




Казанский  
федеральный  
университет

ВЫСШАЯ ШКОЛА  
информационных технологий  
и информационных систем


# Иерархия памяти в CUDA

Эдуард Храмченков

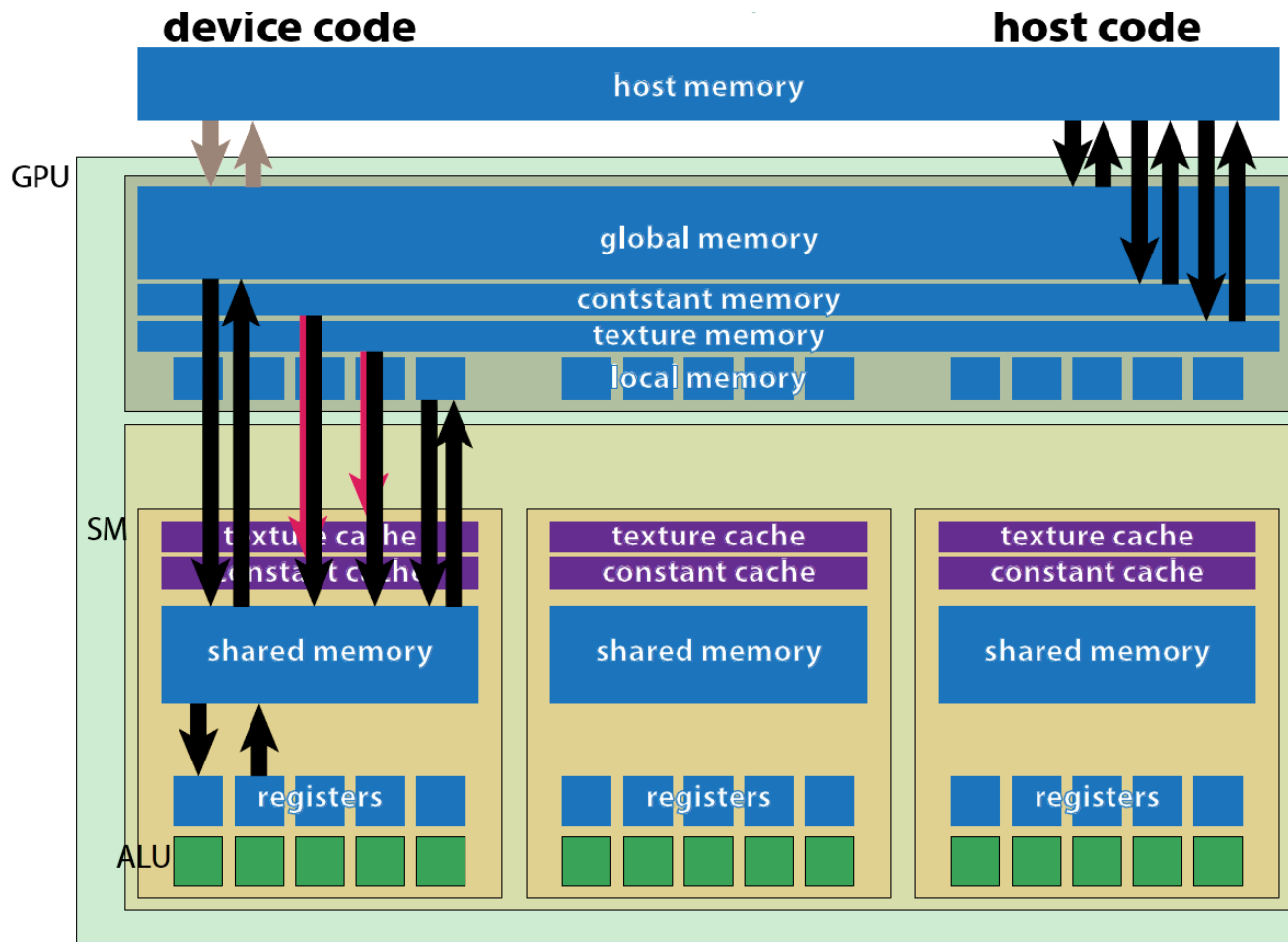
# Память GPU

- ▶ Организация памяти и работы с ней на CPU и GPU существенно отличаются
  - ▶ В CPU основная площадь схемы занята кэш-памятью
  - ▶ В GPU основная часть схемы отведена под вычислительные блоки
  - ▶ Программирование на CPU не предполагает прямого доступа к кэш-памяти и манипуляциям с разными видами памяти
- 

# Память GPU

- ▶ В GPU существует несколько видов памяти – некоторые в мультипроцессорах, другие в DRAM
  - ▶ Часть памяти GPU должно управляться программно
  - ▶ Эффективное использование памяти – один из важнейших элементов написания быстрого кода
- 


# Память GPU



# Память GPU

Тип памяти	Доступ	Видимость	Скорость	Расположение
Регистры	R/W	Per-thread	Высокая	SM
Локальная	R/W	Per-thread	Низкая	DRAM(device)
Разделяемая	R/W	Per-block	Высокая	SM
Глобальная	R/W	Per-grid	Низкая	DRAM(device)
Константная	R/O	Per-grid	Высокая	DRAM(device)
Текстурная	R/O	Per-grid	Высокая	DRAM(device)

# Регистры

- ▶ Распределяются между потоками блока на этапе компиляции
  - ▶ Каждый поток получает в монопольное пользование некоторое количество регистров на все время исполнения ядра
  - ▶ Доступ к регистрам других потоков запрещен
  - ▶ Расположен в мультипроцессоре – минимальная латентность
- 

# Локальная память

- ▶ Размещена в DRAM GPU – высокая латентность порядка 400-800 тактов
- ▶ В локальную память попадают
  - Union
  - Динамические массивы
  - Структуры и статические массивы большого размера
- ▶ Все переменные, если ядро использовало все регистры

# Текстурная память

- ▶ Текстурная память и аппаратные схемы интерполяции объединены на GPU в текстурные блоки
- ▶ Расположена в DRAM GPU
- ▶ Обладает независимым кэшем – высокая скорость доступа
- ▶ Доступны всем потокам сетки, но только на чтение
- ▶ Запись с CPU с помощью CUDA API
- ▶ Объем равен свободному объему DRAM, доступ ведется через специальный кэш, работа с данными как с текстурами




# Текстурная память

- ▶ В текстурном блоке аппаратно реализованы следующие функции:
  - Фильтрация текстурных координат
  - Билинейная или точечная интерполяция
  - Разумное возвращаемое значение в случае, когда значения текстурных координат выходят за допустимые границы
  - Обращение по нормализованным или целочисленным координатам
  - Возвращение нормализованных значений
  - Кэширование данных

# Текстурная память

- ▶ В отличие от методов работы с глобальной памятью, работа с текстурной памятью более трудоемка
- ▶ Для использования текстурной памяти необходимо осуществить связывание (bind) данных из глобальной памяти в текстуру
- ▶ Работа с текстурами ведется через текстурную ссылку, данные хранятся в x-y-z координатах структуры

# Константная память

- ▶ Расположена в DRAM GPU
  - ▶ Обладает независимым кэшем – высокая скорость доступа
  - ▶ Доступна для чтения с GPU, для чтения и записи с хоста при помощи функций CUDA API
  - ▶ Подходит для размещения небольшого количества данных, которые будут доступны всем нитям
- 

# Константная память

```
// Скопировать данные из памяти CPU в константную память GPU
cudaError_t cudaMemcpyToSymbol(const char * symbol, const void * src,
size_t count, size_t offset, enum cudaMemcpyKind kind);

// Скопировать данные из константной памяти GPU в память CPU
cudaError_t cudaMemcpyFromSymbol(void * dst, const char * symbol, size_t
count, size_t offset, enum cudaMemcpyKind kind );

//Асинхронная версия cudaMemcpyToSymbol
cudaError_t cudaMemcpyToSymbolAsync(const char * symbol, const void * src,
size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t
stream);


//Асинхронная версия cudaMemcpyFromSymbol
cudaError_t cudaMemcpyFromSymbolAsync(void * dst, const char * symbol,
size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t
stream);
```

# Константная память

```
__constant__ float contsData [256]; // константная память GPU
float hostData [256]; // данные в памяти CPU

// Скопировать данные из памяти CPU в константную память GPU
cudaMemcpyToSymbol(constData, hostData, sizeof(data), 0,
cudaMemcpyHostToDevice);
```

# Глобальная память

- ▶ Глобальная память – память DRAM GPU
  - ▶ Высокая латентность – около 800 тактов
  - ▶ Может выделяться как с CPU при помощи CUDA API, так и потоками на GPU, с помощью malloc
  - ▶ Начиная с архитектуры Fermi кэшируется, но эффективность кэширования незначительна
  - ▶ Может использоваться всеми потоками сетки
  - ▶ Минимизация доступа к глобальной памяти – основной метод создания эффективного кода
- 

# Глобальная память

```
//Выделить память на GPU, вызов с хоста
cudaError_t cudaMalloc(void ** devPtr, size_t size);

//Освободить память на GPU, вызов с хоста
cudaError_t cudaFree(void * devPtr);

//Выделить память на GPU с выравниваем, сдвиг определяется pitch
cudaError_t cudaMallocPitch(void ** devPtr, size_t * pitch,
size_t width, size_t height);

//Копирование данных между хостом и GPU, kind задает направление
cudaError_t cudaMemcpy( void * dst, const void * src, size_t
size, enum cudaMemcpyKind kind );
```

# Глобальная память

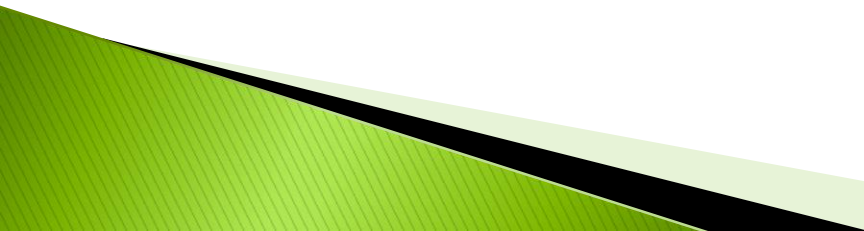
- ▶ Указатель от `cudaMalloc` имеет смысл только для адресного пространства GPU.
- ▶ Направление копирования данных определяются параметром `kind`:
  - `host -> device` – `cudaMemcpyHostToDevice`
  - `device -> host` – `cudaMemcpyDeviceToHost`




# Глобальная память

- ▶ В CUDA используется прямая адресация глобальной памяти GPU
- ▶ Для хранения адресов подходят обычные указатели, и для них корректна адресная арифметика
- ▶ Если память выделяется с помощью *cudaMalloc*, то выделенный диапазон имеет смысл только в контексте GPU-ядра
- ▶ Память, выделенная на одном GPU некорректна по отношению к другому GPU

# Глобальная память

- ▶ Копирование данных в глобальной памяти внутри одного GPU обычно производится на порядок быстрее, чем внутри памяти хоста
  - ▶ Копирование данных между памятью хоста и памятью GPU значительно медленнее
    - Внутри карты – порядка 144 GBps
    - Между хостом и GPU – порядка 8 GBps (PCI-Express 3.0)
    - Между хостом и несколькими GPU – порядка 12 GBps (PCI-Express 3.0)
- 

# Разделяемая память

- ▶ Shared (разделяемая память) – один из важнейших типов памяти:
  - ▶ Расположена в мультипроцессоре, но выделяется на уровне блоков
  - ▶ Каждый блок получает в распоряжение одно и то же количество разделяемой памяти
  - ▶ Размер 48КБайт (+16Кбайт кэша на Fermi)
  - ▶ Низкая латентность – такая же как у регистров
  - ▶ Может использоваться всеми потоками блока
- 

# Разделяемая память

- ▶ Эффективно использовать разделяемую память как буфер, вместо обращения к глобальной памяти
- ▶ Объем разделяемой памяти делится поровну между всеми блоками потоков, запущенными на мультипроцессоре
- ▶ Размер разделяемой памяти может быть задан в CUDA-ядре при определении массивов с атрибутом `__shared__` или в параметрах запуска ядра

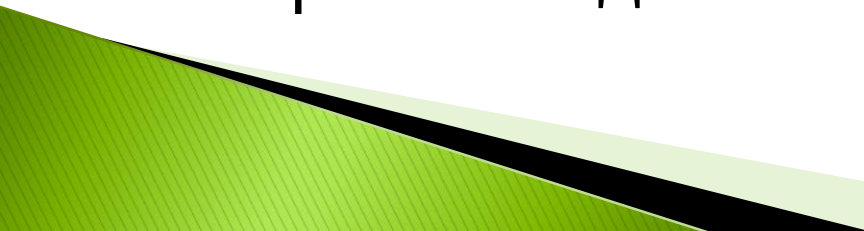
# Разделяемая память

- ▶ Ситуация – поток А изменяет значение переменной X в разделяемой памяти, поток В должен читать измененное значение
- ▶ Какое значение в реальности считает поток В
- ▶ Порядок выполнения потоков не определен – результат операции будет undefined
- ▶ Чтобы избежать race condition необходима барьерная синхронизация потоков  
\_syncthreads()


# Разделяемая память

- ▶ Типичный алгоритм:
  - Загрузка данных из global в shared
  - `__syncthreads ()`
  - Вычисления над этими данными
  - `__syncthreads ()`
  - Запись данных из shared в global

# Кэширование

- ▶ Начиная с архитектуры Fermi используются L1 и L2 кэши
  - ▶ Кеш L1 находится на каждом мультипроцессоре
  - ▶ Кеш L2 общий и имеет размер 768 Кбайт
  - ▶ Длина кэш-линии составляет 128 байт
  - ▶ Если размер слова для каждой нити равен 4 байтам, то запросы в память всех 32 потоков варпа объединяются в один
- 

# Кэширование

- ▶ Кеш L1 и разделяемая память расположены на одном физическом носителе – 64Кбайт
  - ▶ Объем памяти делится между кэшем и разделяемой памятью в соотношении 48/16 МБ или 16/48 МБ
  - ▶ Выбор контекста кэш/L1 осуществляется для каждого ядра
- 



# Глобальная память

```
//48 КБ кэш, 16 КБ L1
```

```
cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferShared);
```

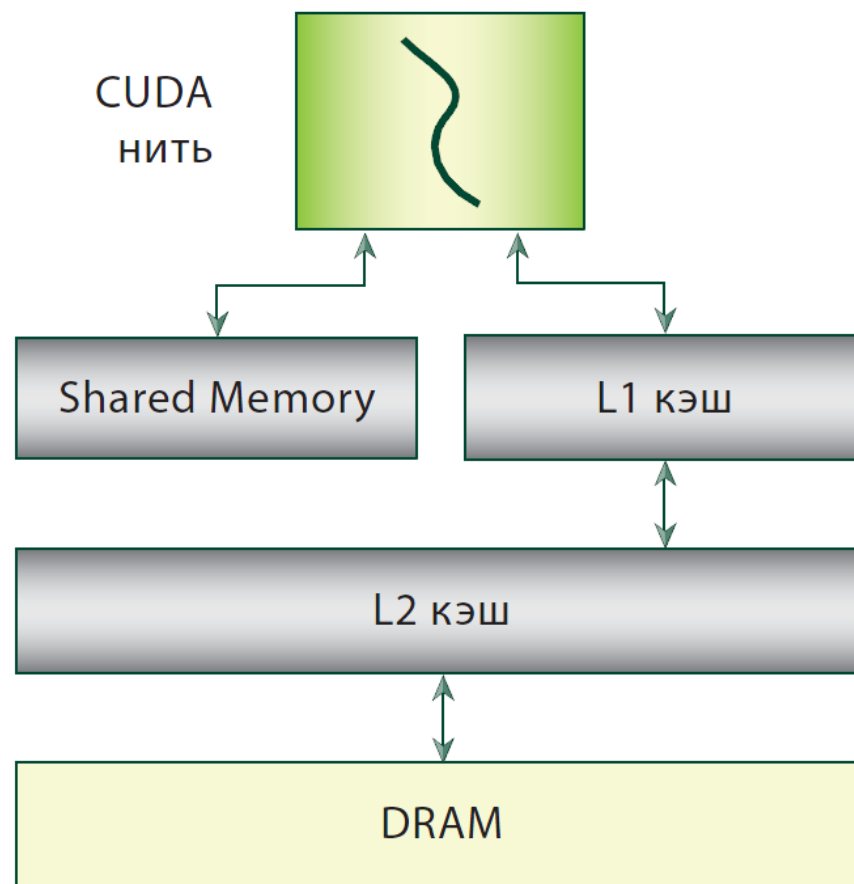
```
//48 КБ L1, 16 КБ кэш
```

```
cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferL1);
```


```
//Используется текущий контекст
```

```
cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferNone);
```

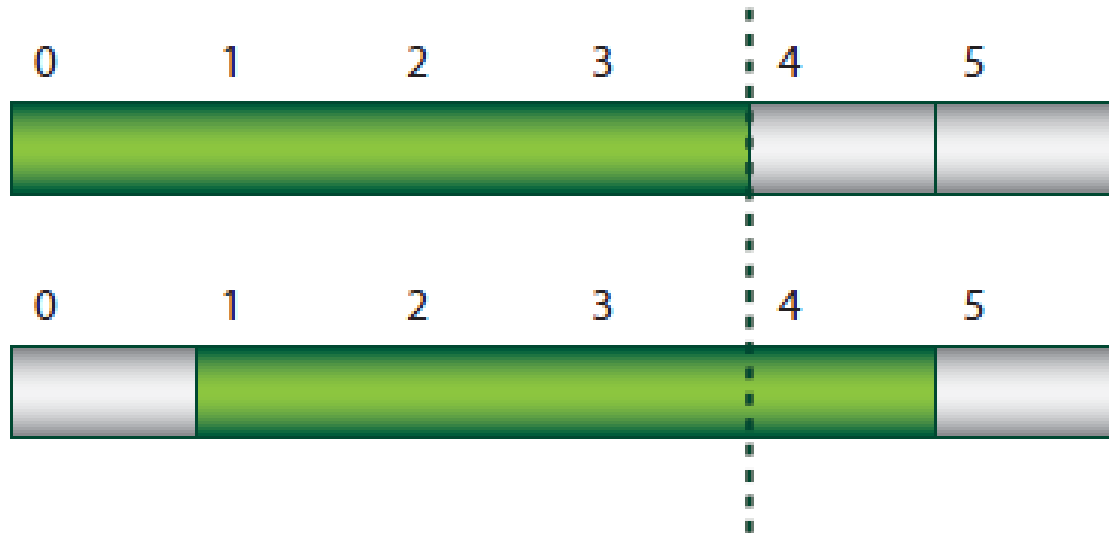
# Кэширование



# Оптимизация доступа в память

- ▶ Выравнивание - при чтении и записи значений глобальной памяти на низком уровне используются выровненные 32-, 64- и 128-битные слова
  - ▶ Вся выделяемая память CUDA всегда выровнена по 256 байт
  - ▶ Если адрес объекта невыровнен – требуется больше обращений к памяти
  - ▶ Чем выше латентность типа памяти, тем важнее выравнивание
- 


# Оптимизация доступа в память




# Оптимизация доступа в память

- ▶ `struct vec3{float x, y, z};`
- ▶ Размер `vec3` – 12 байт
- ▶ При создании массива `vec3` с выравненным базовым адресом, выравненным окажется только каждый 4й элемент
- ▶ Решение
  - Фиктивный элемент
  - `struct __attribute__((aligned(16)))`

# Оптимизация доступа в память

- ▶ Coalescing – объединение запросов потоков варпа в одно обращение к непрерывному блоку глобальной памяти
  - ▶ Начиная с архитектуры Fermi все запросы варпа объединяются в один
  - ▶ При использовании L1 кэша можно свести к минимуму эффекты от невыравненных адресов данных
- 

# Оптимизация доступа в память

- ▶ Доступ к памяти эффективнее для структур массивов, чем для массива структур
  - ▶ В случае если обращение нитей идет к разным областям памяти, объединения не происходит
  - ▶ Решение – использовать разделяемую память для создания временного буфера с данными для варпов
- 

# Пример 0

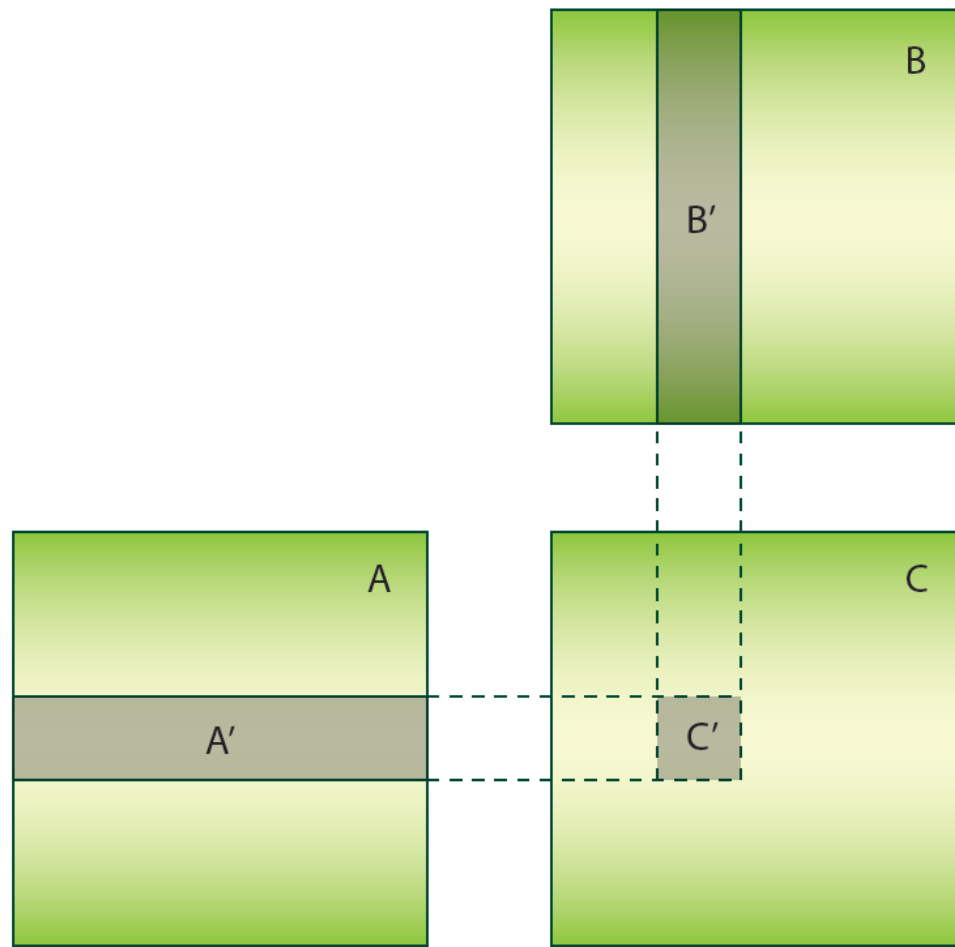
- ▶ Квадратные матрицы  $n \times n$
- ▶  $n$  – степень двойки
- ▶ Двумерная сетка блоков, двумерные блоки  $16 \times 16$
- ▶ «Наивная» реализация
  - На каждый элемент  $2N$  арифметических операций и  $2N$  обращений к глобальной памяти
  - Производительность ограничена пропускной способностью памяти



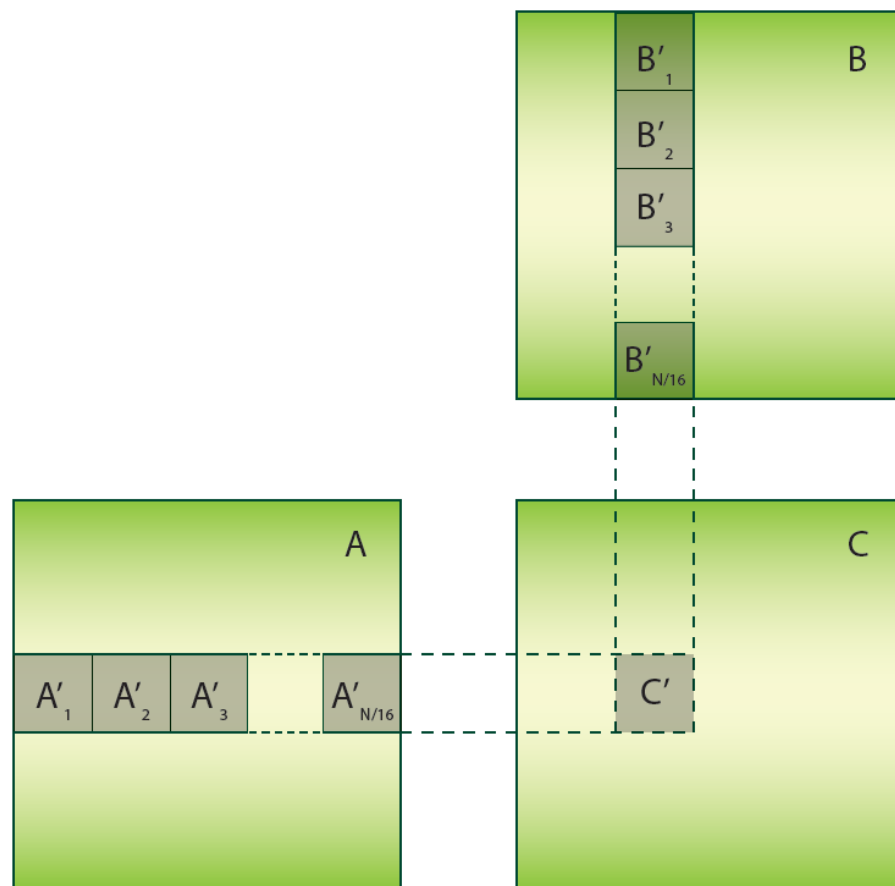
# Пример 0

- ▶ Оптимизация «наивного» алгоритма:
  - Считаем  $n$  кратным 16
  - Для вычисления подматрицы  $C'$  необходимы лишь полосы  $A'$  и  $B'$  размером  $n \times 16$  исходных матриц  $A$  и  $B$
  - При  $n > 1024$  полосы не лезут в разделяемую память
  - Выход – разбиение полос на подматрицы  $16 \times 16$
  - $C'$  – сумма попарных произведений подматриц из этих полос


# Пример 0



# Пример 0




# Оптимизация доступа в память

- ▶ Для повышения пропускной способности разделяемая память разбита на 32 банка, каждый из которых может выполнить одно чтение или запись 32-битного слова
  - ▶ Подряд идущие 32-битные слова попадают в различные подряд идущие банки
  - ▶ Если все 32 нити варпа обращаются к 32 32-битным словам, находящимся в разных банках, то данные будут получены без дополнительных задержек
- 


# Оптимизация доступа в память

Банк 0	Банк 1	Банк 2	...	Банк 30	Банк 31
<i>слово 0</i>	<i>слово 1</i>	<i>слово 2</i>	...	<i>слово 30</i>	<i>слово 31</i>
<i>слово 32</i>	<i>слово 33</i>	<i>слово 34</i>	...	<i>слово 62</i>	<i>слово 63</i>
...	...	...	...	...	...

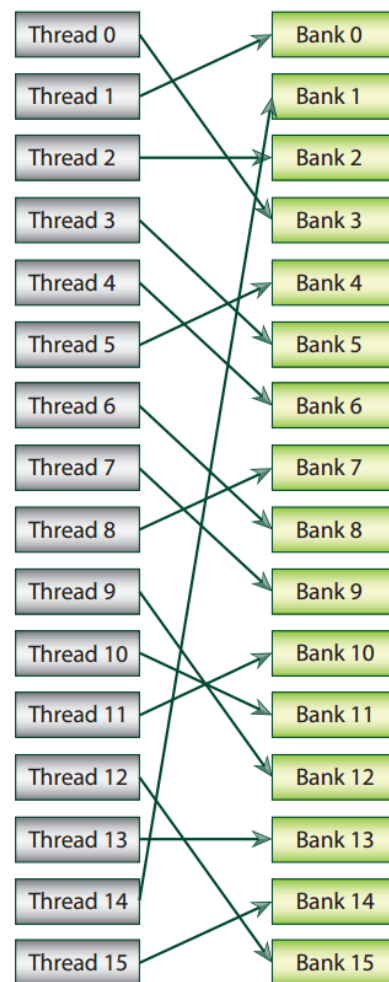
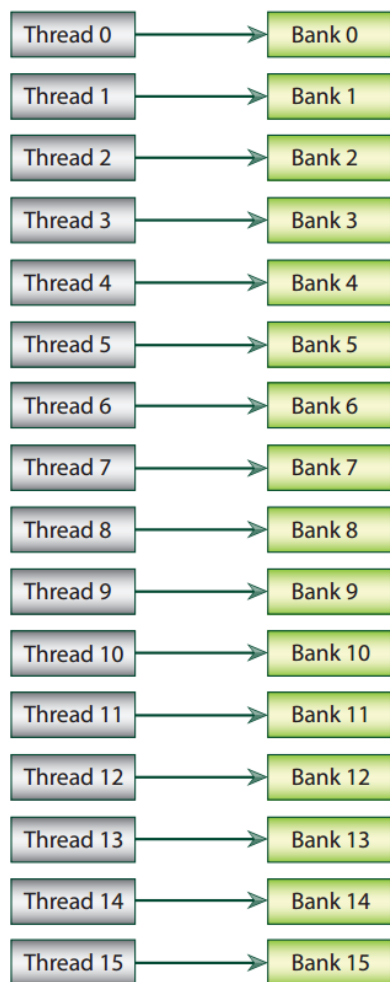
# Оптимизация доступа в память

- ▶ Такая организация разделяемой памяти позволяет оптимизировать типичные задачи для GPU
  - ▶ Обращения к одному банку могут быть выполнены только последовательно
  - ▶ Одновременный запрос данных из одного банка несколькими нитями называется конфликтом банков и характеризуется порядком конфликта – максимальным числом обращений в один банк
- 

# Оптимизация доступа в память

- ▶ Если имеет место конфликт второго порядка хотя бы для одного банка, то скорость доступа к разделяемой памяти снижается вдвое
  - ▶ Особым случаем является обращение всех 32 нитей варпа к одному и тому же элементу одного банка: в этом случае включается режим broadcast-запроса, и конфликта не возникает
- 

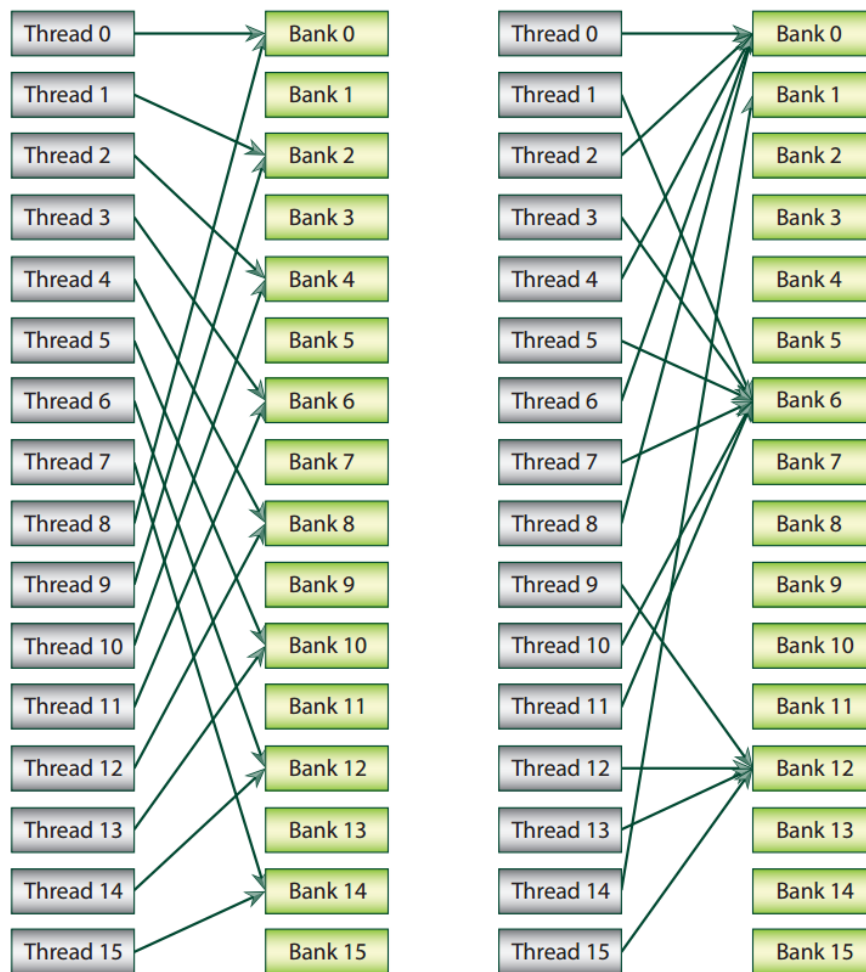
# Оптимизация доступа в память



OK



# Оптимизация доступа в память



FAIL

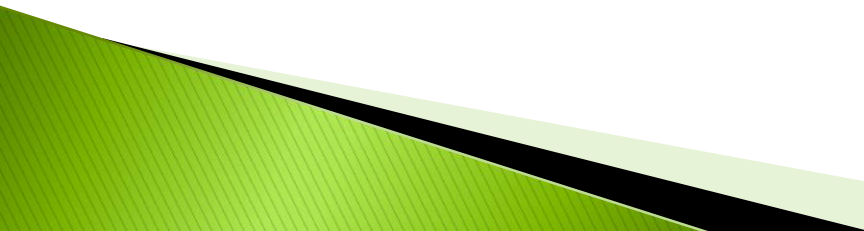
# Оптимизация доступа в память

```
// Нет конфликтов
shared_float buf [128];
float v = buf [baseIndex + threadIdx.x];


// Конфликт 4-го порядка
shared_char buf [128];
char v = buf [baseIndex + threadIdx.x];

// Конфликт 2-го порядка
shared_short buf [128];
short v = buf [baseIndex + threadIdx.x];
```

# Пример 1

- ▶ Транспонирование матрицы
  - ▶ Наивный алгоритм
  - ▶ Оптимизация – блочный алгоритм
  - ▶ При блочном транспонировании элементы вспомогательной матрицы расположенные в одном столбце будут храниться в одной банке памяти
  - ▶ Оптимизация – увеличить размер вспомогательной матрицы на 1
- 


# Pinned-память

- ▶ Память на хосте организована в виде страниц
  - ▶ GPU не может получать данные непосредственно со страниц памяти
  - ▶ Драйвер CUDA сначала выделяет специальную pinned-память на хосте, копирует туда данные из основной памяти хоста, потом забирает данные из pinned-памяти
- 

# Pinned-память

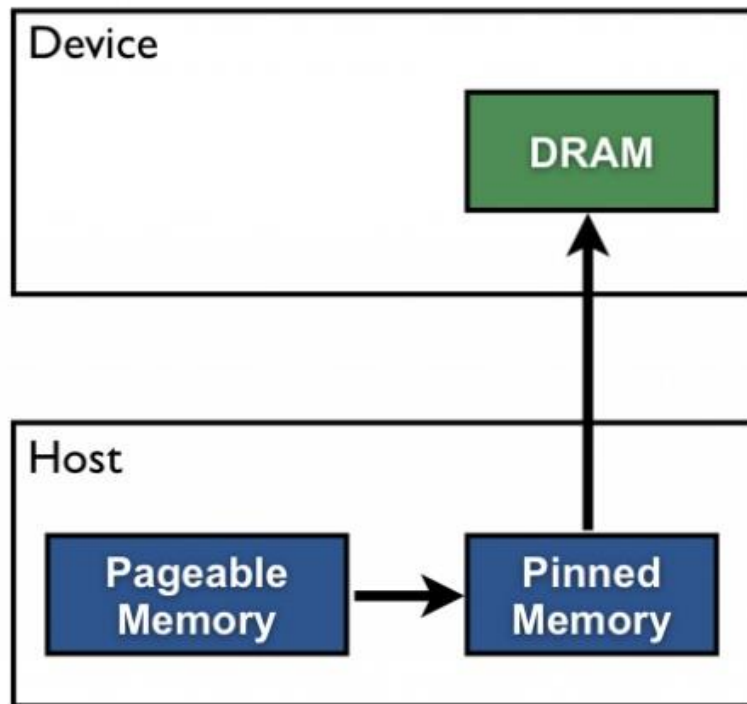
- ▶ Pinned-память фактически выполняет роль буфера между DRAM CPU и GPU
- ▶ Можно избежать ненужного копирования памяти на хосте из основной в pinned сразу размещая данные в pinned-памяти
- ▶ На устройствах с CC  $\geq 2.0$  данные размещенные на хосте в pinned-памяти доступны с GPU без явного перемещения при помощи указателей

# Pinned-память

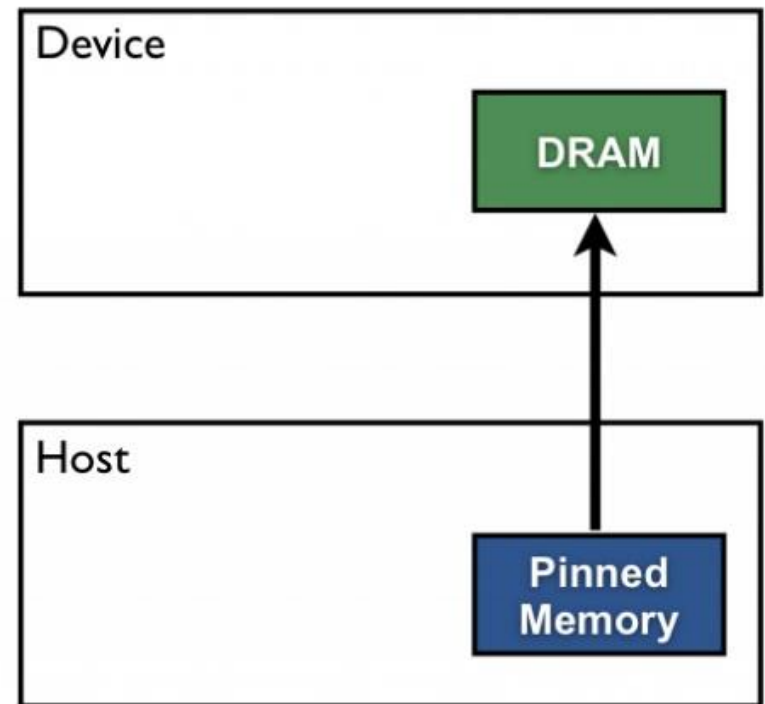
- ▶ Использование pinned-памяти – способ достичь максимального быстродействия при копировании данных между хостом и GPU
  - ▶ Выделение большого количества pinned-памяти способно серьезно замедлить работу ОС
  - ▶ Функцию следует использовать для организации активного буфера обмена между хостом и GPU
- 

# Pinned-память

## *Pageable Data Transfer*



## *Pinned Data Transfer*



# Pinned-память

```
//Выделить pinned-память на хосте  
cudaMallocHost((void**)&h_aPinned, bytes)  
  
//Выделить pinned-память на хосте с дополнительными параметрами  
cudaError_t cudaHostAlloc ( void** pHost, size_t size, unsigned  
int flag)  
  
//Освободить pinned-память на хосте  
cudaError_t cudaFreeHost( void* ptr)  
  
//Вызовы замещают собой malloc/free
```



# Пример 2

- ▶ Сравнение копирования данных с хоста на GPU и обратно в цикле
- ▶ Размер данных 1МБ


# Zero-copy

- ▶ Концепция взаимодействия хоста и GPU без лишнего копирования данных
- ▶ Этот прием использует pinned-память выделенную при помощи функции *cudaHostAlloc* с флагом *cudaHostAllocMapped*
- ▶ Позволяет использовать указатель на память хоста в ядрах GPU
- ▶ Особенно эффективен на системах с интегрированной графикой Nvidia Tegra

# Пример 3

- ▶ Zero copy с использованием буфера в pinned-памяти
- ▶ Эффективное решение для организации постоянного обмена данными между хостом и GPU

# Unified Virtual Addressing

- ▶ Начиная с CUDA 4.0 существует Unified Virtual Addressing (UVA) – общее виртуальное адресное пространство всех GPU и хоста
  - ▶ Адреса больше не пересекаются между собой
  - ▶ Доступ по указателям из GPU кода, вне зависимости от их фактического размещения
- 

# Unified Virtual Addressing

- ▶ Копирование данных между хостом и GPU с флагом *cudaMemcpyDefault* – направление определяется автоматически
- ▶ Копирование данных между двумя GPU напрямую (peer-to-peer), без копирования в хост-буфер
- ▶ Zero-copy – указатели возвращаемые *cudaHostAlloc* сразу доступны в ядрах GPU, без вызова *cudaHostGetDevicePointer*

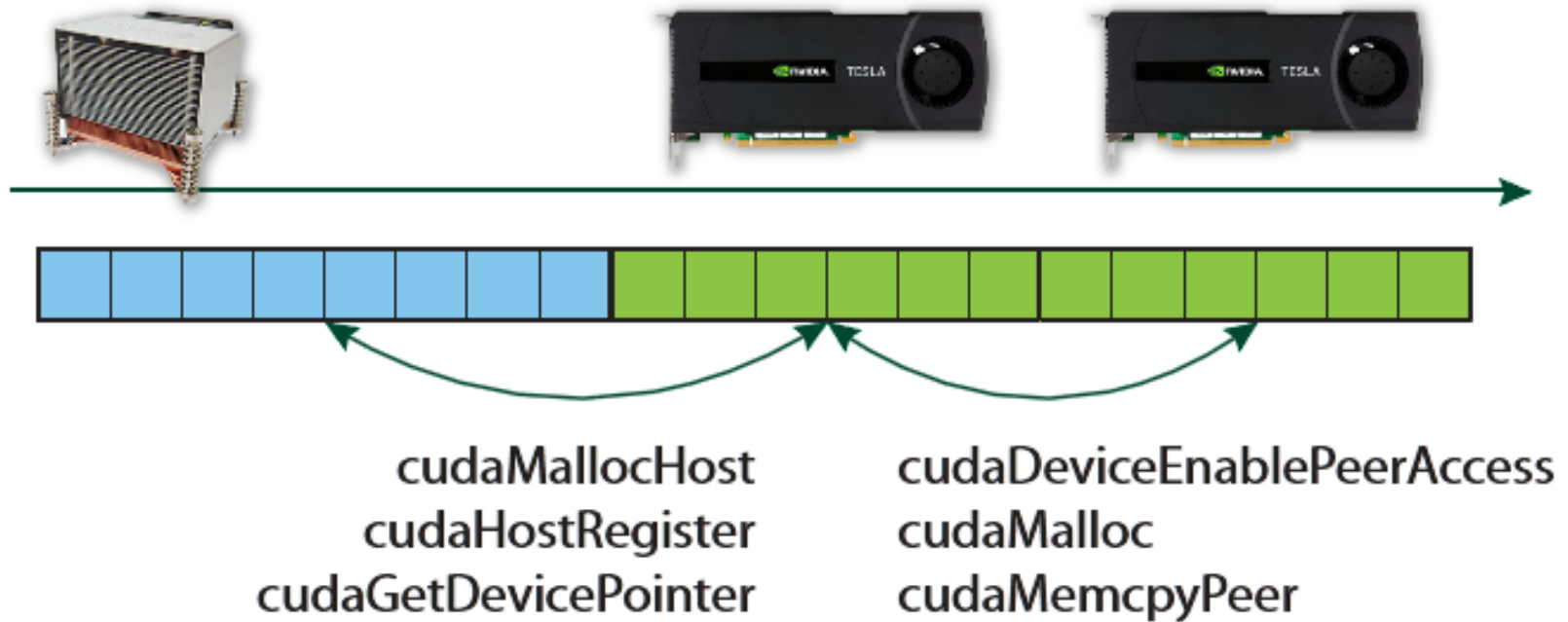
# Unified Virtual Addressing

```
//Делаем активным первый GPU
cudaSetDevice(gpuid_0);
//Включаем возможность P2P коммуникации
cudaDeviceEnablePeerAccess(gpuid_1, 0);


//Делаем активным первый GPU
cudaSetDevice(gpuid_1);
//Включаем возможность P2P коммуникации
cudaDeviceEnablePeerAccess(gpuid_0, 0);

//Копируем буфер с gpu_0 на gpu1 без участия хоста
cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault)
```

# Unified Virtual Addressing

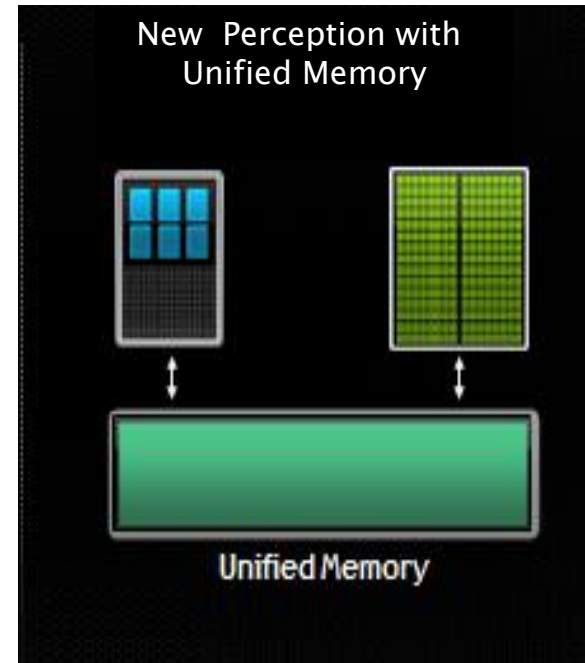
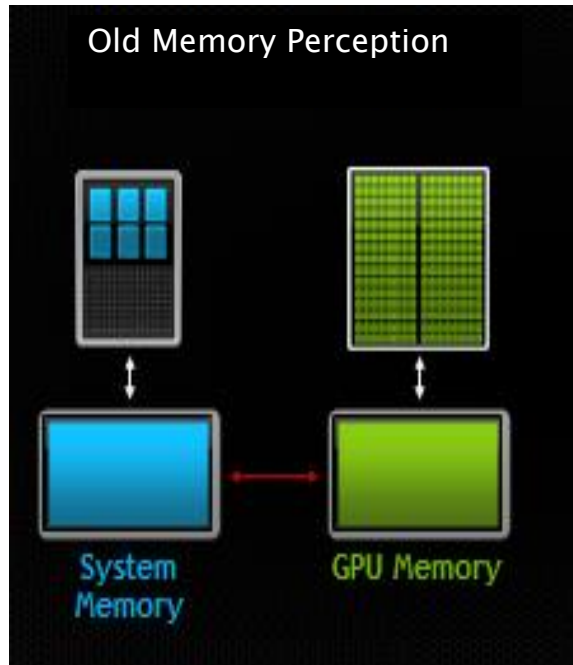


# Unified Memory

- ▶ Начиная с CUDA 6.0 и архитектуры GPU Kepler появилась Unified Memory – развитие концепции UVA
  - ▶ Одинаковые указатели для CPU и GPU памяти
  - ▶ CUDA автоматически перемещает данные между хостом и GPU
  - ▶ Создание кода значительно упрощается – нет нужды в ручном копировании
- 




# Unified Memory



# Пример 4

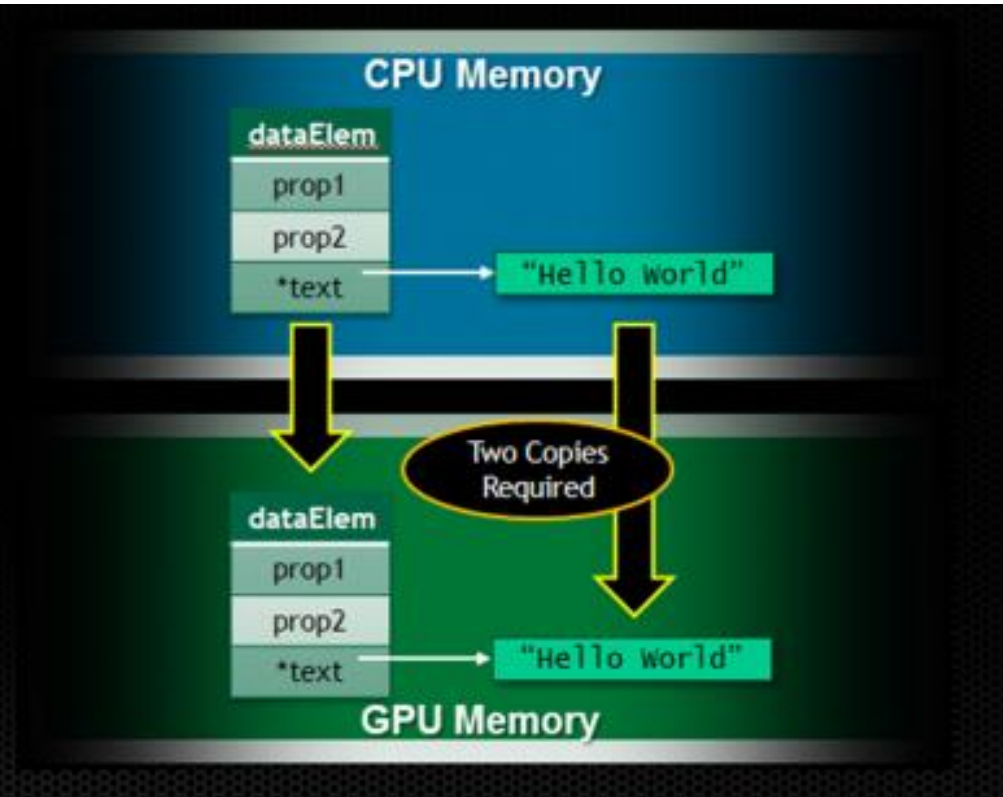
- ▶ Суммирование векторов с применением UV
- ▶ Сравнить с Примером 1 из Лекции 1
- ▶ Вопрос – почему такая разница в производительности?

# Unified Memory

- ▶ С UV исчезла необходимость в «глубоком копировании» при копировании структурированных данных (структур и объектов) содержащих указатели
  - ▶ «Глубокое копирование» – копирование как значения по указателю так и самого указателя
  - ▶ Это дает возможность легко передавать на GPU такие структуры данных как связные списки
- 

# Unified Memory


```
struct dataElem  
{  
    int prop1;  
    int prop2;  
    char *text;  
};
```



# Пример 5

- ▶ Передача структуры с указателем
- ▶ Новая возможность:
  - класс с конструктором копирования и переопределенными операторами new/delete
  - легкое копирование экземпляров класса на GPU и обратно

# Unified Memory

- ▶ В текущей версии память выделяется на GPU который установлен как активный
  - ▶ Миграция данных осуществляется на страницах (размер страниц устанавливается ОС)
  - ▶ Страницы которые помечены хостом(GPU) прозрачно перемещаются обратно на GPU(хост) в случае необходимости
- 

# Unified Memory

- ▶ Максимальный размер автоматически управляемой памяти – минимальная память совместимого GPU
- ▶ При запуске ядра у GPU эксклюзивный доступ к общей памяти

# Unified Memory

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    y = 20; // Ошибка доступа, ядро еще может работать
    cudaDeviceSynchronize();
    return 0;
}
```

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    cudaDeviceSynchronize();
    y = 20; // GPU неактивен, OK
    return 0;
}
```





Казанский  
федеральный  
университет

ВЫСШАЯ ШКОЛА  
информационных технологий  
и информационных систем

# Вопросы

[ekhramch@kpfu.ru](mailto:ekhramch@kpfu.ru)