автоматически векторизован.
- Использование Array notation может приводить к повышению производительности и из-за лучшей оптимизации кода.



Использование элементарных функций



Элементарные функции

- ❑ Одним из требований для автоматической векторизации кода является отсутствие вызовов функций внутри векторизованного кода (не считая функций, встроенных компилятором).
- ❑ В некоторых ситуациях не получается обеспечить автоматическое встраивание, например, из-за сложной структуры или большого объема функции.
- ❑ Ручное встраивание сложных функций часто нежелательно с точки зрения качества кода и удобства поддержки.
- ❑ В этом случае для векторизации кода можно использовать механизм **элементарных функций (Elemental functions)** из Intel Cilk Plus.



Использование элементарных функций

- ❑ Векторизируемая операция выносится в функцию со специальной нотацией `__declspec(vector)`.
- ❑ В теле функции выполняются операции над одним элементом данных, а сама функция может вызываться для нескольких элементов данных одновременно при векторном исполнении цикла.



Использование элементарных функций

- В рассматриваемом примере создадим элементарную функцию, выполняющую одну итерацию цикла:

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

```
__declspec(vector) void f(float* a, float* b,  
    float* c, float* d, int n, int i) {  
    a[i] = b[i] * c[i];  
    c[i] = a[i] + b[i] - d[i];  
}
```

```
void vectorization_elemental(float* a, float* b,  
    float* c, float* d, int n) {  
    for (int i = 0; i < n; i++)  
        f(a, b, c, d, n, i);  
}
```



Векторизация циклов с вызовами математических функций



Пример

- Рассмотрим цикл с вычислением экспоненты:

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

- Отчет о векторизации показывает, что цикл векторизуется.
- Однако в процессоре нет векторных команд для вычисления экспоненты (в отличие от векторных команд для сложения или умножения).
- Тогда каким образом была произведена векторизация?



SVML

- ❑ Компиляторы Intel содержат библиотеку реализаций математических функций для короткого вектора аргументов **SVML** (short vector math library).
- ❑ В случае векторизации цикла компилятор вставляет вызовы функций SVML.
- ❑ Если по каким-то причинам цикл не векторизуется, то вставляются вызовы обычных реализаций из скалярной библиотеки математических функций **LibM**.



- При большом количестве итераций цикла может иметь смысл предварительно вычислить значения математических функций сразу для всех итераций цикла (если их аргументы известны заранее).
- Для такого случая идеально подходит библиотека **VML** (vector math library), являющаяся частью Intel MKL (math kernel library).

Использование VML

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



Сравнение версий с VML и SVMML

- ❑ Оба варианта векторизуют и вычисление экспоненты, и цикл со сложением, поэтому являются достаточно эффективными и существенно превосходят не векторизованную версию.
- ❑ То, какая из них является более эффективной, зависит от количества итераций цикла n .
- ❑ В таких случаях наблюдается следующее поведение:
 - при малом количестве итераций цикла порядка десятков или сотен, предпочтительна версия с использованием SVMML,
 - при значительном количестве итераций цикла предпочтительна версия с использованием VML.



Более сложные примеры векторизации



Задача

- При вычислении справедливой цены опциона Европейского типа в популярной двухфакторной модели НЖМ возникает код следующего вида:

```
double* arr = (double*)
    malloc(size*sizeof(double));
arr[0] = 1.1;
for (i = 0; i < size - 1; i++)
{
    double c0 = sigma1*pow(arr[i], alpha);
    double c1 = sigma2*pow(arr[i], beta);
    arr[i+1] = (c0*z1[i] + c1*z2[i]) + 1;
}
```



Анализ возможностей векторизации

- ❑ Очевидно, все элементы массива **arr** зависимы между собой и векторизация цикла **for** невозможна.
- ❑ Однако в данном случае возможна векторизация вычисления **row** внутри цикла.
- ❑ Так как компилятор применяет автоматическую векторизацию лишь для циклов, необходимо трансформировать две скалярные операции в цикл из двух итераций.
- ❑ Кроме того, необходимо использовать **#pragma vector always**, в противном случае цикл, вероятно, не будет векторизован, так как компилятор сочтет его слишком коротким.



Векторизация операций pow

```
double* arr = (double*)
    malloc(size*sizeof(double));
arr[0] = 1.1;
for (i = 0; i < size - 1; i++)
{
    double c[2], sigma[2] = {sigma1, sigma2};
    double power[2] = {alpha, beta};
    #pragma vector always
    for (int k=0; k < 2; k++)
        c[k] = sigma[k] * pow(arr[i], power[k]);
    arr[i+1] = (c0*z1[i] + c1*z2[i]) + 1;
}
```



Векторизация и if

- В той же задаче далее необходимо произвести суммирование по всем путям Монте-Карло, для которых выгода положительна, и возникает код следующего вида:

```
for (int i = 0; i < size; i++)
{
    double s = coeff * exp(arr[i]);
    double payoff = s - K;
    if (payoff > 0.0)
        sum = sum + payoff;
}
```



Векторизация и if

- ❑ Хотя условные операции в целом нежелательны для векторизации, при простой структуре условия они не являются помехой.
- ❑ Данный цикл векторизуется с использованием векторного сравнения нескольких пар чисел с плавающей запятой.
- ❑ Другим возможным вариантом написания цикла была бы замена условия на явное взятие максимума:

```
for (int i = 0; i < size; i++) {  
    double s = coeff * exp(arr[i]);  
    double payoff = s - K;  
    sum = sum + std::max(payoff, 0.0);  
}
```



Использование VML

- При большой длине цикла **size** (в данной задаче это количество путей Монте-Карло и, поэтому, является весьма большим) также имеет смысл вычислять значение экспоненты для всех аргументов с использованием VML.
- При этом код приобретает вид (в предположении, что содержимое массива **arr** после завершения цикла не используется):

```
vdExp(n, arr, arr);  
for (int i = 0; i < size; i++)  
    sum = sum + std::max(coeff * arr[i] - K, 0.0);
```



Дополнительные задания и литература



Задания для самостоятельной работы

- Для рассматриваемого в разделе 2 примера сравните время работы исходной (скалярной) и векторизованной версий на центральном процессоре и сопроцессоре Intel Xeon Phi. Сравните время работы версий с использованием различных средств векторизации. Объясните полученный результат.
- Реализуйте умножение матрицы на вектор. Воспользуйтесь отчетом о векторизации и проверьте, происходит ли векторизация вашей реализации? При необходимости внесите изменения для обеспечения векторизации. Зависит ли векторизация от того, как хранится матрица: по строкам или по столбцам?



Задания для самостоятельной работы

- ❑ Проведите исследование, аналогичное предыдущему заданию, для операции вычисления матричного произведения по определению.
- ❑ Рассмотрите пример из раздела 5. Экспериментально определите минимальное количество итераций цикла, при котором предварительное вычисление всех экспонент с использованием VML становится более эффективным по сравнению с использованием SVMML.



Литература

1. Rahman R. Intel® Xeon Phi™ Coprocessor Vector Microarchitecture. URL: [<http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture>]
2. Green R. Vectorization Essentials. URL: [<http://software.intel.com/en-us/articles/vectorization-essential>]
3. Jeffers J., Reinders J. Intel Xeon Phi Coprocessor High Performance Programming // Morgan Kaufmann, 2013.



Авторский коллектив

- Бастраков Сергей Иванович,
ассистент кафедры
Математического обеспечения ЭВМ факультета ВМК ННГУ
bastrakov@vmk.unn.ru

