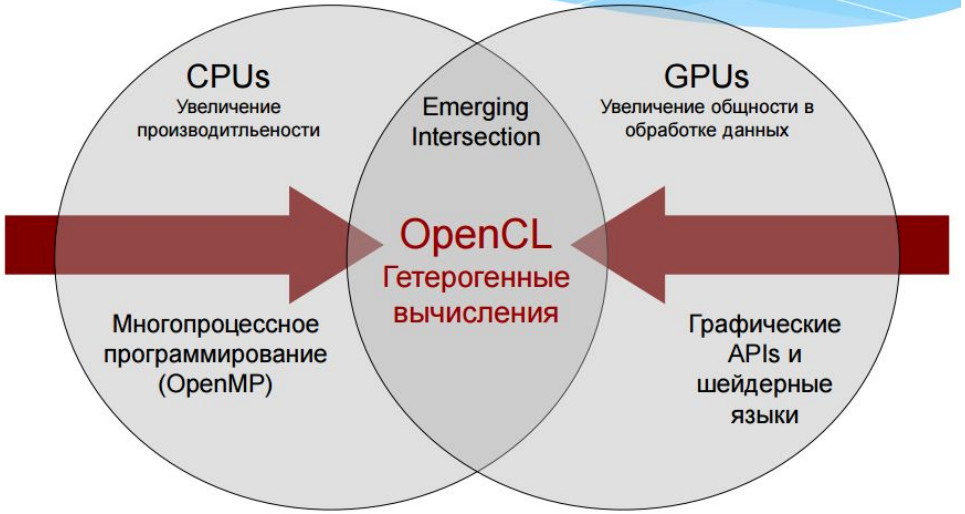


Параллельное и распределенное программирование.

Технология программирования гетерогенных систем OpenCL.

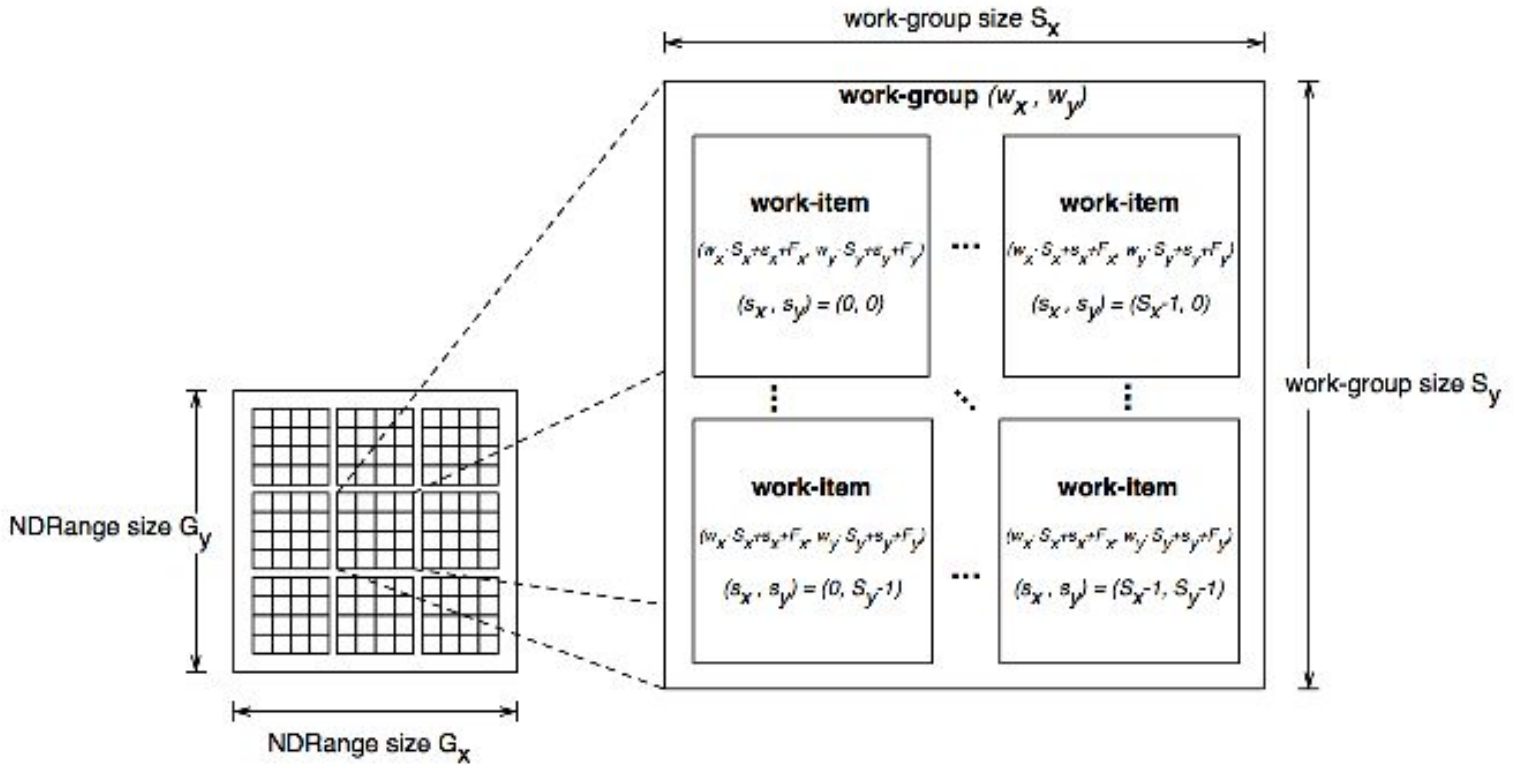
Лекция 4

Модель потоков GPU

План

- Wavefronts и warps
- Планирование потоков на графических процессорах
- Синхронизация

Модель исполнения. Индексное пространство

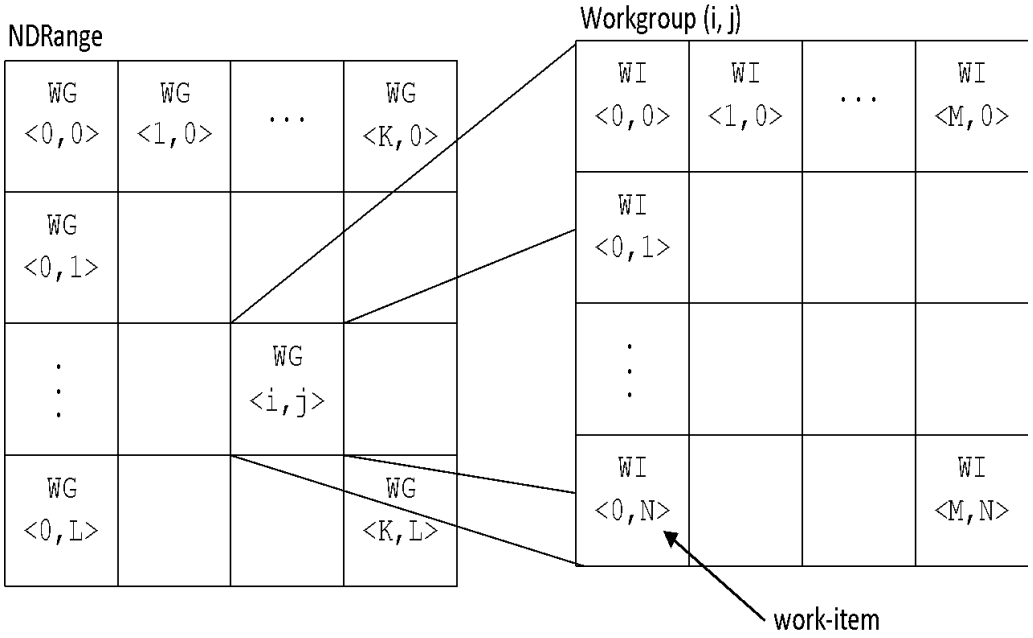


Gx , Gy – глобальные размеры;
Sx, Sy – локальные размеры рабочей группы;
Fx, Fy – глобальное смещение рабочей группы;

gx , gy – глобальный идентификатор;
sx, sy – локальный идентификатор.

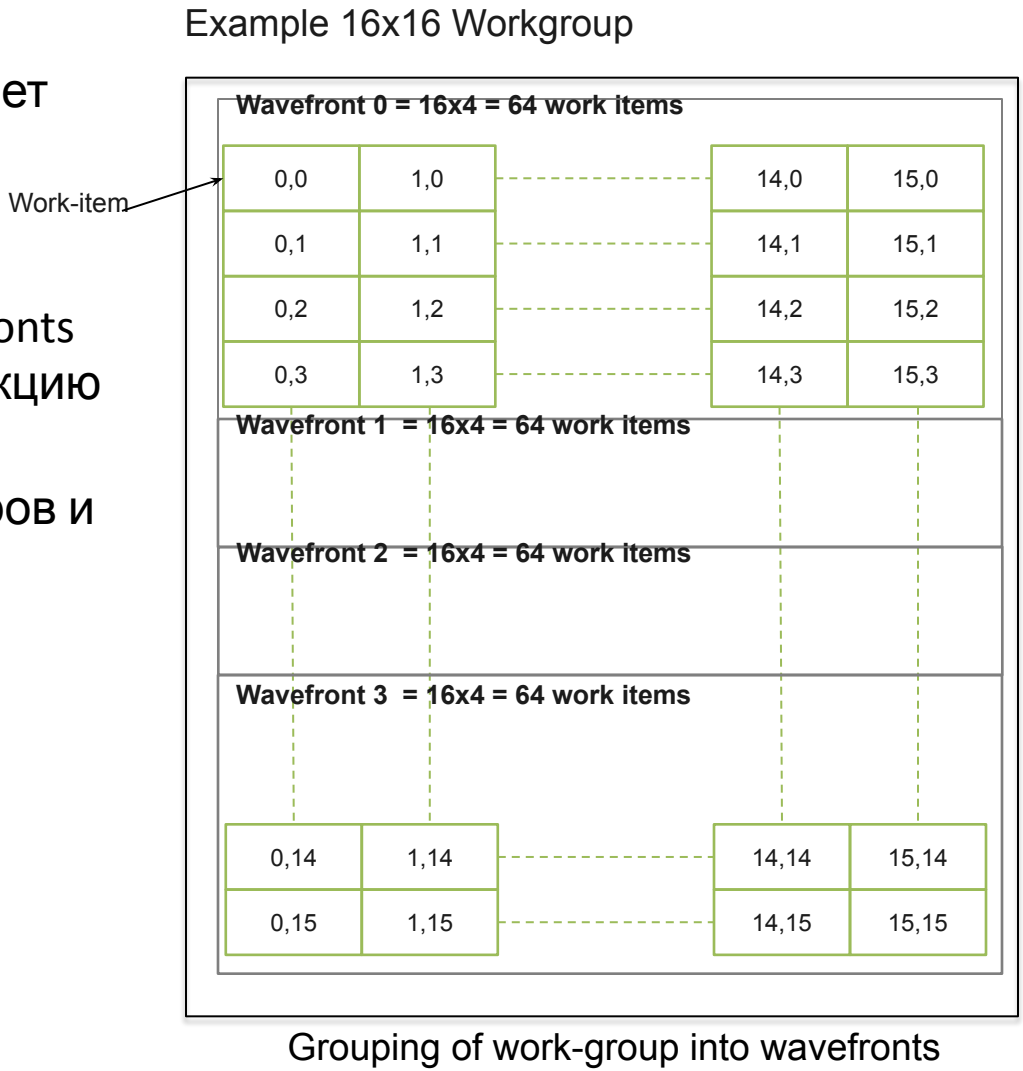
Work Groups относительно HW Threads

- Ядра OpenCL структурированы в рабочие группы, которые сопоставляются с вычислительными единицами устройства
- Вычисление единиц на графических процессорах состоит из элементов обработки SIMT
- Рабочие группы автоматически разбиваются на аппаратные планируемые группы потоков для оборудования SIMT
  - Эта «планируемая группа» известна как a wavefront (AMD) или a warp (NVIDIA)



Work-Item планирование

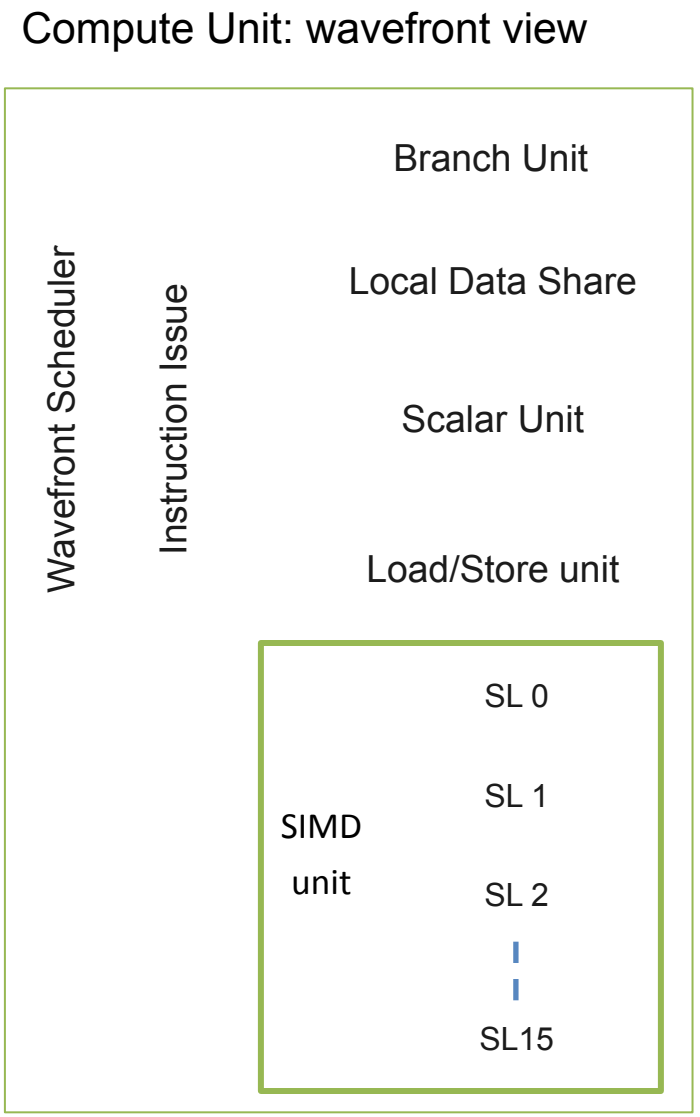
- Аппаратное обеспечение создает wavefronts, группируя рабочие элементы рабочей группы
  - Сначала по размеру X
- Все рабочие элементы в wavefronts выполняют одну и ту же инструкцию
- Рабочие элементы имеют собственное состояние регистров и могут свободно выполнять ветвления



Grouping of work-group into wavefronts

Wavefront планирование

- Размер Wavefront - 64 рабочих элемента
  - Векторные инструкции выполняются каждым рабочим элементом
  - Скалярные инструкции выполняются один раз для полного wavefront
  - Инструкции по векторной загрузке / хранению содержат один адрес для каждой рабочей единицы
- SIMD-полоса (SL) выполняет одну векторную инструкцию
  - 16 потоковых ядер выполняют 16 векторных инструкций на каждом цикле



Wavefront планирование

- В случае небезопасного чтения-записи (RAW) один wavefront остановится на четыре дополнительных цикла
  - Если доступен другой wavefront, то латентность скрывается
- Два волновых фронта (128 рабочих элементов) полностью скрывают задержку RAW
  - Первый wavefront выполняется четыре цикла
  - Еще один wavefront запланирован на следующие четыре цикла
  - Затем запускается первый wavefront снова
- При работе с глобальной памятью латентность намного больше
  - За это время вычислительный блок может обрабатывать другие независимые wavefronts

GPU загруженность

- Локальная память и регистры сохраняются в вычислительном блоке после того, как запланирована рабочая группа
  - Традиционное переключение контекста не используется
- Количество активных wavefronts, поддерживаемых на единицу измерения, ограничено (локальной памятью, регистрами)
- Количество активных wavefronts, возможных на вычислительной единице, может быть выражено с использованием показателя, называемого заполнением
- Увеличенное количество активных wavefronts позволяет лучше скрывать задержку

Поток управления

- Хотя рабочие элементы имеют уникальные счетчики программ, на практике они выполняются на SIMD-оборудовании
- Рабочие элементы могут выполнять другую ветку программы относительно других рабочих элементов wavefront, используя маскирование или предикацию
- На практике разветвление должно быть ограничено гранулярностью wavefront, когда это возможно, для полного использования блоков выполнения SIMD

Контроль потоков

- Как обрабатываются рабочие элементы с разными условиями выполнения, когда одна и та же команда выдается для всех рабочих элементов в wavefront?
- Предикация - это метод смягчения затрат, связанных с условными ветвями
  - Эффективно в случае коротких разделов кода
  - Компиляторы могут заменить выражения «switch» или «if then else», используя предикацию ветвления

Предикация для GPUs

- Предикат - это код условия, для которого задано значение true или false на основе условного
- Оба случая условного потока планируются к исполнению
  - Выполнены инструкции с истинным предикатом
  - Инструкции с ложным предикатом не записывают результаты или не читают операнды
- Удаляет ветви
  - Эффективно для коротких условий

```
__kernel void test()
{
    int  tid= get_local_id(0);

    if( tid %2 == 0)
        Do_Some_Work();
    else
        Do_Other_Work();
}
```

Predicate = True for work-items 0,2,4….

Predicate = False for work-items 1,3,5….

Predicates switched for the else condition

Расходящийся поток управления

- Случай 1: все нечетные рабочие элементы будут выполняться, если if, в то время как все четные рабочие элементы выполняют условие else. Блок if и else должен быть выпущен для каждого wavefront
- Случай 2: все рабочие элементы первого wavefront будут выполнять случай if, тогда как другие wavefronts будут выполнять случай else. В этом случае для каждого wavefront выполняется только if или else

Case 1

```
int tid = get_local_id(0)
if ( tid % 2 == 0) // even work-Items
    DoSomeWork()
else  // odd work-Items
    DoSomeWork2()
```

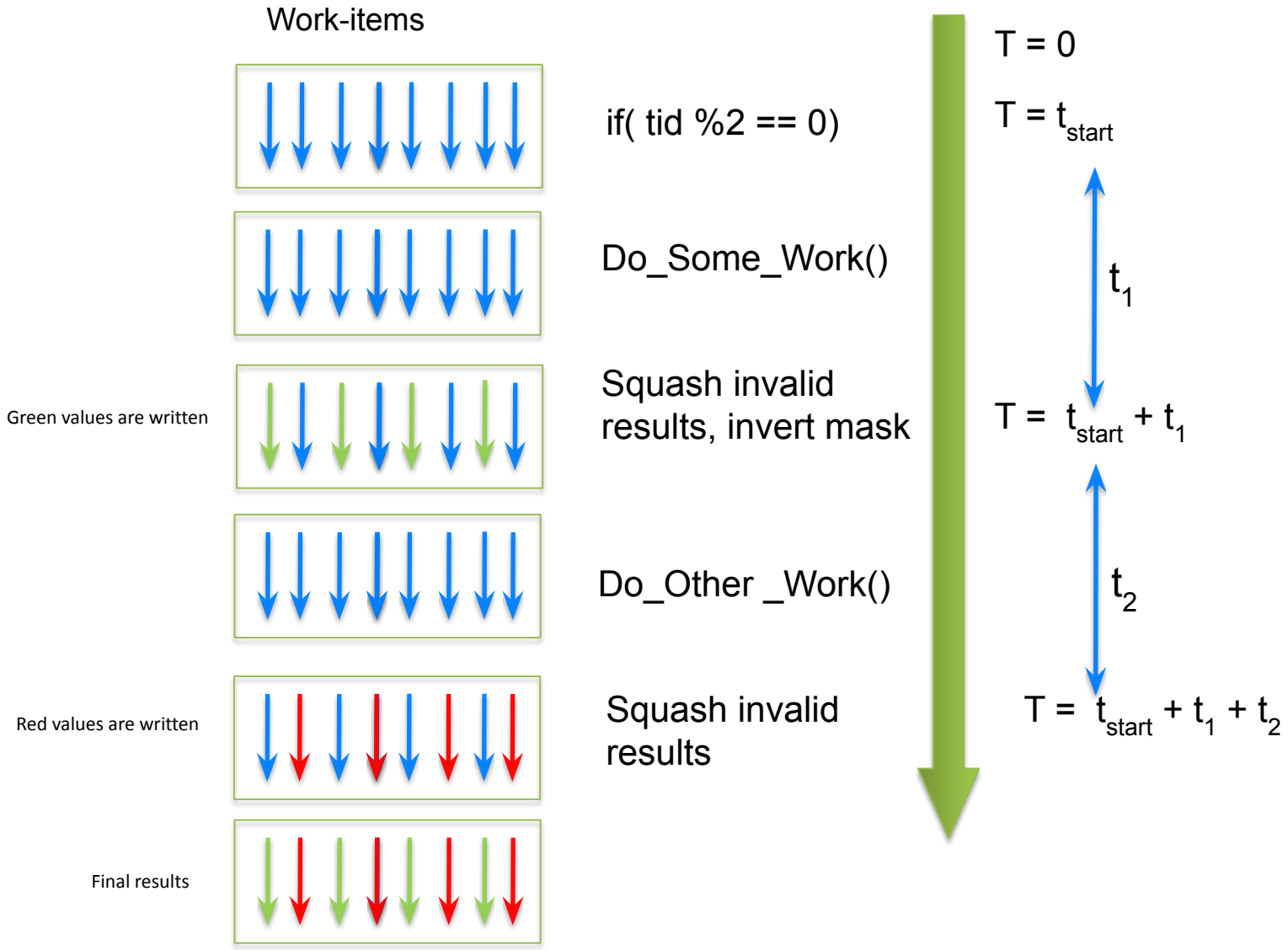
With divergence

Case 2

```
int tid = get_local_id(0)
if ( tid / 64 == 0)  // first wavefront
    DoSomeWork()
else if (tid /64 == 1) // second wavefront
    DoSomeWork2()
```

Without divergence

Влияние предикатов на производительность



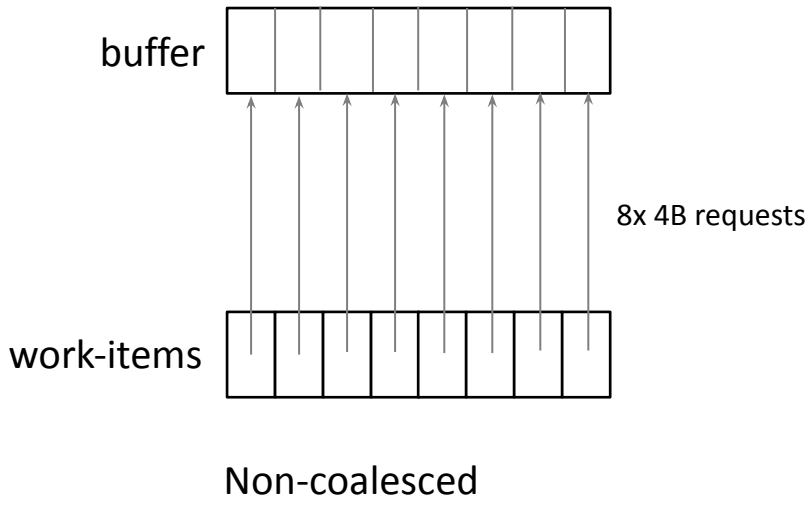
От забора до барьера

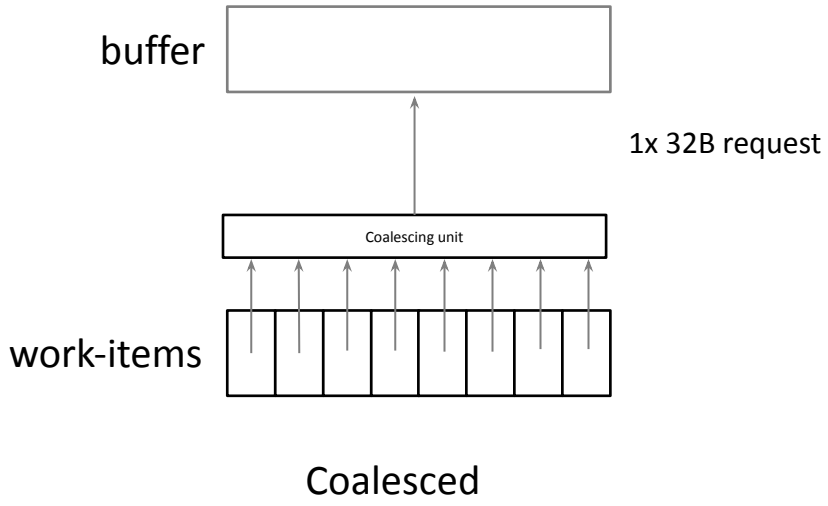
- CL_{LOCAL,GLOBAL}_MEM_FENCE
- Забор
  - mem_fence(cl_mem_fence_flags flags)
  - {read|write}_mem_fence
    (cl_mem_fence_flags flags)
- Барьер
  - barrier(cl_mem_fence_flags flags)

Оптимизация функции ядра

Coalescing при доступе к памяти

- Рабочие элементы обращаются к элементам буфера
  - Один запрос к памяти будет сгенерирован для каждого рабочего элемента
  - Аппаратное обеспечение GPU поддерживает объединение нескольких запросов





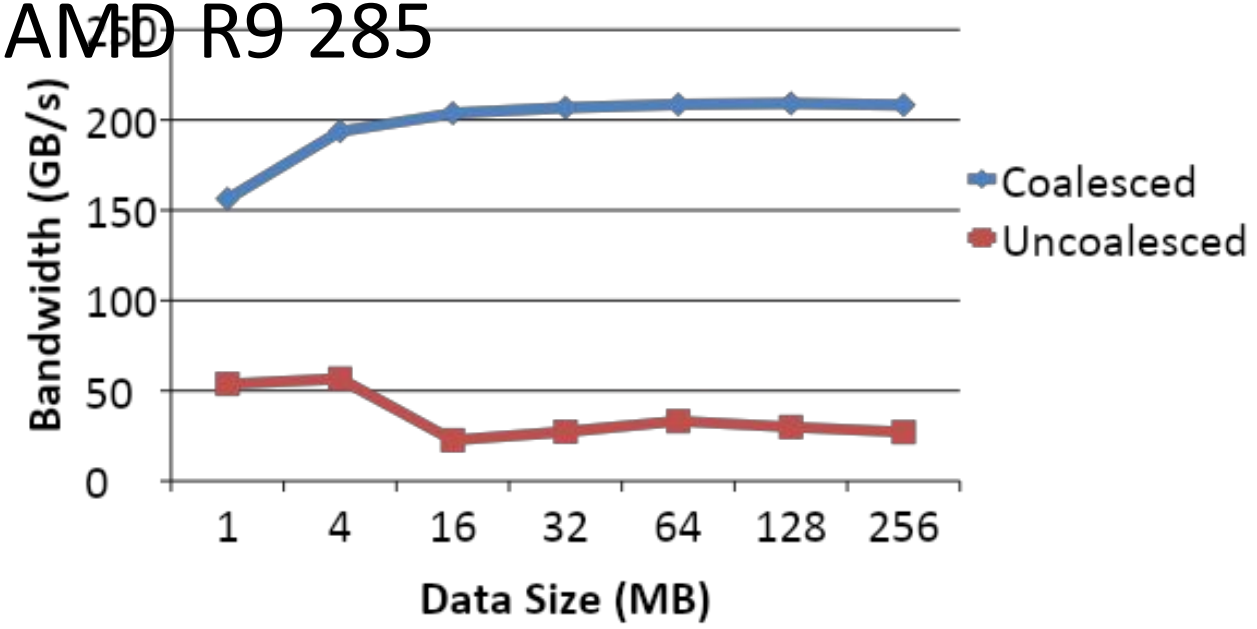
Coalescing при доступе к памяти

- Для аппаратного обеспечения 64 рабочих элемента образуют wavefront и должны выполнять одну и ту же инструкцию SIMD образом

- Для GPU AMD R9 290X доступ к памяти из 16 последовательных рабочих элементов может быть объединен

Coalescing при доступе к памяти

- Производительность глобальной памяти для простого копирования данных ядра полностью объединенного и полностью несовместного доступа на GPU AMD R9 285



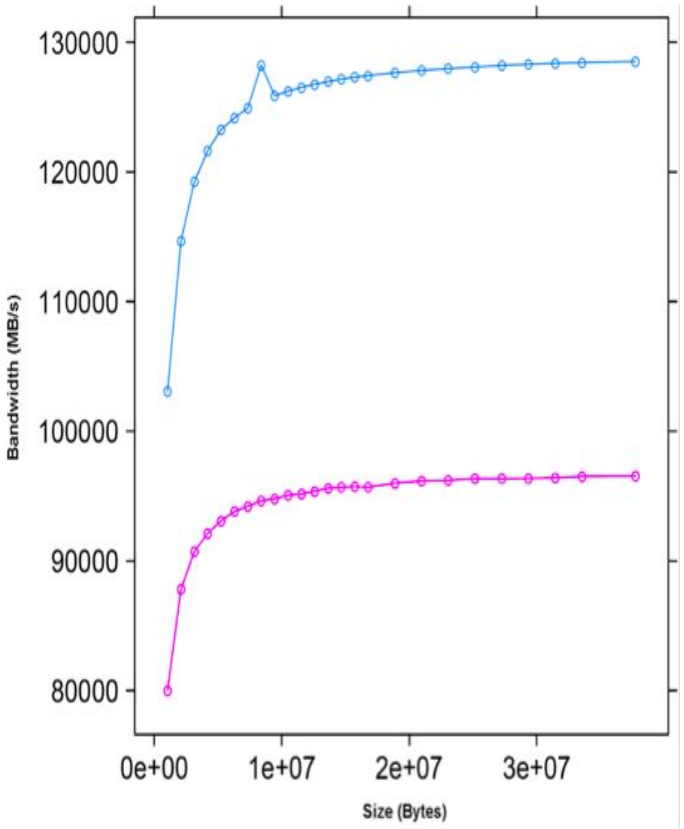
Векторизация

- Векторизация позволяет одному рабочему элементу выполнять сразу несколько операций
- Явная векторизация достигается путем использования векторных типов данных (таких как float4) в программе
  - Когда число добавляется к типу данных, тип данных становится массивом этой длины
  - Операции могут выполняться на векторных типах данных, подобно обычным типам данных
    - Каждый ALU будет работать с элементом данных типа float4

Векторизация

- Векторизация повышает производительность памяти на AMD Northern Islands и Ever

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void
Copy1(__global const float * input,
      __global float * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```



Локальная память

- На графических процессорах локальная память отображает в высокоскоростную память с малой задержкой, расположенную на микросхеме
  - Полезна для обмена данными между рабочими элементами в рабочей группе
  - Доступ к локальной памяти обычно намного быстрее, чем доступ к глобальной памяти (даже к кешированной глобальной памяти)
  - Доступ к локальной памяти обычно не требует объединения
- Компромисс заключается в том, что использование локальной памяти ограничит количество рабочих групп в программе при исполнении

Константная память

- Константная память - это пространство памяти для хранения данных, к которым одновременно обращаются все рабочие элементы
  - Обычно сопоставляет специализированное кэширующее оборудование с фиксированным размером
  - Он НЕ должен использоваться для общих входных данных (например, входного буфера), который доступен только для чтения
- Примеры полезных данных для размещения в постоянной памяти
  - Фильтры свертки, параметры и т. д.

Загруженность

- Рабочие элементы из рабочей группы запускаются вместе на вычислительном устройстве
- Если ресурсов достаточно, несколько рабочих групп могут одновременно сопоставляться с одним и тем же вычислительным блоком
  - Wavefronts из нескольких рабочих групп можно поменять местами, чтобы скрыть латентность
- Ресурсы фиксируются (количество регистров, размер локальной памяти, максимальное количество волновых фронтов)
  - Любое из ограничений ресурсов может ограничить количество рабочих групп на вычислительной единице
- Термин «загруженность» используется для описания того, насколько хорошо используются ресурсы вычислительной единицы

Загруженность: регистры

- Регистры являются одним из основных ограничивающих факторов для больших ядер
- На текущих графических процессорах максимальное количество регистров, требуемых ядром, должно быть доступно для всех рабочих элементов рабочей группы
  - Пример. Рассмотрим графический процессор с 16384 регистрами на единицу расчета, на котором выполняется ядро, для которого требуется 35 регистров на каждый рабочий элемент
  - Каждый вычислительный блок может выполнять не более 468 рабочих элементов
    - Это влияет на выбор размера рабочей группы
      - Рабочая группа 512 не возможна
      - Одновременно допускается только 1 рабочая группа из 256 рабочих элементов, хотя может работать еще 212 рабочих элементов
      - Разрешены 3 рабочие группы из 128 рабочих мест, что позволяет планировать 384 рабочих места и т. Д.

Загруженность: регистры

- Рассмотрим другой пример:
  - Графический процессор имеет 16384 регистра на единицу расчета
  - Размер рабочей группы ядра фиксирован в 256 рабочих единицах
  - Ядро в настоящее время требует 17 регистров на каждый рабочий элемент
- С учетом информации каждая рабочая группа требует регистров 4352
  - Это позволяет использовать 3 активные рабочие группы, если регистры являются единственным ограничивающим фактором
- Если код может быть реструктурирован, чтобы использовать только 16 регистров, тогда возможно 4 активные рабочие группы

Загруженность: локальная память

- Графические процессоры имеют ограниченную локальную память на каждом вычислительном блоке
  - 64 КБ локальной памяти на графических процессорах AMD
- Локальная память ограничивает количество активных рабочих групп на единицу измерения
- В зависимости от ядра данные на рабочую группу могут быть фиксированы независимо от количества рабочих элементов (например, гистограмм) или могут варьироваться в зависимости от количества рабочих элементов (например, умножения матрицы, свертки)

Загруженность: Work-items/work-groups

- У графических процессоров есть аппаратные ограничения на максимальное количество рабочих элементов на рабочую группу
  - OpenCL ограничивает рабочие группы до 256 рабочих элементов
- Графические процессоры AMD имеют ограничения на SIMD для числа wavefronts
  - 40 wavefronts (2560 рабочих элементов) на единицу измерения
  - Для 44 графических процессоров Compute Unit, таких как R9 290X, может быть до 40x44 = 1760 wavefronts, активных на устройстве

Загруженность: ограничивающие факторы

- Минимальным из этих трех факторов является то, что ограничивает активное количество рабочих элементов (или занятости) вычислительной единицы
- Взаимодействие между факторами является сложным
  - Ограничивающий фактор может отношение как к work-item, так и wavefront
  - Изменение размера рабочей группы может повлиять на использование регистров или локальной памяти
  - Небольшое сокращение любого фактора (например, использование регистра) может привести к активной работе другой рабочей группы
- AMD CodeXL отображает эти факторы визуально, позволяя визуализировать компромиссы

Сопоставление потоков

- Сопоставление потоков определяет, какие потоки будут получать доступ к данным
  - Правильные сопоставления могут согласовываться с оборудованием и обеспечивать большие преимущества в производительности
  - Неправильные сопоставления могут быть катастрофическими для производительности
- Анализ шаблона доступа к памяти на многопараллельном графическом процессоре сводится к задаче эффективного отображения потоков на шаблоны доступа к данным алгоритма

Сопоставление потоков

- Используя различные сопоставления, один и тот же поток может быть назначен для доступа к различным элементам данных
  - В приведенных ниже примерах показаны три разных возможных сопоставления потоков с данными (при условии, что идентификатор потока используется для доступа к элементу)

```
int group_size =
get_local_size(0) *
get_local_size(1);

int tid =
get_group_id(1) *
get_num_groups(0) *
group_size +
get_group_id(0) *
group_size +
get_local_id(1) *
get_local_size(0) +
get_local_id(0)
```

Mapping

```
int tid =
get_global_id(1) *
get_global_size(0) +
get_global_id(0);
```

```
int tid =
get_global_id(0) *
get_global_size(1) +
get_global_id(1);
```

Thread IDs

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

*assuming 2x2 groups

Сопоставление потоков

- Рассмотрим последовательный алгоритм умножения матрицы

```
for (i1=0; i1 < M; i1++)
    for (i2=0; i2< N; i2++)
        for (i3=0; i3 < P; i3++)
            C[i1 ][ i2 ] += A[i1 ][ i3 ]*B[i3 ][ i2 ];
```

- Этот алгоритм подходит для декомпозиции выходных данных
  - Мы создадим потоки NM
  - Каждый поток будет выполнять вычисления P
- Индексное пространство должно быть MxN или NxM?

Сопоставление потоков

- Отображение нитей 1: с пространством индексов MxN :

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

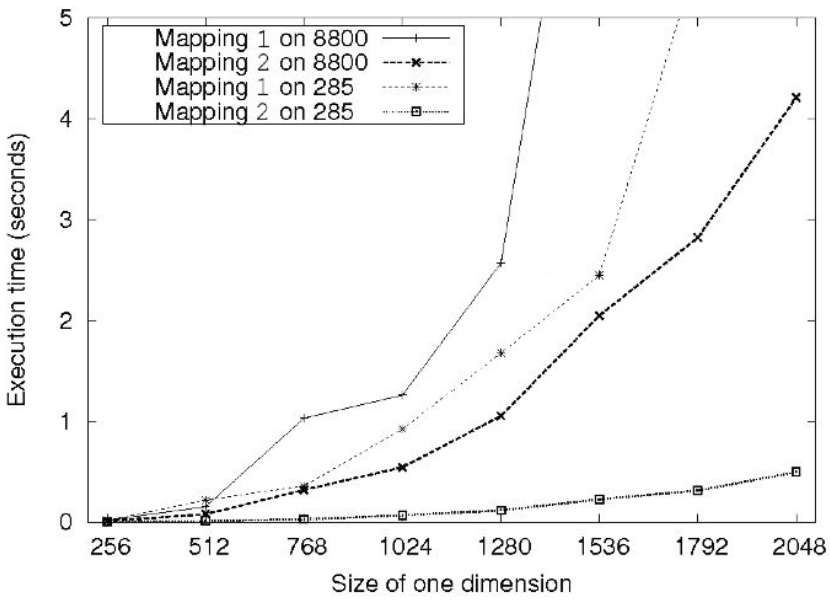
- Отображение нитеи 2: с пространством индекса NxM :

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

- Оба отображения производят функционально эквивалентные версии программы

Сопоставление потоков

- На этом рисунке показано выполнение двух сопоставлений потоков на графических процессорах NVIDIA GeFor



- Обратите внимание, что отображение 2 намного превосходит производительность для обоих графических процессоров

Сопоставление потоков

- Расхождение во времени выполнения между сопоставлениями связано с доступом к данным на глобальной шине памяти
  - Предполагая, что данные в строке сохраняются последовательно в памяти
  - Чтобы обеспечить совместный доступ, последовательные потоки в одном wavefronts должны отображаться в столбцы (второе измерение) матриц
    - Это даст совместные обращения в матрицах B и C
    - Для Matrix A итератор i3 определяет шаблон доступа для данных с большими строками, поэтому отображение нитей не влияет на него

Сопоставление потоков

- В отображении 1 последовательные потоки (tx) сопоставляются с разными строками матрицы C, а непоследовательные потоки (ty) отображаются в столбцы матрицы B
  - Отображение вызывает неэффективные обращения к памяти

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

Сопоставление потоков

- В отображении 2 последовательные потоки (tx) отображаются в последовательные элементы в матрицах B и C
  - Доступ к обеим этим матрицам будет объединен
    - Степень объединения зависит от рабочих групп и размеров данных

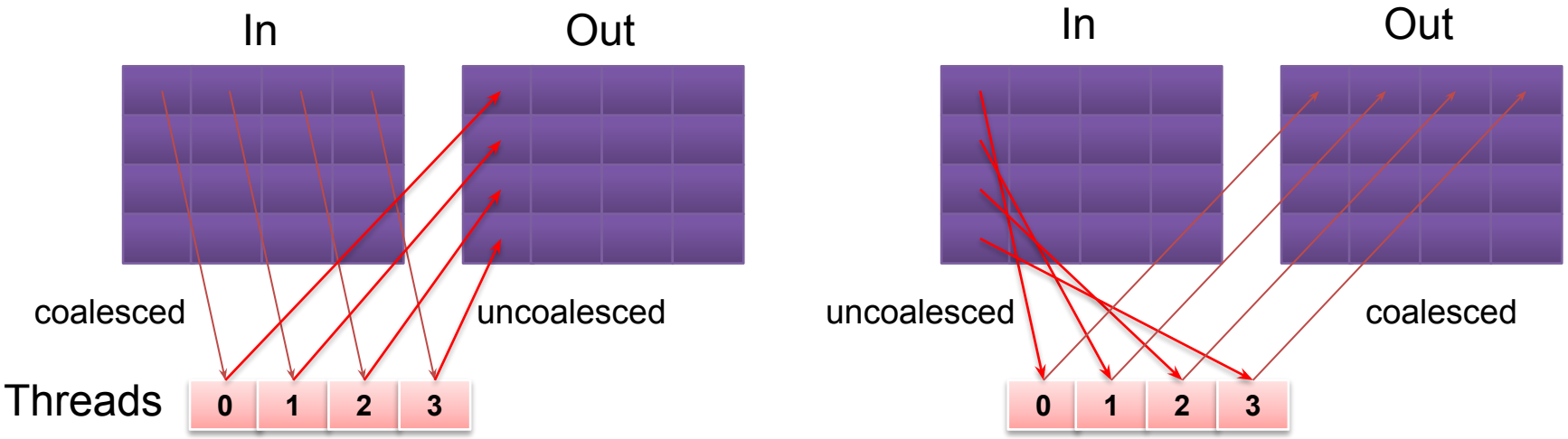
```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

Сопоставление потоков

- В общем случае потоки могут быть созданы и сопоставлены с любым элементом данных, управляя значениями, возвращаемыми функциями идентификатора потока
- Следующий пример переноса матрицы отобразит, как идентификаторы потоков могут быть изменены для обеспечения эффективного доступа к памяти

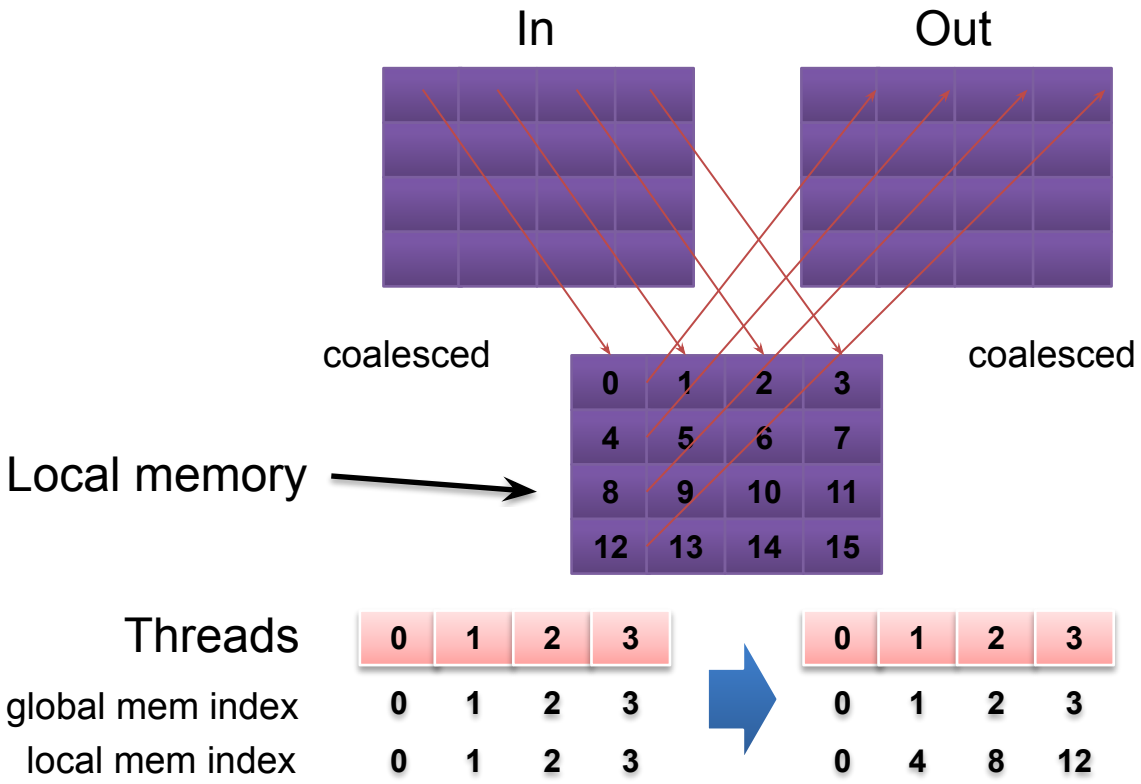
Транспонирование матрицы

- Трансформация матрицы - это простой метод
  - Out(x,y) = In(y,x)
- Независимо от того, какое сопоставление выбрано, одна операция (чтение / запись) приведет к совместному доступу, в то время как другая (запись / чтение) создает неизолированные обращения
  - Обратите внимание, что данные должны быть прочитаны во временное место (например, регистр) перед записью в новое место



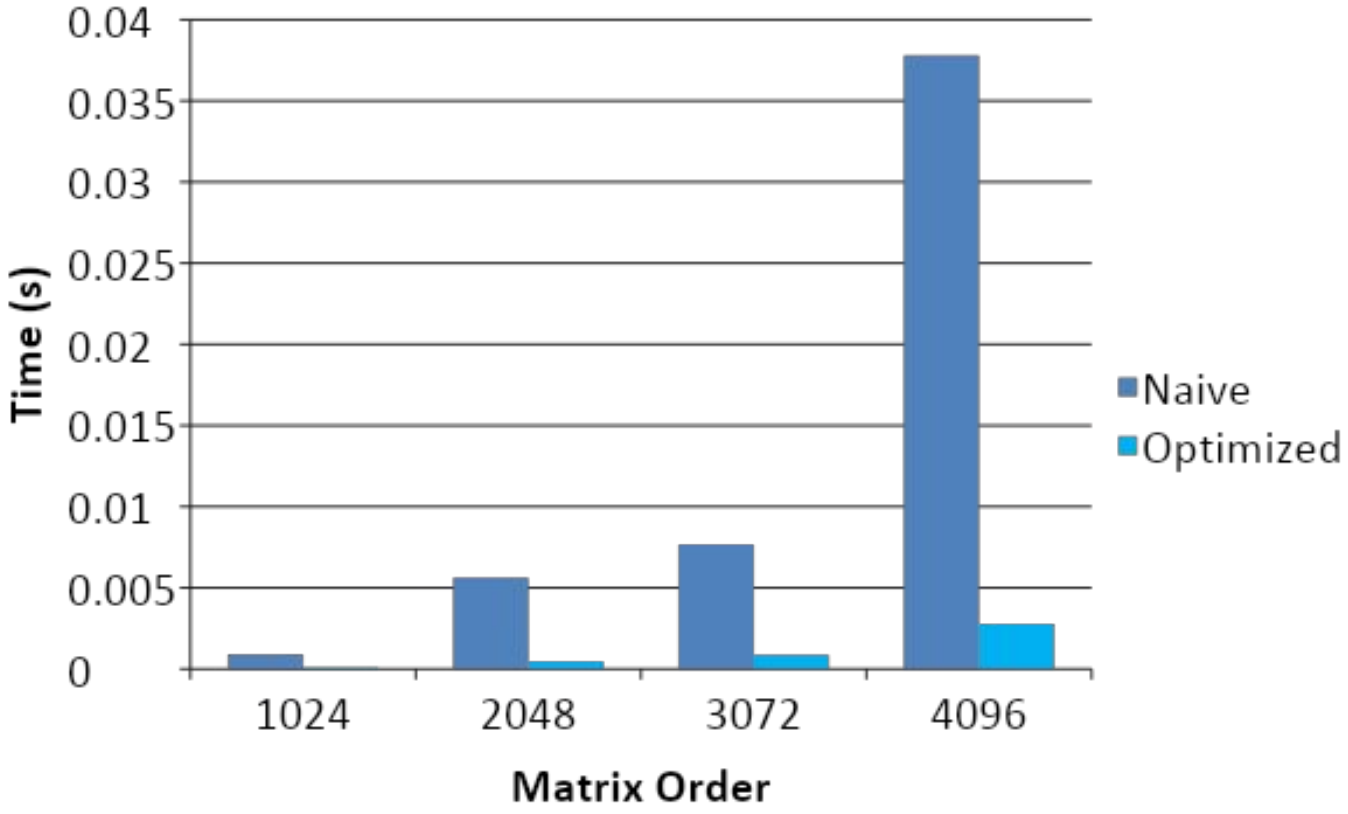
Транспонирование матрицы

- Если локальная память используется для буферизации данных между чтением и записью, мы можем изменить распределение потоков, чтобы обеспечить совместный доступ в обоих направлениях
  - Обратите внимание, что рабочая группа должна быть квадратной



Транспонирование матрицы

- На следующем рисунке показано сравнение производительности двух ядер транспонирования для матриц размера NxM на AMD 5870 GPU
  - «Оптимизированный» использует локальную память и переназначение потоков



Выводы

- Хотя писать простую программу OpenCL относительно легко, оптимизация кода может быть сложнее
  - Объединение доступа к памяти
  - Векторизация
  - Локальная память
  - Константная память
- При создании рабочих групп необходимо учитывать аппаратные ограничения (количество регистров, размер локальной памяти и т. д.)
  - Рабочие группы должны быть соответствующим образом рассчитаны, чтобы максимально увеличить количество активных рабочих элементов и правильно скрыть задержки
- Отображение резьбы и ее влияние на доступ к памяти имеют решающее значение для производительности ядра OpenCL

События, профилирование и отладка

События

- События используются для синхронизации между отдельными командами
  - т. е. создать граф зависимостей команд
- Явная синхронизация требуется для
  - Команд вне очереди
  - Нескольких командных очередей
- События также используются для хранения информации о времени, возвращаемой устройством

События

- В дополнение к заданию зависимостей события используются для базового профилирования команд
- Профилирование с использованием событий должно быть включено явно при создании очереди команд
  - CL_QUEUE_PROFILING_ENABLE должен быть установлен флаг
  - Требование, чтобы среда выполнения для создания временных меток для событий могла замедлить выполнение
- Дескриптор для хранения информации о событии может быть передан для всех команд clEnqueue *
  - Когда задействованы команды, такие как clEnqueueNDRangeKernel и clEnqueueReadBuffer, информация о синхронизации записывается в предоставленном событии

Использование событий

- Используя события можем:
  - Измерить время выполнения вызовов clEnqueue *, таких как выполнение ядра или явная передача данных
  - Использовать события для расписания асинхронных передач данных между хостом и устройством
  - профилирование приложения для анализа потока выполнения
- Временные метки времени совместимы как для процессоров, так и для графических процессоров

Профилирование с событиями

```
cl_int      clGetEventProfilingInfo (cl_event event,
                                     cl_profiling_info param_name,
                                     size_t param_value_size,
                                     void *param_value,
                                     size_t *param_value_size_ret)
```

- **clGetEventProfilingInfo позволяет нам запрашивать cl_event для получения желаемых значений счетчика**
- **Информация о сроках, возвращаемая как типы данных cl_ulong**
  - **Возвращает временную метку для наносекундной гранулярности**

Профилирование с событиями

- В таблице показаны типы событий, описанные с использованием cl_profiling_info перечисляемого типа

Event Type	Description
CL_PROFILING_COMMAND_QUEUED	Команда помещается в очередь команд хостом
CL_PROFILING_COMMAND_SUBMIT	Команда отправляется хостом на устройство, связанное с командной очередью.
CL_PROFILING_COMMAND_START	Команда запускает выполнение на устройстве.
CL_PROFILING_COMMAND_END	Команда закончила выполнение на устройстве.
CL_PROFILING_COMMAND_COMPLETE	Команда и все ее дочерние команды завершили выполнение на устройстве

Профилирование с событиями

- События OpenCL могут быть легко использованы для синхронизации ядер.

```
clGetEventProfilingInfo( event_time, CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

- Этот метод является надежным для оптимизации производительности, поскольку он использует счетчики с устройства

- Принимая разницу между стартовыми и конечными временными метками, мы вычитаем накладные расходы, такие как время, проведенное в командной очереди.

```
clGetEventProfilingInfo( event_time, CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &starttime, NULL);
```

```
unsigned long elapsed =  (unsigned long)(endtime - starttime);
```

Профилирование с событиями

- Прежде чем получать информацию о времени, мы должны убедиться, что события, которые нас интересуют, завершились
- Существуют различные способы ожидания событий:
  - clWaitForEvents (numEvents, eventList)
  - clFinish (commandQueue)
- Разрешение таймера можно получить из флага CL_DEVICE_PROFILING_TIMER_RESOLUTION при вызове clGetDeviceInfo

Получении информации о событии

```
cl_int      clGetEventInfo (cl_event event,
                            cl_event_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

- **clGetEventInfo может использоваться для возврата информации о объекте события**
- **Он может возвращать сведения о командной очереди, контексте, типе команды, связанной с событиями, статусе выполнения**
- **Эта команда может использоваться вместе с синхронизацией, предоставляемой clGetEventProfilingInfo как часть структуры профилирования высокого уровня для отслеживания команд**

Пользовательские события

- OpenCL 1.1 и выше определяют объект события пользователя. В отличие от команд clEnqueue * пользовательские события могут быть установлены пользователем

```
cl_event        clCreateUserEvent (cl_context context, cl_int *errcode_ret)
```

- Когда мы создаем пользовательское событие, статус имеет значение CL_SUBMITTED

```
cl_int          clSetUserEventStatus (cl_event event, cl_int execution_status)
```

- clSetUserEventStatus используется для установки статуса выполнения объекта пользовательского события.
- Пользовательское событие может быть установлено только CL_COMPLETE

Пользовательские события

- Простой пример пользовательских событий, которые запускаются и используются в командной очереди

```
// Create user event which will start the write of buf1
user_event = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ..., 1, &user_event , NULL);
// The write of buf1 is now enqueued and waiting on user_event

X = foo(); // Lots of complicated host processing code

clSetUserEventStatus(user_event, CL_COMPLETE);

// The clEnqueueWriteBuffer to buf1 can now proceed
```

Список ожидания

- Список ожидания - это массивы типа cl_event
- Все методы clEnqueue * также принимают

```
cl_int  clWaitForEvents (cl_uint num_events, const cl_event *event_list)
```

- OpenCL определяет waitlists с установкой приоритета

Wait Lists

```
cl_int  clEnqueueBarrierWithWaitList (cl_command_queue command_queue,
                                      cl_uint num_events_in_wait_list,
                                      const cl_event *event_wait_list,
                                      cl_event *event)
```

- Обеспечивает завершение списка событий до того, как конкретная команда выполнится

```
cl_int  clEnqueueMarkerWithWaitList (cl_command_queue command_queue,
                                     cl_uint num_events_in_wait_list,
                                     const cl_event *event_wait_list,
                                     cl_event *event)
```

- Обеспечивает отметку места в очереди с уникальным объектом события, который можно использовать для синхронизации

Event Callbacks

```
cl_int        clSetEventCallback (cl_event event,
                                  cl_int command_exec_callback_type,
                                  void (CL_CALLBACK *pfn_event_notify)(cl_event event,
                                                                       cl_int event_command_exec_status,
                                                                       void *user_data),
                                  void *user_data)
```

- OpenCL 1.1 или выше позволяет зарегистрировать callbacks для определенного статуса выполнения команды
  - Обратные вызовы событий могут использоваться для установки новых команд на основе изменений состояния события в неблокирующем режиме
  - Использование блокирующих версий clEnqueue * функции OpenCL в callbacks приводят к неопределенному поведению
- Callbacks принимает cl_event, статус и указатель на пользовательские данные в качестве параметров

Синхронизация командных очередей

- Способы синхронизации очереди команд

  Flush:

  ```
  cl_int clFlush (cl_command_queue command_queue)
  ```

  – Отправляет все команды в очередь на вычислительное устройство

  – Нет г… clFlush вернется

  Finish:

  ```
  cl_int clFinish (cl_command_queue command_queue)
  ```

  – Блокирует хост, ожидая завершения всех команд в очереди кома…

  Barrier:

  ```
  cl_int clEnqueueBarrierWithWaitList (cl_command_queue command_queue,
                                       cl_uint num_events_in_wait_list,
                                       const cl_event *event_wait_list,
                                       cl_event *event)
  ```

  – Задает точку синхронизации: все предыдущие команды в очереди завершены до того, как будут выполнены другие команды

Отладка с использование **printf**

- Начиная с OpenCL 1.2, OpenCL C поддерживает печать во время выполнения с помощью printf
- printf точно соответствует определению, найденному в стандарте C99
- printf может использоваться для печати информации о потоках или помощи в отслеживании ошибок
- printf работает путем вывода буферизации до конца выполнения и передачи вывода обратно на хост
  - Важно, чтобы ядро завершило загрузку печатной информации
  - Комментировать код, следующий за printf, является хорошей техникой, если в ядре наблюдается сбой

Отладка с использование **printf**

- В следующем примере выводится информация о потоках, пытающихся выполнить неправильный доступ к памяти

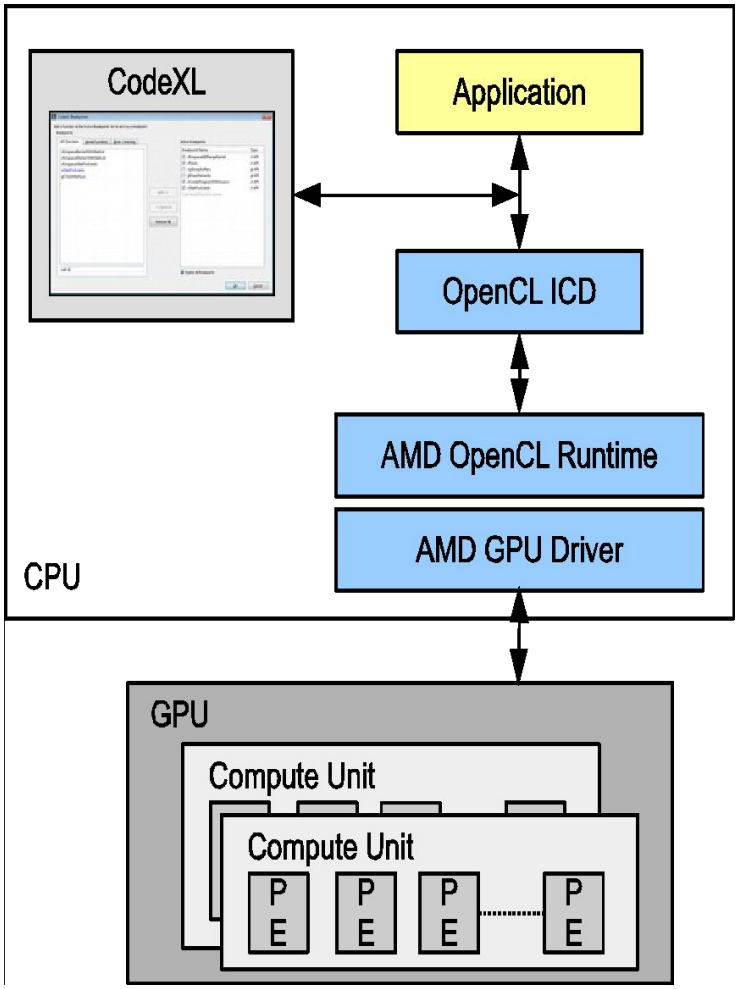
```
int myIdxX = ... // column index for addressing a matrix
int myIdxY = ... // row index for addressing a matrix
if(myIdxX < 0 || myIdxX >= cols ||
   myIdxY < 0 || myIdx >= rows)
  {
    printf("Work item %d,%d: bad index (%d, %d)\n",
get_global_id(1), get_global_id(0),                    myIdxX,
myIdxY));
  }
```

CodeXL

- Интегрированный профайлер, анализатор ядра и отладчик, разработанные AMD
- Режим профилирования
  - Собирает данные производительности из среды исполнения OpenCL и графических процессоров AMD во время выполнения
- Режим анализа
  - Статически компилирует, анализирует и дизассемблирует ядра OpenCL для графических процессоров AMD
- Режим отладки
  - Отлаживает приложение, перейдя через вызовы OpenCL API, и исходный код ядра
  - Просматривает параметры функции и уменьшает потребление памяти

Отладка с использовнием CodeXL

- CodeXL перехватывает вызовы OpenCL API между приложением и OpenCL ICD
- CodeXL может выполнять отладку на уровне API
  - Записать историю вызовов OpenCL API
  - Информация о программе и ядре
  - Данные изображения и буфера
  - Проверка памяти
  - Статистика использования API
  - Контрольные точки функции ядра



Профилирование с CodeXL

- Режим профилирования
  - Трассировка временной шкалы приложения графического процессора
  - Счетчики производительности графического процессора при выполнении ядра
  - Сбор информации о производительности ЦП

Просмотр временной шкалы приложения

- Обеспечивает визуальное представление выполнения приложения

Просмотр трассировки API-интерфейса хоста

- Выводит список всех вызовов OpenCL API, сделанных каждым потоком хоста в приложении

Host Thread 12328 | Summary

Call Index	Interface	Parameters	Result	Device Block	Kernel Occupancy	CPU Time	Device Time
102	clSetKernelArg	0x000000000458F620;6;8;[0x468AD00]	CL_SUCCESS			0.0003	
103	clEnqueueNDRangeKernel	0x0000000000514170;0x000000000458F620;1;NULL;[32768];[128];0;NULL;NULL	CL_SUCCESS	nbody sim	50%	0.1530	59.6324
104	clFlush	0x0000000000514170	CL_SUCCESS			0.0031	
105	clEnqueueMapBuffer	0x0000000000514170;0x000000000468AB90;CL_TRUE;CL_MAP_READ;0;524288;0;NULL;NULL...	0x0000000007A77...	512.0 KB MAP_BU...		64.2462	1.2806
106	clSetKernelArg	0x000000000458F620;0;8;[0x468AB90]	CL_SUCCESS			0.0017	
107	clSetKernelArg	0x000000000458F620;1;8;[0x468AD00]	CL_SUCCESS			0.0003	
108	clSetKernelArg	0x000000000458F620;5;8;[0x468A8B0]	CL_SUCCESS				
109	clSetKernelArg	0x000000000458F620;6;8;[0x468AA20]	CL_SUCCESS			0.0003	
110	clEnqueueNDRangeKernel	0x0000000000514170;0x000000000458F620;1;NULL;[32768];[128];0;NULL;NULL	CL_SUCCESS	nbody sim	50%	0.0626	54.8519
111	clFlush	0x0000000000514170	CL_SUCCESS			0.0048	
112	clEnqueueUnmapMemObject	0x0000000000514170;0x000000000468AB90;0x0000000007A77000;0;NULL;NULL	CL_SUCCESS			0.0205	
113	clFlush	0x0000000000514170	CL_SUCCESS			0.0068	
114	clEnqueueMapBuffer	0x0000000000514170;0x000000000468A8B0;CL_TRUE;CL_MAP_READ;0;524288;0;NULL;NULL...	0x0000000078B2...	512.0 KB MAP_BU...		2.9424	1.6532
115	clSetKernelArg	0x000000000458F620;0;8;[0x468A8B0]	CL_SUCCESS			0.0017	
116	clSetKernelArg	0x000000000458F620;1;8;[0x468AA20]	CL_SUCCESS			0.0003	
117	clSetKernelArg	0x000000000458F620;5;8;[0x468AB90]	CL_SUCCESS				
118	clSetKernelArg	0x000000000458F620;6;8;[0x468AD00]	CL_SUCCESS				
119	clEnqueueNDRangeKernel	0x0000000000514170;0x000000000458F620;1;NULL;[32768];[128];0;NULL;NULL	CL_SUCCESS	nbody sim	50%	0.0623	46.4723
120	clFlush	0x0000000000514170	CL_SUCCESS			0.0034	

Сбор счетчиков производительности ядра GPU

- Счетчики производительности ядра графического процессора можно использовать для поиска возможных узких мест при выполнении ядра

	Method	tion	ThreadID	Time	VGPRs	SGPRs	FCStacks	ernelOccupanc	Wavefronts	VALUInsts	SALUInsts	VFetchInsts	SFetchInsts	VWriteInsts	VALUUtilization (%)	VALUBusy (%)	SALUBusy (%)
1	nbody_sim_kl ...	1	12616	61.77452	46	48	NA	50	512	626723	53279	2	32781	2	100	59.79	6.55

Вывод

- Отладка
  - printf может быть легким методом отладки
  - режим отладки CodeXL
- Профилирование: события OpenCL позволяют нам использовать модель исполнения и синхронизацию для повышения производительности приложений
  - Конструкции синхронизации очереди команд для крупнозернистого управления
  - Использование событий для мелкозернистого контроля над приложением
  - OpenCL 1.1 или выше позволяет более сложную обработку событий и добавляет обратные вызовы, а также обеспечивает события, которые могут быть вызваны пользователем
  - Режим профилирования CodeXL может генерировать информацию синхронизации без изменения исходного кода