**Сравнение реализации вычисление интеграла в CPU и GPU с использованием механизмов CUDA.**

**П.С. Кондратьев**[[1]](#footnote-1)

Краткая аннотация: Использование GPU и технологии CUDA для неграфических вычислений является эффективным решением для увеличения скорости самих вычислений, и как следствие уменьшения временных и материальных затрат.

**Введение**

Недавние разработки ведущих производителей микросхем, таких как NVIDIA, со всей очевидностью показали, что будущие микропроцессоры и крупные высокопроизводительные вычислительные системы (НРС) будут гибридными (гетерогенными). В их основу будут положены компоненты двух основных типов в разных пропорциях:

* мульти ядерные и многоядерные центральные процессоры: количество ядер будет и дальше возрастать из- за желания поместить все больше компонентов на один кристалл, не упираясь в барьер мощности, памяти и параллелизма на уровне команд;
* специализированное оборудование и массивно-параллельные ускорители: например, графические процессоры (GPU) от NVIDIA в последние годы превзошли стандартные CPU в производительности вычислений с плавающей точкой. Да и программировать их стало так же просто, как многоядерные GPU (если не проще).

Руководствуясь идеей о том, что, какие методы распараллеливания существуют было взято сравнение реализации вычисление интеграла в CPU и GPU с использованием механизмов CUDA, так как эта тема актуальна. На просторах интернета и официальной документации CUDA, приводят различные примеры работы, но ничего не связано с быстродействием вычисления интеграла на определенном промежутке.

Результатом работы будут являться две программы, выполняющие вычисления на графическом процессоре и две программы выполняющие расчёты на центральном процессоре. При использовании как графического, так и центрального процессоров, алгоритмы и численные методы проведения расчётов в соответствующих программах, одинаковы.

Для оценки скорости вычисления на CPU и GPU, вычислим интеграл методом прямоугольников на отрезке от 0 до 1.

**Программная функция вычисления в CPU**

Основная задача CPU, если говорить простыми словами, это выполнение цепочки инструкций за максимально короткое время. CPU спроектирован таким образом, чтобы выполнять несколько таких цепочек одновременно или разбивать один поток инструкций на несколько и, после выполнения их по отдельности, сливать их снова в одну, в правильном порядке. Каждая инструкция в потоке зависит от следующих за ней, и именно поэтому в CPU так мало исполнительных блоков, а весь упор делается на скорость выполнения и уменьшение простоев, что достигается при помощи кэш-памяти и конвейера.

Программная функция вычисления интеграла сводиться к простым алгоритмическим действиям, таких как нахождения шага сетки в методе прямоугольника и последующих вычислений функции на отрезке.

**Особенности использования GPU и программная функция вычисления интеграла в его среде**

Графический процессор основан на так называемой архитектуре SIMT (SingleInstruction, MultipleThread). Технология CUDA позволяет определять специальные функции – ядра (kernels), которые выполняются параллельно на CPU в виде множества различных потоков (threads). Таким образом, ядро является аналогом потоковой функции. Каждый поток исполняется на одном CUDA-ядре, используя собственный стек инструкций и локальную память.

Отдельные потоки группируются в блоки потоков (threadblock) одинакового размера, при этом каждый блок потоков выполняется на отдельном мультипроцессоре. Потоки внутри блока потоков могут эффективно взаимодействовать между собой с помощью общих данных в разделяемой памяти и синхронизации. Кроме того, потоки могут взаимодействовать при помощи глобальной памяти и атомарных операций.

На аппаратном уровне потоки блока группируются в так называемые варпы (warps) по 32 элемента (на всех текущих устройствах), внутри которых все потоки параллельно выполняют одинаковые инструкции (по принципу SIMD). Важным моментом является то, что потоки фактически выполняют одну и ту же команды, но каждая со своими данными. Поэтому если внутри варпа происходит ветвление (например, в результате выполнения оператора if), то все нити варпа выполняют все возникающие при этом ветви. По этой причине операции ветвления могут негативно сказываться на производительности – различные пути не могут выполняться параллельно (в тоже время потоки одного варпа, выполняющие один путь, работают параллельно).

В свою очередь, блоки потоков объединяются в решетки блоков потоков (gridofthreadblocks). Следует отметить, что взаимодействие потоков из разных блоков во время работы ядра затруднено: отсутствуют явные инструкции синхронизации, взаимодействие возможно через глобальную память и использованием атомарных функций (другим вариантом является разбиение ядра на несколько ядер без внутреннего взаимодействия между потоками разных блоков).

**Планирование экспериментов (получаем максимальный эффект от распараллеливания)**

Для объекта вычисления интеграла был получен набор данных для вычисления по методы прямоугольника. Эксперимент заключается в доказательстве использование GPU и технологии CUDA для неграфических вычислений эффективным решением для увеличения скорости самих вычислений, и как следствие уменьшения временных и материальных затрат.

Поэтому для проверки скорости вычислений на CPU и GPU, будут использованы программы, производящие идентичные вычисления, также для сравнения будет сделана рабочая нагрузка для оценки работоспособности, для выявления плюсов и минусов данного подхода к использованию вычисления интеграла.

**Проведение экспериментов и анализ результатов**

Для эксперимента с CPU был написан метод прямоугольников с рабочей нагрузкой (блоков разбиения функции) на отрезке от i = 100 до 1000000 с шагом = i \* 10. Время рассчитывалось с использованием функции clock() описываемой в time.h и возвращает число тиков от момента загрузки программы. Тик обычно равен 1 миллисекунде, но для возможности в последующем работать с другой длительностью такта в time.h фиксируется константа CLOCKS\_PER\_SEC (время изерения было взять в нс).

Для эксперимента с GPU было взято максимальное количество нитей и блоков (взята в настойках CUDA DEVICES).

int threadsPerBlock = 1024;

int blocksPerGrid = (SIZE\_DARR + threadsPerBlock - 1) / threadsPerBlock;

Листинг 1. Определение границ и центра

Рабочая нагрузка была определена такая же, как и на CPU (отрезок от 100 до 1000000).

Были реализованы основные этапы алгоритма для реализации эксперимента:

**Основные этапы CUDA-программы**

1. Хост выделяет нужное количество памяти на устройстве.,

Хост копирует данные из своей памяти в память устройства.

Хост стартует выполнение определенных ядер на устройстве.

Устройство выполняет ядра

Хост копирует результаты из памяти устройства в свою память.

Естественно, для наибольшей эффективности использования GPU нужно чтобы соотношение времени, потраченного на работу ядер, к времени, потраченному на выделение памяти и перемещение данных, было как можно больше.

Ниже представлена таблица результатов времени вычисления интеграла методом прямоугольников на CPU и GPU от увеличения количества блоков разбиения функции (Size) для подсчета интеграла (табл. 1).

Таблица 1. временя вычисления интеграла методом прямоугольников на CPU и GPU

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size | 100 | 1000 | 10000 | 100000 | 1000000 |
| CPU | 0 ms | 0 ms | 2 ms | 11 ms | 135 ms |
| GPU | 1.03 ms | 1.04 ms | 1.26 ms | 1.94 ms | 4.46 ms |

Из полученных результатов видно, что время, затраченное на расчёты с применением технологии CUDA меньше (в данном случаи примерно на 30% при 1000000). При многократном повторении расчётов, средний выигрыш в скорости остаётся примерно на таком же уровне.

Так же из таблицы видно, что при маленьких значениях CUDA работает значительно хуже, чем CPU и из этого можно сделать вывод то CUDA использовать лучше при больших числовых данных, так как время тратиться на создание блоков и нитей, выделения памяти для передачи данных в GPU и после чего извлечения данных оттуда. Поэтому можно выделить оптимальные надстройки для CUDA, для более эффективной работы при больших значениях.

**Оптимизации вычислений на CUDA**

* Важно свести к минимуму передачу данных между памятью GPU и ОЗУ
* Исходные данные для расчета загружаются в память GPU единожды перед началом моделирования. Далее обмен между ОЗУ и GPU отсутствует кроме моментов времени, результаты которых нужно сохранить.
* Вычисления всех уравнений, кроме расчета накачки, происходят параллельно
* Все загружаемые в память GPU данные хранятся в глобальной памяти

Таблица 2. Основные технические характеристики ПК

|  |  |
| --- | --- |
| Компонент | Характеристики |
| Процессор (CPU) | I5 – 5700U 2,5 – 2,5 ГГц, 2 ядра, 4 потока |
| Оперативная память (RAM) | DDR4 8 ГБ |
| Графический контроллер | Intel HD Graphics 620  NVIDIA GeForce 940MX |
| Объем памяти видеокарты | 1 Гб  2 Гб |

Заключение

В ходе исследовательской работы были выявлены **Основные различия между архитектурами CPU** (центральный процессор) и **GPU**:

1. Ядра CPU созданы для исполнения одного потока последовательных инструкций с максимальной производительностью
2. GPU проектируются для быстрого исполнения большого числа параллельно выполняемых потоков инструкций.
3. Универсальные процессоры оптимизированы для достижения высокой производительности единственного потока команд, обрабатывающего и целые числа, и числа с плавающей точкой. При этом доступ к памяти случайный.
4. Разработчики CPU стараются добиться выполнения как можно большего числа инструкций параллельно, для увеличения производительности. Для этого, начиная с процессоров IntelPentium, появилось суперскалярное выполнение, обеспечивающее выполнение двух инструкций за такт. Но у параллельного выполнения последовательного потока инструкций есть определённые базовые ограничения и увеличением количества исполнительных блоков кратного увеличения скорости не добиться.
5. У видеочипов работа простая и распараллеленная изначально. Видеочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели. Обработка полигонов и пикселей независима, их можно обрабатывать параллельно, отдельно друг от друга. Поэтому, из-за изначально параллельной организации работы в GPU используется большое количество исполнительных блоков, которые легко загрузить, в отличие от последовательного потока инструкций для CPU.
6. GPU отличается от CPU ещё и по принципам доступа к памяти. В GPU он связанный и легко предсказуемый — если из памяти читается пиксель текстуры, то через некоторое время придёт время и для соседних пиксель. Да и при записи то же — пиксель записывается во фреймбуфер, и через несколько тактов будет записываться расположенный рядом с ним. Поэтому организация памяти отличается от той, что используется в CPU. Видеочипу, в отличие от универсальных процессоров, просто не нужна кэш-память большого размера, а для текстур требуются лишь несколько килобайт.
7. Работа с памятью у GPU и CPU несколько отличается. Так, не все центральные процессоры имеют встроенные контроллеры памяти, а у всех GPU обычно есть по несколько контроллеров. Кроме того, на видеокартах применяется более быстрая память, и в результате видеочипам доступна в разы большая пропускная способность памяти, что также весьма важно для параллельных расчётов, оперирующих с огромными потоками данных.
8. В универсальных процессорах большие количества транзисторов и площадь чипа идут на буферы команд, аппаратное предсказание ветвления и огромные объёмы чиповой кэш - памяти. Все эти аппаратные блоки нужны для ускорения исполнения немногочисленных потоков команд. Видеочипы тратят транзисторы на массивы исполнительных блоков, управляющие потоками блоки, разделяемую память небольшого объёма и контроллеры памяти на несколько каналов. Вышеперечисленное не ускоряет выполнение отдельных потоков, оно позволяет чипу обрабатывать нескольких тысяч потоков, одновременно исполняющихся чипом и требующих высокой пропускной способности памяти.
9. Универсальные центральные процессоры используют кэш - память для увеличения производительности за счёт снижения задержек доступа к памяти, а GPU используют кэш или общую память для увеличения полосы пропускания. CPU снижают задержки доступа к памяти при помощи кэш-памяти большого размера, а также предсказания ветвлений кода. Эти аппаратные части занимают большую часть площади чипа и потребляют много энергии. Видеочипы обходят проблему задержек доступа к памяти при помощи одновременного исполнения тысяч потоков — в то время, когда один из потоков ожидает данных из памяти, видеочип может выполнять вычисления другого потока без ожидания и задержек.
10. Есть множество различий и в поддержке многопоточности. CPU исполняет 1-2 потока вычислений на одно процессорное ядро, а видеочипы могут поддерживать до 1024 потоков на каждый мультипроцессор, которых в чипе несколько штук. И если переключение с одного потока на другой для CPU стоит сотни тактов, то GPU переключает несколько потоков за один такт.
11. Кроме того, центральные процессоры используют SIMD блоки для векторных вычислений, а видеочипы применяют SIMT (одна инструкция и несколько потоков) для скалярной обработки потоков. SIMT не требует, чтобы разработчик преобразовывал данные в векторы, и допускает произвольные ветвления в потоках.

Ниже приведен график зависимости времени работы от числа разбиения на прямоугольники интеграла.

Рис. 1. График зависимости времени выполнения от объема входных данных

Список литературы

1. Д Сандерс Технология CUDA в примерах. Введение в программирование графических процессов / Д. Сандерс, Э. Кэндрот–М.: ДМК Пресс, 2011. –232 с.
2. http://www.ixbt.com/video3/cuda-1.shtml - NVIDIA CUDA - неграфические вычисления на графических процессорах (Дата обращения: 13.05.2019).
3. <http://top50.supercomputers.ru> - TOP50 суперкомпьютеров процессорах (Дата обращения: 17.05.2019).
4. http://habr.com/ru/post/54707/ - CUDA: Как работает GPU процессорах (Дата обращения: 13.05.2019).
5. <https://habr.com/ru/post/211194/> - Пример оптимизации вычислений на CUDA (Дата обращения: 13.05.2019).

1. 432027, Ульяновск, ул. Северный Венец, 32, УлГТУ, e-mail: Pablo.osamu@yandex.ru [↑](#footnote-ref-1)