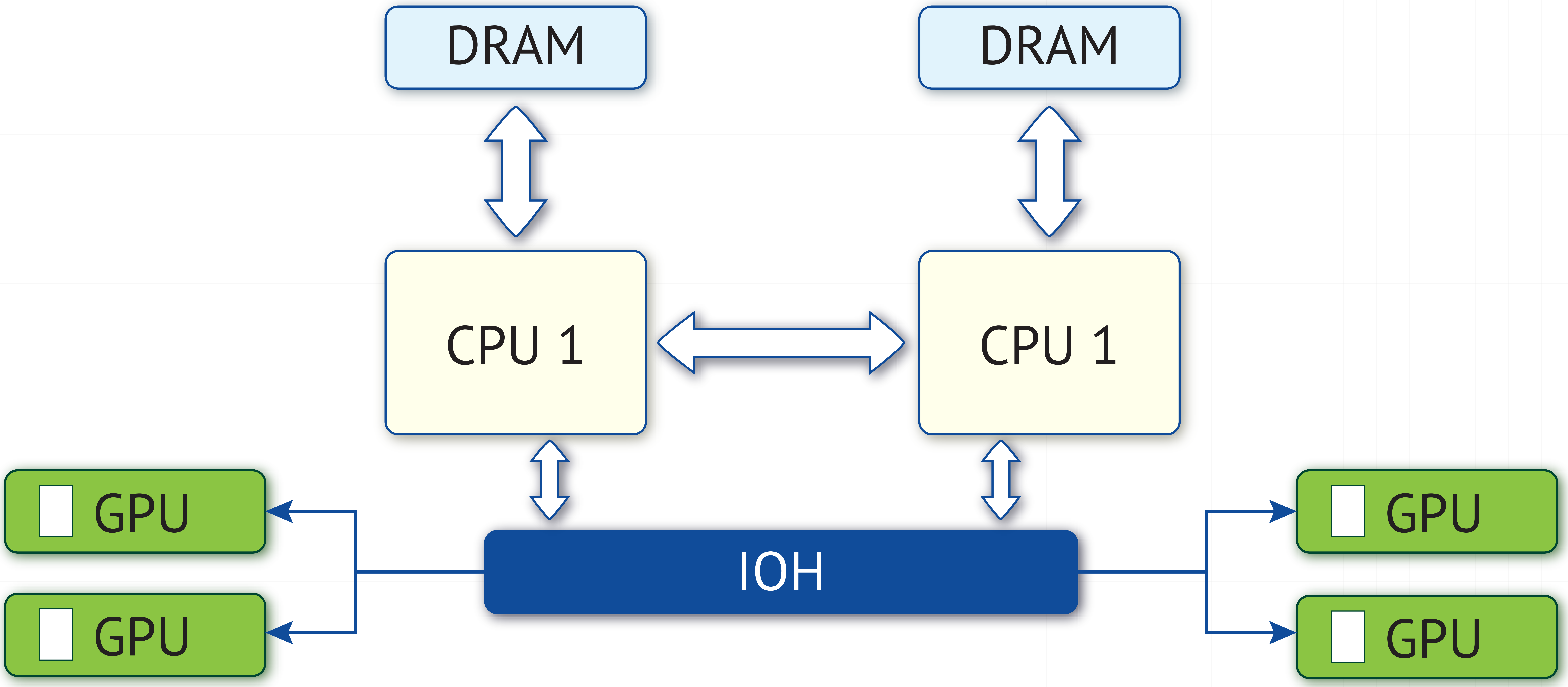
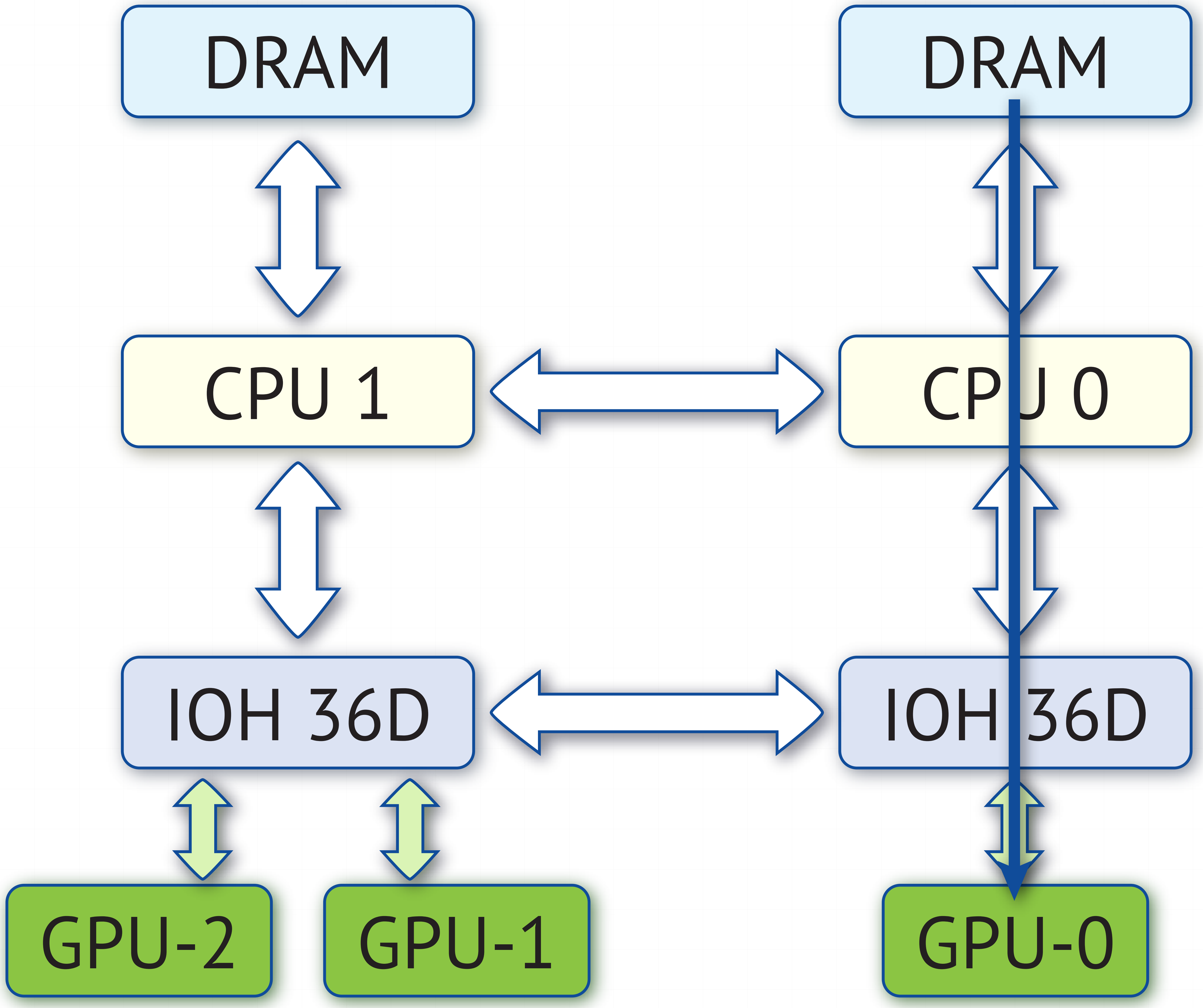
Гибридная архитектура



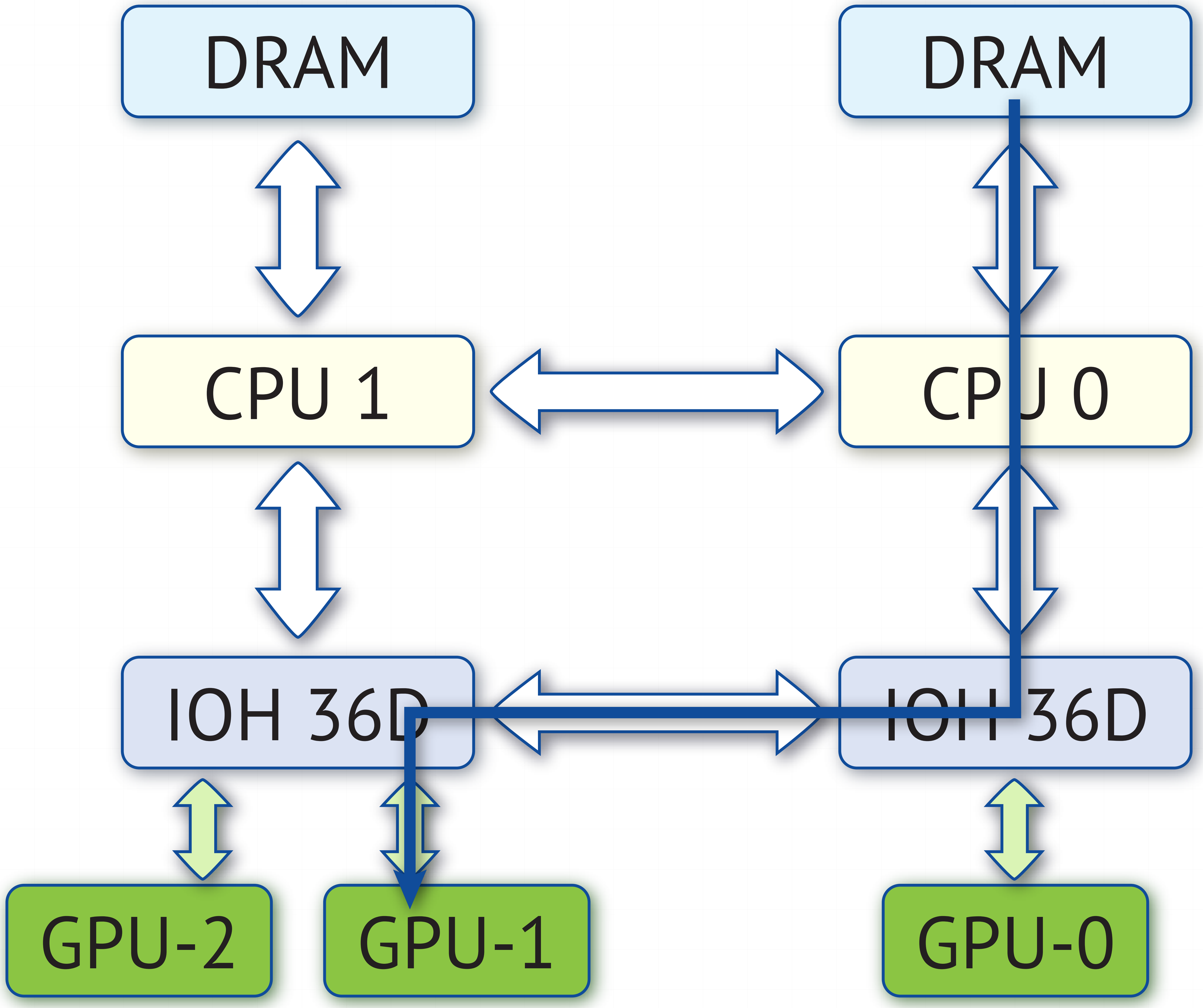
Замечания по NUMA

- Хост (CPU) NUMA(Non Uniform Memory Access) влияет на передачу через PCIe в системами с двумя IOH
  - Меньшая пропускная способность при передаче на “дальний” GPU
  - Дополнительный скачок через QPI
  - Касается любого PCIe устройства, не только GPU
    - Например, сетевые карты
- Старайтесь запускать CPU потоки на сокете, ближнем к IOH чипу GPU
  - Можно использовать numactl, GOMP_CPU_AFFINITY, KMP_AFFINITY, и т.д.
- Количество скачков по PCIe незначительно влияет на скорость передачи

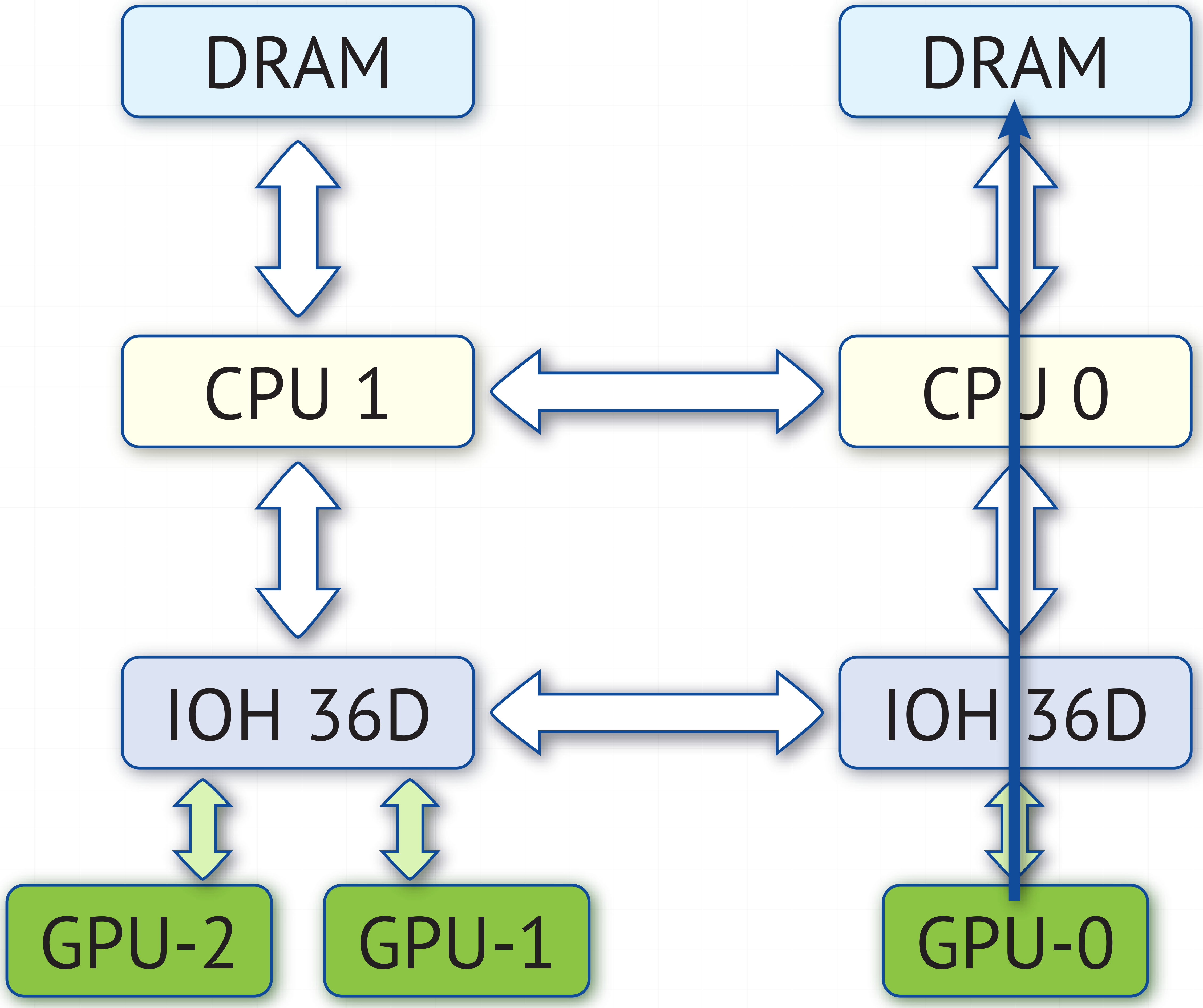
"Local" H2D Copy: 5.7 GB/s



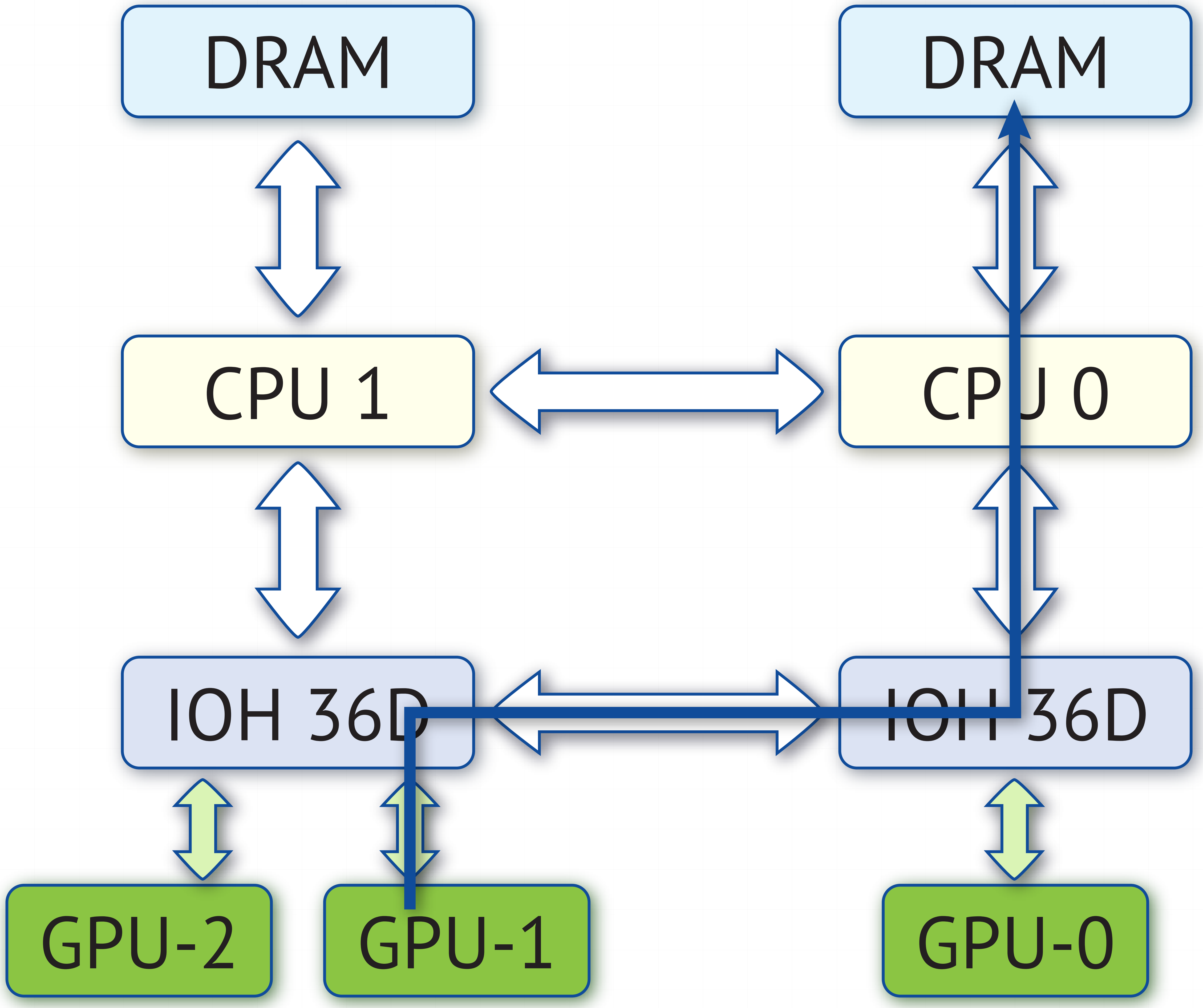
"Remote" H2D Copy: 4.9 GB/s



"Local" D2H Copy: 6.3 GB/s



"Remote" H2D Copy: 4.9 GB/s



Роль CUDA

- **CUDA** – это API для организации вычислений на GPU, но не на всём кластере
- Для работы с более сложные системами необходимо использовать дополнительные программные модели

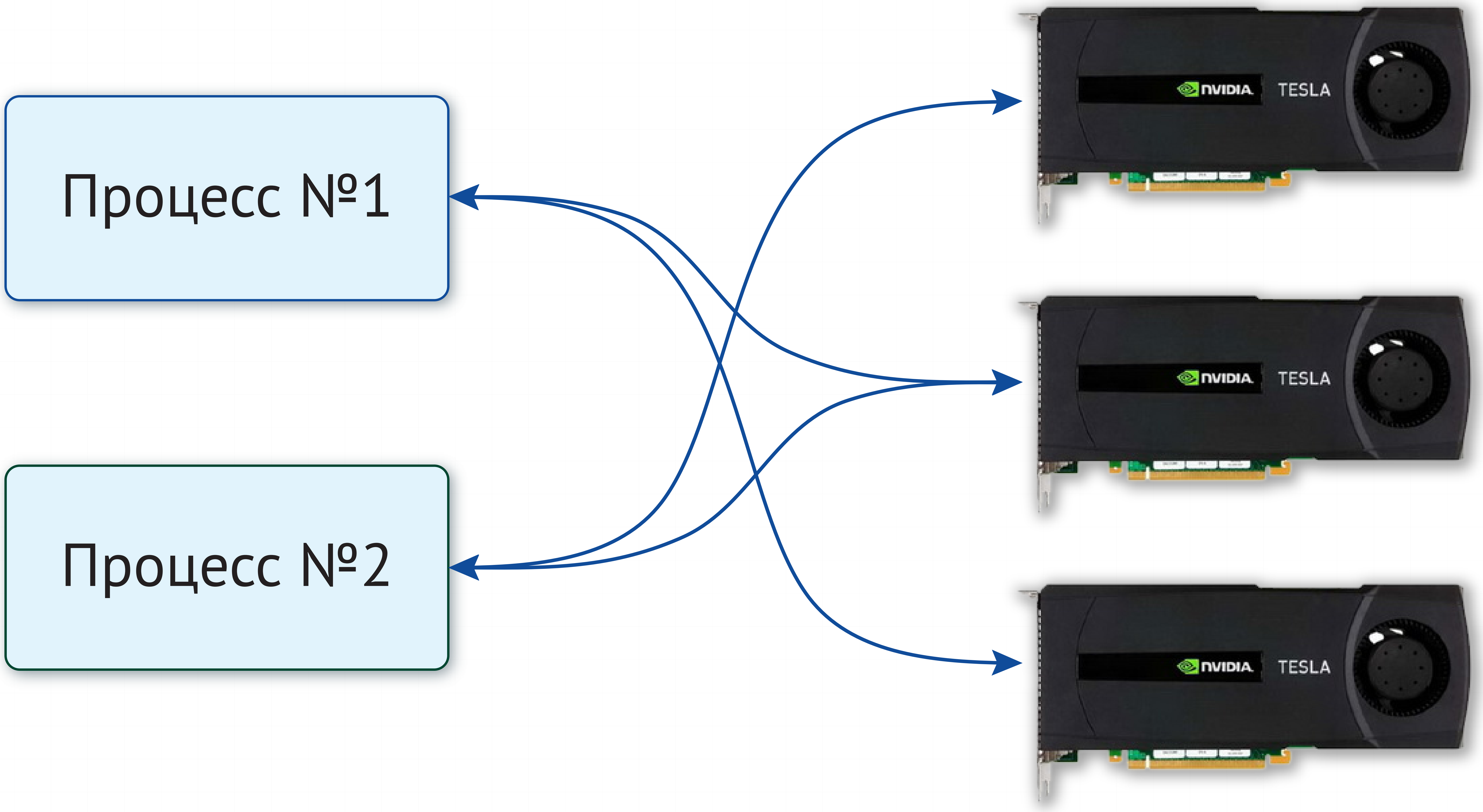
Несколько GPU на одном узле

- Способы контроля GPU:
  - Один CPU поток
  - Несколько CPU потоков относящихся к одному процессу
  - Несколько CPU процессов

ОС: процессы

- Собственный *контекст* в системе (память, файлы, …)
- Управление процессами происходит через системные вызовы (сравнительно дорого)
- Каждый CPU процесс имеет свое адресное пространство
- Процесс может создать несколько потоков в разделяемом адресном пространстве

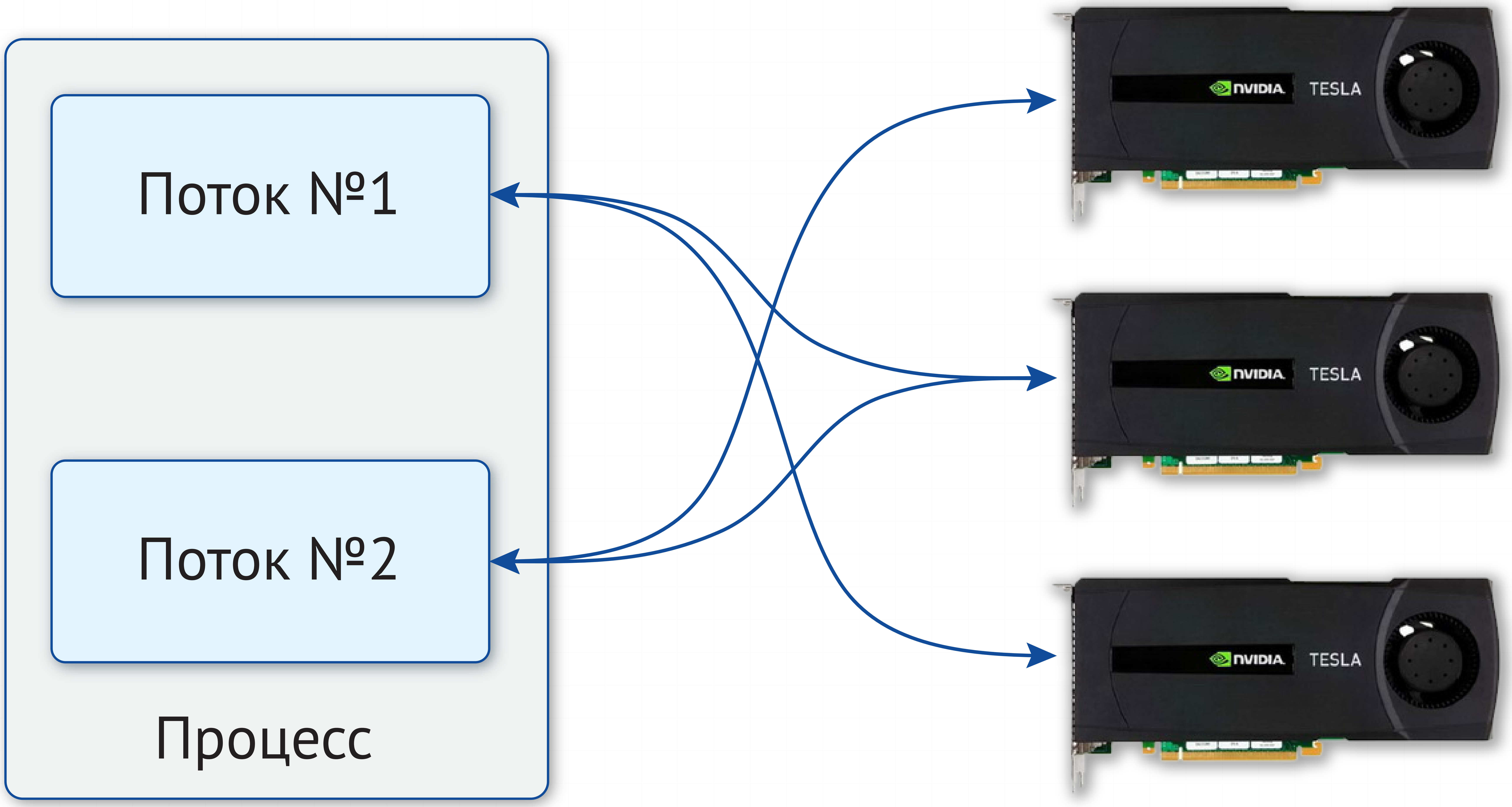
ОС: процессы



ОС: потоки

- Общий доступ ко всей памяти родительского процесса, локальная память потока
- В управлении потоками ОС может играть меньшую роль

ОС: процессы


Поток №1
Поток №2
Процесс
NVIDIA TESLA
NVIDIA TESLA
NVIDIA TESLA

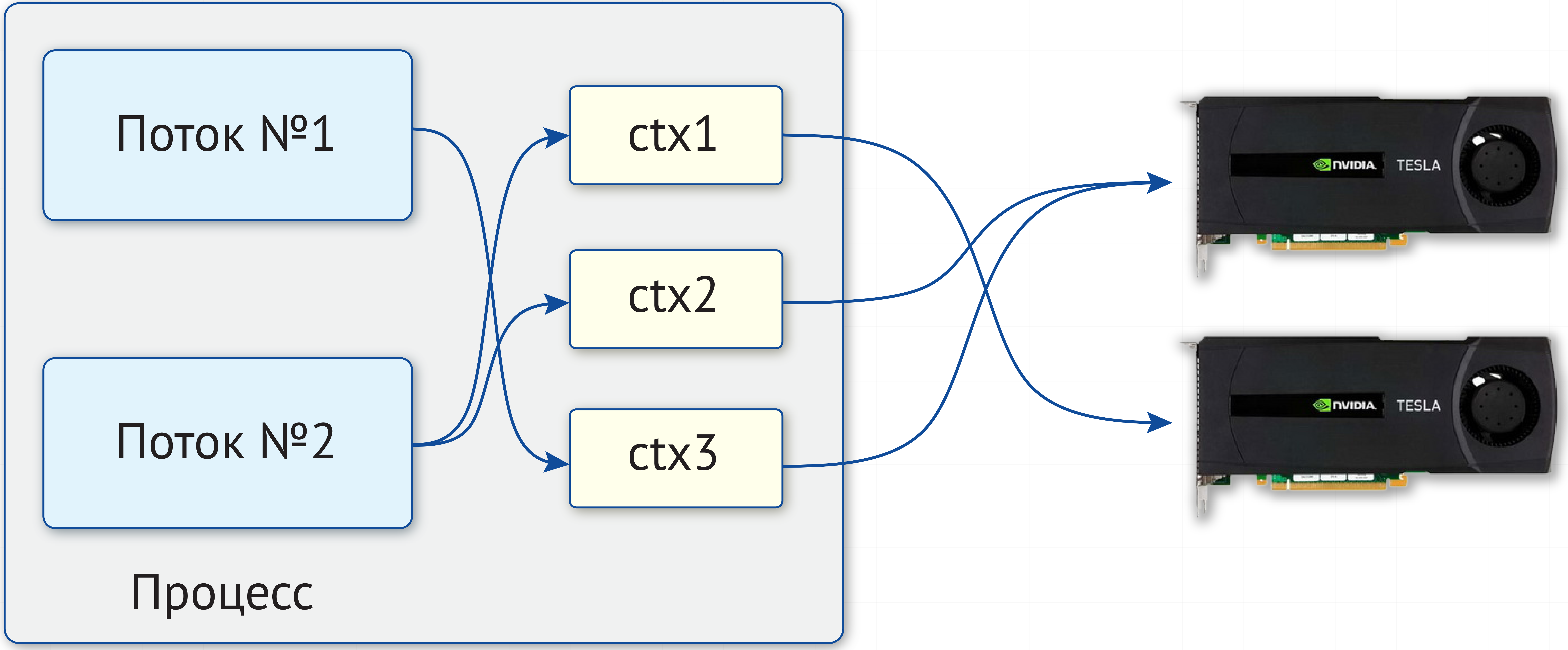
Контекст устройства (CUDA)

- Контекст – привязанная к определённому устройству управляющая информация (выделенная device-память, результат операции, ...)
- При обращении к устройству многие CUDA-вызовы требуют существование контекста

Контекст устройства (CUDA)

- Изначально поток/процесс не имеет текущего CUDA-контекста
- Если в процессе/потоке нет текущего контекста, то он будет создан неявно при необходимости
- Одно устройство может иметь несколько контекстов

Контекст устройства (CUDA)



Контекст устройства (CUDA)

- Каждый процесс/поток может иметь не более одного текущего контекста
- Процесс/поток может создавать и удалять контексты, менять текущий контекст

Управление контекстами

- CUDA API:
  - Контекст устройства создаётся неявно
  - Переключение контекста: cudaSetDevice(<номер_устройства>)
- Driver API:
  - cuCtxCreate/cuCtxDestroy
  - uCtxPushCurrent/cuCtxPopCurrent

Пример №1

Создать процесс, использующий одновременно несколько GPU с помощью cudaSetDevice

Реализация: serial/serial_cuda/

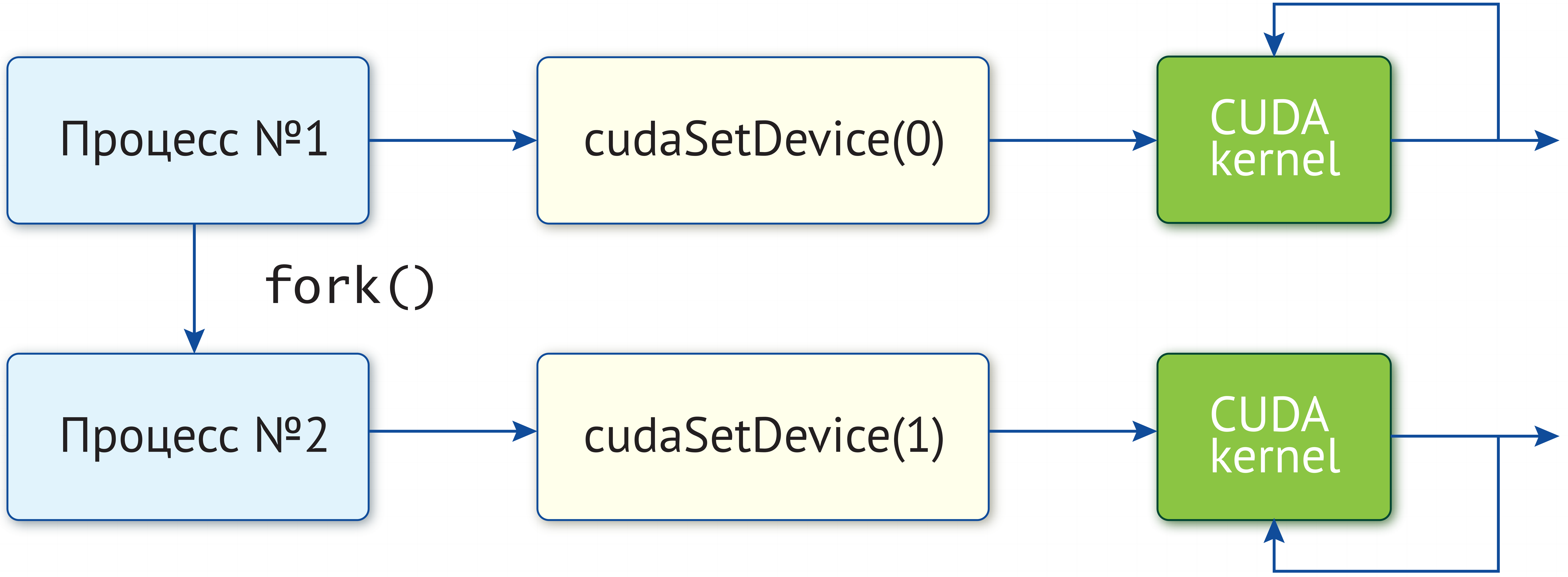
Пример №2

Создать несколько процессов, обрабатывающих независимые данные на одном или нескольких GPU

Если в системе один GPU, использовать его во всех процессах.

Реализация: unix/process_fork_cuda/

Пример №2


Процесс №1
cudaSetDevice(0)
CUDA kernel
fork()
Процесс №2
cudaSetDevice(1)
CUDA kernel

fork( )

```
// Call fork to create another process.
// Standard: "Memory mappings created in the parent
// shall be retained in the child process."
pid_t fork_status = fork();

// From this point two processes are running the same code, if no errors.
if (fork_status == -1)
{
    fprintf(stderr, "Cannot fork process, errno = %d\n", errno);
    return errno;
}

// By fork return value we can determine the process role:
// master or child (worker).
int master = fork_status ? 1 : 0, worker = !master;

// Get the process ID
int pid = (int)getpid;
```

Message Passing Interface (MPI)

- Реализация – библиотека, демон
- Единый код исполняется множеством параллельных процессов
- Демоны MPI контролируют запуск и состояние процессов на узлах вычислительной сети
- Демоны MPI контролируют запуск и состояние процессов на узлах вычислительной сети

Message Passing Interface (MPI)

- Порождение множества процессов
  mpirun, mpiexec
- Инициализация, деинициализация
  MPI_Init, MPI_Finalize
- Обмен данными
  MPI_Send, MPI_Recv, MPI_Bcast, …
- Синхронизация
  MPI_Barrier, …

GPU-память в командах MPI

Поддержка использования девайс-адресов в MPI командах с CUDA 4.0

Доступна в OpenMPI trunk (Rolf vandeVaart)

```
[dmikushin@sm06 forge]$ svn co http://svn.open-mpi.org/svn/ompi/trunk ompi-trunk

[dmikushin@sm06 forge]$ cd ompi-trunk/

[dmikushin@sm06 ompi-trunk]$ ./autogen.pl

[dmikushin@sm06 ompi-trunk]$ mkdir build

[dmikushin@sm06 ompi-trunk]$ cd build

[dmikushin@sm06 build]$ ../configure --prefix=/home/dmikushin/opt/openmpi_gcc-trunk
--with-cuda

[dmikushin@sm06 build]$ make install
```

MPI_Send / MPI_Recv

- В MPI_Send/_Recv – device-указатели din1/din2

```
float *din1, *din2;
cuda_status = cudaMalloc((void**)&din1, size);
...
cuda_status = cudaMalloc((void**)&din2, size);
...
MPI_Request request;
int inext = (iprocess + 1) % nprocesses;
int iprev = iprocess - 1; iprev += (iprev < 0) ?
nprocesses : 0;

// Pass entire process input device buffer directly to input device buffer
// of next process.
mpi_status = MPI_Isend(din1, n * n, MPI_FLOAT, inext, 0, MPI_COMM_WORLD, &request);
mpi_status = MPI_Recv(din2, n * n, MPI_FLOAT, iprev,  0, MPI_COMM_WORLD, NULL);
mpi_status = MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Пример №3

С помощью MPI создать несколько процессов, обрабатывающих независимые данные на GPU и обменивающихся результатами через память GPU.

Реализация: mpi/mpi_cuda_p2p/

POSIX threads (pthread)

- Реализация – библиотека
- Пользователь явно управляет созданием, завершением потоков и их свойствами
- Пользователь явно управляет взаимодействием потоков

POSIX threads (pthread)

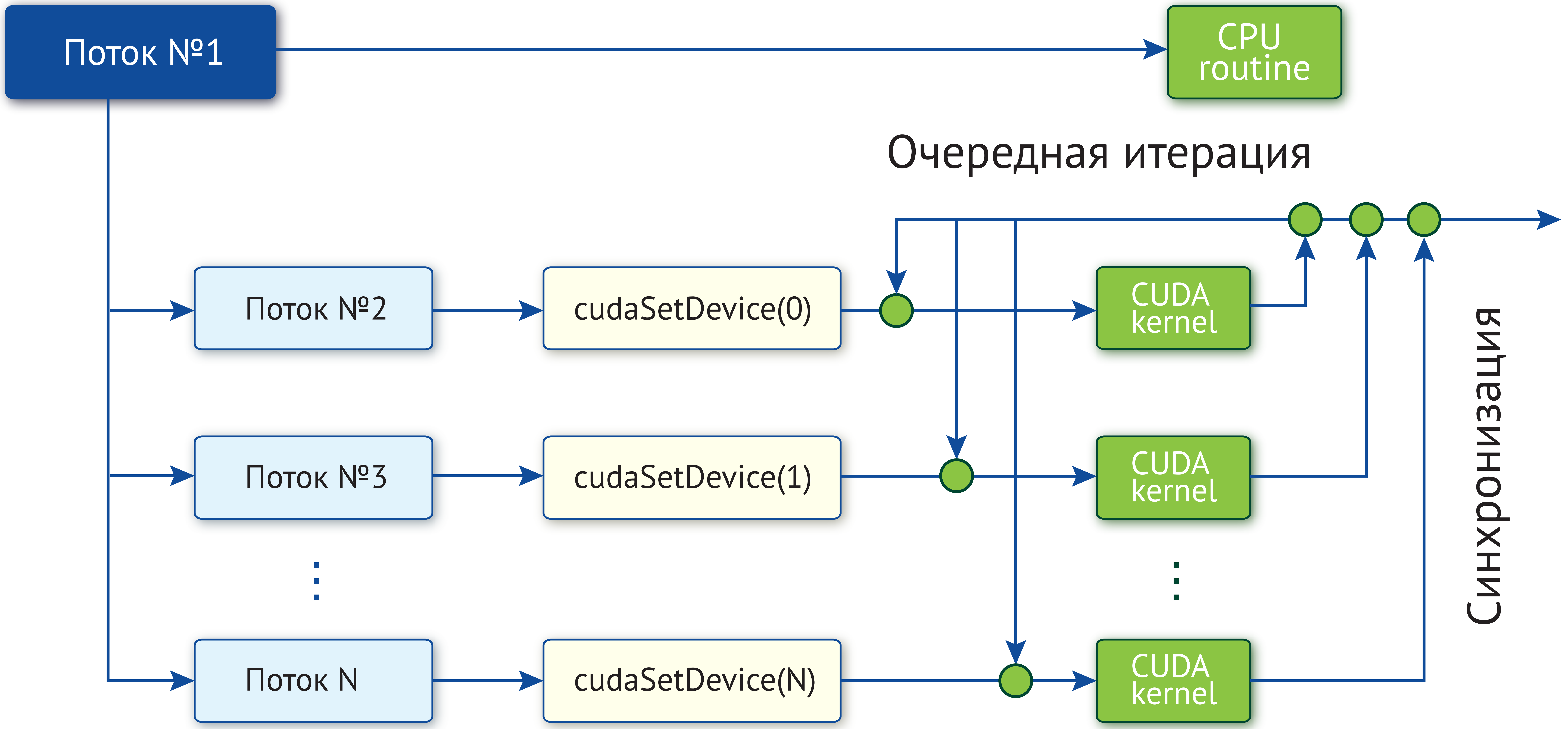
- Порождение и ожидание завершения потока
  pthread_create, pthread_join
- Критическая секция
  pthread_mutex_lock, pthread_mutex_unlock, …
- Барьерная и условная синхронизация
  pthread_barrier_wait, pthread_cond_wait

Пример №4

С помощью pthread создать несколько потоков, параллельно обрабатывающих независимые данные на GPU и обменивающихся результатами через память GPU.

Реализация: pthread/pthread_cuda_p2p/

Пример №4


Поток №1
CPU routine
Очередная итерация
Поток №2
cudaSetDevice(0)
CUDA kernel
Поток №3
cudaSetDevice(1)
CUDA kernel
Поток N
cudaSetDevice(N)
CUDA kernel
Синхронизация

OpenMP

- Реализация – директивы (расширения языков C, Fortran, …), библиотека
- Созданием и завершением потоков управляет runtime-библиотека, некоторые свойства потоков могут быть заданы пользователем явно
- Пользователь явно управляет взаимодействием потоков

OpenMP

- Параллельное исполнение
  #pragma omp parallel
- Число потоков
  omp_get_num_threads(), OMP_NUM_THREADS
- Параллельные циклы
  #pragma omp parallel for

Пример №5

С помощью OpenMP создать несколько потоков, параллельно обрабатывающих независимые данные на GPU и CPU.

Реализация: openmp/openmp_cuda/

omp section-s, parallel for

```
// For each CUDA device found create a separate thread
// and execute the thread_func.
#pragma omp sections
{
    // Section for GPU threads.
    #pragma omp section
    {
    }

    // Section for CPU thread.
    #pragma omp section
    {
    }
}
```

omp section-s, parallel for

```
// Section for GPU threads.
#pragma omp section
{
    #pragma omp parallel for
    for (int i = 0; i < ndevices; i++)
    {
        config_t* config = configs + i;
        config->idevice = i;
        config->step = 0;
        config->nx = nx; config->ny = ny;
        config->inout_cpu = inout + np * i;
        config->status = thread_func(config);
    }
}
```

omp section-s, parallel for

```
// Section for CPU thread.
#pragma omp section
{
    // In parallel main thread launch CPU function equivalent
    // to CUDA kernels, to check the results.
    control = inout + ndevices * np;
    float* input = inout + (ndevices + 1) * np;
    for (int i = 0; i < nticks; i++)
    {
        pattern2d_cpu(1, configs->nx, 1, 1, configs->ny, 1,
        input, control, ndevices);
        float* swap = control;
        control = input;
        input = swap;
    }
    float* swap = control;
    control = input;
    input = swap;
}
```

Boost

- Создание потока
  boost::thread, boost::bind
- Синхронизация
  boost::mutex, boost::barrier

Пример №6

С помощью Boost создать несколько параллельных потоков, пошагово обрабатывающих независимые данные на GPU и CPU с синхронизацией после каждого шага.

Реализация: boost/boost_cuda/

www.parallel-computing.pro

Спасибо! ☺ Вопросы?