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 №6
Элементы оптимизации прикладных программ для Intel Xeon Phi: Intel MKL, Intel VTune Amplifier XE

Горшков А.В.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

1. Использование библиотеки Intel MKL при программировании на сопроцессоре Intel Xeon Phi

В данном разделе рассматриваются модели использования библиотеки Intel Math Kernel Library при программировании на Intel Xeon Phi. Дается обзор способов вызова функций библиотеки, а также рекомендации по повышению производительности приложений.

Intel Math Kernel Library (Intel MKL) [7] является одной из самых производительных библиотек математических функции для работы на аппаратном обеспечении компании Intel. Библиотека включает в себя основные функции, используемые при разработке сложных высокопроизводительных программных комплексов.

Библиотека содержит функционал из следующих областей:

- Линейная алгебра (BLAS, LAPACK, работа с разреженными данными);
- Быстрое преобразование Фурье;
- Векторные функции (тригонометрические, гиперболические, экспоненциальные и логарифмические, возведение в степень и взятие корня, округление);
- Векторные генераторы случайных чисел и функции математической статистики;
- Интерполяция данных.

На текущий момент Intel MKL поддерживает параллельное выполнение как на системах с общей памятью (в частности, на сопроцессорах Intel MIC), так и на кластерах (Рис. 1). Поддержка сопроцессора Intel Xeon Phi появилась в библиотеке с версии 11.0.

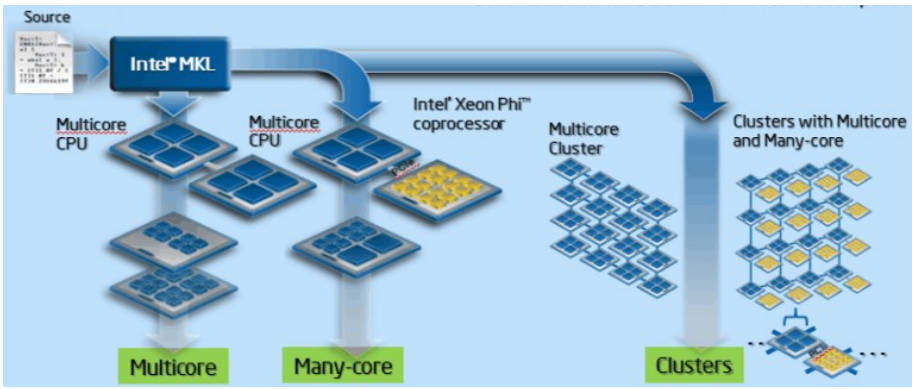

Рис. 1. Типы вычислителей, поддерживаемые библиотекой Intel MKL [1]

Поддержка сопроцессора включает в себя возможность исполнения кода библиотеки Intel MKL одновременно на центральном процессоре и сопроцессоре, позволяя получать все преимущества от использования гетерогенного режима вычислений. Код библиотеки был оптимизирован для работы с 512-битными SIMD инструкциями.

Для работы с библиотекой программисту доступны три модели (Рис. 2):

- Автоматический offload (Automatic Offload, AO) – прозрачная модель гетерогенных вычислений;
- Offload с помощью компилятора (Compiler Assisted Offload, CAO) – предоставляет возможности контроля offload'а.
- Выполнение только на сопроцессоре (Native Execution) – использование сопроцессоров в качестве независимых узлов.

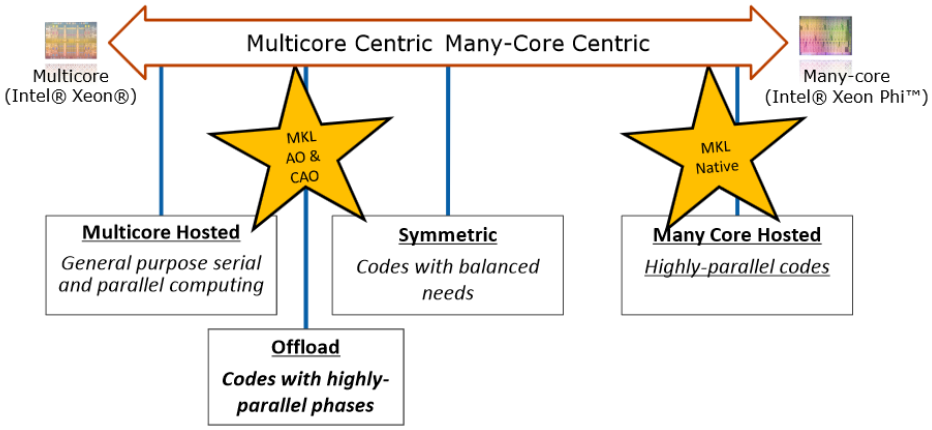

Рис. 2. Модели исполнения Intel MKL [1]

1.1. Automatic Offload (AO)

Модель автоматического offload'а является наиболее простым способом, позволяющим эффективно использовать возможности библиотеки Intel MKL на системах с одним или несколькими сопроцессорами.

Данная модель практически не требует изменения существующего кода, написанного для центрального процессора. Использование сопроцессора библиотекой Intel MKL происходит автоматически при вызове функций библиотеки. Это означает, что все обмены данными и передача управления ускорителю происходят внутри вызова функции без участия программиста.

По умолчанию, MKL заботится и о балансировке нагрузки на систему. Библиотека сама решает, нужно ли в данном случае использовать сопро-

цессор, и как распределить нагрузку между процессором и сопроцессором. Основной критерий использования ускорителя для выполнения той или иной функции – эффективность. Т.е. если данная функция при данных входных параметрах будет работать лучше на CPU, то ускоритель использоваться не будет. В случае если в момент вызова функции сопроцессор занят другой задачей, для вычисления этой функции будет использован центральный процессор.

Распределение нагрузки также может быть выполнено автоматически. При этом используются все доступные в системе сопроцессоры и CPU, количество ускорителей и процент нагрузки на них выбирается библиотекой исходя из достижения лучшей производительности в каждом конкретном случае. Это позволяет автоматически эффективно использовать все доступные вычислительные ресурсы системы. Следует отметить, что программист может задавать желаемое распределение нагрузки самостоятельно.

Для того чтобы начать работать с AO, достаточно включить этот режим. Из программы это делается вызовом функции:

```
mkl_mic_enable();
```

Возможно также использование переменной окружения:

```
MKL_MIC_ENABLE=1
```

Отметим, что если в системе не установлено ни одного сопроцессора, функции Intel MKL будут работать на CPU без дополнительных накладных расходов.

Еще раз заметим, что исполняться на сопроцессоре будут только те функции, для которых существует эффективная реализация. В версии Intel MKL 11.0 на сопроцессоре будут выполняться только функции BLAS 3-го уровня *GEMM, *TRSM и *TRMM. В следующих версиях библиотеки планируется расширить поддержку сопроцессора.

Для указанных выше функций выполнение на сопроцессоре зависит и от входных размеров матрицы:

- Функции *GEMM выполняются на ускорителе, если M, N > 2048;
- Функции *TRSM/*TRMM выполняются на ускорителе, если M, N > 3072.

Для квадратных матриц вычисления на сопроцессоре происходят быстрее.

Для того чтобы задать желаемое распределение нагрузки между процессором и сопроцессорами можно воспользоваться функцией:

```
mkl_mic_set_Workdivision(MKL_TARGET_MIC, 0, 0.5);
```

В данном примере указывается, что на нулевой сопроцессор должно приходиться 50% общей нагрузки.

Такого же эффекта можно добиться с помощью переменной окружения:

```
MKL_MIC_0_WORKDIVISION=0.5
```

Обратим внимание, что данные команды являются лишь советами среде выполнения Intel MKL и реально могут не исполняться либо исполняться не точно.

Для того чтобы отключить режим AO после того, как он был задействован, необходимо либо вызвать функцию:

```
mkl_mic_disable();
```

либо перенести всю вычислительную нагрузку на центральный процессор вызовом функции:

```
mkl_mic_set_workdivision(MIC_TARGET_HOST, 0, 1.0);
```

либо воспользоваться переменной окружения:

```
MKL_HOST_WORKDIVISION=100
```

Для более эффективной работы режима автоматического offload'а рекомендуется избегать использования ядра операционной системы ускорителя для вычислений, т.к. это ядро обычно используется для выполнения передач данных и очистки памяти.

Приведем пример настройки соответствующей переменной окружения для случая сопроцессора с 60 ядрами и 4 потоками на ядро:

```
MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-
236:1]
```

В данном примере для вычислений резервируются первые 59 ядер, т.е. первые 236 потоков.

Дополнительно следует явно привязать потоки хоста к ядрам, чтобы избежать миграции потоков:

```
KMP_AFFINITY=granularity=fine,compact,1,0
```

Обратите внимание на различие имен переменных окружения для хоста и сопроцессора. В случае Intel MIC, к имени переменной добавляется приставка MIC_ (OMP_NUM_THREADS/MIC_OMP_NUM_THREADS и т.п.).

Подробнее об этих переменных окружения для Intel компилятора и описанных выше параметрах можно почитать здесь [9].

1.2. Compiler Assisted Offload (CAO)

В данной модели процесс offload'а явно контролируется программистом с помощью директив компилятора. По сути, данная модель является обычной offload моделью программирования ускорителя, а значит, позволяет

пользоваться всеми возможностями компилятора для переноса части вычислений на сопроцессор. Подробности о программировании в режиме offload можно найти в лекции №5.

Рассмотрим пример вызова функции для умножения матриц:

```
#pragma offload target(mic) \
in(transa, transb, N, alpha, beta) \
in(A:length(matrix_elements)) \
in(B:length(matrix_elements)) \
in(C:length(matrix_elements)) \
out(C:length(matrix_elements) alloc_if(0))
{
    sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B,
        &N, &beta, C, &N);
}
```

В данном случае явно указан код (функция умножения матриц), который переносится на сопроцессор, а также описаны все действия по управлению передачей данных.

В отличие от режима AO, где на Intel Xeon Phi может работать только некоторое подмножество функций Intel MKL, текущая модель позволяет запускать на сопроцессоре абсолютно все функции библиотеки. Однако это не означает, что во всех случаях удастся получить лучшую производительность, чем при работе только на CPU.

Модель CAO позволяет использовать все offload возможности компилятора для достижения лучшей производительности, в частности оптимизировать работу с данными на сопроцессоре, явно указывая моменты выделения/удаления памяти. Это позволяет, например, организовать переиспользование данных на сопроцессоре и тем самым обеспечить снижение объема передаваемой на хост (или с хоста) информации:

```
__declspec(target(mic)) static float *A, *B, *C, *C1;

// Transfer matrices A, B, and C to coprocessor and do not
// de-allocate matrices A and B
#pragma offload target(mic) \
in(transa, transb, M, N, K, alpha, beta, LDA, LDB, LDC) \
in(A:length(NCOLA * LDA) free_if(0))\
in(B:length(NCOLB * LDB) free_if(0)) \
inout(C:length(N * LDC))
{
    sgemm(&transa, &transb, &M, &N, &K, &alpha, A, &LDA,
        B, &LDB, &beta, C, &LDC);
}

// Transfer matrix C1 to coprocessor and reuse
// matrices A and B
#pragma offload target(mic) \
```

```
in(transa1, transb1, M, N, K, alpha1, \
beta1, LDA, LDB, LDC1) \
nocopy(A:length(NCOLA * LDA) alloc_if(0) free_if(0)) \
nocopy(B:length(NCOLB * LDB) alloc_if(0) free_if(0)) \
inout(C1:length(N * LDC1))
{
    sgemm(&transa1, &transb1, &M, &N, &K, &alpha1,
        A, &LDA, B, &LDB, &beta1, C1, &LDC1);
}

// Deallocate A and B on the coprocessor
#pragma offload target(mic) \
nocopy(A:length(NCOLA * LDA) free_if(1)) \
nocopy(B:length(NCOLB * LDB) free_if(1)) \
{ }
```

Явное управление запуском кода на ускорителе позволяет также делать перекрытие вычислений с обменом данными, либо обеспечивать одновременную работу CPU и ускорителя. Однако возможности автоматической балансировки нагрузки теряются.

При использовании модели CAO прежде всего следует избегать ненужных обменов данными между хостом и сопроцессором (по аналогии с приведенным выше примером). Также как и для AO, следует освободить ядро операционной системы ускорителя от вычислений. И наконец, имеет смысл работать с увеличенным до 2 МБ размером страницы памяти. Для этого следует использовать переменную окружения, инициализированную как:

```
MIC_USE_2MB_BUFFERS=64K
```

При этом будут использоваться страницы размером в 2 МБ (64 КБ для данной переменной является пороговым значением, начиная с которого размер страниц памяти будет увеличен до 2 МБ).

Заметим также, что в рамках одной программы возможно использование обеих этих моделей. Одни вызовы можно делать в режиме AO, другие в режиме CAO. Единственное ограничение здесь состоит в необходимости явно указывать распределение нагрузки для AO вызовов, иначе все они будут использовать только CPU.

1.3. Выполнение только на сопроцессоре

Режим исполнения только на сопроцессоре предполагает использование только сопроцессоров без CPU. Каждый сопроцессор представляет собой отдельный вычислительный узел, который может обмениваться данными с другими узлами посредством MPI сообщений.

Данная модель предполагает написание программы так, как это делается для обычного центрального процессора, а затем ее компиляцию с ключом “–mmic”. Запуск полученного бинарного файла должен осуществляться непосредственно на сопроцессоре.

Иными словами, это обычная модель программирования с выполнением кода только на Intel Xeon Phi.

При использовании этой модели рекомендуется задействовать все доступные потоки ускорителя, например, для Intel Xeon Phi с 60 ядрами и 4 потоками на ядро:

```
MIC_OMP_NUM_THREADS=240
```

Переменную KMP_AFFINITY рекомендуется устанавливать как [9]:

```
KMP_AFFINITY=explicit,proclist=[1-
240:1,0,241,242,243],granularity=fine
```

Также рекомендуется использовать большие 2 МБ страницы памяти.

1.4. Рекомендации по выбору модели программирования

При выборе модели программирования для вашего приложения следует обратить внимание на следующие факторы:

- Если код имеет высокую степень параллелизма либо необходимо использовать ускорители как отдельные вычислительные узлы, то имеет смысл использовать модель выполнения только на сопроцессоре;
- Если в вашем случае доля вычислений на единицу памяти велика и вам нужны функции *GEMM, *TRMM, *TRSM либо функции LU и QR факторизации (появятся в ближайших релизах), тогда лучше выбрать модель AO;
- Если в программе есть участки вычислений, подходящие для перекрытия передач данных либо возможно переиспользование участков памяти на сопроцессоре, тогда можно использовать модель CAO.

Отметим также, что в случае недостаточной производительности в режимах offload, вы всегда можете легко перейти на использование CPU.

2. Оптимизация приложений с помощью Intel VTune Amplifier XE

В данном разделе описаны подходы к оптимизации программ для Intel Xeon Phi с использованием инструмента профилировки приложений Intel

VTune Amplifier XE. Дается краткий обзор Intel VTune Amplifier, приводятся способы запуска профилировщика на сопроцессоре как в режиме GUI, так и с помощью командной строки. Описываются основные метрики эффективности, получаемые с помощью профилировки, на которые следует обратить внимание при оптимизации приложений.

Основная рекомендация при оптимизации программ для Intel Xeon Phi состоит в том, что первым шагом должна стать оптимизация приложения для центрального процессора.

Для того чтобы выделить те участки программы, которые нуждаются в оптимизации прежде всего, имеет смысл воспользоваться инструментом Intel VTune Amplifier XE. Применение hotspot анализа покажет те функции и участки программы, на которые тратится больше всего времени. Часто этого бывает достаточно. А если требуется более детальная оптимизация, тогда можно обратиться к другим типам анализа с целью получения низкоуровневой информации о ходе выполнения приложения.

Отметим, что найти кандидатов для оптимизации можно и с помощью отчетов компилятора Intel. В частности, можно получить информацию о функциях и циклах, занимающих больше всего времени, а также о среднем, минимальном и максимальном числе итераций этих циклов. Для этого следует собирать приложение с ключами:

```
-profile-functions -profile-loops=all -profile-loops-
report=2
```

Результаты профилировки, которые будут записаны в файлы в текущей директории по окончании работы приложения, можно будет посмотреть либо в виде таблицы (dump файл), либо с помощью специального инструмента с GUI – Loop Profile Viewer (xml файл).

В процессе оптимизации приложения необходимо поддерживать его корректность, что особенно актуально при распараллеливании. Для выявления ошибок многопоточности можно использовать инструмент Intel Inspector XE. Также для эффективного распараллеливания приложения полезно иметь возможность анализа его выполнения с точки зрения работы потоков в нем. Такая возможность присутствует в Intel VTune Amplifier XE.

Однако Intel Inspector XE и анализ многопоточного исполнения в Intel Amplifier XE поддерживаются только для CPU. Поэтому:

- Рекомендуется использовать Intel Inspector XE для вашего кода с **отключенной функцией offload'а** для выявления в нем таких ошибок, как зависимость по данным, тупики и т.п. После исправления всех выявленных ошибок можно включать offload режим и продолжать отладку на сопроцессоре;

- Рекомендуется использовать инструменты анализа эффективности параллельных приложений в Intel VTune Amplifier XE для вашего кода с **отключенной функцией offload'а** для выявления проблем эффективности распараллеливания. И только после того, как удастся устранить все, что возможно, переходить на работу с сопроцессором и проводить на нем дальнейшую оптимизацию. При этом следует обратить внимание на эффективность синхронизации, т.к. число потоков на ускорителе значительно превосходит это число на обычном CPU. А также следует позаботиться о балансировке нагрузки опять же в силу значительно большего числа потоков.

2.1. Краткий обзор инструмента Intel VTune Amplifier XE

Инструмент Intel VTune Amplifier XE является профилировщиком производительности и масштабируемости приложений на многоядерных системах. Входит в состав набора для разработки ПО Intel Parallel Studio XE.

Инструмент, в частности, позволяет:

- Находить функции и участки кода, на выполнение которых расходуется больше всего времени. Анализирует стеки вызовов и исходный код;
- Определять количество внутренних событий процессора, которые влияют на производительность. Например, промахи кэша разных уровней, неверно предсказанные ветвления и др.;
- Определять время ожидания в блокировках потоков, а также уровень загрузки CPU.

Intel VTune Amplifier XE позволяет настроить желаемые параметры анализа работы приложения, а также включает в себя определенное количество предварительно настроенных типов анализа, наиболее используемые из которых:

- **Hotspots**. Предназначен для выявления «узких мест» в программе. Определяет, какие функции или участки программы работают дольше всего. В основном используется на первом этапе оптимизации для выявления областей кода, требующих ускорения.
- **Concurrency**. Этот тип анализа показывает эффективность использования ядер процессора во время выполнения программы. Демонстрирует качество распараллеливания кода и участки, которые следует распараллелить.
- **Locks and Waits**. Показывает точки блокировки и время ожидания потоков. Предназначен для оценки эффективности используемой схемы синхронизации.

Кроме описанных выше типов анализа в инструменте присутствует возможность сбора информации о событиях микроархитектурного уровня, таких как доступ к кэшам различного уровня и промахи кэша, доступ к памяти, неверно предсказанные ветвления и др. Присутствуют и различные готовые типа анализа, направленные на выявление определенных проблем с производительностью (например, проблем доступа в кэши или память).

Инструмент доступен для операционных систем семейства Windows и Linux.

2.2. Профилировка приложений на Intel Xeon Phi

Intel VTune Amplifier XE позволяет выполнять профилировку приложений непосредственно на сопроцессоре. На текущий момент пользователю доступны следующие готовые типы анализа:

- **Lightweight Hotspots.** Позволяет определить функции и участки кода, на выполнение которых тратится больше всего времени. Аналогичен hotspots, но статистика собирается с использованием специальных регистров процессора для мониторинга производительности [10].
- **General Exploration.** Позволяет выявить микроархитектурные особенности, отрицательно влияющие на производительность. Это могут быть, например, частые промахи L1 или L2 кэша, промахи TLB кэша или степень векторизации кода.
- **Bandwidth.** Предназначен для анализа пропускной способности памяти.

На текущий момент поддерживается только одна технология сбора данных о работе приложения – Event-Based Sampling. Эта технология опирается на использование специальных аппаратных регистров (Performance Monitoring Units), предназначенных для учета различных низкоуровневых событий, происходящих во время работы программы. В текущих сопроцессорах Intel Xeon Phi на ядро приходится 2 регистра, накапливающих информацию о событиях, специфичных для потока или ядра. Присутствуют также 4 регистра за пределами ядра, не обладающих информацией о потоках и ядрах.

Соответственно за один запуск можно получить информацию максимум о 2 событиях ядра и 4 внешних событиях. Если нужно больше информации, то Intel VTune Amplifier XE будет выполнять ваше приложение несколько раз.

В дополнение к существующим типам анализа пользователь может создавать свои типы для получения информации о других интересующих его событиях. Проще всего новый тип анализа создавать на основе существу-

ющего. Для этого необходимо скопировать наиболее подходящий тип анализа и добавить в него интересующие вас счетчики событий.

Подробнее о событиях сопроцессора Intel Xeon Phi можно почитать в документации PMU [10].

Далее рассмотрим процесс запуска профилировки приложения на Intel Xeon Phi. Предполагается, что все действия выполняются на машине с подключенным к ней сопроцессором.

Запуск процесса профилировки из GUI.

Рассмотрим процесс запуска приложения с помощью GUI компонента Intel VTune Amplifier XE. Во-первых, создаем новый проект, в рамках которого будем исследовать нужное нам приложение. В случае offload программы заполняем поля свойств проекта следующим образом (вкладка Target):

- Application: полный путь к исполняемому файлу;
- Application parameters: параметры приложения (если необходимо);
- Working directory: путь к рабочей директории, где будут храниться результаты профилировки (обычно не имеет значения).

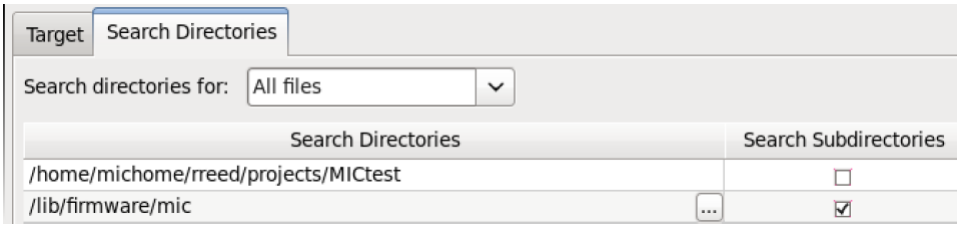

Рис. 3. Настройка путей к исходным файлам приложения в Intel VTune Amplifier XE [2]

На вкладке Search Directories нужно указать путь к исходным файлам приложения для возможности навигации по коду программы при просмотре результатов (Рис. 3).

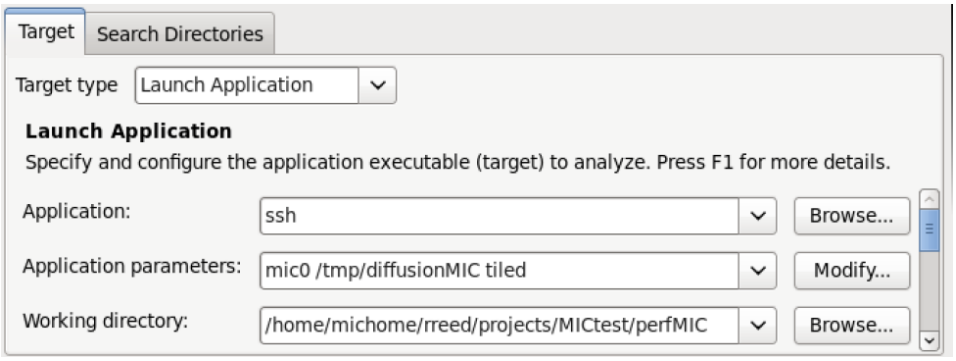

Рис. 4. Настройка параметров проекта в Intel VTune Amplifier XE [2]

Если выполняется запуск программы в режиме работы только на сопроцессоре, то в качестве приложения для запуска указывается ssh, а само приложение указывается в качестве параметра в поле Application Parameters (Рис. 4).

Следующий шаг – выполнение профилировки приложения. Для этого необходимо выбрать пункт меню «New Analysis…», выбрать нужный тип анализа в дереве типов из папки «Knights Corner Platform Analysis» и начать профилировку (Рис. 5).

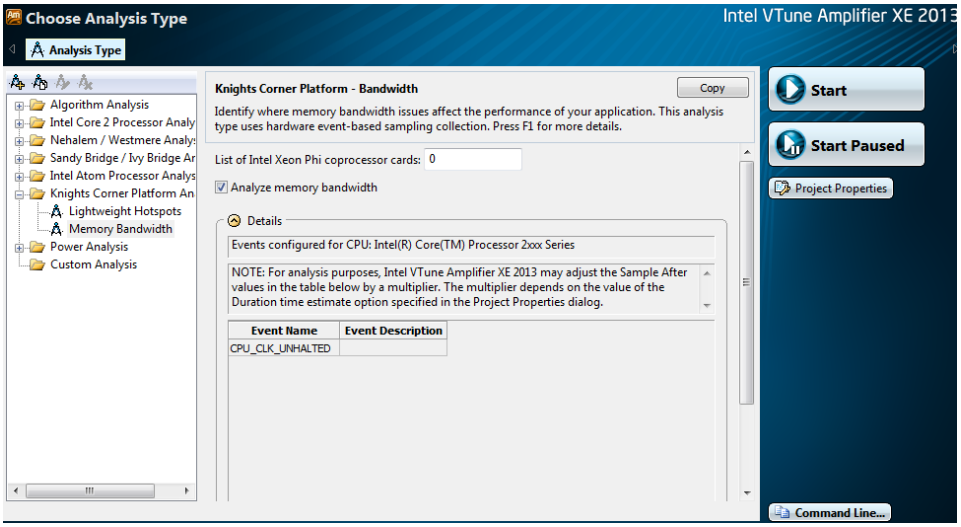

Рис. 5. Выбор типа анализа в Intel VTune Amplifier XE

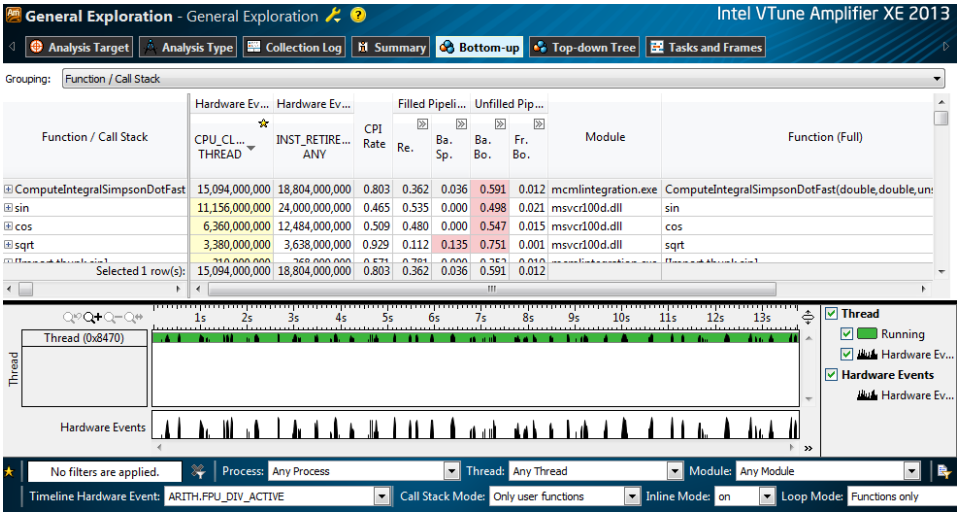

Рис. 6. Результаты профилировки в Intel VTune Amplifier XE

Для создания собственного типа анализа на основе существующего достаточно нажать на кнопку «Copy» и в открывшемся окне добавить нужные счетчики. Также обратите внимание, что нажав на кнопку «Command Line…» вы получите командную строку для текущего типа анализа, с помощью которой можно запустить профилировку из консоли.

По завершении профилировки результаты будут отражены на экране (Рис. 6).

Запуск процесса профилировки из командной строки.

Для работы с Intel VTune Amplifier XE из командной строки прежде всего необходимо создать все нужные переменные окружения. Это можно сделать вызовом специального скрипта из папки с установленным инструментом:

```
source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
```

После этого можно запускать процесс профилировки.

Для offload приложения это делается командой:

```
amplxe-cl –collect knc-lightweight-hotspots –knob target-
cards=0,1 –result-dir ./offload_cmd -- ./offload.out
```

Рассмотрим подробнее ключи запуска профилировщика.

Ключ «–collect knc-lightweight-hotspots» говорит о том, что будет проводиться один из предварительно настроенных типов анализа. В данном примере используется Lightweight Hotspots. Остальные типы имеют имена «knc-general-exploration» и «knc-bandwidth» соответственно.

Следующий ключ «–knob target-cards=0,1» говорит о том, на каких сопроцессорах запускать анализ приложения. Имеет смысл только для offload приложений.

Ключ «–result-dir ./offload_cmd» указывает на директорию, куда будут записаны результаты анализа.

И, наконец, параметр «-- ./offload.out» говорит о том, какое приложение (в нашем случае это «./offload.out») и с какими аргументами должно быть запущено.

Для приложений в режиме исполнения только на сопроцессоре командная строка будет такой:

```
amplxe-cl –collect knc-lightweight-hotspots –reslut-dir
./native-cmd -- ssh mic0 “export LD_LIBRARY_PATH=~/;
export OMP_NUM_THREADS=244; export KMP_AFFINITY=balanced;
./native.out”
```

Обратите внимание, что в качестве приложения для запуска используется команда ssh, после которой следует имя узла сопроцессора, и далее команды для выполнения на этом сопроцессоре.

Приведем пример того, как из командной строки запустить свой собственный тип анализа:

```
amplxe-cl –collect-with runsa-knc –knob event-
config=CPU_CLK_UNHALTED,L2_DATA_READ_MISS_MEM_FILL:sa=1000,
L2_DATA_WRITE_MISS_MEM_FILL,L2_VICTIM_REQ_WITH_DATA,SNP_HIN
T_L2,HWP_L2MISS –knob target-cards=0,1 –result-dir
./custom-cmd -- ./offload.out
```

Для указания того, что будет использоваться пользовательский тип анализа на ускорителе, используется ключ «–collect-with runsa-knc». Конкретные события, результаты по которым вы хотите получить, описаны в качестве параметров ключа «–knob event-config=…». Используемые здесь имена событий описаны по ссылке [10].

Напомним, что нужный вам тип анализа можно настроить через GUI приложение, после чего там же получить командную строку для его запуска. Описание дополнительных аргументов приложения ample-cl можно узнать из его справки:

```
amplxe-cl –help
```

Просмотр результатов анализа, полученных после запуска профилировщика из командной строки, может быть осуществлен двумя методами.

Первый метод предполагает использование GUI приложения. Вам нужно скопировать результаты анализа с удаленной на локальную машину с GUI, после чего открыть файл *.amplxe с помощью GUI приложения Intel VTune Amplifier XE. Отметим, что это предпочтительный метод работы с результатами, так как он является наиболее удобным и наглядным.

Второй метод использует исключительно возможности командной строки.

Для просмотра общей статистики по конкретному запуску необходимо выполнить команду:

```
amplxe-cl –report summary –r ./offload_cmd/
```

Здесь «./offload_cmd/» это директория с результатами анализа. Ключ «-report» указывает на тип выводимой информации. Например, если мы хотим получить список наиболее медленных функций, тогда нужно запросить вывод данных о «горячих точках»:

```
amplxe-cl –report hotspots –r ./offload_cmd/
```

Еще один вариант – получение информации об аппаратных событиях, произошедших за время работы приложения:

```
amplxe-cl –report hw-events –r ./offload_cmd/
```

Результаты выдачи можно фильтровать по имени процесса или модуля:

```
amplxe-cl -report hotspots -filter process=offload_main -
filter module=offload.out -r ./offload_cmd/
```

Можно записывать выдачу в файл:

```
amplxe-cl -report hotspots -report-output ./vtune-
output.txt -r ./offload_cmd/
```

2.3. Метрики для оценки эффективности приложений на Intel Xeon Phi

В данном разделе приведено описание основных метрик для оценки эффективности приложений на Intel Xeon Phi с помощью Intel VTune Amplifier XE. Приведенные здесь метрики и рекомендации по оптимизации кода актуальны и для CPU, однако конкретные значения этих метрик приводятся только для сопроцессоров Intel Xeon Phi.

Количество тактов на инструкцию (cycles per instruction, CPI).

Эта метрика показывает среднее число тактов процессора, которое требуется для выполнения одной инструкции. Иными словами это индикатор того, как сильно латентность доступа к памяти влияет на производительность приложения.

Данная метрика может вычисляться относительно аппаратного потока либо относительно ядра процессора. Чем меньше этот показатель, тем лучше работает приложение. Для сопроцессоров Intel Xeon Phi минимальные значения этого показателя приведены в Таблица 1.

Таблица 1. Минимальные теоритические показатели CPI на ядро и на поток для сопроцессоров Intel Xeon Phi

Число аппаратных потоков на ядро	Минимальный (лучший) показатель CPI на ядро	Минимальный (лучший) показатель CPI на поток
1	1.0	1.0
2	0.5	1.0
3	0.5	1.5
4	0.5	2.0

Для вычисления описанных выше характеристик Intel VTune Amplifier XE использует события CPU_CLK_UNHALTED и INSTRUCTIONS_EXECUTED. В частности:

- CPI Per Thread = CPU_CLK_UNHALTED / INSTRUCTIONS_EXECUTED
- CPI Per Core = (CPI Per Thread) / (Число используемых аппаратных потоков)

Во время применения определенных шагов по оптимизации приложения рекомендуется следить за этими характеристиками. Приемлемым можно считать следующие значения этих метрик:

- CPI Per Thread <= 4.0;
- CPI Per Core <= 1.0.

Если одна из характеристик больше соответствующего значения – это повод задуматься о дополнительной оптимизации. Причем большие значения CPI следует расценивать как повод к уменьшению латентности доступа к памяти.

Таким образом, задача оптимизации состоит в том, чтобы уменьшать значение CPI. И приведенные ниже метрики и рекомендации по их улучшению помогают этой цели добиться. Однако следует отметить, что, например, при использовании векторизации CPI может увеличиваться. И это нормально. Связано это с тем, что мы переходим от скалярных инструкций к векторным. И при этом на выполнение одной векторной инструкции может потребоваться больше тактов. Но не стоит забывать, что за одну векторную инструкцию мы выполним больше операций.

Объем вычислений на единицу данных (compute to data access ratio).

Данная метрика позволяет оценить средний объем вычислений, который приходится на единицу данных в вашем приложении. Очевидно, что чем больше этот показатель, тем эффективнее будет работать программа.

Выделяют два типа этой метрики:

- L1 Compute to Data Access Ratio – используется для оценки того, насколько ваше приложение подходит для переноса на Intel Xeon Phi. Для того чтобы программа могла эффективно выполняться на сопроцессоре, она должна быть векторизованной, и в идеале выполнять несколько операций с одними и теми же данными (или линейками кэша). Данная метрика вычисляет среднее число векторных операций, приходящихся на один доступ к L1 кэшу.
- L2 Compute to Data Access Ratio – показывает среднее число векторных операций, приходящихся на один доступ к L2 кэшу.

Для вычисления этих метрик используются следующие события:

- VPU_ELEMENTS_ACTIVE – число векторных операций на поток;
- DATA_READ_OR_WRITE – число операций чтения и записи в L1 кэш данных на поток;
- DATA_READ_MISS_OR_WRITE_MISS – число L1 кэш промахов при чтении и записи на поток.

Описанные выше метрики вычисляются по формулам:

- L1 Compute to Data Access Ratio = VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE;
- L2 Compute to Data Access Ratio = VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_WRITE_MISS.

Приведем некоторые рекомендации для оценки эффективности приложения с помощью этих метрик. Приложения, приемлемо работающие на сопроцессоре, должны обладать следующими значениями метрик:

- L1 Compute to Data Access Ratio < показателя интенсивности векторизации (см. **Векторизация**);
- L2 Compute to Data Access Ratio < 100 * (L1 Compute to Data Access Ratio).

Для улучшения этих показателей следует увеличить плотность вычислений посредством векторизации, а также сократить число обращений к памяти. Обратите внимание эффективную работу с кэш памятью, пользуйтесь выравниванием данных.

Латентность доступа к памяти.

Высокая латентность доступа к данным существенно снижает эффективность приложения. Для оценки влияния этого фактора рекомендуется использовать следующую метрику:

- Оценка влияния латентности (Estimated Latency Impact) = (CPU_CLK_UNHALTED – EXEC_STAGE_CYCLES – DATA_READ_OR_WRITE) / DATA_READ_OR_WRITE_MISS.

Здесь EXEC_STAGE_CYCLES – число тактов процессора, на которых поток выполнял вычислительные операции. Остальные события описаны выше.

Применять оптимизацию здесь следует тогда, когда значение этого показателя **больше 145**. Для оптимизации следует повышать локальность данных, используя программную предвыборку данных, блочный доступ к данным в кэш памяти, потоковые операции работы с данными и выравнивание.

Использование TLB кэша.

Неэффективное использование TLB кэша приводит к увеличению латентности доступа к памяти и, как следствие, снижению производительности приложений.

Для оценки эффективности доступа в TLB кэш используются следующие показатели:

- L1 TLB miss ratio = DATA_PAGE_WALK / DATA_READ_OR_WRITE;
- L2 TLB miss ratio = LONG_DATA_PAGE_WALK / DATA_READ_OR_WRITE
- L1 TLB misses per L2 TLB miss = DATA_PAGE_WALK / LONG_DATA_PAGE_WALK

Здесь DATA_PAGE_WALK – число промахов L1 TLB кэша, а LONG_DATA_PAGE_WALK – число промахов L2 TLB кэша.

Необходимость в оптимизации здесь появляется, если:

- L1 TLB miss ratio > 1%;
- L2 TLB miss ratio > 0.1%;
- L1 TLB misses per L2 TLB miss > 1.

Для улучшения ситуации следует обратить внимание на эффективность использования кэша и уменьшать латентность доступа к памяти.

Если отношение (L1 TLB miss / L2 TLB miss) достаточно велико, можно попробовать использовать страницы TLB кэша большего размера.

Если в коде есть циклы, в теле которых на каждой итерации выполняются действия с разными участками данных, лучше разбить такой цикл на несколько более маленьких.

Векторизация.

Рассмотрим такую метрику, как интенсивность векторизации. Она показывает, насколько эффективно векторизован ваш код:

Интенсивность векторизации (vectorization intensity) = VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED.

Здесь VPU_INSTRUCTIONS_EXECUTED – число векторных инструкций, выполняемых потоком, VPU_ELEMENTS_ACTIVE – число активных векторных элементов на векторную инструкцию, или, другими словами, число векторных операций (за одну векторную инструкцию может выполняться несколько операций).

Оптимизировать нужно, если этот параметр **меньше 8** при использовании чисел двойной точности и **меньше 16** при использовании чисел одинарной точности.

Компилятор Intel может выполнять автоматическую векторизацию вашего кода. Для получения информации о том, какие циклы были векторизованы, а какие – нет, используйте соответствующие отчеты компилятора.

Для подсказки компилятору используйте директивы #pragma ivdep, #pragma simd и др. Для ручной векторизации используйте возможности технологии Intel Cilk Plus.

Следите за выравниванием данных при векторизации.

Пропуская способность памяти.

Величина показателя пропускной способности памяти вычисляется следующим образом:

- Read bandwidth = (L2_DATA_READ_MISS_MEM_FILL + L2_DATA_WRITE_MISS_MEM_FILL + HWP_L2MISS) * 64 / CPU_CLK_UNHALTED
- Write bandwidth = L2_VICTIM_REQ_WITH_DATA + SNP_HITM_L2) * 64 / CPU_CLK_UNHALTED
- Memory Bandwidth = (Read bandwidth + Write bandwidth) * (Частота процессора в ГГц).

Здесь:

- L2_DATA_READ_MISS_MEM_FILL – число операций чтения, приводящих к обращению к оперативной памяти, включая операции предвыборки;
- L2_DATA_WRITE_MISS_MEM_FILL – число операций записи, приводящих к обращению к оперативной памяти на чтение, включая операции предвыборки;
- L2_VICTIM_REQ_WITH_DATA – число замещений данных, приводящих к обращению к оперативной памяти на запись;
- HWP_L2MISS – число аппаратных предвыборок, которые привели к L2 кэш промаху;
- SNP_HITM_L2 – число событий возникающих в случае, когда данные, измененные в кэше одного ядра, нужны другому ядру;
- CPU_CLK_UNHALTED – число тактов процессора.

Если эта величина < 80 GB/сек (практический максимум для 8 контроллеров памяти равен 140 GB/сек), тогда имеет смысл выполнять соответствующую оптимизацию.

Для этого следует улучшить локальность данных в кэшах, использовать операции потоковой работы с данными, задействовать программную предвыборку.

Подробнее об этих метриках и приемах оптимизации приложений для Intel Xeon Phi можно узнать здесь [4, 5].

Литература

Использованные источники информации

1. Intel Corporation. Advanced Intel Xeon Phi Coprocessor Workshop, Intel Math Kernel Library 11.0. Support for Intel Xeon Phi Coprocessor, September 2012.
2. Intel Corporation. Advanced Intel Xeon Phi Coprocessor Workshop, Performance Tuning for Intel Xeon Phi Coprocessors, September 2012.
3. L. Belinda. Intel VTune Amplifier XE video tutorial 5: Using the command line, 2013 [http://software.intel.com/ru-ru/videos/intel-vtune-amplifier-xe-video-tutorial-5-using-the-command-line]
4. D. Mackay. Optimization and Performance Tuning for Intel Xeon Phi Coprocessors, Part 1: Optimization Essentials, 2012 [http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization]
5. S. Cepeda. Optimization and Performance Tuning for Intel Xeon Phi Coprocessors, Part 2: Understanding and Using Hardware Events, 2012 [http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding]

Дополнительная литература

6. J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. -Morgan Kaufmann, 2013. -432 p.

Информационные ресурсы сети Интернет

7. Intel Math Kernel Library Documentation [http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation]
8. Intel Developer Zone [http://software.intel.com/en-us/mic-developer]
9. Intel Compiler Documentation. Thread Affinity Interface [http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_openmp_thread_affinity.htm]
10. Intel Xeon Phi Coprocessor. Performance Monitoring Units Documentation [http://software.intel.com/sites/default/files/forum/278102/intelr-xeon-phitm-pmu-rev1.01.pdf]