Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс
«Введение в принципы функционирования и применения современных мультиядерных архитектур (на примере Intel Xeon Phi)»

Лабораторная работа №2
Оптимизация прикладных программ для Intel Xeon Phi с использованием Intel C/C++ Compiler. Векторизация

Бастраков С.И.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

Введение

Данная лабораторная работа посвящена базовым вопросам векторизации программ для центральных процессоров и сопроцессоров архитектуры Intel Many Integrated Core (MIC). Под векторизацией понимается использование специализированных типов данных, регистров и инструкций, позволяющих одновременное, векторное, выполнение однотипных операций над разными данными. Такая парадигма называется SIMD – Single Instruction Multiple Data. Векторное исполнение операций является одним из видов параллельной обработки данных, в данном случае параллелизм осуществляется на уровне одного ядра центрального процессора или сопроцессора Intel Xeon Phi (а не на уровне одновременного использования нескольких ядер, как при использовании технологий параллельного программирования OpenMP и TBB).

Современные центральные процессоры Intel позволяют одновременно выполнять 4 (наборы инструкций SSE) или 8 (набор инструкций AVX) операций с одинарной точностью, а для вычислений с двойной точностью – 2 (наборы инструкций SSE) или 4 (набор инструкций AVX) операции. Сопроцессоры Intel Xeon Phi обладают 512-битными векторными регистрами и могут векторно исполнять 16 операций с одинарной точностью и 8 с двойной точностью. Полностью скалярный (не использующий векторные инструкции) способен использовать лишь малую часть вычислительных возможностей Intel Xeon Phi: 12.5% в одинарной точности и 6.25% в двойной точности. Таким образом, эффективное использование векторных инструкций является одним из ключевых аспектов достижения высокой производительности на центральных процессорах и сопроцессорах Intel Xeon Phi. Обеспечение эффективной векторизации часто является одним из наиболее трудоемких процессов при оптимизации прикладных программ.

В данной лабораторной работе рассматриваются базовые вопросы векторизации. Иллюстрируются основные способы использования векторных инструкций на сопроцессоре Intel Xeon Phi. Изложение производится на простых примерах, позволяющих показать базовые техники векторизации. В конце лабораторной работы предлагается несколько более сложных примеров. При этом не ставится цель всестороннего рассмотрения всех или некоторых средств векторизации или анализа сложных случаев. Напротив, упор сделан на обзорном рассмотрении различных средств на простых примерах и представлении многообразия доступных в настоящее время средств векторизации и общей идеологии их использования. Соответствующий теоретически материал содержится в лекции 4, некоторые более «тонкие» вопросы векторизации рассматриваются в лекции 5.

Векторизация одинаково важна для всех режимов использования сопроцессора Intel Xeon Phi (offload, симметричный, только сопроцессор), во

всех примерах используется режим только сопроцессора (native). Для эффективного использования Xeon Phi важно использование параллелизма как на уровне ядер, так и на уровне векторных инструкций. Данные уровни параллелизма являются в некотором смысле независимыми, поэтому в данной лабораторной работе рассматривается лишь параллелизм, связанный с векторизацией и программы являются однопоточными. Все техники и соображения о векторизации являются полностью релевантными и для многопоточных приложений.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучение базовых техник векторизации кода на Intel Xeon Phi.

Данная цель предполагает решение следующих основных задач:

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

1.2. Структура работы

Работа построена следующим образом. Рассматривается простой пример, в котором векторизация на первый взгляд возможна, однако не производится компилятором из-за наличия потенциальных зависимостей. На данном примере демонстрируются различные техники векторизации: использование ключевого слова restrict и #pragma ivdep для гарантии отсутствия потенциальных зависимостей, явная векторизация через #pragma simd, использование Array notation и элементарных функций. Рассматриваются вопросы векторизации циклов с вызовами математических функций. В завершение работы рассматриваются более сложные примеры.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	Intel Xeon Xeon E5-2690 (2.9 GHz, 8 ядер)
Сопроцессор	Intel Xeon Phi 7110X
Память	64 GB
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик, отладчик	Intel C/C++ Compiler 13

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий:

1. Кратко напомнить студентам об основных принципах векторизации кода. Обратить внимание на важность векторизации для полного раскрытия вычислительного потенциала Intel Xeon Phi.
2. Рассказать об использовании отчета о векторизации. Рассмотреть простой пример, когда векторизация на первый взгляд возможна, но не выполняется компилятором. Объяснить, почему компилятор не может произвести векторизацию из-за потенциальных зависимостей.
3. Объяснить механизм принятия компилятором решения о возможности и целесообразности векторизации. Рассмотреть способы предоставления компилятору дополнительной информации об отсутствии зависимостей: ключевое слово restrict и #pragma ivdep.
4. Вкратце рассмотреть другие средства векторизации: #pragma simd, Array notation, элементарные функции. Объяснить их отличия от ранее рассмотренных средств.
5. Рассмотреть случай векторизации цикла с вызовом математических функций.
6. Разобрать более сложные примеры.
7. Сформулировать выводы, дать задания для самостоятельной работы.

2. Векторизация циклов компилятором. Использование директив

Одним из основных способов использования векторных инструкций в программах на C/C++ и Fortran как на центральных процессорах, так и на со-

процессорах Intel Xeon Phi, является векторизация циклов компилятором. Под векторизацией цикла понимается одновременное, векторное, исполнение нескольких итераций цикла с использованием векторных инструкций. Естественно, такое исполнение возможно не для всех циклов. Компилятор осуществляет проверку возможности и целесообразности векторизации и, в случае выполнения обоих условий, генерирует код с использованием векторных инструкций. С помощью специальных средств языка и директив компилятора программист имеет возможность предоставить дополнительную информацию или дать определенные гарантии, которые могут оказать влияние на решение компилятора о векторизации цикла.

Различные техники векторизации будем демонстрировать на следующем простом, но типичном с точки зрения векторизации, примере:

```
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];
    }
}
```

Очевидно, итерации цикла в данной функции являются независимыми и векторизация данного цикла потенциально возможна (если массивы **a**, **b**, **c**, **d** не накладываются друг на друга). Проверим, генерирует ли компилятор код с использованием векторных инструкций. Для этого создадим функцию **main**, в которой выделим память для массивов и вызовем данную функцию. Для того, чтобы предотвратить ее встраивание, реализуем функцию **vectorization_simple** в отдельном файле и используем **#pragma noinline**. Данные усилия прилагаются для имитации реальной ситуации, когда компилятор не будет встраивать сложные функции и, следовательно, не будет обладать дополнительной информацией.

```
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 noinline
    vectorization_simple(a, b, c, d, n);
```

```
    delete[] a;
    delete[] b;
    delete[] c;
    delete[] d;
    return 0;
}
```

Для проверки того, произведена ли векторизация и, при отрицательном ответе, вывода причин, в компиляторе Intel есть возможность вывода отчета о векторизации. Данный отчет выводится при компиляции с ключом **-vec-report[n]**, где вместо [n] необходимо подставить число, определяющее степень подробности отчета (чем больше число, тем более подробный генерируется отчет). В данной лабораторной работе будет использоваться 3-й уровень подробности, обычно дающий достаточное количество информации (**-vec-report3**).

Скомпилируем данную программу для Intel Xeon Phi в режиме только сопроцессора с выводом отчета о векторизации в файл report.txt:

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

Файл report.txt содержит следующую информацию (функция **vectorization_simple** была реализована в файле vectorization_simple.cpp):

```
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 VECTOR-
IZED
vectorization_simple.cpp(5): (col. 5) remark: loop was not
vectorized: existence of vector dependence
vectorization_simple.cpp(8): (col. 1) remark: vector de-
pendence: assumed FLOW dependence between c line 8 and b
line 7
…
vectorization_simple.cpp(8): (col. 1) remark: vector de-
pendence: assumed OUTPUT dependence between c line 8 and a
line 7
```

Для каждого цикла указывается имя файла с исходным кодом и номер строки в скобках. Цикл в функции **main** с инициализацией массивов был векторизован (**LOOP WAS VECTORIZED**). А цикл в функции **vectorization_simple** не был векторизован из-за наличия зависимости по данным между итерациями, что показано сообщением **remark: loop was not vectorized: existence of vector dependence**. После этого в отчете следует перечисление обнаруженных зависимостей, в приводимом тексте отчета значительная часть списка зависимостей заменена на многоточие.

На первый взгляд данный результат может показаться неожиданным: цикл является очень простым, итерации «очевидно» независимы, но, 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.

Важно иметь ввиду, что компилятор не имеет права произвести некорректную векторизацию цикла, которая привела бы к несовпадающим с невекторизованной версией результатам (не считая допустимого изменения порядка операций). Таким образом, при векторизации делаются лишь преобразования, для которых доказана эквивалентность. В связи с этим компилятор вынужден быть очень консервативным и, в отсутствие дополнительной информации, любая возможная зависимость является препятствием для векторизации. В рассматриваемой функции у компилятора 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 невекторизованного исполнения цикла).

В то же время программист может обладать информацией о том, что массивы **a**, **b**, **c**, **d** никогда не пересекаются, что и происходит в рассматриваемом примере. 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. Они рассматриваются в следующих двух подразделах.

2.1. 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];
    }
}
```

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
```

Одному циклу в данном случае соответствует три строки в отчете о векторизации в связи с тем, что в целях повышения эффективности компилятор генерирует три цикла. Основной цикл выполняет большую часть итераций и работает с выровненными данными, а два других (PEEL и REMAINDER) являются вспомогательными и при необходимости выполняют несколько начальных и конечных итераций.

2.2. 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, но недоказанные зависимости не принимаются во внимание (считаются несуществующими). Таким образом, если существование зависимости доказано, то векторизация произведена не будет, но все недоказанные зависимости более не будут препятствием для векторизации.

Рассматриваемый пример с использованием **#pragma ivdep** приобретает следующий вид:

```
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];
    }
}
```

В результате использования **#pragma ivdep** недоказанные зависимости, препятствовавшие векторизации исходной версии, игнорируются и векторизация цикла производится успешно, что подтверждается отчетом:

```
vectorization_ivdep.cpp(6): (col. 5) remark: LOOP WAS VEC-
TORIZED
vectorization_ivdep.cpp(6): (col. 5) remark: PEEL LOOP WAS
VECTORIZED
vectorization_ivdep.cpp(6): (col. 5) remark: REMAINDER LOOP
WAS VECTORIZED
```

В ряде случаев (рассматриваемый пример к ним не относится) компилятор может счесть, что векторизация возможна, но для данного цикла неэффективна. Если же программист считает, что векторизация все равно будет эффективна, можно дать соответствующую рекомендацию компилятору при помощи **#pragma vector always**. Часто **#pragma ivdep** и **#pragma vector always** используются вместе. Важно отметить, что **#pragma vector always** является лишь рекомендацией и в случае обнаружения зависимости или наличия других веских причин векторизация все равно произведена не будет.

2.3. Использование #pragma simd

Директивы **#pragma ivdep**, **#pragma vector always** и подобные директивы являются 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. Рассмотрим использование **#pragma simd** на примере:

```
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:

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

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

По сравнению с **#pragma ivdep**, **#pragma simd** дает программисту гораздо больший контроль над векторизацией и, в частности, позволяет векторизовать структурно сложный код (в том числе со вложенными циклами или сложными объектно-ориентированными конструкциями), но, с другой стороны, требует от программиста более глубокого понимания процесса векторизации.

#pragma simd имеет несколько необязательных параметров, которые позволяют осуществлять контроль над векторизацией. Например, **reduction** позволяет выполнить редукцию (например, найти сумму или максимум) и действует аналогично одноименному параметру в OpenMP. Остановимся подробнее на параметре **vectorlength**, позволяющем задать, какое количество итераций цикла будет одновременно исполняться при векторизации цикла. Желаемое количество задается в скобках, при задании нескольких значений компилятор выберет одно из них. Данная возможность удобна для векторизации циклов, в которых потенциал векторизации ограничен: малое количество соседних итераций независимы, а «более далекие» итерации зависимы.

Для иллюстрации данного эффекта рассмотрим следующий цикл:

```
for (int i = 4; i < n; i += 4)
{
    a[i] = a[i - 1] + 1;
    a[i + 1] = 2 * a[i - 1] - a[i - 2];
    a[i + 2] = a[i - 2] + a[i - 3];
    a[i + 3] = a[i - 2] - a[i - 4];
}
```

Элементы a[i], ..., a[i + 3] вычисляются независимо друг от друга и зависят лишь от элементов a[i - 4], ..., a[i - 1]. Таким образом, для данного цикла невозможно векторное исполнение более 4 итераций. Однако возможна частичная векторизация с помощью #pragma simd vectorlength:

```
#pragma simd vectorlength(4)
for (int i = 4; i < n; i += 4)
{
    a[i] = a[i - 1] + 1;
    a[i + 1] = 2 * a[i - 1] - a[i - 2];
    a[i + 2] = a[i - 2] + a[i - 3];
    a[i + 3] = a[i - 2] - a[i - 4];
}
```

Данный код будет использовать не полную длину векторных регистров и, поэтому, не является максимально эффективным. Его производительность будет ниже, чем у полностью векторизованного кода, но выше, чем у полностью скалярного кода.

3. Использование 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. Для этого 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.

Для простейшего примера сложения двух векторов запись через **Array notation** имеет следующий вид:

```
float A[100], B[100], C[100];
A[:] = B[:] + C[:];
```

Для рассматриваемого класса алгоритмов работа с **Array notation** интуитивно проста: бинарные операции выполняются поэлементно, вызов функций осуществляется для каждого элемента массива. Кроме того, поддерживаются специальные функции для выполнения редукции и других широко используемых операций. Подробнее использование **Array notation** рассматривается в лекции 5.

Вернемся к примеру из предыдущего раздела. Основной цикл имеет вид:

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

Данную операцию можно записать в векторном виде с использованием Array notation следующим образом:

```
#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.

4. Использование элементарных функций

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. Векторизуемая 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. На подобные функции накладывается ряд ограничений, они рассмотрены в лекции 5.

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);
}
```

5. Векторизация циклов с вызовами математических функций

Рассмотрим следующий пример цикла с вызовом функции для вычисления 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]);
}
```

Использование **#pragma ivdep** предотвращает проблему с потенциальными зависимостями, аналогичную возникавшей в рассмотренном ранее примере.

Отчет о векторизации показывает, что данный 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?

Ответ состоит в том, что 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). При ее использовании код примет вид:

```
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];
}
```

Оба варианта векторизуют и вычисление экспоненты, и цикл со сложением, поэтому являются достаточно эффективными и существенно превосходят невекторизованную 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 SVML, при значительном количестве итераций цикла предпочтительна версия с использованием VML. Данное соотношение зависит от используемого процессора, модели компилятора и других факторов, эмпирическая проверка

для различного числа итераций цикла оставляется читателю на самостоятельную работу.

6. Более сложные примеры векторизации

В данном разделе рассматриваются более реалистичные примеры векторизации, возникающие из прикладных задач. Для данных примеров демонстрируются сложности и подход к векторизации. Выбор и применение подходящего средства для векторизации оставляется читателю для самостоятельной работы.

Рассмотрим две типичные операции, часто встречающиеся в вычислительных процедурах финансовой математики.

При вычислении справедливой цены опциона Европейского типа в популярной двухфакторной модели HJM 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;
}
```

Очевидно, все элементы массива **arr** зависимы между собой и векторизация цикла for невозможна. Однако в данном случае возможна векторизация вычисления **pow** 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. Преобразованный код имеет вид:

```
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;
}
```

Цикл с вычислением функции **pow** будет векторизован и превратится в один вызов векторной реализации pow из SVML.

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;
}
```

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);
}
```

В данном случае цикл также векторизуется.

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);
```

7. Дополнительные задания

1. 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.
2. Реализуйте умножение матрицы на вектор. Воспользуйтесь отчетом о векторизации и проверьте, происходит ли векторизация вашей реализации? При необходимости внесите изменения для обеспечения век-

торизации. Зависит ли векторизация от того, как хранится матрица: по строкам или по столбцам?

3. Проведите исследование, аналогичное предыдущему заданию, для операции вычисления матричного произведения по определению.
4. 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 SVML.

8. Литература

8.1. Использованные источники информации

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.