# НЕСКОЛЬКО ПОДХОДОВ К ОПТИМИЗАЦИИ АЛГОРИТМА ВЕЙВЛЕТ-ПРЕОБРАЗОВАНИЯ РЕАЛИЗОВАННОГО НА ГРАФИЧЕСКОМ ПРОЦЕССОРЕ

А.Г. Пани, М.Н. Точёная
*Тольяттинский Государственный Университет, Тольятти, Россия*
e-mail: ag.panin@gmail.com, m.tochenaya@gmail.com

**Аннотация**

В данной работе описаны особенности реализации алгоритмов вейвлет-преобразований для платформы NVIDIA CUDA. Архитектура CUDA обладает сложной структурой памяти, и для того, чтобы максимально использовать вычислительную мощность графического процессора, необходимо уделить пристальное внимание оптимизации работы с памятью. В статье описаны несколько различных способов работы с памятью, приводится сравнение скорости работы. Так же из-за того, что для полноценной загрузки GPU нужны тысячи нитей, важным вопросом является разделение данных на подзадачи. В данной работе приводится зависимость скорости работы алгоритма от различных параметров при разделении данных. Ещё одной особенностью реализации, рассмотренной в статье, является оптимизация арифметических выражений.

## Введение

В настоящее время вейвлет-преобразование используется для решения разнообразных задач, нередко заменяя обычное преобразование Фурье. Это наблюдается во многих областях, включая компьютерную графику и обработку изображений, анализ ДНК, общую обработку сигналов, сжатие данных и распознавание речи [2, 3, 5, 6].

При последовательной реализации алгоритмы вейвлет-анализа не обладают должным быстродействием, что затрудняет их применение для обработки данных в реальном времени [6]. Поэтому актуальность данной работы обуславливается необходимостью повышения скорости работы существующих алгоритмов, одним из способов которого является распараллеливание.

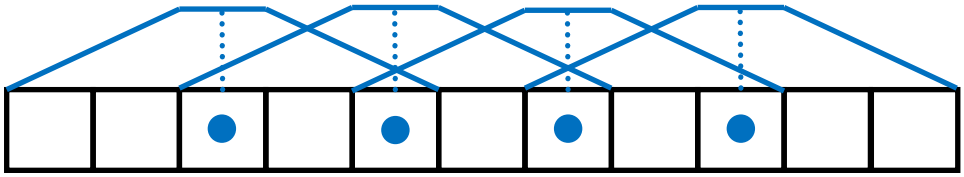
Достижение ускорения возможно за счет разработки на базе существующего алгоритма нового, ориентированного на архитектуру CUDA. Эта программно-аппаратная архитектура, позволяющая производить вычисления с использованием графического процессора, строится на концепции, что GPU выступает в роли массивно параллельного сопроцессора к CPU [1], имеет определенный подход к распределению подзадач между процессорами, а также сложную структуру памяти. А именно, CUDA предоставляет несколько типов памяти, каждая из которых обладает характеристиками: доступ, уровень выделения, скорость работы. Реализовать простейший алгоритм решения вейвлет-преобразования на CUDA не трудно, но оптимизировать таким образом, чтобы обеспечить максимальное ускорение по сравнению с последовательной версией алгоритма, значительно трудней.

В данной работе описано несколько подходов к оптимизации алгоритма вейвлет-преобразования и показана эффективность каждого из способов.

## Распределение данных

Вопрос распределения данных по процессорам и связь этого распределения с эффективностью параллельной программы является основным вопросом параллельного программирования. Поэтому для достижения максимального ускорения необходимо правильно выделить подзадачи.

Для вычисления вейвлет-преобразования функции в точке *A* требуется владеть только частью этой функции. А именно, нам необходимо знать значения функции в некоторой окрестности точки *A*, размер которой равен длине вейвлета. Есть различные способы применения вейвлет-преобразования [5, 6]. Мы будем применять вейвлет-преобразование не к каждой точке, а через одну. Разработанные алгоритмы достаточно легко модифицировать для обработки каждой точки функции, поэтому в дальнейшем мы не будем заострять на этом внимание. Благодаря тому, что для каждой точки производятся одинаковые вычисления, нашу задачу легко можно разбить на независимые подзадачи, каждая из которых заключается в вычислении значения вейвлет-преобразования в одной точке функции. Такое разделение на подзадачи приводит к тому, что они будут иметь очень большой объём общих данных. Это показано на следующем рисунке.

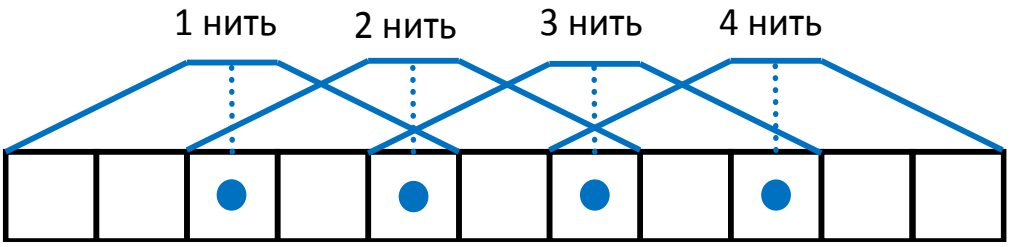


Точками обозначены элементы массива, в которые мы сдвигаем преобразованный вейвлет. Дугами обозначены окрестности тех элементов массива, к которым применяется данный вейвлет.

Во многих архитектурах параллельных вычислительных систем при таком распределение данных пришлось бы содержать множество копий одних и те же данных в локальной памяти различных потоков. Это существенно увеличило бы требуемый объём памяти, и, в конечном итоге, время работы алгоритма. Архитектура CUDA хороша тем, что обладает несколькими типами памяти, имеющими разные уровни доступа и разную латентность. А именно, у каждой нити имеются регистры, у каждого блока есть разделяемая память, работающая почти так же быстро, как и регистры, и все блоки владеют сравнительно медленной глобальной памятью [1].

Таким образом, целесообразно разместить в одном блоке несколько смежных подзадач, расположив их данные в разделяемой памяти. Это позволит нам не только избежать дублирования данных в подзадачах, находящихся в одном блоке, но и обеспечить максимальную скорость доступа к данным.

Как было сказано выше, нити на GPU обладают крайне небольшой стоимостью создания и для эффективной загрузки процессора необходимо использовать много тысяч нитей. Очевидно, в нашем случае лучше всего распределять подзадачи между нитями так, чтобы каждая нить обрабатывала только одну подзадачу. Это продемонстрированно на следующем рисунке.




Количество нитей равно половине длины входного массива.

## Оптимизация

Можно выделить следующие направления оптимизации:

- ❖ оптимизация работы с памятью;
- ❖ оптимизация арифметических операций;
- ❖ изменение количества нитей в блоке;
- ❖ изменение объёма работы, выполняемой нитью.

## Оптимизация работы с памятью

Архитектура CUDA обладает сложной структурой памяти, и для того, чтобы максимально использовать вычислительную мощность графического процессора, необходимо уделить пристальное внимание оптимизации работы с памятью. Автором было реализовано несколько различных способов работы с памятью, описание которых приведено ниже.

### Первый способ

Этот способ самый простой, но и самый медленный. Данные для преобразования и значения вейвлета помещаем в глобальную память GPU, каждая нить читает нужные ей элементы из глобальной памяти и производит вычисления. Конечный результат хранится так же в глобальной памяти.

Рассмотренный способ явно упирается в скорость доступа к глобальной памяти. Для получения аппроксимирующей и детализирующей частей к одной точке массива применяется два различных фильтра. Поэтому происходит двукратное обращение нити к одним и тем же данным, лежащих в глобальной памяти, то есть очень много избыточных чтений. К тому же, один и тот же элемент входного массива требуется нескольким смежным нитям, что ещё больше увеличивает количество избыточных обращений к глобальной памяти. Так как этот вид памяти обладает очень высокой латентностью, описанный способ не будет давать должного ускорения.

### Второй способ

Отличается от первого способа тем, что данные для преобразования перед началом обработки копируются в разделяемую память. Вейвлет по-прежнему располагается в глобальной памяти.

Разделяемая память работает существенно быстрее глобальной – задержки при доступе составляют порядка 4-6 тактов против 400-600 [4]. В результате время работы ядра сократилось на 38%.

**Третий способ**

Реализован на основе второго способа. Отличие заключается в том, что вейвлет располагаются в константной памяти.

Константная память кэшируется и благодаря этому работает почти так же быстро, как регистровая. Она имеет сравнительно малый объём – 64 кбайт, доступ к ней из кода, выполняющегося на GPU, возможен только для чтения [1]. Но это не является препятствием в данной ситуации, так как количество ненулевых элементов вейвлета в среднем не более 10, и у нас нет необходимости их изменять.

Использование константной памяти позволило сократить время работы ядра ещё на 73%.

**Четвёртый способ**

Возможности повышения эффективности за счёт использования различных типов памяти практически исчерпаны, поэтому сокращение времени работы после внесения в алгоритм дальнейших улучшений будет уже не так велико.

Как известно, для получения значения вейвлет-преобразования в данной точке необходимо сложить произведения вейвлета на значения функции в соответствующих точках. Данный способ отличается от предыдущего тем, что для хранения промежуточного результата сложения используется регистровая память. После окончания вычислений результат помещается в глобальную память.

Уменьшение времени работы по сравнению с предыдущим способом оказалось незначительным и составило 1%. Это можно объяснить тем, что латентность глобальной памяти успешно компенсировалась средствами CUDA.

**Пятый способ**

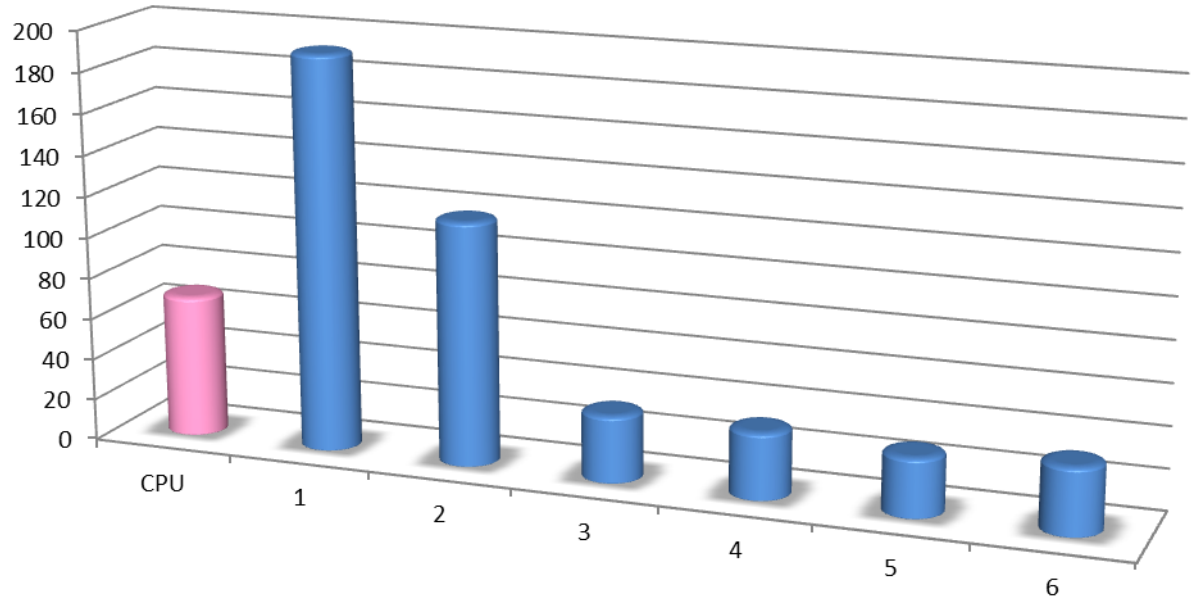
Этот способ отличается от предыдущего использованием константной памяти для передачи параметров в ядро и небольшими изменениями алгоритма определения индекса элемента массива, к которому текущая нить должна применить вейвлет-преобразование. Результат оказался довольно неожиданным – использование каждой из этих модификаций по отдельности увеличивало время работы ядра, но при совместном использовании время работы сократилось на 12%. Видимо, это связано с наложением эффекта от модификации кода с оптимизацией, проводимой компилятором.

**Шестой способ**

Этот способ отличается от пятого способом копирования данных из глобальной памяти в разделяемую, благодаря чему удалось получить объединение запросов к памяти [1]. Это удалось благодаря такому способу копирования данных, при котором соседние нити копируют соседние элементы, а не через один, как ранее. Но это привело к увеличению накладных расходов на определение индексов элементов, подлежащих копированию, в результате чего время работы увеличилось на 13%.

## Сравнение производительности

Результат оптимизации памяти показан на диаграмме. Для сравнения приводится время работы подобного алгоритма на CPU. Тестирование проводилось с использованием видеоадаптера на процессоре Nvidia GeForce 8400 GS и центрального процессора AMD Athlon 64 3800+.



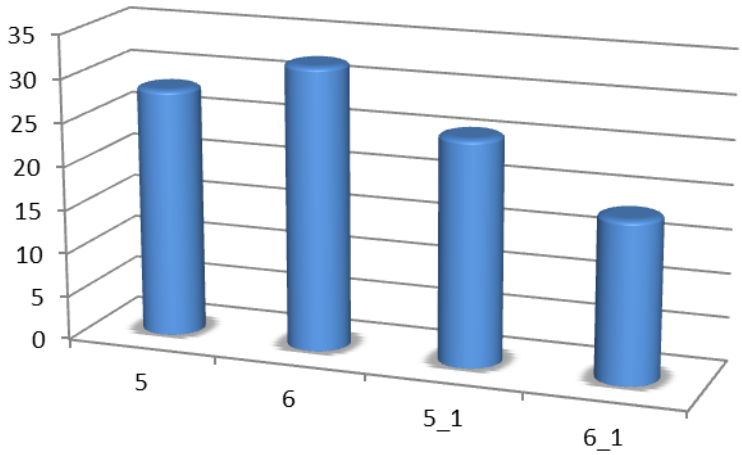

Можно заметить, что оптимизация работы с памятью критически важна для достижения максимальной эффективности алгоритма.

**Оптимизация арифметических операций**

Операции сложения, вычитания, умножения и побитовые операции выполняются быстро – всего за несколько тактов. Но операция деления, которая необходима для определения индексов элементов для нитей, является для GPU NVIDIA очень дорогой [1, 4]. Так как размер сигнала, использовавшегося для тестирования алгоритма, является степенью числа 2, операцию деления можно заменить побитовым сдвигом. В том случае, если сигнал будет иметь другой размер, будет выгодно разделить его на части таким образом, чтобы количество элементов в каждой части было степенью числа 2, и обрабатывать по частям.

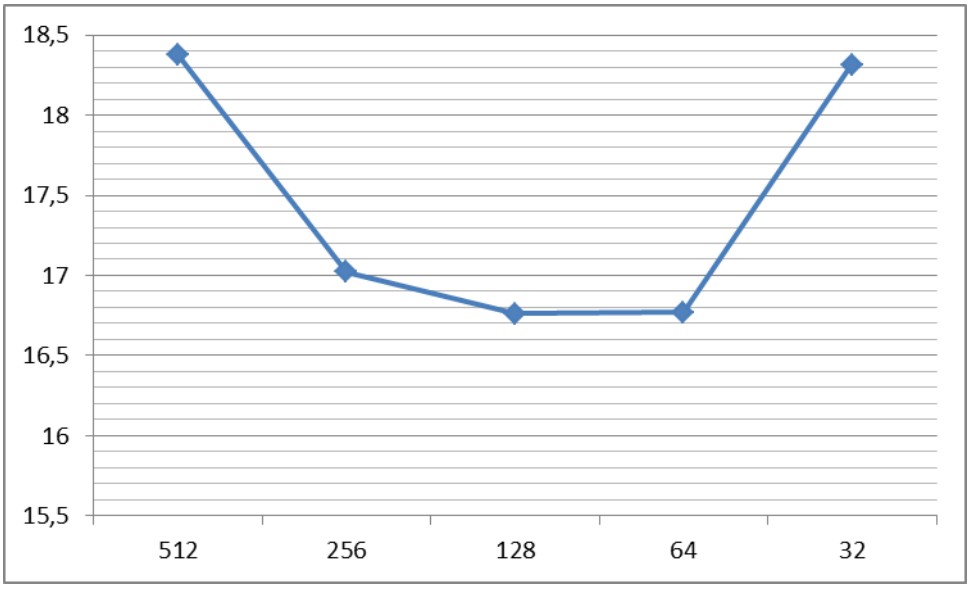
За основу был взят алгоритм, реализующий шестой способ работы с памятью. Несмотря на то, что он работает медленнее пятого, потенциальное ускорение у него выше, так как в нём выполняется больше операций деления и работа с памятью ведётся эффективнее.

После реализации описанных изменений в пятой и шестой версиях алгоритма предположение подтвердилось. Замена операции деления на операцию побитового сдвига позволила не только сократить накладные расходы, связанные с определением индексов, но и сделала хорошо заметным прирост производительности от объединения запросов к глобальной памяти. В итоге пятый способ стал быстрее на 10%, а шестой – на 43% и стал самым быстрым, опережая пятый с той же оптимизацией на 28%.




**Оптимизация количества нитей в блоке**

На следующем графике представлено скорость работы при различном количестве нитей в блоке. Общее количество нитей остаётся постоянным.




Можно увидеть что при количестве нитей 128 в блоке алгоритм работает более эффективно. Мы получили уменьшение времени работы ядра на 9%. То есть самый оптимальный размер блока – 128 нити.

**Выбор размера подзадачи**

От выбора размера подзадачи зависит общее количество нитей.

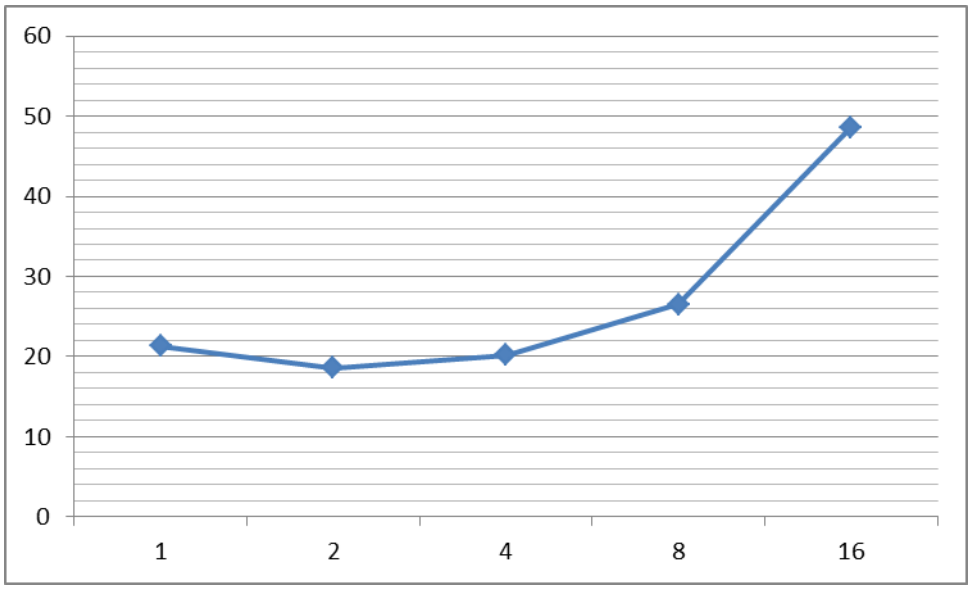
Для полноценной загрузки GPU нужны тысячи нитей [1, 4]:

- ❖ для покрытия латентности операций чтения / записи;
- ❖ для покрытия латентности инструкций, оперирующих числами с плавающей точкой.

Важным моментом при работе с памятью является то, что при большом числе нитей, выполняемых на мультипроцессоре (мультипроцессор поддерживает до 768 нитей), время ожидания warp'ом доступа к памяти может быть использовано для выполнения других warp'ов. Чередование вычислений с обращениями к памяти позволяет более оптимально использовать ресурсы GPU.

 Так, первому warp'у нужен доступ к памяти. Управление передается другому warp'у, выполняется одна команда для него, далее управление опять передается следующему warp'у и т. д. Для того чтобы избежать «простоя» мультипроцессора, достаточно обеспечить большое количество warp'ов, которые смогут выполняться в то время, когда первый warp ждет данных из памяти.

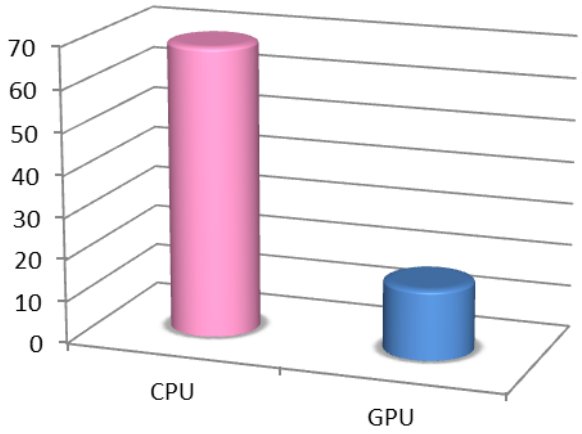
Во всех предыдущих тестах одна нить вычисляла значение и аппроксимации, и детализации для одной точки, общее количество нитей составляло половина от количества элементов в массиве. В данном тесте нити вычисляют от одного до шестнадцати значений выходного массива.




На графике видно, что наибольшее быстродействие достигается при размере подзадачи равном 2. При больших значениях эффективности падает из-за уменьшения числа нитей, при меньших – из-за увеличения доли накладных расходов в целом у всех нитей.

## Сравнение производительности

На данном графике представлены сравнение скорости работы последовательного алгоритма и самого быстрого параллельного алгоритма. Ускорение составило 76%.




## Литература

1. Боресков А.В., Харламов А. А. Основы работы с технологией CUDA. - М.: ДМК Пресс, 2010. - 232 с.: ил.

2. Воробьев В.И. Грибунин В.Г. – Теория и практика вейвлет-преобразования. – ВУС, 1999. С1-204.

3. Добеши И. Десять лекций по вейвлетам. – Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001, 464 стр.

4. Дымченко Л. Параллельные вычислительные процессоры NVIDIA: настоящее и будущее. URL: http://nvworld.ru/articles/cuda_parallel/

5. Киселев А. Основы теории вейвлет-преобразования. URL: http://www.basegroup.ru/library/cleaning/intro-to-wavelets/

6. Левкович-Маслюк Л. Дайджест вэйвлет-анализа, в двух формулах и 22 рисунках. URL: http://www.computerra.ru/offline/1998/236/1123/