Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Мисонижник Александр Владимирович

Реализация программной транзакционной памяти на архитектуре CUDA

Курсовая работа

Научный руководитель:
ст. преп. Сартасов С. Ю.

Санкт-Петербург
2017

Оглавление

Введение

Научные исследования в области GPGPU показывают, что проблема обеспечивания согласованного одновременного доступа к разделяемой памяти является более актуальной, чем когда-либо. В среде с тысячами потоков увеличивается вероятность появления ошибок, а такие ситуации как *livelock* и *deadlock* становятся более распространенным явлением. Синхронизация в таких средах, как правило, достигается с помощью блокировок построенных из атомарных операций. Тем не менее, синхронизация на основе блокировок требует значительного количества усилий от программиста для достижения функциональной корректности и желательной производительности. Поэтому, чтобы получать наибольшую выгоду из использования GPU в широком спектре реальных нагрузок, крайне важно упростить параллельное программирование для GPU архитектуры.

Транзакционная память (*Transactional Memory*, TM)[3] позволяет использовать атомарные операции на произвольном множестве ячеек памяти. Атомарное и изолированное исполнения блоков кода реализуется во внутренней структуре TM. Транзакции устраняют ошибки, обычно связанных с блокировками (*priority inversion, convoying, deadlock*)[3].

В этой работе речь пойдёт о STM (*Software Transactional Memory*) - TM реализованной на программном уровне. Существует ещё *Hardware Transactional Memory*, это реализация транзакционной памяти на уровне железа.

1. Цель работы

Целью работы является реализация алгоритма STM на архитектуре CUDA.

Вся работа разбита на список задач.

1. Составить обзор-описание STM
2. Сделать обзор на существующие алгоритмы реализации STM на GPU
3. Написать собственную реализацию, взяв за основу готовые алгоритмы, протестировать её на работоспособность
4. Модифицировать алгоритм STM

2. Описание STM

Транзакция - это конечная последовательность машинных команд, которые выполняются одним потоком и удовлетворяют следующим свойствам:

- Сериализуемость (*Serializability*): Транзакции выполняются последовательно, а это значит, что этапы одной транзакции не перемешиваются с этапами другой. До фиксирования ни один поток не знает об изменениях, которые должна внести эта транзакция

- Атомарность (*Atomicity*): Каждая транзакция совершает последовательность предварительных изменений в общей памяти. Когда транзакция завершается, она либо фиксируется, делая изменения видимыми для других потоков, либо прерывается, откатывая все изменения

STM предоставляет следующие примитивные инструкции для доступа к памяти:

- *TRead* считывает значение из разделяемой памяти и записывает его в приватное поле транзакции

- *TWrite* указывает, что в данное поле разделяемой памяти должно быть записано значение

Так же STM предоставляет следующие инструкции для изменения состояния транзакции:

- *TVaildate* вызывается перед фиксированием транзакции. Это удается только тогда, когда никакая другая транзакция не обновила любое значение в наборе данных транзакции, и никакая другая транзакция не прочитала любое значение в наборе для записи этой транзакции. Если это удастся, изменения сделанные в ее наборе для записи становятся видимыми для других потоков. Если это не удается, все изменения в наборе записи отбрасываются. В любом случае, *TValidate* возвращает индикатор успеха (*true*) или провала (*false*)

- *TAbort* откатывает все изменения тразакции.
- *TCommit* пытается сделать предварительные изменения транзакции постоянным. Это удается только тогда, когда *TValidate* возвращает *true*

Комбинируя эти примитивы, программист может определить индивидуальные операции чтения, записи и обновления, которые работают на произвольных областях памяти, а не на отдельных ячейках.

3. Обзор существующих решений

3.1. GPU-STM

GPU-STM[10] - это STM основанная на блокировках и машинных словах. В основе GPU-STM используется TBV (*time-based validation*) - механизм детектирования конфликтов, использующий глобальные версионные блокировки для управления всей памятью. Каждая версионная блокировка хранит идентификатор текущей версии определенной области памяти. При получении доступа к области памяти, транзакция сохраняет идентификатор версии. Перед фиксированием транзакции сравниваются идентификаторы в транзакции и блокировке. Если они различаются, то транзакция отменяется. Такой подход влечет за собой появление ложных срабатываний, если две транзакции пытаются получить доступ к значениям, хранящимся в одной области памяти, так как значения могут быть разными, а значит операции над ними могут проводиться параллельно. Это в свою очередь влечет ухудшение производительности. SIMT (*Single Instruction Multiple Threads*)[8] модель усиливает побочный эффект от ложных срабатываний. Чтобы уменьшить количество ложных срабатываний GPU-STM использует VBV (*value-based validation*): каждая транзакция хранит значение ячейки, к которой получила доступ, и сравнивает его с текущим значением в ячейке перед фиксированием. Последовательное применение этих двух методов, называющееся *hierarchical validation*, обеспечивает гораздо меньший уровень ложных срабатываний, что является плюсом для SIMT модели, несмотря на ухудшение производительности.

Для решения конфликтов взятия блокировки используется алгоритм *encounter-time lock sorting*: все блокировки отсортированы в определенном порядке, занимание блокировок происходит в том же порядке, в котором они отсортированы[10]. Это значит что во всех потоках блокировки будут заниматься в определенном порядке.

Так же GPU-STM предоставляет некоторую изолированность каждой транзакции, но не гарантирует непрозрачность (*opacity*)[2], так как

транзакции, которые должны прерваться, продолжают выполняться, пока не провалят валидацию. Непрозрачность - основной критерий корректности TM, поэтому эта реализация не подходит для того, чтобы быть основой для собственной реализации.

3.2. Lightweight Software Transactions on GPUs

Этот алгоритм имеет три варианта реализации: ESTM, PSTM и ISTM[4]. Каждая из этих реализаций нацелена на различные нагрузки, так что программист может использовать ту, которая лучше подходит для соответствующей ситуации.

Все три решения используют спекулятивную (*speculative execution*) запись[9], а значит им необходимо использовать журнал отмен (список измененных ячеек памяти, а также значения, которые хранили в них до внесения изменений). Чтобы избежать *livelock*-ситуаций при прерывании транзакций, используется экспоненциальное откладывание (*exponential backoff*)[6].

ESTM использует сложную разделяемую структуру метаданных. Эта структура, называемая теневой памятью (*shadow-memory*)[7], имитирует метаданные других STM, такие как массивы блокировок или счетчики версий. Она состоит из теневых записей, которые содержат в себе идентификатор версии, идентификатор потока, который получил доступ последним, бит отвечающий за запись и чтение, бит отвечающий за разделяемый доступ и блокировку, которая отвечает за возможность изменения метаданных в структуре.

Конфликты, возникающие из-за записи после чтения, обнаруживаются при помощи этой структуры. Если бит записи установлен, транзакция сразу же отменяется. Все остальные метаданные являются сокрытыми внутри потока и состоят из журналов чтения и отмены. Для прохождения валидации транзакция сначала проверяет, была ли изменена теневая запись, затем проверяет была ли изменена область памяти. Если на каком-то из этих шагов были выявлены изменения, то транзакция прерывается, а изменения в теневой записи откатывают-

ся с помощью журнала отмены. В случае успешной фиксации теневая запись обновляется.

PSTM использует подобную систему, однако рассчитан на нагрузки, которые часто пишут в те же области памяти, что и считывают. Вместо битов, указывающих была ли прочитана или изменена область памяти, PSTM использует идентификатор потока, который получил доступ последним. Если он равен нулю (зарезервированный идентификатор) или идентификатору текущей транзакции, транзакция продолжается, иначе прерывается. Когда транзакция прерывается или фиксируется, она освобождает все теневые записи, которые она изменила, записав ноль вместо значения идентификатора потока. Эти операции используют атомарную операцию CAS (*Compare-and-Swap*), а это приравнивается к использованию *fine-grained* блокировок. Для избежания *livelock*-ситуаций используется экспоненциальное откладывание.

ISTM реализованна на версионных блокировках, чтобы позволить операции невидимого чтения (*invisible read*)[1]. В этой реализации теневая память содержит версионные блокировки. Младший значимый бит счетчика версий используется в качестве блокировки. Этот тип блокировки очень похож на другие *time-based* STM. Если поток пытается прочитать заблокированную область памяти, то он прерывается, если он не владеет этой блокировкой. Если блокировка свободна, поток считывает значение, сохраняет его адрес и версию в локальный журнал чтения. Блокировка занимается только для записи. Для фиксирования транзакции ISTM проверяет считанное локальное значение, сравнивая его с глобальной версией. Если версии различны, поток освобождает занятые блокировки и откатывает изменения, иначе освобождает блокировки и увеличивает их счетчики.

3.3. PR-STM

PR-STM[5] - это основанная на блокировках STM, использующая версионные блокировки для для валидации и нахождения конфликтов. Она работает таким же образом как и ISTM: операции чтения невиди-

мы, однако операции записи отложенные. Глобальная таблица блокировок также отличается от ISTM: отображение не должно быть один к одному, то есть несколько областей памяти могут использовать одну блокировку. Хоть это и может увеличить количество конфликтов, зато уменьшаются расходы на память для хранения метаданных. Такая гибкость очень важна, так как регистровая память GPU является весьма ограниченной.

Самое важное нововведение PR-STM - статический менеджер транзакций. Каждому потоку присваивается уникальный приоритет, и блокировка представляет собой двухэтапный процесс, включающий занимание двух блокировок: блокировку записи и пред-блокировку. Потоки с высоким приоритетом могут забрать себе пред-блокировки у потоков с низким приоритетом. Это значит, что при обнаружении конфликта взятия блокировки, как минимум один поток будет продолжать работу, избегая *livelock* и *deadlock* ситуаций.

По сравнению с GPU-STM, такой метод позволяет получить гораздо меньше накладных расходов. Сортировка блокировок в GPU-STM - нетривиальная операция. Если она происходит во время фиксирования, сложность равна $n * log(n)$.

В PR-STM потоки обнаруживают конфликт только тогда, когда не смогли занять блокировку или когда пытаются прочитать область заблокированную для записи. Валидация начинается с проверки локальных версий значений с глобальными для каждого значения входящего в набор для чтения и записи. Каждое успешное сравнение занимает пред-блокировку области с этим значением. Если все пред-блокировки взяты, валидация продолжается. Далее надо заново проверить все пред-блокировки, не заняли ли какую-то из них другие потоки. Если проверка не пройдена, транзакция прерывается и освобождаются все блокировки, в противном случае транзакция готова к фиксации.

Относительно низкие расходы на память дают большую степень свободы в модификации конструкции этой STM.

3.4. Сравнение

В процессе описания существующих решений были выявлены критерии для выбора основу для собственной реализации STM: универсальность, свобода для модификации, непрозрачность. GPU-STM не поддерживает непрозрачность, к тому же нетривиальная операция сортировки блокировок увеличивает накладные расходы и, как следствие, ограничивает свободу для модификации . Lightweight Software Transactions on GPUs имеет целых три реализации, каждая из которых опирается на различные условия использования, а значит теряет универсальность. PR-STM же подходит по всем критериям: непрозрачность, универсальность, имеет низкие расходы на память, а следовательно даёт большую свободу для модификации. Исходя из сравнения, за основу для реализации был взят алгоритм PR-STM.

4. Реализация

Работа GPU существенно отличается от работы CPU, и это необходимо учитывать. В дополнение к большому числу потоков, группы потоков на GPU выполняются как части варпов (*warps*). Потоки в одном и том же варпе используют один счетчик инструкций и, таким образом, выполняют одну и ту же команду одновременно. К тому же возможны *livelock* и *deadlock* ситуации, потому что потоки из одного варпа не могут скоординировать свои обращение к блокировкам.

Для предотвращения *livelock* и *deadlock* ситуаций будет использоваться алгоритм *"lock-stealing"*, который требует, чтобы каждому потоку был назначен статический приоритет. Это позволяет потоку с приоритетом n отнять блокировку, которая в настоящее время принадлежит любому потоку с приоритетом меньше чем n. Так как каждый поток имеет уникальный приоритет, это устраняет *deadlock* ситуации, поскольку любой поток может определить своё следующее действие при обнаружении заблокированных данных. *Livelock* ситуации так же устраняются, так как потоки не будут постоянно красть друг у друга блокировки.

Прежде чем совершить фиксацию, поток пытается провести валидацию, пред-заблокировав общие данные. Пред-блокировка может быть отобрана потоком с наибольшем приоритетом. Если валидация прошла успешно, поток может зафиксировать транзакцию. Реализованы невидимое чтение и возможность прервать транзакцию, если конфликт будет обнаружен на ранней стадии. Это позволяет снизить затраты на ложные конфликты, когда поток прерывает работы при обнаружении данных, заблокированных транзакцией, которая сама будет отменена в будущем.

Реализация написана для архитектуры NVIDIA CUDA на языке CUDA C, сборка производилась компилятором NVCC. Далее будут рассмотрены детали реализации.

4.1. Метаданные

Разделяемая память представляет из себя массив элементов одного типа. Для реализации конкурентного доступа к этой памяти используется два типа метаданных: глобальные метаданные, которые используются всеми потоками, и локальные метаданные, которые являются приватными для каждого отдельного потока.

4.1.1. Глобальные

Global Lock Table. Таблица блокировок должна быть доступна для всех потоков GPU, следовательно, она находится в глобальной памяти. Каждому элементу из массива общих данных соответствует уникальная блокировка в таблице блокировок. Чтобы повысить масштабируемость системы, можно изменить количество элементов, которые покрываются одной блокировкой. За это отвечает хэш-функция, которая преобразует указатель на данные в индекс элемента в таблице блокировок. Например, когда хэширующая функция имеет конфигурацию 1:1, каждый элемент из общих данных имеет свою собственную блокировку. Хотя эта конфигурация требует наибольшего объёма памяти, она минимизирует вероятность ложных конфликтов связанных с общими блокировками. Каждая запись в блокировке представляет из себя целое число без знака, стоящее из версии значения (11 бит), приоритета владельца блокировки (19 бит), показателя пред-блокировки (1 бит) и блокировки (1 бит).

4.1.2. Локальные

- *Local Read Set* - набор данных для чтения, каждая из которых состоит из указателя на память, версии значения и значения, считанных текущим потоком
- *Local Write Set* - набор данных для записи, содержащих указатель на память и значение, которое поток хочет в эту память записать

- *Local Lock Set* - набор данных, которые состоят из индекса блокировки в таблице блокировок, версии блокировки и показателя окончательной блокировки. Использование механизма контроля версий блокировок вместе с индексированием потоков предоставляет данные алгоритму, когда транзакция хочет отнять блокировку

Было рассмотрено три подхода в организации локальных метаданных: динамический массив, односвязный список и статический массив. Первые два основываются на размещении данных в куче в глобальной памяти. Таким образом можно динамически выделять нужное количество памяти для потока. Третий подход обязывает хранить данные в локальной памяти - регистрах и L1 кэше. Это значит, что для каждого потока может выделено только заранее фиксирование количество памяти. Из-за этого возникает ограничение на размер памяти, которую можно редактировать в процессе транзакции.

Первые два похода не подошли из-за слишком маленького размера кучи в глобальной памяти. Их просто нецелесообразно было использовать при работе с достаточно большим количеством потоков, потому что размера памяти не хватает для хранения всем локальных метаданных. Статический массив хоть и имеет ограничения на размер редактируемой памяти, зато даёт большую свободу в выборе количества потоков. В качестве структуры для хранения локальных метаданных был выбран именно статический массив.

4.2. Операции

STM состоит из нескольких функций, которые вызываются во время важных событий во время выполнения транзакции: *txStart, txRead, txWrite, txValidate, txCommit.*

txStart вызывает перед началом или перезапуском транзакции. Функция инициализирует локальные метаданные и устанавливает флаг прерывания транзакции в значение *false.*

txRead вызывается для того, чтобы прочитать общие данные из глобальной памяти. Вызывающий поток проверяет, заблокированы ли общие данные другим потоком, и если это так, то поток устанавливает флаг прерывания транзакции, которые указывает, что транзакция должна прекратить свою работу, и её надо перезапустить. Если данные не заблокированы, поток проверяет, содержатся ли эти данные в *Local Write Set*, и если это так, то поток возвращает сохраненное значение из *Local Write Set*. Если в локальном наборе общие данные отсутствуют, то поток извлекает значение из глобальной памяти. Затем поток добавляет значение и версию блокировки в *Local Read Set* и возвращает значение.

txWrite записывает значение, которое поток хочет записать в общие данные, в *Local Write Set*. Сначала поток проверяет, заблокированы ли данные, и, если это так, устанавливает флаг прерывания транзакции. Если данные не заблокированы, поток проверяет, находятся они в локальном наборе для записей. Если нашлись - поток перезаписывает значение. В ином случае поток создаёт новую запись в *Local Write Set*.

txValidate вызывается то того как может произойти фиксация транзакции. Поток пытается заблокировать все общие данные, которые он намеревается изменить, и выполняет проверку всех данных, которые он прочитал. Поток пытается пред-блокировать данные, чтобы определить, имеет ли он наивысший приоритет. Затем поток проверяет все данные из своего набора для чтения, проверяя, что их версии не изменились. Если проверка прошла успешно, поток пытается заблокировать все данные. Если данные заблокированы, то поток может зафиксировать транзакцию. Если на каком-то из этих шагов произошла неудача, транзакция должна прерваться.

txCommit вызывается только тогда, когда валидация завершилась успехом. Поток записывает все данные из локального набора записи в глобальную память и вызывает функцию `__threadfence()`. Эта функция обеспечивает всем потокам видимость изменений произошедших до вызова функции. Без неё модель памяти GPU может привести к переупорядочиванию инструкций потока, что может привести к несогласо-

ванности общих данных. `__threadfence()` гарантирует, что изменения общих данных в глобальной памяти будут видны всем потокам до того, как будут освобождены какие-либо блокировки. Затем поток обновляет бит версии в глобальной таблице блокировок для каждой блокировки из локального набора блокировок. Бит версии либо инкрементируется, либо изменяется на ноль, если значение версии достигло максимума.

4.3. Политика управления конфликтами

В качестве блокировки используется 32-битное целое число. Блокировки используются для защиты общих данных и реализации *"lock-stealing"* алгоритма. Различные биты блокировки представляю следующее:

- Первые 11 бит блокировки представляют текущую версию этой блокировки. Версия увеличивается при каждом успешном выполнении транзакции с участием этой блокировки
- Биты 12-30 представляют приоритет того потока, который в настоящий момент пред-заблокировал эту блокировку (если такой поток существует). Чем ниже значение, тем выше приоритет
- 31-й бит указывает, была ли эта блокировка предварительно заблокирована. Пред-заблокированный блокировки могут быть отняты у потоков с более низким приоритетом и приобретены потоками с более высоким приоритетом
- Последний бит указывает, заблокирована ли в данный момент блокировка. Как только этот бит установлен, никакие другие потоки не могут получить эту блокировку

Для управления блокировками необходимы три обработчика:

tryPreLock вызывается всякий раз, кода поток пытается предварительно заблокировать общие данные. Для каждой блокировки в локальном наборе блокировок поток проверяет, различны ли версии блокировок в локальном наборе и в глобальной таблице и заблокирована

ли блокировка. Затем поток проверяет, была ли блокировка предварительно заблокирована другим потоком с более высоким приоритетом. Если какое-либо из этих условий истинно, тогда поток освобождает все блокировки, которые предварительно заблокировал и прерывает транзакцию. В ином случае поток пытается предварительно заблокировать блокировку с использованием функции `atomicCAS`. Если CAS терпит неудачу, тогда другой поток должен получит доступ к блокировке. Затем все предыдущие действия повторяются до тех пор, пока транзакция не будет прервана или CAS удастся, а значит поток будет иметь самый высокий приоритет из всех потоков пытающихся предварительно заблокировать эту блокировку.

tryLock вызывается, когда поток успешно блокирует каждую блокировку в своём локальном наборе блокировок. Поток пытается заблокировать каждую предварительно заблокированную блокировку с помощью CAS. Если какой-либо CAS не удаётся, значит блокировка была украдена потоком с более высоким приоритетом, а исходный поток должен освободить все блокировки и прервать транзакцию.

releaseLock вызывается после фиксации или прерывания транзакции. Все блокировки, которые были пред-заблокированы или заблокированы освобождаются. Предварительно заблокированные блокировки должны освобождаться CAS, так как блокировки уже мог украсть другой поток.

4.4. Проверка на работоспособность

Для тестирования был выбрана задача "производитель-потребитель". N потоков пытаются последовательно записать k значений в буфер. M потоков пытаются забрать значения из буфера. $N \neq M$. Работа программы остановится, когда все N запишут все k значений, а буфер окажется пуст. При корректном завершении работы программы проверка считается успешно пройденной.

Каждая операция добавления и забирания элемента была организована в транзакцию. Использовались два целых безнаковых счетчика:

текущее количество элементов в буфере и количество потоков завершивших записывать значения. Для буфера и счетчиков должны обрабатываться одной таблицей блокировок, чтобы можно было изменять их в течении отдной транзакции. Из-за того, что STM может работать только с памятью состоящей из одного типа, буфер состоит из элеметов того же типа, что и счетчики. Было проведено тестирование на видеокарте NVIDIA GeForce GTX 765M, при $N = 10$, $M = 20$, $k = 100000$. Тестирование прошло успешно.

5. Модификация

Реализация, которая рассматривалась выше, не поддерживала возможность использования одной таблицы блокировок для общих данных различных типов. Также нельзя было тонко настроить отображение из таблиц блокировок в общие данные. Именно эти две возможности были добавлены при модификации.

Достигнуто это было путём добавления дополнительных данных при инициализации глобальной таблицы блокировок.

Info Global Lock Table - это массив из элементов типа `uint2`, который является парой из двух целых беззнаковых чисел. Каждый элемент массива разграничивает определённую область общих данных. Первое значение является размером данных, которые надо разграничить, а второе - размером слов, из которых состоят общие данные. Порядок элементов в массиве определяет порядок разбиения общих данных на области, которые разграничиваются с помощью значений из массива.

Из-за того, что теперь можно использовать разные типы данных, пришлось изменить записи в локальных наборах для чтения и записи. Для каждой записи дополнительно нужно хранить размер типа значения, которое было считано или будет записано. Для хранения самого значения используется указатель на локальную память типа `void*`.

По сравнению с изначальной версией, усложнилось вычисление хэш-функции. Если в начальной версии вычислительная сложность была равна $O(1)$, то теперь она равна $O(n)$, где n - количество элементов в массиве textitInfo Global Lock Table.

Тестирование на работоспособность проходило так же, как и для изначальной реализации, только теперь буфер не обязан был состоять из элементов того же типа, что и счетчики. Тестирование прошло успешно.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Исследованы и проанализированы теоретические результаты в области STM для GPU
- Написана и протестирована на работоспособность одна из реализаций STM для CUDA
- При реализации были применены и сравнены различные подходы в организации данных с учетом архитектуры CUDA
- Придумана и реализована модификация для этой реализации, которая упрощает работу с STM

Список литературы

[1] Attiya Hagit, Fatourou Panagiota. Disjoint-Access Parallelism in Software Transactional Memory // Transactional Memory Foundations, Algorithms, Tools, and Applications.

[2] Guerraoui R., Kapalka M. On the Correctness of Transactional Memory, // PPaPP'08. — 2008.

[3] Herlihy M., Moss J. E. B. Transactional memory: architectural support for lock-free data structures, // ISCA'93. — 1993.

[4] Holey A., Zha A. Lightweight Software Transactions on GPUs, // Parallel Processing (ICPP), 2014 43rd International Conference on. — 2014.

[5] Q. Shen C. Sharp W. Blewitt G. Ushaw, Morgan G. PR-STM: Priority Rule Based Software Transactions for the GPU // Euro-Par 2015: Parallel Processing. — 2015.

[6] Wikipedia. Exponential backoff. — URL: `https://en.wikipedia.org/wiki/Exponential_backoff`.

[7] Wikipedia. Shadow memory. — URL: `https://en.wikipedia.org/wiki/Shadow_memory`.

[8] Wikipedia. Single instruction, multiple threads. — URL: `https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads`.

[9] Wikipedia. Speculative execution. — URL: `https://en.wikipedia.org/wiki/Speculative_execution`.

[10] Y. Xu R. Wang N. Goswami T. Li L. Gao, Qian D. Software transactional memory for gpu architectures, // Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. — 2014.