libgpuvm: организация автоматического обмена данными между хостом и ГПУ

А.В. Адинец[1,2]

[1]НИВЦ МГУ им. М.В.Ломоносова
[2]Объединённый институт ядерных исследований

В настоящее время графические процессоры активно используются высокопроизводительными приложениями. Важным аспектом написания таких приложений является организация обмена данными между памятью хоста и ГПУ. В настоящее время такие обмены необходимо выполнять вручную, что затрудняет написание ГПУ-приложений. В данной работе предлагается подход к автоматизации передачи данных. Данные передаются на ГПУ при вызове ядра, после чего блокируются. Обратно они передаются при обработке страничного прерывания, вызванного обращением к ним на хосте. libgpuvm поддерживает данные, выделенные в динамической памяти, а также многопоточные приложения. OpenCL-совместимые ГПУ на ОС Linux поддерживаются без изменения кода их драйверов. Система предназначена прежде всего для интеграции поддержки ГПУ в языки программирования. В данной работе также описывается интеграция libgpuvm в систему NUDA.

1. Введение

В настоящее время графические процессорные устройства (ГПУ) активно используются для решения вычислительно ёмких задач. В дискретных ГПУ, наиболее мощных и популярных, память ГПУ отделена от памяти хоста. И хотя ГПУ может обращаться хост-памяти напрямую, как правило, это на порядок медленнее использования ГПУ-памяти. Поэтому данные необходимо копировать в ГПУ-память для обработки, а потом копировать обратно.

При использовании низкоуровневых средств программирования ГПУ, таких как CUDA [1] или OpenCL [2], программист организует обмен данными вручную. Однако с появлением высокоуровневой поддержки ГПУ в языках программирования, а также использующих ГПУ библиотеки возникает потребность в автоматизации процесса обмена данными. Способ организации обмена данных должен предоставлять простой интерфейс, и в то же время не приводить к большим накладным расходам.

Известен ряд способов автоматизации обмена данными между хостом и ГПУ. Самый простой заключается в копировании данных на ГПУ перед вызовом ядра, и копировании их обратно после его завершения. Мы назовём его *полным копированием*. Такой подход предоставляет простой интерфейс, однако приводит к большим накладным расходам. И хотя для ряда задач, например, задачи многих тел, накладные расходы малы, для остальных, таких как сеточные или итерационные методы, накладные расходы велики, так как там копирование данных на хост обычно не требуется.

Для решения проблемы накладных расходов также предолжено ряд путей. Один из них — директивы или аннотации, указывающие, что данные уже расположены в памяти ГПУ. Одной из форм таких директив являются *регионы данных*: данные внутри региона используются только на ГПУ, и не копируются обратно между вызовами ядра. Это снижает накладные расходы, поскольку тот же объём копирования теперь приходится на более количество вызовов ядер. Данный подход используется, например, в коммерческих компиляторах PGI [3] и CAPS HMPP [4], а также предложенный стандарт OpenACC [5]. Однако дополнительные директивы замусоривают код, кроме того, они требуют аннотирования параметров тех функций, которые работают с ГПУ. Компилятору также приходится отслеживать, какие массивы находятся на ГПУ, а какие нет.

Другим подходом является введение новых типов данных, похожих на массивы. Память для них выделяется на хосте и на ГПУ, а реализация обеспечивает автоматическую синхронизацию данных между ними. Такие типы могут быть введены в любой язык, поддерживающий классы и перегрузку оператора индексирования. Этот подход называется *автосинхронизация*. В дополнение к указателям на память хоста и ГПУ, такие массивы содержат информацию о том, какая копия данных в настоящий момент актуальна. При использовании массива в ГПУ-ядре данные актуализируются на ГПУ; при обращении к массиву на хосте — актуализируются на хосте. Если данные не являются актуальными выполняется копирование данных; если же они уже актуальны, то связанных с копированием накладных расходов удаётся избежать. Преимуществом данного подхода является простая и эффективная реализация отслеживания актуальности во время выполнения, что упрощает компилятор. Основным же недостатком подхода является необходимость введения новых типов, что в терминах реализации языка опять выливается в директивы и аннотации, и приводит к замусориванию программы.

Наш подход, называемый подходом *libgpuvm* [6], похож на подход автосинхронизации. Однако передача данных с ГПУ обратно на хост выполняется при обработке страничного прерывания в пользовательском режиме (проще говоря, обработке сигнала `SIGSEGV`), а не в перегруженном операторе индексации.

Это простая идея, и она может быть легко реализована, если необходимые данные выровнены на границу страницы. Такая реализация представлена в библиотеке GMAC [7], которая также предоставляет свой аллокатор памяти. Однако если речь идёт о поддержке ГПУ в языке программирования, массив может быть выделен не в той функции или библиотеке, в которой он используется, и скорее всего, не будет выравнен на границу страницы. libgpuvm поддерживает память, выделенную функции `malloc()` или подобного аллокатора динамической памяти. libgpuvm также поддерживает работу в многопоточных средах с несколькими ГПУ, а также работает с несколькими широко используемыми реализациями OpenCL без дополнительных изменений в коде уровня ядра.

Данная статья организована следующим образом. В разделе 2 описывается идея libgpuvm, а также интерфейс, который она предоставляет программам и системам времени выполнения языков. В разделе 3, описываются подробности реализации libgpuvm и её внутренние структуры данных. В разделе 4 описывается интеграция libgpuvm в систему времени выполнения языков программирования, на примере системы программирования ГПУ NUDA [8]. Наконец, в разделе 5 обсуждаются возникшие проблемы и направления дальнейшей работы.

2. Идея и интерфейс libgpuvm

Мы предполагаем, что приложение исполняется на *хосте*, многоядерном процессоре с общей памятью, и, вообще говоря, нескольких *устройствах* (ГПУ, но могут быть и многоядерные ЦПУ), каждый из которых имеет свою ОЗУ, логически отдельную от ОЗУ хоста. Приложение состоит из набора *потоков* на хосте, которые могут динамически порождаться и завершаться, а также *вызывать ядра* на устройствах. Любой поток на хосте может использовать любое устройство. *Хост-массив* — диапазон адресов ОЗУ хоста, определяемый начальным адресом и длиной (в байтах). Ввиду ограничений реализации, хост-массив должен быть выделен при помощи `malloc()` или подобного динамического аллокатора. Хост-массив не может быть выделен в стеке или глобальных данных программы. Хост-массивы могут использоваться либо несколькими потоками на хосте, либо на одном из устройств. Диапазоны адресов различных хост-массивов не пересекаются. С хост-массивом может быть ассоциирован *буфер на устройстве*, вообще говоря, на нескольких устройствах. На одном устройстве с одним хост-массивом может быть ассоциировано не более одного буфера. В настоящее время в качестве буферов на устройстве используются буфера OpenCL.

Для каждого массива будем рассматривать моменты начала и окончания работы ядер, которые его используют, и соответствующие промежутки времени. Будем говорить, что хост-массив *используется на хосте*, если происходит обращение по одному из его адресов на хосте. Хост-массив *используется на устройстве*, если на этом устройстве происходит обращение к ассоциированному с этим массивом буферу. Хост-массив *совместно используется корректно*, если в каждый промежуток времени он используется либо на хосте, возможно, несколькими потоками, или на одном из устройств. Если же хост-массив совместно используется одновременно на нескольких устройствах, или на каком-то устройстве и хосте, его *совместное использование некорректно*. Таким образом, хост-массив становится как бы единицей совместного использования, т.к. он может использоваться только в одном месте в каждый промежуток времени. Если совместное использование всех массивов корректно, libgpuvm обеспечивает корректность совместного использования в интуитивном смысле, т.е. как если бы массив был расположен в общей памяти, совместно используемой хостом и устройствами. Если же совместное использование массива некорректно, могут возникать гонки за ресурсы, даже если множества адресов, используемые хостом и различными устройствами, не пересекаются. В этом случае libgpuvm не гарантирует корректность описанной выше абстракции разделяемой памяти.

С каждым хост-массивом ассоциируется информация об актуальности. Информация хранится в ассоциативном массиве, который индексируется начальным адресом массива. Как и в подходе с автосинхронизацией, данные актуализируются на ГПУ, когда они используются в ГПУ-ядре, и актуализируются на хосте, когда к ним происходит обращение на хосте. Если данные уже актуальны в нужном месте, копирования не происходит, за счёт чего удаётся избежать накладных расходов. Однако механизм отслеживания обращения к данным на хосте другой. Когда завершается исполнение ГПУ-ядра, диапазон адресов хост-массива *защищается* на хосте при помощи системного вызова `mprotect()`. Когда после этого происходит обращение к данным на хосте, механизм защиты памяти ЦПУ генерирует страничное прерывание, которое отправляется вызвавшему его потоку в виде сигнала. Данные передаются обратно с ГПУ обратно на хост в обработчике сигнала, после чего исполнение хост-программы может продолжаться. Если данные совместно используются несколькими устройствами, они копируются с того устройства, которое содержит их актуальную копию, т.е. на котором они в последний раз использовались.

Для повышения гибкости упрощения интеграции с существующими библиотеками и средами времени выполнения libgpuvm является достаточно минималистичной: она обеспечивает только синхронизацию данных между хостом и устройствами, и всё, что для этого необходимо. Прочие функции, такие как инициализация устройств, выделение и освобождение памяти на устройстве и установка параметров ядра, выполняются самим приложением. Интерфейс libgpuvm отражает минималистичный дизайн самой библиотеки.

Прототипы библиотечных функций libgpuvm приведены на рис. 2. Все они начинаются с префикса `gpuvm_`, и возвращают код ошибки, или 0 в случае успешного завершения. Инициализация libgpuvm выполняется вызовом `gpuvm_init()`, в который передаётся список “устройств”, т.е. очередей команд OpenCL, ассоциированных с ними. В дальнейшем устройства идентифицируются по номеру в этом списке. Дополнительные флаги задают используемый интерфейс взаимодействия с устройством (сейчас только OpenCL), а также дополнительные настройки, например, выполнять ли сбор статистики. Инициализация выполняется только один раз. Статистику можно получить при помощи вызова `gpuvm_stat()`. Некоторые сведения доступны всегда, в то время как другие, например, суммарное время передачи данных, требует включения сбора статистики на очередят команд OpenCL и в библиотеке libgpuvm.

Для того, чтобы хост-массив можно было использовать на устройстве, его необходимо *связать* с буфером вызовом `gpuvm_link()`. Хост-массив может быть связан с несколькими буферами, но на одном устройстве у него может быть только один буфер. Флаги

указывают, расположены ли данные изначально на хосте или на устройстве. Когда хост-массив больше не требуется использовать на устройстве, его отребуется отсоединить вызовом `gpuvm_unlink()`.

Начало использования хост-массива на устройстве обозначается вызовом `gpuvm_kernel_begin()`. Флаги обозначают, будет ли массив использован только на чтение, или также и на запись (сейчас поддерживается только последнее). Массив будет актуализирован на соответствующем устройстве, и если потребуется, будет выполнено копирование данных. Окончание использования массива на устройстве обозначается вызовом `gpuvm_kernel_end()`. При этом будет установлена защита на диапазон адресов массива на хосте, если она не была установлена ранее, так что при следующем обращении к массиву на хосте произойдёт страничное прерывание. Между началом и окончанием использования массива на устройстве на этом устройстве может быть вызвано любое число ядер; но корректное использование массива на хосте гарантируется только после вызова `gpuvm_kernel-end()`.

```
// libgpuvm initialization
int gpuvm_init(unsigned ndevs, void** devs, int flags);

// linking a host array to a device buffer
int gpuvm_link(void *hostptr, size_t nbytes, unsigned idev,
  void *devbuf, int flags);

// unlinking a host array from the device buffer
int gpuvm_unlink(void *hostptr, unsigned idev);

// start using array on device
int gpuvm_kernel_begin(void *hostptr, unsigned idev, int flags);

// finish using array on device
int gpuvm_kernel_end(void *hostptr, unsigned idev);

// get some statistical information
int gpuvm_stat(int param, void *val);
```

Рис. 1. Прототипы библиотечных вызовов libgpuvm

3. Реализация libgpuvm

Идея обработки страничных прерываний в пользовательском режиме, хотя и звучит не обычно, сама по себе новой не является. Защита памяти широко распространена в ЦПУ уже достаточно давно, а ОС предоставляет информацию об этом в приложения, например, при помощи сигнала `SIGSEGV`. Защита памяти в современных ЦПУ обычно интегрирована в механизм страничной трансляции, и выполняется на уровне страниц. Размер страницы на процессорах x86 составляет 4 КБ; у других архитектур похожий минимальный размер страницы. И хотя в процессоре имеется поддержка страниц большего размера, по умолчанию ОС использует страницы минимального размера.

Таким образом, минимальная гранулярность защиты памяти равна размеру страницы. Соответственно, большинство приложений обработки страничных прерываний в пространстве пользователя требуют, чтобы данные в памяти занимали страницы целиком. Это, в свою очередь, требует использования специального аллокатора памяти. Однако память, с которой будет работать libgpuvm, выделена аллокатором по умолчанию, и в другой части

приложения; соответственно, нельзя рассчитывать на использование специального аллокатора. Поэтому libgpuvm должна поддерживать работу с любой динамически выделенной памятью; единственным требованием является отсутствие совместно используемых страниц со стеком или глобальными данными программы. Это требуется, чтобы установка защиты на диапазоны адресов не нарушала работу механизма вызова функций и обработки сигналов. Отсутствие требования к выравниванию приводит к двум типам проблем:

- **ложное совместное использование**, когда несколько массивов используют ту же самую страницу памяти, и поэтому ставить включать/отключать их защиту надо также совместно
- **вмешательство в работу OpenCL**, когда выделенная `malloc()`-ом память, используемая реалиацией OpenCL, оказывается в страницах, на которые libgpuvm ставит защиту

Реализация libgpuvm решает две эти проблемы в многопоточной среде с несколькими ГПУ. Для этого, в дополнение к хост-массивам, вводятся ещё два вида диапазонов адресов:

- *регионы защиты*, или просто *регионы*, диапазоны памяти, которые выравнены на границы страниц, покрывают страницы целиком, и служат единицей защиты памяти
- *подрегионы*, являющиеся пересечениями регионов и хост-массивов, которые служат единицей синхронизации между хостом и устройством; информация об актуальности поддерживается именно на уровне подрегионов

Диапазоны адресов разных регионов не пересекаются. Диапазоны адресов разных подрегионов не пересекаются. Каждый подрегион целиком содержится в одном регионе и хост-массиве. Регион может содержать несколько подрегионов, и должен содержать как минимум один подрегион. Подрегионы одно региона организованы в связанный список, отсортированный по адресу начала подрегиона. Каждый хост-массив содержит от одного до трёх подрегионов. Если хост-массив целиком помещается в одну страницу памяти, или занимает любое количество страниц целиком, он состоит из одного подрегиона. Иначе “средняя” часть массива, которая занимает целиком некоторое количество страниц, образует один подрегион, а оставшиеся “голова” и “хвост”, также образуют по одному подрегиону; одна из трёх частей в этом случае может отсутствовать.

Регионы организованы в *дерево регионов*, бинарное дерево, упорядоченное по адресу региона. Эта структура данных используется для поиска региона или хост-массива по адресу, получаемому в библиотечных вызовах или обработчиках сигналов. Для нормального функционирования libgpuvm память для её динамических структур данных выделяется отдельным аллокатором, который получает страницы от ОС. Таким образом, страницы данных, используемые libgpuvm, никогда не защищаются.

Поскольку финальное копирование данных в реализациях OpenCL выполняется в режиме пользователя, перед началом копирования необходимо снять защиту с региона памяти. В многопоточной среде это означает, что остальные потоки могут начать использовать данные до того, как они станут актуальными. Более того, если страничное прерывание возникнет в одном из рабочих потоков OpenCL, защита должна быть снята сразу для обеспечения нормальной работы приложения; таким образом, та же проблема возникает и в случае одного потока приложения.

Мы рассматривали два способа решения этой проблемы. Первый заключается в остановке всех потоков приложения перед снятием защиты, и их возобновлении после завершения копирования. Хотя это решение работает, остановка потоков является узким метсом в многопоточных приложениях. Вторым способом избежать снятия защиты перед копированием является запись данных по другому виртуальному адресу, который соответствует тем же страницам физической памяти. И хотя код в Linux-ядре для записи в файл

`/proc/self/mem` существует, сейчас он по умолчанию отключён [9]. Поэтому, для обеспечения стабильной работы, мы остановились на 1-м варианте, связанном с остановкой потоков.

Для остановки потоков текущая реализация использует полный список потоков процесса. На Linux его можно получить путём чтения каталога `/proc/self/task`. Поскольку чтение каталога каждый раз занимает много времени, а новые потоки порождаются редко, список потоков кэшируется. При каждой остановке потоков проверяется время обновления каталога, и он повторно читается только в случае, если он изменился с последнего раза.

С каждым потоком при помощи упорядоченного по его идентификатору двоичного дерева ассоциирован семафор. Чтобы остановить поток, ему отправляется сигнал реального времени, по которому вызывается зарегистрированный libgpuvm обработчик. Обработчик просто блокируется на семафоре потока; чтобы возобновить поток, семафор разблокируется.

Исполнение некоторых потоков не должно останавливаться. Например, не должны останавливаться рабочие потоки OpenCL, так как это приведёт к взаимным блокировкам. Отдельно хранится список идентификаторов потоков, которые не должны останавливаться; он проверяется, когда строится бинарное дерево с семафорами. Потоки OpenCL порождаются при создании контекстов и очередей; чтобы определить их идентификаторы, вводится специальный библиотечный вызов `gpuvm_pre_init()`, который должен быть вызван один раз до и один раз после создания контекстов и очередей. Оба раза он проверит список потоков, а а разницу занесёт как потоки, которые не останавливаются. Разумеется, рабочие потоки libgpuvm также не должны останавливаться. В настоящее время таких потока два, каждый обрабатывает запросы из своей очереди. *Поток снятия защиты* снимает защиту памяти с регионов, а также отвечает за остановку и запуск других потоков. *Поток копирования* инициирует передачу данных с устройства на хост, и дожидается её окончания.

Далее приводится описание работы библиотечных вызовов и обработчиков сигналов libgpuvm. Вызов `gpuvm_init()` инициализирует внутренние структуры данных libgpuvm, устанавливает обработчики сигналов и порождает рабочие потоки. Все остальные вызовы используют глобальную блокировку чтения-записи; это позволяет упростить архитектуру системы. Обработчик `SIGSEGV`, `gpuvm_kernel_begin()` и рабочие потоки ставят глобальную блокировку на чтение; остальные вызовы используют блокировку на запись. Кроме того, используется блокировка чтения-записи на бинарное дерево семафоров потоков. Поток снятия защиты блокирует блокирует её на запись, когда добавляются новые потоки блокируют её на чтение.

Вызов `gpuvm_link()` создаёт для новых хост-массивов внутреннюю структуру данных, разделяет их на подрегионы и находит регионы для вставки подрегионов или создаёт новые, а также ассоциирует буфер на устройстве с хост-массивом. Хост-массив также проверяется по дереву регионов; если уже существует массив с пересекающимся, но не полностью совпадающим диапазоном адресов, возвращается ошибка. Вызов `gpuvm_unlink()` снимает связь хост-массива с буфером устройства. Если это был последний буфер, ассоциированный с массивом, удаляется сам массив, а также связанные с ним подрегионы и, если в них больше нет подрегионов, регионы. Если диапазон адресов региона был защищён, защита снимается.

При вызове обработчика сигнала `SIGSEGV` по страничному прерыванию адрес обращения проверяется по дереву регионов. Если адрес не относится к региону libgpuvm, вызывается предыдущий обработчик сигнала. Если же адрес относится к обслуживаемому libgpuvm региону, регион передаётся потоку снятия защиты. После снятия защиты регион получает статус *ожидающего* и передаётся потоку копирования для актуализации. Поток копирования дожидается завершения копирования всех подрегионов, после чего ставит сообщение об этом в очередь потока снятия защиты. При получении этого сообщения поток снятия защиты уменьшает счётчик ожидающих регионов. Как только этот счётчик достигает нуля, остановленные потоки возобновляются.

Поток, на котором произошла обработка сигнала, дожидается снятия защиты. В случае, если это рабочий поток OpenCL, его исполнение возобновится сразу после снятия защиты, обеспечивая нормальную работу приложения. Если же это был поток приложения, он будет остановлен, и возобновлён только после актуализации данных.

По вызову `gpuvm_kernel_begin()`, данные актуализируются на устройстве. Для этого для каждого подрегиона выполняются следующие действия:

- если подрегион актуален на том же устройстве, копирования не выполняется
- если подрегион актуален на хосте, он копируется на устройство и обозначается как актуальный на устройстве
- если подрегион актуален на другом устройстве, он сначала актуализируется на хосте путём вызова страничного прерывания на подрегионе; после этого выполняются те же действия, что и в предыдущем пункте

По вызову `gpuvm_kernel_end()`, каждый из подрегионов хост-массива помечается как актуальный на устройстве, где он был использован, после чего на соответствующие регионы ставится защита, если она не была установлена.

4. Использование libgpuvm

В качестве примера, libgpuvm используется для обеспечения прозрачного совместного использования массивов между хостом и ГПУ с низкими накладными расходами в системе NUDA. NUDA (= Nemerle Unified Device Architecture) представлет собой систему расширения для языка Nemerle [10] для поддержки высокоуровневого программирования ГПУ. Ранее в NUDA использовался подход автосинхронизации, что потребовало введения новых типов для ГПУ-массивов. Для конечных пользователей эти типы были скрыты за аннотациями. И хотя это хорошо работает для массивов, которые используются только в той функции или классе, где они объявлены, в случае использования массива в нескольких функциях это выглядит не очень хорошо. У каждой такой функции требуется добавить дополнительные аннотации на параметры, а это публичная функция, требовалось создавать её версию, которая работает с обычными массивами.

С использованием libgpuvm стало возможным снизить накладные расходы на передачу данных, и при этом предоставить конечному пользователю интерфейс, не отличающийся от интерфейса обычных массивов. Для обеспечения поддержки libgpuvm в NUDA пришлось добавить дополнительный код, в основном для ленивого выделения буферов на устройстве и связывания массивов. В передаваемый на ГПУ код была добавлена поддержка обычных массивов в дополнение к ГПУ-массивам.

Одной из проблем, которые требовалось решить, было удаление выделенных на ГПУ массивов. Текущая версия NUDA работает со средой выполнения mono версии 2.0.1, в которой по умолчанию используется консервативный сборщик мусора Бёма [11]. Он не перемещает выделенные объекты в памяти, что упрощает интеграцию. Когда места для очередного буфера на устройстве становится недостаточно, сборщик мусора вызывается вручную. После этого, мы проходим по списку массивов, которые связаны с буферами на устройстве, и удаляем буфера тех из них, которые были собраны сборщиком мусора. Если даже после этого памяти для буфера на устройстве нет, выбрасывается исключение.

Здесь мы сравниваем подходы libgpuvm автосинхронизации и полного копирования на примере реализации поддержки массивов на ГПУ в NUDA. Сравниваются простота использования и накладные расходы. В качестве модельной задачи мы используем программу, которая выполняет фиксированное количество итераций вида $x^{n+1} = Ax^n + b$, т.е. последовательность умножения матрицы на вектор. Мы специально выбрали задачу, в которой

вычислительная нагрузка невелика по сравнению с объёмами передачи данных, чтобы лучше увидеть накладные расходы; в более вычислительно ёмких приложениях они будут ещё ниже. Код состоит из двух функций, первая из которых содержит внешний цикл по n, а вторая реализует умножение матрицы на вектор. Код для libgpuvm и полного копирования одинаковый и приведён на рис. 2. Код для автосинхронизации приведён на рис. 3. Код, реализующий инициализацию массивов, замеры времени и печать результата опущен для краткости. Мы также предполагаем, что матрица уже транспонирована, для более эффективной реализации на ГПУ. В представленном на рисунках коде `do` — макрос цикла произвольной размерности, `nuwork()` — макрос, отправляющий цикл на ГПУ и задающий размер блока рабочей группы, `<->` — макрос обмена значений переменных. Дополнительно, на рис. 3 можно видеть аннотации `nunew` и `nuarr`, применение которых соответственно к операции выделения памяти под массив и к объявлению переменных-массивов превращает их в операции и объявления ГПУ-массивов. Эти аннотации позволяют сделать код более удобным для чтения.

```
iterate(niters : int, n : int) : void {
  mutable x = array(n) : array[float];
  mutable x1 = array(n) : array[float];
  def a = array(n, n) : array[float];
  def b = array(n) : array[float];
  // ... initialize
  do(it in niters) {
    matvecmul(x1, a, x, b, n);
    x1 <-> x;
  }
  // ... print x
}  // iterate()
matvecmul(x1 : array[float], a : array[2, float], x : array[float],
  b : array[float], n : int) : void {
  nuwork(128) do(i in n) {
    mutable r = b[i];
    do(j in n) r += a[j, i] * x[j];
    x1[i] = r;
  }
}  // matvecmul()
```

Рис. 2. Код для подходов libgpuvm и полного копирования

Мы используем две тестовые системы, характеристики которых приведены в таблице 1. Там же приведены измеренные нами затраты на выполнение ряда важных для нас операций. Было проведено 5 типов экспериментов: для полного копирования (fullcopy), для автосинхронизации (auto-sync), для libgpuvm (libgpuvm), а также дополнительные тесты libgpuvm-copy и libgpuvm-false. Тест libgpuvm-copy использует libgpuvm, но добавляет между итерациями обращения к данным на хосте, что приводит к полному копированию данных на каждой итерации. Этот тест демонстрирует накладные расходы в (маловероятном) худшем случае использования libgpuvm, когда все данные попеременно используются на ГПУ и на хосте. В тесте libgpuvm-false также используется libgpuvm, но между итерациями происходит обращение к первой странице одного из векторов, что вызывает копирование одной страницы данных обратно на хост. В NUDA (и mono) это может возникнуть в том случае, если вызвать на массиве какой-то метод, например, получение длины. Это связано с тем, что заголовок массива хранится вместе с его данными. По сути, данный тест демонстрирует накладные расходы в типичном случае ложного совместного использования, отсюда его

```
iterate(niters : int, n : int) : void {
  mutable x = nunew array(n) : array[float];
  mutable x1 = nunew array(n) : array[float];
  def a = nunew array(n, n) : array[float];
  def b = nunew array(n) : array[float];
  // ... initialize
  do(it in niters) {
    matvecmul(x1, a, x, b, n);
    x1 <-> x;
  }
  // ... print x
}  // iterate()
matvecmul(nuarr x1 : array[float], nuarr a : array[2, float],
  nuarr x : array[float], nuarr b : array[float], n : int) : void {
  nuwork(128) do(i in n) {
    mutable r = b[i];
    do(j in n) r += a[j, i] * x[j];
    x1[i] = r;
  }
}  // matvecmul()
```

Рис. 3. Код для подхода автосинхронизации

название.

Результаты работы тестов для ГПУ NVidia и AMD и представлены в таблицах 2 и 3, соответственно. Использовался размер вектора 2048 (размер матрицы 2048×2048), и выполнялись 1000 итераций умножения матрицы на вектор. Перед основным циклом выполнялся цикл "разогрева", чтобы скомпилировать байт-код и ядра ГПУ; время его выполнения исключено из приведённых данных. Времена приведены в микросекундах, в расчёте на одну итерацию. Времена исполнения ядра и копирования измерялись при помощи профилировки OpenCL, как разность между временами `COMMAND_START` и `COMMAND_END`; использовались только блокирующие команды OpenCL. Остальные времена измерялись при помощи функции `gettimeofday()`. Более конкретные значения временных параметров:

- *общее время* — время, затраченное на одну итерацию
- *время устройства* — время исполнения ядра на устройстве
- *время копирования* — время копирования данных на устройство и с него
- *время накладных расходов* — время накладных расходов OpenCL (по большей части) и NUDA на вызов ядра и копирование данных; оно измеряется как разность между временем выполнения команд, измеренных вручную, и временем исполнения команд устройством, полученное при помощи профилировки OpenCL
- *время обработки сигнала* — время, затраченное на обработку сигнала, начиная от момента перед остановкой потоков, и заканчивая моментом после возобновления последнего потока
- *прочее время* — время, которое остаётся от общего времени после вычитания остальных факторов

Таблица 1. Системы, используемые для тестирования libgpuvm + NUDA

Название	**fermi**	**amd**
ГПУ	NVidia Fermi C2050	AMD Radeon HD5830
Версия драйвера	280.13	11.9
Реализация OpenCL	CUDA 4.0	AMD APP 2.5
ЦПУ	Intel Core 2 Quad, 2.4 GHz	Intel Core 2 Duo, 2.2 GHz
Версия ОС	Debian 6.0, 2.6.32-amd64	Ubuntu 11.10, 2.6.38-13-server
Время запуска ядра	55 μc	100 μc
Время запуска копирования	150 μc	600 μc
Время обработки сигнала	52 μc	33 μc

Таблица 2. Результаты экспериментов на ГПУ NVidia Fermi C2050

Тест	**full-copy**	**auto-sync**	**libgpuvm**	**libgpuvm copy**	**libgpuvm false**
Общее время, μc	20655	1058	1077	25678	1247
Время устройства, μc	1005	998	998	1005	999
Время копирования, μc	18097	10	10	21542	16
Время накладных расходов, μc	1447	44	44	2322	136
Время обработки сигналов, μc	0	0	0.38	538	52
Прочее время, μc	106	6	25	271	44

Ясно, что время исполнения ядра на устройстве не зависит от используемого подхода. В варианте с полным копированием ясно доминирует копирование, в то время как в вариантах с libgpuvm или автосинхронизацией, накладные расходы “распределяются” на большое число итераций, и в среднем доминирует исполнение на устройстве. Это именно то, к чему стремятся разработчики ГПУ-приложений. В целом, накладные расходы libgpuvm по сравнению с автосинхронизацией составляют 1–2%, в основном из-за большего числа операций при установке параметров ядра, примерно на 6 μc на параметр. Но мы считаем, что упрощение интерфейса передачи данных более чем компенсирует дополнительные накладные расходы.

Тест libgpuvm-copy демонстрирует наихудший случай. Накладные расходы вырастают на 25–30% по сравнению с просто полным копированием. Однако лишь 4% составляет прирост за счёт обработки страничных прерываний. Ещё 2–4% занимают прочие расходы, в

Таблица 3. Результаты экспериментов на ГПУ AMD Radeon HD5830

Test	**full-copy**	**auto-sync**	**libgpuvm**	**libgpuvm copy**	**libgpuvm false**
Общее время, μc	23273	1399	1436	30268	2367
Время устройства, μc	1322	1321	1321	1322	1321
Время копирования, μc	17549	7	8	17877	82
Время накладных расходов, μc	4330	69	82	10181	877
Время обработки сигналов, μc	0	0	0.23	446	33
Прочее время, μc	72	2	24	442	54

частности, выполнение вызова `mprotect()` на матрице размером 64МБ, и связанная с этим очистка кэша памяти и кэша трансляции. Оставшиеся 15–25% — это дополнительные расходы на исполнение команд OpenCL: хотя объём копирования тот же, количество команд составляет 18 с использованием libgpuvm, и 8 просто с полным копированием. В случае AMD это отражается как рост накладных расходов, в случае NVidia — как рост времени копирования. Может быть, это связано с различными реализациями копирования, или с различной трактовкой “времени исполнения копирования на устройстве”. Однако ясно, что в этом случае большинство накладных расходов связано с реализацией OpenCL, а не с libgpuvm напрямую.

Тест libgpuvm-false представляет промежуточный случай, когда накладные расходы связаны только с копированием небольшого массива. Дополнительные расходы для ГПУ NVidia составляют 15% по сравнению с использованием данных только на ГПУ. Половина этого связана с libgpuvm, а половина с реализацией OpenCL. В случае AMD накладные расходы составляют 60%, большая часть которых составляет время инициализации копирования OpenCL. Это совершенно точно является проблемой реализации OpenCL, и мы ожидаем, что в следующих версиях AMD APP SDK проблема будет решена, и накладные расходы упадут до уровня реализации NVidia. Но даже с таким уровнем накладных расходов использование libgpuvm даёт на порядок лучшие результаты, чем полное копирование.

5. Обсуждение и дальнейшая работа

Мы предложили подход к обеспечению абстракции разделяемой памяти между хостом и ГПУ. Наше решение поддерживает динамически выделенную хост-память и может работать с ведущими реализациями OpenCL в многопоточной среде с несколькими ГПУ. Продемонстрирована интеграция libgpuvm в NUDA для обеспечения более удобного обмена данными между хостом и ГПУ. Проведённые эксперименты показали, что даже в случае некоторого ложного совместного использования подход libgpuvm на порядок быстрее, чем полное копирование. А в маловероятном худшем случае большая часть накладных расходов связана с реализацией OpenCL, а не с обработкой сигналов.

Возможно несколько направлений будущего развития. Прежде всего, это реализация поддержки других интерфейсов взаимодействия с ГПУ, прежде всего CUDA. Во-вторых,

это поддержка использования ГПУ-памяти как кэша для хост-данных. Это упростит интеграцию нашей библиотеки в существующие языки. В-третьих, это более эффективная поддержка многопоточных приложений, с копированием данных до снятия защиты. Наконец, это более эффективная поддержка различных паттернов совместного использования данных хостом и ГПУ. Например, это использование данных на ГПУ только на чтение, что достаточно типично, или же использование на хосте только некоторых порций ГПУ-массива.

Разработанная библиотека может быть использована для более удобной и эффективной организации взаимодействия с ГПУ в прикладных библиотеках и системах времени выполнения в реализациях языков программирования. В связи с появлением более высокоуровневых инструментов работы с ГПУ становится актуальным.

Литература

1. NVidia Corporation. NVidia CUDA C Programming Guide, Version 4.0, 2011. — 05.
2. Khronos Group. The OpenCL Specification, version 1.2, document revision 15, 2011. — 11.
3. The Portlang Group. PGI Accelerator Programming Model for Fortran & C, 2010. — 11. http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf.
4. Caps Enterprise. CAPS HMPP Workbench User Guide, Version 2.3.1, 2010. — 06.
5. OpenACC Application Programming Interface, Version 1.0, 2011. — 11. http://www.openacc-standard.org/Downloads/OpenACC.1.0.pdf.
6. Adinetz A. V. libgpuvm project on GitHub. 2011. — 12. http://github.com/canonizer/libgpuvm.
7. Gelado I., Stone J. E., Cabezas J. et al. http://dx.doi.org/http://doi.acm.org/10.1145/1736020.1736059An asymmetric distributed shared memory model for heterogeneous parallel systems // Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems. ASPLOS '10. New York, NY, USA: ACM, 2010. Pp. 347–358. http://doi.acm.org/10.1145/1736020.1736059.
8. Adinetz A. V. Extran and NUDA Programmer's Guide. 2011. — 01. http://sourceforge.net/projects/nuda/files/0.0.5/extran-guide.pdf/download.
9. enable writing to /proc/pid/mem. http://lwn.net/Articles/435018/.
10. Skalski K., Moskal M., Olszta P. Meta-programming in Nemerle.
11. Boehm H., Demers A., Shenker S. Mostly Parallel Garbage Collection // Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation. SIGPLAN Notices 26. New York, NY, USA: ACM, 1991. — 06. Pp. 157–164.