УДК 004.424.5.032.24

Г.В. Воронцов, А.П. Преображенский, О.Н. Чопоров

ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ УСКОРЕНИЯ АЛГОРИТМОВ ПАРАЛЛЕЛЬНОЙ СОРТИРОВКИ RADIX НА GPGPU

Воронежский институт высоких технологий
Воронежский государственный технический университет,
Воронеж, Россия

В данной работе проводится анализ алгоритма параллельной сортировки RADIX на графических процессорах (GPGPU). Вначале рассматривается наивный алгоритм поразрядной сортировки. При этом используются два вида поразрядной сортировки - по младшим и старшим разрядам. Приведен пример их использования. С тем, чтобы увеличить производительность алгоритма поразрядной сортировки предлагается использовать параллельное решение, хотя при этом возникают дополнительные проблемы, требующие своего решения. Анализируются возможные подходы по распараллеливанию, предложенные различными авторами. В рассматриваемом алгоритме данные хранятся в памяти графического процессора, и сортировка выполняется непосредственно на GPU. Этот алгоритм параллельной Radix сортировки состоит из 3-х подсистем: подсчет двоичных комбинаций в текущем разряде, префикс суммирование, окончательное соответствие ключей с вычисленными позициями. Первым шагом алгоритма является процесс подсчета частоты каждого элемента в последовательности. Для осуществления этого параллельным образом происходит разделение входного массива на блоки. Далее вычисляется локальная частота всех возможных элементов для каждого блока. Затем для каждой маски проводится префиксное суммирование. На следующем шаге получаются из локальных списков частот глобальные. Приведены результаты моделирования, продемонстрировавшие увеличение в несколько раз быстродействие предлагаемого алгоритма по сравнению с известными.

Ключевые слова: параллельная сортировка, алгоритм, данные, процессор.

Введение. Эффективная сортировка является ключевым требованием для многих алгоритмов, так как часто возникает необходимость переупорядочивать входные данные, чтобы можно было дополнительно их исследовать. Ускорение существующих методов, а также разработка новых подходов к сортировке имеет решающее значение для многих графических алгоритмов реального времени, систем баз данных, численного моделирования и т.д. Это одна из самых фундаментальных операций по организации постоянно растущих массивов данных. Хотя и существуют оптимальные алгоритмы сортировки для CPU, эффективная параллельная сортировка на GPU все еще является проблемой.

В данной работе предлагается алгоритм, позволяющий проводить сортировку в несколько раз быстрее по сравнению с существующими подходами.

Особенности существующих алгоритмов сортировки. Для начала рассмотрим наивный алгоритм поразрядной сортировки. Суть поразрядной сортировки заключается в том, что мы не сравниваем элементы массива друг с другом, а сравниваем их разряды: от младшего к старшему, или от старшего к младшему (зависит от задачи). Причем, при каждой итерации элементы группируются по самому младшему разряду (сначала все, заканчивающиеся на 0, затем заканчивающиеся на 1 и так до 9, или, если сравниваются биты, то вначале 0, потом 1). Сложность наивного алгоритма линейна, т.е. равна $O(n)$.

Различают два вида поразрядной сортировки, первый – сортировка по младшим разрядам (LSD radix sort – least significant digit radix sort), пример представлен на рис. 1, второй же – по старшим разрядам (MSD radix sort – most significant digit radix sort), пример представлен на рис. 2.

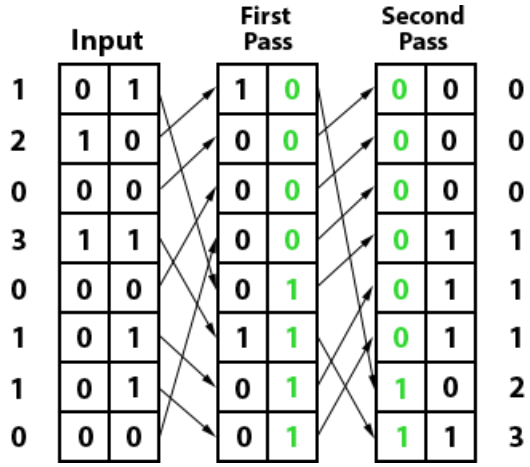

Рисунок 1 – LSD тип сортировки

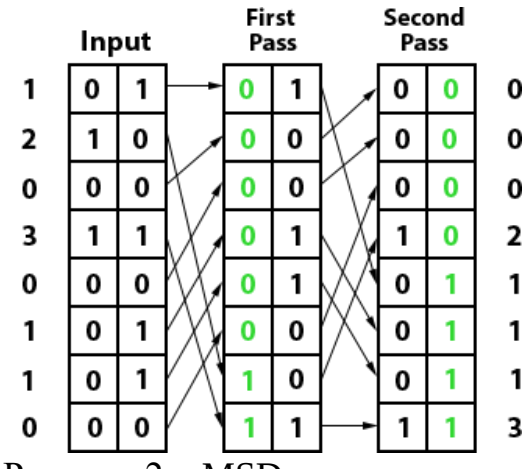

Рисунок 2 – MSD тип сортировки

При LSD сортировке (по младшим разрядам) получается порядок по возрастанию: 0, 1, 2, 3, 4, 20, 31 и т.д. То есть, значения вначале

сортируются по единицам, затем по десяткам, сохраняя порядок по единицам внутри десятков, затем по сотням и т.д.

При MSD сортировке (по старшим разрядам) значения вначале сортируются по наибольшему разряду, потом уже по менее большому, и так до наименьшего. В итоге, если для чисел, то получается примерная последовательность: 1, 10, 100, 2, 200 и т.д. MSD сортировка больше подходит для строк: можно отсортировать строки в алфавитном порядке.

Также, стоит упомянуть, что если при поразрядной сортировке вначале выбирать не 0-й бит, а 1-й, то можно получить последовательность, отсортированную в порядке уменьшения.

Для увеличения производительности алгоритма поразрядной сортировки можно прибегнуть к использованию параллельного решения. Тем более, после выхода технологий неспециализированных вычислений на графическом процессоре, появилась возможность использовать массивный параллелизм современных архитектур GPU для сортировки больших объемов данных с очень высокой скоростью. Но, с переходом на графический процессор появляется ряд проблем:

- Распараллеливание последовательного алгоритма сортировки – у каждой итерации, и каждой операции сравнения разрядов есть зависимости от предыдущих вычислений.
- Доступ к памяти – глобальная память у GPU очень медленная, а разделяемая (shared) память имеет ряд ограничений.
- Синхронизация потоков и общая глобальная синхронизация – часть операций необходимо синхронизировать между потоками, чтобы не было конфликтов, а итерации – синхронизировать глобально.

Стоит заметить, что есть еще одна проблема, которую стоит решить – обобщение алгоритма сортировка, чтобы его можно было применять к произвольным типам данных, например, к отрицательным числам, или числам с плавающей запятой, или строкам и т.д. Но, обычно, эта проблема решается реализацией различных алгоритмов для каждого типа данных.

Для решения всех этих проблем в данной статье будет рассматриваться параллельный алгоритм поразрядной сортировке, который был разработан Linh Ha, Jens Kruger, Claudio T. Silva [1].

Параллельные сортировочные сети уже давно признаны в качестве предпочтительного способа достижения высокой производительности в суперкомпьютерных системах, например, таких как машины Cray. Еще в 1968 году, Batcher [3] предложил сортировочные сети на основе сравнения: merge sort и bitonic sort. Эти алгоритмы, имея субоптимальную сложность равную $O(n \log n)$, до сих пор используются в параллельных архитектурах. Оптимальный алгоритм на основе сравнения был описан Ajtai M. [4]. Также, алгоритм параллельной сортировки был описан Leighton [5] для

гиперкуба n-процессора с использованием случайных операций. Но добиться сложности $O(n \log n)$ в алгоритмах параллельной сортировки на основе сравнения достаточно сложно.

Параллельные алгоритмы сортировки на основе подсчета являются альтернативой подходам, основанными на сравнении. Они могут снизить общую сложность до $O(n)$ или даже до $O(\frac{n}{p})$, где p – количество процессоров. Наиболее известным алгоритмом сортировки подсчетом является параллельный алгоритм Radix сортировки. В 1991 году Zagha и Blelloch реализовали параллельный алгоритм поразрядной сортировки на восьмипроцессорной машине Cray Y-MP [6]. Результаты показали линейную зависимость от количества процессоров и были значительно лучше, чем результаты параллельного алгоритма быстрой сортировки на той же машине.

Рассматриваемый в данной статье алгоритм похож на подход Zagha и Blelloch, но, архитектура машины Cray отличается от архитектуры современных графических процессоров, так что, реализация отличается от их подхода. Также, рассматриваемый алгоритм отличается от алгоритма, который разработан Linh Ha, Jens Kruger, Claudio T. Silva [1], так как Direct Compute отличается от технологии CUDA, хотя общие понятия одни и те же.

Особенности предлагаемого алгоритма сортировки. Реализация алгоритма, который рассматривается в данной статье, нацелена на высокопроизводительную сортировку на платформе графического процессора (GPU – General Processing Units). Сложность алгоритма равна $O(n \log n)$. Общие вычисления на GPU (GPGPU) достигли отличных успехов, выйдя в новую вычислительную платформу с высокой энергоэффективностью и высокой масштабируемостью, которая используется во множестве приложений вне сферы компьютерной графики и визуализации, например, в научных вычислениях, геометрической обработке, базы данных, обработке изображений и т.д.

Сама по себе сортировка на графическом процессоре не является такой уж новой, но она стала важным компонентом для приложений GPU. В рассматриваемом алгоритме данные хранятся в памяти графического процессора, и сортировка выполняется непосредственно на GPU, а не как в ранних реализациях Kipfer [7] – когда вычисления производятся на GPU, но данные постоянно перегоняются с CPU на GPU. Также, Sintorn и Assarsson [8] предложили гибридный алгоритм сортировки, на основе векторизованной merge сортировки в сочетании с bucket сортировкой с использованием атомных GPU-операций. Согласно их результатам, гибридный алгоритм в два раза быстрее, чем самый быстрый алгоритм bitonic сортировки. Однако, их подход требует атомных функций, которые,

в свою очередь, требуют последовательных обновлений данных, тем самым теряя большую часть возможностей параллельной обработки графического процессора. Столь же быстрая производительность была заявлена Sengupta [9], который использовал оптимизированную параллельную префиксную суммарную технику Harris [10] для реализации binary-radix сортировки на графическом процессоре с использованием технологии CUDA.

Рассматриваемый же алгоритм radix сортировки в несколько раз быстрее по сравнению с указанными выше алгоритмами. Поэтому, его можно применять не только к проблемам, связанных с графикой, но и к любым другим задачам, которые требуют максимальной производительности в сортировке больших массивов данных.

Сам алгоритм параллельной Radix сортировки, который мы рассмотрим, состоит из 3-х подсистем:

- Подсчет двоичных комбинаций в текущем разряде
- Префикс суммирование
- Окончательное соответствие ключей с вычисленными позициями

Еще, можно включить подсистему проверки порядка, чтобы закончить алгоритм раньше, если последовательность отсортирована. Но, как показала практика, в реальных задачах очень редко наступает момент, когда до окончания основного алгоритма последовательность является уже отсортированной.

Для большего понимания алгоритма, будем использовать двухбитовые целые числа без знака, хотя алгоритм позволяет сортировать любые n-разрядные числа, в том числе и числа с плавающей запятой. Обобщенный алгоритм представлен в приложении (см. алгоритм 1).

Первым шагом алгоритма является процесс подсчета частоты каждого элемента в последовательности (рис. 3в). Чтобы сделать это параллельно, мы разделяем входной массив на блоки (рис. 3а). Дело в том, что, деля данные на небольшие куски и выполняя операции для этих кусков параллельно и независимо, мы можем эффективно использовать все доступные мощности GPGPU. Однако, произвольно выбрать разделение нельзя, так как, чтобы получить максимальную производительность, необходимо учитывать латентность памяти и время на синхронизацию потоков. Для уменьшения латентности можно увеличить кол-во потоков (например, выбрать 128 или 256 потоков на блок). Для уменьшения времени синхронизации и увеличении производительности путем поддержки принципа SIMD необходимо выбирать такое количество потоков в блоке, чтобы оно было кратно 32 (в CUDA, Direct Compute или

OpenCL 32 потока равны одному Warp). Тогда можно гарантировать совместное считывание, записи, вычисления для всех потоков.

Также, необходимо учитывать пропускную способность глобальной памяти – она намного ниже, чем у разделяемой памяти – кэша, который доступен всем потокам в блоке. Так что, стоит все промежуточные результаты сохранять в shared memory, но, при ее использовании, обязательно необходимо учитывать размер кэша, а он, в текущем поколении GPU, равен 48 кб. Также, разделяемая память не сохраняет данные между вызовами шейдеров, поэтому какие-то глобальные значения там не получится хранить.

Стоит пояснить, что массив Block Sum виртуально разделен на группы, где каждая группа хранит в себе счетчики для определенной цифры. А размер группы равен количеству блоков во входящем массиве. Например, первая группа хранит счетчики для каждого блока для цифры 0, вторая группы – для цифры 1, третья – 2 и т.д. Для иллюстрации мы выбрали размер блока из четырех элементов (рис. 3).

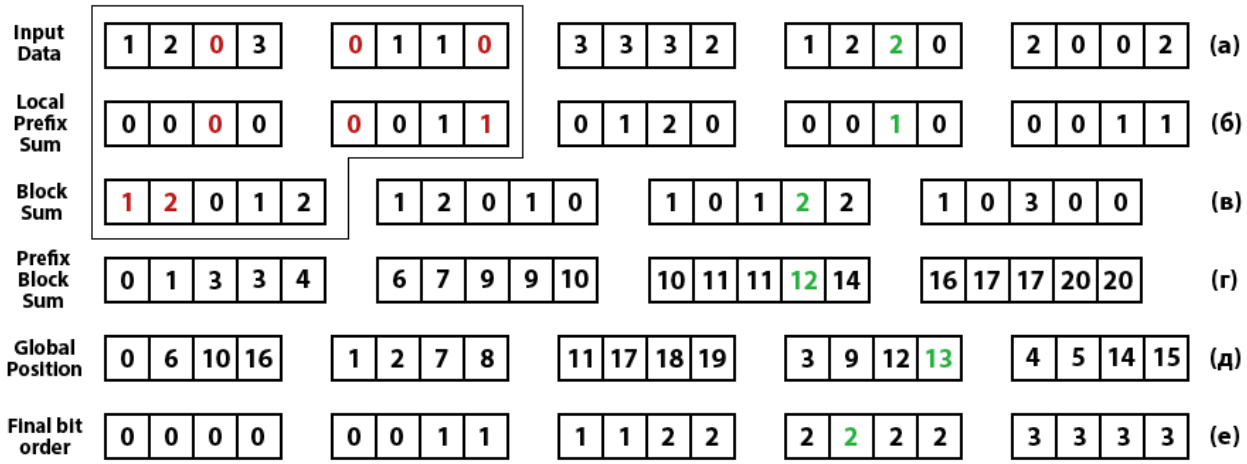

Рисунок 3 – основные шаги параллельного алгоритма Radix сортировки

Следующим шагом необходимо вычислить локальную частоту всех возможных элементов для каждого блока: генерируем четыре счетчика (так как элементы в последовательности 2-х битные, то максимальное количество возможных чисел равно $2^n = 2^2 = 4$, где n – кол-во бит), потом генерируем локальную маску для каждой последовательности бит в каждом блоке, где для каждой ячейки последовательности 0 – нет вхождения, а 1 – есть (рис. 4б для числа 0).

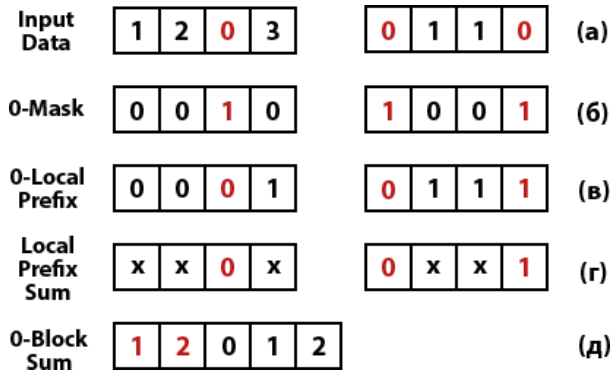

Рисунок 4 – основные шаги для блоков

Далее, для каждой маски проводим префиксное суммирование (рис. 4в для числа 0) и результат сохраняем в отдельный массив. Также, стоит заметить, что наибольшего быстродействия можно добиться в том случае, если маски сразу сохранить в двумерный массив, который расположен в разделяемой памяти, а потом провести операцию префиксной суммы сразу для всех масок одновременно. При этом, каждый поток берет элемент из каждой маски по своему локальному индексу в блоке (см. алгоритм 2 в приложении).

Параллельный алгоритм префиксной суммы хорошо описан Guy E. Blelloch в статье “Prefix Sums and Their Applications” [11], также, алгоритм представлен в виде псевдокода в приложении (см. алгоритм 3 и алгоритм 4).

При большом количестве элементов и большой битности чисел появляется проблема увеличения использования разделяемой памяти. Чем больше бит за раз мы сортируем, тем больше надо выделять памяти. А при хранении масок эта проблема встает очень остро и очень сильно ограничивает нас. Поэтому, Lin Ha, Jens Kruger и Claudio T. Silva предложили способ объединить все локальные маски в одну (рис. 4г). Они заметили, что для каждая цифра относится только к локальному префиксному результату маски в соответствующей позиции этой цифры. Поэтому, вместо использования множества разделенных массивов для хранения масок, можно использовать один массив, размер которого будет равен размеру блока. Это решение существенно снижает потребление памяти. А локальная префиксная сумма (рис. 4г) показывает порядок данных в отсортированной части одного и того же бита счетчика.

На следующем шаге необходимо провести операцию префиксной суммы над массивом Block Sum (рис. 3г), чтобы получить из локальных списков частот глобальные. После получения префиксной суммы (это можно сделать по алгоритму, который представил G. E. Blelloch [11]) Block Sum, остается только вычислить глобальные позиции для каждого

элемента из входящей последовательности. Они вычисляются довольно просто по формуле:

$$P = M_d[n] + m,$$

где:

- M_d – prefix Block Sum
- d – последовательность бит
- n – индекс блока
- m – значение из локальной префиксной суммы

Например, вычислим глобальную позицию для цифры 2 в 4-м блоке (помечено зеленым на рис. 3). Тогда значения будут равны: $m = 1$, $n = 3$, $d = 2$. Подставив в формулу выше получим:

$$P = M_2[3] + 1 = 12 + 1 = 13.$$

Полученные результаты. Параллельный алгоритм поразрядной сортировки, который исследуется в этой статье, был реализован с помощью технологии от компании Microsoft Direct Compute. Данная технология предоставляет новый тип шейдеров – вычислительные шейдеры (Compute shaders), что позволяет написать код на высокоуровневом шейдерном языке HLSL версии 5.0. Сам же язык предоставляет API для работы с видеокартой. Результаты приведены на рис. 5

Алгоритм был разделен на несколько шейдеров-шагов:

- Вычисление локальной префиксной маски для каждого блока и подсчет вхождений элементов в каждый блок
- Вычисление префиксной суммы массива Block Sum
- Вычисление окончательной позиции и перемещение элементов на вычисленные позиции.

Все шейдеры выполняются последовательно каждую итерацию цикла. Всего итераций – 8, так как сортировка проходит сразу по 4-м битам 32-х битных элементов входящего массива.

Подсчет вхождений элементов в каждый блок реализован с помощью функции атомарного добавления – InterlockedAdd, что предполагает некоторую задержку, так как данная функция выполняется последовательно для всех потоков, но, данная задержка крайне мала.

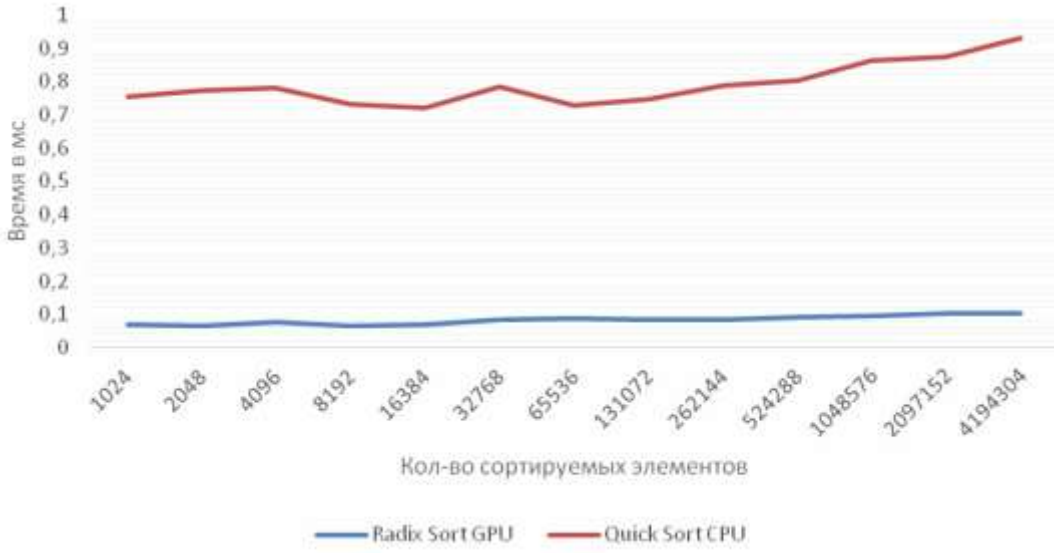

Рисунок 5 – Сравнение GPU Radix Sort и CPU Quick Sort

Стоит отметить, что при реализации функции поиска префиксной суммы последовательности, можно столкнуться с конфликтом банков. Конфликт возникает тогда, когда два или более потоков одного Warp (32 потока) осуществляют доступ к байтам, которые принадлежат разным 32-х битным словам, которые находятся в одном банке памяти разделяемой памяти. Поэтому, если конфликт возник, доступ к разделяемой памяти осуществляет не параллельно, а последовательно, что существенно увеличивает время выполнения алгоритма. Чтобы не было конфликтов, пришлось добавлять смещения (пустые ячейки) при записи в shared memory, и считывать также с учетом смещения. Это уменьшает объем доступной разделяемой памяти.

Все тесты проводились на Nvidia GeForce 1060 6Gb GTX и Intel i7 7700. Для сравнения был выбран алгоритм быстрой сортировки, который чаще всего используется на CPU.

Вывод. В работе предложен алгоритм поразрядной сортировки с увеличенной производительностью за счет распараллеливания вычислений. На основе проведенного машинного эксперимента продемонстрировано, что предлагаемый алгоритм параллельной сортировки обладает более высокой производительностью по сравнению с существующими алгоритмами.

Приложение

Алгоритм 1 Обобщенная поразрядная сортировка

for bit = 0 to 32 **do**
for all blocks in parallel **do**
    Generate local masks on the each bits
    synchthread
    Perform scan prefix sum on the local masks
    Output total number of each scan path to block sum array
end for
Perform scan prefix sum on the block sum array
Compute the new global position
Map element to the right position
bit ← *bit* + *2*
end for

Алгоритм 2 Вычисление локальной префиксной суммы

Allocate 4 counting arrays, *cnt*[4]
for all threadId in thread block **do**
Extract 2 bits combination from the input *data*[*threadId*]
for b = 0 to 4 **do**
    cnt[*b*][*threadId*] ← (b == *extract_bits*)
end for
synchthread
Built sum tree, *Алгоритм 3*
Down sweep the tree, *Алгоритм 4*
synchthread
Output cnt[*extract_bits*][*threadId*] to global memory
end for

Алгоритм 3 Up-Sweep step

for d = 0 to $\log_2 n - 1$ **do**
  for all i = 0 to $(n-1)/2^{d+1}$ in parallel **do**
    #pragma unroll
    for b = 0 to 4 **do**
      cnt[*b*][$i + 2^{d+1} - 1$] ← *cnt*[*b*][$i + 2^{d} - 1$] +
        cnt[*b*][$i + 2^{d+1} - 1$]
      end for
  end for
end for

Алгоритм 4 Down-Sweep step

```
for b = 0 to 4 do
    cnt[b][n − 1] ← 0
end for
for d = log2 n − 1 downto 0 do
    for all i = 0 to (n − 1)/2^(d+1) in parallel do
            #pragma unroll
            for b = 0 to 4 do
                    t ← cnt[b][ i + 2^d − 1]
                    cnt[b][ i + 2^d − 1] ← cnt[b][ i + 2^(d+1) − 1]
                    cnt[b][ i + 2^(d+1) − 1] ← t + cnt[b][ i + 2^(d+1) − 1]
            end for
    end for
end for
```

ЛИТЕРАТУРА

1. Linh Ha Fast 4-way parallel radix sorting on GPUs / Ha Linh, Jens Kruger, Claudio T.Silva [электронный ресурс]: http://www.sci.utah.edu/~csilva/papers/cgf.pdf.
2. Wikipedia, Radix Sort [электронный ресурс]: https://en.wikipedia.org/wiki/Radix_sort.
3. Batcher K.Sorting Networks and Their Applications / K. Batcher // Proceedings of the AFIPS Spring Joint Computing Conference, vol. 32, 1968, pp.307-314.
4. Ajtai M. An 0(n log n) sorting network / M.Ajtai, J.Komlós, E.Szemerédi //In: STOC ’83: Proceedings of the fifteenth annual ACM symposium on Theory of computing, 1983, pp. 1– 9.
5. Leighton T. Tight bounds on the complexity of parallel sorting / T.Leighton // In: STOC ’84: Proceedings of the sixteenth annual ACM symposium on Theory of computing (New York, NY, USA, 1984), ACM, pp. 71–80.
6. Zagha M. Radix sort for vector multiprocessors / M.Zagha, G. E.Blelloch // In: Supercomputing ’91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing (New York, NY, USA, 1991), pp. 712–721.
7. Kipfer P. UberFlow: a GPU-based particle engine / P.Kipfer, M.Segal, R.Westermann // In: HWWS ’04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA, 2004), ACM Press, pp. 115– 122.
8. Sintorn E. Fast parallel GPU-sorting using a hybrid algorithm / E.Sintorn, U.Assarsson //In: Workshop on General Purpose Processing on Graphics

Processing Units (GPGPU) (2007). [электронный ресурс]: http://www.cse.chalmers.se/~uffe/hybridsort.pdf.
9. Sengupta S. Scan primitives for GPU computing / S.Sengupta, M.Harris, Y.Zhang, J. D.Owens // In: Graphics Hardware 2007 (Aug. 2007), ACM, pp. 97–106.
10. Harris M., Sengupta S., Owens J. D. Parallel prefix sum (scan) with cuda / Harris M., Sengupta S., Owens J. D. // GPU Gems 3. Boston: Addison Wesley, 2007. 851–876.
11. Guy E. Blelloch Prefix Sums and Their Applications / Guy E. Blelloch // [электронный ресурс]: https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf.

G. V. Vorontsov, A. P. Preobrazhenskiy, O. N. Choporov

THE RESEARCH OF POSSIBILITIES OF ACCELERATION OF PARALLEL ALGORITHMS FOR RADIX SORTING ON GPGPU

Voronezh Institute of high technologies
Voronezh state technical University, Voronezh, Russia

This paper analyzes parallel RADIX sorting on GPGPU. First, they consider the naive algorithm of radix sorting. It is indicated that using two types of radix sorting - at Junior and senior level. Given an example of their use. In order to increase the performance of the radix sorting algorithm is proposed to use a parallel solution, although this raises additional issues that require resolution. Analyzes possible approaches for the parallelization proposed by various authors. In the proposed algorithm the data is stored in GPU memory, and sorting is performed directly on the GPU. This algorithm is a parallel Radix sort consists of 3 subsystems: counting binary combinations in the current category, prefix summing, the final key according to the computed positions. The first step of the algorithm is the process of counting the frequency of each element in the sequence. To have it done in a parallel way there is a separation of the input array into blocks. It then computes the local frequency of all possible elements for each block. Next, for each mask is the prefix summation. The next step is to obtain from the local lists of the frequency of the global. Simulation results demonstrated the increase of several times the performance of the proposed algorithm in comparison with the known.

Keywords: parallel sorting, algorithm, data, processor.

REFERENCES

1. Linh Ha Fast 4-way parallel radix sorting on GPUs / Ha Linh, Jens Kruger, Claudio T.Silva URL:: http://www.sci.utah.edu/~csilva/papers/cgf.pdf.
2. Wikipedia, Radix Sort URL:: https://en.wikipedia.org/wiki/Radix_sort.
3. Batcher K.Sorting Networks and Their Applications / K. Batcher // Proceedings of the AFIPS Spring Joint Computing Conference, vol. 32, 1968, pp.307-314.

4. Ajtai M. An 0(n log n) sorting network / M.Ajtai, J.Komlós, E.Szemerédi //In: STOC ’83: Proceedings of the fifteenth annual ACM symposium on Theory of computing, 1983, pp. 1– 9.
5. Leighton T. Tight bounds on the complexity of parallel sorting / T.Leighton // In: STOC ’84: Proceedings of the sixteenth annual ACM symposium on Theory of computing (New York, NY, USA, 1984), ACM, pp. 71–80.
6. Zagha M. Radix sort for vector multiprocessors / M.Zagha, G. E.Blelloch // In: Supercomputing ’91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing (New York, NY, USA, 1991), pp. 712–721.
7. Kipfer P. UberFlow: a GPU-based particle engine / P.Kipfer, M.Segal, R.Westermann // In: HWWS ’04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA, 2004), ACM Press, pp. 115– 122.
8. Sintorn E. Fast parallel GPU-sorting using a hybrid algorithm / E.Sintorn, U.Assarsson //In: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU) (2007). URL:: http://www.cse.chalmers.se/~uffe/hybridsort.pdf.
9. Sengupta S. Scan primitives for GPU computing / S.Sengupta, M.Harris, Y.Zhang, J. D.Owens // In: Graphics Hardware 2007 (Aug. 2007), ACM, pp. 97–106.
10. Harris M., Sengupta S., Owens J. D. Parallel prefix sum (scan) with cuda / Harris M., Sengupta S., Owens J. D. // GPU Gems 3. Boston: Addison Wesley, 2007. 851–876.
11. Guy E. Blelloch Prefix Sums and Their Applications / Guy E. Blelloch // URL:: https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf.