Распараллеливание на графические процессоры тестов NAS NPB3.3.1 на языке Fortran DVMH*

В.Ф. Алексахин[1], В.А. Бахтин[1], О.Ф. Жукова[1], А.С. Колганов[1], В.А. Крюков[1],

Н.В. Поддерюгина[1], М.Н. Притула[1], О.А. Савицкая[1], А.В. Шуберт[2]

ФГБУН Институт прикладной математики им. М.В. Келдыша РАН[1], МГУ им. Ломоносова[2]

В докладе описываются преобразования последовательных версий тестов NAS из пакета NPB3.3.1 (EP, MG, BT, LU, SP) и спецификации их параллельного выполнения посредством DVMH-директив, необходимые для их эффективной работы на кластерах с графическими процессорами. Исследуется влияние различных вариантов распараллеливания на эффективность выполнения программ. Сравниваются характеристики тестов, разработанных на высокоуровневом языке Fortran-DVMH (далее FDVMH), с их реализацией на низкоуровневом языке OpenCL, выполненной исследователями из Сеульского национального университета.

1. Введение

В последнее время появляется много вычислительных кластеров с установленными в их узлах ускорителями. В основном, это графические процессоры компании NVIDIA. В 2012 году начинают появляться кластеры с ускорителями другой архитектуры – Xeon Phi от компании Intel. Так, в списке Top500 [1] самых высокопроизводительных суперкомпьютеров мира, объявленном в ноябре 2013 года, 53 машины имеют в своем составе ускорители, из них 39 машин имеют ускорители NVIDIA, 14 – Intel, 2 – AMD/ATI. Данная тенденция заметно усложняет процесс программирования кластеров, так как требует освоения на достаточном уровне сразу нескольких моделей и языков программирования. Традиционным подходом можно назвать использование технологии MPI для разделения работы между узлами кластера, а затем технологий CUDA (или OpenCL) и OpenMP для загрузки всех ядер центрального и графического процессоров.

С целью упрощения программирования распределенных вычислительных систем были предложены высокоуровневые языки программирования, основанные на расширении директивами стандартных языков, такие, как HPF [2], Fortran-DVM [3,4], C-DVM [3,5]. Также были предложены модели программирования и соответствующие основанные на директивах расширения языков для возможности использования ускорителей, такие, как HMPP [6], PGI Accelerator Programming Model [7], OpenACC [8], hiCUDA [9].

Распараллеливание на графических ускорителях (ГПУ) циклов без зависимостей, будь то ручное или с использованием высокоуровневых средств, обычно не вызывает больших идеологических трудностей, так как целевая массивно-параллельная архитектура хорошо подходит для их обработки. При распараллеливании циклов с зависимостями возникают проблемы, обусловленные ограниченной поддержкой синхронизации потоков выполнения на ГПУ и моделью консистентности глобальной памяти.

2. Обзор пакета тестов NASA NPB версии 3.3

NAS Parallel Benchmarks [13] — комплекс тестов, позволяющий оценивать производительность суперкомпьютеров. Они разработаны и поддерживаются в NASA Advanced Supercomputing (NAS) Division (ранее NASA Numerical Aerodynamic Simulation Program), расположенном в NASA Ames Research Center. В версии 3.3 пакет NPB включает в себя 11 тестов.

* Исследование выполнено при финансовой поддержке грантов РФФИ № 13-07-00580, 14-01-00109 и Программ фундаментальных исследований президиума РАН №15, №16 и №18.

В данной статье рассматривается распараллеливание на ГПУ тестов EP, MG, BT, LU, SP, для которых ранее были разработаны параллельные версии для кластеров на высокоуровневом языке Fortran-DVM:
MG (Multi Grid) – множественная сетка. Тест вычисляет приближенное решение трехмерного уравнения Пуассона («трехмерная решетка») в частных производных на сетке NxNxN с периодическими граничными условиями (функция на всей границе равна 0 за исключением заданных 20 точек). Размер сетки N определяется классом теста. Тестирует возможности системы выполнять как длинные, так и короткие передачи данных.
EP (Embarrassingly Parallel) – чрезвычайно параллельный. Тест вычисляет интеграл методом Монте-Карло – тест «усложненного параллелизма» для измерения первичной вычислительной производительности плавающей арифметики. Этот тест может быть полезен, если на кластере будут решаться задачи, связанные с применением метода Монте-Карло. В алгоритме также учитывается время на форматирование и вывод данных.
BT (Block Tridiagonal) – блочная трехдиагональная схема. Тест решает синтетическую систему нелинейных дифференциальных уравнений в частных производных (трехмерная система уравнений Навье-Стокса для сжимаемой жидкости или газа), используя блочную трехдиагональную схему с методом переменных направлений.
SP (Scalar Pentadiagonal) – скалярный пентадиагональный. Тест решает синтетическую систему нелинейных дифференциальных уравнений в частных производных (трехмерная система уравнений Навье-Стокса для сжимаемой жидкости или газа), используя скалярную пятидиагональную схему.
LU (Lower-Upper) – разложение при помощи симметричного метода Гаусса–Зейделя. Тест решает синтетическую систему нелинейных дифференциальных уравнений в частных производных (трехмерная система уравнений Навье-Стокса для сжимаемой жидкости или газа), используя метод симметричной последовательной верхней релаксации.

Разделим данные тесты на две группы: EP и MG – тесты, в которых не требуется распараллеливать циклы с регулярными зависимостями по данным (т.е. в FDVM-версиях этих тестов нет параллельных циклов со спецификациями ACROSS) и BT, SP, LU – тесты, в которых требуется распараллеливать циклы с регулярными зависимостями по данным (т.е. в FDVM-версиях этих тестов есть параллельные циклы со спецификациями ACROSS).

Характеристики полученных тестов после преобразования, с использованием FDVMH:

В тесте EP всего 1 параллельный цикл.

В тесте MG всего 15 тесно-гнездовых параллельных циклов.

В тесте BT всего 44 тесно-гнездовых параллельных циклов. Из них 6 циклов имеют зависимость по одному из трех измерений, причем зависимое измерение в различных циклах соответствует различным измерениям обрабатываемых массивов.

В тесте SP всего 25 тесно-гнездовых параллельных циклов. Из них 6 циклов имеют зависимость по одному из трех измерений, причем зависимое измерение в различных циклах соответствует различным измерениям обрабатываемых массивов.

В тесте LU всего 107 тесно-гнездовых параллельных циклов. Из них 2 цикла имеют зависимость по всем трем измерениям.

3. Реализация и особенности тестов EP и MG

3.1 Тест EP

В данном тесте порождаются пары псевдослучайных нормально распределенных чисел и вычисляются частоты их попадания в каждый из десяти выбранных полуинтервалов [k, k+1), где k меняется от 0 до 9. Тест содержит один единственный цикл, витки которого можно выполнять независимо друг от друга. При параллельном выполнении этого цикла на кластере никаких обменов между процессорами не требуется, а по окончанию его выполнения производится объединение результатов редукционной операции суммирования, накопленных на каждом процессоре в массиве из 10 элементов. Для эффективного отображения на ГПУ в DVM-программе параллельный цикл объявлен вычислительным регионом.

3.2 Тест MG

Тест MG реализует алгоритм многосеточного метода решения задачи Пуассона. В данном тесте содержатся два вида циклов, в которых сосредоточена основная вычислительная нагрузка. К первому виду относятся циклы проецирования на более грубую сетку – при спуске по V-циклу и аппроксимирования на уточненную сетку, при восхождении по V-циклу, ко второму – циклы интерполяции решения на основании невязки.

При вычислении значений в одной сетке по значениям другой сетки требуется согласованное распределение этих сеток между процессорами. Для этого в FDVM версии данная проблема решалась с помощью директивы REALIGN, которая позволяет перераспределить уже распределенный массив. Так, как переход от одной сетки к другой сетке происходит очень часто, то их перераспределение ведет к замедлению работы программы. Это замедление при работе на ГПУ оказалось очень значительным из-за низкой скорости обменов между памятью ЦПУ и памятью ГПУ (по сравнению со скоростью вычислений на ГПУ). Поэтому в FDVMH-версии было решено сократить количество операций REALIGN во время вычислений за счет использования дополнительной памяти для хранения нескольких экземпляров сетки с разным распределением ее элементов между процессорами. Для этого были использованы средства языка Fortran DVMH для работы с динамическими массивами и указателями.

3.3 Оптимизация циклов теста MG

В данном тесте четыре основных вычислительных цикла. Каждый из них построен по одинаковому принципу. Поэтому рассмотрим один цикл и его оптимизацию. Директива PARALLEL позволяет сопоставить витки тесно-гнездовых циклов элементам распределенных массивов. В данном случае тесно-гнездовые циклы будут исполняться параллельно (см. **Рис. 3.1**).

```
!DVM$ PARALLEL (i3,i2,i1) ON u(i1,i2,i3)
    do i3=2,n3-1)
      do i2=2,n2-1
        do i1=2,n1-1
          u(i1,i2,i3) = u(i1,i2,i3) + c(0) * r(i1,i2,i3) + c(1) *
  ( r(i1-1,i2,i3) + r(i1+1,i2,i3) + r(i1,i2-1,i3) + r(i1,i2+1,i3)
  + r(i1,i2,i3-1) + r(i1,i2,i3+1) )  + c(2) * (r(i1,i2-1,i3-1)
  + r(i1,i2+1,i3-1) + r(i1,i2-1,i3+1) + r(i1,i2+1,i3+1)
  + r(i1-1,i2-1,i3) + r(i1-1,i2+1,i3) + r(i1-1,i2,i3-1)
  + r(i1-1,i2,i3+1) + r(i1+1,i2-1,i3) + r(i1+1,i2+1,i3)
  + r(i1+1,i2,i3-1) + r(i1+1,i2,i3+1))
        enddo
      enddo
    enddo
```

Рис. 3.1. Цикл процедуры psinv до преобразований

Не трудно заметить, что, например, для каждой фиксированной итерации по i2 необходимы данные массива R по оставшимся не фиксированным индексам (i1, i3) и всем их комбинациям (i1±1, i3±1). В итоге, получается, что каждая нить, исполняющая виток цикла по i2, будет считывать повторяющиеся данные соседних нитей по индексам i1 и i3. Для повторного использования данных внесем цикл по i2 внутрь и изменим директиву следующим образом:

!DVM$ PARALLEL (i3, i1) ON u(i1, *, i3)

Для того чтобы повторно использовать значения в этом цикле, для каждой фиксированной пары (i1, i3) и (i1±1, i3±1) заведем три переменные, объявленные как приватные – r1, r1_p1 и r1_m1, которые будут соответствовать индексам i2, i2+1, i2-1. Далее необходимо считать данные в r1_m1 и r1 до основного цикла, добавить считывание в r1_p1 в цикле до вычислений, заменить соответствующие выражения в цикле на считанные, а после вычислений – сохранить значения r1 и r1_p1 в переменные r1_m1 и r1 соответственно для следующей итерации. После всех преобразований, в цикле будет 5 групп (см. **Рис. 3.2**).

```
!DVM$ PARALLEL (i3,i1) ON u(i1,*,i3)
!DVM$& ,PRIVATE (i2, r1,r1_m1,r1_p1, r2,r2_m1,r2_p1, r3,r3_m1,r3_p1,
```

```
r4,r4_m1,r4_p1, r5,r5_m1,r5_p1)
      do i3=2,n3-1
        do i1=2,n1-1
          r1_m1 = r(i1,1,i3) ! загрузка первых значений до цикла
          r1 = r(i1,2,i3)
          r2_m1 = r(i1-1,1,i3)
          r2 = r(i1-1,2,i3)
          r3_m1 = r(i1+1,1,i3)
          r3 = r(i1+1,2,i3)
          r4_m1 = r(i1,1,i3+1)
          r4 = r(i1,2,i3+1)
          r5_m1 = r(i1,1,i3-1)
          r5 = r(i1,2,i3-1)

          do i2=2,n2-1
               r1_p1 = r(i1,i2+1,i3) ! загрузка следующих значений
               r2_p1 = r(i1-1,i2+1,i3)
               r3_p1 = r(i1+1,i2+1,i3)
               r4_p1 = r(i1,i2+1,i3+1)
               r5_p1 = r(i1,i2+1,i3-1)

               u(i1,i2,i3) = u(i1,i2,i3)+
               c_0 * r1 + c_1 * (r2 + r3 + r1_m1 + r1_p1 + r4 + r5) +
               c_2 * (r4_m1 + r4_p1 + r5_m1 + r5_p1 + r2_m1 + r2_p1 +
               r(i1-1,i2,i3-1) + r(i1-1,i2,i3+1) + r3_m1 + r3_p1 +
               r(i1+1,i2,i3-1) + r(i1+1,i2,i3+1))

               r1_m1 = r1  ! сохранение считанных значений
               r1 = r1_p1
               r2_m1 = r2
               r2 = r2_p1
               r3_m1 = r3
               r3 = r3_p1
               r4_m1 = r4
               r4 = r4_p1
               r5_m1 = r5
               r5 = r5_p1
          enddo

        enddo
      enddo
```

Рис. 3.2. Цикл процедуры psinv после преобразований

Тем самым количество необходимых чтений из глобальной памяти существенно сокращается за счет сохранения уже считанных значений в цикле i2 при переходе с одной итерации на другую. На классе C время выполнения программы сократилось на 50% по сравнению с программой без данных оптимизаций, однако время выполнения программы на ЦПУ стало значительно хуже.

4. Реализация и особенности тестов BT, SP, LU

4.1 Обзор типов зависимостей

Для лучшего представления характера зависимости по данным абстрагируемся от алгоритма каждого из тестов и рассмотрим типовые программы, а именно – метод переменных направлений, который используется в тестах BT и SP и метод последовательной верхней релаксации (Successive over-relaxation - SOR), который используется в тесте LU.

Ниже представлен пример программы для метода переменных направлений (см. **Рис. 4.1**).

```
       program adi
      parameter(nx=400,ny=400,nz=400,maxeps=0.01,itmax=100)
      integer nx,ny,nz,itmax
      double precision eps,relax,a(nx,ny,nz)
      call init(a,nx,ny,nz)
      do it = 1,itmax
       eps=0.D0
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i-1,j,k) + a(i+1,j,k)) / 2
         enddo
        enddo
       enddo
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i,j-1,k) + a(i,j+1,k)) / 2
         enddo
        enddo
       enddo
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          eps = max(eps, abs(a(i,j,k) - a(i,j,k-1)+a(i,j,k+1)) / 2))
          a(i,j,k) = (a(i,j,k-1)+a(i,j,k+1)) / 2
         enddo
        enddo
       enddo
      if(eps.lt.maxeps) goto 3
      enddo
3     continue
      end
```

Рис. 4.1. Реализация метода переменных направлений

В данной программе три тесно-гнездовых цикла и в каждом из них есть зависимость по одному из трех измерений. В первом гнезде циклов самый внутренний цикл может быть выполнен только последовательно, так как существует зависимость по данным массива A. В связи с этим – при выполнении данного цикла на ГПУ – время чтения из памяти будет большим, так как данные для параллельных витков оставшихся двух циклов будут расположены не подряд. Одним из способов решения данной проблемы является перестановка местами двух первых измерений массива A. Но тогда похожая проблема возникнет во втором гнезде циклов. Решить проблему можно аналогичным образом, например, поменяв местами первые два измерения обратно. Следовательно, нельзя подобрать начальное расположение массива таким, чтобы все три гнезда циклов выполнялись максимально эффективно.

Рассмотрим пример другой программы, реализующий метод последовательной верхней релаксации (см. **Рис. 4.2**). В данной программе всего одно гнездо циклов, в котором происходят вычисления. В отличие от метода переменных направлений – в данном методе основной цикл имеет зависимость по всем своим измерениям, что приводит к значительным трудностям при распараллеливании даже в модели OpenMP.

```
            program  sor
            parameter(n1=1000,n2=1000,n3=1000,itmax=100,
            maxeps=0.5e-6,w=0.5)
            real a(n1,n2,n3), eps, w
            integer  itmax

            call init(a, n1, n2, n3)
            do it = 1,itmax
```

```
           eps = 0.
          do k = 2, n3-1
           do j = 2, n2-1
            do i = 2, n1-1
             s = a(i,j,k)
             a(i,j,k) = (w/4)*( a(i-1,j,k)+a(i+1,j,k)+
               a(i,j-1,k)+a(i,j+1,k)+ a(i,j,k-1)+a(i,j,k+1))
               + (1-w)*a(i,j,k)
             eps = max(eps, abs(s - a(i,j,k)))
            enddo
           enddo
          enddo
           if (eps .lt. maxeps) goto 4
          enddo
    4     continue
          end
```

Рис. 4.2. Реализация метода последовательной верхней релаксации

4.2 Алгоритм отображения циклов с зависимостями в DVM-системе

Для отображения данных циклов в DVM-системе применяется следующий алгоритм [12]. Пространством витков данного цикла назовем множество кортежей всех принимаемых значений индексных переменных цикла. В рассматриваемом цикле есть прямая и обратная зависимость по измерениям I, J и K, следовательно, его пространство витков не может быть отображено на блок нитей ГПУ, так как все нити исполняются независимо. Одним из известных методов отображения подобных циклов является метод гиперплоскостей. Все элементы, лежащие на гиперплоскости, могут быть вычислены независимо друг от друга.

При таком порядке выполнения витков цикла возникает проблема эффективного доступа к глобальной памяти в силу того, что параллельно обрабатываются не соседние элементы массивов, что приводит к значительной потере производительности (примерно в 10 раз). Причем в отличие от первой рассмотренной программы – задать первоначальное расположение массива будет еще сложнее, так как элементы вычисляются по гиперплоскостям. Наличие в программе циклов с зависимостями более чем по одному измерению и циклов без зависимостей усложняют задачу, так как для эффективного исполнения потребуется разное расположение данных для таких циклов.

Для того чтобы избавить программиста от выше описанных сложностей, в языке FDVMH существуют следующие возможности: поддержка циклов с зависимостями указанием спецификации ACROSS в директиве PARALLEL и динамическое переупорядочивание массивов.

4.3 Спецификация ACROSS

В основе реализации спецификации ACROSS – описанный выше метод гиперплоскостей. Для того чтобы указать, что цикл имеет зависимость, необходимо добавить в спецификацию ACROSS список массивов, по которым есть зависимость (см. **Рис. 4.3**).

```
!пример для метода переменных направлений
      !DVM$ PARALLEL (k,j,i) on A(i,j,k), ACROSS(A(1:1,0:0,0:0))
      do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i-1,j,k) + a(i+1,j,k)) / 2
         enddo
        enddo
       enddo
!пример для метода SOR
      !DVM$ PARALLEL (K,J,I) on A(I,J,K), ACROSS(A(1:1,1:1,1:1))
      !DVM$& ,PRIVATE(s), REDUCTION(MAX(eps))
            do k = 2, n3-1
```

```
          do j = 2, n2-1
           do i = 2, n1-1
            s = a(i,j,k)
            a(i,j,k) = (w/4)*(
>                 a(i-1,j,k)+a(i+1,j,k)+
>                 a(i,j-1,k)+a(i,j+1,k)+
>                 a(i,j,k-1)+a(i,j,k+1)+
>                         ) + (1-w)*a(i,j,k)
            eps = max(eps, abs(s - a(i,j,k)))
           enddo
          enddo
         enddo
```

Рис. 4.3. Расстановка спецификации ACROSS для двух методов

4.4 Механизм динамического переупорядочивания массивов в DVMH

Для оптимизации доступа к глобальной памяти ГПУ в системе поддержки выполнения DVMH-программ был реализован механизм динамического переупорядочивания массивов, который перед выполнением на ГПУ параллельного цикла проверяет соответствие измерений цикла и измерений используемых в нем массивов, и, при необходимости, переупорядочивает некоторые массивы таким образом, чтобы доступ к элементам осуществлялся наилучшим образом — соседние нити блока работали бы с соседними ячейками глобальной памяти.

Данный механизм осуществляет любую необходимую перестановку измерений массива, а также так называемую диагональную трансформацию, в результате которой соседние элементы на диагоналях (в плоскости необходимых двух измерений) располагаются в соседних ячейках памяти, что позволяет применять технику выполнения цикла с зависимостями по гиперплоскостям без значительной потери производительности на операциях доступа к глобальной памяти ГПУ.

4.5 Преобразования исходных текстов программ

В тестах BT и SP используется метод переменных направлений. В тесте LU используется метод SSOR – симметричной последовательной верхней релаксации, состоящий из двух трехмерных циклов с положительными и отрицательными шагами с тремя зависимыми измерениями.

Для того чтобы распараллелить циклы с помощью FDVMH, необходимо преобразовать программу таким образом, чтобы циклы стали тесно-гнездовыми. Для этого, например, в LU, были подставлены процедуры blts и buts, чтобы получились трехмерные тесно-гнездовые циклы. В SP и BT были объединены циклы в процедуре compute_rhs для более эффективного выполнения. В процедурах x_solve, y_solve и z_solve, реализующих метод переменных направлений, были расширены временные массивы на одно измерение, чтобы трехмерные циклы можно было выполнять на ГПУ.

4.6 Оптимизации полученных программ

Программист, используя высокоуровневый язык FDVMH, оперирует с последовательной версией программы. Следовательно, необходимо уметь писать «хорошую» последовательную программу, чтобы из нее получалась эффективная параллельная программа. Знание конечной архитектуры может быть полезным при выполнении оптимизаций.

Современные ЦПУ имеют трехуровневый кэш. Размер кэша первого уровня равен 64КБ и содержится на всех вычислительных ядрах процессора. Размер кэша второго уровня варьируется от 1 до 2МБ. Кэш третьего уровня является общим для всего ЦПУ и имеет размер порядка 12-15МБ.

Современные ГПУ имеют двухуровневый кэш. Размер кэша первого уровня равен 64КБ. Используется для разделяемой памяти и вытеснения регистров. Для разделяемой памяти доступно не более 48КБ. Содержится в каждом вычислительном блоке. Максимальный размер

кэша второго уровня составляет 1,5МБ и является общим для всего ГПУ. Используется для кэширования данных, загружаемых из глобальной памяти ГПУ. В самом современном чипе ГПУ GK110 имеется 15 вычислительных блоков. Получается, что на один блок приходится примерно 48КБ кэша первого уровня и 102КБ кэша второго уровня. По сравнению с ЦПУ, это очень мало, поэтому операции чтения из глобальной памяти графического процессора дороже, чем из оперативной памяти центрального процессора. Все оптимизации будут направлены на уменьшение количества чтений из глобальной памяти и на увеличение вычислительной нагрузки, тем самым повышается соотношение количества вычислительных операций к операциям чтения из глобальной памяти ГПУ.

4.6.1 Тест LU

Основная вычислительная нагрузка сосредоточена в четырех циклах процедур blts и bult. В каждой из процедур сначала происходит инициализация четырех массивов типа double precision, размер которых равен 25 * $Class^3$, где Class – размер класса рассматриваемой задачи (например, для класса C Class = 162). В итоге, для класса C необходимо около 3Гб памяти. Так как после инициализации следующее гнездо циклов имеет зависимость по трем измерениям, то по описанному выше принципу работы DVMH-программы, произойдет диагонализация данных массивов. Так как массивы после инициализации используются только в процедурах blts и bult, выражения для инициализации элементов массивов были подставлены непосредственно в тело гнезда циклов с зависимостью, а сами массивы удалены, что привело к перевычислению значений данных массивов на каждом витке. Тем самым для выполнения данного теста на классе C требуется на 3Гб меньше памяти, что позволило запустить эту программу на ГПУ Tesla c2050.

Еще одно преобразование, примененное в данном тесте, – использование приватных переменных для часто используемых элементов массивов, которые отображаются компилятором NVCC на регистры. Элемент массива часто используется, если количество, прежде всего, записей и чтений больше, чем количество чтений для его загрузки на регистры и записью для сохранения. Для пояснений рассмотрим следующий фрагмент программы (см. **Рис. 4.4**).

```
!DVM$ PARALLEL (k,j,i) on u(i,j,k,*), PRIVATE(m1,m2,m)
do k = 2,nz-1
  do j = 2,ny-1
   do i = 2,nx-1
    m1 = 2
    m2 = 3
    do m = 1,5
      u(i,j,k,m) = u(i,j,k,m1) + u(i,j,k,m2)
    enddo
    do m = 1,5
      u(i,j,k,m) = u(i,j,k,m1+1) + u(i,j,k,m2+1)
    enddo
    do m = 1,5
      u(i,j,k,m) = u(i,j,k,m1-1) + u(i,j,k,m2-1)
    enddo
   enddo
  enddo
 enddo
```

Рис. 4.4. Исходный цикл

В данном фрагменте массив U используется на чтение и запись, причем по последнему измерению массив не распределен. Всего делается 15 записей и 45 чтений. А для загрузки на регистры необходимо 5 чтений и столько же потребуется для сохранения результата. В итоге, в текст программы необходимо добавить два цикла по загрузке и сохранению массива, а также заменить все обращения в теле цикла новой переменной (см. **Рис. 4.5)**

```
!DVM$ PARALLEL (k,j,i) on u(i,j,k,*), PRIVATE(m1,m2,m,u_)
do k = 2,nz-1
  do j = 2,ny-1
```

```
        do i = 2,nx-1
          do m=1,5
            u_(m) = u(i,j,k,m)
          enddo
         m1 = 2
         m2 = 3
         do m = 1,5
           u_(m) = u_(m1) + u_(m2)
         enddo
         do m = 1,5
           u_(m) = u_(m1+1) + u_(m2+1)
         enddo
         do m = 1,5
           u_(m) = u_(m1-1) + u_(m2-1)
         enddo
          do m=1,5
            u(i,j,k,m) = u_(m)
          enddo
        enddo
      enddo
    enddo
```

Рис. 4.5. Применение оптимизации – замена массива u массивом u_

4.6.2 Тест BT

В данном тесте, как было описано выше, самая большая вычислительная нагрузка содержится в трех процедурах – x_solve, y_solve, z_solve. Так как данные процедуры отличаются только зависимым измерением (x, y и z соответственно), то оптимизации, применяемые к одной из них, будут так же применены и ко всем остальным. Рассматривая процедуру x_solve, можно заметить, что используется большой временный массив lhs типа double precision, размер которого равен 75 * $Class^3$, где Class – класс задачи (например, для класса C Class = 162). Тем самым, данный массив на классе C занимает примерно 2,3 Гб памяти. Получаем те же самые проблемы, что и в тесте LU: нехватка памяти и долгое время динамического переупорядочивания. Решить проблему можно описанным выше способом – сокращением длины массива за счет повторной инициализации элементов массива непосредственно в теле цикла процедуры x_solve. Таким образом, в силу специфики задачи, вспомогательный массив можно сократить на одно измерение, а именно в число раз, соответствующее классу задачи. Для класса C массив сократится в 162 раза и будет занимать около 15мб.

4.6.3 Тест SP

Так как данный тест отличается от BT алгоритмом, используемым в процедуре x_solve, а характер зависимости такой же, то все описанные оптимизации применяются аналогично. Стоит добавить, что изначально используется временный массив lhs типа double precision, размера 15 * $Class^3$. Для класса C массив занимает примерно 500мб и сократить его в данной программе нельзя.

5. Полученные результаты. Сравнение с OpenCL

Для оценки эффективности распараллеливания с помощью FDVMH использовались две версии каждой из программ: первоначальный последовательный вариант, в котором никаких преобразований не делалось и преобразованный и оптимизированный вариант с директивами FDVMH. Тестирование производилось на суперкомпьютере K100 [14], имеющем процессоры Intel Xeon X5670 и ГПУ NVIDIA Tesla C2050 с архитектурой Fermi и на сервере с процессорами Intel core i7 и ГПУ NVIDIA GeForce GTX Titan с самой новой архитектурой Kepler. Для

сравнения, были получены времена выполнения этих тестов, реализованных на низкоуровневом языке OpenCL исследователями из Сеульского национального университета [11].

Ниже (см. **Таблица 5.1**; **Рис. 5.1**) приведены результаты сравнения эффективности распараллеливания тестов BT, LU, SP, MG, EP на классах A, B, C. На **Рис. 5.1** показано ускорение FDVMH версий тестов по сравнению с OpenCL. В **Таблице 5.1** указаны времена и ускорения FDVMH и OpenCL-программ по отношению к исходным последовательным версиям тестов, исполненные на одном ядре процессора Intel Xeon X5670. Прочерком в ячейках таблицы обозначены те варианты запусков, для которых не хватило памяти на ГПУ. Стоит отметить, что реализация теста BT на классе C OpenCL версии требует более 6Гб памяти (в 4 раза больше, чем требуется FDVMH-программе), что не позволило сравнить OpenCL и FDVMH версии этого теста.

Таблица 5.1. Эффективность распараллеливания тестов

Задача		CPU, Xeon X5670	FORTRAN DVMH				OpenCL			
			Tesla C2050 (с ECC)		GeForce GTX Titan (без ECC)		Tesla C2050 (с ECC)		GeForce GTX Titan (без ECC)	
Тест	Класс	Время, секунд	Время, секунд	Ускорение	Время, секунд	Ускорение	Время, секунд	Ускорение	Время, секунд	Ускорение
BT	A	52,69	15,63	3,37	5	10,54	75,5	0,7	22,41	2,35
	B	221,9	61,74	3,59	18,7	11,87	272,42	0,81	71,7	2,97
	C	951,0	192,29	4,94	56,86	16,73	-	-	-	-
SP	A	36,6	9,82	3,73	3,23	11,33	22,95	1,59	6,09	6,01
	B	154,7	33,4	4,63	11,37	13,61	86,35	1,79	28,53	5,42
	C	637,73	117,55	5,43	35	18,22	433,77	1,47	137,7	4,63
LU	A	40,31	7,0	5,76	5,14	7,84	8,86	4,55	4,49	8,9
	B	170	21,0	8,10	12,51	13,59	30,23	5,62	13,49	12,6
	C	779	70,5	11,05	37,46	20,80	127,95	6,09	44,65	17,4
MG	A	1,41	0,18	7,8	0,11	12,8	0,20	7,05	0,09	20
	B	6,62	0,82	8,07	0,50	13,24	0,94	7,04	0,42	15,7
	C	55,17	5,62	9,8	3,08	17,9	7,83	7,04	3,71	14,8
EP	A	7,97	0,24	33,2	0,38	20,9	0,25	31,88	0,43	18,5
	B	31,85	0,73	43,6	1,12	28,4	0,82	38,8	1,26	25,2
	C	55,17	2,68	20,5	4,14	13,3	2,89	19	4,51	12,2

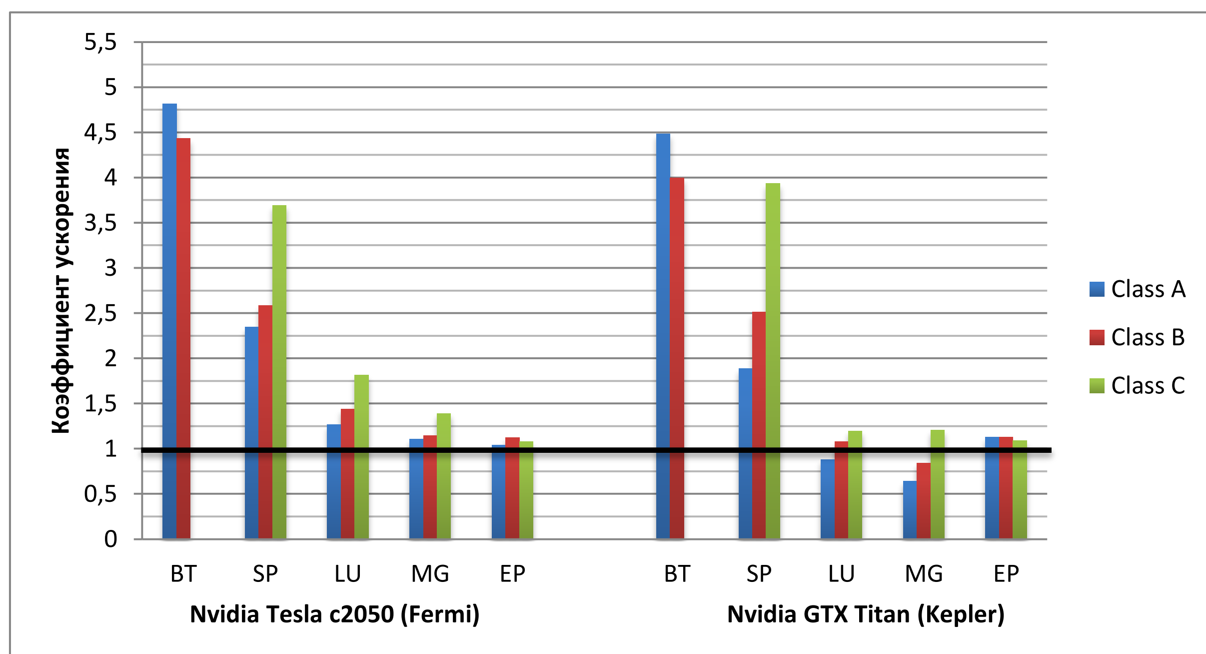

Рис. 5.1. Ускорение DVHM-программ по сравнению с OpenCL-программами.

В **Таблице 5.2** указано количество строк и слов рассматриваемых программ, а так же количество используемой памяти. Для DVMH в некоторых случаях необходимо большее количество памяти для работы механизма динамического переупорядочивания массивов. В таблице количество памяти для DVMH-программ состоит из количества памяти без использования динамического переупорядочивания и количества памяти, необходимое для динамического переупорядочивания.

Таблица 5.2. Использование памяти и количества строк, слов

Задача		Исходная версия		DVMH версия		OpenCL версия	
Тест	Класс	память в Гб	количество строк, слов	память в Гб	количество строк, слов	память в Гб	количество строк, слов
BT	A	0,041	4092, 13588	0,088+0,044	3478, 18427	0,346	14678, 55932
	B	0,166		0,356+0,175		1,2	
	C	0,665		1,425+0,79		> 6	
SP	A	0,043	3469, 10547	0,063+0,005	3437, 10744	0,154	11179, 41714
	B	0,174		0,253+0,04		0,340	
	C	0,698		1,014+0,158		1,38	
LU	A	0,036	4148, 18173	0,039+0,005	2682, 15772	0,116	9439, 41426
	B	0,142		0,158+0,04		0,249	
	C	0,558		0,634+0,158		0,776	
MG	A	0,440	1686, 5028	0,518	2315, 7713	0,503	4030, 13791
	B	0,440		0,518		0,503	
	C	3,31		3,50		3,55	
EP	A	0,07	659, 2614	0,1	704, 2865	0,1	928, 3792
	B	0,07		0,1		0,1	
	C	0,07		0,1		0,1	

6. Заключение

В результате проделанной работы были получены оптимизированные и распараллеленные с помощью FDVMH тесты LU, BT, SP, MG и EP. Трудоемкость распараллеливания с помощью FDVMH и OpenCL можно грубо оценить количеством добавленных строк или слов в исходную программу. В **Таблице 5.2** можно увидеть, что размер полученных DVMH-программ отличается от исходных последовательных не более чем на 45%. Размер OpenCL-программ отличается в 2-3 раза. В тестах с зависимостями эффективность OpenCL-программ крайне мала. Эффективность распараллеливания FDVMH- версии теста EP незначительно выше эффективности распараллеливания OpenCL версии этого теста. Эффективность распараллеливания DVMH-версии теста MG выше эффективности распараллеливания OpenCL версии этого теста на 50%, если использовать архитектуру Fermi, и на 25%, если использовать архитектуру Kepler, на классе C. Если рассматривать ресурсоемкие задачи (класс C), то FDVMH-программы показали большую эффективность, чем OpenCL на всех рассматриваемых тестах.

В дальнейшем планируется распараллелить с помощью FDVMH остальные тесты и сравнить их с реализациями на OpenCL и OpenACC. Планируется также исследование эффективности распараллеливания тестов на платформе Intel Xeon Phi.

Литература

1. Top500 List – November 2013 | TOP500 Supercomputer Sites
   URL: http://top500.org/list/2013/11/ (дата обращения 24.11.2013)

2. High Performance Fortran
   URL: http://hpff.rice.edu/ (дата обращения 01.12.2013)

3. Н.А. Коновалов, В.А. Крюков, А.А. Погребцов, Н.В. Поддерюгина, Ю.Л. Сазанов. Параллельное программирование в системе DVM. Языки Fortran-DVM и C-DVM. Труды Между-

народной конференции "Параллельные вычисления и задачи управления" (PACO’2001) Москва, 2-4 октября 2001 г., с. 140-154

4. Н.А. Коновалов, В.А. Крюков, С.Н. Михайлов, А.А. Погребцов. “Fortran DVM - язык разработки мобильных параллельных программ”, "Программирование", № 1, 1995, стр 49-54.
5. Н.А. Коновалов, В.А. Крюков, Ю.Л. Сазанов. C-DVM - язык разработки мобильных параллельных программ. "Программирование", № 1, 1999, стр 54-65.
6. Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP™: A Hybrid Multi-core Parallel Programming Environment.
7. OpenACC
URL: http://www.openacc-standard.org/ (дата обращения 24.11.2013)
8. T.D. Han and T.S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 1, pp. 78-90, Jan. 2011
9. В.А. Бахтин, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами. – Вестник Южно-Уральского государственного университета, серия "Математическое моделирование и программирование", №18 (277), выпуск 12 – Челябинск: Издательский центр ЮУрГУ, 2012, с. 82-92
10. Pennycook S.J., Hammond S.D., Jarvis S.A., Mudalige G.R. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark. ACM SIGMETRICS Performance Evaluation Review – Special issue on the 1st international workshop on performance modeling, benchmarking and simulation of high performance computing systems (PMBS 10). 2011. Vol. 38, Issue 4. P. 23–29.
11. Seo S., Jo G., Lee J. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. 2011 IEEE International Symposium on. Workload Characterization (IISWC). 2011. P. 137–148.
12. В.А. Бахтин, А.С. Колганов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула. Отображение на кластеры с графическими процессорами циклов с зависимостями по данным в DVMH-программах. // Труды Международной суперкомпьютерной конференции «Научный сервис в сети Интернет: все грани параллелизма», сентябрь 2013 г., г. Новороссийск. – М.: Изд-во МГУ, 2013, с. 250-257
13. NAS Parallel Benchmarks
URL: http://www.nas.nasa.gov/publications/npb.html (дата обращения 25.11.2013)
14. Гибридный вычислительный кластер К-100
URL: http://www.kiam.ru/MVS/resourses/k100.html (дата обращения 25.11.2013)