NickWare Group Science
Бабич Н. А.
15.01.2018

ЭФФЕКТИВНЫЕ ВЫЧИСЛЕНИЯ В ГЕТЕРОГЕННЫХ СИСТЕМАХ

Введение в гетерогенные системы

Современные вычислительные системы, используемые специалистами и учёными, могут включать в себя, кроме процессора общего назначения, не один специализированный сопроцессор, каждый из которых предназначен для решения определённого круга задач. Такие системы называются гетерогенными (от греч. ἕτερος — разный; γένω — рождать) [1].

Современная история гетерогенных систем начинается с возникновения идеи использования графических процессоров видеокарт для неграфических вычислений.

Центральный процессор (CPU) представляет собой универсальный процессор или процессор общего назначения, оптимизированный для достижения высокой производительности единственного потока команд, обрабатывающего и целые числа, и числа с плавающей точкой. Графический процессор (GPU) устроен принципиально иначе. Он изначально проектировался для выполнения огромного количества параллельных потоков. Причём эти потоки распараллелены изначально, и никаких накладных расходов на распараллеливание в графическом процессоре нет [2]. В этом и заключается принципиальное отличие графических процессоров от центральных.

Таким образом, благодаря архитектуре графического процессора вычисления на нём выполняются параллельно, что существенно увеличивает скорость работы некоторых алгоритмов. Некоторых – потому что не всякий алгоритм можно разделить на отдельные независимые подзадачи, как этого требует идеология параллельных вычислений [3].

GPGPU – шаг к гетерогенным вычислениям

Когда возникла необходимость использовать графический процессор для неграфических вычислений, предпринимались попытки применения графических API (таких, как Direct3D и OpenGL) для этих целей. Такой подход был сопряжён с определёнными трудностями: разработчикам программного обеспечения было необходимо изучать принципы работы с 3D-объектами (шейдерами, текстурами и т. д.) и «подгонять» вычислительные задачи под их специфику.

Эти сложности послужили толчком к появлению идеологии GPGPU (General-purpose computing on GPU — вычисления общего назначения на графических процессорах) в 2001 году. Через некоторое время появилось расширение для языка программирования C под названием BrookGPU. Оно служило своего рода «прослойкой» между графическими API и

языком C – программистам больше не требовалось работать с Direct3D и OpenGL напрямую. Можно было работать в привычной среде, а компилятор, с помощью специальных библиотек, реализовывал взаимодействие с графическим процессором на низком уровне [4]. Крупные корпорации также заинтересовались идеей неграфических вычислений на графическом процессоре.

Таким образом, в 2007 году корпорация NVIDIA представила новую архитектуру графических процессоров G80, а также разработанную специально для неё новую технологию программирования параллельных вычислений CUDA (Compute Unified Device Architecture) [5].

Вершинные и пиксельные конвейеры были заменены на единое арифметическое устройство, способное выполнять вершинные, геометрические, пиксельные и арифметические вычисления, а благодаря модели SIMT (Single-Instruction Multiple-Thread) одна инструкция применяется к нескольким потокам. Каждый поток имеет прямой доступ к общей разделяемой памяти, что значительно увеличивает производительность [5].

Технология CUDA позволила программистам выполнять вычисления на графическом процессоре используя специальные функции, добавляемые библиотеками, на языках высокого уровня. Параллельные участки программы выполняются на графическом процессоре как так называемые ядра (kernel) – особые элементы, состоящие из большого набора параллельных потоков (thread), выполняющих элементарные операции. Взаимодействующие друг с другом потоки объединяются в блоки (thread block), в пределах которых они имеют общую разделяемую память. В свою очередь, блоки объединяются в сетку (grid). Часть программы, которая выполняется на центральном процессоре, называется host. Она выполняет управляющие функции по работе с устройством [6].

В компании AMD (в то время – ATI) решили не разрабатывать свою технологию программирования параллельных вычислений под свой графический процессор, аналогичную CUDA, а поддержали разработку технологии OpenCL (Open Computing Language). Таким образом, в 2009 году был выпущен набор разработчика ATI Stream SDK v2.0 с поддержкой OpenCL, который теперь называется AMD Accelerated Parallel Processing (AMD APP) [7]. Немного позднее и NVIDIA включила поддержку OpenCL в состав своего набора разработчика.

Дальнейшие усилия AMD были направлены на увеличение скорости обмена данными между центральным и графическим процессором, ведь в случае низкой пропускной

способности все преимущества вычислений на графическом процессоре пропадают. Для решения этой проблемы была разработана технология Fusion, позволяющая разместить CPU и GPU на одной подложке. В 2011 году был выпущен первый APU (Accelerated Processing Unit) AMD Llano. Графический процессор не имел отдельной, собственной памяти, а использовал оперативную память наряду с центральным, что позволило существенно ускорить обмен данными между процессорами и вывести гетерогенные вычисления на новый уровень [8].

Однако, при этом, каждый из процессоров по-прежнему имел собственную виртуальную память, хоть и физически это было одно устройство. Чтобы передать данные от одного процессора к другому приходилось копировать данные из одного участка оперативной памяти в другой, что тоже не является оптимальным решением. Тогда возникла идея организации общей виртуальной памяти, что позволило бы передавать от одного процессора к другому только указатели, без копирования самих данных. Эта концепция получила название Heterogeneous System Architecture (HSA). В 2012 году был образован фонд HSA Foundation, который занимается разработкой спецификаций HSA [9].

Корпорация Intel также не отставала от конкурентов – разработчики также объединили центральный и графический процессор в одном кристалле и активно поддерживали средства для разработки приложений с использованием OpenCL. В 2016 году состоялся выпуск вычислительного сопроцессора Intel Xeon Phi, основанного на архитектуре Intel MIC (Many Integrated Core – большое количество ядер в одной микросхеме). Он представляет собой многоядерный чип (более 60 ядер архитектуры x86), подключаемый к ПК в качестве платы расширения PCI Express [10]. Он хорошо подходит для параллельных вычислений.

OpenCL – ключ к гетерогенным вычислениям

OpenCL – открытый стандарт для описания параллельных вычислений на языках высокого уровня. Он первоначально был разработан в компании Apple Inc. Apple внесла предложения по разработке спецификации в комитет Khronos. 16 июня 2008 года была образована рабочая группа Khronos Compute для разработки спецификаций OpenCL. В неё вошли Apple, NVIDIA, AMD, IBM, Intel, ARM, Motorola и другие компании. Работа велась в течение пяти месяцев, по истечении которых организация Khronos Group представила первую версию стандарта. OpenCL 1.0 был впервые показан общественности 9 июня 2008 [11].

Поскольку стандарт открытый и поддерживается огромным количеством компаний, один раз написанный программный код можно компилировать для выполнения на множестве различных устройств (в том числе, и на FPGA) практически без изменений, при условии, что устройство поддерживает технологию OpenCL и производитель предоставил набор разработчика для этого устройства со всеми необходимыми библиотеками. Это даёт значительные преимущества OpenCL над CUDA и, по сути, выносит его далеко за рамки концепции GPGPU. На рисунке 1 показана схема взаимодействия средств разработки.

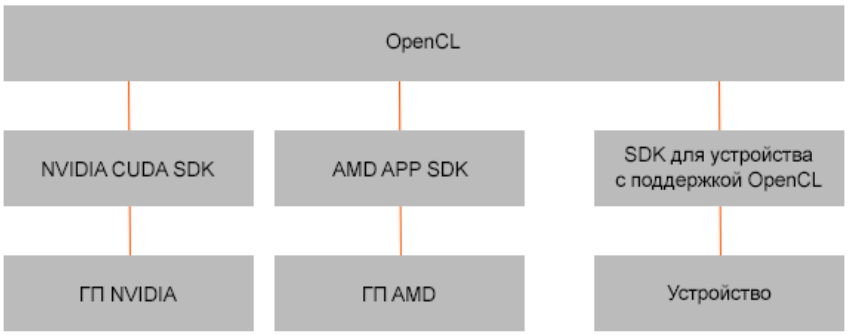

Рисунок 1 – Схема взаимодействия средств разработки

Модель программирования OpenCL в целом схожа с моделью CUDA, но имеет несколько важных отличий. Во-первых, в OpenCL специальным спецификатором нужно помечать только те функции, которые выполняются на устройстве (спецификатор __kernel). Во-вторых, в OpenCL предусмотрены специальные функции, позволяющие получить информацию о глобальном индексе конкретного потока и размере вычислительного пространства, в то время как в CUDA необходимо вычислять эти значения самостоятельно. В-третьих, OpenCL предоставляет гораздо больше возможностей для гибкого управления параллельным выполнением задач посредством очереди (cl_command_queue). В-четвёртых, OpenCL поддерживает компиляцию kernel прямо во время выполнения host-программы. Различия в терминологиях технологий представлены в таблице 1 [12]. Общее количество work-item`ов также называется глобальной рабочей группой, а work-item`ы, входящие в один work-group – локальной рабочей группой. Размер глобальной рабочей группы должен быть кратен размеру локальной.

Таблица 1 – Различия в терминологии CUDA и OpenCL

Терминология CUDA	Терминология OpenCL
Thread	Work-item
Thread block	Work-group
Shared memory	Local memory
Local memory	Private memory

О других различиях можно прочитать на официальной странице AMD [12]. На ней также описываются нюансы перехода от программной модели CUDA к программной модели OpenCL.

От теории к практике

Для того, чтобы показать эффективность использования OpenCL, было проведено несколько экспериментов. В качестве тестовой задачи была взята задача факторизации чисел (разделения на простые множители). В качестве алгоритма факторизации был взят метод пробных делений. Этот подход основывается на идее, что если у числа существует простой делитель, отличный от него самого, то он не превышает корня из числа. Для факторизации числа нужно перебрать все числа в промежутке $[2, \sqrt{N}]$, и попытаться разделить разлагаемое число на каждое из них по очереди (найти остаток от деления). Сложность алгоритма составляет $O(\sqrt{N})$, где N – разлагаемое на простые множители число [13]. Следует отметить, что алгоритм хорошо распараллеливается и подходит для экспериментов.

Алгоритм был построен в двух реализациях на языке C++: для выполнения на центральном процессоре и на графическом, с использованием OpenCL. В качестве kernel в OpenCL-реализации была взята функция нахождения остатка от деления (то есть, каждый поток выполнял операцию нахождения остатка от деления факторизуемого числа на соответствующее число в промежутке $[2, \sqrt{N}]$). Используемый тестовый стенд базировался на основе шестиядерного центрального процессора AMD FX-6300 с тактовой частотой 3.6 ГГц (для вычислений использовалось только одно ядро) и видеокарты NVIDIA GeForce GTX 1060 с тактовой частотой графического процессора 1.7 ГГц и 1280 CUDA-ядрами.

Таким образом, было проведено $9 \cdot 10^6$ факторизаций числа 9998241 для каждой из реализаций. На рисунке 2 показано время, затраченное на вычисления для разных реализаций.

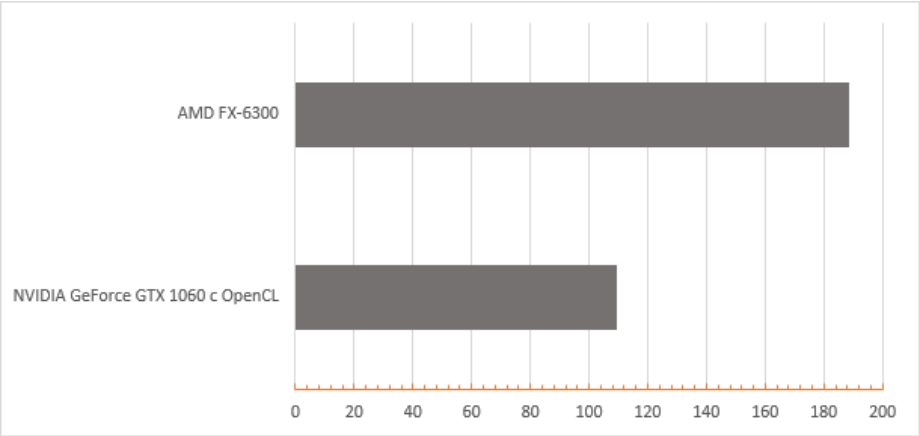

Рисунок 2 – Зависимость времени выполнения (горизонтальная ось, секунды) от реализации (вертикальная ось)

Как видно из графика, вычисления на графическом процессоре выполняются примерно в 1.73 раза быстрее, чем на центральном. Однако, учитывая все описанные преимущества графических процессоров перед процессорами общего назначения, такое увеличение скорости вычислений кажется незначительным. Дело в том, что размер глобальной рабочей группы (и количество потоков) в OpenCL-реализации равен числу 3161 (целая часть корня из факторизуемого числа 9998241), а, так как оно простое, размер локальной рабочей группы был выбран равным единице (единственный доступный вариант в данном случае). Такой вариант является самым неоптимальным – получается, что в одно и то же время мог выполняться только один поток. Это нивелирует практически все преимущества архитектуры графических процессоров и идеологии распараллеливания вычислений.

Чтобы избежать такого падения производительности необходимо увеличить размер локальной рабочей группы. Оптимальным её размером является число, кратное значению параметра Preferred Work Group Size Multiple (PWGSM), что подтверждают рекомендации, указанные на сайте Intel [14]. Его можно получить, обратившись к методу getWorkGroupInfo объекта cl::Kernel с параметром CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE [15]. Следуя тем же рекомендациям, размер локальной группы должен быть кратен 8. Также стоит учитывать общее количество вычислительных ядер устройства – размер локальной группы должен быть таким, чтобы это количество было ему кратно. Чем меньше частей, на которые будет оно поделено, тем лучше, так как будет меньше ресурсов затрачиваться на управление этими

группами. Здесь важно отметить, что размер группы ограничен определённым максимальным значением, которое можно получить, вызвав метод getInfo объекта cl::Device с параметром CL_DEVICE_MAX_WORK_GROUP_SIZE [15].

Таким образом, учитывая все указанные требования, можно получить наиболее оптимальный размер локальной рабочей группы, при котором скорость вычислений будет максимальна. Для графического процессора, используемого в тестовом стенде, он будет равен 640. Как уже отмечалось, размер глобальной рабочей группы должен быть кратен размеру локальной. Определившись с размером последней, выберем размер глобальной рабочей группы равным 3200. Рабочих потоков по прежнему будет 3161, а остальные будут завершаться без вычислений (этого можно добиться, добавив специальное условие выхода в функцию kernel`a). На рисунке 3 показаны результаты измерения времени вычислений с подобранными оптимальными значениями рабочих групп.

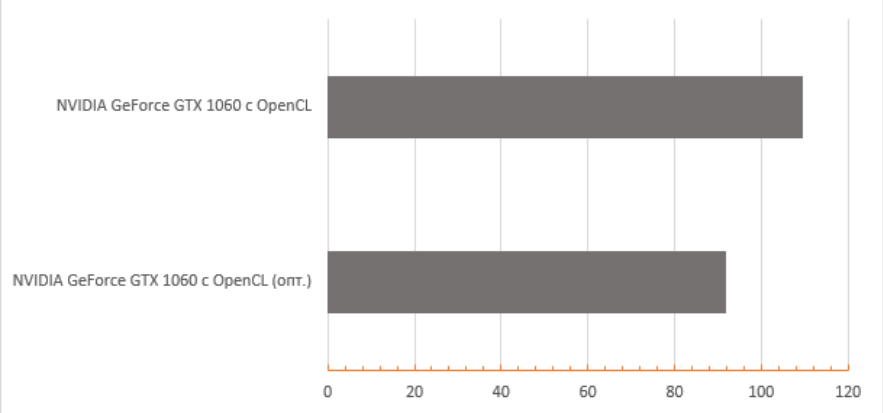

Рисунок 3 – Зависимость времени выполнения (горизонтальная ось, секунды) от реализации (вертикальная ось)

Как видно из графика, оптимизация размеров рабочих групп ускорила вычисления примерно на 18 секунд. По сравнению с реализацией программы для центрального процессора, такая OpenCL-реализация даёт прирост скорости в 2 раза. Использование других устройств, например, ПЛИС, дало бы возможность ещё значительнее снизить время, затрачиваемое на вычисления.

Автором статьи была разработана библиотека OpenCL Optimization Library (OCLOL), которая позволяет автоматически подбирать размеры глобальной и локальной рабочих групп, учитывая «объём» вычислений и характеристики устройства, на котором они

будут производиться. Для использования библиотеки достаточно создать объект класса OCLOL, передав ему объекты cl::Device и cl::Kernel, информацию об «объёме» вычислений (фактический размер глобальной рабочей группы) и общее количество вычислительных ядер устройства, а затем вызвать метод PerformOptimization(). Подобранные размеры глобальной и локальной рабочих групп можно получить методами GetGlobalWorkGroupSize() и GetLocalWorkGroupSize() соответственно. Использование этой библиотеки может дать прирост производительности до 16%. Библиотека доступна для загрузки в открытом репозитории GitHub [16].

Следует отметить, что с выходом OpenCL 2.0 появилась возможность задавать неоднородные рабочие группы: объём глобальной рабочей группы может быть не кратен объёму локальной. При запуске выполняемого модуля, он попытается объединить как можно больше work-item`ов в группы указанного размера, а оставшееся количество будет объединено в группу меньшего размера [17].

Таким образом, развитие гетерогенных систем вышло далеко за рамки концепции GPGPU. В настоящее время создаются новые решения, позволяющие эффективнее решать задачи самых разных отраслей. Появляются новые сопроцессоры, рассчитанные, например, на работу с нейронными сетями или на более эффективное выполнение параллельных алгоритмов. Технология OpenCL же представляет собой универсальный и доступный инструмент для работы с аппаратным обеспечением в гетерогенных системах. Однако, при использовании данной (и не только) технологии программистам следует учитывать особенности архитектуры используемого устройства, чтобы не потерять её преимуществ и сделать вычисления максимально быстрыми.

Список использованных источников

1. Ушаков Д. Н. Большой толковый словарь современного русского языка: 180000 слов и словосочетаний – М. : Альта-Принт [и др.], 2008. –1239 с.
2. Архитектура современных графических процессоров. URL: http://radeon.ru/articles/technology/chiparch/ (дата обращения: 01.01.2018).
3. Бабич Н. А. Параллельные вычисления в среде MATLAB // Студенческий: электрон. научн. журн. 2017. № 19 (19). URL: https://sibac.info/journal/student/19/89438 (дата обращения: 01.01.2018).
4. Вычисления на GPU: мифы и реальность. URL: http://compress.ru/article.aspx?id=23724 (дата обращения: 02.01.2018).

5. Развитие гетерогенных вычислительных систем типа CPU-GPU. URL: http://mcst.ru/files/557429/ed0cd8/50f326/000000/d.a._petrykin_razvitie_geterogennyh_vychislitelnyh_sistem_tipa_cpu-gpu.pdf (дата обращения: 02.01.2018).
6. CUDA C Programming Guide. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (дата обращения: 02.01.2018).
7. APP SDK – A Complete Development Platform. URL: https://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/ (дата обращения: 02.01.2018).
8. AMD Fusion APU Era Begins. URL: http://www.amd.com/en-us/press-releases/Pages/amd-fusion-apu-era-2011jan04.aspx (дата обращения: 05.01.2018).
9. Гетерогенная архитектура для CPU, GPU и DSP. URL: https://www.osp.ru/os/2013/08/13037850/ (дата обращения: 05.01.2018).
10. Процессоры Intel Xeon Phi. URL: https://www.intel.ru/content/www/ru/ru/products/processors/xeon-phi/xeon-phi-processors.html (дата обращения: 05.01.2018).
11. Khronos Group. OpenCL. URL: https://www.khronos.org/registry/OpenCL/ (дата обращения: 01.01.2018).
12. OpenCL and the AMD APP SDK. URL: https://developer.amd.com/resources/articles-whitepapers/opencl-and-the-amd-app-sdk/ (дата обращения: 01.01.2018).
13. Gardner M. A New Kind of Cipher that Would Take Millions of Years to Break // Sci. Amer. — New York City: Nature Publishing Group, 1977. — Iss. 237.
14. Work-Group Size Considerations. URL: https://software.intel.com/sites/landingpage/opencl/optimization-guide/Work-Group_Size_Considerations.htm (дата обращения: 10.01.2018).
15. Khronos OpenCL Registry. URL: https://www.khronos.org/registry/OpenCL/ (дата обращения: 10.01.2018).
16. OpenCL Optimization Library, открытый репозиторий GitHub. URL: https://github.com/nickware44/OCLOL (дата обращения: 10.01.2018).
17. OpenCL 2.0 Non-Uniform Work-Groups. URL: https://software.intel.com/en-us/articles/opencl-20-non-uniform-work-groups (дата обращения: 10.01.2018).