Содержание

Введение

Сопроцессоры Intel Xeon Phi часто используются для повышения производительности современных суперкомпьютеров. Согласно выпуску рейтинга TOP500 за июнь 2017 года, 14 систем в списке используют Xeon Phi в качестве ускорителя, а 13 — в качестве загружаемых автономных процессоров (включая текущую систему Cori с производительностью 27.8 PFLOP/s в NERCS и 24.9 PFLOP/s в Oakforest-PACS в JCAHPC в Японии, занимающие шестую и седьмую позиции в списке соответственно).

Модели программирования для сопроцессора Xeon Phi включают так называемый режим разгрузки. В этом случае основное приложение выполняется на центральном процессоре, а на сопроцессоре выполняется только часть кода, обозначенного программистом. Для этой цели можно использовать несколько сред выполнения и библиотек. Можно использовать, например, прагмы Language Extensions for Offload (LEO), включенные в компилятор Intel. Целевые прагмы, похожие на LEO, теперь также включены в стандарт OpenMP 4.5. Другой подход заключается в использовании API низкоуровневого интерфейса разгрузки сопроцессора (COI) или Симметричный коммуникационный интерфейс (SCIF) оба предоставлены Intel. Кроме того, можно использовать внешние библиотеки, такие как библиотека Hetero Streams Library (hStreams) или библиотека гHeterogeneous Active Messages (HAM)

В то время как первое поколение карт Intel Xeon Phi под кодовым названием Knights Corner работает как сопроцессор и позволяет пользователям разгружать вычисления по шине PCIe, поколение Knights Landing используется в качестве автономных процессоров. Однако модельпрограммирования разгрузки расширяется за счет разгрузки по структуре если ускорители не подключены напрямую к хосту через PCIe. Существующие коды, использующие стратегию разгрузки, совместимы с разгрузкой поверх структуры и поэтому могут запускаться с минимальными изменениями на суперкомпьютерах, где узлы Xeon и Xeon Phi создают отдельные подкластеры (cluster booster).

1. Архитектуры сопроцессора Intel Xeon Phi

Intel Xeon Phi состоит из 61 ядра , соединенных высокопроизводительным двунаправленным соединением на кристалле. Сопроцессор работает под управлением операционной системы Linux и поддерживает все важные инструменты разработки Intel, такие как компилятор C/C++ и Fortran, MPI и OpenMP, высокопроизводительные библиотеки, такие как MKL, отладчик и инструменты трассировки, такие как Intel VTune Amplifier XE. Традиционные инструменты UNIX на сопроцессоре поддерживаются через BusyBox, который объединяет крошечные версии многих распространенных утилит UNIX в один небольшой исполняемый файл. Сопроцессор подключен к процессору Intel Xeon через шину PCI Express (PICe). Осуществление виртуализированный стек TCP/IP позволяет получить доступ к сопроцессору как к сетевому узлу.

1.2. Компиляция под сопроцессор.

Для достижения хорошей производительности следует помнить о следующих пунктах.

- Данные должны быть выровнены до 64 байт (512 бит) для архитектуры MIC,в отличие от 32 байт (256 бит) для AVX и 16 байт (128 бит) для SSE.

- Из-за большой ширины SIMD (64 байта). Векторизация для архитектуры MIC даже важнее, чем для Intel Xeon. Архитектура MIC предлагает новые инструкции, такие как сбор/распределение, плавное умножение-сложение, маскированные векторные инструкции и т.д., которые позволяют распределить больше циклов на сопроцессоре, чем на хосте на базе Intel Xeon.

- Используйте прагмы, такие как pragma ivdep, #pragma vector always, #pragma vector aligned, #pragma т.д. Достижения авто векторизации. Автовекторизация включена на уровне оптимизации по умолчанию. • Разрешить

компилятору генерировать отчеты о векторизации с помощью опции компилятора --vecreport2 чтобы увидеть, были ли циклы векторизованный для MIC (сообщение «*MIC* Loop был векторизован» и т. д.). Варианты -opt-report-phase hlo (Отчет оптимизатора высокого уровня) или -opt-report-phase ipo_inl (Встраивание отчета) также может быть полезным. • Явное векторное программирование также возможно с помощью языковых расширений Intel Cilk Plus (обозначение массива C/ C++, векторные элементарные функции) или новых конструкций SIMD из OpenMP 4.0 RC1. • Элементарные функции вектора могут быть объявлены с помощью ___attributes___((vector)). Затем компилятор генерирует векторизованную версию скалярной функции, которую можно вызвать из векторизованного цикла.

1.3. OpenMP

OpenMP — это интерфейс прикладного программирования, поддерживающий мультиплатформенное многопроцессорное программирование с общей памятью на языках C, C++ и Fortran. OpenMP — важный инструмент, используемый во многих математических библиотеках для использования нескольких ядер на вычислительном узле с общей памятью. Максимальный параллелизм, который OpenMP будет стремиться использовать, часто устанавливается с помощью переменной среды. OMP_NUM_THREADS.

Чтобы позволить пользователю легко контролировать количество узлов, процессов и потоков OpenMP, которые использует его параллельная программа Matlab/Octave, система LLSC использует нашу многоядерную инфраструктуру запуска pMatlab и ее простой интерактивный синтаксис параллельного запуска.

Распараллеливание OpenMP на машине с сопроцессором Intel Xeon + Xeon Phi может применяться как на хосте, так и на устройстве с -openmp вариант компилятора. Он может работать на хосте Xeon, изначально на сопроцессоре Xeon Phi, а также в различных схемах разгрузки. Он представлен в коде обычными операторами прагмы для любого случая.

В приложениях, работающих как на хосте, так и на сопроцессоре Xeon Phi, потоки OpenMP не мешают друг другу, а операторы разгрузки выполняются на устройстве в зависимости от доступных ресурсов. Если на сопроцессоре Xeon Phi нет свободных потоков или их меньше, чем запрошено, параллельная область будет выполняться на хосте. Операторы прагмы разгрузки и операторы прагмы OpenMP независимы друг от друга и должны присутствовать в коде. В последней конструкции применяется обычная семантика общих и частных данных.

На каждом хост-процессоре Xeon доступно 16 потоков, а на каждом сопроцессоре Xeon Phi — в 4 раза больше ядерных потоков. Для схем разгрузки максимальное количество потоков, которые можно использовать на устройстве, равно 4-кратному количеству ядер минус одно, поскольку одно ядро зарезервировано для ОС и ее служб. Разгрузка на сопроцессор Xeon Phi может быть выполнена в любое время несколькими центральными процессорами до тех пор, пока не будут заполнены доступные ресурсы. Если 12 свободных потоков нет, задача, предназначенная для разгрузки, может быть выполнена на холсте.

1.3.1 Циклическое планирование

OpenMP допускает четыре различных вида планирования циклов — статическое, динамическое, управляемое и автоматическое. Таким образом можно контролировать количество итераций, выполняемых разными потоками. Расписание можно использовать для установки планирования цикла во время компиляции. Другой способ управления этой функцией — указать расписание (время выполнения) в своем коде и выберите планирование цикла во время выполнения, установив OMP_SCHEDULE переменная окружения.

1.3.2 Улучшение масштабируемости

Если объем работы, который должен быть выполнен каждым потоком, нетривиален и состоит из вложенных циклов for, можно использовать collapse() директива, указывающая, сколько циклов for связано с конструкцией цикла OpenMP. Это часто улучшает масштабируемость приложений OpenMP.

1.5 pyMIC

1.5.1Модуль разгрузки Python для Intel

Python один из наиболее популярных языков программирования в индустрии, использовать его достаточно легко, а скорость разработки на нём высока.

В последние годы Python привлекает особое внимание в сообществах специализирующихся на высокопроизводительных вычислениях или HPC (high performance computing).

Такие дополнения и библиотеки как Numpy и SciPy предоставляют эффективную реализацию ключевый алгоритмов и структур дынных по эффективности сравнимой такими языками как C или Fortran, что позволяет на ставить под сомнения Python как язык для высокопроизводительный вычислений.

Потребность HPC в скорости порождает необходимость в сопроцессорном оборудовании которое бы ускоряло многочисленные вычисления с плавающей точкой.

Вычислительные блоки общего назначения (GPGPU) и сопроцессор Intel Xeon Phi являются примерами дискретных плат расширения для получения дополнительной вычислительной мощности сверх ЦП.

Наиболее известным решением для выгрузки задач на Intel Xeon Phi является pyMIC. Intel Manycore Platform Stack (MPSS) поставляется с версией Python которая поддерживает работу с сопроцессором "из коробки". Через сетевой API можно подключиться к экземпляру Python выполняемому на сопроцессоре для выполнения вычисления на нём.

1.5.2Модуль pyMIC

Рассмотрим ключевые требования к модулю pyMIC, посмотрим на

внутренние части его интерфейса и разберемся как работает выгрузка из Python приложений.

Дизайн, ключевой принцип в дизайне pyMIC заключается в необходимости предоставить простой для использования и легкий в плане производительности интерфейс на уровне языка Python. В то же время программисты должны иметь полный контроль над данным которые они передают и выгружают для избежания потенциально неэффективного использования ресурсов или недетерминированности при исполнении выгруженного кода. По причине общей известности Numpy в качестве пакета для работы с многомерными наборами данных, pyMIC нацелен на работу с классом ndarray и его операторами.

Выгрузка базируется на программных компонентах Intel LEO и Intel Cilk. И хотя подход Intel LEO является очень прямым в вопросе выгрузки регионов кожа помеченных прагмами компилятора или директивами, подобный подход для Python потребует специализированного синтаксиса Python для того чтобы ставить в коде тэги на выгрузку. При том что этот подход кажется более правильным, pyMIC избегает запутанности в угоду простоте. Вместо этого перенимается модель выгрузки от Intel Cilk которая использует функции как основные элементы для выгрузки. Это обоснованно тем, что в случае Numpy большинство требовательных к вычислениям операций реализованы через вызовы функций. К тому же большинство кодов направленных на высокопроисводительные вычисления выставляют наружу свои функции или вызовы к библиотекам типа MKL. В то время как Intel Cilk использует разделенную виртуальную память для обработки пересылки данных незаметно, в pyMIC требуются явные преобразования для избежания скрытых оверхедов при пересылке которые программистам бывает трудно обнаружить.

2. ПРОГРАММНО-АППАРАТНАЯ ЧАСТЬ

2.1 Суперкомпьютер ВГУ

На факультете компьютерных наук имеется суперкомпьютер. Архитектура этого суперкомпьютера соответствует гибридным вычислительным системам, когда у нас имеется 1 процессоров и один или несколько сопроцессоров под его управлением. Подразумевается, что сопроцессоры работают под руководством центрального процессора, который решает задачи, которые не удается эффективно распараллелить на большое количество ядер, в то время как сопроцессор решает задачи массового параллелизма. В составе нашей машины процессоры intel xeon и по 2 сопроцессора на каждом узле. 20 Xeon, 14 Xeon phi и 12 nvideo tesla. В данном случае нас интересуют сопроцессоры intel xeon phi. Физически xeon phi является отдельным самодостаточным узлом, 61 ядро с 4 потока (244 в сумме) и 16 гигабайтами памяти, единственное что его отличается от самостоятельного хоста это отсутствие собственной памяти (жесткого диска).

3. ПРАКТИЧЕСКАЯ ЧАСТЬ.

3.1 Постановка задачи.

Необходимо организовать перемножение двух квадратных матриц на сопроцессоре Intel Xeon Phi в двух режимах работы: native и offload(с использование модуля pyMIC).

3.2 Основы перемножения матриц.

Пусть даны две прямоугольные матрицы *A* и *B* размерности *l* x *m* и *m* x *n* соответственно:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \cdots & a_{lm} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}.$$

Тогда матрица C размерностью *l* x *n:*

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \cdots & c_{ln} \end{bmatrix},$$

в которой:

$$c_{ij} = \sum_{r-1}^{m} a_{ir} b_{rj} \quad (i = 1, 2, \ldots l;\ j = 1, 2, \ldots n).$$

называется их произведением.

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первом сомножителе равно числу строк во втором; в этом случае говорят, что матрицы согласованы. В частности, умножение всегда выполнимо, если оба сомножителя — квадратные матрицы одного и того же порядка.

Таким образом, из существования произведения *AB* вовсе не следует

существование произведения *BA*.

Произведение матриц AB состоит из всех возможных комбинаций скалярных произведений вектор-строк матрицы A и вектор-столбцов матрицы B. Элемент матрицы AB с индексами i, j есть скалярное произведение i-ой вектор-строки матрицы A и j-го вектор-столбца матрицы B.

Иллюстрация справа демонстрирует вычисление произведения двух матриц *A* и *B*, она показывает как каждые пересечения в произведении матриц соответствуют строкам матрицы *A* и столбцам матрицы *B*. Размер результирующей матрицы всегда максимально возможный, то есть для каждой строки матрицы *A* и столбца матрицы *B* есть всегда соответствующее пересечение в произведении матрицы.

Значения на пересечениях, отмеченных кружочками, будут:

$$\begin{aligned} x_{1,2} &= (a_{1,1}, a_{1,2}) \cdot (b_{1,2}, b_{2,2}) \\ &= a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ x_{3,3} &= (a_{3,1}, a_{3,2}) \cdot (b_{1,3}, b_{2,3}) \\ &= a_{3,1}b_{1,3} + a_{3,2}b_{2,3} \end{aligned}$$

В общем случае, произведение матриц не является коммутативной операцией. К примеру:

$$\underset{3\times4\text{ matrix}}{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 & 2 & 3 & 4 \end{bmatrix}} \underset{4\times5\text{ matrix}}{\begin{bmatrix} \cdot & \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & b & \cdot \\ \cdot & \cdot & \cdot & c & \cdot \\ \cdot & \cdot & \cdot & d & \cdot \end{bmatrix}} = \underset{3\times5\text{ matrix}}{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & x_{3,4} & \cdot \end{bmatrix}}$$

элемент $x_{3,4}$ произведения матриц, приведённых выше,вычисляется следующим образом

$$x_{3,4} = (1, 2, 3, 4) \cdot (a, b, c, d) = 1 \cdot a + 2 \cdot b + 3 \cdot c + 4 \cdot d$$

Увидеть причины описанного правила матричного умножения легче всего, рассмотрев умножение вектора на матрицу.

Последнее естественно вводится исходя из того, что при разложении векторов по базису действие (любого) линейного оператора *A* даёт выражение компонент вектора **v'** = *A***v**:

$$v'_i = \sum_j A_{ij} v_j$$

То есть линейный оператор оказывается представлен матрицей, векторы — векторами-столбцами, а действие оператора на вектор — матричным умножением вектора-столбца слева на матрицу оператора (это частный случай матричного умножения, когда одна из матриц — вектор-столбец — имеет размер n x 1).

(Равно переход к любому новому базису при смене координат представляется полностью аналогичным выражением, только v'_i в этом случае уже не компоненты нового вектора в старом базисе, а компоненты старого вектора в новом базисе; при этом A_{ij} – элементы матрицы перехода к новому базису).

Рассмотрев последовательное действие на вектор двух операторов: сначала A, а потом B (или преобразование базиса A, а затем преобразование базиса B), дважды применив нашу формулу, получим:

$$v''_i = \sum_j B_{ij} \sum_k A_{jk} v_k = \sum_j \sum_k B_{ij} A_{jk} v_k = \sum_k \sum_j (B_{ij} A_{jk}) v_k,$$

откуда видно, что композиции BA действия линейных операторов A и B (или аналогичной композиции преобразований базиса) соответствует матрица, вычисляемая по правилу произведения соответствующих матриц:

$$(BA)_{ik} = \sum_j B_{ij} A_{jk}.$$

Определённое таким образом произведение матриц оказывается совершенно естественным и очевидно полезным (даёт простой и универсальный способ вычисления композиций произвольного количества линейных преобразований).

3.3 Программная реализация.

3.3.1 Запуск программ на сопроцессоре Intel Xeon Phi в режиме “native”.

Для тестирования работы сопроцессора были написан и распараллелен код перемножения двух квадратных матриц с использованием библиотеки openMP. Для того чтобы распараллелить задачи используется внешний цикл, который распределяет задачу по потокам. Код был скомпиллирован с помощью кросскомпилляции для архитектуры Intel Xeon Phi. Программа была перенесена на сопроцессор через ssh соединение и запущена в режиме native.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

const int N = 244;
void main()
{
   int i, j, k;
   int a[N][N];
   int b[N][N];
   int c[N][N];

   srand(time(0));
   omp_set_num_threads(N);
   #pragma omp for
   for(i = 0; i < N; i++)
   {
      for(j = 0; j < N; j++)
      {
         a[i][j] = rand();
         b[i][j] = rand();
      }
    }

   #pragma omp for
   for(i = 0; i < N; i++)
   {
      for(j = 0; j < N; j++)
      {
         for(k = 0; k < N; k++)
         {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
```

```
    }

    printf("The product of matrix a and matrix b is: \n");
    for(i = 0; i < N; i++)
    {
        for(j = 0; j < N; j++)
        {
            printf("%d\t", c[i][j]);
        }
        printf("\n");
    }
}
```

3.3.2 Запуск программ на сопроцессоре Intel Xeon Phi в режиме “offload” через модуль pyMIC.

Для запуска перемножения матриц с помощью модуля pyMIC необходимо собрать библиотечную функцию для перемножения матриц (matrix_multiplication.so). Чтобы генерировать эту библиотеку мы создали файл make

```
echo matrix_multiplication.c
icl -nologo -Qmic -I..\..\include -I"%MKLROOT%\include" -O2 -fPIC -shared -L"%MKLROOT%/lib/mic" -lmkl_intel_lp64 -lmkl_core -lmkl_intel_thread -lpthread -o matrix_multiplication.so matrix_multiplication.c
```

Код функции выглядит следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <pymic_kernel.h>

PYMIC_KERNEL
void multiplication(const double *A, const double *B, double *C, const long int *nrows, const long int *ncols)
{
    for(int i = 0; i < *nrows; i++)
    {
        for(int j = 0; j < *ncols; j++)
        {
            for(int k = 0; k < *ncols; k++)
            {
                *(C+i*10+j) += *(A+i*10+k) * *(B+k*10+j);
```

```
        }
      }
    }
}
```

В следующем коде вызывается функция multiplication для преумножения через модуль pymic:

```
from __future__ import print_function

import pymic as mic
import numpy as np
import sys
import time

# load the library with the kernel function (on the target)
device = mic.devices[0]
library = device.load_library("matrix_multiplication.so")

# use the default stream
stream = device.get_default_stream()

ds = 10
m, n, k = ds, ds, ds
if len(sys.argv) > 1:
    sz = int(sys.argv[1])
    m, n, k = sz, sz, sz

# construct some matrices
np.random.seed(10)
a = np.random.random(m * k).reshape((m, k))
b = np.random.random(k * n).reshape((k, n))
c = np.zeros((m, n))

# associate host arrays with device arrats
offl_a = stream.bind(a)
offl_b = stream.bind(b)
offl_c = stream.bind(c)

# convert a and b to matrices (eases MxM in numpy)
Am = np.matrix(a)
Bm = np.matrix(b)

# print the input
print("input:")
print("---------------------------------------")
print("a=", a)
print("b=", b)
print()
print()

# print the input of numpy's MxM if it is small enough
np_mxm_start = time.time()
Cm = Am * Bm
np_mxm_end = time.time()
```

```
print("numpy gives us:")
print("-------------------------------------")
print(Cm)
print("checksum:", np.sum(Cm))
print()
print()

print('++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++')

# invoke the offloaded dgemm
c[:] = 0.0
offl_c.update_device()
np_mic_start = time.time()
stream.invoke(library.multiplication, offl_a, offl_b, offl_c, m,k)
stream.sync()
np_mic_end = time.time()


offl_c.update_host()
stream.sync()
print(offl_c)
```

Список литературы

1. Best Practice Guide - Intel Xeon Phi / Nevena Ilieva, Michael Schliephake, Sami Saarinen, Volker Weinberg - LRZ Germany 2014.
2. Optimizing Xeon Phi for Interactive Data Analysis / Chansup Byun, Jeremy Kepner, William Arcand, David Bestor, William Bergeron, Matthew Hubbell, Vijay Gadepally, Michael Houle, Michael Jones, Anne Klein, Lauren Milechin, Peter Michaleas, Julie Mullen, Andrew Prout, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther - MIT Lincoln Laboratory Supercomputing Center, MIT Computer Science & AI Laboratory, MIT Mathematics Department, MIT Department of Earth, Atmospheric and Planetary Sciences
3. Evaluation of the Intel Xeon Phi offload runtimes for domain decomposition solvers / Lukas Maly, Jan Zapletala, Michal Mertaa, Lubomir Riha, Vit Vondraka - IT4Innovations, Dept. of Applied Mathematics, VŠB – Technical University of Ostrava CzechRepublic.
4. Intel Corporation. Intel R Manycore Platform Software Stack (MPSS), 2014. https://software.intel.com/en- us/articles/ intel- manycore- platform- software stack- mpss.
5. Intel Corporation. Intel R Xeon Phi™ Coprocessor System Software Developers Guide, 2014. Document number 328207-003EN.
6. C.J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire. Offload Runtime for the Intel R Xeon Phi™ Coprocessor. Technical report, Intel Corpora- tion, March 2013. Available at https://software.intel.com/en-us/articles/offload-runtime-forthe-intelr-xeon-phitm-coprocessor.
7. Williams, Virginia (2011), Multiplying matices in O(n2.3727 time)
8. Кибернетический сборник. Новая серия. Вып. 25. Сб. статей 1983 — 1985 гг.: Пер. с англ. — М.: Мир, 1988 — В.Б. Алекссев. Сложность умножения матриц.