




Казанский
федеральный
университет

ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

Многопоточность Профилирование Библиотеки CUDA

Эдуард Храмченков

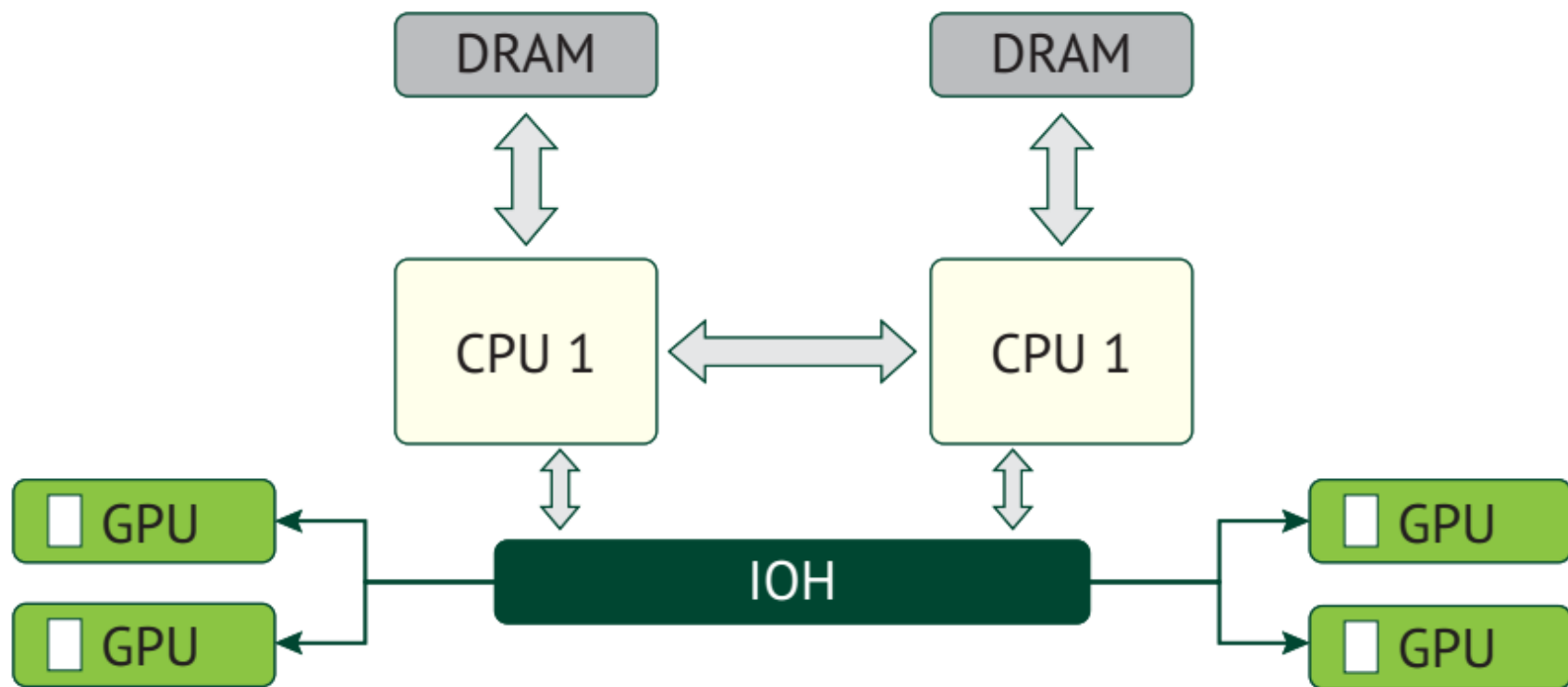
Многокарточные системы

- ▶ Скорость обмена между GPU и хостом зависит от количества линий обмена данными в шине
 - ▶ Например если 2 слота PCI-Ex16 и PCI-Ex8 висят на одной шине, то 2 GPU в этих слотах будут работать как PCI-Ex8
 - ▶ Для полноценного использования нескольких GPU требуется материнская плата с несколькими отдельными шинами и слотами PCI-Ex16
- 

Многокарточные системы

- ▶ Для кластерной системы с узлами содержащими GPU характерны дополнительные расходы на передачу данных из хоста одного узла в GPU другого
- ▶ CUDA работает только в контексте GPU одного узла, поэтому для кластерного приложения необходимо дополнительно использовать другие программные модели

Многокарточные системы




Единицы исполнения

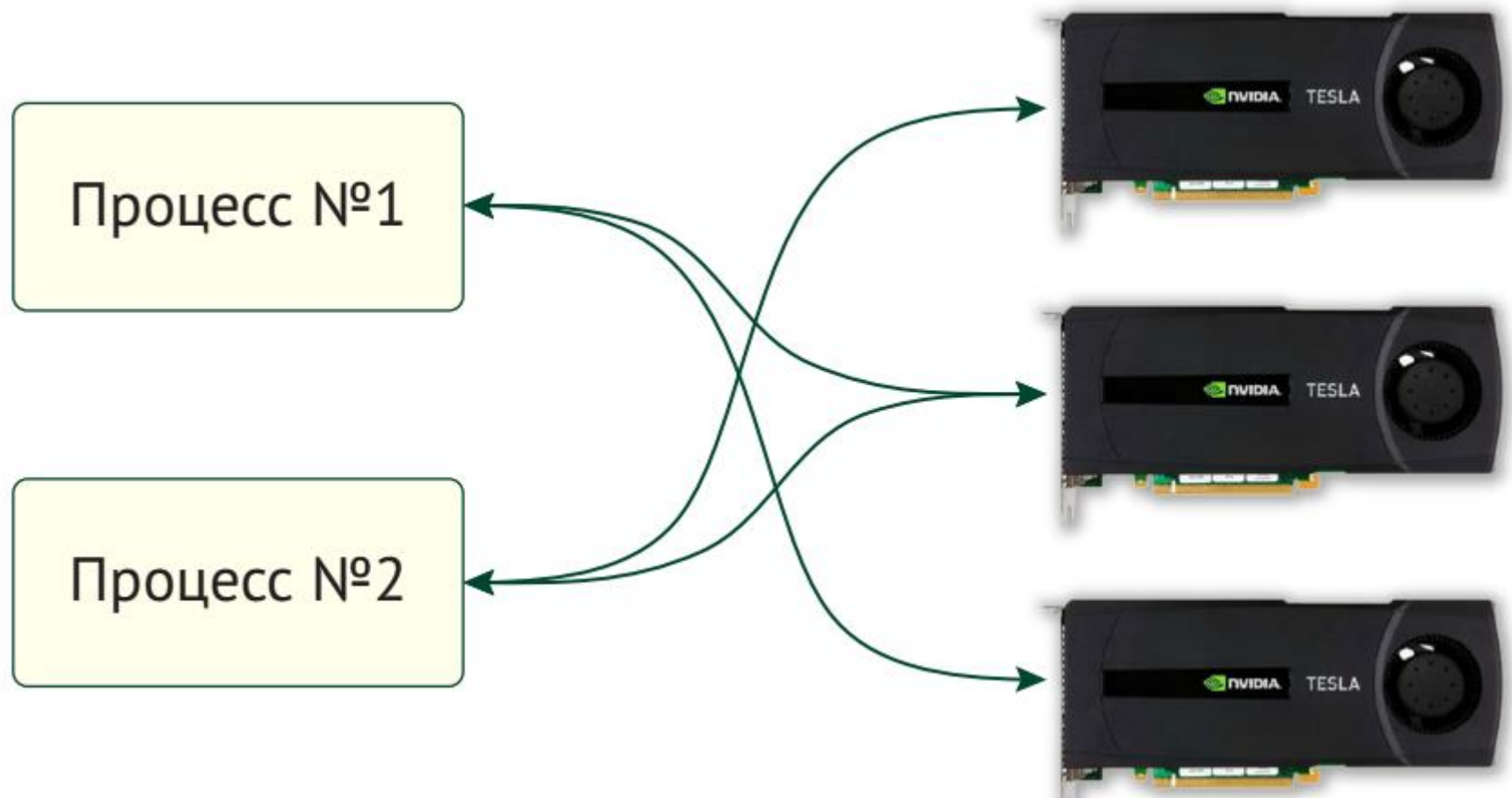
► Процесс

- Наличие контекста в системе
- Собственное адресное пространство
- Запрет на доступ в память другого процесса
- Управление процессами – через системные вызовы

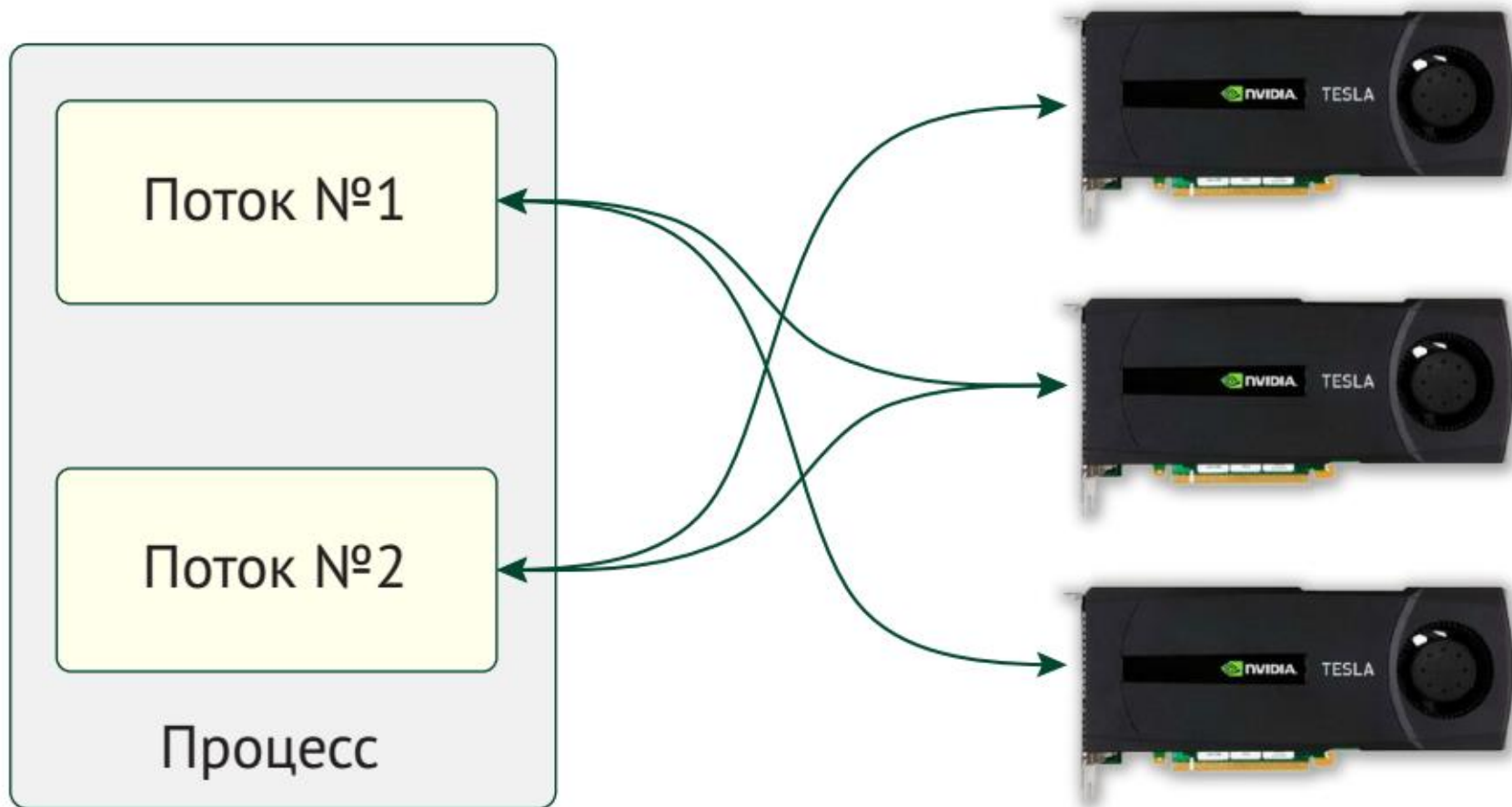
► Поток

- Общий доступ к памяти процесса-родителя
 - Легче и быстрее чем процессы
 - Могут управляться на уровне приложения
- 

Единицы исполнения




Единицы исполнения



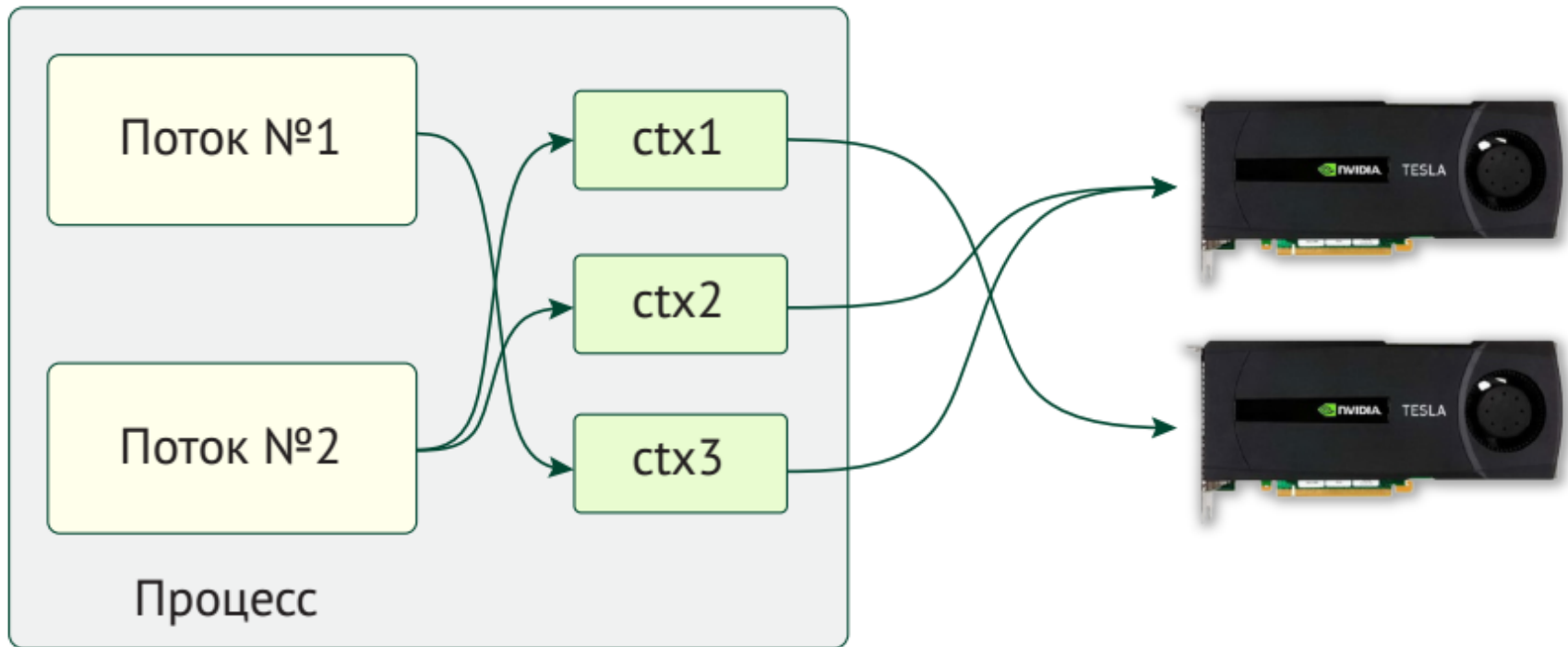
Контекст устройства CUDA

- ▶ Контекст – контейнер с управляющей информацией, характеризующей состояние конкретного GPU
- ▶ Контекст неявно создается при первом вызове функции CUDA API в конкретном процессе/потоке
- ▶ Одно устройство может иметь несколько контекстов

Контекст устройства CUDA

- ▶ Процессы/потоки могут управлять GPU только через контекст
 - ▶ Контексты можно создавать, удалять, выбирать активный из существующих
 - ▶ В каждый момент времени в каждом потоке хоста может быть только один активный контекст
 - ▶ Управление контекстами может быть организовано средствами CUDA API
- 

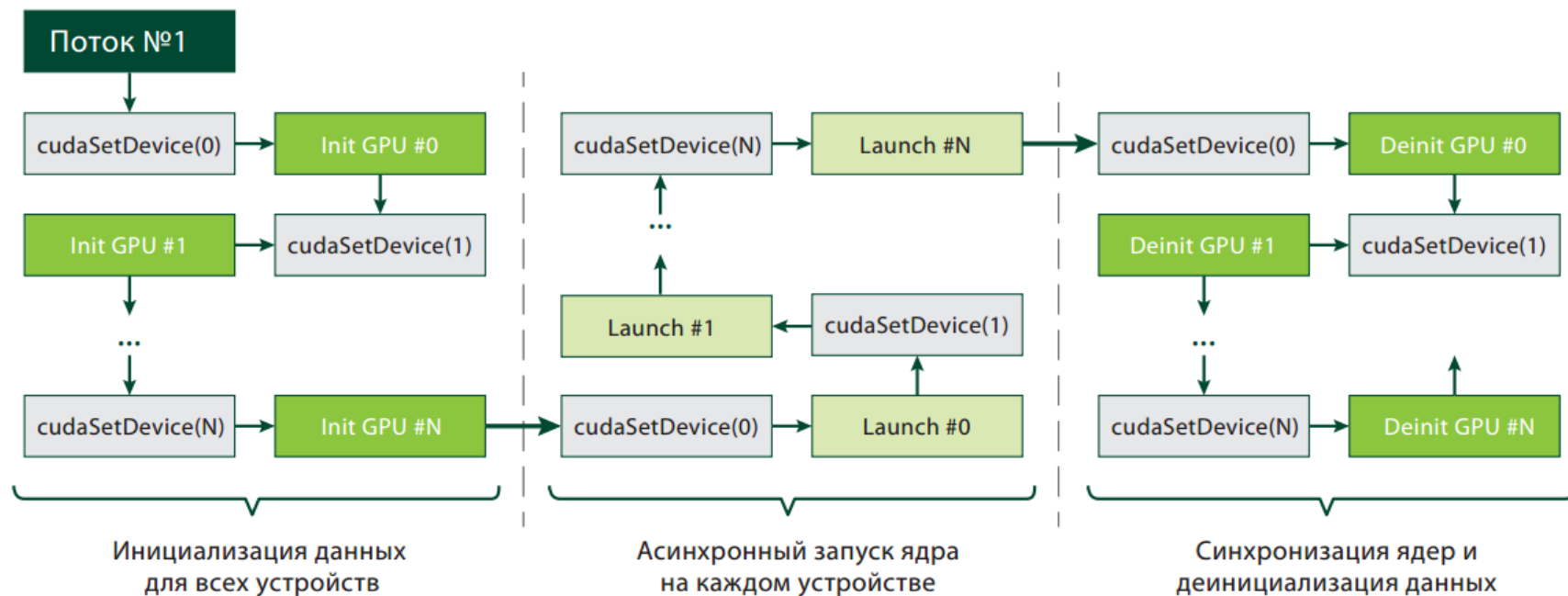
Контекст устройства CUDA



Пример 0

- ▶ Типичная схема работы с несколькими GPU
 - Цикл по устройствам для выделения памяти и инициализации данных
 - Цикл по устройствам для запуска ядер – запуск асинхронный, возврат управления происходит до окончания работы ядра
 - Цикл для синхронизации устройств, обработки данных и освобождения памяти

Пример 0



MPI + CUDA

- ▶ Наиболее общим способом организации работы нескольких GPU при помощи нескольких процессов является MPI программа
- ▶ Для современных версий MPI поддерживается передача адресов GPU через вызовы MPI_Send/MPI_Recv

Пример 1

- ▶ Предыдущий пример переписанный под процессы MPI


Пример 2

- ▶ Обмен данными GPU между потоками


Пример 3

- ▶ ПримерNº0 переписанный под потоки OpenMP

CUDA Streams

- ▶ Поток(очередь) команд для запуска ядер и обмена данными между GPU и хостом
 - ▶ Управляются пользователем через CUDA API
 - ▶ Stream – программная абстракция не связанная с системными потоками исполнения
 - ▶ Каждый stream определяет идентификатор – аргумент для команд CUDA
- 

CUDA Streams

- ▶ Команды с одним значением stream выполняются последовательно
 - ▶ Команды с разными значениями stream выполняются независимо
 - ▶ По умолчанию все команды работают со значением stream = 0
 - ▶ Начиная с CUDA 7.0 есть возможность задать каждому потоку/процессу исполнения свой default stream
- 

Пример 4

- ▶ Многопоточность через CUDA streams
- ▶ Небольшие ядра асинхронно запускаются в разных streams
- ▶ Каждое ядро занимает 1 SM

CUDA Streams

- ▶ Streams могут использоваться для оптимизации доступа в память
- ▶ Если копирование данных хост-GPU, GPU-хост и работа ядра занимают примерно одинаковое время, то можно получить выигрыш в производительности
- ▶ Память должна быть pinned типа


CUDA Streams

```
for (int i = 0; i < nStreams; ++i)
{
    int of = i * streamSize;

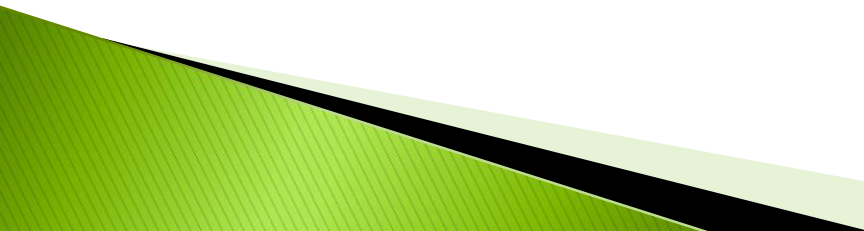
    cudaMemcpyAsync(&d_a[of], &a[of], nB, cudaMemcpyHostToDevice, stream[i]);

    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);

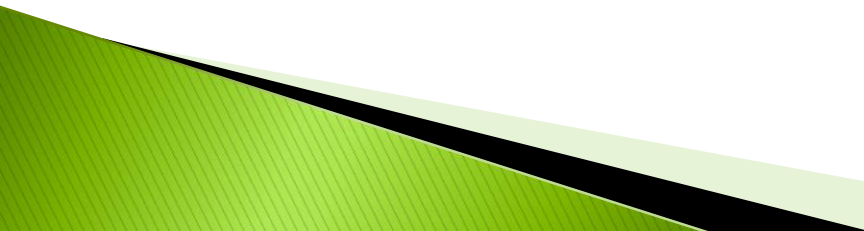
    cudaMemcpyAsync(&a[of], &d_a[of], nB, cudaMemcpyDeviceToHost, stream[i]);
}
```



CUDA events

- ▶ Событие – объект CUDA API для измерения времени выполнения операций CUDA
 - ▶ Тип объекта – `cudaEvent_t`
 - ▶ Функции CUDA API позволяют
 - Создавать и уничтожать события
 - Выделять начало и конец профилируемого участка кода
 - Проверять/ожидать наступления события
 - Замерять время выполнения
- 

CUDA events

- ▶ Событие – объект CUDA API для измерения времени выполнения операций CUDA
 - ▶ Тип объекта – `cudaEvent_t`
 - ▶ Функции CUDA API позволяют
 - Создавать и уничтожать события
 - Выделять начало и конец профилируемого участка кода
 - Проверять/ожидать наступления события
 - Замерять время выполнения
- 

Пример 5

- ▶ Измерение времени выполнения через CUDA events

Профилировщик CUDA

- ▶ В CUDA есть встроенный профилировщик
- ▶ Активация через переменную окружения `COMPUTE_PROFILE`
- ▶ Результат профилирования записывается в log-файл
- ▶ Кроме того, можно воспользоваться командой `nvprof`:
 - `$nvprof [options] [application] [application args]`

Дебаггер CUDA

- ▶ Дебаггер CUDA – `cuda-gdb`
- ▶ Это расширение стандартного `dbg` для отладки CUDA программ
- ▶ Для отладки необходимо иметь хотя бы 1 свободный GPU (занятый выводом графики GPU свободным не считается)
- ▶ При компиляции указываются ключи `-g` (для `gdb`) и `-G` (для `cuda-gdb`)

Библиотеки CUDA

► Библиотеки Nvidia:

- CUBLAS – основные операции линейной алгебры
- CUSPARSE – основные операции линейной алгебры для разреженных матриц
- CUFFT – прямое и обратное быстрое ДПФ
- CURAND – генерация псевдослучайных чисел


► Независимые библиотеки

- CUSP, VexCL, MAGMA, openCV, etc

Пример 6

- ▶ Перемножение матриц в CUBLAS

Thrust

- ▶ Библиотека обработки данных на GPU
 - ▶ Интерфейс схож с STL
 - ▶ Содержит реализацию базовых алгоритмов
 - ▶ Можно использовать как «блоки» для построения более сложного кода
 - ▶ Предоставляет высокий уровень абстракции, скрывая низкоуровневую реализацию
 - ▶ Не позволяет управлять нитями, блоками и памятью
- 

Пример 7

- ▶ Пример функции от двух векторов, аналогично Примеру №1 из Лекции №1
- ▶ В *thrust::transform* можно использовать:
 - Библиотечные функторы (*thrust::plus<T>*, etc)
 - Функтор вида *func()* обернутый в структуру/класс
 - Placeholders
 - Лямбда-функции с ключом компиляции *--expt-extended-lambda* (с CUDA 7.5)
- ▶ Сравнить производительность

Thrust

- ▶ Thrust поддерживает унарные, бинарные, тернарные трансформации и трансформации общего вида
- ▶ Конструкция *zip_iterator* преобразует n входных последовательностей в n -местный кортеж (*tuple*)
- ▶ Тогда *thrust::transform* становится унарным преобразованием над кортежем

Пример 7

- ▶ Модификация Примера №7 с использованием кортежей

Пример 8

- ▶ Редукция с использованием Thrust
- ▶ Оператор редукции:
 - По умолчанию суммирование
 - Может содержать другие встроенные функторы `thrust` (`thrust::max_element`, `thrust::inner_product`, etc)
 - Функторы определенные пользователем
- ▶ Список доступных функторов и подробное описание – RTFM

Пример 9

- ▶ Редукция с трансформацией для вычисления нормы

Thrust

► Другие алгоритмы Thrust

- Префиксные суммы

thrust::inclusive_scan/thrust::exclusive_scan

- Сортировка *thrust::sort/thrust::stable_sort*
(аналогично STL версиям)

- Сортировка по ключам

thrust::sort_by_key/thrust::stable_sort_by_key
(аналогично STL версиям)

- Переупорядочивание *copy_if, partition, remove, remove_if, unique*

Thrust

```
#include <thrust/scan.h>
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::inclusive_scan(data, data + 6, data); // in-place scan
// data is now {1, 1, 3, 5, 6, 9}
```

```
#include <thrust/scan.h>
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::exclusive_scan(data, data + 6, data); // in-place scan
// data is now {0, 1, 1, 3, 5, 6}
```

```
#include <thrust/sort.h>
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

```
#include <thrust/sort.h>
const int N = 6;
int keys[N] = {1, 4, 2, 8, 5, 7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys + N, values);
// keys is now {1, 2, 4, 5, 7, 8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Thrust и CUDA

- ▶ В Thrust предусмотрена взаимная совместимость данных с CUDA API
 - Преобразование итератора Thrust в указатель CUDA C
 - Преобразование указателя CUDA C в итератор Thrust

Thrust и CUDA

```
// Создать вектор на устройстве
thrust::device_vector<int> d_vec(4);
// Получить указатель на память вектора
int* ptr = thrust::raw_pointer_cast(&d_vec[0]);
// Использовать указатель при вызове ядра на CUDA C
my_kernel<<< N / 256, 256 >>>(N, ptr);
// Использовать указатель в функции из CUDA API
cudaMemcpyAsync(ptr, ... );
```

```
// Указатель на память устройства
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));
// Преобразовать указатель с помощью device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);
// Использовать device_ptr в алгоритмах Thrust
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);
// Обратиться к памяти устройства с помощью device_ptr
dev_ptr[0] = 1;
// Освободить память
cudaFree(raw_ptr);
```



Казанский
федеральный
университет

ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

Вопросы

ekhramch@kpfu.ru