

ПРОГРАММИРОВАНИЕ СИСТЕМ С НЕСКОЛЬКИМИ ГРАФИЧЕСКИМИ ПРОЦЕССОРАМИ (MULTI-GPU)

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

ПРЕДПОСЫЛКА

- Зачем вообще использовать Multi GPU?
 - Нужно обрабатывать модели большего размера (не хватает памяти)
 - Нужны более быстрые вычисления
 - Нужна большая энергоэффективность

CUDA CONTEXT

- Контексты устройств неявно создаются при инициализации CUDA-runtime
- На каждом устройстве создается по одному контексту – «primary-контекст»
- Все нити программы совместно их используют
- Инициализация CUDA-runtime происходит неявно, при первом вызове любой функции, не относящейся к Device / Version Management (см. Toolkit Reference Manual)

CUDA CONTEXT

- В каждой CPU-нити может быть только один активный контекст в каждый момент времени
- `cudaSetDevice(n)` - переключение между устройствами (=между контекстами)
- `cudaDeviceReset()` - уничтожает primary-контекст, активный в данный момент
 - При этом будет освобождена вся память, выделенная в контексте
 - При необходимости, новый контекст будет неявно создан в дальнейшем

CUDA CONTEXT AND CUDA STREAM

- Функция `cudaStreamCreate` создает соответствующий ресурс в активном контексте
- Если активный контекст отличен от того, в котором создан поток/событие:
 - Отправление команды в поток вызовет ошибку
 - `cudaEventRecord()` для события вызовет ошибку
 - `cudaEventElapsedTime()` вызовет ошибку, если события созданы в разных контекстах

ПРИМЕР

```
cudaSetDevice(0);
cudaStream_t s0;
cudaStreamCreate(&s0); // создать поток на device 0

cudaSetDevice(1); // переключить контекст на device 1

cudaStream_t s1;
cudaStreamCreate(&s1); // создать поток на device 1

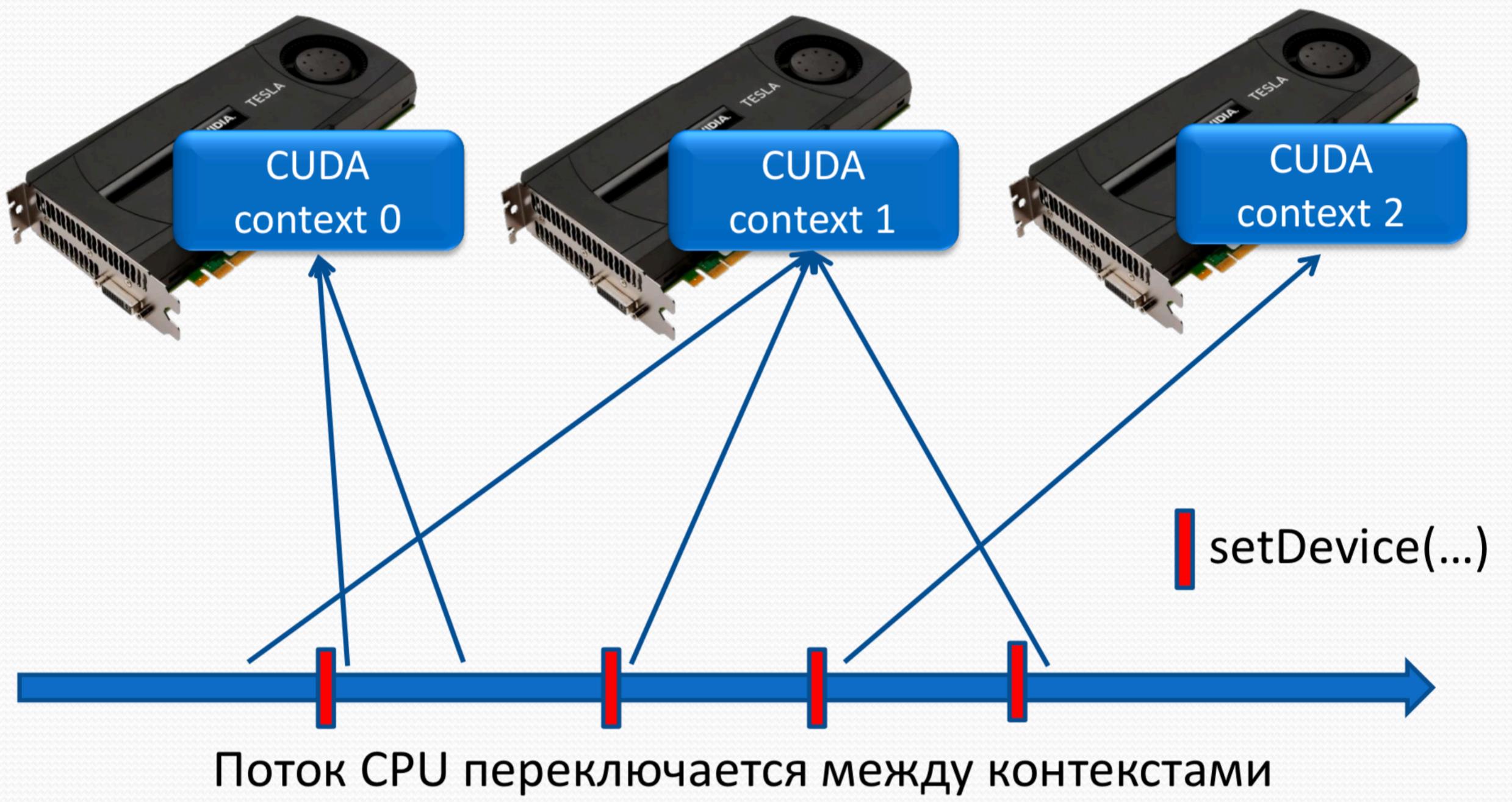
MyKernel<<<100, 64, 0, s1>>>();
MyKernel<<<100, 64, 0, s0>>>(); // ошибка !!
```

- А сделать как правильно?

MULTI GPU

- Два основных возможных подхода:
 - Multi-GPU & single CPU thread (switching contexts)
 - Multi-GPU & multiple CPU threads (using OpenMP)

Multi-GPU & single CPU thread



ПРИМЕР: ВЫДЕЛЕНИЕ ПАМЯТИ

- Пусть у нас есть `deviceCount` GPU-устройств
- `cudaMalloc` блокирует нить, выделения памяти будут производиться последовательно!

```
for(int device = 0; device < deviceCount; device++)  
{  
    cudaSetDevice(device);  
    cudaMalloc(...);  
}
```

ПРИМЕР: ЗАПУСК ЯДЕР

```
for(int device = 0; device < deviceCount; device++)
```

```
{
```

```
    cudaSetDevice(device);
```

```
    cudaMemcpyAsync(...);
```

```
    kernel<<<...>>> (...);
```

```
    cudaMemcpyAsync(....);
```

```
}
```

ПРИМЕР: СИНХРОНИЗАЦИЯ

- Выполняем синхронизацию в отдельном цикле!

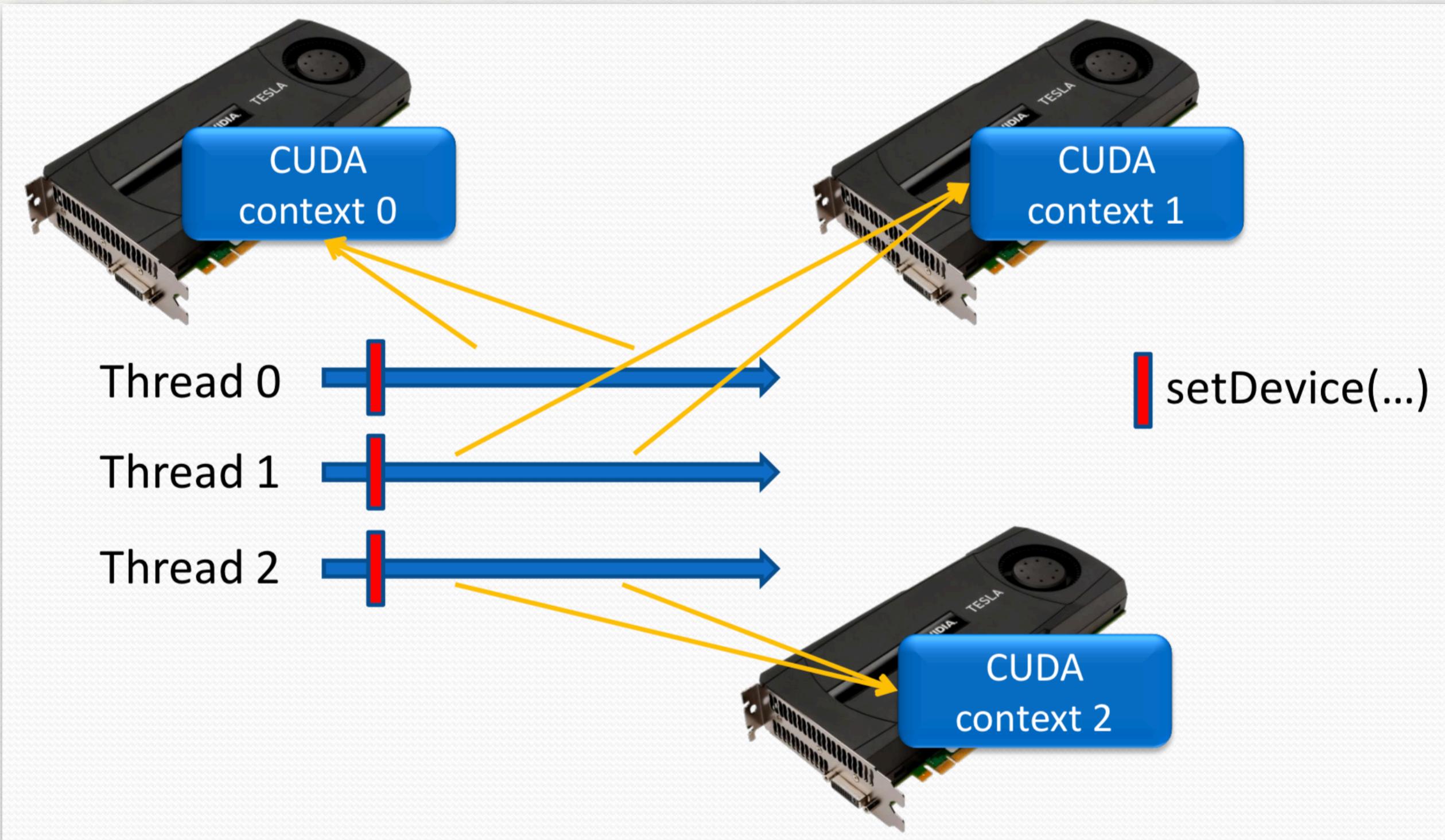
```
for(int device = 0; device < deviceCount; device++)  
{  
    cudaSetDevice(device);  
    cudaDeviceSynchronize();  
}
```

NVVP

Последовательные выделения памяти



MULTI-GPU & MULTIPLE CPU THREADS (OPENMP)



КАК КОМПИЛИРОВАТЬ?

- Поддержка OpenMP встроена в популярные компиляторы Intel icc/ifort, gcc/gfortran, MS cl, IBM xlс
- Обычный компилятор компилирует OpenMP директивы и функции при указании специального флага компиляции (для распознавания директив) и линковки (для линковки отр-функций)
- `icc -fopenmp`
- `gcc -fopenmp`
- `cl -fopenmp`
- `xlc -qsmp`

OPENMP + NVCC

- Два подхода
- Единая компиляция:

```
nvcc -Xcompiler -fopenmp -arch=sm_20 main.cu
```

- Раздельная компиляция:

```
nvcc -arch=sm_20 kernel.cu
```

```
$gcc -fopenmp -I/opt/cuda/include main.c
```

```
$gcc -fopenmp -L/opt/cuda/lib -lcudart main.o kernel.o
```

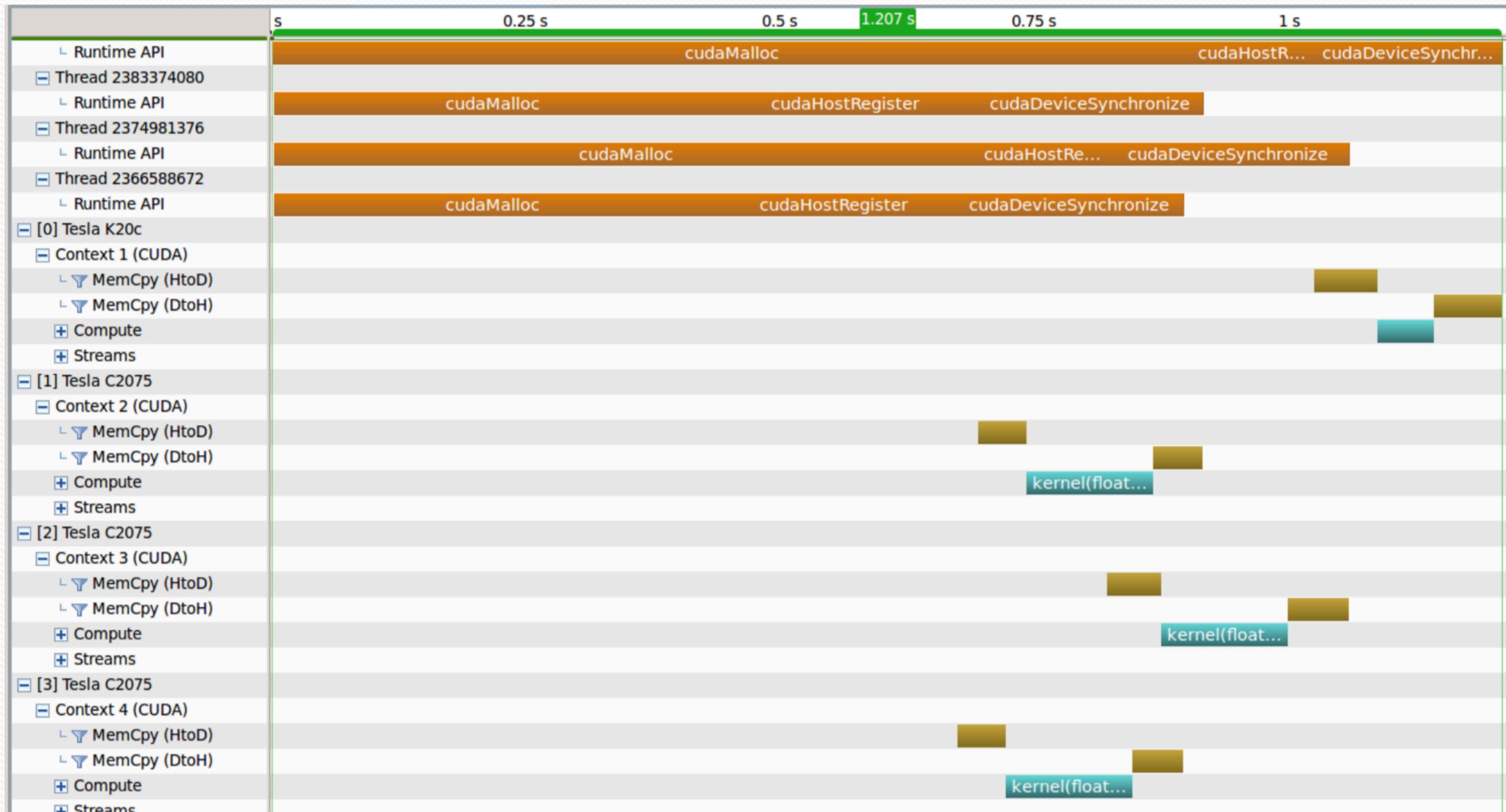
- Раздельная компиляция предпочтительнее, так как могут использоваться более оптимизированные компиляторы для оптимизации CPU кода

УЛУЧШАЕМ ВЫДЕЛЕНИЕ ПАМЯТИ ПРИ ПОМОЩИ OPENMP

- Теперь выделения памяти будут происходить параллельно
- Не требуется отдельных циклов для синхронизации и запуска команд на GPU

```
int deviceCount;  
cudaGetDeviceCount(&deviceCount);  
#pragma omp parallel num_threads(deviceCount)  
{  
    int device = omp_get_thread_num();  
    cudaSetDevice(device); // устанавливаем для каждого потока свой контекст  
    cudaMalloc(devPtr + device, elemsPerDevice * sizeof(float));  
  
    cudaHostRegister(...);  
  
    // прочие команды, копирования, запуски ядер  
    cudaDeviceSynchronize();  
}
```

NVVP (MULTIPLE THREADS)



ОБМЕНЫ МЕЖДУ GPU

- При поддержке Unified памяти peer-to-peer обмены между памятью разных GPU делаются неявно при использовании обычных функций `cudaMemcpy*`
- dst и src указывают на память на разных устройствах
- Если Unified память не поддерживается или нужно явно указать, что это peer-to-peer копирование, используются функции `cudaMemcpyPeer*`

PEER-TO-PEER

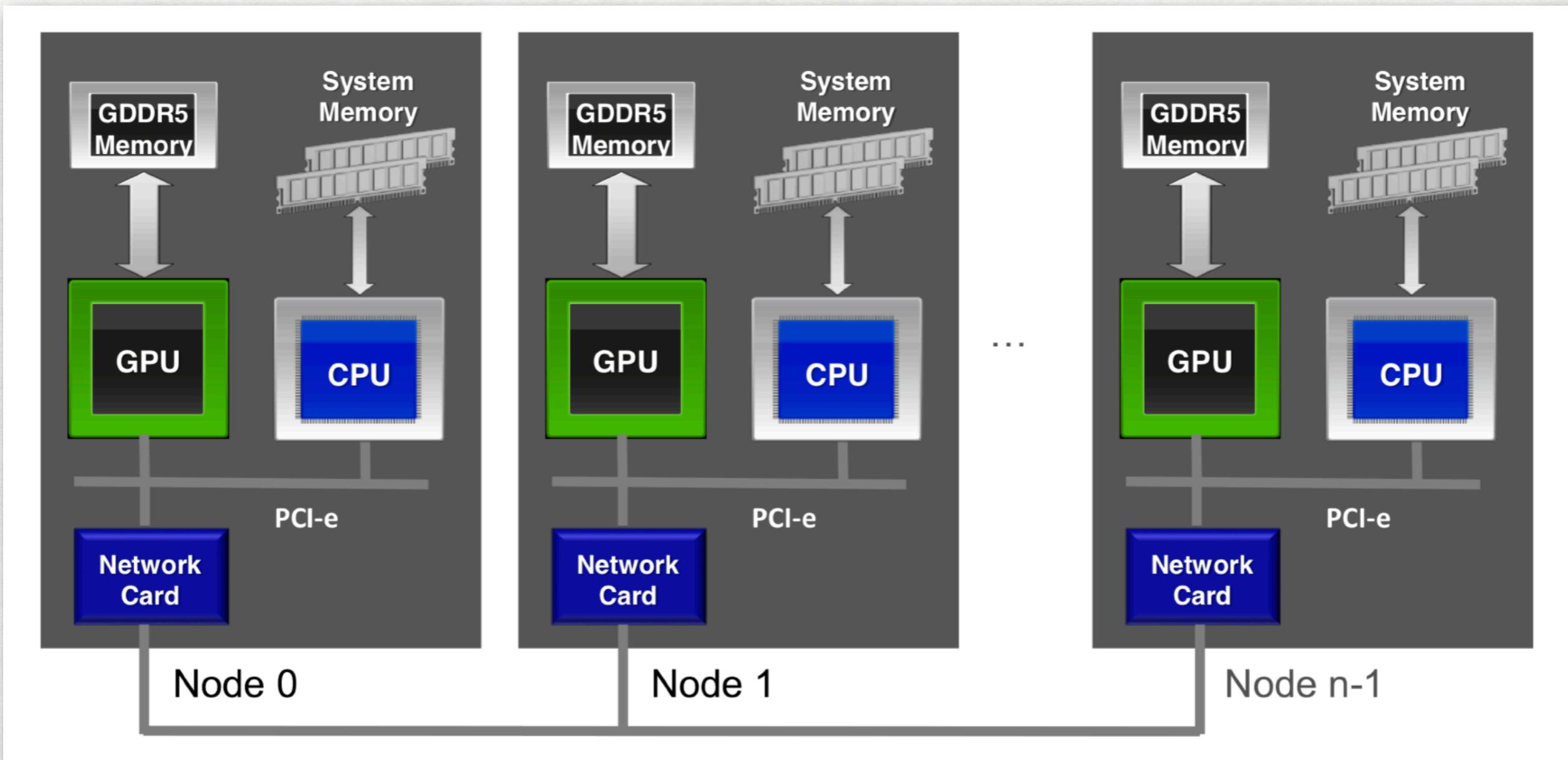
- Нужно явно указать номера устройств, между которыми происходит обмен
- `cudaError_t cudaMemcpyPeer (void* dst, int dstDevice, const void* src, int srcDevice, size_t count)`
- `cudaError_t cudaMemcpyPeerAsync (void* dst, int dstDevice, const void* src, int srcDevice, size_t count, cudaStream_t stream=0)`
- Обе функции не блокируют хост
- `cudaMemcpyPeer` начнется только когда завершатся все команды на обоих устройствах (и на активном), отправленные до неё
- Параллельно с `cudaMemcpyPeer` не могут выполняться другие команды на обоих устройствах (и на активном)
- `cudaMemcpyPeerAsync` лишена этих ограничений

ПРИМЕР Р2Р КОПИРОВАНИЯ

```
cudaSetDevice(0); // Переключились на device 0  
float* p0, *p1;  
size_t size = 1024 * sizeof(float);  
cudaMalloc(&p0, size); // Выделили на device 0  
cudaSetDevice(1); // Переключились на device 1  
cudaMalloc(&p1, size); // Выделили на device 1  
cudaSetDevice(0); // Переключились на device 0 MyKernel<<<1000, 128>>>(p0);  
// Запуск на device 0  
cudaSetDevice(1); // Переключились на device 1  
cudaMemcpyPeer(p1, 1, p0, 0, size); // Копировать p0 to p1  
MyKernel<<<1000, 128>>>(p1); // Запуск на device 0
```

MULTI GPU + MPI

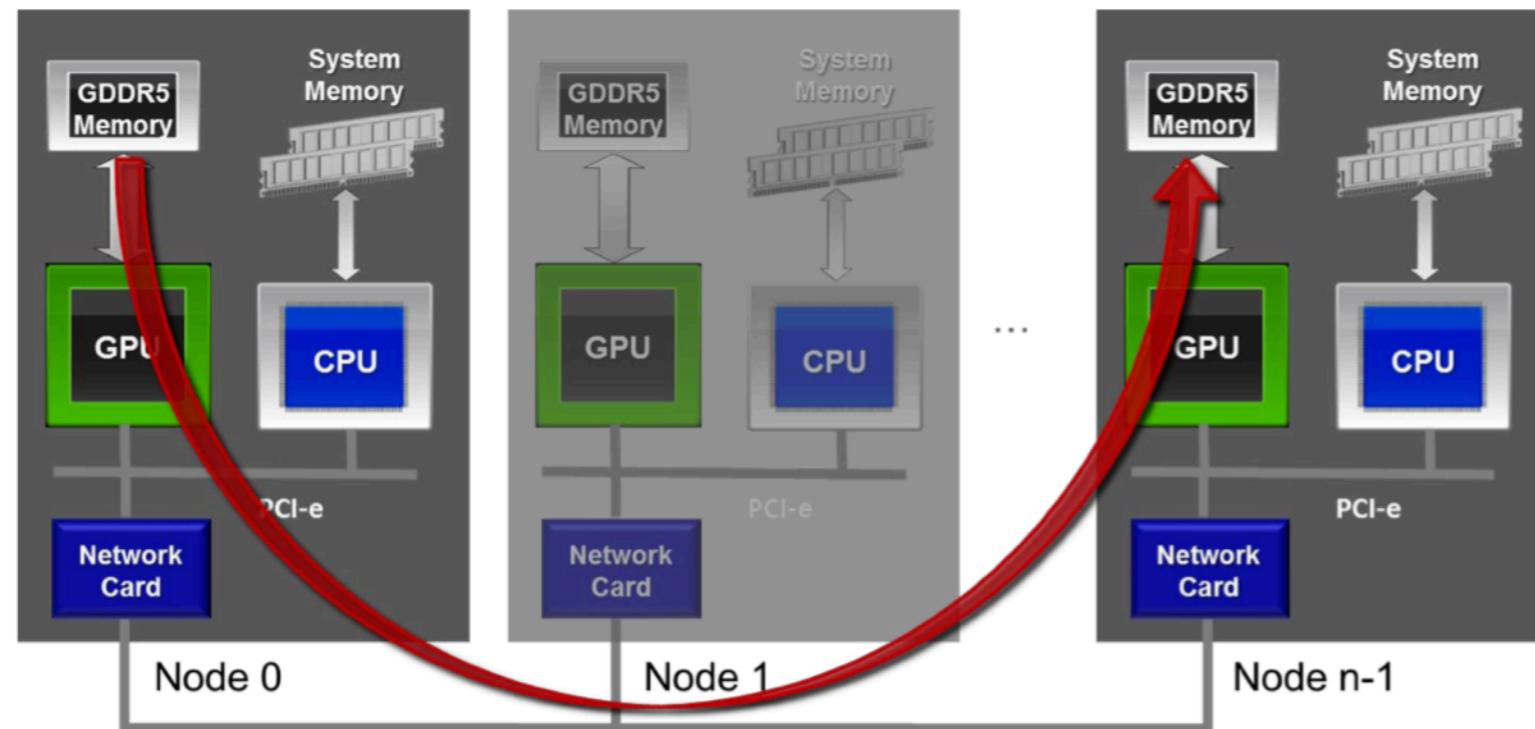
MPI + CUDA



ЧТО ТАКОЕ MPI?

- MPI - стандарт для обмена данных между процессами посредством передачи сообщений, определяет API для передачи сообщений
- Операции типа точка-точка: напр. MPI_Send, MPI_Recv
- Коллективные операции, напр. MPI_Reduce
- Существуют различные операции (с открытым исходным кодом и коммерческие) – обертки для компиляторов C/C++, Fortran, Python – MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT и др.

MPI + CUDA, ОБЫЧНЫЕ ОБМЕНЫ

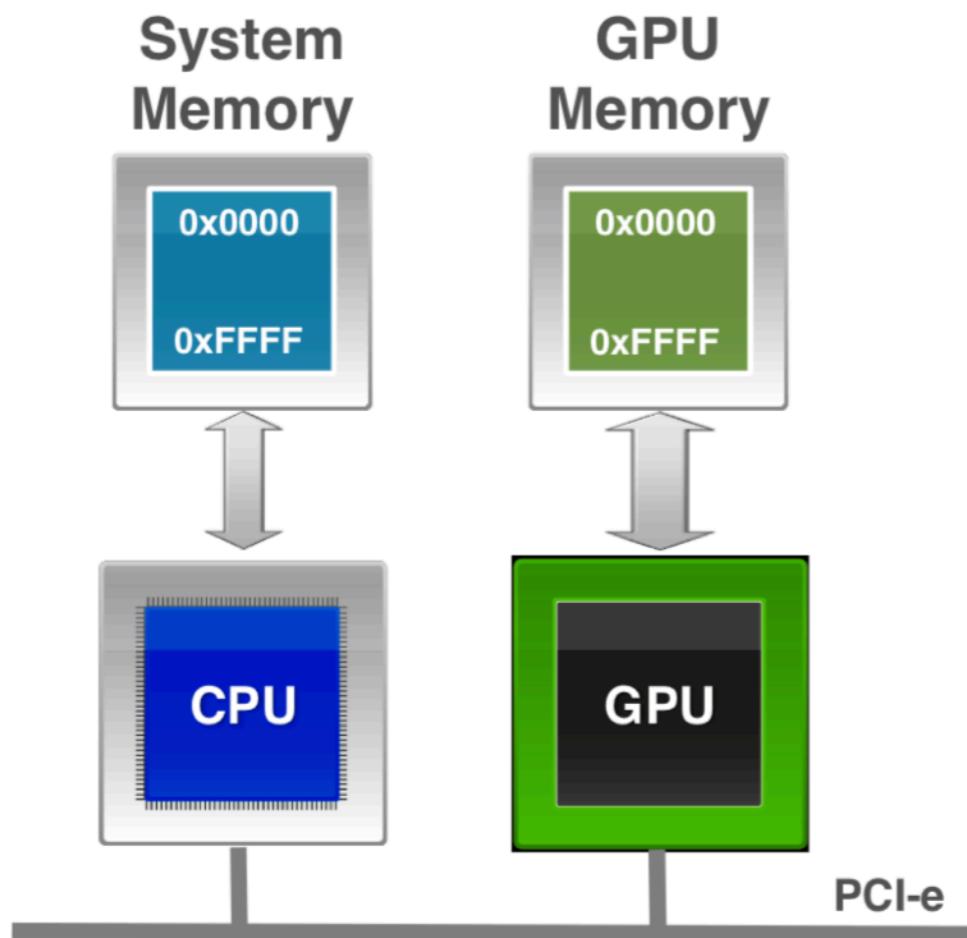


```
//MPI rank 0
MPI_Send(s_buf_d, size, MPI_CHAR, n-1, tag, MPI_COMM_WORLD);

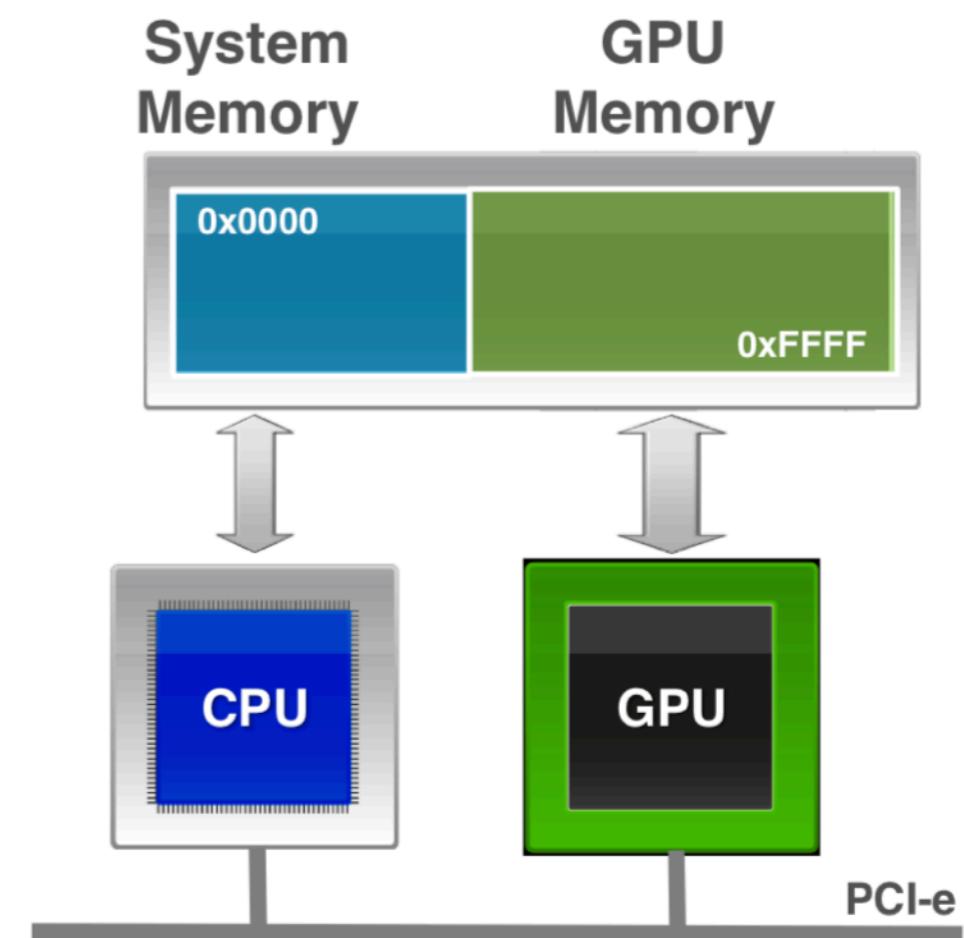
//MPI rank n-1
MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
```

UVA & NON-UVA

No UVA : Separate Address Spaces



UVA : Single Address Space



UVA & NON-UVA EXAMPLE

- UVA и CUDA-aware MPI

//MPI rank 0

MPI_Send(s_buf_d,size,...);

//MPI rank n-1

MPI_Recv(r_buf_d,size,...);

- без UVA и обычный MPI

//MPI rank 0

cudaMemcpy(s_buf_h,s_buf_d,size,...); MPI_Send(s_buf_h,size,...);

//MPI rank n-1

MPI_Recv(r_buf_h,size,...); cudaMemcpy(r_buf_d,r_buf_h,size,...);