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)»

Лекция №4
Векторные расширения Intel Xeon Phi

Мееров И.Б.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

Введение

Начиная с середины XX века, программирование проделало большой путь и постоянно продолжает развиваться. Вспомним не такие уж далекие времена, когда наши коллеги писали свои программы в машинных кодах, непосредственно работая со справочником, содержащим информацию о поддерживаемом в процессоре наборе команд. Сейчас уже трудно представить, что в стиле «3E; 24; 5F…» можно написать сколько-нибудь серьезную программу, а ведь когда-то это было реальностью. В 1992 году в 10 классе школы одноклассник автора данной лекции самостоятельно, без чьей-либо помощи, кроме, быть может, моральной, написал в машинных кодах среду программирования с текстовым редактором и транслятором языка BASIC для ПК «Корвет». В то же время Ваш покорный слуга и его друзья-школьники по обрывочным, перепечатанным много раз брошюрам осваивали и применяли на практике язык Ассемблер для ПК «ZX SPECTRUM», построенного на базе восьмиразрядного процессора Z80. Да, в то время тоже можно было писать на BASIC и других высокоуровневых языках, но в этом случае быстродействие программ оставляло желать лучшего, а доступ к ряду возможностей процессора был закрыт вовсе. В наши времена программирование в машинных кодах практически стало историей, а разработка программ на ассемблере имеет смысл в весьма ограниченном числе случаев. Одна из причин этого – невероятно возросший интеллект компиляторов языков высокого уровня C, C++, Fortran, ориентированных на высокопроизводительные вычисления (high performance computing, HPC). Ни на миг не отвлекаясь от главной задачи – генерации корректного кода, оптимизирующие компиляторы умудряются выполнять многочисленные эквивалентные (точно или с точностью до разрешенных отклонений, например, при изменении порядка операций с плавающей запятой) преобразования, изменяя наш код до неузнаваемости и обеспечивая производительность, часто сопоставимую с хорошей программой на ассемблере. Отметим, что оптимизирующие компиляторы (в частности, одни из лучших представителей данной группы – Intel C/C++ Comiler, Intel Fortran Compiler) умеют использовать новые наборы команд и генерировать код, ориентированный на современные центральные процессоры.

Остановимся на данном аспекте подробнее. Что значит «код, ориентированный на современные центральные процессоры»? Важный момент состоит в использовании новых команд и наборов команд, добавленных в процессор его производителем. Чтобы понять, почему это важно, обсудим возможные перспективы расширения функциональности процессора в ре-

зультате добавления новых команд. Первая мысль, которая приходит в голову, звучит так: почему бы инженерам не поместить в набор команд часто используемые в научных и инженерных приложениях, составляющих основу области HPC, вычисление математических функций (sin, cos, exp, pow, …), генератор случайных чисел и многие другие полезные возможности? К сожалению, аппаратная реализация подобных «команд» весьма сложна и требует добавления значительного числа дополнительных транзисторов, что приведет к неприемлемому увеличению размера, энергопотребления и стоимости процессора. Таким образом, любое расширение набора команд является важным событием, не остается незамеченным в индустрии и быстро «подхватывается» разработчиками компиляторов. Действительно, раз уж несмотря на все трудности создатели процессора смогли добавить дополнительные команды, имеет смысл предоставить доступ к ним широкому кругу разработчиков ПО, использующих языки программирования высокого уровня.

Данная лекция рассматривает одно из важных направлений развития наборов команд в рамках современных архитектур – векторные расширения. Что такое векторные расширения? Это дополнение в набор команд процессора, содержащее специализированные типы данных, регистры и инструкции, ориентированные на использование парадигмы SIMD – Single Instruction Multiple Data. Суть парадигмы SIMD заключается в выполнении одной командой однотипных операций над пакетом данных. При этом реализуется один из видов параллельной обработки данных, что открывает возможность существенного повышения производительности вычислительных программ. Разумеется, не во всех практических приложениях векторизация – использование векторных инструкций – позволяет добиться существенного прироста производительности. Казалось бы – все просто. Пусть мы обрабатываем числа двойной точности, имеем специальные регистры, позволяющие хранить по 4 таких числа, а также арифметические команды, позволяющие векторно выполнять операции (рис. 1).

1.25	2.25	2.0	1.0
		+	
2.0	1.15	1.25	2.0
		=	
3.25	3.4	3.25	3.0

Рис. 1. Векторное сложение

На первый взгляд, мы можем ожидать ускорение вычислений до четырех раз по сравнению с обычным (скалярным) кодом. Однако на практике этого зачастую не происходит. Во-первых, присутствуют определенные накладные расходы на упаковку данных в регистры и распаковку результатов расчета. В наилучшем случае мы можем загрузить/выгрузить данные

одной командой. Для этого необходимо, чтобы данные лежали в памяти последовательно или (для некоторых векторных расширений) с определенным шагом. В последнем случае надо учитывать и эффективность использования особенностей иерархической организации подсистемы памяти. Во-вторых, может так случиться, что есть зависимость между значениями элементов векторов и некоторых результатов расчета – зависимости по данным, что делает невозможным параллельное выполнение операций. В связи с этим в некоторых случаях мы можем наблюдать, что несмотря на наши усилия компилятор отказывается генерировать векторный код, а иногда нам удается его уговорить, но мы не получаем осязаемого выигрыша во времени.

Тем не менее, спектр алгоритмов, для которых векторизация соответствующего программного кода является одним из важных ресурсов повышения производительности, является достаточно большим. Так, для мультимедийных приложений характерно применение алгоритмов, многократно повторяющих однотипные операции над большими массивами данных. При выполнении инженерных расчетов во многих случаях удается успешно векторизовать хотя бы часть программы и получить выигрыш в производительности. Отметим также, что в современных центральных процессорах размер вектора в случае вычислений с одинарной точностью составляет 4 (наборы инструкций SSE) и 8 (набор инструкций AVX), а для вычислений с двойной точностью – 2 (наборы инструкций SSE) и 4 (набор инструкций AVX). Таким образом, избегая векторизации расчетных частей алгоритма, мы теряем до 87,5% производительности, что весьма существенно. В случае использования Xeon Phi мы можем констатировать еще больший вклад векторизации в общую производительность программы, поскольку размер вектора увеличился вдвое по сравнению с AVX. Таким образом, как на традиционных процессорах, так и, тем более, на ускорителях Xeon Phi умелое использование векторных расширений является одним из ключевых механизмов, влияющих на производительность программы.

Дальнейшее изложение материала о векторизации построено следующим образом. В данной лекции дается общее описание векторных расширений в Intel Xeon Phi и подходы к их использованию в программах на языках C/C++. В лекции №5 рассматриваются более «тонкие» особенности использования указанных расширений при написании прикладных программ на языках высокого уровня с использованием возможностей компиляторов Intel. Техника применения векторных вычислений формируется в лабораторной работе №4, на примерах иллюстрирующей некоторые типовые проблемы векторизации кода и методы их решения. В дальнейших практических работах векторизация неизменно упоминается и применяется в качестве одного из ключевых способов повышения производительности программ.

1. Векторные расширения. Краткий обзор

В настоящее время векторные расширения присутствуют в наборах команд процессоров различных производителей и архитектур. Так, в центральных процессорах Intel соответствующие наборы команд эволюционировали по пути MMX – SSE – SSE2 – SSE3 – SSE4 – AVX, преодолев путь от векторной целочисленной арифметики, организованной на специализированных 64-разрядных регистрах, и ориентированной преимущественно на обработку звука и видео, до широкого спектра арифметических операций с плавающей запятой, работающей на 256-разрядных регистрах, и операций управления работой подсистемы памяти. Рассматривая векторные расширения, необходимо упомянуть дополнение 3DNow!, реализованное в процессорах AMD, а также расширение NEON, активно развивающееся в процессорах архитектуры ARM, широко распространенных во встраиваемых системах. Рассмотрим кратко векторные расширения, реализованные в Intel Xeon Phi. Данное описание не претендует на полноту. Оно призвано составить общее впечатление о сути вопроса для лучшего понимания того, как соотносится расчетный код на C/C++ и машинный/ассемблерный коды, порождаемые компилятором. Данное понимание весьма полезно при оптимизации программ по скорости их работы. Желающие познакомиться с вопросом в большем объеме могут обратиться к источникам [1] и [3].

1.1. Типы данных и регистровый пул

В ядрах ускорителей Intel Xeon Phi сделан следующий шаг в развитии векторных расширений набора команд. Так, уже привычные для программистов MMX, SSE, AVX не поддерживаются. Однако вместо этого в каждом ядре сконструирован специальный модуль векторной обработки (VPU, vector processor unit), содержащий 32 512-разрядных регистра (zmm-регистры; отметим двукратное увеличение размера регистра по сравнению с AVX) и открывающий новые возможности по обработке данных с использованием векторных инструкций вида $a = a + b * c$ с однократным округлением (FMA, fused multiply–add), а также некоторых других возможностей.

Итак, учитывая размер векторного регистра, на каждом ядре Intel Xeon Phi мы можем одновременно производить действия над 16 32-битными целыми числами или 8 64-битными целыми числами или 16 числами с плавающей запятой одинарной точности или 8 числами с плавающей запятой двойной точности. Кроме того, поддерживаются операции с комплексными данными. Заметим, что в отличие от привычных наборов команд в векторных расширениях Intel Xeon Phi большинство операций являются тернарными – имеют 2 аргумента и один результат, что по некоторым данным (см., например, [1]), приводит к 20% росту производительности по сравне-

нию с традиционной схемой работы, основанной на двух операндах. Для FMA все три операнда являются аргументами, при этом один из них в то же время принимает результат.

1.2. Обзор основных типов операций

Поддерживаются следующие основные операции:

1. *Арифметические операции*: сложение, вычитание, умножение, деление, FMA (для вычислений с плавающей запятой).

2. *Операции преобразования типов*, позволяющие выполнять повышающие и понижающие преобразования согласно определенным правилам (см. [1, 3]).

3. *Логические операции*, позволяющие выполнять векторные сравнения, находить минимум и максимум и т.д.

4. *Операции доступа к данным*. В эту группу входят команды загрузки данных из памяти в регистр и выгрузки данных из регистра в память. В новом наборе команд поддерживаются так называемые scatter/gather-инструкции, позволяющие оперировать с данными, хранящимися в памяти с пропусками с определенным шагом. Также реализованы команды предвыборки данных, позволяющие выполнять упреждающую загрузку в кэш-память. Важно отметить, что выравнивание данных в памяти существенно влияет на производительность. Этот вопрос будет подробно рассмотрен позже. При выполнении операций доступа к данным может быть указано, что с точки зрения программиста те или иные данные не имеет смысла кешировать (nontemporal data). Для оптимизации кода с учетом этого факта при записи в память результатов вычислений, идентифицированных нами как nontemporal data, предназначены так называемые streaming stores, использование которых приводит к уменьшению накладных расходов.

Также реализованы следующие возможности:

1. При выполнении операций может применяться маскирование: при помощи специального регистра может быть указано, над какой частью регистра-операнда выполнять операцию. По умолчанию такой специальный регистр содержит все единицы, а операции выполняются над всеми операндами.

2. При выполнении операций может производиться перестановка/размножение частей операндов при помощи соответствующих модификаторов (swizzle). Аналогичные манипуляции могут выполняться, когда один из аргументов указывает на данные в оперативной памяти (broadcast). Еще большими возможностями по перестановкам обладает т.н. «смешивание» (shuffle), обладающее неограниченными возможностями по перемещению байтов в регистре.

1.3. Расширенная поддержка математических функций

В рамках Intel Xeon Phi была реализована расширенная поддержка некоторых математических функций. Так, появились команды для вычисления в одинарной точности функций 1/x, 1/sqrt(x), $\log_2(x)$, 2^x. Первые три инструкции имеют латентность 4 такта и пропускную способность 1 такт, последняя – латентность 8 тактов и пропускную способность 2 такта. Нетрудно видеть, что sqrt(x), a^x и деление могут быть вычислены при помощи указанных функций. Вопрос о векторизации циклов, содержащих вызовы математических функций, будет рассмотрен позже.

2. Векторизация в программах на языке высокого уровня

Итак, вычислительные ядра Intel Xeon Phi могут выполнять однотипные вычисления, оперируя векторами из целых чисел или чисел с плавающей запятой, обладают широким спектром математических операций, логических операций, операций с битами, операций работы с памятью. Основной вопрос, волнующий прикладного программиста, желающего применить этот аппарат на практике, формулируется так: как написать программу, чтобы задействовать эти возможности? Попробуем дать ответ на этот вопрос.

Для использования векторных инструкций есть следующие пути:

1. Использовать высокопроизводительные специализированные библиотеки, эффективно использующие векторные инструкции.

   Это хороший вариант при условии, что все, что нужно программе, реализовано в библиотеке. Такое на практике встречается достаточно редко.

2. Написать программу на C/C++ или Fortran и откомпилировать ее тем транслятором, который «знает» про соответствующие наборы команд.

   Остановимся на этом варианте подробнее. Итак, если мы рассматриваем вопрос в контексте традиционных центральных процессоров, то мы можем констатировать тот факт, что Intel C/C++ Compiler и Intel Fortran Compiler по-прежнему являются одними из лучших оптимизирующих компиляторов, однако и другие компиляторы постепенно «учатся» использовать векторные расширения SSE и AVX. В частности, это относится к компилятору gcc. Говоря по Intel Xeon Phi, необходимо отметить, что на сегодняшний день только компиляторы Intel могут генерировать код для этого ускорителя, что, в принципе, неудивительно, учитывая, что устройство появилось на рынке только в этом году. Таким,

образом, для использования векторных расширений при условии программирования исключительно с использованием обычных возможностей языка высокого уровня нам необходим компилятор Intel.

Пусть мы используем компилятор Intel. Значит ли это, что все наши вычисления по мановению волшебной палочки станут векторными? Конечно, нет. Мы должны учитывать тот факт, что компилятор умеет векторизовывать (реализовывать с использованием векторных инструкций) те вычисления, которые содержаться в циклах. При этом мы должны стремиться к соблюдению следующих условий:

- Крайне желательно минимизировать всевозможные зависимости по данным. Чем меньше зависимостей, тем больше потенциала для параллелизма. Чем больше возможностей параллельного выполнения, тем больше шансов, что компилятор распознает этот факт и внедрит соответствующие команды в код. Иногда это проще сказать, чем сделать. Как избавиться от зависимостей, если они возникают из природы метода? Часто, несмотря ни на что, можно переупорядочить и сгруппировать вычисления, чтобы найти независимые фрагменты. Иногда это потребует серьезных усилий.

- Нежелательно вызывать функции в вычислительно трудоемких циклах. Однако вызов функции тоже не приговор. Компилятор может встроить код функции и успешно векторизовать цикл.

- Необходимо следить за выравниванием данных в памяти, то есть за их размещением с «правильных» адресов, кратным определенному числу байт (размеру кэш-линии, размеру векторного регистра). Загрузка/выгрузка выровненных данных происходит значительно быстрее, кроме того, мы избавляемся от ситуаций, когда часть информации попадает в одну кэш-линию, а часть – в другую. Последнее не имеет прямого отношения к векторизации, но не становится от этого менее важным. Выравнивания можно добиться как для обычных статических массивов, так и для динамических массивов посредством директив __declspec(aligned) и функций __mm_malloc().

- Необходимо стараться обеспечить однородный доступ к памяти, когда мы загружаем/выгружаем данные, лежащие последовательно (крайне желательно) либо с одинаковым шагом (допустимо). В этих случаях загрузка/выгрузка может быть выполнена одной командой. В противном случае данные тоже могут быть собраны в векторный регистр, но это потребует нескольких команд, что приведет к большим накладным расходам и, скорее всего, нивелирует весь выигрыш от векторизации, а иногда и вовсе приведет к замедлению.

- Необходимо сводить к минимуму смешивание объектов разных типов данных в выражениях.

- Необходимо по возможности избавляться от условных операторов в теле внутреннего цикла. Раньше компилятор в принципе отказывался векторизовывать такие циклы. Сейчас в ряде случаев ему удается это сделать.

Пример цикла с вычислением максимума:

```
  #pragma simd
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    s = s + max(a[i],0);
}
```

Пример цикла (линейный поиск):

```
  #pragma simd
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    if (a[i] == key) {
      Index = i;
      break;
    }
```

Оба цикла успешно векторизуются компилятором. Подробнее данные примеры обсуждаются в лабораторной работе по векторизации.

Весь возможный список рекомендаций этим не исчерпывается, но приведенные, пожалуй, являются наиболее принципиальными. Среди других достаточно важных тезисов необходимо отметить следующие:

- по возможности не использовать объектно-ориентированные конструкции в расчетах; объектный подход – мощный инструмент для анализа, проектирования и разработки больших программных комплексов, но часто затрудняющий компиляторную оптимизацию. Тем не менее, существуют способы умелого сочетания высокоуровневой объектно-ориентированной архитектуры и низкоуровневого программирования вычислительно трудоемких участков кода;

- стремиться к достаточно существенному объему работы во внутренних циклах; заметим, что компилятор преимущественно векторизует именно внутренние циклы, поэтому необходим простор для его деятельности.

Пример кода, автоматически векторизуемого компилятором, может выглядеть следующим образом:

```
void test(float * restrict a,
          float * restrict b,
          float * restrict c,
          int n)
```

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

Обратите внимание на ключевое слово restrict. При помощи него мы говорим компилятору, что доступ в указанную память будет осуществляться только через указатель, использованный при объявлении. Это необходимо, что компилятор зафиксировал факт отсутствия зависимостей по данным между массивами a, b и c и успешно векторизовал цикл.

3. Использовать специальные ключи и директивы компилятора.

   Мы можем написать программу на C/C++ или Fortran, подать ее на вход компилятору Intel, указать ключи O2 (O3) – оптимизация по скорости, и, возможно, некоторые циклы в программе будут векторизованы. Как узнать, что именно векторизовалось, что не векторизовалось и почему? Как помочь компилятору? Эти и некоторые другие темы будут подробно рассмотрены в следующей лекции и при выполнении лабораторной работы по векторизации. Пока скажем лишь, что компилятор умеет выдавать отчет о результатах векторизации, а также содержит значительное количество специальных директив для помощи векторизатору, объясняющие ему, что массивы выровнены, зависимостей нет и т.д.

   Пример кода, автоматически векторизуемого компилятором с нашей помощью, может выглядеть следующим образом:

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

Или так:

```
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];
}
```

Указанные директивы (ivdep и simd) говорят компилятору о том, что массивы не пересекаются, т.е. нет зависимостей по данным. Однако возможности директивы simd значительно шире и в настоящий момент для помощи векторизатору рекомендуется использовать именно ее. Подробнее эти вопросы будут рассмотрены в следующей лекции.

Директива vector aligned в примере указывает, что границы массивов, использованных в теле цикла, выровнены, что позволяет компилятору использовать команды выровненного доступа к памяти.

4. Использовать возможности Array Notation и Elemental Function в рамках технологии Intel Cilk Plus.

   Технология Intel Cilk Plus позволяет разрабатывать эффективные параллельные программы для систем с общей памятью, по сравнению с OpenMP упрощая обучение начинающих параллельному программированию, а также предоставляя мощные, логичные и достаточно простые средства организации параллелизма с использованием механизма логических задач. Наряду с этим, в Cilk Plus добавлена так называемая Array Notation, что позволяет записывать вычисления в циклах как бы без самих циклов, явно показывая компилятору, что эти вычисления можно «положить» на векторную архитектуру.

   Пример программы может выглядеть следующим образом:

```
void test(float * restrict a,
          float * restrict b,
          float * restrict c,
          int n)
{
    c[0:n] = a[0:n] * b[0:n] + a[0:n];
}
```

   Кроме того, в Intel Cilk Plus вводится специальный вид функций, т.н. Elemental Function. Эти функции, описанные специальным образом, могут выполнять операции над единицей данных. Такие функции могут быть использованы для векторизации и распараллеливания циклов.

   Подробнее Array Notation и Elemental Functions рассматриваются в лекции 5.

5. Использовать классы интринсиков для SIMD.

   В данном методе использования векторных инструкций из языков программирования высокого уровня идет речь о применении специальных классов, написанных на C++ и предназначенных для хранения и обработки упакованных данных. Сами классы могут быть написаны с использованием ассемблерных вставок или интринсиков – специальных функций, в основном однозначно соответствующих инструкциям, присутствующим в наборе команд.

6. Использовать векторные функции-интринсики.

   Данный метод предполагает использование специальных типов данных, соответствующих типам данных, аппаратно поддерживаемым

процессором/сопроцессором, и функций-интринсиков, упомянутых ранее.

7. Написать реализацию на ассемблере.

Последние три приема являются достаточно сложными и в той или иной степени затрудняют перенос программ на другое программно-аппаратное окружение. В то же время, они дают возможность низкоуровневого описания последовательности действий, что позволяет реализовать в точности то, что задумал программист, а не то, как его понял компилятор. При этом трудоемкость разработки, отладки, поддержки и развития подобных кодов существенно возрастает. Подробное обсуждение указанных приемов выходит за рамки данного курса.

3. Векторизация и математические функции

Отдельно необходимо обсудить важный вопрос о сочетании векторизации и математических функций, вызываемых в циклах, так как именно на вычисление этих функций приходится основное время работы значительного числа прикладных программ.

Ранее мы установили факт наличия в наборе команд Intel Xeon Phi специальных инструкций для вычисления четырех математических функций. Как быть с остальными функциями?

Рассмотрим следующий фрагмент кода:

```
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] + sinf(a[i]);
}
```

Казалось бы, мы видим вызов функции в теле цикла. Однако отчет векторизатора скажет нам о том, что цикл успешно векторизован. В чем же дело? Неужели в наборе команд все-таки есть векторный синус?

Для того чтобы понять суть происходящего, обсудим вопрос о возможных механизмах вычисления математических функций.

Во-первых, компилятор языка C имеет специальный модуль LibM, содержащий реализации основных математических функций. В случае если цикл не векторизовался или его не было вовсе, компилятор подставляет в машинный код вызовы функций LibM. Интересующиеся могут открыть файл math.h и обнаружить там значительное количество прототипов функций, удовлетворяющее потребностям многих прикладных программ. Указанные

функции обычно присутствуют в редакции для одинарной и двойной точности.

Сравните производительность математических функций из LibM для разных компиляторов. Часто простая перекомпиляция программы с использованием Intel Compiler приводит к существенному сокращению времени работы.

Во-вторых, компиляторы Intel имеют в своем составе модуль SVML (short vector math library), содержащий реализации математических функций, подразумевающие их вычисление для короткого вектора аргументов. Возможные длины векторов определяются количеством чисел, которое может быть упаковано в xmm (в SSE), ymm (в AVX) и zmm (в векторных расширениях Intel Xeon Phi) регистр. Выясняется, что можно сократить время работы в расчете на вычисление значения функции в одной точке, если считать значения сразу в нескольких точках. Это обстоятельство обуславливает преимущество использования функций SVML против функций LibM. Компилятор автоматически использует функции SVML, если он может векторизовать цикл, а вызов математической функции – единственное, что ему мешает это сделать. В результате вычисления становятся векторными.

В-третьих, в некоторых случаях значение n в приведенном выше примере является достаточно большим, но при этом мы располагаем достаточными объемами памяти, чтобы сначала вычислить все n значений функции sinf(), а потом использовать их в расчетах. Это может быть сделано следующим образом с использованием функций модуля VML (vector math library), распространяемого в составе математической библиотеки Intel MKL (Math Kernel Library).

```
void test(float * a, float * b, float * c, int n)
{
  // В принципе, в данном примере можно обойтись без
  // использования дополнительной памяти, сохранив все
  // значения sin() в массиве c[]
  float d[n];
  vsSin(n, a, d);
  #pragma simd
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    c[i] = a[i] * b[i] + d[i];
}
```

На достаточно больших длинах использование VML способно привести к существенному выигрышу в производительности[1].

Отдельно рекомендуется изучить вопрос о настройке точности вычислений в математических функциях. Такие возможности имеются как в базовом компиляторе, так и в модулях, предоставляемых библиотекой MKL. Обсуждение этого вопроса выходит за рамки данного курса.

Заключение

В следующей лекции будут рассмотрены вопросы оптимизации приложений для Intel Xeon Phi, включая подробное изучение вопроса о векторизации вычислений в программах на языке C.

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. Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual.

[1] В зависимости от архитектуры и реализации SVML и LibM. Результаты и тенденции на Xeon и Xeon Phi могут отличаться.