Data Oriented Design

Alexey Boreskov

About Data Oriented Design

- Пришел из программирования игр на консолях
- Специфика консолей – жесткие аппаратные требования
- Фокус – написание высокоэффективного кода с учетом специфики аппаратной платформы
- Сейчас очень широко используется OOP, но у него есть серьезные недостатки с точки зрения быстродействия
- Строим код вокруг данных, главное – данные
- Мне лично ряд мест напомнил CUDA

Специфика hardware на данный момент

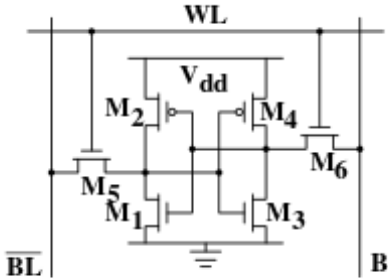
- Большинство алгоритмов были сформулированы более 40 лет назад
- То же самое относится и к основным концепциям и подходам OOP
- В то время частоты процессора не сильно отличались от частоты, с которой работала память.
- Если взять за 1 соотношение между скоростью CPU и скоростью доступа к памяти в то время, то сейчас это соотношение **более 100**

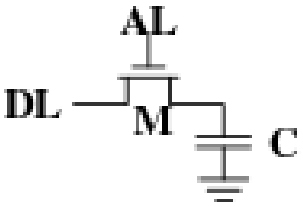
Специфика на данный момент

- Доступ к внешней (off-chip) памяти может стоить очень дорого
- Работа с памятью должна быть *cache-friendly*
- Речь идет о последовательном доступе элементам данных, лежащих в одной большой области памяти
- Идеальная структура данных `std::vector`
- Очень жесткие требования по работе с памятью с самого начала были в CUDA, если посмотреть, то они сильно похожи на то, что мы имеем сейчас
- Строим дизайн именно вокруг данных и доступа к ним – как это сделать *cache-friendly*

Почему возникла такая ситуация с памятью и как устроена память

На самом деле есть два основных типа памяти – статическая и динамическая (SRAM и DRAM)





Статическая (слева) – очень быстрая, быстрый произвольный доступ, но требует постоянного питания и 6 транзисторов

Динамическая(справа) – просто конденсатор. Очень дешевая, но требует refresh (сейчас уже не требует, сделано прямо в кристалле)

Почему возникла такая ситуация с памятью и как устроена память

- Поэтому обычная память это именно DRAM. Она дешевая, ее можно много разместить на кристалле
  - SDRAM – Synchronous DRAM
- Для адресации адрес фактически состоит из нескольких частей, обычно адрес строки (row) и адрес столбца. Они передаются по очереди и это занимает время.
- Но при чтении последовательных данных, можно сэкономить для CAS, чтение идет быстрее.
- Чтение идет блоками, обычно 64 бита
- Что значит DDR4-1600 (или PC4-12800)
  - 1600 – transfers per second
  - 12800 - peak transfer rate (Mb/sec)
  - Это просто скорость в байтах/секунду, она растет.
  - Латентность не растет (memory clock 200MHz, bus clock 800MHz)

Caches

- Чем дальше от ядра – тем медленнее. Основная память удалена, доступ идет через FSB, поэтому она медленная.
- В самом CPU размещаются кэши – размещенная прямо на кристалле быстрая SRAM память, кэширующая доступ к медленной DRAM. К ним доступ через BSB.
  - За счет того, что прямо в кристалле – не нужен FSB и т.п.
- Несколько уровней кэшей
  - L1i/L1d – самая маленькая и самая быстрая
  - L2 – медленнее, но объем больше (256К)
  - L3 – еще медленнее, но объем еще больше (2-8Мб)
- L1 – раздельно для кода и данных, кэшируются виртуальные адреса
- L2,L3 – кэширование физических адресов

Caches

Каждое ядро имеет свои L1i/L1d, свой L2.

L3 обычно общий.

MESI – протокол синхронизации кэшей отдельных ядер – цель – все ядро должны видеть один и тот же образ памяти (как она видна через кэши). Могут быть и другие протоколы, но цель одна.

Пример. Есть массив `int`. Нужно найти число 7 используя несколько ядер.

```
size_t count;          // общий счетчик
size_t count [4];      // свой для каждой нити
```

Что будет происходить с доступом к счетчикам ???

Caches

- https://habrahabr.ru/post/346250/
- Трехголовый процессор для Xbox 360, общий кэш L2 1Мб
  - 1Мб мало для трех ядер
  - Сделали запись минуя L2 сразу в L1
  - В какой-то момент ядра начинают видеть память (через свои кэши) по-разному
  - Но в дампе памяти при креше все отлично

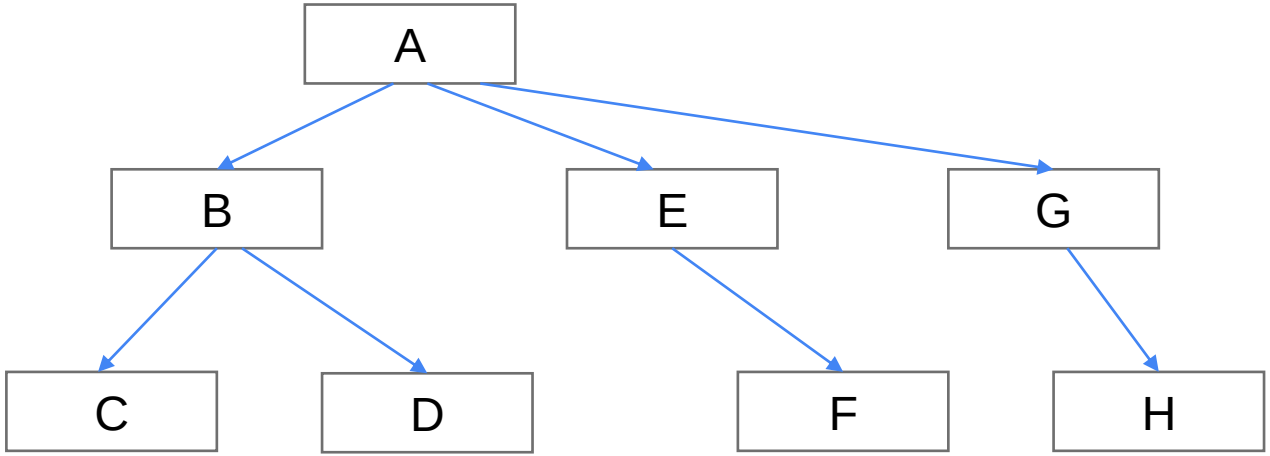
CUDA, OpenGL shaders

- Запись в DRAM одной нитью на GPU не сразу становится видимой другим выполняемым нитям
- Нужна специальная синхронизация – fence
- Причина – оптимизация и группирование записи в память

OOP

- Традиционно авторы по DOD ругают OOP
- Возможно некоторая неприятная специфика Xbox 360/PS3
- ООП Головного Мозга
  - Многие через это проходят, включая меня
  - Выглядит очень здорово и красиво
  - Часто вообще не учитывает performance
- OOP отлично работает, просто нужно следить за критическими местами
  - Иначе очень красивый дизайн и совсем не красивое быстродействие

Типичный доступ к данным в OOP



Как все это с точки зрения кэша ???

Типичный доступ к данным в ООР

- В результате имеем некогерентный паттерн доступа, который практически не использует кэш
- Много косвенного доступа – `a->b->c->d`
- Как результат такого доступа – работа с памятью идет очень медленно, постоянные cache miss
- Типичная организация данных создана без учета кэша – без учета того, к каким данным будет осуществляться одновременный доступ

Data Oriented Design

Рассмотрим простейший пример – нужно сделать map, переводящий адреса в какие-то данные (например, bbox)

Классический вариант

```
struct {
      void * key;
      bbox    data;
} table [N];
```

В чем проблема – мы будем перебирать `key`, но значения `data` также будут попадать в кэш, снижая его эффективность

Data Oriented Design

Предлагаемый вариант -

```
struct {
        void * key;
} keys [N];
struct {
        bbox            data;
} datas [N];
```

В случае перебора ключей эффективность использование кэша сильно вырастает.

Мы сперва находим ключ в массиве ключей, а потом – соответствующие ему данные

Data Oriented Design

Давайте идти от данных – как нам надо организовать данные для максимально быстрой работы с ними ?

- Большие массивы однородных элементов
  - Вместо сложных графов
- Структурируем по доступу к данным – те элементы, к которым обычно обращаются вместе, должны лежать вместе
- Вместо графа объектов получаем просто набор массивов, которые мы последовательно обходим
- Используем *prefetch* – при такой организации данных это довольно просто
- *Structure of Arrays* vs *Arrays of Structures*
- Выравнивание, оптимальный порядок размещения данных (bool внутри группы float)

Data Oriented Design

Пример такой организации данных, вместо

```
class X {
      A a;
      B b;
      C c;
      D d;
```

…

Мы группируем и разбиваем исходя из доступа, т.е. Исходный класс разбивается на несколько простых структур, каждая из которых лежит в своем массиве.

Критерий для разбиения – совместный доступ к данным – что из этого нам одновременно нужно в кэше

CPU Pipeline

- Есть и другая сторона – это сам CPU. Сейчас CPU сильно конвейеризован – выполнение команды разбивается на много шагов (десятки), которые идут по конвейеру.
- Все это хорошо, пока мы имеем дело с последовательным кодом и мы знаем, какие команды будут далее.
- Условные переходы могут это легко разрушить – если предсказание перехода ошиблось, то конвейер сбрасывается и мы теряем быстродействие.
- С этой точки зрения виртуальные функции – это двойной доступ к памяти (потенциальный cache miss) и переход по адресу из памяти, т.е. сброс конвейера

CPU Pipeline

- Реальный пример разработки под PS3

```
if ( dirty ) // проверяем, нужно ли
                        // пересчитать
  recomputeBBox ();
```

- Попытка оптимизировать расчеты
- На самом деле пропуск расчета на PS3 обходится дороже, чем сам расчет из-за сброса конвейера
- Есть предсказатель переходов, обычно просто статистика
- Ошибка предсказателя стоит сброса конвейера

CPU Pipeline

Как к этому подходит DOD –

- Значения разных типов (классов) просто хранятся в разных массивах
  - Никаких виртуальных функций – просто разные операции применяются к разным массивам
- Аналогичный подход к условным конструкциям – просто разбиваем на несколько массивов в зависимости от условий и используем разные функции обработки для каждого из них
  - В крайнем случае просто сортируем по условиям
- Curiously Recurring Template Pattern (CRTP)
- Boost::likely/unlikely

```
if ( likely ( condition ) ) statement;
if ( unlikely ( condition ) ) statement;
```

CPU Pipeline

Интересный пример: есть массив рандомных данных, суммируем числа, удовлетворяющие критерию:

```
for (unsigned i = 0; i < 100000; ++i){          // Primary loop
     for (unsigned c = 0; c < arraySize; ++c)
          if (data[c] >= 128)
               sum += data[c];
```

Интересное наблюдение – если данные отсортировать, то скорость возрастает с 11.54 до 1.93 сек.

- Для не отсортированных данных предсказать перехода для условного оператора работает крайне плохо (он основан на статистике)
- Для отсортированных данных он работает фактически со 100% точностью, т.е. выигрыш идет именно за счет `if`

Вызовы функций

- inline
  - Никакого вызова и передачи параметров, самый дешевый способ
- Обычный вызов
  - Передача параметров (иногда их очень много)
  - Сам вызов
- Виртуальная функция
  - Читаем адрес VMT
  - Читаем из нее адрес метода
  - Делаем вызов по адресу с передачей параметров
  - Кэши сильно помогают, но все равно дорого
- CRTP – заменяем вызов виртуальной функции на inline

Where there is one, there are many

Часто мы пишем код для выполнения какой-либо проверки над одним объектом.

Однако по факту обычно эту проверку нужно выполнять для целой группы объектов, зачастую довольно большой.

Использование факта проверки сразу целой группы позволяет в ряде случаев писать более эффективный код.

The general case is MANY.

If you are going to write general code, then write it for the statistically common case.

What Every Programmer Should Know About Memory

Очень правильный рассказ о памяти, кэше и всем, что с этим связано.

http://futuretech.blinkenlights.nl/misc/cpumemory.pdf

Рекомендую ☺

Еще злобный разбор кода на C++

https://macton.smugmug.com/Other/2008-07-15-by-Eye-Fi/n-xmKDH/i-rns75wt

Спасибо всем, внесшим свой вклад