УДК 621.9.048.7

ОПТИМИЗАЦИЯ ВЫЧИСЛЕНИЙ ОБЩЕГО НАЗНАЧЕНИЯ НА ГРАФИЧЕСКОМ УСКОРИТЕЛЕ

Павловский А. В.

Белорусский национальный технический университет, Минск, Беларусь, tactic_pk@mail.ru

GPGPU (англ. General-purpose computing for graphics processing units, вычисления общего назначения на графических процессорах) – техника использования графического процессора видеокарты, который обычно имеет дело с вычислениями только для компьютерной графики, чтобы выполнять расчёты в приложениях для общих вычислений, которые обычно проводит центральный процессор. Это стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры для неграфических данных.

На сегодняшний день существует несколько реализаций данной технологии от различных компаний. Среди них:

- CUDA – технология разработанная компанией Nvidia, позволяющая программистам реализовывать на языке программирования Си (а также C++/C#) алгоритмы, выполнимые на графических процессорах ускорителей GeForce восьмого поколения и старше.
- OpenCL – является языком программирования задач, связанных с параллельными вычислениями на различных графических и центральных процессорах.
- AMD FireStream – технология GPGPU, позволяющая программистам реализовывать алгоритмы, выполнимые на графических процессорах ускорителей от компании ATI.
- DirectCompute – интерфейс программирования приложений (API), который входит в состав DirectX (набора API от Microsoft), который предназначен для работы на IBM PC-совместимых компьютерах под управлением операционных систем семейства Microsoft Windows.

Однако, не смотря на многообразие технологий принципы составления программ использующих графические процессоры (GPU) являются общими, поэтому не имеет смысла глубоко изучать каждую из этих технологий, а достаточно сконцентрироваться на одной из них.

Данная работа посвящена обзору общих принципов программирования расчетных ядер (kernel) и их оптимизации в рамках технологии DirectCompute.

Графические процессоры ориентированы на увеличение пропускной способности памяти по отношению к центральным процессорам (CPU). В графических процессорах имеется множество вычислительных блоков (мультипроцессоров), которые могут обрабатывать информацию параллельно. А так же при помощи специальных техник скрывается аппаратная задержка памяти (latency) – промежуток времени между запросом данных из ячейки памяти и их фактическом получении. Однако для эффективного использования ресурсов GPU необходимо следить за равномерным распределением нагрузки на все доступные мультипроцессоры.

Вычислительная архитектура DirectCompute основана на концепции *одна команда на множество данных* (Single Instruction Multiple Data, SIMD) и понятии мультипроцессора. Концепция SIMD подразумевает, что одна инструкция позволяет одновременно обработать множество данных. Например, команда addps в процессоре Pentium 3 и в более новых моделях Pentium позволяет складывать одновременно 4 числа с плавающей точкой одинарной точности.

Мультипроцессор – это многоядерный SIMD процессор, позволяющий в каждый определенный момент времени выполнять на всех ядрах только одну инструкцию. Каждое ядро мультипроцессора скалярное, т.е. оно не поддерживает векторные операции в чистом виде.

Под устройством (device) в данной работе подразумевается видеоадаптер, поддерживающий DirectCompute. Рассмотривается GPU только как логическое устройство, избегая конкретных деталей реализации.

Хостом (host) в данной работе называется программа в обычной оперативной памяти компьютера, использующую CPU и выполняющую управляющие функции по работе с GPU.

Логически графическое устройство можно представить как набор мультипроцессоров и драйвер DirectCompute.

DirectCompute-программа разбивает параллельную работу на группы потоков и отправляет на выполнение (dispatch) множество групп для выполнения вычислений. Один dispatch – это трехмерная сетка групп (может содержать сотни тысяч потоков). Каждая группа – это трехмерная сетка потоков (может содержать десятки сотен потоков). Каждый поток – это один вызов на исполнение кода программы.

На рисунке 1 показана условная схема модели параллельного выполнения кода

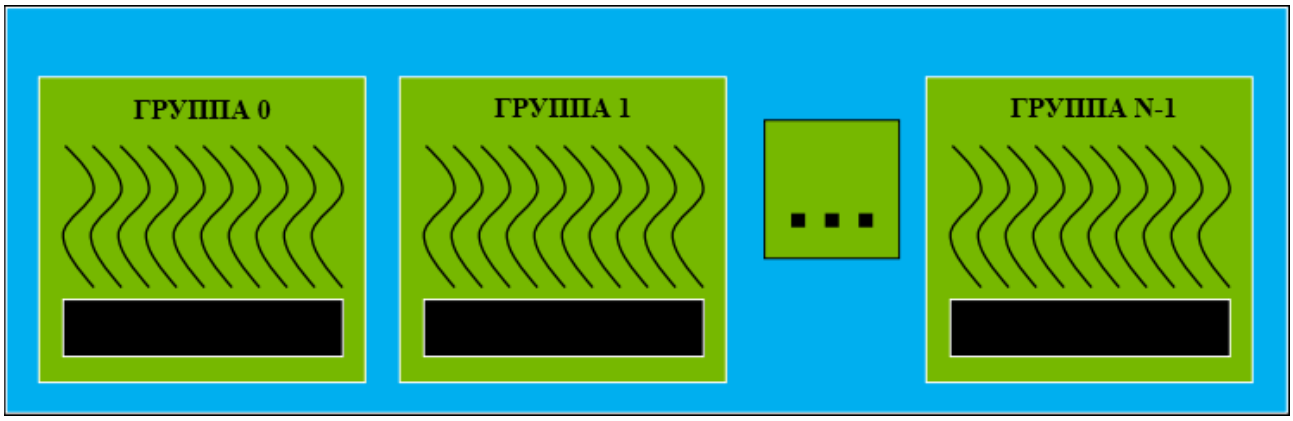

Рисунок 1 – Модель параллельного выполнения DirectCompute

Потоки в пределах одной группы выполняются конкурентно (concurrently), а потоки в разных группах **могут** выполняться конкурентно.

В общем случае для идентификации потоков и групп используются индексы, которые представляют собой трехмерные векторы. Для каждого потока будут известны индекс потока внутри группы (SV_DispatchThreadID) и индекс группы внутри сетки (SV_GroupID). При запуске все потоки будут отличаться только этими индексами. Фактически, именно через эти индексы осуществляется управление, определяя, какая именно часть данных обрабатывается в каждом потоке.

Группа потоков выполняется на мультипроцессоре частями, или пулами, называемыми warp. Размер warp на текущий момент равен 32 потокам. Задачи внутри пула warp исполняются в SIMD стиле, т.е. во всех потоках внутри warp одновременно может выполняться только одна инструкция. Однако современных архитектурах количество процессоров внутри одного мультипроцессора равно 16, а не 32. Из этого следует, что не весь warp исполняется одновременно, он разбивается на 2 части (half-warp), которые выполняются последовательно (т.к. процессоры скалярные).

Для написания оптимального кода ядра необходимо понимать, что ветвления как в GPGPU, так и в SIMD программировании являются одной из причин неэффективного использования вычислительных ресурсов. Это происходит потому, что в один и тот же момент времени все потоки внутри warp могут исполнять только одну и ту же инструкцию. Поэтому для организации возможности ветвления потока исполнения программы, архитекторам аппаратного обеспечения и приходиться использовать неэффективные техники. К примеру по-

следовательно выполнить сначала одну ветвь для потоков которым необходимо идти по ней, а затем другую, для остальных потоков. Таким образом мультипроцессору приходится исполнять обе ветки. Однако технология DirectCompute позволяет не терять производительность если потоки исполнения расходятся в разных пулах warp. То есть вредны только те ветвления, на которых потоки расходятся внутри одного пула потоков warp.

Взаимодействие между потоками (синхронизация и обмен данными) возможно только внутри группы. Обмен данными возможен через разделяемую память, так как она общая для всех задач внутри группы.

В DirectCompute выделяют шесть видов памяти (рис. 2). Это регистры, локальная, глобальная, разделяемая, константная и текстурная память.

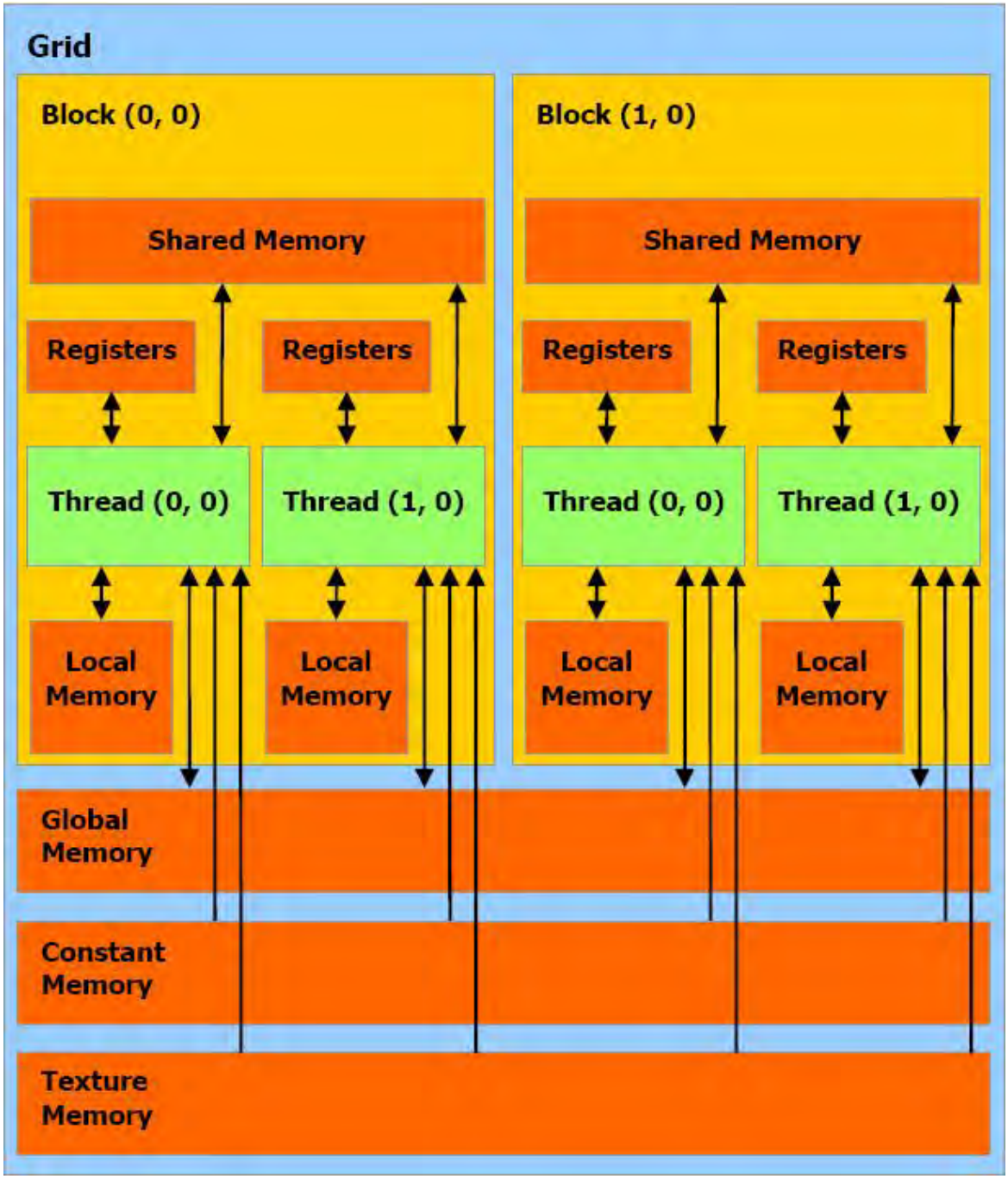

Рисунок 2 – Виды памяти в DirectCompute

Такое обилие обусловлено спецификой видеокарты и первичным ее предназначением, а также стремлением разработчиков сделать систему как можно дешевле, жертвуя в различных случаях либо универсальностью, либо скоростью.

Регистр самая быстрая из представленных память в GPU. Каждый мультипроцессор содержит свой набор регистров недоступный из других мультипроцессоров.

Локальная память может использоваться для хранения локальных данных процедур. Физически локальная память является аналогом глобальной памяти, и работает с той же скоростью.

Глобальная память самая медленная. Она не кэшируется однако, в отличие от всех остальных, позволяет производить операции записи. Глобальная память необходима в основном для сохранения результатов работы программы перед отправкой их на хост.

Разделяемая память – это некэшируемая, но быстрая память. Ее и рекомендуется использовать как управляемый кэш. Ее можно использовать для обмена данными между потоками в пределах одной группы.

Константная память кэшируется и из GPU доступна только для чтения. Доступ к ней в общем случае довольно быстрый.

Текстурная память кэшируется и оптимизирована под выборку 2D данных.

Работа с памятью является немаловажным аспектом в оптимизации вычислительного ядра.

Для работы с глобальной памятью графические процессоры архитектурно оптимизированы для 128-битных операций чтения и записи. То есть для оптимального использования ресурсов GPU, необходимо чтобы каждая манипуляция с памятью изменяла за раз четыре четырех-байтных значения. Потому что при этом придется произвести только одно обращение к памяти. Но так как это не всегда возможно, в современных графических процессорах реализована, так называемая техника объединения операций обращения к памяти (Memory Coalescing Technique).

Глобальная память GPU реализована при помощи динамической памяти с произвольным доступом (DRAM). Чтение из такой памяти относительно очень медленный процесс. Поэтому современная DRAM использует параллельный процесс: каждый раз, когда происходит обращение к произвольному адресу памяти, дополнительно производятся обращения к последовательности адресов, которая включает первоначально запрошенный адрес. И если приложение использует данные из последовательных ячеек памяти, прежде чем обращаться к другой случайной ячейке, производительность динамической памяти приближается к её максимальной пропускной способности.

Наилучшим способом обращения к глобальной памяти является такой при котором все одновременно выполняющиеся инструкции доступа к памяти в пределах одного warp обращаются к последовательным ячейкам памяти. Например, поток 0 обращается к ячейке N, поток 1 – к ячейке N + 1, … поток 31 – к ячейке N + 31. В таком случае все эти обращения объединяются в одно единственное обращение к глобальной памяти.

Еще одним способом увеличения производительности является избегание конфликтов банков разделяемой памяти. Разделяемая память имеет 32 так называемых банков (bank), которые организованы таким образом, что последовательные 32-бита отнесены к последовательным банкам. Пропускная способность разделяемой памяти равна 32-м битам на каждый банк за один такт. Так как она размещена на чипе мультипроцессора, латентность разделяемой памяти в общем случае может быть в сто раз меньше глобальной памяти. Конфликт банка возникает, когда два или более потоков одновременно обращаются к любым байтам внутри одного банка. Если два или более потоков обращаются к любым байтам разных банков – конфликта не возникает.

Так же для увеличения производительности необходимо как можно реже передавать данные с GPU на хост и обратно, так как это самая медленная из всех операций.

В качестве примера использования описанных техник оптимизации расчетов на графических ускорителях предлагается рассмотреть оптимизацию решения задачи суммирования

всех элементов массива. Такая задача отлично укладывается в модель параллельного программирования и послужит отличным примером оптимизации.

В общем случае такая задача решается при помощи подхода основанного на древовидной структуре (рис. 3). Здесь приведена схема сложения элементов массива содержащего восемь целых чисел.

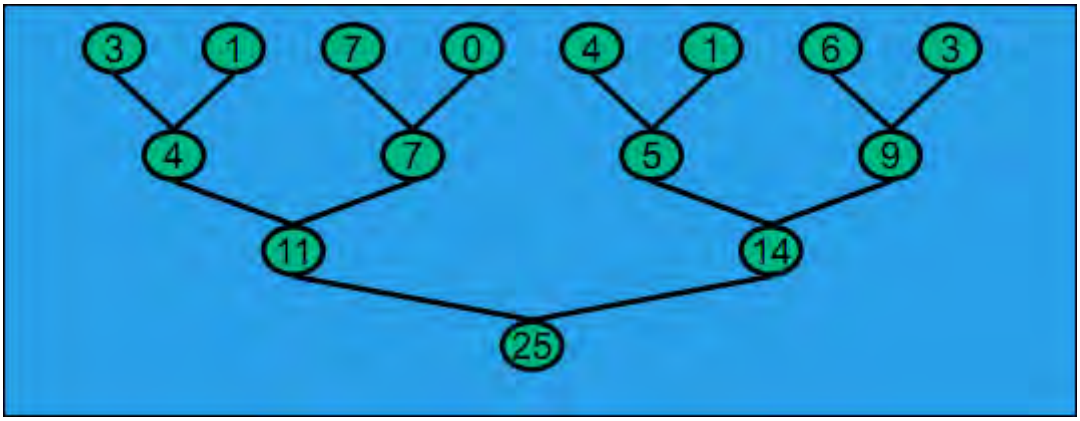

Рисунок 3 – Параллельное сложение элементов массива

Однако для того чтобы была возможность решить задачу для любого размера массива необходимо его разбить на меньшие массивы, размер которых позволяет использовать все мультипроцессоры GPU одновременно, но не превышает общего возможного количества потоков. Каждую такую часть массива обрабатывается, а результат сохраняется в отдельный массив на хосте, который, впоследствии, так же обрабатывается на графическом ускорителе. Таким образом, можно решать данную задачу для массива любого размера.

В качестве метрик, по которым будет анализироваться производительность кода, выбрано время выполнения программы для массива размером 2^{22} и скорость передачи данных в памяти. В качестве аппаратного обеспечения в данном тесте используется видеокарта G80 от Nvidia, максимальная расчетная скорость передачи данных которой составляет 86,4 GB/s.

Шаг 1.

На рисунке 4 показана схема выполнения суммирования с непоследовательной адресацией.

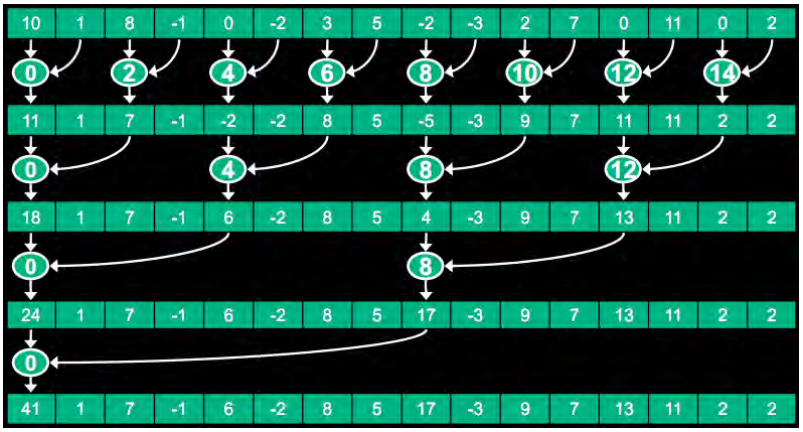

Рисунок 4 – Непоследовательная адресация

В прямоугольниках показаны числа массива в памяти. В овалах показаны индексы потоков, выполняющих суммирование. Стрелками отмечены обращения к ячейкам памяти. Каждая следующая строка обозначает следующий шаг цикла.

Таблица 1 – Результат первого шага

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		

Шаг 2.

На данный момент работают потоки с номерами 0, 2, 4 и т.д. Как можно заметить такой подход ведет к ветвлению для разных потоков в пределах одного wrap. От этого нужно избавляться в первую очередь. Для этого необходимо изменить код программы так, чтобы рабочими потоками были потоки с номерами 0, 1, 2 … (рис. 5).

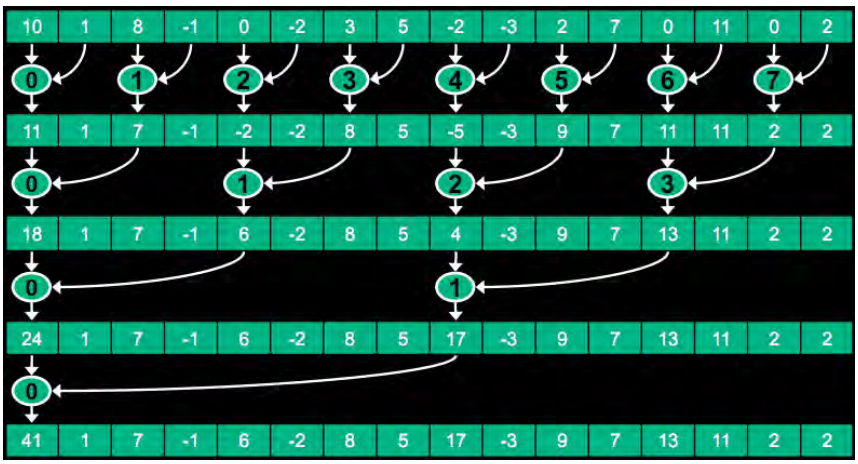

Рисунок 5 – Непоследовательная адресация с последовательными потоками

Теперь ветвления в пределах одного warp не будет.

Таблица 2 – Результат второго шага оптимизации

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		
Шаг 2	3,456	4,854	2,33	2,33

Шаг 4.

Следующим шагом оптимизации будет избавление от конфликтов банков разделяемой памяти. Для этого необходимо избавиться от непоследовательной адресации в потоках. Так как в данном случае происходят одновременные обращения к данным в одном и том же банке разделяемой памяти. На рисунке 6 представлена схема работы программы с последовательной адресацией.

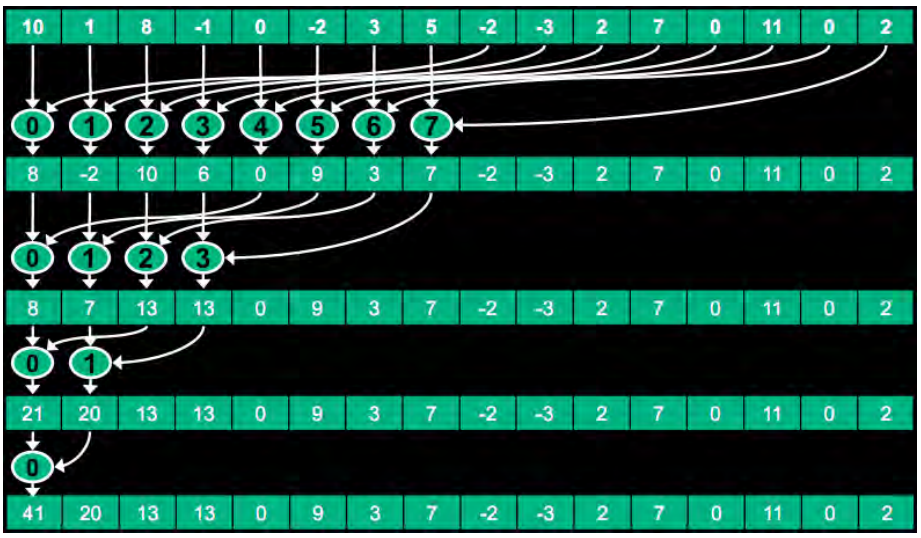

Рисунок 6 – Последовательная адресация

Как видно теперь каждый поток обращается к ячейкам разделяемой памяти с последовательными адресами и конфликта банков не возникает.

Таблица 3 – Результат третьего шага оптимизации

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		
Шаг 2	3,456	4,854	2,33	2,33
Шаг 3	1,722	9,741	2,01	4,68

Шаг 4.

Теперь проблемой является то, что на первом шаге цикла половина потоков простаивает. Оптимизируя на данном шаге, мы уменьшаем количество групп вдвое и вместо единственной загрузки в разделяемую память выполняем две, и на этапе второй загрузки складываем уже загруженные в разделяемую память элементы с только что полученными из общей памяти. Таким образом, мы избавляемся от простаивающих потоков на первом шаге.

Таблица 4 – Результат четвертого шага оптимизации

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		
Шаг 2	3,456	4,854	2,33	2,33
Шаг 3	1,722	9,741	2,01	4,68
Шаг 4	0,965	17,377	1,78	8,34

Шаг 5.

Теперь скорость передачи данных в памяти равна 17 GB/s! Но это еще далеко до расчетного предела в 86 GB/s. Теперь «бутылочным горлышком» являются накладные расходы на лишние инструкции, такие вспомогательные инструкции которые не загружают информацию и не производят вычислений. Например, инструкции циклов.

Так как на каждом следующем шаге цикла количество активных потоков уменьшается, когда активных потоков остаются меньше 32 DirectCompute их всех может одновременно запустит в одном warp. Для этого нужно развернуть цикл и когда остаются меньше 32 активных потоков выполнить инструкции для них вне цикла.

Результат оптимизации представлен в таблице 5.

Таблица 5 – Результат пятого шага оптимизации

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		
Шаг 2	3,456	4,854	2,33	2,33
Шаг 3	1,722	9,741	2,01	4,68
Шаг 4	0,965	17,377	1,78	8,34
Шаг 5	0,536	31,289	1,8	15,01

Шаг 6.

Так как во время компиляции количество потоков в группе известно (для данного GPU оно максимальное количество потоков равно 512) то и количество итераций цикла известно на момент компиляции. Таким образом, можно полностью избавиться от цикла.

Таблица 6 – Результат шестого шага оптимизации

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		
Шаг 2	3,456	4,854	2,33	2,33
Шаг 3	1,722	9,741	2,01	4,68
Шаг 4	0,965	17,377	1,78	8,34
Шаг 5	0,536	31,289	1,8	15,01
Шаг 6	0,381	43,996	1,41	21,16

Шаг 7.

На последнем шаге оптимизации было сделано изменение касательно первоначальной загрузки данных в разделяемую память. Теперь вместо двух чтений из глобальной памяти производится столько, сколько нужно для последующей оптимальной работы. Это количество зависит от общего количества элементов в массиве.

Таблица 7 – Результат седьмого шага оптимизации

	Время выполнения, мс	Скорость передачи данных, GB/s	Увеличение производительности на шаге	Суммарное увеличение производительности
Шаг 1	8,054	2,083		
Шаг 2	3,456	4,854	2,33	2,33
Шаг 3	1,722	9,741	2,01	4,68
Шаг 4	0,965	17,377	1,78	8,34
Шаг 5	0,536	31,289	1,8	15,01
Шаг 6	0,381	43,996	1,41	21,16
Шаг 7	0,268	62,671	1,42	30,04

Таким образом используя представленные техники оптимизации расчетов общего назначения с помощью графических процессоров почти удалось достичь расчетной пропускной способности памяти GPU.

ЛИТЕРАТУРА

[1] CUDA. Best practices. [Electronic resource]. – Электронные данные. – Режим доступа : http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3rYpyYuq2

[2] Bases of GPU optimisation [Electronic resource]. – Электронные данные. – Режим доступа : http://my-it-notes.com/2013/06/bases-of-gpu-optimisation/

[3] DirectCompute Pre Conference Tutorial [Electronic resource]. – Электронные данные. – Режим доступа : http://on-demand.gputechconf.com/gtc/2010/presentations/S12312-DirectCompute-Pre-Conference-Tutorial.pdf

[4] CUDA memory coalescing [Electronic resource]. – Электронные данные. – Режим доступа : http://homepages.math.uic.edu/~jan/mcs572/memory_coalescing.pdf

[5] CUDA: синхронизация блоков [Electronic resource]. – Электронные данные. – Режим доступа : http://habrahabr.ru/post/151897/

[6] Введение в технологию CUDA [Electronic resource]. – Электронные данные. – Режим доступа : http://cgm.computergraphics.ru/issues/issue16/cuda