




Казанский
федеральный
университет

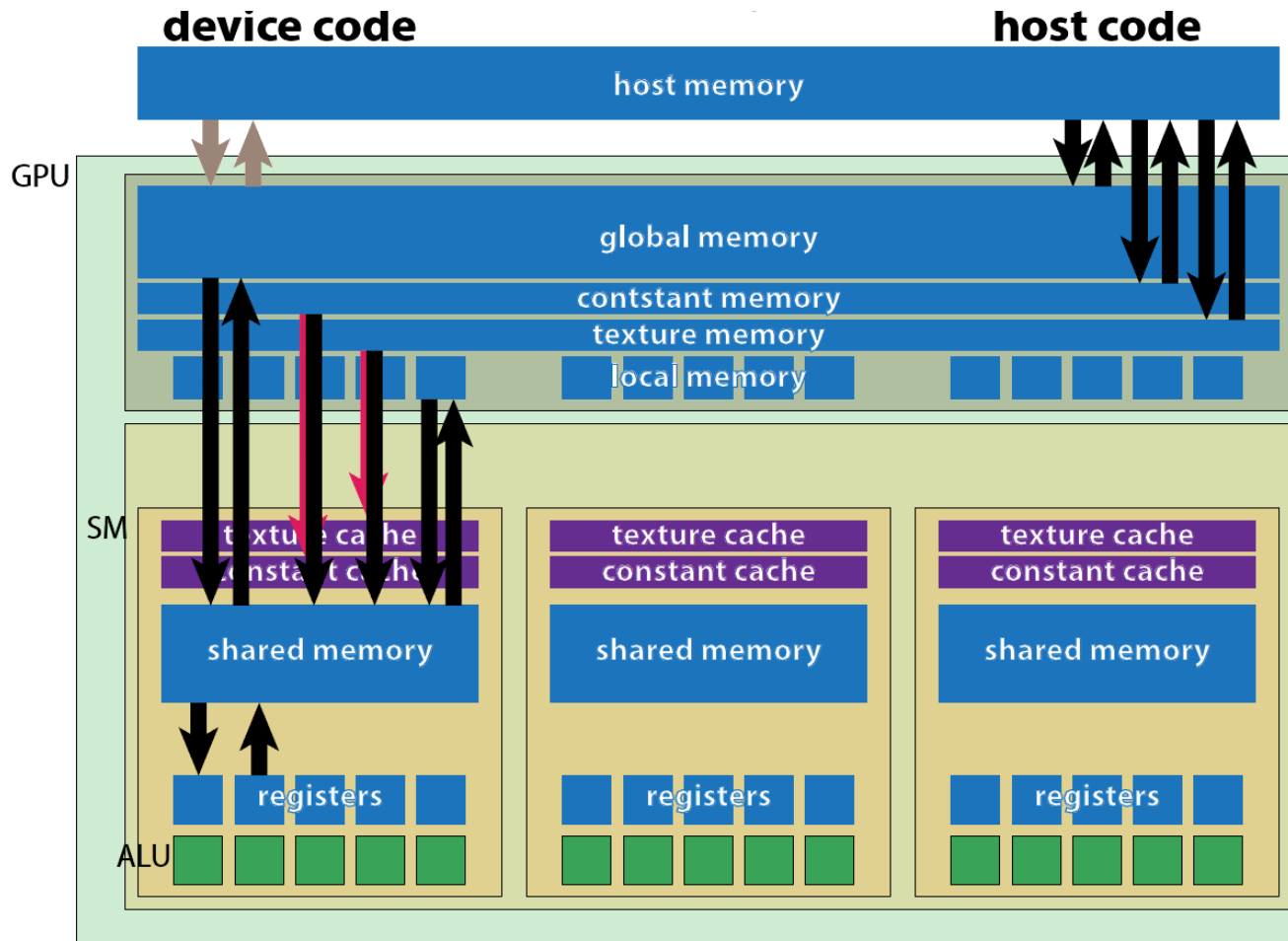
ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

Иерархия памяти CUDA

Память GPU

- ▶ На GPU существует несколько видов памяти, отличающихся расположением и скоростью доступа
 - ▶ Программист может вручную управлять размещением данных в той или иной памяти
 - ▶ Эффективное использование памяти – залог быстрого кода
- 


Память GPU



Память GPU

Тип памяти	Доступ	Видимость	Скорость	Расположение
Registers	R/W	Thread	Высокая	SM
Local	R/W	Thread	Низкая	DRAM GPU
Shared	R/W	Block	Высокая	SM
Global	R/W	Grid	Низкая	DRAM GPU
Constant	R/O	Grid	Высокая	DRAM GPU
Texture	R/O	Grid	Высокая	DRAM GPU


Регистры

- ▶ Распределяются между нитями блока на этапе компиляции
 - ▶ Доступ к регистрам других нитей запрещен
 - ▶ Расположены в мультипроцессоре – высокая скорость доступа
 - ▶ Размер – порядка нескольких КБ на нить
 - ▶ В регистрах размещаются локальные переменные
- 

Локальная память

- ▶ Размещена в DRAM GPU – низкая скорость доступа
- ▶ В локальную память попадают
 - Объединения (unions)
 - Динамические массивы
 - Структуры и массивы большого размера
- ▶ Все переменные, если ядро использовало всю память регистров

Глобальная память

- ▶ Память DRAM GPU
 - ▶ Низкая скорость доступа
 - ▶ Выделяется с хоста через функции CUDA API
 - ▶ Может использоваться всеми потоками сетки
 - ▶ Основная память для данных копируемых с хоста
- 

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);
```

//Копирование данных, 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`:
 - `cudaMemcpyHostToHost`: Host -> Host
 - `cudaMemcpyHostToDevice`: Host -> Device
 - `cudaMemcpyDeviceToHost`: Device -> Host
 - `cudaMemcpyDeviceToDevice` : Device -> Device

Глобальная память

- ▶ Для хранения адресов памяти GPU используются обычные указатели
- ▶ Для таких указателей корректна адресная арифметика
- ▶ Адресные пространства CPU и GPU – это разные адресные пространства*
- ▶ Память, выделенная на одном GPU некорректна по отношению к другому GPU

Кэширование

- ▶ Начиная с архитектуры Fermi кэшируется – L1 и L2 кэши
- ▶ Кеш L1 находится на каждом SM на одном кристалле с разделяемой памятью
- ▶ Кеш L2 общий для всех нитей
- ▶ Программист может задать конфигурацию L1/разделяемая память

Кэширование

//48 КБ кэш, 16 КБ L1

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

//48 КБ L1, 16 КБ кэш

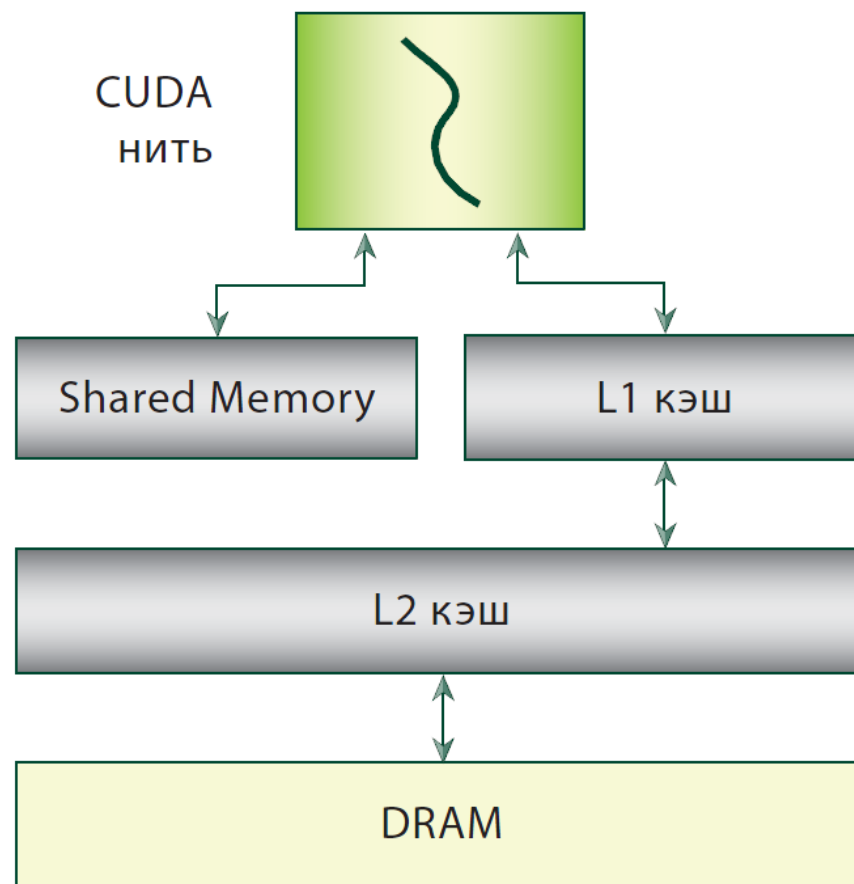
```
cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferL1);
```

//Используется текущий контекст


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



Кэширование



Coalescing

- ▶ Coalescing – объединение запросов варпа к непрерывному блоку глобальной памяти
 - ▶ С архитектуры Fermi все запросы варпа к глобальной памяти объединяются в один
 - ▶ Если нити варпа обращаются к разным областям памяти, запросы не объединяются
 - ▶ Решение – использовать разделяемую память для создания временного буфера
- 


Pinned-память

- ▶ GPU не может получать данные непосредственно со страниц памяти хоста
- ▶ Драйвер CUDA
 - Выделяет специальную pinned-память на хосте
 - Копирует туда данные из основной памяти хоста
 - Забирает данные из pinned-памяти

Pinned-память

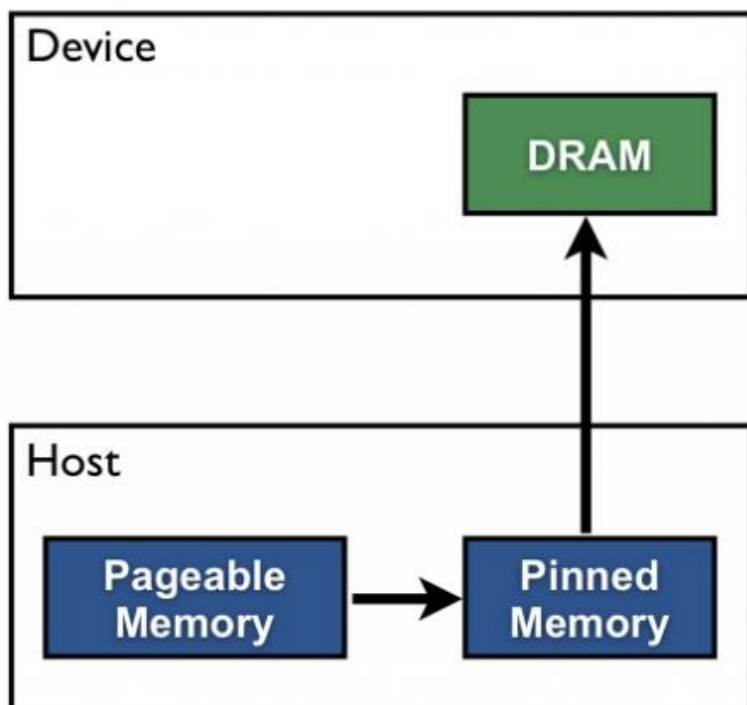
- ▶ Pinned-память фактически выполняет роль буфера между DRAM CPU и GPU
- ▶ Можно избежать ненужного копирования памяти сразу размещая данные в pinned-памяти
- ▶ На устройствах с CC ≥ 2.0 данные размещенные на хосте в pinned-памяти доступны с GPU без явного перемещения

Pinned-память

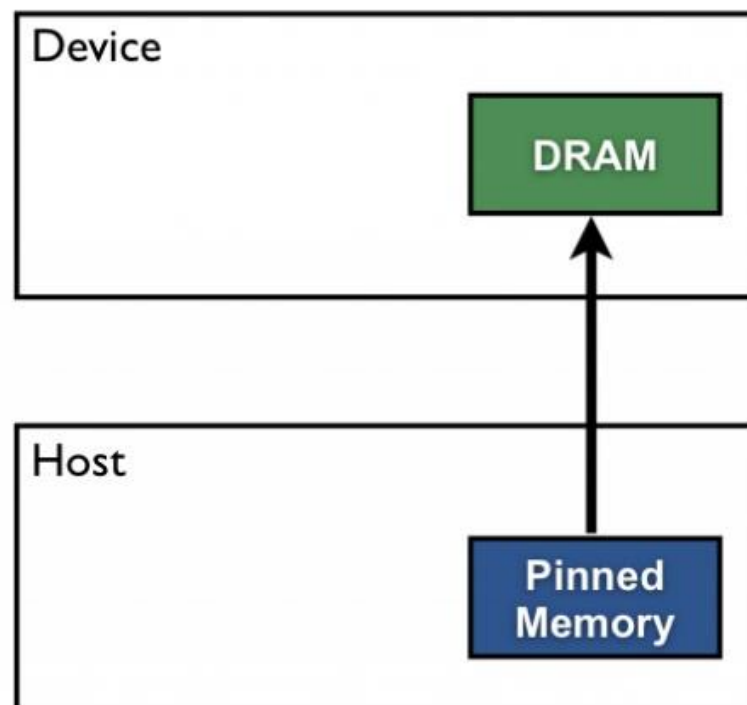
- ▶ Использование pinned-памяти дает максимальное быстродействие при копировании данных между хостом и GPU
 - ▶ Выделение большого количества pinned-памяти может замедлить работу ОС
 - ▶ Pinned-память подходит для создания быстрого буфера небольшого размера между хостом и девайсом
- 

Pinned-память

Pageable Data Transfer



Pinned Data Transfer




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

Zero-copy

- ▶ Сравнение копирования данных с хоста на GPU и обратно в цикле
- ▶ Размер данных 1МБ
- ▶ Zero copy с использованием буфера в pinned-памяти

Разделяемая память

- ▶ Расположена в SM, выделяется на уровне блоков
 - ▶ Каждый блок получает в распоряжение одинаковое количество разделяемой памяти
 - ▶ Размер до 96 КБайт
 - ▶ Высокая скорость доступа – как у регистров
 - ▶ Может использоваться всеми потоками блока
- 

Разделяемая память

- ▶ Переменные в разделяемой памяти указываются со спецификатором `__shared__`
- ▶ Массивы в разделяемой памяти могут иметь как статический, так и динамический размер
- ▶ В последнем случае размер такого массива в байтах передается в качестве третьего параметра вызова ядра

Разделяемая память

```
__shared__ float buf[1024]; //статический массив в разделяемой  
памяти
```


```
extern __shared__ float buf[]; //динамический массив в разделяемой  
памяти
```

```
...  
kernel<<<num_threads, num_block, 1024*sizeof(float)>>>(...);
```

Разделяемая память

- ▶ Пример – транспонирование матрицы
- ▶ Двумерная сетка нитей и блоков
- ▶ Разделяемая память – двумерный буфер


Конфликт банков

- ▶ Разделяемая память разбита на 32 банка
 - ▶ Каждый банк выполняет одно чтение или запись 32-битного слова
 - ▶ Подряд идущие 32-битные слова попадают в подряд идущие банки
 - ▶ В случае обращения всего варпа к подряд идущим словам, уменьшаются оверхэд чтения памяти
- 

Конфликт банков

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

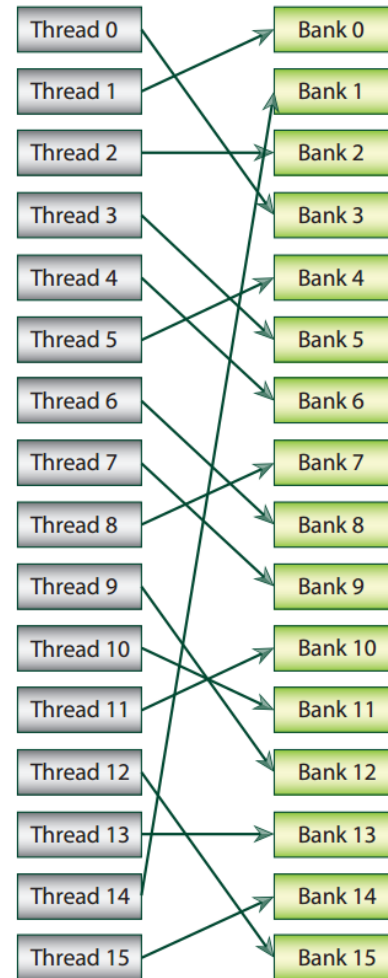
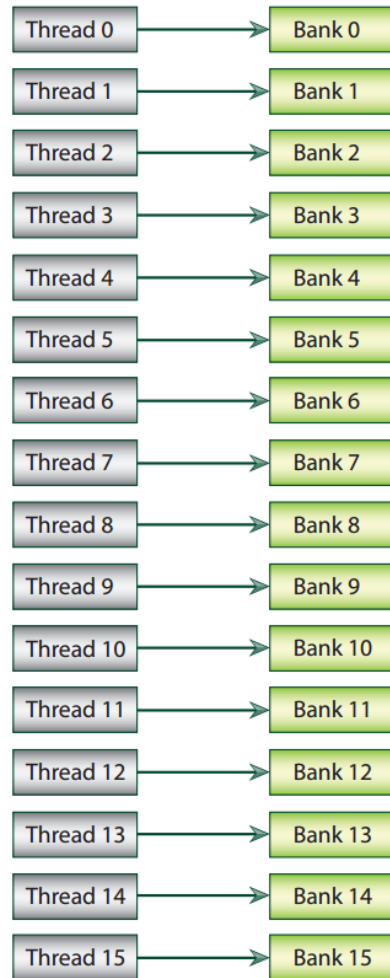
Конфликт банков

- ▶ Обращения к одному банку могут быть выполнены только последовательно
 - ▶ Конфликт банков – одновременный запрос данных из одного банка несколькими нитями
 - ▶ Порядок конфликта – максимальное число обращений в один банк
- 

Конфликт банков

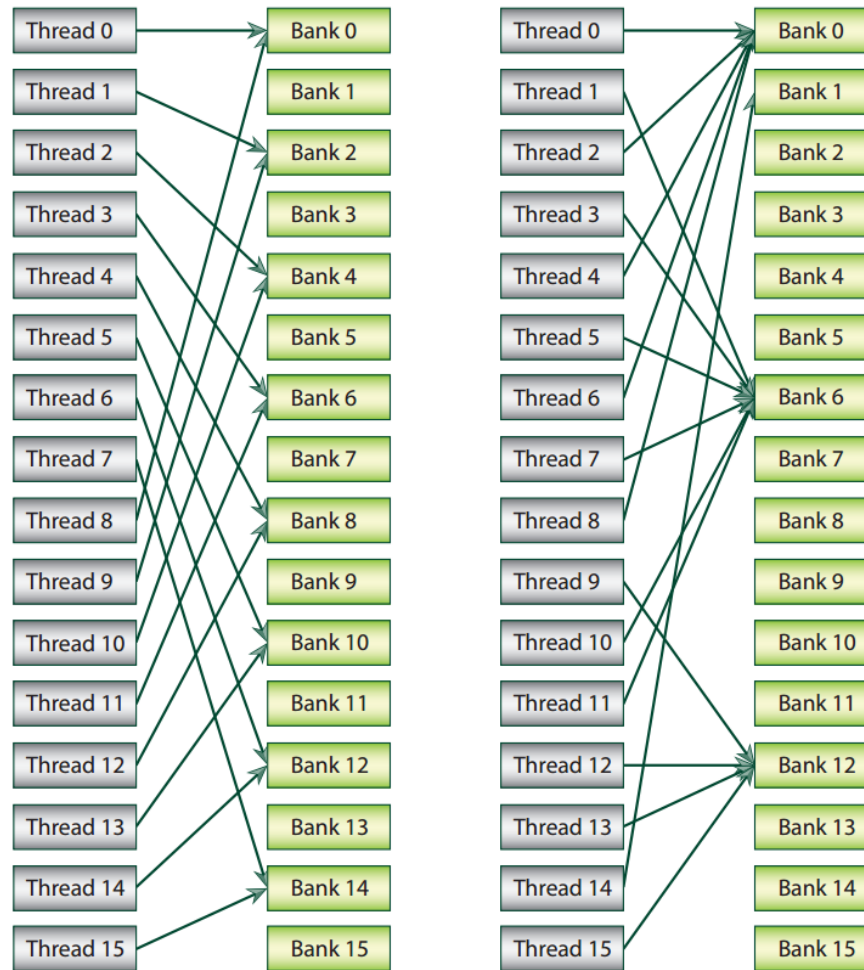
- ▶ Конфликт второго порядка для одного банка – двукратное снижение скорости доступа к разделяемой памяти
- ▶ Особый случай – обращение всех 32 нитей варпа к одному и тому же элементу одного банка, конфликта не возникает

Конфликт банков



OK

Конфликт банков



FAIL

Конфликт банков

```
// 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];
```

Конфликт банков

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

Константная память

- ▶ Расположена в DRAM GPU
- ▶ Доступна всем нитям сетки только на чтение, запись в эту память производится с хоста
- ▶ Переменные в этой памяти объявляются со спецификатором `__constant__`
- ▶ Объем константной памяти – 64КБ на блок

Константная память

- ▶ Все нити варпа должны обращаться к одному и тому же адресу в константной памяти
- ▶ Если нити варпа обращаются к разным адресам, их запросы становятся последовательными и обрабатываются по очереди


Константная память

- ▶ Высокая скорость доступа
 - Один запрос в константную память транслируется на полуварп – $1/16$ от всех полного числа запросов
 - Неизменность данных позволяет обеспечить агрессивное кэширование
 - Скорость доступа как у регистров

Константная память

- ▶ Пример – применение матрицы поворота на угол `angle` ко множеству точек плоскости
- ▶ X и Y координаты точек хранятся в отдельных плоских массивах для улучшения паттерна доступа в память
- ▶ Матрица поворота – в константной памяти

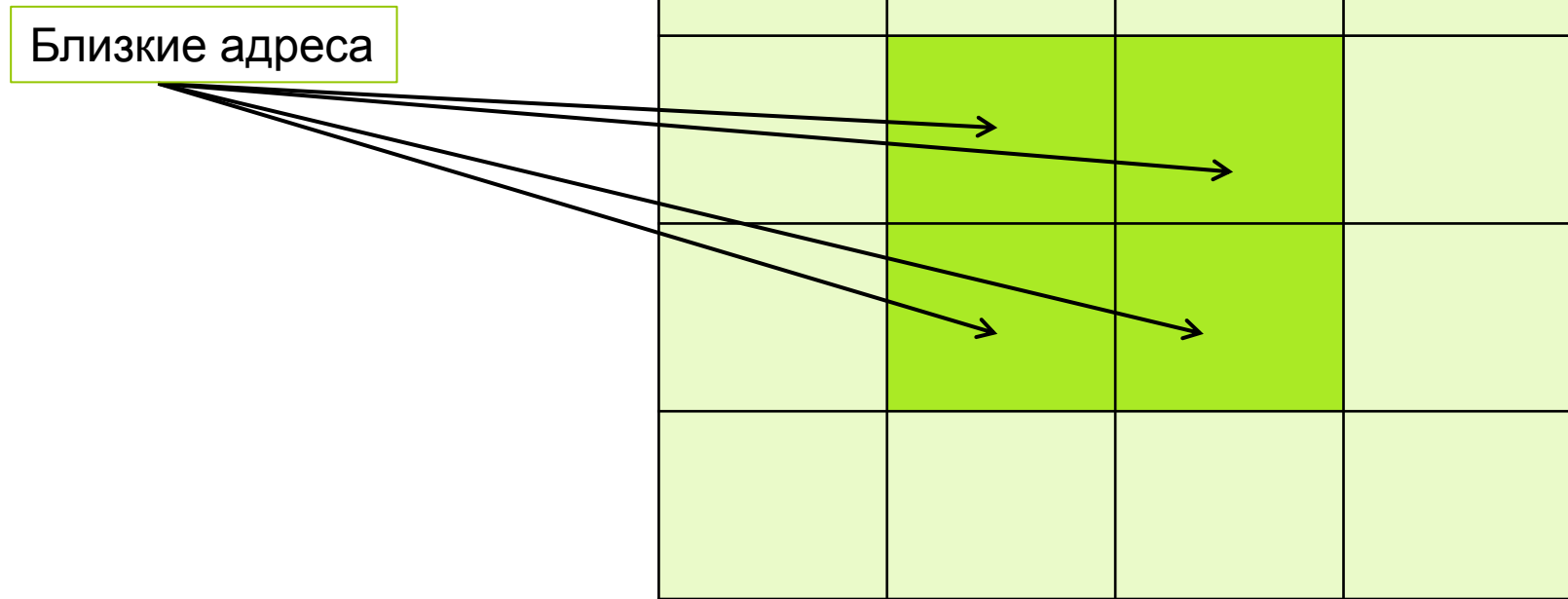
Текстурная память

- ▶ Расположена в DRAM GPU
 - ▶ Обладает независимым кэшем – высокая скорость доступа
 - ▶ Доступна всем нитям сетки только на чтение, запись в эту память производится с хоста
 - ▶ Объем равен свободному объему DRAM
- 

Текстурная память

- ▶ Предназначены для графических операций, операции с данными как с текстурами
- ▶ Хорошо работают на данных с высокой пространственной локальностью – нити одного варпа обращаются к «близким» адресам
- ▶ Наиболее эффективны на 2D массивах данных


Текстурная память



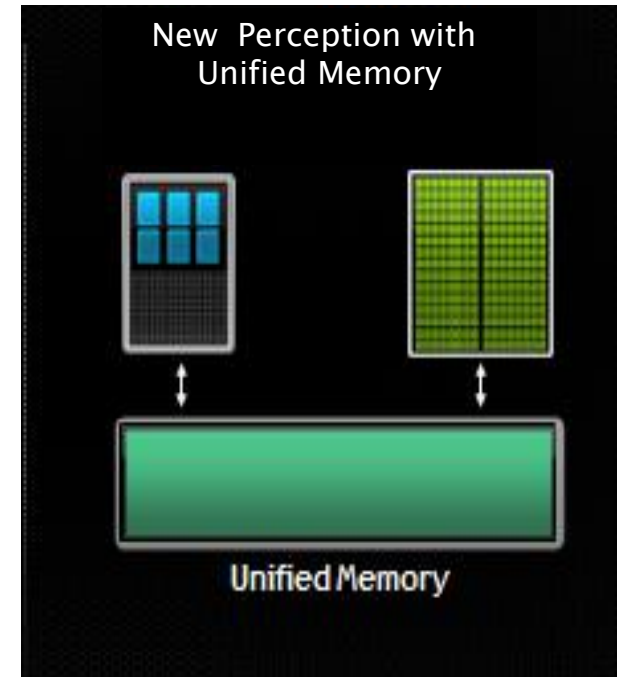
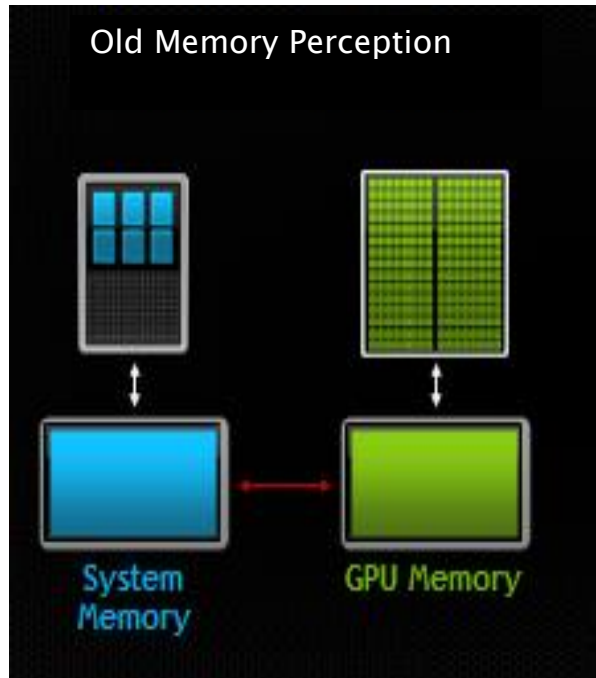
Unified Virtual Addressing

- ▶ 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
- ▶ Копирование данных между хостом и GPU с флагом `cudaMemcpyDefault`

Unified Memory

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

Unified Memory



Unified Memory

- ▶ Новый спецификатор памяти `__managed__` – позволяет создавать глобальные переменные видимые и на хосте и на GPU
- ▶ Внимание: при каждом запуске ядра происходит копирование необходимых данных с хоста на девайс и обратно
- ▶ Следует иметь это в виду при запуске ядер в циклах

Unified Memory

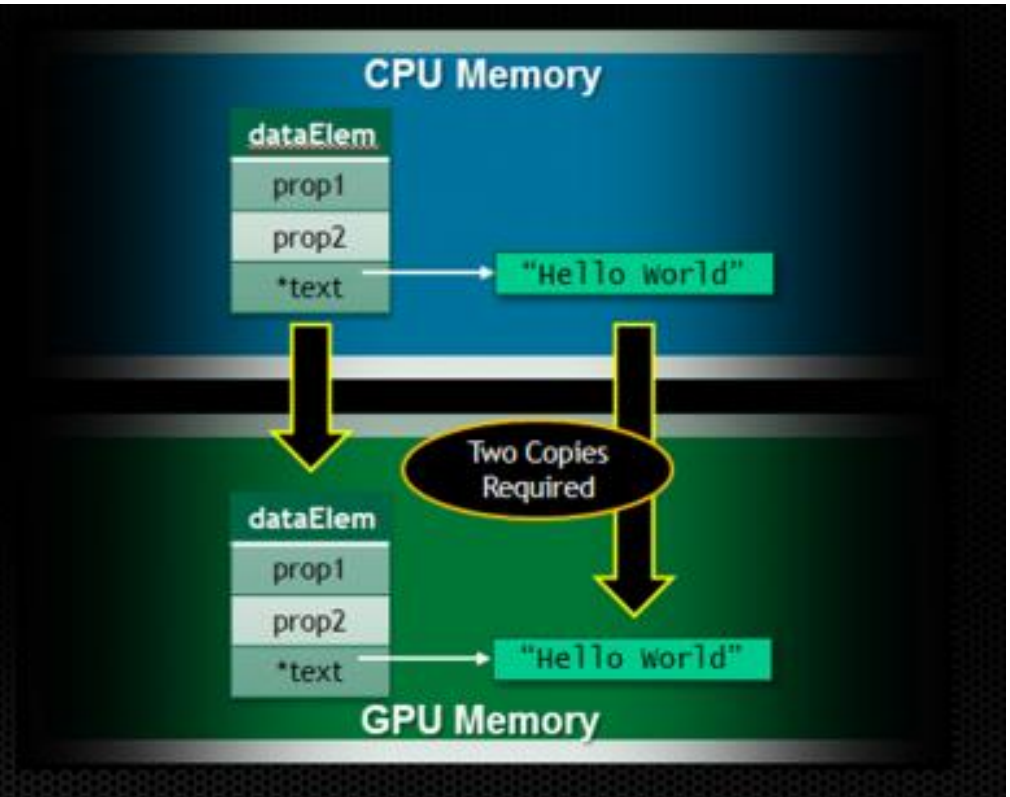
- ▶ Пример – поворот вектора переписанный под UM
- ▶ Обратите внимание на увеличившееся время выполнения ядер – неявное копирование памяти

Unified Memory

- ▶ С UM исчезла необходимость в глубоком копировании
- ▶ 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;  
};
```




Unified Memory

```
void launch(dataElem *elem)
{
    dataElem *d_elem;
    char *d_name;
    int namelen = strlen(elem->name) + 1;

    // Выделяем память под структуру и под поле name
    cudaMalloc(&d_elem, sizeof(dataElem));
    cudaMalloc(&d_name, namelen);

    // Отдельно копируем структуру, значение поля name и значение указателя
    cudaMemcpy(d_elem, elem, sizeof(dataElem), cudaMemcpyHostToDevice);
    cudaMemcpy(d_name, elem->name, namelen, cudaMemcpyHostToDevice);
    cudaMemcpy(&(d_elem->name), &d_name, sizeof(char*), cudaMemcpyHostToDevice);
    kernel<<< ... >>>(d_elem);
}
```

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



Unified Memory

```
class Managed{
public:
    void *operator new(size_t len){
        void *ptr;
        cudaMallocManaged(&ptr, len);
        cudaDeviceSynchronize();
        return ptr;
    }
    void operator delete(void *ptr){
        cudaDeviceSynchronize();
        cudaFree(ptr);
    }
};
```

```
//ссылка по указателю
class String : public Managed{
    int length;
    char *data;

public:
    //конструктор копирования
    String (const String &s){
        length = s.length;
        cudaMallocManaged(&data, length);
        memcpy(data, s.data, length);
    }
};
```

```
class dataElem : public Managed{
public:
    int prop1;
    int prop2;
    String name;
};
```

Обязательное
наследование от
Managed

Unified Memory

```
__global__ void kernel_by_point(dataElem *data){ ... } //по указателю

__global__ void ker_by_ref(dataElem &data){ ... } //по ссылке

__global__ void kernel_by_val(dataElem data){ ... } //по значению

int main(){
    dataElem *data = new dataElem;

    ...

    kernel_by_point<<<1,1>>>(data); //передача параметра по указателю

    kernel_by_ref<<<1,1>>>(*data); //передача параметра по ссылке

    kernel_by_val<<<1,1>>>(*data); //передача параметра по значению – копия!
}
```



Казанский
федеральный
университет

ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

Вопросы

ekhramch@kpfu.ru