Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели

Андрей Белеванцев <abel@ispras.ru>, Алексей Кравец <kayrick@ispras.ru>,
Александр Монаков <amonakov@ispras.ru>

Аннотация. В данной работе предлагается способ автоматической генерации кода для стандарта OpenCL из гнезд циклов без зависимостей по данным между итерациями для программ на языках Си, Си++ и Фортран. Для генерации используется инфраструктура GRAPHITE компилятора GCC, использующая полиэдральную модель для анализа пространства итераций и пространства данных цикла. Описывается выполненная реализация и проведенные эксперименты, показывающие наилучшие результаты на вычислительных программах, основную часть которых составляют гнезда циклов.

Ключевые слова: оптимизации программы, полиэдральная модель, GCC, OpenCL**.**

1. Введение

Одной из проблем параллельных вычислений являлось отсутствие единого стандарта, охватывающего как многоядерные процессоры, так и графические ускорители. Для многопоточных приложений использовалось OpenMP или POSIX Threads, для графических ускорителей – nVidia CUDA[10-12].

Для решения данной проблемы была разработана программная модель OpenCL [1, 4, 6], которая позволяет программисту писать параллельный код для многоядерных процессоров и графических ускорителей (в дальнейшем будем называть такие устройства акселераторами). Несмотря на то, что данная модель становится все более популярной, в GCC на данный момент отсутствует возможность использования OpenCL в качестве целевого языка для генерации параллельного кода.

Целью данной работы является реализация преобразования гнезд циклов без зависимостей по данным в OpenCL код с использованием инфраструктуры GRAPHITE (GIMPLE Represented as Polyhedra with Interchangeable Envelopes) [7]. Преобразование гнезда циклов заключается в замене кода гнезда, выполняющегося на процессоре, на вызов функции-ядра OpenCL, выполняющей соответствующие вычисления на некотором акселераторе. Так же необходима вставка вызовов всех требуемых для работы OpenCL функций (создание контекста выполнения, создание и компиляция OpenCL-ядер, обмен

данными между центральным процессором и акселератором). Полученная в результате программа может быть выполнена на любом акселераторе, поддерживающем стандарт OpenCL, при наличии соответствующих библиотек времени выполнения.

2. Программная модель OpenCL

Программная модель OpenCL позволяет программисту описывать функции, которые будут параллельно выполнены на некотором акселераторе или наборе акселераторов, доступных на данной машине. Для задания конкретного акселератора служит понятие контекста, который задает класс требуемого устройства (многоядерный процессор, графический процессор) и понятие очереди команд, которая соответствует определенному акселератору.

В основе программной модели OpenCL лежит понятия ядра (kernel). Ядро — это функция, которая будет выполнена параллельно на акселераторе определенным количеством потоков Для того, чтобы отличать ядра-функции и ядра процессора в последующем изложении, понятие ядра всегда используется в смысле функции. Ядра определяются программистом в виде функций на некотором расширении языка Си. Создание ядра разделено на несколько этапов, каждому из которых отвечает вызов соответствующей функции OpenCL на центральном процессоре. В начале требуется преобразовать исходный код одного или нескольких ядер, заданный в виде массива строк, в OpenCL программу, далее полученная программа должна быть скомпилирована для нужного устройства. После компиляции OpenCL программы из нее могут быть получены ядра, отвечающие функциям в ее исходном коде.

Для запуска ядра на акселераторе необходимо указать его аргументы. Аргументами ядра могут быть как скалярные значения, так и адреса буферов в памяти акселератора. Так как в общем случае акселератор не имеет прямого доступа к памяти центрального процессора (например, когда акселератором является графический ускоритель), ядра работают с адресами в памяти акселератора. OpenCL предоставляет механизм выделения буферов памяти на акселераторе и обмена данными между памятью центрального процессора и акселератора.

Запуск нужного ядра с указанием требуемого количества потоков осуществляется с помощью вызова функции библиотеки OpenCL на центральном процессоре. Акселератор, на котором будет выполняться ядро задается очередью команд, которой передается запрос на выполнение ядра. Таким образом, программа состоит из последовательного кода, который выполняется на центральном процессоре (пользовательский код и вызовы библиотеки OpenCL), и параллельного кода, которые выполняется на акселераторе (ядра).

3. Преобразование гнезд циклов в ядра OpenCL

Получение гнезд циклов, над которыми будет производиться преобразование, выполняется в рамках инфраструктуры GRAPHITE компилятора GCC. Результат работы GRAPHITE может быть получен в виде набора SCoP-ов (Static Control Part), каждый из которых представлен в виде CLAST (структуры данных, хранящей SCoP в виде операторов циклов, условных операторов и базовых блоков). Так же в процессе работы GRAPHITE для всех циклов, входящих в каждый SCoP, производится анализ на наличие зависимостей между итерациями. При этом необходимо, чтобы для всех обращений к данным внутри SCoP индексная функция была бы аффинной, а все циклы, входящие в него имели бы аффинные границы и константные шаги. Полученные гнезда циклов могут быть восстановлены как в виде циклов на центральном процессоре, так и в виде OpenCL ядер[13]. Ниже приведено описание алгоритма восстановления GIMPLE из CLAST в процессе которого некоторые гнезда циклов будут заменены на запуск OpenCL ядер на акселераторе.

3.1. Определение целевых гнезд циклов

В первую очередь необходимо определить, какие гнезда циклов будут преобразованы в ядра OpenCL. Данные гнезда должны удовлетворять определенному набору требований, которые можно разделить на две основные группы – требования корректности работы и требования скорости работы. Требования корректности работы включают в себя требование отсутствия зависимостей по данным между ядрами, выполняющимися параллельно, и синхронизацию буферов в памяти акселератора и центрального процессора.

Для проверки наличия зависимостей между итерациями рассматриваются все обращения к данным в гнезде циклов, кроме обращения к тем данным, которые могут быть приватизированы в теле ядра. Наличие зависимостей между итерациями цикла означает, что цикл будет выполняться на центральном процессоре, но в его теле могут содержаться циклы без зависимостей, которые могут быть выполнены на акселераторе. Таким образом, для определения гнезда циклов, которое будет преобразовано в ядро, выполняемое на акселераторе, в первую очередь необходимо выделить идеальные гнезда циклов, между итерациями которых нет зависимостей. Такое гнездо циклов может быть замещено параллельным выполнением его тела на акселераторе без нарушения требований корректности.

Для определения требований к скорости работы воспользуемся формулой Амдаля. При этом необходимо учитывать, что накладные расходы на запуск ядра являются последовательным кодом. Рассмотрим гнездо циклов *N*, которое может быть преобразовано в ядро, выполняемое на акселераторе (то есть, весь код данного ядра является параллельным). Пусть время, затрачиваемое на выполнение данного цикла на центральном процессоре

равно $T_{base}(N)$, время, затрачиваемое на накладные расходы на запуск ядра, равно $T_{opencl}(N)$. Тогда, по закону Амдаля, ускорение, получаемое при выполнении данного гнезда циклов на акселераторе, не превышает

$$\frac{1}{\frac{T_{opencl}(N)}{T_{base}(N)+T_{opencl}(N)}+\frac{T_{base}(N)}{(T_{base}(N)+T_{opencl}(N))*p}}$$

где p - число вычислительных элементов акселератора. Выигрыш от замены гнезда циклов *N* на запуск ядра будет в том случае, когда базовое время работы превышает время, полученное с помощью закона Амдаля.

$$T_{base} > T_{opencl} + \frac{T_{base}}{p} \quad (1)$$

При этом, необходимо заметить, что большую часть накладных расходов составляют расходы на копирование данных из памяти акселератора в память центрального процессора и наоборот. Таким образом, можно сказать, что $T_{opencl}(N) = O((S_{data}(N))) + T_{const}$, где $S_{data}(N)$ является пространством данных гнезда циклов *N*. Так как точная оценка T_{opencl} затруднена, в качестве $T_{opencl}(N)$ будем рассматривать $(S_{data}(N)($. Аналогично, точная оценка $T_{base}(N)$ так же затруднена, однако, исходя из того, что *N* является гнездом циклов можно считать, что $T_{base}(N) = O((S_{iter}(N)))$, где $S_{iter}(N)$ является пространством итераций цикла *N*.

Таким образом, неравенство (1) преобразуется в следующее неравенство:

$$(S_{iter}(N)(> (S_{data}(+ \frac{(S_{iter}(N)(}{p*C} \quad (2)$$

Где *C* — некоторая константа. Исходя из того, что функции обращения ко всем данным в *N* является аффинной, можно упростить неравенство (2), заменив мощность пространств на их размерности, что приводит к следующей эвристике:

Для выполнения преобразования гнезда циклов *N* в запуск ядра на устройстве требуется, чтобы выполнялось следующее неравенство:

$$dim(S_{iter}) > dim(S_{data}) \quad (3)$$

Таким образом, для выполнения преобразования необходимо, чтобы глубина гнезда циклов была больше, чем максимальная размерность данных,

копирование которых требуется для запуска данного ядра. Однако, данная эвристика не отбрасывает те гнезда циклов, для которых $T_{base} \approx T_{opencl}$, в случае, когда для их запуска не требуется обмен данными между акселератором и устройством. Рассмотрим гнездо циклов *N*. Если все гнездо циклов будет преобразовано в запуск ядра, то неравенство (1) приобретет следующий вид:

$$T_{base} > T_{const} + \frac{T_{base}}{p} \quad (4)$$

Где T_{const} — время, затрачиваемое на служебные операции OpenCL (передача аргументов ядра, запуск ядра и т.д.). Однако, если самый внешний цикл гнезда не может быть преобразован в ядро, для внутреннего гнезда циклов неравенство будет выглядеть так:

$$T_{base} > T_{const} * N_{iter1} + \frac{T_{base}}{p} \quad (5)$$

Где N_{iter1} — число итераций самого внешнего цикла. Таким образом, получаем, что следует избегать распараллеливания циклов, находящихся глубоко в рассматриваемом гнезде циклов.

Это дает следующую эвристику:

Для выполнения преобразования гнезда циклов N_{sub} находящегося на глубине *Depth* в гнезде циклов *N* необходимо, чтобы глубина гнезда циклов N_{sub} была больше, чем *Depth*.

3.2. Восстановление исходного кода ядра

После получения гнезда циклов, которое будет преобразовано в ядро, необходимо восстановить исходный код этого гнезда. Данная задача может быть разбита на три подзадачи – восстановление инструкций, отвечающих телу рассматриваемого гнезда циклов, восстановление объявления переменных и типов, вставка инструкций и переменных, отвечающих за OpenCL часть ядра.

Все восстанавливаемые инструкции можно разделить на два класса – инструкции, восстанавливаемые из структур CLAST и инструкции, восстанавливаемые из базовых блоков представления GIMPLE. К первым относятся операторы `for` и `if`, для которых генерируется соответствующий код на языке Си при обходе дерева CLAST, соответствующего данному гнезду циклов. Ко вторым – все остальные инструкции [2, 7], для которых код генерируется с помощью процедуры синтаксического анализатора языка Си компилятора GCC, модифицированной нами для обработки инструкций GIMPLE.

Все имена, к которым идет обращения в коде ядра, должны быть объявлены в ядре либо как локальные переменные, либо как параметры ядра, переданные

ему при запуске. К первым относятся переменные, объявленные внутри гнезда циклов в исходном коде; итераторы циклов; временные переменные, созданные при преобразовании исходного кода в GIMPLE; служебные переменные и типы, специфичные для OpenCL ядер (например, идентификатор ядра). Остальные переменные считаются параметрами ядра и описываются как параметры генерируемой функции.

```
for (scat_1 = 0; scat_1 <= 99; scat_1++)
  for (scat_3 = 0; scat_3 <= 99; scat_3++)
    some_code (scat_1, scat_3);
```

Рис. 1: Базовое гнездо циклов

```
__kernel void
opencl_auto_function_0 (int ocl_mod_0, int ocl_first_0,
int ocl_mod_1,
int ocl_first_1, int ocl_base_1, int ocl_base_0) {
      size_t opencl_global_id = get_global_id (0);
      int scat_1 = ((opencl_global_id / opencl_base_0)
               % opencl_mod_0) * 1 + opencl_first_0;
      int scat_3 = ((opencl_global_id / opencl_base_1)
               % opencl_mod_1) * 1 + opencl_first_1;
      some_host_code (scat_1, scat_3);
}
```

Рис. 2: Полученный код ядра

Наконец, вставка кода, отвечающего за OpenCL часть ядра, выполняется по двум причинам. Во-первых, при замене гнезда циклов на ядро необходимо в каждой нити, выполняющей данное ядро, вычислить соответствующие значения итераторов исходных циклов (см. пример кода на рисунке 1 и сгенерированный код на рисунке 2, где в ядро выделяется оператор `some_code`), используя уникальный номер потока, получаемый с помощью вызова `get_global_id` [6]. Во-вторых, необходимо обходить ограничения реализаций стандарта OpenCL (в нашем случае – реализации AMD), например, невозможность объявления многомерных массивов [4], что приводит к тому, что в коде ядра необходимо генерировать объявления многомерных массивов с помощью указателей (для примера на рисунке 3 функции, производящей умножение матриц, будет сгенерирован код на рисунке 4).

```
float A[100][100], B[100][100], C[100][100];
int foo5 () {
      int i,j,k;
      for (i = 0; i <= 99; i ++)
```

```
        for (j = 0; j <= 99; j ++) {
          float sum = 0.0;
          for (k = 0; k <= 99; k++)
            sum += A[i][k] * B[k][j];
          C[i][j] = sum;
      }
}
```

Рис. 3. Исходная функция с обращениями к многомерным массивам

```
__kernel void
opencl_auto_function_0 (..., __global float
*oclFTmpArg0,
__global float *oclFTmpArg1, __global float
*oclFTmpArg2) {
      typedef __global float oclFTmpType0[100];
      oclFTmpType0 *C = (oclFTmpType0*)oclFTmpArg0;
      typedef __global float oclFTmpType1[100];
      oclFTmpType1 *A = (oclFTmpType1*)oclFTmpArg1;
      typedef __global float oclFTmpType2[100];
      oclFTmpType2 *B = (oclFTmpType2*)oclFTmpArg2;
      ...;
}
```

Рис. 4. Полученные объявления типов для многомерных массивов

Общая структура кода, получаемая из CLAST, показана на рисунке 5.

3.3. Создание контекста и очереди команд, создание и запуск ядер

Перед вызовом любых функций OpenCL необходимо создать контекст и очередь команд [1]. При этом требуется, чтобы они были созданы один раз за все время работы программы. Значения контекста и очереди команд хранится в глобальных переменных, доступных всей программе. В начале каждой функции, в которой присутствуют вызовы функций OpenCL, проверяется значения переменной, хранящей значение контекста. Если значение равно нулевому указателю, то создается новый контекст и очередь команд, которые присваиваются соответствующим переменным (см. рисунок 6).

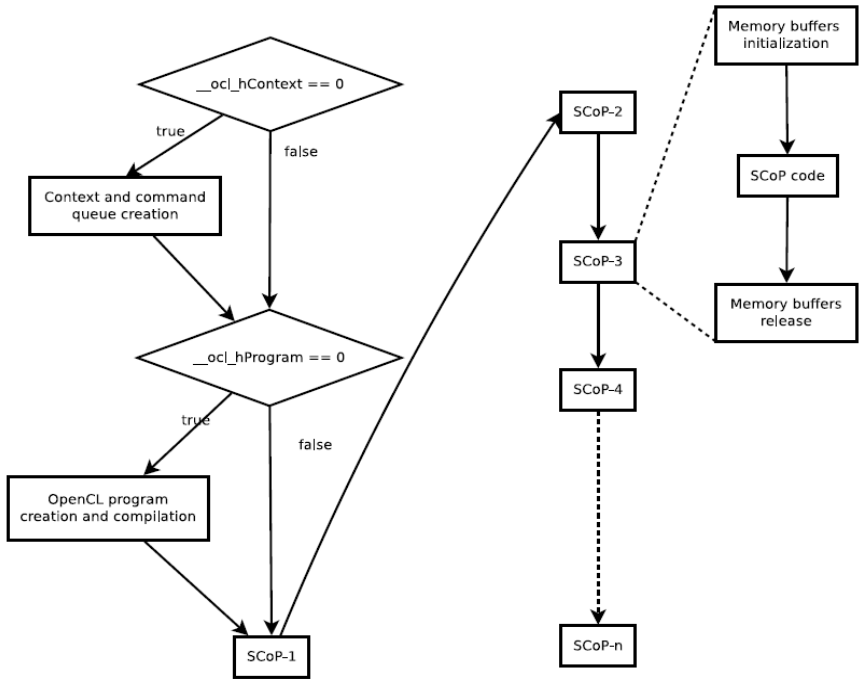

Рис. 5. Структура кода, генерируемого из структур CLAST

```
cl_context __ocl_hContext = 0;
if (__ocl_hContext == 0) {
      __ocl_hContext
      = clCreateContextFromType (0, CL_DEVICE_TYPE_GPU,
      0, 0, 0);
      clGetContextInfo (__ocl_hContext,
CL_CONTEXT_DEVICES, 0, 0,
              &nContextDescriptorSize);
      cl_device_id * aDevices = malloc
      (nContextDescriptorSize);
      clGetContextInfo (__ocl_hContext,
CL_CONTEXT_DEVICES,
              nContextDescriptorSize, aDevices, 0);
      __ocl_hCmdQueue = clCreateCommandQueue
      (__ocl_hContext,
                  aDevices[0], 0, 0);
}
```

Рис. 6. Инициализация контекста в начале функции

Далее, для каждой функции создается статическая переменная, хранящая OpenCL-программу, полученную из SCoP данной функции. Для этого в начало функции вставляется код, проверяющий значение данной переменной. Если значение равно нулевому указателю, то будет выполнено создание и компиляция OpenCL-программы. Нужные ядра создаются из полученной программы уже в нужном SCoP, так как основные затраты идут именно на компиляцию программы, а не на получение нужного ядра из скомпилированной программы.

При запуске ядра количество нитей, в которых оно будет работать, задается границами пространства итераций исходного гнезда циклов. Скалярные параметры передаются напрямую с учетом того, что OpenCL требует передачи указателя на параметр (например, `clSetKernelArg (kernel, index, sizeof (arg_type), &_ocl_scalar_arg)`). Для массивов или указателей передается указатель на соответствующий буфер в памяти акселератора. Этот указатель передается, так же как и скалярный параметр.

3.4. Работа с памятью

Передача данных из памяти центрального процессора в память акселератора и из памяти акселератора в память центрального процессора может занимать существенное время, что приводит к требованию минимизации количества таких передач в полученной программе.

В процессе анализа SCoP для каждого буфера на акселераторе поддерживается информация об актуальности данных, которые содержатся в нем и в соответствующей области памяти центрального процессора, соответствующей этому буферу. В начале работы для каждого массива или указателя, который используется на акселераторе в данной SCoP, в начале SCoP производится выделение буфера памяти на акселераторе. Так как требуемый размер буфера может стать известен только перед началом выполнения SCoP, то выделение буфера на акселераторе необходимо вставлять в начало каждого SCoP. Основной задачей в данном случае является определение базового объекта в памяти центрального процессора и размера нужного буфера на акселераторе. В качестве базового объекта для обращения к массиву или указателю выступает адрес массива или указатель соответственно. Для каждого обращения к соответствующему объекту в текущем SCoP вычисляется максимальное смещение относительно базового объекта в данном обращении, и размер буфера берется как максимальное смещение плюс единица. Все выделенные буферы инициализируются значениями из соответствующих областей памяти центрального процессора.

В процессе работы кода, полученного из SCoP, часть вычислений будет производиться на центральном процессоре и часть на акселераторе. Для корректности полученного кода необходимо, чтобы перед началом вычислений память содержала нужные данные. Алгоритм, определяющий

операции копирования памяти, необходимые для корректности, и места их вставки, описан в следующем подразделе. В конце работы для каждого SCoP происходит очистка памяти. Для всех буферов на акселераторе, данные в которых были изменены, но еще не скопированы в память центрального процессора, производится соответствующее копирование данных. После копирования памяти все буферы на акселераторе удаляются.

3.4.1. Копирование памяти в процессе работы программы

Для каждого объекта в памяти центрального процессора и соответствующего ему буфера на акселераторе на этапе генерации кода поддерживается информация о том, являются ли данные в данной памяти корректными. Корректность данных изменяется по следующему алгоритму:

1. Изначально данные корректны и в памяти центрального процессора и в памяти акселератора.
2. Если производится запись в буфер на акселераторе, то в соответствующей области памяти центрального процессора данные объявляются некорректными.
3. Если производится запись в область памяти центрального процессора, то в соответствующем буфере на акселераторе данные объявляются некорректными.
4. Если производится копирования из памяти центрального процессора в буфер на акселераторе, то данные в этом буфере объявляются корректными.
5. Если производиться копирования из буфера акселератора в область памяти центрального процессора, то данные в этой области памяти объявляются корректными.

Таким образом, получаем следующие правила корректности (их выполнение гарантирует, что не возникнет ситуации, когда и объект в памяти центрального процессора и соответствующий ему буфер на акселераторе содержат некорректные данные):

1. Перед запуском ядра на акселераторе необходимо, чтобы все буферы памяти, используемые этим ядром, содержали корректные данные.
2. Перед выполнением участка кода на центральном процессоре необходимо, чтобы все объекты в памяти, используемые в этом кода, содержали корректные данные.
3. Для копирования из памяти центрального процессора в буфер на акселераторе необходимо, чтобы данные в копируемой области были корректными.
4. Для копирования буфера акселератора в область памяти центрального процессора необходимо, чтобы данные в копируемом буфере были корректными.

Построим алгоритм, который определяет, какие операции копирования для выполнения правил корректности необходимо вставить в программу, и в каких местах должна быть выполнена вставка. Будем рассматривать код в виде дерева циклов, вершинами которого являются циклы, а листьями — последовательные блоки кода (в дальнейшем будем называть блоки кода и вложенные циклы операторами). В данном дереве прямыми потомками вершины S, соответствующей циклу S´, являются вершины, соответствующие операторам цикла S´. При этом порядок потомков S отвечает порядку операторов в теле S´. Под последовательным блоком кода в данном случае следует понимать участок кода на центральном процессоре или запуск ядра на акселераторе.

В процессе работы с деревом оно анализируется сверху вниз в глубину, причем для каждой вершины дерева ее дети рассматриваются в порядке, отвечающем порядку соответствующих операторов в теле цикла. Для каждого оператора B определим множества last(B), pred(B) и modify(B), вычисляемые по следующим правилам:

1. Если B является блоком, то last(B)={B}, modify(B) – множество всех объектов в памяти центрального процессора и буферов на акселераторе, модифицируемых в данном блоке.
2. Если B является циклом, то last(B)=last(B´), где B´ является последним блоком или циклом в теле B, modify(B)=□ modify(Bj), где B_j – операторы, из которых состоит тело цикла B.
3. Если B_i не является первым оператором в теле цикла, то pred(Bi)=last(Bj), где Bj является прямым предшественником Bi.
4. Если же B_i является первым оператором в теле цикла, то pred(Bi)= pred(L) □ last(Bj), где L – цикл, в теле которого находится Bi, а B_j – последний оператор этого цикла.

Для каждого блока поддерживается информация о том, в каких буферах акселератора и областях памяти данные являются корректными по окончании работы блока. Перед началом выполнения рассматриваемого гнезда циклов все буферы корректны как на акселераторе, так и соответствующие им области памяти центрального процессора. В дальнейшем корректность изменяется по описанным выше правилам, причем буфер M является корректным перед началом работы блока B тогда и только тогда, когда он корректен по окончании работы всех блоков из pred(B). Тогда алгоритм определения того, какие операции копирования необходимо вставить перед блоком для выполнения правил корректности, будет иметь следующий вид:

1. Для текущего блока B определяется множество pred(B).
2. Если блок будет выполняться на центральном процессоре, то:
    a. Для всех объектов в памяти, которые используются в данном блоке, проверяется, являются ли данные в них корректными

по окончании работы всех блоков из pred(B).

    b. Если после какого-либо блока из pred(B) данные не являются корректными, то для данного объекта памяти требуется вставка копирования соответствующего буфера с акселератора в память центрального процессора.
    c. Обновляется информация о корректности буферов памяти по окончании работы блока B.
3. Если блок выполняется на акселераторе, то:
    a. Для всех буферов акселератора, которые используются в данном блоке, проверяется, являются ли данные в них корректными по окончании работы всех блоков из pred(B).
    b. Если после какого-либо блока из pred(B) данные не являются корректными, то для данного объекта памяти требуется вставка копирования соответствующего объекта из памяти центрального процессора на акселератор.
    c. Обновляется информация о корректности буферов памяти по окончании работы блока B.

После определения блоков, перед которыми требуется вставить копирования памяти, осталось определить точное место в программе, куда будет вставлено каждое копирование. Для минимизации числа копирования памяти действует правило, что копирование вставляется в тело наиболее внешнего цикла (при условии сохранения корректности). Для определения цикла, в тело которого следует вставить копирование объекта m для блока B, ищется наиболее вложенный цикл L, содержащий в себе B, и такой, что m∈modify(L). Тогда копирование объекта m должно быть вставлено в тело цикла L прямо перед блоком B или циклом, содержащим этот блок.

4. Результаты

Полученный компилятор тестировался на трех наборах тестов – PolyKernels [5], SPEC CPU2000 [3], Polyhedron 2005 Benchmark Suite [8]. Тесты запускались на четырехъядерном процессоре, то есть максимальным возможным ускорением является четырехкратное ускорение по сравнению с той же программой, выполняемой полностью на одном ядре.

В данном случае важно отметить, что, так как центральный процессор и акселератор являлись одним и тем же процессором, передача памяти в процессе работы не требовалась.

Тесты содержат программы на языках Си, Си++ и Фортран.

Тесты PolyKernels представляют собой программы, где основная часть вычислений сконцентрирована в одном гнезде циклов. Такие гнезда при отсутствии в них зависимостей могут быть эффективно преобразованы в

вызов OpenCL-ядра. Результаты запуска тестов на CPU представлены в таблице 1.

Название	Время работы		Ускорение
	GRAPHITE	GRAPHITE-OpenCL	
jac.c	5.87	5.92	x0.99
jac2d.c	86.24	86.28	x0.99
adi.c	0.13	0.13	x1.00
fdtd1d.c	0.93	0.97	x0.96
fdtd-2d.c	3.49	3.5	x0.99
gs.c	1.27	1.32	x0.96
gemver.c	0.28	0.3	x0.91
lud.c	1	0.99	x1.01
mmm.c	22.24	6.41	x3.46
mvt.c	1.36	1.82	x0.74
sor.c	4.43	4.43	x1.00
ssymm.c	14.55	4.22	x3.44
ssyr2k.c	17.86	17.83	x1.00
ssyrk.c	59.29	59.72	x0.99
strmm.c	59.89	59.69	x1.00
strsm.c	59.73	59.93	x0.99
tmm.c	19.06	6.99	x2.72
trisolv_if.c	0.07	0.07	x0.99
trisolv.c	25	26.02	x0.96

Таблица 1. Результаты PolyKernels на CPU

Дополнительно этот же набор тестов был запущен на nVidia GPU (GeForce GTX 260) (таблица 2). В данном случае, так как центральный процессор и акселератор не имеют общей памяти, передача данных между ними является необходимой. Как видно из результатов, для некоторых тестов передача данных уменьшила выигрыш от параллельного выполнения по сравнению с соответствующими результатами на CPU. При этом время выполнения некоторых тестов уменьшилось в несколько раз.

Название	Время работы		Ускорение
	GRAPHITE	GRAPHITE-OpenCL	
jac.c	9.35	9.34	x1.00
jac2d.c	84.58	84.73	x0.99
adi.c	0.09	0.08	x1.05
fdtd1d.c	1.54	1.52	x1.01
fdtd-2d.c	3.65	3.64	x1.00
gs.c	1.5	1.5	x0.99
gemver.c	1.36	1.03	x1.32
lud.c	1.33	1.34	x0.99
mmm.c	34.82	0.91	x37.90
mvt.c	3.51	1.48	x2.36
sor.c	1.06	1.06	x1.00
ssymm.c	31.34	14.94	x2.09
ssyr2k.c	15.62	15.87	x0.98
ssyrk.c	51.82	49.95	x1.03
strmm.c	54.14	54.91	x0.98
strsm.c	52.62	53.05	x0.99
tmm.c	35.25	11.51	x3.06
trisolv_if.c	0.1	0.1	x0.99
trisolv.c	42.95	42.21	x1.01

Таблица 2. Результаты PolyKernels на GPU

Результаты работы набора тестов SPEC CPU 2000 приведены в таблице 3. Тесты SPEC CPU представляют из себя программы среднего объема, в которых тривиально распараллеливаемые циклы достаточно простые и распределены по всему коду программы. Это приводит к некоторому увеличению накладных расходов по сравнению с предыдущим набором тестов, так как количество запускаемых ядер больше. При этом общая доля кода, выполняемого параллельно меньше, чем в случае PolyKernels.

Название	Время работы	Ускорение	
		С эвристиками	Без эвристик
wupwise	76.1	x0.99	x1.07
swim	125	x1	x1.00
mgrid	115	x0.99	x0.99
mesa	59.2	x1.02	x0.99

Название	Время работы	Ускорение	
galgel	69.2	x0.96	x0.91
art	40.2	x1.00	x1.00
equake	53.7	x1	X
facerec	96.5	x1	x0.99
ammp	118	x0.98	X
lucas	83.3	x1.01	x1.07
fma3d	145	x0.99	x0.91
sixtrack	113	x0.99	X
apsi	135	x1	x0.98
gzip	97.3	x1.00	X
vpr	74.7	x0.99	x0.99
gcc	52.5	x0.99	X
mcf	94.7	x1.01	x1.00
parser	130	x1	x0.47
eon	43.9	x0.99	x0.98
perlbmk	67.4	x0.99	X
gap	50.9	x1	x0.99
vortex	80.7	x0.99	x0.99
bzip2	74.6	x1.00	X
twolf	115	x1.01	x0.98

Таблица 3: Результаты SPEC CPU 2000 на CPU

Для тестов SPEC CPU 2000 для демонстрации работы эвристик выбора эффективных для распараллеливания гнезд циклов мы приводим ускорение работы с отключенными и включенными эвристиками соответственно. В случае отключенных эвристик возможен значительный рост времени выполнения программы. Например, время работы теста `parser` выросло более чем в два раза. В этом тесте было выделено 6 гнезд циклов, причем каждое гнездо состояло из одного цикла, и эти гнезда находились в трех различных функциях программы. Это привело к существенным временным затратам на компиляцию OpenCL-программ и выполнение служебных операций, сопутствующих каждому запуску OpenCL ядра. При этом время работы гнезд циклов, соответствующие полученным ядрам, составляло небольшую часть времени работы исходной программы, что привело к тому, что выигрыш по времени, полученный за счет параллельного выполнения этих циклов, незначителен по сравнению с накладными расходами. В случае же включенных эвристик почти все гнезда циклов отбрасываются на основании того, что время выполнения служебных операций, требуемых для запуска

ядра, будет больше, чем время, сэкономленное за счет параллельного выполнения гнезда цикла.

Наконец, результаты работы тестов Polyhedron 2005 представлены в таблице 4. Поведение этих тестов схоже со SPEC CPU 2000.

Название	Время работы		Ускорение
	GRAPHITE	GRAPHITE-OpenCL	
ac	10.91	10.96	x0.99
aermod	35.02	34.9	x1.00
air	0.24	0.23	x1.04
capacita	41.24	46.89	x0.87
channel	2.56	3.13	x0.81
doduc	43.71	43.09	x1.01
fatigue	9.45	9.49	x0.99
gas_dyn	7.93	7.81	x1.01
induct	48.31	48.62	x0.99
linpk	22.52	22.5	x1.00
mdbx	12.92	12.99	x0.99
nf	21.93	21.81	x1.00
protein	37.12	37.75	x0.98
rnflow	38.02	38.09	x0.99
test_fpu	10.81	12.1	x0.89
tfft	2.5	2.63	x0.95

Таблица 4: Результаты Polyhedron 2005 на CPU

Из результатов тестов видно, что накладные расходы при запуске большого числа небольших ядер могут занимать существенное время работы полученной программы. В некоторых случаях время, затрачиваемое на вызов служебных функций OpenCL, превышает выигрыш по времени, полученный за счет параллельного выполнения некоторых циклов программы. Таким образом, наилучший результат достигается на вычислительных программах, основную часть которых составляют гнезда циклов. В таких программах время, затрачиваемое на накладные расходы мало по сравнению с временем, затрачиваемым на сами вычисления.

5. Заключение

В данной работе была реализовано преобразование кода, заменяющее некоторые гнезда циклов на запуск ядер на акселераторе. Было реализовано

восстановление кода на языке Си из структур CLAST, преобразование этого кода в код OpenCL ядра, эвристики, определяющие оправданность замены гнезда циклов на запуск ядра, а так же вставка вызовов всех необходимых для запуска ядра функций. Тестирование показало, что данное преобразование достигает наилучшего результата на больших гнездах циклов, перенося основную часть вычислений на акселератор.

В дальнейшем планируется расширять множество конструкций языка, которые могут быть восстановлены в код ядра, а также уменьшить требования к виду циклов, которые могут быть преобразованы в запуск ядра.

Работа поддержана контрактом с Минобрнауки РФ № 07.514.11.4001

Список литературы

[1] ATI. OpenCL Programming Guide, March 2010. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, pp. 7-16, Juan-les-Pins, September 2004.

[3] Standard Performance Evaluation Corporation. Spec CPU 2000. http://www.spec.org/cpu

[4] Khronos OpenCL Working Group. The OpenCL 1.1 Specification, September 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[5] IBM Polykernels. http://groups.google.com/group/gcc-graphite/

[6] NVIDIA OpenCL JumpStart Guide, April 2009. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

[7] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. Design of graphite and the polyhedral compilation package. In Proceedings of GCC Summit 2009, Ottawa, Canada.

[8] Polyhedron Software. Polyhedron 2005 benchmark suite. http://www.polyhedron.com/polyhedron_benchmark_suite.html

[9] Альфред Ахо, Рави Сети, Джеффри Ульман. Компиляторы: Принципы, Технологии, Инструменты. Издательский дом Вильямс, 2008.

[10] NVIDIA. CUDA Programming Guide, 2010. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[11] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. High Performance Embedded Architectures and Compilers, 2010, pp. 111-125.

[12] A. Monakov and A. Avetisyan. Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs. Embedded Computer Systems: Architectures, Modeling, and Simulation, 2009, pp. 289-297.

[13] A. Kravets, A. Monakov, and A. Belevantsev. GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops. Proceedings of the GCC Developers' Summit, 2010, pp. 7-6.