Инструменты анализа и разработки эффективного кода для параллельных архитектур

Александр Монаков amonakov@ispras.ru, Евгений Велесевич evel@ispras.ru, Владимир Платонов soh@ispras.ru, Арутюн Аветисян <arut@ispras.ru>

Аннотация. В работе предлагаются методы поддержки разработки эффективных программ для современных параллельных архитектур, включая гибридные. Описываются специализированные методы профилирования, предназначенные для программиста, занимающегося распараллеливанием существующего кода, либо для поиска неэффективного использования кеша в многопоточных программах. Рассматривается задача автоматической генерации параллельного кода для гибридных архитектур. В задачах, где для повышения производительности на гибридных архитектурах необходима существенная переработка структур данных или алгоритмов, может использоваться автотюнинг для специализации под конкретную задачу и аппаратуру во время выполнения. Показана оптимизация умножения разреженных матриц на GPU и ее применение для ускорения расчётов в пакете OpenFOAM.

Ключевые слова: оптимизация программ, профилирование, OpenCL, CUDA, разреженные матрицы, OpenFOAM

1. Введение

Эволюция архитектуры графических акселераторов привела к тому, что в настоящее время они являются массивно-параллельными вычислительными устройствами, применение которых уже не ограничивается задачами графического рендеринга. За счет специализированной архитектуры они могут достигать более высокой энергоэффективности и производительности по сравнению с современными многоядерными процессорами. Для облегчения разработки неграфических вычислений на акселераторах были созданы модели программирования CUDA [1] и OpenCL [2].

В модели программирования CUDA явно отображены низкоуровневые принципы работы акселератора. Это дает возможность детально исследовать производительность и выполнять оптимизацию кода, что было важно для обеспечения успешного развития модели. С другой стороны, использование специализированной низкоуровневой модели увеличивает сложность разработки.

Программные решения, использующие акселераторы, не всегда создаются с нуля: напротив, возникает необходимость доработки существующих программ для использования графических акселераторов с целью повышения эффективности расчетов. Для крупных программных проектов перенос всех алгоритмов на акселератор требует неоправданных трудозатрат. В этих случаях требуется анализ с целью выявления участков кода, отвечающих наибольшим затратам времени выполнения, и последующий перенос их на акселератор с сохранением внешних интерфейсов.

Несмотря на то, что OpenCL и CUDA обеспечивают переносимость между различными поколениями акселераторов (в случае OpenCL – также и переносимость между графическими акселераторами и процессорами различных производителей), для достижения высокой производительности требуется разработка отдельных вариантов кода для различных устройств. Даже при использовании CUDA изменение баланса между количеством регистров, активных нитей выполнения и объема разделяемой памяти при выходе новых поколений акселераторов требует пересмотра оптимизаций кода, что означает усложнение долговременной поддержки кода. Частично эта проблема может быть решена за счет разработки механизма автоматической настройки параметров кода, выполняющегося на акселераторе. Автотюнинг позволяет проверить только изначально заложенные в его схему варианты кода. Это компромиссное решение, позволяющее улучшить эффективность без затрат на ручной анализ и оптимизацию.

Цель данной работы — разработка инструментов и библиотек, поддерживающих продуктивность разработки программного кода и обеспечение высокой производительности для широкого спектра акселераторов. Для достижения этой цели предлагаются следующие подходы:

1) Профилирование последовательного кода для поиска тех участков, перенос которых на параллельную архитектуру актуален в первую очередь.
2) Автоматическая кодогенерация для параллельной гибридной архитектуры в процессе компиляции.
3) Автоматическая адаптация библиотечных функций.

Оригинальный код:
`for (init; test; step)`
`  stmt;`
 	
Преобразованный код:
`{`
`  __trace_start(loop_id);`
`  for (init;`
`__trace_test(test); step)`
`    stmt;`
`}`
	`@ match_for @`
	`expression init, test,`
	`step;`
	`statement stmt;`
	`position testp;`
	`@@`
	`- for`
	`+ {`
	`__trace_start(loop_id);`
	`for`
	`( init;`
	`- test@testp`
	`+ __trace_test(test)`
	`; step) stmt`
	`+ }`

Рис. 1. Общий вид инструментированного цикла for и соответствующий Coccinelle-скрипт

Статья имеет следующую структуру. В разд. 2 предлагаются специализированные методы профилирования, предназначенные для программиста, занимающегося распараллеливанием существующего кода, либо для поиска неэффективного использования кеша в многопоточных программах. В разд. 3 рассматривается задача автоматической генерации параллельного кода для гибридных архитектур. В задачах, когда для повышения производительности на гибридных архитектурах необходима существенная переработка структур данных или алгоритмов, может использоваться автотюнинг для специализации под конкретную задачу и аппаратуру во время выполнения. Это демонстрируется в разд. 4, где показана оптимизация умножения разреженных матриц на GPU и применение для ускорения расчётов в пакете OpenFOAM.

2. Инструменты профилирования кода

В случаях, когда перед программистом стоит задача переноса большой базы кода на параллельную архитектуру с целью повышения производительности,

необходимо выделить участки кода, анализ и перенос которых должен производиться в первую очередь. Как правило, распараллеливание кода выполняется в первую очередь на уровне циклов. Соответственно, необходимо собрать информацию о том, каким циклам соответствуют наибольшие затраты времени выполнения (с учетом времени, затраченного во всех вызывавшихся подпрограммах).

Когда для переносимых программ доступны входные данные, поведение на которых отражает реальные сценарии использования, наиболее удобным подходом к сбору характеристик времени выполнения является динамический анализ (профилирование). Кроме того, инструментацию и профилирование можно использовать для поиска типичных ошибок, приводящих к ухудшению производительности параллельных программ, как показано в разделе 2.3.

```
cairo-tor-scan-converter.c:1885:
 glitter_scan_converter_render:
  31056899836    nsec:    11365462    iters:    2733
nsec/iter:
  181819 loops: 63 iters/loop
cairo-image-surface.c:3025:
 _composite_boxes:
  18235085135    nsec:    101067    iters:    180426
nsec/iter:
  101066 loops: 1 iters/loop
cairo-image-surface.c:3031:
 _composite_boxes:
  18204913778 nsec: 187791 iters: 96942 nsec/iter:
  101067 loops: 2 iters/loop
cairo-tor-scan-converter.c:1932:
 glitter_scan_converter_render:
  17959504548 nsec: 76624515 iters: 234 nsec/iter:
  5108301 loops: 15 iters/loop
```

Рис. 2. Пример результатов профилирования циклов

2.1. Профилирование циклов

В настоящее время для профилирования программ используются методы, основанные на периодической записи указателя выполняемой инструкции (и, возможно, стека вызовов). Собираемой таким образом информации не достаточно для программиста, занимающегося поиском параллельных циклов, так как она не привязана к иерархии циклов. Таким образом, необходимо дополнительно собирать динамические характеристики циклов:

- Количество входов в цикл
- Количество итераций
- Время, проведенное в цикле

Запись о времени, затраченном в цикле, позволяет легко идентифицировать важные для распараллеливания циклы. Без этого было бы необходимо выполнять нетривиальную пост-обработку профилировочной информации. Запись количества входов и итераций позволяет программисту исключить из рассмотрения циклы, которые не итерируются вообще, либо содержат слишком мало итераций чтобы распараллеливание имело смысл.

Выполнять такое вспомогательное профилирование можно за счет инструментации кода. Возможно два подхода к организации инструментирования. В первом подходе предлагается для каждого обнаруженного цикла в предзаголовок добавлять код, увеличивающий счетчик входов в цикл, а в заголовочный блок встраивать код, увеличивающий счетчик итераций и добавляющий время последней итерации к счетчику общего времени, затраченного в цикле. Этот подход проще в реализации, но требует выполнения кода инструментации на каждой итерации цикла.

Эти накладные расходы можно сократить, если преобразовать граф потока управления цикла в форму с одним входом и одним выходом. Это позволит перенести большую часть инструментирующего кода (замер времени и запись результатов) из тела цикла к единственному выходу.

Для языков C и C++ инструментирование можно выполнять как source-to-source преобразование кода с помощью Coccinelle [3] (рис. 1).

2.2. Оценка расстояния повторного использования

В некоторых случаях программиста интересует расстояние переиспользования (reuse distance) для некоторых массивов, расположенных в памяти. Пусть функция с прототипом void compute(char out[], size_t out_n, const char in[], size_t in_n) генерирует out_n элементов массива out[], не содержит доступов к глобальным переменным и не содержит побочных эффектов; таким образом, результат ее работы зависит только от содержимого массивов out и in на момент вызова. Это означает, что вычисления можно производить в произвольный момент между точкой вызова и следующим после нее обращениям к массивам out (на чтение или запись) или in (на запись). В частности, вычисления можно вынести в отдельную нить, которая может выполняться параллельно с основной программой, что приведет к ускорению на параллельной архитектуре, если compute выполняет достаточно интенсивные вычисления.

Кроме того, оценка расстояния повторного использования может использоваться для специализации кода в зависимости от уровня кеша, в который должны попасть вычисленные значения.

Для замеров времени до следующего доступа можно использовать механизм защиты виртуальной памяти. Вызов mprotect(void *addr, size_t len, int prot) позволяет изменить права доступа к региону памяти [addr, addr+len-1] в соответствии с маской prot. При следующем доступе программа получит сигнал SIGSEGV, обработчик которого имеет доступ к адресу, который программа пыталась прочитать или записать. Соответственно, обработчик может определить, какому из ранее защищенных регионов принадлежит этот адрес, записать время доступа, восстановить доступ на чтение и запись и продолжить выполнение программы.

2.3. Оценка частоты промахов кеша и поиск ситуаций ложного разделения

Кеширование часто используемых данных из оперативной памяти является одним из ключевых методов обеспечения высокой производительности современных процессоров. Кеши не является общими ресурсами: каждое процессорное ядро, как правило, имеет свой кеш первого уровня; в многопроцессорных системах каждый физический процессор имеет свою иерархию кешей. Кеш хранит множество выровненных участков оперативной памяти, называемых строками. Один и тот же участок оперативной памяти может одновременно находиться в кешах разных процессоров; когда один из процессоров модифицирует эти данные, необходимо обновить или удалить их из кеша другого процессора. Эта задача решается с использованием протокола согласования кешей (cache coherency protocol).

Поскольку модификация даже одного байта приводит к обновлению целой строки кеша, могут возникать ситуации, когда нити, параллельно выполняющиеся на разных ядрах, поочередно обновляют разные переменные, попадающие в одну строку кеша. В таких случаях обновление на одном процессоре будет приводить к вытеснению соответствующей строки из кеша другого процессора и сериализовывать выполнение нитей, как если бы они обновляли одну и ту же переменную. Такое поведение известно как ложное разделение (false sharing) и ухудшает производительность параллельных программ. Соответственно, желательно иметь инструменты для анализа эффективности использования кеша, в том числе, поиска ситуаций ложного разделения.

Для оценки количества промахов можно использовать инструментацию кода. Основная идея заключается в том, чтобы собрать прореженную трассу доступов к памяти, включая расстояния повторного использования, и затем оценить количество промахов, исходя из собранной трассы и некоторой теоретической модели кеша. Хагерстен [4] предлагает вероятностную модель для оценки эффективности использования кеша со случайным вытеснением на основе расстояния повторного использования строк кеша. Недостатком его подхода для поиска ситуаций ложного разделения в многопоточных программах является необходимость дополнительной синхронизации при

сборе трассы. Нашей целью будет обход этой необходимости, так как она может существенно замедлить выполнение программы.

Пусть *L* — количество строк в кеше; тогда вероятность того, что строка, находящаяся в кеше, уже не будет в нем после *n* промахов, равна:

$$f(n) = 1 - (1 - 1/L)^n$$

Пусть *r(i)* — вероятность промаха при *i*-ом обращении, а *A(i)* — значение расстояния повторного использования для *i*-го обращения. Тогда количество промахов кеша между *i*-м и *(i-A(i)-1)*-м обращениями можно оценить как:

$$\sum_{j=i-A(i)}^{i-1} r(j)$$

Тогда вероятность того, что на момент обращения *i* искомая строка не будет находиться в кеше:

$$r(i) = f\left(\sum_{j=i-A(i)}^{i-1} r(j)\right)$$

Предположив, что вероятность промаха постоянна, т.е. *r(i)* = *R* для всех *i* (на некотором участке трассы программы), Хагерстен получает основное уравнение своей модели:

$$R * N = N_{cold} + \sum_{i=1}^{N_{nocold}} f(A(i) * R)$$

В многопоточном случае модели Хагерстена запись в трассе помимо строки кеша и расстояния повторного использования содержит информацио о том, была ли запись в эту строку в другой нити приложения между первым и вторым обращением в первой нити. Такой подход к сбору информации о записях в другой нити требует синхронизации.

Но возможен и другой подход — вычисление вероятности того, что во время обращения, записанного в трассу, и следующего обращения к этой строке в этой же нити, была запись в ту же строку в другой нити (на основе трасс обращений к памяти с временными метками для каждой нити приложения). Эта вероятность и будем искомым процентом промахов когерентности.

Для решения поставленной задачи требуется вычислить априорную вероятность того, что если промах когерентности произошел на обращении,

записанном в трассе, то запись, благодаря которой произошел промах, была записана в трассу другой нити (таких записей может быть несколько).

Пусть N_A — количество записей в кеш-строку A в исследуемом участке трассы второй нити, N — количество всех записей в участке. P — вероятность выборки обращения для записи в трассу. T — продолжительность исследуемого участка. T_A — время между первым и вторым доступом к A в первой нити. Тогда количество записей в участке $N_w = N/P$, среднее время на между записями $T_w = T/N_w$, количество записей во второй нити между первым и вторым обращением в первой нити $N_{Ag} = T_A/T_w$.

Вероятность того, что между первым и вторым доступом будет k записей в ту же строку в другой нити составит:

$$P_{Ak} = C_k^{N_{Ag}} P_A^K (1 - P_A)^{N_{Ag}-k}, \text{где} P_A = N_A/N$$

Тогда вероятность, что в трассе будет обращение, благодаря которому произошел промах (хотя бы одно из них):

$$P_{Ad} = 1 - \sum_{k=1}^{N_{Ag}} P_{Ak} (1 - P)^k$$

А вероятность, что промах произошел, независимо от того, попал он в трассу или нет:

$$P_{Am} = 1 - (1 - P_A)^{N_{Ag}}$$

Однако, нам известна статистическая вероятность попадания промаха в трассу:

$$\bar{P}_{Ad} = \bar{N}_{Ad}/N, \text{где} \bar{N}_{Ad} — \text{количество замеченных промахов.}$$

Тогда условная вероятность попадания промаха в трассу при наличии такого промаха:

$$\frac{\bar{P}_{Ad}}{P} = \frac{P_{Ad}}{P_{Am}}$$

Из этого уравнения можно получить P — искомую вероятность промахов когерентности, или же процент обращений к памяти, повлекших промах когерентности.

Для достаточно точного определения P_{Ad} требуется намного более частая выборка обращений для записи в трассу, так как вероятность, что между выбранными с вероятностью P обращениями в другой нити была выбрана запись, влекущая промах когерентности тоже равна P, т.е. P_{Ad} имеет порядок P^2. Поэтому P следует выбирать как квадратный корень из вероятности в модели Хагерстена.

3. Автоматическая генерация параллельного OpenCL-кода во время компиляции

В некоторых случаях возможно получение параллельного кода автоматически во время компиляции. Несмотря на то, что в результате распараллеливания вручную опытный программист выдаст более эффективный код, автоматическая кодогенерация является способом повышения производительности существующего кода на новых архитектурах.

В рамках предыдущих работ [5], [6] был разработан метод автоматической генерации OpenCL-кода для параллельных гнезд циклов. Использование OpenCL в качестве платформы для выполнения параллельного кода обеспечивает переносимость результирующей программы между как многоядерными процессорами, так и гибридными архитектурами с акселераторами, поддерживающих OpenCL. Отметим, что при использовании OpenCL часть кода программы выделяется в виде отдельных функций – «ядер», которые подаются библиотеке времени выполнения в виде исходного кода. Таким образом, компиляция выделенного кода выполяется во время выполнения драйвером OpenCL. Программа, использующая OpenCL, должна настроить контекст выполнения и передать среде выполнения код ядер. Далее перед запуском выполнения ядер программа должна скопировать на акселератор массивы, к которым будет обращаться ядро (если выполнение происходит на гибридной системе и акселератор имеет доступ к отдельному пространству памяти) и указать значения аргументов ядра. Ядро может выполняться на акселераторе параллельно с основной программой, но копирование вычисленных результатов с акселератора в память программы будет приводить к синхронизации.

В качестве инфраструктуры для реализации анализа и трансформации циклов использовалась система GRAPHITE компилятора GCC. GRAPHITE обеспечивает возможность высокоуровневого анализа циклов в рамках полиэдральной модели. Результат работы GRAPHITE для гнезда циклов представляет собой структуру SCoP (Static Control Part), включающую абстрактное представление исходного графа потока управления в виде CLAST, а также пространство итераций и расписание выполнения. Кроме того, для массивов, к которым выполняются обращения, строится индексная функция, отображающая элемент в пространстве итераций и оператор программы в смещение в массиве. Для возможности построения этого представления требуется, чтобы все циклы имели константные шаги, а индексы в доступах к элементам массивов аффинно зависели от итераторов циклов.

После выполнения анализа и преобразования расписания циклов GRAPHITE выполняет восстановление промежуточного представления GIMPLE из CLAST. На этом шаге структуры передачи управления и индукционные переменные циклов генерируются заново, но помимо этого переписывания оригинального GIMPLE-кода не требуется. Для генерации OpenCL-кода

требуется определить гнезда циклов, для которых возможно параллельное выполнение, и выполнить восстановление из CLAST в гибридный GIMPLE и OpenCL код. В виде GIMPLE необходимо сгенерировать ряд обращений к среде выполнения OpenCL:

1) инициализацию контекста (если еще не производилась);
2) компиляцию кода ядра для SCoP (при первом выполнении SCoP);
3) выделение памяти и копирование массивов на акселератор;
4) установку аргументов ядра;
5) собственно запуск ядра на выполнение;
6) копирование результатов и освобождение памяти на акселераторе.

Таким образом, восстановленный GIMPLE-код не зависит от графа потока управления цикла. Все поведение исходного цикла восстанавливается из CLAST в код на OpenCL C, при этом требуется реализовать выдачу OpenCL-кода из внутреннего представления GIMPLE для базовых блоков цикла. Результирующий код ядра на OpenCL C попадает в скомпилированную программу как константная строка и подается драйверу OpenCL во время выполнения.

4. Автоматическая адаптация

Современные графические акселераторы обеспечивают высокую производительность и энергоэффективность за счет специализированной массивно-параллельной архитектуры, сочетающей параллелизм на уровне независимых вычислительных блоков (мультипроцессоров в терминах CUDA) и параллелизм на уровне ALU, синхронно выполняющих одну команду для группы нитей. Количество групп нитей, которые могут быть одновременно активны на мультипроцессоре, зависит от нескольких факторов:

- количество регистров, требующихся для каждой нити;
- объем общей памяти, требующихся для блоков нитей;
- количество нитей в блоке.

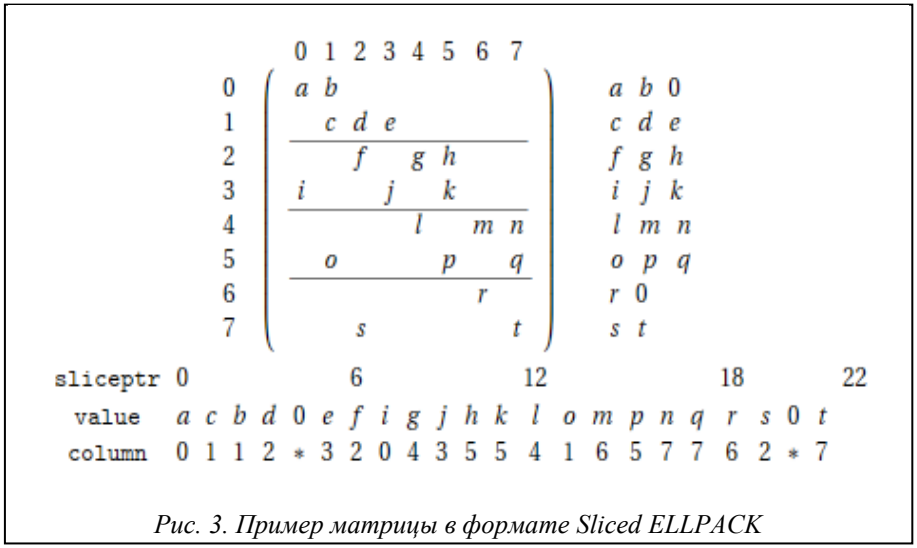


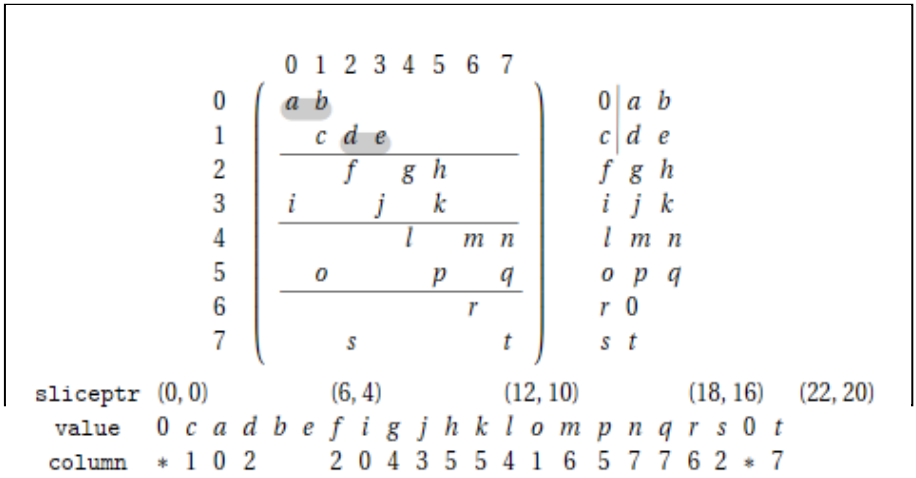
Рис. 3. Пример матрицы в формате Sliced ELLPACK

Количество активных нитей влияет на достигнутую производительность. Увеличение количества нитей позволяет скрывать задержки между зависимыми инструкциями за счет перекрытия выполнения различных нитей на одних и тех же ALU. С другой стороны, уменьшение количества нитей в сочетании с увеличением параллелизма на уровне команд в пределах одной нити (что повышает требования к количеству регистров на нить) в ряде случаев позволяет достичь более высокой производительности.

Как правило, аналитическое предсказание наилучшей конфигурации затруднено или невозможно. Кроме того, производительность может зависеть от неучтенных в аналитической модели факторов.

Для многократно использующихся функций имеет смысл выполнять автоматическую настройку реализации для конкретной архитектуры, на которой будет выполняться код. Автоматическая адаптация как средство повышения производительности используется в библиотеках ATLAS (работа с плотными матрицами) и FFTW (быстрое преобразование Фурье).

Обработка разреженных матриц (а именно, умножение матрицы на вектор) является важным вычислительным ядром в итерационных методах решения систем линейных уравнений, часто использующихся в задачах вычислительной гидродинамики. Особенностью ядра является высокая требовательность к пропускной способности памяти в сочетании с нерегулярными доступами к элементам вектора. Это затрудняет оптимизацию и приводит к низкой загрузке вычислительных ресурсов.



При оптимизации умножения разреженных матриц на GPU основным фактором является эффективность использования пропускной способности памяти. Это означает, что, с одной стороны, необходимо по возможности уменьшать объем памяти, требуемый для хранения матрицы; с другой стороны, необходимо обеспечить, чтобы обращения из синхронно выполняющихся групп нитей выполнялись к соседним ячейкам памяти (это требование архитектуры GPU для достижения высокой пропускной способности памяти). Обеспечить одновременное выполнение этих требований затруднительно: наиболее эффективными с точки зрения доступа к памяти на GPU являются блочные форматы и такие форматы как ELLPACK, но они приводят к высокому расходу памяти для матриц, не имеющих блочную структуру или имеющие различное количество ненулевых элементов в строках. Эффективный с точки зрения расхода памяти формат CSR плохо подходит для GPU, так как при независимой обработке отдельных строк шаблон обращений к памяти будет неэффективным.

Для реализации умножения разреженных матриц на GPU предлагается использовать гибридный формат Sliced ELLPACK [7] (рис. 3). Базовый формат параметризован числом строк в слайсе матрицы S. Исходная разреженная матрица делится на слайсы из S подряд идущих строк, каждый из которых независимо хранится в формате ELLPACK (как две плотные матрицы K × S, где K — максимальное количество ненулевых элементов в строке; матрицы хранят значение и номер столбца ненулевого коэффициента). Итоговая структура данных содержит конкатенацию плотных матриц слайсов и вспомогательный массив, хранящий смещения плотных матриц для каждого слайса. Для S = 1 этот формат совпадает с обычным CSR, а для S = N — количество строк исходной матрицы — с ELLPACK.

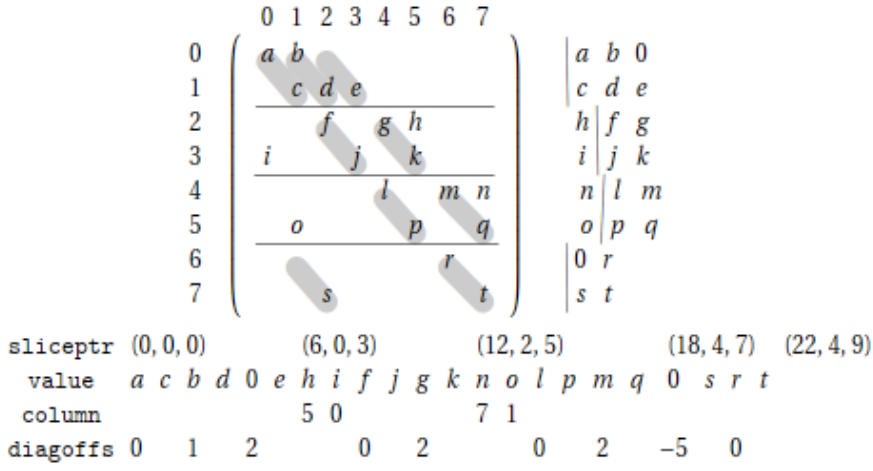


Рис. 5. Пример матрицы с диагональными структурами

Домножение на каждый слайс может выполняться полностью независимо, при этом использование формата ELLPACK в пределах слайса позволяет достичь высокой производительности на GPU. Для матриц с переменным числом ненулевых элементов в строке снижение S позволяет сократить расход памяти, но при этом снижает параллелизм в пределах слайса.

Таким образом, для базового формата Sliced ELLPACK вычислительное ядро параметризовано по S и по Nt — количеству нитей в CUDA-блоке. В качестве кандидатов на значения Nt будем рассматривать только 64, 128, 256, 512 в силу того, что для CUDA предпочтительны размеры блоков нитей, равные степени двойки (иначе возникают конфликты при доступе к регистрам), 512 — максимальный размер блока, а конфигурации с блоками меньших размеров неэффективны. Потребуем также, чтобы S было делителем Nt. Возникающее пространство конфигураций позволяет перебрать все варианты для конкретной разреженной матрицы за короткий период перед решением системы уравнений.

Отметим, что часто в задачах вычислительной гидродинамики матрица с фиксированным расположением ненулевых элементов (но разными значениями коэффициентов, что не влияет на оптимальную конфигурацию ядра) участвует в решении нескольких систем уравнений (при этом каждая система может требовать сотни умножений на матрицу в итерационном методе). Благодаря этому выполненный для первой системы процесс автотюнинга не требуется повторять впоследствии.

Базовый формат Sliced ELLPACK допускает ряд модификаций, сокращающих потребление памяти:

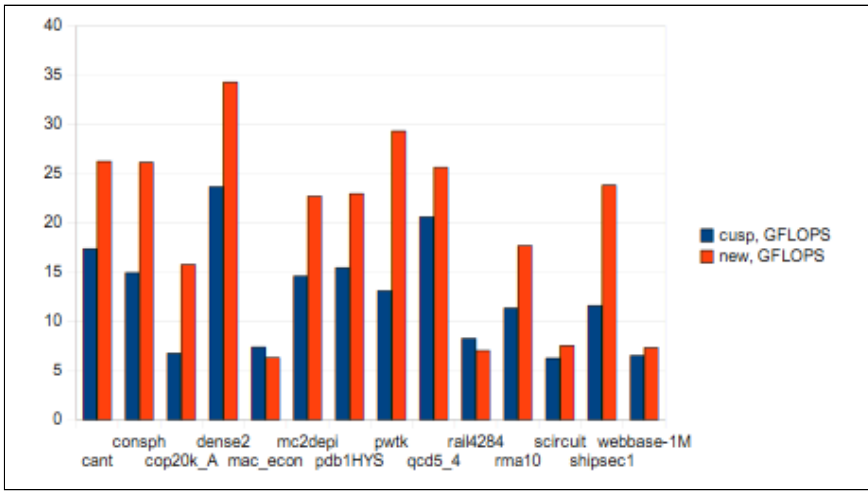


Рис. 6. Производительность реализации умножения разреженных матриц в сравнении с библиотекой cusp на акселераторе Tesla M2090

1) Использование слайсов переменного размера. В этом случае S не фиксировано для всей матрицы, а выбирается отдельно для каждого слайса с целью исключить появление слайсов с большим количеством явно хранимых нулевых элементов из-за большого разброса количества ненулевых элементов в строках слайса.
2) Использование блочных структур (рис. 4). Разреженные матрицы могут не иметь блочной структуры, но доля элементов, принадлежащих плотным однострочным блокам 1 × 2 или 1 × 4 может быть существенна. Поскольку загрузка таких блоков может быть эффективно выполнена на GPU, имеет смысл расширить формат так, чтобы часть элементов можно было хранить в виде плотных блоков. В каждой строке слайса при этом придется выделять одинаковое количество блоков.
3) Использование диагональных структур (рис. 5). Это расширение основывается на аналогичном наблюдении, что доля элементов может располагаться вдоль диагоналей матрицы. При этом достигается дальнейшее сокращение объема памяти, так как для каждого диагонального блока требуется кроме его коэффициентов хранить только одно целое число (смещение относительно главной диагонали). Как и в предыдущем случае, количество и расположение блоков выбирается независимо для каждого слайса [8].

Выбор варианта формата для конкретной матрицы и акселератора производится также за счет автонастройки. В некоторых случаях возможно использование эвристик для выбора подходящего варианта.

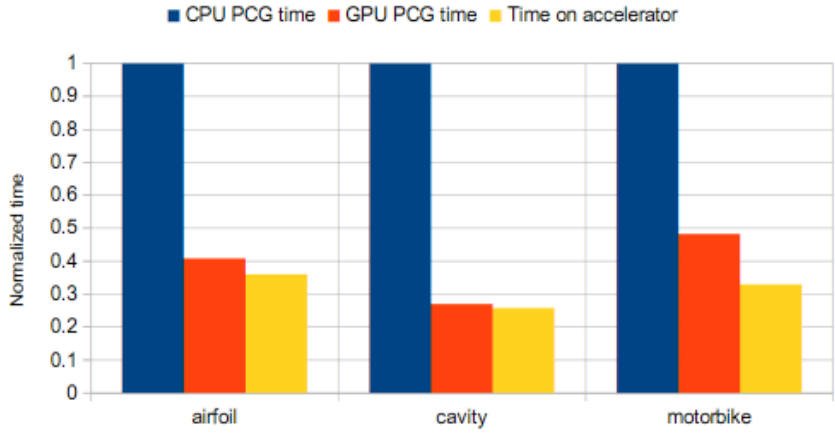


Рис. 6. Производительность реализации умножения разреженных матриц в сравнении с библиотекой cusp на акселераторе Tesla M2090

Разработанная на основе описанного подхода библиотека работы с разреженными матрицами используется в модуле решения систем линейных уравнений в пакете OpenFOAM. Модуль реализует решение линейных систем методом сопряженных градиентов в соответствии с общим интерфейсом, используемым в OpenFOAM для методов решения линейных систем. Поддерживается решение систем в параллельном режиме, когда несколько процессов OpenFOAM взаимодействуют через MPI [9][10].

5. Заключение

В рамках этой работы разработаны методы профилирования кода, поддерживающие задачу переноса кода на параллельные архитектуры. Первый метод направлен на поиск циклов, которые имеет смысл распараллеливать в первую очередь. Второй метод позволяет измерять время переиспользования данных для подпрограмм, которые можно выносить в отдельную нить вычислений.

Разработан метод автоматической генерации параллельного кода во время компиляции для многоядерных или гибридных платформ, поддерживающих OpenCL.

Разработан метод настройки параметров структуры данных и запуска вычислительного ядра для задачи умножения разреженных матриц. Реализованный метод применяется в загружаемом модуле для пакета OpenFOAM, выполняющем решение систем линейных уравнений с использованием GPU акселераторов.

Список литературы

[1] NVIDIA. CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide

[2] Khronos Group. OpenCL. http://www.khronos.org/opencl/
Coccinelle: A Program Matching and Transformation Tool for Systems Code. http://coccinelle.lip6.fr/

[3] E. Berg, H. Zeffer, E. Hagersten. A Statistical Multiprocessor Cache Model. In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006.

[4] А. Белеванцев, А. Кравец, А. Монаков. Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели. Труды Института системного программирования РАН, том 21, стр. 5-22. Москва, 2011

[5] A. Kravets, A. Monakov, A. Belevantsev: GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops. In Proceedings of the GCC Developers' Summit: 9-18, Ottawa, October 2010

[6] A. Monakov, A. Lokhmotov, A. Avetisyan: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In HiPEAC 2010: 111-125, Italy, January 2010

[7] A. Monakov, A. Avetisyan: Specialized Sparse Matrix Formats and SpMV Kernel Tuning for GPUs. In GPU Technology Conference 2012, USA, May 2012

[8] A. Monakov, V. Platonov: Accelerating OpenFOAM with Parallel GPU Linear Solver. In 8th OpenFOAM Workshop, South Korea, June 2013

[9] А. Монаков. Оптимизация расчётов в пакете OpenFOAM на GPU. Труды Института системного программирования РАН, том 22, Стр. 223-232. Москва, 2012. DOI: 10.15514/ISPRAS-2012-22-14.

Analysis and development tools for efficient programs on parallel architectures

Alexander Monakov amonakov@ispras.ru, ISP RAS, Moscow, Russia
Eugene Velesevich evel@ispras.ru, ISP RAS, Moscow, Russia
Vladimir Platonov soh@ispras.ru, ISP RAS, Moscow, Russia
Arutyun Avetisyan arut@ispras.ru, ISP RAS, Moscow, Russia

Abstract. The article proposes methods for supporting development of efficient programs for modern parallel architectures, including hybrid systems. First, specialized profiling methods designed for programmers tasked with parallelizing existing code are proposed. The first method is loop-based profiling via source-level instrumentation done with Coccinelle tool. The second method is memory reuse distance estimation via virtual memory protection mechanism and manual instrumentation. The third method is cache miss and false sharing estimation by collecting a partial trace of memory accesses using compiler instrumentation and estimating cache behavior in postprocessing based on the trace and a cache model. Second, the problem of automatic parallel code generation for hybrid architectures is discussed. Our approach is to generate OpenCL code from parallel loop nests based on GRAPHITE infrastructure in the GCC compiler. Finally, in cases where achieving high efficiency on hybrid systems requires significant rework of data structures or algorithms, one can employ auto-tuning to specialize for specific input data and hardware at run time. This is demonstrated on the problem of optimizing sparse matrix-vector multiplication for GPUs and its use for accelerating linear system solving in OpenFOAM CFD package. We propose a variant of "sliced ELLPACK" sparse matrix storage format with special treatment for small horizontal or diagonal blocks, where the exact parameters of matrix structure and GPU kernel launch should be automatically tuned at runtime for the specific matrix and GPU hardware.

Keywords: optimization, profiling, OpenCL, CUDA, sparse matrices, OpenFOAM

References

[1]. NVIDIA. CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide
[2]. Khronos Group. OpenCL. http://www.khronos.org/opencl/
[3]. Coccinelle: A Program Matching and Transformation Tool for Systems Code. http://coccinelle.lip6.fr/
[4]. E. Berg, H. Zeffer, E. Hagersten. A Statistical Multiprocessor Cache Model. In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006.
[5]. A. Belevantsev, A. Kravets, A. Monakov. Avtomaticheskaya generaciya OpenCL-koda iz gnyozd ciklov s pomoshhyu polie`dral'noy modeli [Automatically generating OpenCL

code from loop nests via a polyhedral model]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, p. 5-22, 2011. (In Russian)

[6]. A. Kravets, A. Monakov, A. Belevantsev: GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops. In Proceedings of the GCC Developers' Summit: 9-18, Ottawa, October 2010

[7]. A. Monakov, A. Lokhmotov, A. Avetisyan: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In HiPEAC 2010: 111-125, Italy, January 2010

[8]. A. Monakov, A. Avetisyan: Specialized Sparse Matrix Formats and SpMV Kernel Tuning for GPUs. In GPU Technology Conference 2012, USA, May 2012

[9]. A. Monakov, V. Platonov: Accelerating OpenFOAM with Parallel GPU Linear Solver. In 8th OpenFOAM Workshop, South Korea, June 2013

[10]. A. Monakov. Optimizaciya raschyotov v pakete OpenFOAM na GPU [On Optimizing OpenFOAM GPU solvers]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 22, p. 223-232, 2012. DOI: 10.15514/ISPRAS-2012-22-14. (In Russian)