БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет НиДО

Специальность ИиТП

Курсовая работа

«Расширение возможностей ОС с использованием GPU»

по дисциплине «Операционные Системы и Среды»

Выполнил студент: Дегтярев А.А.

группа 393551

Зачетная книжка № 902021-26

Минск 2016

**Введение**

В современном мире существует большой класс задач, которые требуют обработки/проведения довольно простых операций на огромных объемах данных. В класс таких задач входит обработка и анализ, изображений, сигналов, звука, расчет физических процессов, машинное обучение ИИ, биологические расчеты и прочее. Процессоры с классической архитектурой не подходят для эффективного решения таких задач, FPGA и специализированные архитектуры дорогостоящие и менее гибки, GPU же почти идеальный вариант. В данной работе, я постараюсь подробнее рассказать о различиях данных архитектур, раскрыть в чем же преимущество использования гетерогенных систем и GPU, в частности. И каким образом можно расширить возможности ОС с использованием GPU как на прикладном, так и на системном уровне.

Практически все передовые технологии основываются на работе с огромными массивами данных: Big Data, AI, Deep Learning, Machine Learning, на слуху у прогрессивного человечества. И это технологии, позволяющие решать наиболее сложные задачи, стоящие перед человечеством во всех областях от транспорта до здравоохранения. Для работы над таким объемом данных требуется огромные мощности, и очевидным способом оптимизировать работу является ее распараллеливание, а для работы отлично подойдут **высокопараллельные** вычислительные системы. Выходом может служить объединение большого числа компьютеров в сеть, и выполнение распределенных вычислений на виртуальной компьютерной архитектуре(кластере). Но если же сфокусироваться на оптимизации процесса для единственной машины, то решение в использовании специфической архитектуры.

Многие современные процессоры имеют несколько ядер, возможность **параллельного(не конкурентного)** исполнения кода, а так же набор расширений (AVX, SIMD/MIMD), для дополнительной поддержки параллелизма, однако зачастую этого бывает недостаточно:

* Современные CPU слишком сложны для многих вычислительных задач (если задача очень простая, то его использование неэффективно как с точки зрения цены и энергии, так и времени). Они **оптимизированы для последовательной обработки данных**, параллелизм был «притянут», значительно позже. Переключение контекста для выполнения какой-то задачи, может на практике вылиться в огромные затраты по времени
* GPU были созданы с нуля для решения простых **массивно-параллельных** задач, на максимальной скорости, при минимальной цене и потреблении энергии и благодаря компьютерным играм, GPU стали достаточно популярны и составляют отличную конкуренцию специализированным векторным процессорам и сложным FPGA.
* Прямо сейчас почти любой компьютер имеет GPU, будь то встроенный Intel HD или дискретные карты от AMD(ATI) или NVIDIA

На идее использования GPU для решения «не специфичных» для него задач построен целый раздел информатики который принято называть **gpgpu computing(programming)** или **General Purpose GPU programming.**

**Программная модель. Парадигма Stream processing.**

Массивно-параллельные системы и GPU в частности довольно ограничены в доступных операциях и программировании. Из-за специфичной архитектуры, они могут быть эффективно использованы при использовании парадигмы **потоковой обработки(Stream processing)**. Где поток(stream) – это набор данных которые ожидают одинаковых вычислений. Потоки обеспечивают **параллелизм данных**. Вычислительное ядра(Compute Kernel) – это программный термин, обозначающий функцию, которая применяется к каждому элементу *потока*. Так как GPU обрабатывает элементы независимо, нет возможности использовать статические или общие данные. Для каждого элемента возможно лишь чтение из input буфера, выполнение операций с этими данными и запись в output буфер. Для GPU также важна арифметическая интенсивность( число операций на объем переданных в память данных), так как доступ в память по шине значительно медленнее. Идеальное задача для GPU это большой объем данных, высокий параллелизм и минимальная зависимость между элементами.

Ниже приведены вычислительные ресурсы доступные при работе с GPU:

-Programmable Processors(программируемые обработчики) – вертексные, примитивные, фрагментные и общие пайплайны позволяют исполнять код на потоках данных

-Растеризатор(Rasterizer) – позволяет создавать фрагмент из интерполированных повертексных констант( например текстурыне координаты или цвет).

-Texture Unit(текстурная единица) – интерфейс работы с памятью доступной для чтения

-Frambuffer(буфер кадра) – интерфейс для работы с памятью для записи данных.

Наиболее удобный и естественный способ представления данных для потоков в GPU – в виде двумерных матриц. Потому что они отлично вписываются в схему работы GPU с текстурами(2 мерным массивом цвета). Многие задачи отлично вписываются в эту схему(матричная алгебра, обработка изображений, физические симуляции и прочие)

Особенностью работы GPU стоит так же отметить контроль выполнения или (flow control). Так как в GPU нет сложных механизмов вроде branch prediction в CPU – можно столкнуться с проблемой, когда часть потоков выполнит проход по одной ветви, и только лишь в следующей очереди по второй ветви. Это естественно может сказаться на производительности. Еще сложнее обстоит дело в случае Tiled-архитектур, встречающейся в основном в мобильных GPU.

**Доступные GPU методы:**

**Map –** метод применяетфункцию нанабор элементов в потоке.Например, умножение элемента потока на константу(повысить яркость изображения)

**Reduce –** или свертка - метод применятся на подмножестве потока(вплоть до единичного элемента)

Фильтрация потока(удаления части элементов)

**Scan –** метод так же называется параллельное префиксное суммирование, берет вектор(поток) данных, определенную операцию, например (+) и индекс элемента. В случае если данные на входе [a0,a1,a2,a3] exclusive scan производит вывод [i, a0, a0+a1, a0+a1+a2, …] inclusive scan – [a0,a0+a1,a0+a1+a2 ..]

**Scatter (рассеивание) –**Он позволяет изменить положение данных в сетке. **Gather** – обратная операция(возвращает )

**Sort(сортировка)**-сортирует набор элементов. Наиболее общая реализация на GPU (radix sort, сортировка слиянием)

**Search(поиск) –** позволяет программисту найти нужный элемент внутри потока, также возможно его соседей. Использует параллельный поиск по таблице. Чаще всего используется двоичный поиск на отсортированных элементах.

**Структуры данных:**

Сплошные массивы

Разреженные матрицы

Адаптивные(union подобные) структуры

Какие есть варианты для исполнения кода на GPU:

**CUDA** – закрытая платформа от NVIDIA, работает эксклюзивно на их GPU, с использованием специального драйвера; Компания выпускает помимо обычных пользовательских видеокарт – специализированные (tesla, nvidia-quad, а также серверные решения (nvidia dgx-1). В данный момент это доминирующая технология. Платформа предоставляет небольшой набор расширений для языка программирования, который включает прямую реализацию параллельных алгоритмов. Поддерживает гетерогенные системы, в которых приложение использует CPU и GPU. Последовательные части используют СPU, а параллельные загружена в GPU. Таким образом параллелизм может быть применен к уже существующим приложениям. CPU и GPU имеют разные области памяти и воспринимаются как различные устройства. GPU могут иметь сотни ядер и поддерживать тысячи потоков. Ядра имеют общие ресурсы включая файл регистров и разделяемую память, что позволяет избегать отправки данных по системной шине для обмена данными. Помимо платформы, NVIDIA предоставляет расширения ввиде библиотек, вроде сuFFT(для быстрых преобразований Фурье на GPU), MAGMA(библиотека для вычислений линейной алгебры), cuRAND(генераторы случайных чисел) и прочие. Небольшой пример:

|  |
| --- |
| void sortfile(FILE \*fp, int N) {  char \*data, \*d\_data;  data = (char\*) malloc(N);  cudaMalloc (&d\_data, N);  fread(data, 1, N, fp);  cudaMemcpy(d\_data,data,N,H2D);  qsort<<<...>>>(d\_data,N,1,compare);  cudaMemcpy(data,d\_data,N,D2H);  use\_data(data);  cudaFree(d\_data);  free(data);  } |

**Khronos OpenCL** – открытый стандарт разработанный консорциумом Khronos Group (Amd,Intel и другие), они так же занимались разработкой стандартов OpenGL/Vulkan, OpenVX, SPIR, WebGL и прочих. OpenCL был разработан не только для гетерогенных систем с GPU, но и другими возможными устройствами(DSP,FPGA). Это позволяет легко переносить код на GPU, сопроцессоры и иные системы. Иногда программисты сталкиваются с ситуациями, когда код с целевой платформой в виде данного процессора под OpenCL выполняется быстрее, чем просто на CPU. Данный API поддерживается наибольшим количеством платформ, но все же далек по популярности от CUDA ввиду довольно сложной модели программирования. В ней вычислительная система представлена в виде набора вычислительных устройств, включая CPU и другие акселераторы присоединенные к основному процессору. Функции выполняемые на акселераторах называются Kernels, устройство может иметь несколько *обрабатываемых элементов*(PE). Порядок выполнения функций на этих элементах зависит от драйвера устройства предоставляемый с устройством. Память разделена, и имеет следующую иерархию:

* Глобальная(доступная всем PE) – но имеющая высокую задержку
* Доступная для чтения – небольшая, с низкой задержкой, доступная для записи только управляющим процессором
* Локальная память – доступная внутри группы PE
* Регистровая память, доступная одному элементу.

|  |
| --- |
| \_\_kernel void matvec(\_\_global **const** float \*A, \_\_global **const** float \*x,  uint ncols, \_\_global float \*y)  {  size\_t i = get\_global\_id(0); *// Global id, used as index*  \_\_global float **const** \*a = &A[i\*ncols]; *// Pointer to the i'th row.*  float sum = 0.f; *// Accumulator*  **for** (size\_t j = 0; j < ncols; j++) {  sum += a[j] \* x[j];  }  y[i] = sum;  } |

**Compute Shaders** – доступны из различных API вроде, DirectCompute (Windows), Metal (MacOS), C++ AMP и прочих.

Используя обычные **шейдеры графических API** с DirectX/OpenGL/Metal и других. В качестве input буфера можно использовать 1D,2D,3D - текстуры с данными, VBA,VBO. В качестве output буфера framebuffer, или же рендер-текстуру. Так же возможно использование аппаратной интерполяции.

Стоит отметить что Vulkan API(новый стандарт OpenGL) объединил представление графических шейдеров и вычислительных функций(kernels) в машинно-независимом промежуточном языке SPIR-V.

**Аппаратные особенности**

Ниже я собрал простую таблицу ключевых аппаратных особенностей

|  |  |
| --- | --- |
| **CPU** | **GPU** |
| Latency Intolerance | Latency Tolerance |
| Task Parallelism(Параллелизм на уровне задач) | Data Parallelism(Параллелизм данных) |
| Сложные многопоточные ядра | SIMD (Single Instruction Multiple Data) ядра |
| 10+ потоков | 10000+ потоков |

CPU – процессор с **низкой задержкой** и **низкой пропускной способностью**. Создан, с целью минимизации задержки( например, обработка ввода), кэш – используется для минимизации, и CPU использует максимально сложные инструменты для максимального использования кэша (Предзагрузка инструкций, исполнение вне порядка, контроль исполнения и так далее). CPU нужен большой кэш.

GPU – процессор с **высокой задержкой**, **высокой пропускной способностью**. GPU были разработаны для задач, которым задержка не критична, соответственно, чтобы быть эффективным, GPU должен использовать большие объемы данных. Кэш ускорит работу, но не столь критичен; Место может быть использовано для большего числа простых ядер.  
  


GPU – обеспечивает полностью независимое исполнение каждого потока, что освобождает от необходимости синхронизации. SIMD потоки – минимизируют необходимость в управлении потоками, а также уменьшают дополнительные затраты на планирование, кэширование и прочие атрибуты CPU. Бонусом ко всему выше сказанному является легкая масштабируемость пропускной способности GPU, просто добавьте больше ядер;

**Параллелизм в CPU / GPU**

|  |  |
| --- | --- |
| CPU параллелизм на уровне задач | GPU дата-параллелизм |
| Множество задач соответствуют множеству потоков | Модель SIMD |
| Задачи исполняют разные инструкции | Одна инструкция на разных данных |
| Десятки относительно тяжелых потоков исполняются на десятках ядер | Десятки тысяч легковесных потоков на сотнях ядер |
| Каждый поток явно управляется и планируется | Потоки управляются и планирутся аппаратно |
| Каждый поток должен быть индивидуально запрограммирован | Программируются наборы потоков(batches) |

Ниже я коротко опишу современную архитектуру GPU (Nvidia Kepler):



SM(Streaming multiprocessor) (192 single precision операции за такт, 64 double precision операции за такт). Частота процессора не такая высокая как в CPU, однако огромное число ядер позволяет достичь легко достичь мощности вычислений в несколько TFlops.

Файл регистров(256KB+), Разделяемая память(16-48KB), L1 кэш (16-48KB), Read-only кэш (48KB), Кэш констант (8KB)

Кэш многоуровневый в сумме, до 4мб.

Достаточно большой файл регистров позволяет уменьшить время задержки при переключении контекста. В GPU Pascal файл регистров составляет 14336KB. На CPU файл регистров обычно около сотни килобайт;

Модель исполнения включает **Поток –** как единицу последовательного исполнения кода. Все потоки исполняют одну программу параллельно;

**Блок потоков –** группа*потоков*, которые могут взаимодействовать с использованием легковесной синхронизации, имеют общий кэш;

**Сетка(Grid) –** группа блоков потоков. Не синхронизируются между собой.

Поток исполняется на ядре. Блок потоков на SM. Блоки не мигрируют. Несколько конкурентных блоков потоков могут ожидать исполнения на SM ядре, однако их число сильно ограничено файлом регистров и объемом разделяемой памяти;

Потоки организованы в группу по 32 потока называемые “warps”. Все треды внутри warp исполняют одну и ту же инструкцию одновременно.

Особенность исполнение кода в группах потоков стоит принимать во внимания в случаях использования ветвлений. Так при попадании части потоков из warp в разные ветви исполнения вынуждает блокировать половину результатов и выполнять проход по второму кругу.

**Поддержка на уровне ОС**

Если рассматривать ядро операционных систем, то, к сожалению, GPU и параллельные алгоритмы не используются, только в каких-то специализированных ОС. Понятное, что ОС создана для того чтобы быть максимально гибкой и решать наибольший круг проблем; многие задачи, могут быть фундаментально непараллелизируемыми, долгий случайный доступ к памяти, а также проблема исполнения ветвлений, отсутствие прерываний и исключений, делает GPU малопригодным в качестве вычислительной основы для ОС. Однако есть поле возможных системных задач, которые могли бы быть более эффективно решены в гетерогенной системе с GPU, например, обработка сетевых сообщений, когда можно выполнять обработку без сохранения состояния, криптографические задачи, однако все чаще для этого применяются специализированные аппаратные средства. Где возможно использование параллелизма ОС используют SSE, AES-NI или выделенные аппаратные средства (для шифрования, raid-вычислений, подсчет checksum сетевых пакетов и прочие).

Недавние проекты: программный роутер PacketShader [<http://shader.kaist.edu/packetshader/>] и поддержка зашифрованных соединений SSLShader [http://shader.kaist.edu/sslshader/], демонстрируют возможность решения системных задач на GPU. Предпринимались попытки разработки ОС с поддержкой гетерогенных систем (HELiOS). Barrelfish[http://www.barrelfish.org/], например, трактует многопроцессорную систему, как распределенную cеть, с независимыми микроядрами, общение между которыми устанавливается через систему сообщений. Однако, обе системы опираются на традиционные требования ОС, наличие прерываний, виртуальной памяти, контролируемого переключения контекста и прочих; GPU не поддерживают такие возможности, и больше подходят на роль сопроцессора; Однако если ОС будет воспринимать GPU как сопроцессор с другим набором инструкций, и можно будет таким образом сделать GPU “чуть ближе”, немного уменьшив задержку в передачи данных на стороне CPU, добавив возможность держать потоки GPU готовыми к исполнению кода всегда, то можно будет утверждать о реальном расширении/улучшении возможностей ОС с помощью GPU. **Обработка сетевых пакетов** при максимальной загрузке с использованием PacketShader на видеокарте NVIDIA GTX 480 дает прирост в 3,5-4 раза по сравнению с СPU. SSLShader – дает прирост около 4х раз. Однако оба страдают от относительно большой задержки по отправке данных между памятью CPU<->GPU. Применение AES **шифрования** на GPU дает прирост около 6 раз, по сравнению с тем же процессором. **Поиск(pattern match)**, по некоторым исследованиям дает прирост до 48 раз. Может быть легко применимо в поиске и обнаружении вредоносного кода, фаерволах, поиске по содержимому в файловой системе. Некогда экспериментальная операционная система Singularity OS, использовавшая принцип исполнения только безопасного кода с изоляцией процессов, нежели защитой памяти столкнулся с проблемой сложности анализа кода. Однако позже, разработчик **EigenCFA, анализатора потока исполнения** программ [http://dl.acm.org/citation.cfm?id=1926445] установил практически 72ух кратное ускорение работы при использовании GPU для анализа. Так возможно ускорение и простых базовых алгоритмов, сортировки, поиска, анализ графа. Последний может быть использован в сборщике мусора ОС (GC). Давно известно, что сборщик мусора – тонкое место многих managed(управляемых) языков программирования(С#..).   
  
**Потенциальное ядро ОС** с поддержкой GPU в качестве сопроцессора, должна решить 2 основных проблемы. Это большие временные затраты на **копирование данных между** памятью CPU<->GPU, чтобы дать возможность еще шире использовать GPU, для чувствительных к задержке задач. Если рассматривать СUDA, то существует 2 модели памяти: страничная память (выделяется через malloc()) и прикрепленная(pinned) память, которая выделяется на драйвере устройства. Последняя значительно быстрее. NVIDIA, представила новую систему связи CPU<->GPU **NVLINK,** которая работает в 5-12 раз быстрее чем PCIe. Что значительно ускоряет копирование данных, и тем самым устраняет основной недостаток GPU. Однако они пока доступны только для специализированных компьютеров.

Вторая проблема — это большие временные затраты на **запуск ядра(кода)** на GPU. Именно старт ядра отнимает время, ОС придется ожидать. В рамках всего процесса, это конечно небольшое время, но в случае с постоянным переключением задачи в ОС, хорошо было бы от него избавиться. Ребята из университета UTAH придумали надстройку NSK(Non-Stop Kernel). Технически это небольшой процесс, который запускается один раз и не прекращает свое исполнение. Общение CPU-GPU проходит через систему сообщений, который позволяет перенаправлять сообщения, пока ядро(код) на GPU продолжает исполнение. С традиционным API CUDA это невозможно. NSK ускоряет запуск задач приблизительно в 1.3 раза. ~18.3 мс на 2048 потоков. К сожалению, нет возможности изменять число запущенных процессов во время исполнения.

**Заключение**

Использование GPU в качестве сопроцессора для расширения возможностей ОС, возможно и как выше было доказано, и имеет реальный практический смысл. Многие современные технологии, вроде автономных автомобилей используют специализированные гетерогенные системы с GPU в качестве основного вычислительного устройства, серверные решения со стойками полными видеокарт, для задач машинного обучения ИИ, эта тенденция (глубокое внедрения gpu в вычислительный процесс) имеет устойчивый характер.

**Источники**

<https://github.com/RadeonOpenCompute/hcc>

<https://en.wikipedia.org/wiki/C%2B%2B_AMP>

<https://en.wikipedia.org/wiki/Stream_processing>

CLWrap: Nuisance-Free Control of your GPU - John Colvin | DConf2016

<http://docs.nvidia.com/cuda>

<https://www.khronos.org/opencl/>

[GPGPU and Accelerator Architecture Trends | Nikolai Sakharnykh, NVIDIA](https://www.youtube.com/watch?v=QpcH4cvcm8I)

<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/helios.pdf>

<http://www.cs.utah.edu/~wbsun/kgpu.pdf>

<http://www.ndsl.kaist.edu/~kyoungsoo/papers/packetshader.pdf>