УДК 004.4

МЕТОДИКА ПОВЫШЕНИЯ ЭФФЕКТИВНОСТИ РЕАЛИЗАЦИИ ЗАДАЧ АЛГОРИТМИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ В ГЕТЕРОГЕННОЙ СИСТЕМЕ НА БАЗЕ ГРАФИЧЕСКОГО ПРОЦЕССОРА

Мирзоян Д.И., ассистент, МГТУ МИРЭА, Москва, Россия
E-mail: d.i.mirzoyan@gmail.com

Аннотация. В данной работе проанализированы архитектурные особенности графических процессоров. Предложена методика, основанная на модифицированных блочных алгоритмах, которая дает возможность эффективного решения подобных задач в гетерогенной системе на базе графического процессора. В подтверждение эффективности методики приведены результаты ее использования для алгоритмической обработки символьной информации, а именно, задачи сжатия данных без потерь.
Ключевые слова: графический процессор, блочная обработка, символьные данные, GPGPU, RLE, конвейеризация.

METHODS IMPROVE IMPLEMENTATION PROBLEM OF ALGORITHMIC DATA PROCESSING IN HETEROGENEOUS SYSTEMS GPU BASED

Mirzoyan D.I., assistant, MSTU MIREA, Moscow, Russia
E-mail: d.i.mirzoyan@gmail.com

Abstract. Some of graphics processor’s architectural features are reviewed. Some tasks are demonstrated, for which usage of graphics processor inside hybrid system is initially unpractical due to shown architectural features. There are shown some results in support of efficiency of that methods, specifically results of modified symbolic data processing run-length encoding algorithm, used for lossless data compression.
Keywords: graphics processor, block processing, symbolic data, GPGPU, RLE, pipelining.

Введение

Существуют задачи алгоритмической обработки информации (например: сжатие информации без потерь), реализация хода решения которых, в отличие от задач математической обработки, сильно зависит от конкретных значений исходных данных.

По архитектурным особенностям использование графического процессора в рамках гетерогенной системы при решении подобных задач изначально не является рациональным, но имеет перспективы при комплексной обработке информации. Была построена и применена методика повышения эффективности реализации задач алгоритмической обработки данных в гетерогенной системе на базе графического процессора, основанная на преобразовании исходных традиционных алгоритмов

блочной обработки символьных данных. Предлагаемая методика позволяет путем модификации таких шагов алгоритма, как ветвления и циклы, обойти определенные архитектурные особенности графических процессоров. Экспериментальные исследования эффективности построенного решения показали, что использование разработанной методики при параллельной блочной обработке данных дает для некоторых алгоритмов прирост производительности на полтора порядка.

Особенности архитектуры графических процессоров

Изначально графический процессор был предназначен для решения очень узкого и специфичного класса задач — построения проекции трехмерной модели на экране монитора. Сегодня графический процессор превратился в мощное устройство для решения этого класса задач, с производительностью во много раз превышающую производительность центрального универсального процессора.

К архитектурным особенностям графических процессоров относят следующее:

1. Высокая степень параллелизма. Графический процессор содержит множество так называемых мультипроцессоров, каждый из которых содержит множество одинаковых арифметико-логических устройств, называемых процессорами. Несмотря на, казалось бы, большое количество независимых исполнительных устройств, это не так и они не независимы. В графических процессорах параллелизм имеет место в большей степени по данным — каждый процессор мультипроцессора выполняет в один момент времени одну и ту же инструкцию — но с разными данными. Здесь важно отметить, что некоторые операции, например ветвления, логические, операции двойной точности выполняются на блоках, количество которых в составе мультипроцессора меньше количества собственно процессоров.

2. Строение подсистемы памяти. Здесь «графическое» происхождение процессора видно из наличия специальной памяти — кэша текстур, для которого определены на уровне базовых операции выборки и фильтрации текстур. В большинстве счетных задач эти операции не используются, и кэш текстур выполняет кэширование исходных данных задачи. Кэш констант содержит неизменяемые по ходу решения дополнительные данные в контексте задачи. В отличие от кэша данных, его содержимое одинаково для всех мультипроцессоров, принимающих участие в решении задачи, все время решения этой задачи. Общая память доступна только в рамках одного мультипроцессора. Основное ее назначение — хранение промежуточных результатов вычислений. Память устройства доступна всем мультипроцессорам — это та память, которая по отношению к графическому процессору является внешней (находится не на

том же кристалле). Также следует помнить, что количество быстрой памяти, доступной каждому процессору крайне невелико. Размер файла регистров составляет порядка 512 байт на каждый процессор в мультипроцессоре. Только к регистрам, кэшу констант, общей памяти и кэшу исходных данных доступ осуществляется без каких-либо существенных задержек. В случае, если потребуются данные, которых нет в кэше, задержка их получения может составлять до 400 тактов в современных реализациях. Однако запись данных в глобальную память, без их чтения оттуда — относительно дешевая операция, за счет того что в графическом процессоре есть специальный блок, который буферизует записываемые в память данные. Тем не менее, этот блок имеет ограниченную пропускную способность и очень интенсивная запись в память, даже осуществляющаяся последовательно, может ограничить производительность системы.

3. Механизм работы. Программа, выполняющаяся на графическом процессоре (в принятой терминологии ядро) разбивается по данным на множество потоков. Несколько таких потоков образуют блок потоков, выполняющийся целиком на одном мультипроцессоре (или их фиксированной группе). В свою очередь, блок разбивается на несколько основ (также называется волновой фронт) — групп потоков, одновременно выполняемых на мультипроцессоре(ах). Количество потоков в основе — аппаратно заданная константа, обычно равная учетверенному количеству процессоров в мультипроцессоре. Таким образом, требуется минимум четыре прохода для выполнения одной основы.

4. Характер обрабатываемых данных. Изначально графические процессоры обрабатывали данные в векторной форме — в виде наборов вещественных чисел, обозначающих координаты точек, векторы, матрицы преобразования и т.п., а также наборов целых чисел — цветов отдельных элементов сцены и изображений.

5. Механизм обработки ветвлений. Изначально задача графического процессора не предполагала появления ветвлений в программах обработки данных (различные участки обрабатывались различными программами, а не одной, в которой выбиралась та или иная ветка алгоритма). Когда же поддержка ветвлений была введена, реализовывалась она очень необычным способом — в точке ветвления потоки разделялись на те которые пойдут по одной ветке алгоритма, и те, которые пойдут по другой прозрачно для системы и самой программы (это явление получило название сериализации). Но перегруппировка потоков потребовала бы сложных аппаратных решений, и вследствие этого она невозможна. Поэтому при ветвлении мультипроцессор выполняет каждую ветку алгоритма (исключая, разумеется, тот случай, когда все потоки пошли только по одной ветке), а для процессоров, которые выполняют поток,

пошедший по другой ветке, запрещается запись результата. Соответственно, при большом количестве циклов и/или ветвлений возможна ситуация, в которой один мультипроцессор будет выполнять фактически только один поток за раз — и все преимущества параллельной архитектуры сойдут на нет. Поскольку такая ситуация связана с нарушением одновременной работы множества потоков она получила название разрушение когерентности. Именно это явление препятствует выполнению сложных (по структуре) алгоритмов обработки данных на графических процессорах.

С распространением идеи GPGPU производители графических процессоров задумались об этой области их применения. Это привело к смягчению некоторых из архитектурных особенностей, в частности работы с целыми числами (последние два поколения графических процессоров) и вещественными числами двойной точности (последнее поколение).

Анализ влияния архитектурных особенностей графического процессора на результирующую производительность гетерогенной системы

Большинство вышеперечисленных особенностей практически не сказываются на производительности шейдерных программ и алгоритмов математической обработки данных [7], но в случае алгоритмической обработки, особенно сложной, способны существенно повлиять на результирующую производительность гетерогенной системы. Если рассматривать «канонические» реализации алгоритмов, то становится видно, что большинство из них совершенно непригодны для использования на графическом процессоре, так как демонстрируют удручающую производительность [8]. Оптимизированные для универсальных процессоров реализации также не подходят, не только из-за низкой эффективности [9], но и по причине различия архитектур, что при значительной низкоуровневой оптимизации реализации для универсального процессора, например, с использованием SIMD-инструкций и ассемблерного кода, делает ее совершенно не переносимой на другие вычислительные компоненты.

Таким образом, становится понятной необходимость найти те или иные алгоритмические решения обхода и/или смягчения указанных выше особенностей. Комбинация таких решений позволяет достичь значительного повышения эффективности реализации, в отдельных случаях значительно превосходя производительность центрального универсального процессора гетерогенной системы.

Рассмотрим первую и основную особенность графических процессоров. Именно она в основном и ограничивает спектр применения графических процессоров и гетерогенных систем с их использованием. Графический процессор — массивно-параллельный вычислитель, значительная же часть задач алгоритмической обработки,

вроде того же сжатия данных, строго последовательна. Такие алгоритмы часто изначально неспособны использовать даже более одного универсального процессора. В этом случае применяются классические приемы распараллеливания задачи, чаще все заключающиеся в замене исходного алгоритма эквивалентным, но допускающим распараллеливание. Также ввиду того, что графические процессоры целесообразно применять только при значительном объеме задачи, можно применять блочный подход — данные разделяются на множество небольших блоков, которые обрабатываются параллельно, и результаты обработки которых затем объединяются для получения конечного результата [10].Однако, такие приемы можно использовать, только если исходная задача такое допускает, и те задачи, которые не могут быть эффективно распараллелены на несколько универсальных процессоров, совершенно нерационально решать с помощью графического процессора.

Вторая особенность максимально негативно влияет на алгоритмы, интенсивно использующие память, и особенно, с множественным и случайным доступом к памяти. На универсальных процессорах задержки доступа к памяти в таком случае скрадываются большим объемом встроенной кэш-памяти и продвинутыми алгоритмами кэширования данных. Основным способом смягчения этой проблемы является специальная предварительная подготовка данных в памяти, обычно осуществляемая центральным универсальным процессором. Этот способ особенно эффективен в случае использования сложных структур, а не массивов исходных данных [9].Другим способом смягчения этой особенности является, как ни странно, повышение степени параллелизма. За счет того, что графический процессор способен переключаться на следующий волновой фронт, пока предыдущий(е) ожидают данные из памяти, большая величина задержки доступа скрадывается, и пропускная способность памяти используется с большой эффективностью.

Из результатов, приведенных в [8] видно, что при уменьшении блока данных и повышении степени параллелизма, производительность значительно увеличивалась. Но для многих задач применять небольшие (десятки-сотни байт) блоки бывает нецелесообразно. В таком случае мы можем разбить большие блоки на меньшие, применив конвейеризацию и увеличив тем самым степень параллельности. Под конвейеризацией в данном случае понимается выполнение алгоритма для небольшого кусочка данных за раз, с сохранением его внутреннего состояния, и загрузки этого состояния для следующего кусочка того же блока. Однако данный прием эффективен лишь в случае очень большого размера задачи (много превосходящего размер глобальной памяти графического процессора), и кроме того, тратит некоторую часть

глобальной памяти для хранения состояния алгоритма.

Третья особенность проявляется в том, что объем быстрой памяти (регистров), доступной для потока, сильно ограничен. Если размер регистрового файла для оного процессора в составе мультипроцессора равен 512 байт, у нас есть блок из 4х основ, в каждой из которых 4 потока, и весь регистровый файл делится поровну на все эти потоки, то имеем 512/(4*4) = 32 байта регистров на поток, или 8 32-битных регистров. При большом количестве используемых временных переменных часть их находится в глобальной памяти, а не в регистрах. Лишние обращения в память не слишком благотворно сказываются на производительности, но высокая степень параллельности в значительной степени скрадывает возникающие задержки. В таком случае рекомендуется переработать алгоритм, чтобы он использовал по возможности меньше памяти для хранения промежуточных результатов или использовал другие типы памяти, как продемонстрировано в [8].

Четвертая особенность касается в основном математических методов обработки, и слабо влияет на алгоритмические. Некоторые математические алгоритмы и численные методы были специально переработаны для получения достаточно точных результатов с использованием только чисел одинарной точности. Для многих сфер применения такая точность и не требуется — более чем достаточно одинарной. Кроме того, в последних поколениях графических процессоров блоки двойной точности были улучшены, и использование чисел двойной точности в программах графического процессора приводит к падению производительности в 1.5-3 раза, а не на 1.5-2 порядка, как это было ранее.

Пятая особенность является основным фактором, ограничивающим применение алгоритмической обработки данных на графическом процессоре. Методы алгоритмической обработки, в отличие от математической, обычно содержат множество циклов и ветвлений, часто вложенных. При каждом ветвлении, если не все потоки идут по одному пути выполнения, происходит расщепление волнового фронта, что приводит к потере производительности. При наличии такой возможности, после выполнения участка с ветвлением, волновые фронты снова соединяются.

Но при вложенных циклах и большом количестве ветвлений этот механизм не может отработать полностью. Происходит потеря когерентности потоков — и падение производительности в кратное количество раз. Единственным способом избежать такого катастрофического снижения производительности является максимально возможное сохранение когерентности. Для этого чаще всего необходима переработка реализации алгоритма в той или иной степени. Возможно использование модификации

циклов, условий, а также в некоторых случаях замена условий на математические выражения [4].

В таблице 1 приведены результаты анализа влияния архитектурных особенностей графического процессора на производительность гетерогенной системы для математических и алгоритмических задач обработки данных. В последнем столбце указаны возможные способы смягчения негативного влияния.

Таблица 1.

Влияние архитектурных особенностей графического процессора на производительность гетерогенной системы.

Особенность	Математическая обработка данных	Алгоритмическая обработка данных	Способы смягчения
Высокая параллельность по данным	Не влияет (в соответствующих классах задач)	Большинство алгоритмов последовательны и требуют переработки	Применение распараллеливания, блочная обработка данных
Структура и типы памяти	Потеря производительности при использовании сложных структур вместо массивов данных	Потеря производительности при использовании сложных структур вместо массивов данных и при случайном доступе к данным	Применение специальных структур данных, стратегий размещения данных в памяти, увеличение степени параллельности
Малый объем локальной памяти потока	Не влияет (в соответствующих классах задач)	Сильно ограничивает производительность алгоритмов с большим количеством промежуточных результатов	Переработка алгоритма, использование других видов памяти, увеличение степени параллельности
Неэффективность обработки чисел двойной точности	Потеря производительности до 2х порядков	Не влияет	Использование специальных алгоритмов и/или математических методов, использование последних поколений аппаратных средств
Обработка ветвлений	Почти не влияет	Катастрофическая потеря производительности во многих случаях из-за потери когерентности потоков в случае сложной циклической и условной обработки	Увеличение степени параллельности, модификация алгоритма

Методика повышения эффективности реализации задач алгоритмической обработки данных в гетерогенной системе на базе графического процессора

Следует отметить, что предлагаемая методика применима в основном, к блочным версиям алгоритмов. Она не решает подробно изученную задачу

распараллеливания алгоритма, она считает его уже высоко-параллельным по данным. Хотя ее можно применять и для задач математической обработки данных, наибольший результат она покажет на задачах алгоритмической обработки, например, задачах блочного сжатия данных. Также она не дает гарантий достижения необходимой эффективности, она только позволяет ее увеличить в определенных пределах — каких именно, зависит от конкретного алгоритма.

Рассмотрим порядок применения методики на примере алгоритма из [8]:

```
1  __kernel void MTF_Encode0 (__global uchar *InputData,
2                             __global uchar *OutputData)
3  {
4    int WorkStartIndex = DATA_LEN*get_global_id(0);
5    uchar newindex;
6    uchar newchar;
7    uchar CharTable[256];
8
9    for (int i=0; i<256; i++)
10     CharTable[i] = i;
11
12   for (int i=0; i<DATA_LEN; i++)
13   {
14   newindex = 0;
15
16   for (int j=0; j<256; j++)
17   {
18     if (InputData[WorkStartIndex+i] == CharTable[j])
19     {
20       newindex = j;
21       break;
22     }
23   }
24
25   newchar = CharTable[newindex];
26
27   for (int j=newindex; j>0; j--)
28   {
29     CharTable[j] = CharTable[j-1];
30   }
31
32   CharTable[0] = newchar;
33
34   OutputData[WorkStartIndex+i] = newindex;
35 }
36 }
```

Для начала рассмотрим характер доступа к памяти в данной задаче. Если рассмотреть сам алгоритм, или его реализацию, приведенную выше, станет ясно, что доступ к данным осуществляется линейно, последовательно, по одному байту за итерацию. Запись выходных данных также осуществляется последовательно, по одному байту за итерацию. Если рассмотреть такой характер доступа к данным с точки зрения архитектурных особенностей подсистемы памяти графического процессора, станет ясно, что этот характер неплохо соответствует перечисленным особенностям и

не должен существенно влиять на производительность. Недостатком может быть разве что чтение памяти побайтно, а не более крупными единицами, но в данном случае модификация алгоритма для чтения более крупными блоками не даст прироста производительности по причине простоты самого алгоритма, сравнительно существенных расходов на преобразование данных, и, самое главное, из-за уже значительного использования памяти самим алгоритмом.

Далее рассмотрим характер использования памяти для хранения промежуточных результатов и внутреннего состояния алгоритма. Сразу же становится видно, что внутреннее состояние, содержащее историю последних встретившихся символов, занимает существенную по размеру область памяти, а именно, 256 байт. Причем перезапись этой памяти происходит постоянно и в значительной степени хаотично. Если рассмотреть это свойство алгоритма с точки зрения архитектурных особенностей графического процессора, видно, что такой объем промежуточной памяти не может быть эффективно отображен на быструю память графического процессора, и будет выделен в пределах физической глобальной памяти, доступ к которой, особенно случайный, приводит к существенным задержкам выполнения.

Устранение или смягчение этого ограничения возможно только при очень высокой степени параллельности (причем, вероятно, полная компенсация задержек от размещения этого блока в глобальной памяти возможна только при значительном превышении размера данных над размером физически присутствующей памяти, т.е. нереализуема на практике), или же при смене типа памяти на более быструю.

Поскольку мы не можем разместить данный блок в быстрой регистровой памяти (ее попросту нет в таком объеме, чтобы хватило на все выполняющиеся потоки), необходимо найти другой тип памяти. Им может стать локальная память мультипроцессора — при соответствующей реализации. Следует помнить, что локальная память мультипроцессора является разделяемой между всеми потоками, выполняющимися на последнем, а не индивидуальной памятью каждого потока, какой является регистровая. Следовательно, необходимо организовать деление адресного пространства этого типа памяти между потоками.

Но для начала рассчитаем, хватит ли объема этой памяти. Для тестовой системы, исходя из технической документации, размер локальной разделяемой памяти равен 32 КиБ, чего будет достаточно для 32 КиБ / 256 байт = 128 потоков на мультипроцессор. Это больше, чем число процессоров в мультипроцессоре, следовательно, памяти хватит. Для разделения адресного пространства воспользуемся индексом потока в локальной группе (0-й поток получит первые 256 байт памяти, 1-й — вторые, и так

далее). Другие временные переменные оставим в регистровой памяти, тем более, что теперь ее будет вполне достаточно.

Теперь определим точки потенциального расхождения волновых фронтов и, соответственно, потери когерентности. Обратим внимание на циклы, присутствующие в приведенном фрагменте кода. Первой потенциальной точкой потери когерентности будет основной цикл этого ядра, начинающийся со строки 12. Однако, если проанализировать условие цикла, становится ясно, что этот цикл будет выполняться одинаковое количество раз для каждого потока (DATA_LEN — константа), и не является точкой потери когерентности. Даже будь DATA_LEN переменной с произвольным значением, в данном случае из-за отсутствия операции после цикла, потери когерентности происходить не будет — соответствующие потоки волнового фронта завершат свою работу и не будут расщепляться.

Следующий цикл, начинающийся на строке 16, тоже выглядит как выполняющийся одинаковое число раз в разных потоках. Однако он содержит внутри условие и оператор выхода из цикла. Поскольку этот цикл является вложенным, и обработка после его выполнения может еще продолжаться значительное время, в течение которого этот цикл будет вызван еще множество раз, такая реализация приведет к постоянному расщеплению фронтов на реальных данных (а не, скажем, заполненных нулями или другими одинаковыми значениями массивов — идеальный вариант с точки зрения сохранения когерентности, но совершенно оторванный от реальности), потере когерентности и катастрофическому падению производительности. Отметим этот цикл как точку применения изменений.

Третий цикл начинается на строке 27, и имеет плавающее количество итераций не только для разных потоков, но и для разных прогонах основного цикла, в который он вложен (в случае нетривиальных данных). Следовательно, он также будет приводить к потере когерентности, и его необходимо модифицировать.

Теперь можно приступать к модификациям данной реализации алгоритма. Необходимо предотвратить или уменьшить насколько возможно потерю когерентности. Для этого можно применить два подхода: обеспечить одинаковые пути следования алгоритма или установить принудительную синхронизацию и слияние волновых фронтов в ключевых точках. Таковыми точками будут для первого цикла строка 24, для второго строка 31. Следует помнить, что все потоки должны пройти через синхронизирующие барьеры одинаковое число раз. Чтобы обеспечить одинаковые пути следования, следует изменить циклы так, чтобы они выполнялись всегда одно и то же количество раз. Но по логике работы алгоритма это может быть не

нужно, более того, скорее всего это приведет к ошибкам в расчетах. Следовательно, нужно отсечь паразитные итерации цикла от повреждения результата. Для этого можно применять экранирующие условия (только очень аккуратно – чтобы такое условие не стало новой точкой потери когерентности), либо модифицировать такое условие в математическое выражение. В частности, именно этот подход дает наибольший эффект для 2-го цикла.

В случае обнаружения точки потери когерентности в условии, следует либо модифицировать его (часто бывает достаточно отсечь ветку else, в других случаях это может не давать никакого эффекта), либо заменить действия внутри условного оператора на выражения (если это возможно, что далеко не всегда так). Также может помочь установка синхронизирующего барьера после условного оператора. Наиболее эффективным приемом из перечисленных представляется замена условий на выражения, т.к. в данном случае мы не нагружаем блок обработки ветвлений, все операции выполняются на АЛУ. Однако, при записи данных в память, этот подход может срабатывать не очень хорошо, особенно если запись может осуществляться не каждую итерацию цикла (что типично для алгоритмов сжатия).

В таком случае эффективность можно повысить, применив экранирование «мусорных» операций записи (операций записи, не влияющих по факту на результат) с помощью условных блоков. Хотя при таком подходе мы используем ветвление и условные операции, затраты на установку масок запрета модификации данных потоков (а именно так осуществляется разделение ветвей выполнения в графическом процессоре — обработка происходит в любом случае, просто ее данные никуда не записываются) и обработку ветвления могут оказаться ниже, чем большое количество излишних операций записи в память (особенно по одному адресу).

К сожалению, установить, какой из методов оптимизации будет наиболее эффективным при практическом применении, без эксперимента на реальных (или приближенных к ним) данных, или, по крайней мере, моделирования, невозможно. Более того, иногда эффект от оптимизации некоторых участков совершенно незаметен — так как проявлявшаяся в этом участке особенность могла скрадываться за счет массивного параллелизма.

Наконец, последним шагом методики является уменьшение блока данных (если возможно) и повышение тем самым степени параллелизма за счет использования конвейеризации.

Таким образом, применение предложенной методики повышения эффективности реализации задач алгоритмической обработки данных в гетерогенной

системе на базе графического процессора позволяет достичь прироста производительности реализаций алгоритмов обработки данных в диапазоне от 10% до 2000% в зависимости от алгоритма. Также эта методика позволяет увеличить производительность алгоритмов, применяемых для обработки данных на графических процессорах, повысить эффективность отдельных реализаций некоторых алгоритмов, применение которых было признано неоправданным, до рационального для практических задач уровня.

Заключение

На основании экспериментальной проверки методики можно сделаны выводы:

1. Современные графические процессоры способны эффективно решать задачи не только интенсивной математической вещественной обработки данных, но и осуществлять целочисленные расчеты и алгоритмические преобразования информации с эффективностью, превышающую эффективность центрального процессора.

2. Предложенная методика повышения эффективности реализации задач алгоритмической обработки данных в гетерогенной системе на базе графического процессора позволяет достичь прироста производительности реализаций некоторых алгоритмов блочной обработки данных до достаточного для рационального применения в практических задачах уровня.

3.Наиболее обоснованным в будущем представляется совместное использование центрального и графического процессора гетерогенной вычислительной системы в рамках одной задачи для эффективного решения подзадач на наиболее подходящем вычислительном компоненте с применением методов модификации и оптимизации алгоритмов для соответствующих компонентов, а также использование конвейерных алгоритмов обработки для максимизации эффективности гетерогенной системы.

Список литературы

1. Сайт Национального суперкомпьютерного центра в Тяньцзинь (КНР) Режим доступа: http://www.nscc-tj.gov.cn/en/ (дата обращения 05.04.2013)

2. В.С. Бурцев Развитие специализированных вычислительных систем ПВО и ПРО Режим доступа: http://www.ipmce.ru/about/press/articles/politeh2004/ (дата обращения 05.04.2013)

3. В.Д. Пекелис, Кибернетика — неограниченные возможности и возможные ограничения. Современное состояние. — М.: Наука, 1980.— 208 с. http://scilib-technics.narod.ru/Pekelis32/index.html

4. Роджер Чемберлен, Джозеф Ланкстер, Рон Цитрон, Перспективы разработки

приложений для гетерогенных вычислительных систем. // Университет Вашингтона, 2007 Режим доступа: http://rssi.ncsa.illinois.edu/2007/proceedings/posters/rssi 07_12_poster.pdf (дата обращения 05.04.2013)

5. Основные аспекты применения GPGPU систем // Д.И. Мирзоян // Современные информационные технологии и ИТ-образование Сборник избранных трудов VI Международной научно-практической конференции: учебно-методическое пособие. Под ред. проф. В.А. Сухомлина. - М.: ИНТУИТ.РУ, 2011. - 1052 с. ISBN 978-5-9556-0129-8, с. 988-994

6. Маниш Арора, Архитектура и эволюция систем ЦПУ-ГПУ для универсальных вычислений, Факультет Компьютерных наук и разработок, Университет Калифорнии, Сан-Диего Режим доступа: http://cseweb.ucsd.edu/~marora/files/ papers/REReport_ManishArora.pdf (дата обращения 05.04.2013)

7. Д.И. Мирзоян Сравнение производительности программного обеспечения гетерогенной вычислительной системы с гомогенной и гетерогенной программными средами выполнения. Сборник научных трудов 11-ой НПК «Современные информационные технологии в управлении и образовании», 24 апреля 2012, в трех частях. М.: ФГУП НИИ «Восход»,Часть 2, с.24-32

8. Д.И. Мирзоян Опытная модификация алгоритма обработки данных «Движение к началу» (Move-To-Front, MTF) для гетерогенных вычислительных систем. Научно-технический сборник научных материалов соискателей, докторантов и адъюнктов Академии Государственной противопожарной службы МЧС России,№2, 2012г.,8 с.

9. Дженс Брейтбарт, Тематические исследования по использованию ГПУ и форматов структур данных. Университет Касселя 2008 Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.189.2330&rep=rep1&type=pdf (дата обращения 05.04.2013)

10. Ритеш А. Патель // Яо Жанг // Джейсон Мак // Эндрю Дэвидсон // Джон Д. Овенс Параллельное сжатие данных без потерь с использованием ГПУ Режим доступа: http://www.idav.ucdavis.edu/func/return_pdf?pub_id=1087 (дата обращения 05.04.2013)

11. Алекс Эйрола, Сжатие данных без потерь на системах GPGPU Режим доступа: http://arxiv.org/pdf/1109.2348v1 (дата обращения 05.04.2013)

12. Венбин Фанг, Бингшенг Хе, Кионг Луо Сжатие баз данных с использованием графических процессоров Режим доступа: http://www3.ntu.edu.sg/home/bshe/Fang300.pdf (дата обращения 05.04.2013)