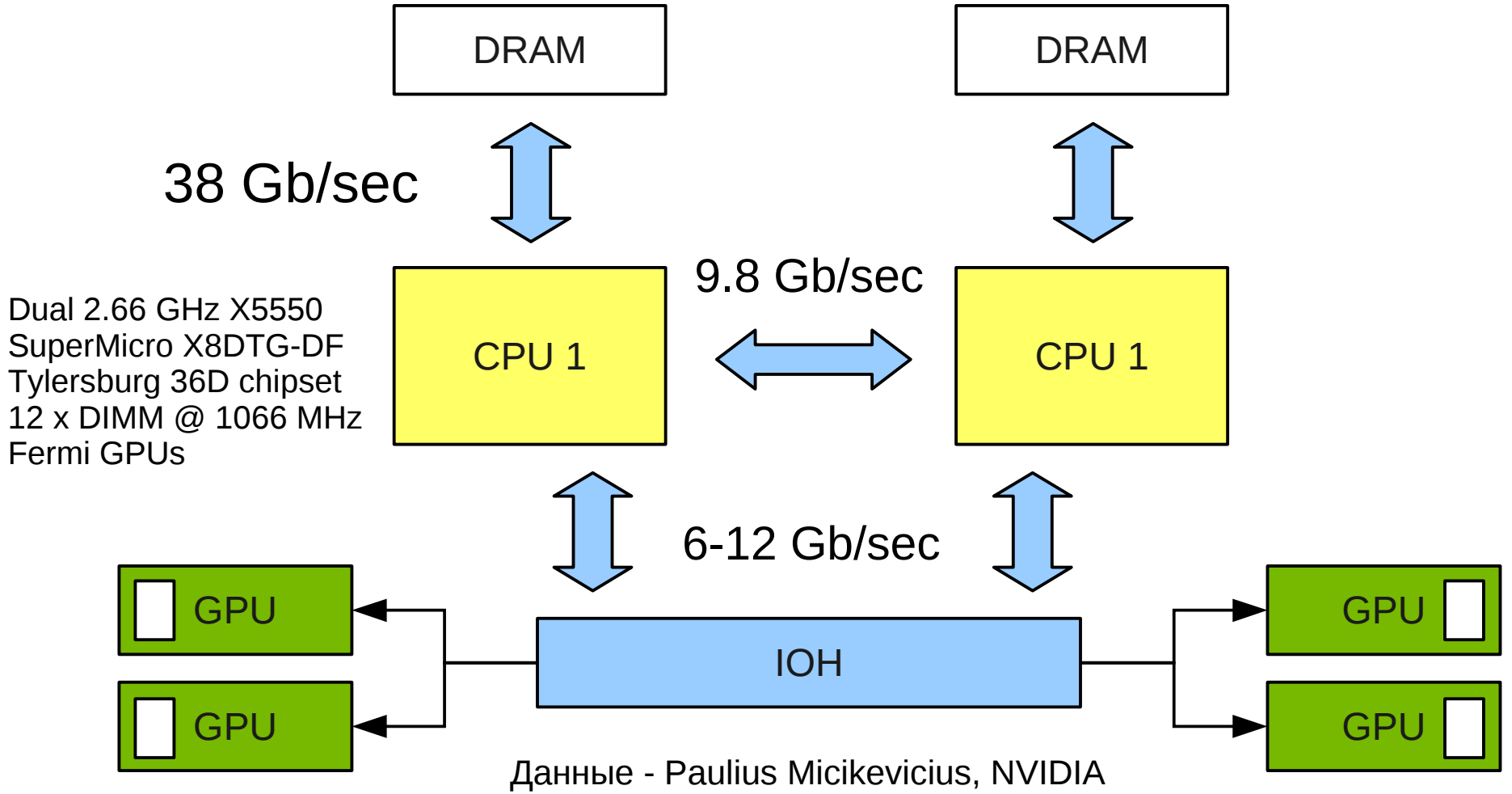
Программирование multi-GPU систем

Дмитрий Микушин

План

- Гибридная архитектура
- Иерархия памяти, процессы, потоки
- GPU контекст
- Примеры multi-GPU программ
- Асинхронные операции, CUDA streams

Гибридная архитектура



2011

Иерархия памяти

- Кеш ядра, сокета
- Ближняя RAM сокета
- Дальняя RAM (через QPI)
- Память PCI-E устройства
- RAM другого выч. узла
- Память чужого PCI-E устройства

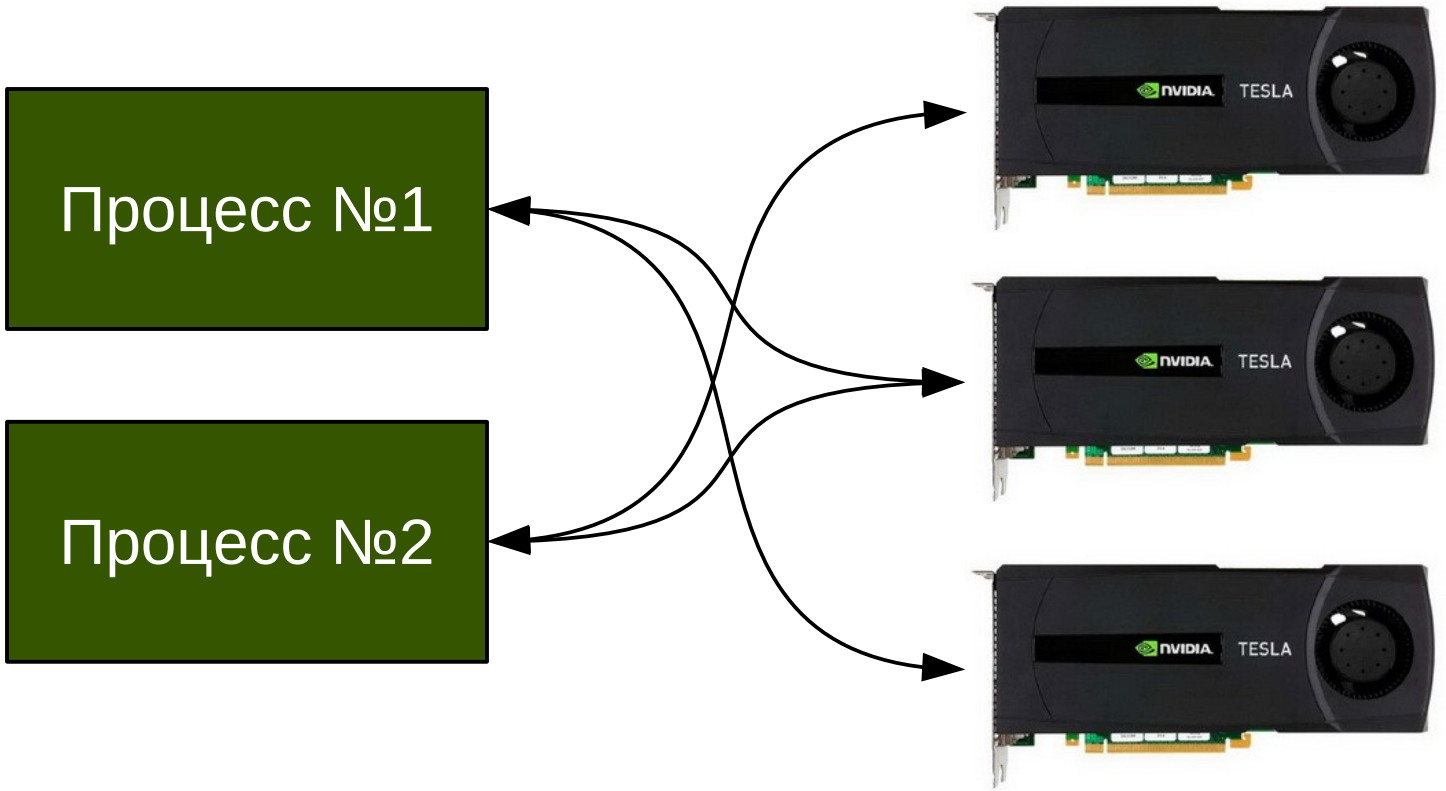
Роль CUDA

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

ОС: процессы

- Собственный *контекст* в системе (память, файлы, ...)
- Управление процессами происходит через системные вызовы (сравнительно дорого)

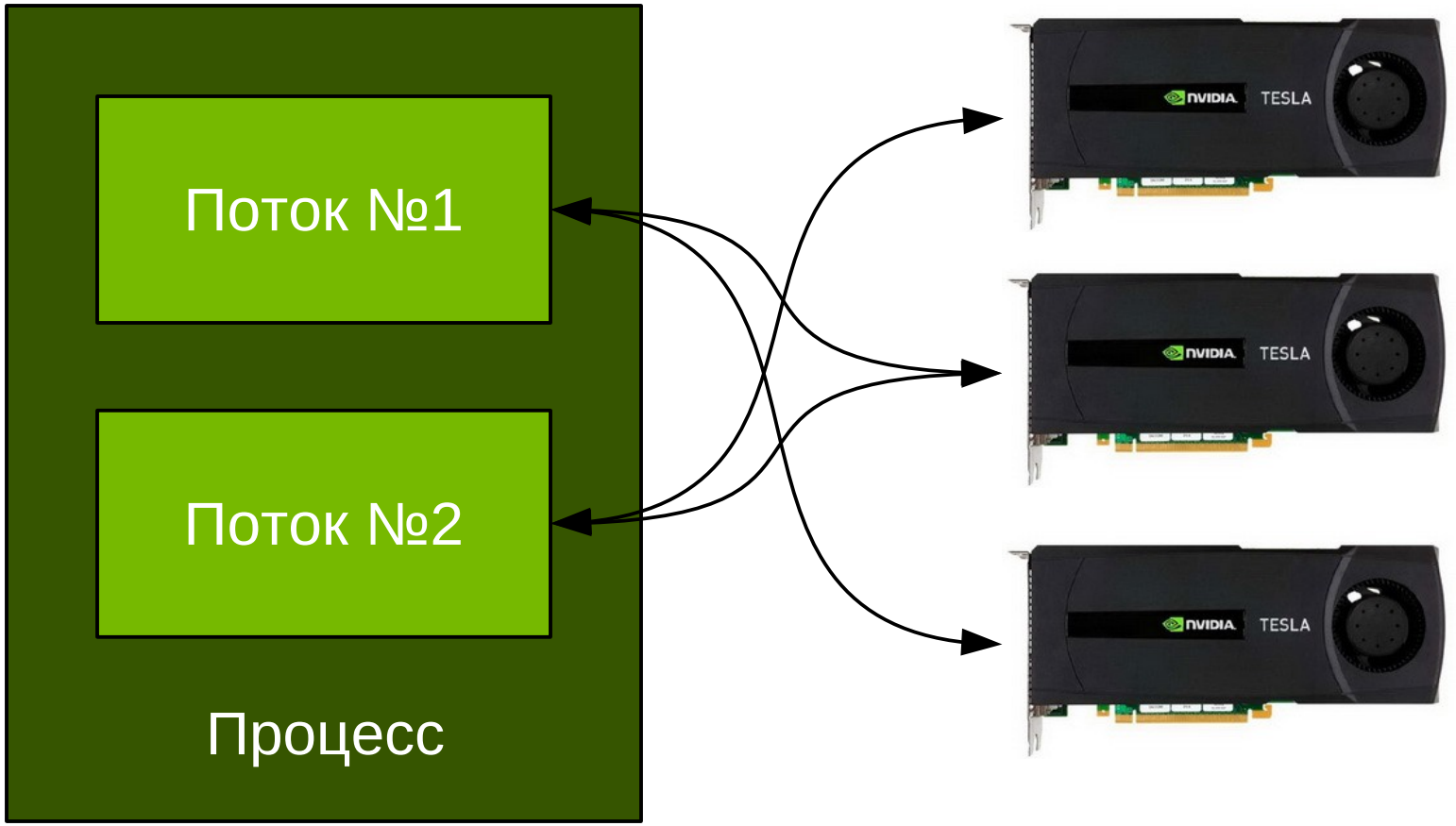
ОС: процессы



ОС: потоки

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

ОС: потоки


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

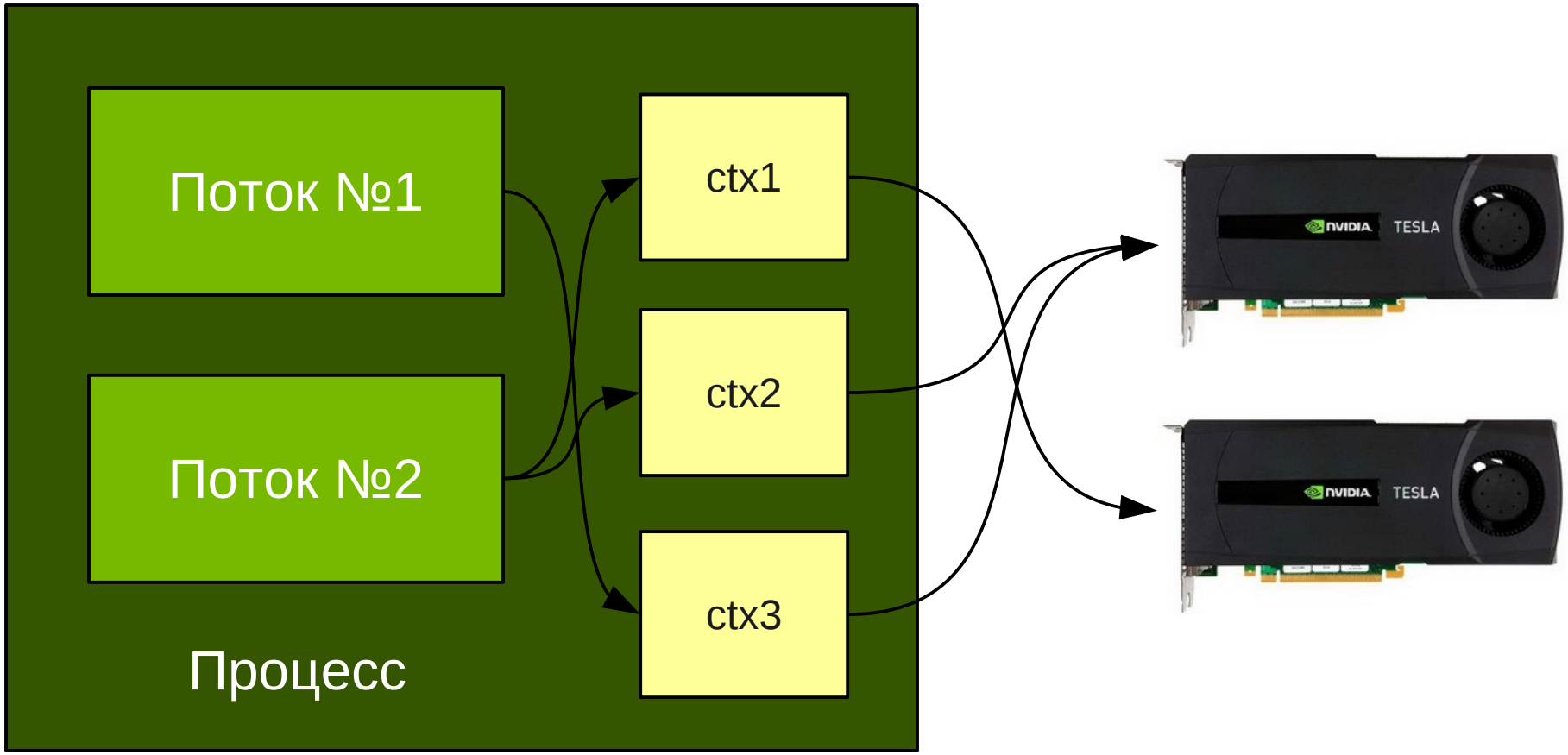
Контекст устройства (CUDA)

- 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

multi-GPU в последовательных приложениях

Полный исходный код примеров – в материалах к лекции

Пример №1

Создать последовательное приложение, параллельно использующее несколько GPU.

Реализация: serial/serial_cuda/

Пример №1

- Каждому устройству явно создаётся контекст (cuCtxCreate)
- Перед выполнением операций с устройством соответствующий контекст делается текущим (cuCtxPushCurrent), после операции – снимается (cuCtxPopCurrent)
- В конце контексты удаляются (cuCtxDestroy)

cuDeviceGet, cuCtxCreate

```
CUdevice dev;
CUresult cu_status = cuDeviceGet(&dev, idevice);
if (cu_status != CUDA_SUCCESS)
{
        fprintf(stderr,
               "Cannot get CUDA device by index %d, status = %d\n",
               idevice, cu_status);
        return cu_status;
}

cu_status = cuCtxCreate(&config->ctx, 0, dev);
if (cu_status != CUDA_SUCCESS)
{
        fprintf(stderr,
              "Cannot create a context for device %d, status = %d\n",
               idevice, cu_status);
        return cu_status;
}
```

cuCtxPushCurrent / *PopCurrent

```
// Set focus on the specified CUDA context.
// Previously we created one context for each thread.
CUresult cu_status = cuCtxPushCurrent(config->ctx);
if (cu_status != CUDA_SUCCESS)
{
        fprintf(stderr,
                "Cannot push current context for device %d, status = %d\n",
                idevice, cu_status);
        return cu_status;
}

// Pop the previously pushed CUDA context out of this thread.
cu_status = cuCtxPopCurrent(&config->ctx);
if (cu_status != CUDA_SUCCESS)
{
        fprintf(stderr,
                "Cannot pop current context for device %d, status = %d\n",
                idevice, cu_status);
        return cu_status;
}
```

cuCtxDestroy

```
cu_status = cuCtxDestroy(config->ctx);
if (cu_status != CUDA_SUCCESS)
{
        fprintf(stderr,
                "Cannot destroy context for device %d\n",
                idevice, cu_status);
        return cu_status;
}
```

multi-GPU в параллельных приложениях

Рассматриваются различные программные модели параллельных вычислений в сочетании с CUDA

Полный исходный код примеров – в материалах к лекции

Open Group / IEEE

- Создание дочернего процесса

  fork
- Файлы с общим доступом

  shm_open, shm_unlink
- Отображение файла в память процесса

  mmap, munmap, msync
- Семафоры

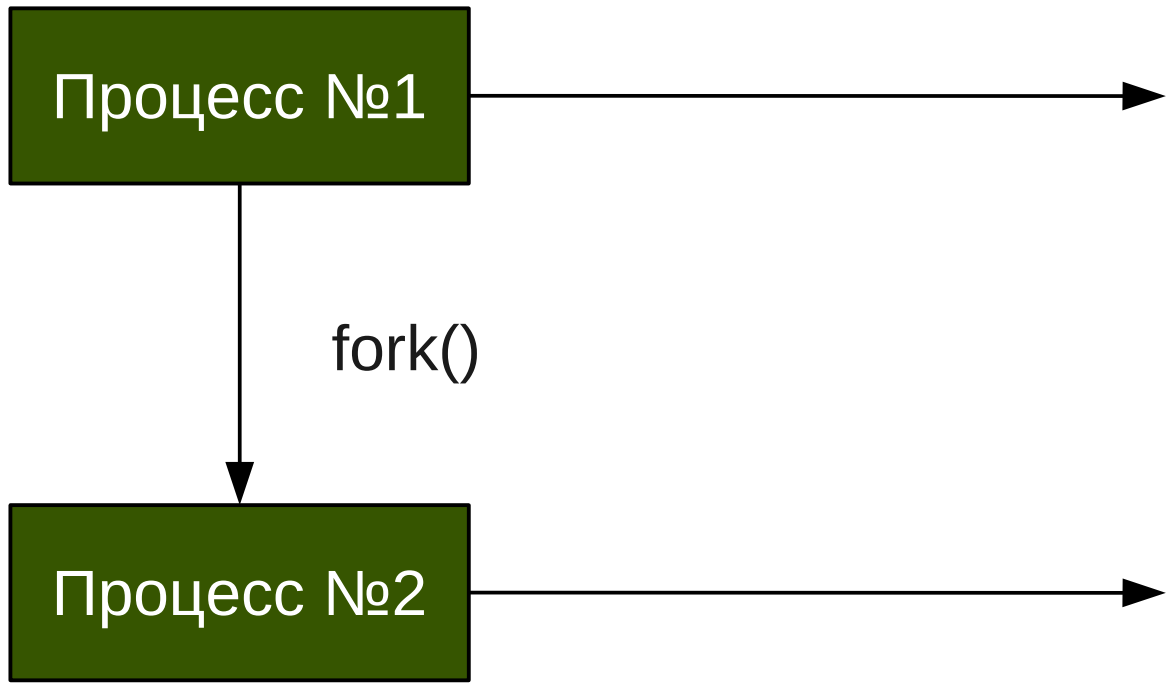
  sem_open, sem_wait, sem_post, sem_unlink

Пример №2

Создать несколько процессов, обрабатывающих независимые данные на CPU

Реализация: unix/process_fork/

Пример №2



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();
```

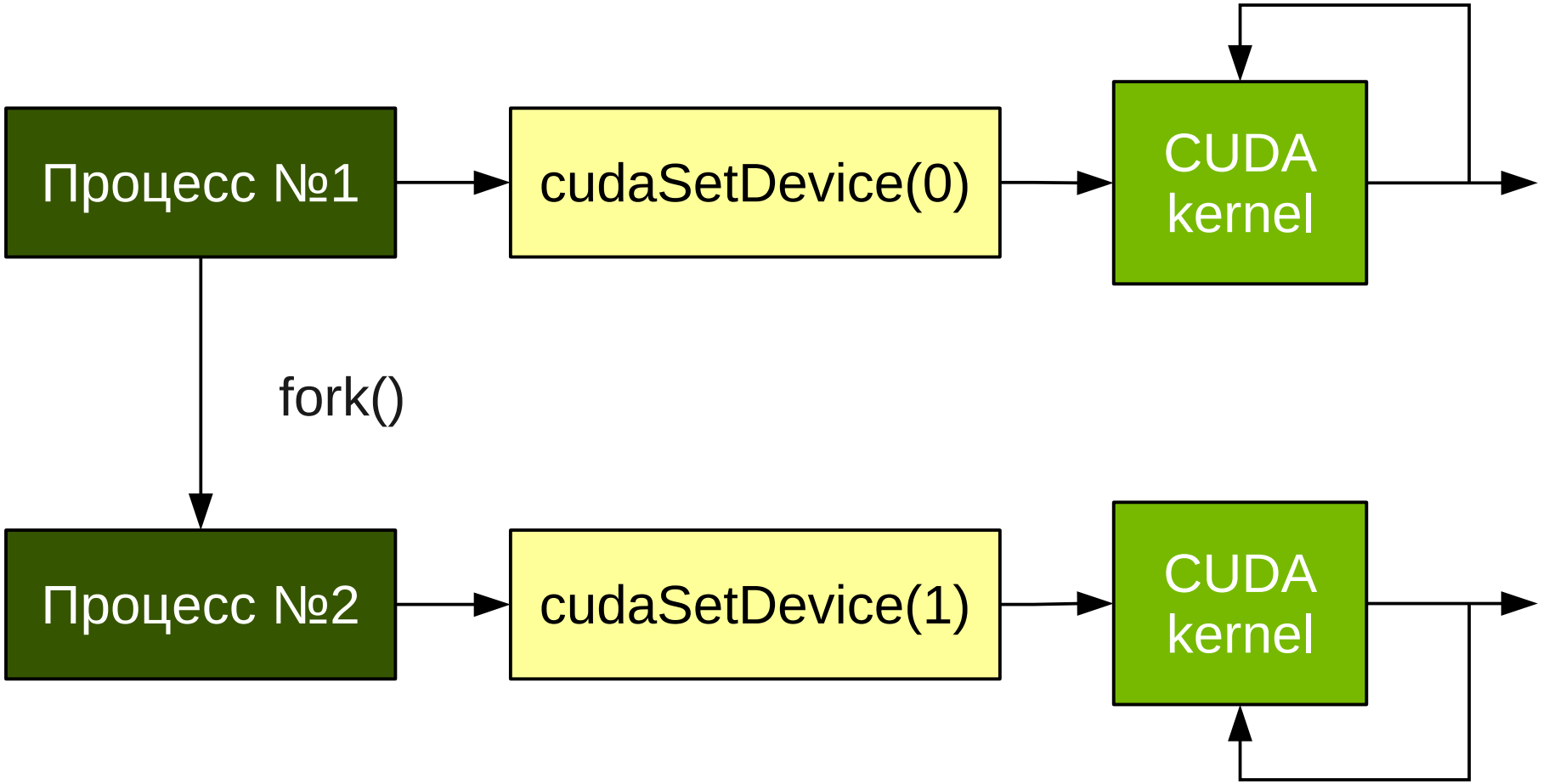
Пример №3

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 №3


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

cudaSetDevice

```
// Use different devices, if more than one present.
if (ndevices > 1)
{
        int idevice = 1;
        if (master) idevice = 0;

        cuda_status = cudaSetDevice(idevice);
        if (cuda_status != cudaSuccess)
        {
                fprintf(stderr,
                        "Cannot set CUDA device by process %d, status = %d\n",
                        pid, cuda_status);
                return cuda_status;
        }
        printf("Process %d uses device #%d\n", pid, idevice);
}
```

Замечание к примеру №3

Если CUDA-контекст был создан **до** вызова fork(), то в порождённом процессе CUDA-вызовы будут работать некорректно.

Пример №4

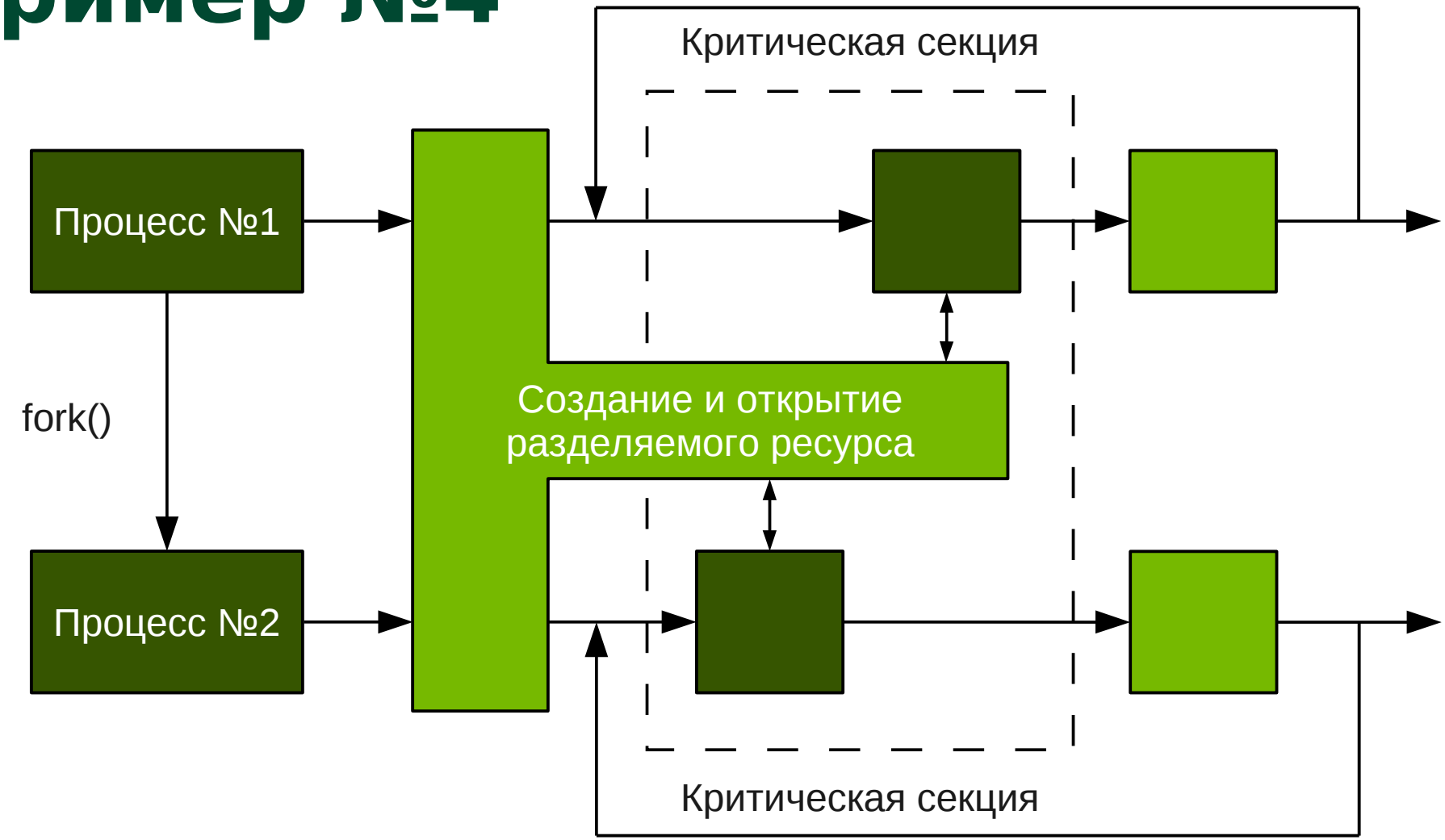
Создать несколько процессов, обрабатывающих независимые данные и обменивающихся результатами через разделяемый буфер.

Реализация: unix/shmem_mmap/

Примитивы синхронизации

- Мютекс (1 сейф – один ключ)
- Семафор (1 сейф – несколько ключей)
- Барьер (лифт)

Пример №4


Критическая секция
Процесс №1
fork()
Процесс №2
Создание и открытие
разделяемого ресурса
Критическая секция

shm_open / shm_unlink

```
// Create shared memory region.
int fd = shm_open("/myshm",
        O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1)
{
        fprintf(stderr, "Cannot open shared region, errno = %d\n", errno);
        return errno;
}

// Unlink shared region.
if (master)
{
        int unlink_status = shm_unlink("/myshm");
        if (unlink_status == -1)
        {
                fprintf(stderr,
                        "Cannot unlink shared region by process %d, errno = %d\n",
                        pid, errno);
                return errno;
        }
}
```

mmap / munmap

```
// Map the shared region into the address space of the process.
char* shared_data = (char*)mmap(0, szmem,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (shared_data == MAP_FAILED)
{
        fprintf(stderr, "Cannot map shared region to memory, errno = %d\n",
                errno);
        return errno;
}

// Unmap shared region.
close(fd);
int munmap_status = munmap(shared_data, szmem);
if (munmap_status == -1)
{
        fprintf(stderr, "Cannot unmap shared region by process %d, errno = %d\n",
                pid, errno);
        return errno;
}
```

msync

```
// Fill shared data array.
sprintf(shared_data, "Initial shared data array state");

// Sync changed content with shared region.
int msync_status = msync(shared_data, szmem, MS_SYNC);
if (msync_status == -1)
{
        fprintf(stderr, "Cannot sync shared memory %p, errno = %d\n",
                shared_data, errno);
        return errno;
}
```

sem_open / sem_unlink

```
// Create semaphore.
sem_t* sem = sem_open("/mysem", O_CREAT, S_IRWXU | S_IRWXG | S_IRWXO, 1);
if (sem == SEM_FAILED)
{
        fprintf(stderr, "Cannot open semaphore by process %d, errno = %d\n",
                pid, errno);
        return errno;
}

// Unlink semaphore.
if (master)
{
        int sem_status = sem_unlink("/mysem");
        if (sem_status == -1)
        {
                fprintf(stderr,
                        "Cannot unlink semaphore by process %d, errno = %d\n",
                        pid, errno);
                return errno;
        }
}
```

sem_wait / sem_post

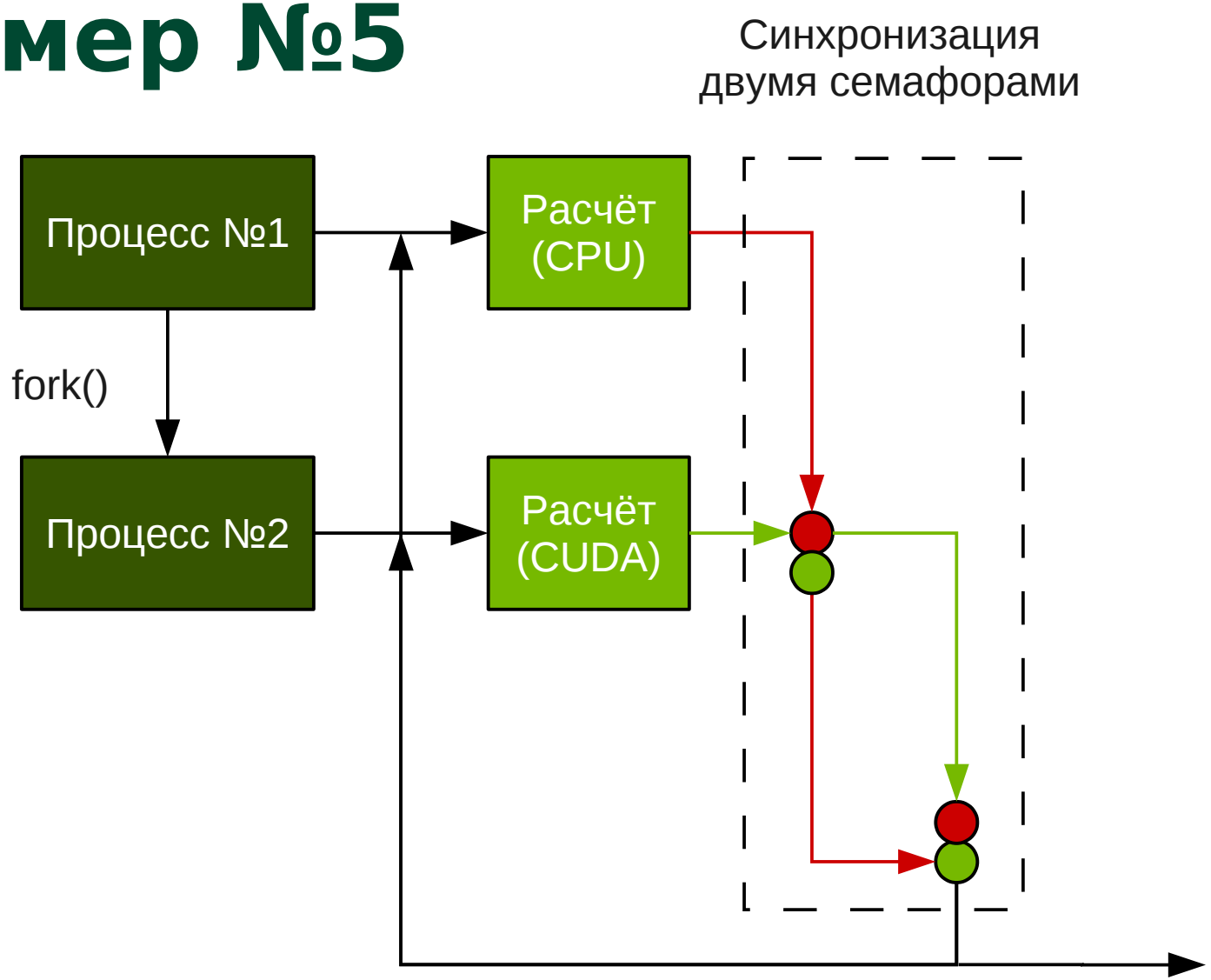
```
// Lock semaphore to begin working with shared data exclusively.
int sem_status = sem_wait(sem);
if (sem_status == -1)
{
        fprintf(stderr,
                "Cannot wait on semaphore by process %d, errno = %d\n",
                pid, errno);
        return errno;
}

// Unlock semaphore to finish working with shared data exclusively.
sem_status = sem_post(sem);
if (sem_status == -1)
{
        fprintf(stderr,
               "Cannot post on semaphore by process %d, errno = %d\n",
               pid, errno);
        return errno;
}
```

Пример №5

Создать несколько процессов, обрабатывающих независимые данные на GPU и CPU и обменивающихся результатами через разделяемый буфер.

Реализация: unix/shmem_mmap_cuda/

Пример №5


Синхронизация
двумя семафорами
Процесс №1
fork()
Процесс №2
Расчёт
(CPU)
Расчёт
(CUDA)

Message Passing Interface (MPI)

- Реализация – библиотека, демон
- Единый код исполняется множеством параллельных процессов
- Демоны MPI контролируют запуск и состояние процессов на узлах вычислительной сети
- Библиотека MPI реализует интерфейс обмена сообщениями (данные, синхронизация) для имеющейся 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, ...

Пример №6

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

Реализация: unix/shmem_mmap_cuda/

MPI_Init / MPI_Finalize

```
// Initialize MPI. From this point the specified
// number of processes will be executed in parallel.
int mpi_status = MPI_Init(&argc, &argv);
if (mpi_status != MPI_SUCCESS)
{
        fprintf(stderr, "Cannot initialize MPI, status = %d\n", mpi_status);
        return mpi_status;
}

mpi_status = MPI_Finalize();
if (mpi_status != MPI_SUCCESS)
{
        fprintf(stderr, "Cannot finalize MPI, status = %d\n",
                mpi_status);
        return mpi_status;
}
```

MPI_Comm_size / *_rank

```
int nprocesses;
mpi_status = MPI_Comm_size(MPI_COMM_WORLD, &nprocesses);
if (mpi_status != MPI_SUCCESS)
{
        fprintf(stderr,
                 "Cannot retrieve the number of MPI processes, status = %d\n",
                mpi_status);
        return mpi_status;
}

// Get the rank (index) of the current MPI process
// in the global communicator.
mpi_status = MPI_Comm_rank(MPI_COMM_WORLD, &config.idevice);
if (mpi_status != MPI_SUCCESS)
{
        fprintf(stderr,
                "Cannot retrieve the rank of current MPI process, status = %d\n",
                mpi_status);
        return mpi_status;
}
```

MPI_Bcast

```
// Let each device to have equal dataset
// in its private array.
mpi_status = MPI_Bcast(config.in_cpu, np, MPI_FLOAT, ndevices,
        MPI_COMM_WORLD);
if (mpi_status != MPI_SUCCESS)
{
        fprintf(stderr,
                "Cannot broadcast input data by process %d, status = %d\n",
                mpi_status);
        return mpi_status;
}
```

MPI_Send / MPI_Recv

```
// On master process perform results check:
// compare each GPU result to CPU result.
if (master)
{
        for (int idevice = 0; idevice < ndevices; idevice++)
        {
                // Receive output from each worker device.
                mpi_status = MPI_Recv(output, np, MPI_FLOAT, idevice, 0,
                        MPI_COMM_WORLD, NULL);
                if (mpi_status != MPI_SUCCESS)
                {
                        fprintf(stderr, "Cannot receive output from device %d, status = %d\n",
                                idevice, mpi_status);
                        return mpi_status;
                }

                // Find the maximum abs difference.
        }
}
else
{
        // Send worker output to master for check.
        MPI_Send(config.in_cpu, np, MPI_FLOAT, ndevices, 0,
                MPI_COMM_WORLD);
        if (mpi_status != MPI_SUCCESS)
        {
                fprintf(stderr, "Cannot send output from device %d, status = %d\n",
                        config.idevice, mpi_status);
                return mpi_status;
        }
}
```

Дополнительные замечания

- Если процессы зависли из-за некорректной синхронизации, то поможет вызов **killall ☻**

```
[marcusmae@T61p process_fork_cuda]$ ps aux | grep fork
500       6909 52.3  0.0  32988  1152 pts/7    R    16:50   0:46 ./process_fork_cuda
500       6911 75.4  0.0  32988  1152 pts/7    R    16:50   1:03 ./process_fork_cuda
500       6913 46.0  0.0  32988  1152 pts/7    R    16:50   0:35 ./process_fork_cuda
500       7013  0.0  0.0 103384   836 pts/7    S+   16:52   0:00 grep --color=auto fork
[marcusmae@T61p process_fork_cuda]$ killall -9 process_fork_cuda
```

POSIX threads (pthread)

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

POSIX threads (pthread)

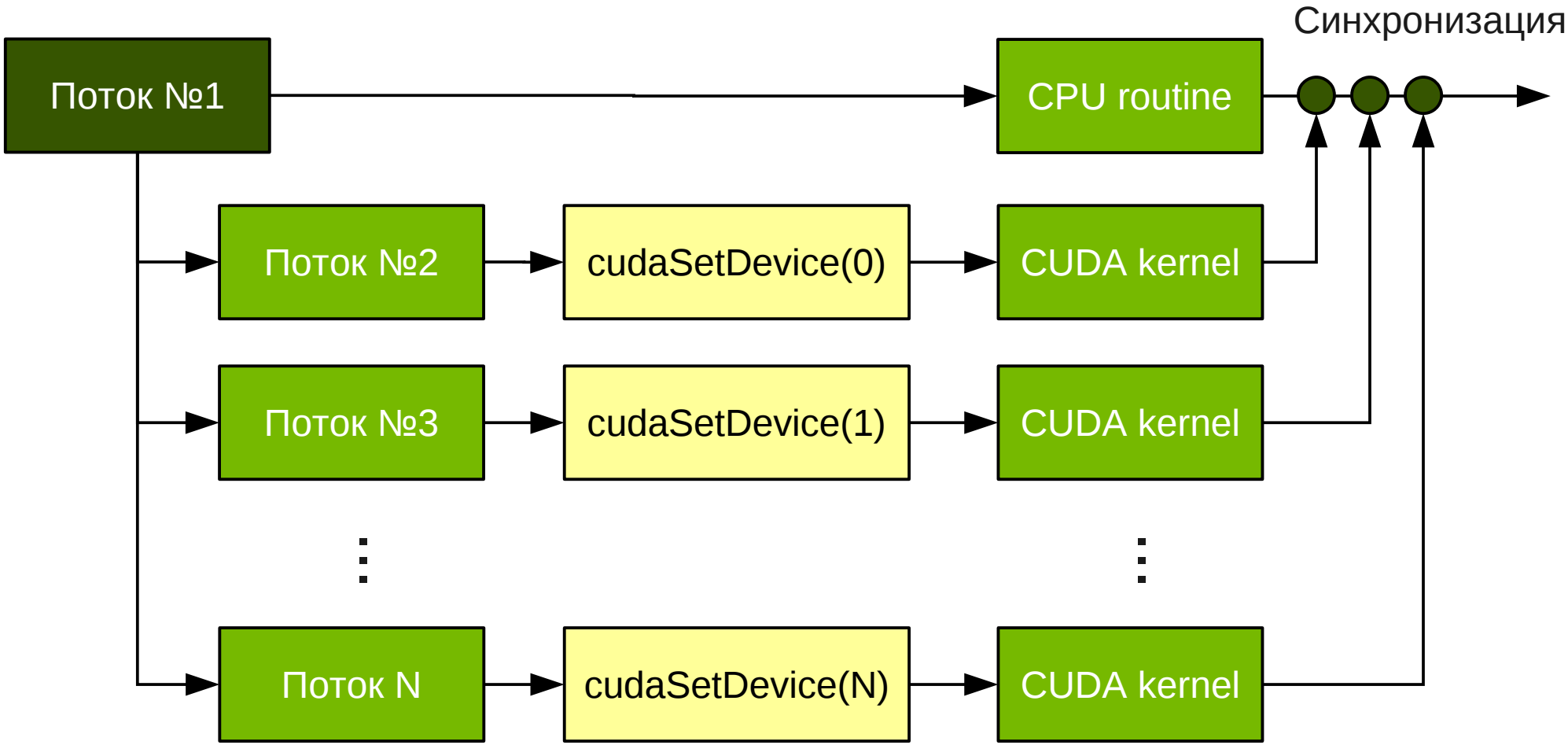
- Порождение и ожидание завершения потока
  - pthread_create, pthread_join
- Критическая секция
  - pthread_mutex_lock, pthread_mutex_unlock, ...
- Барьерная и условная синхронизация
  - pthread_barrier_wait, pthread_cond_wait

Пример №7

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

Реализация: pthread/pthread_cuda/

Пример №7


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

pthread_create

```
// For each CUDA device found create a separate thread
// and execute the thread_func.
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;

        int status = pthread_create(&config->thread, NULL,
                thread_func, config);
        if (status)
        {
                fprintf(stderr,
                        "Cannot create thread for device %d, status = %d\n",
                        i, status);
                return status;
        }
}
```

pthread_exit, pthread_join

```
pthread_exit(NULL);

...

// Wait for device threads completion.
// Check error status.
int status = 0;
for (int i = 0; i < ndevices; i++)
{
        pthread_join(configs[i].thread, NULL);
        status += configs[i].status;
}
```

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 №8

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;
}
```

Замечание к примеру №8

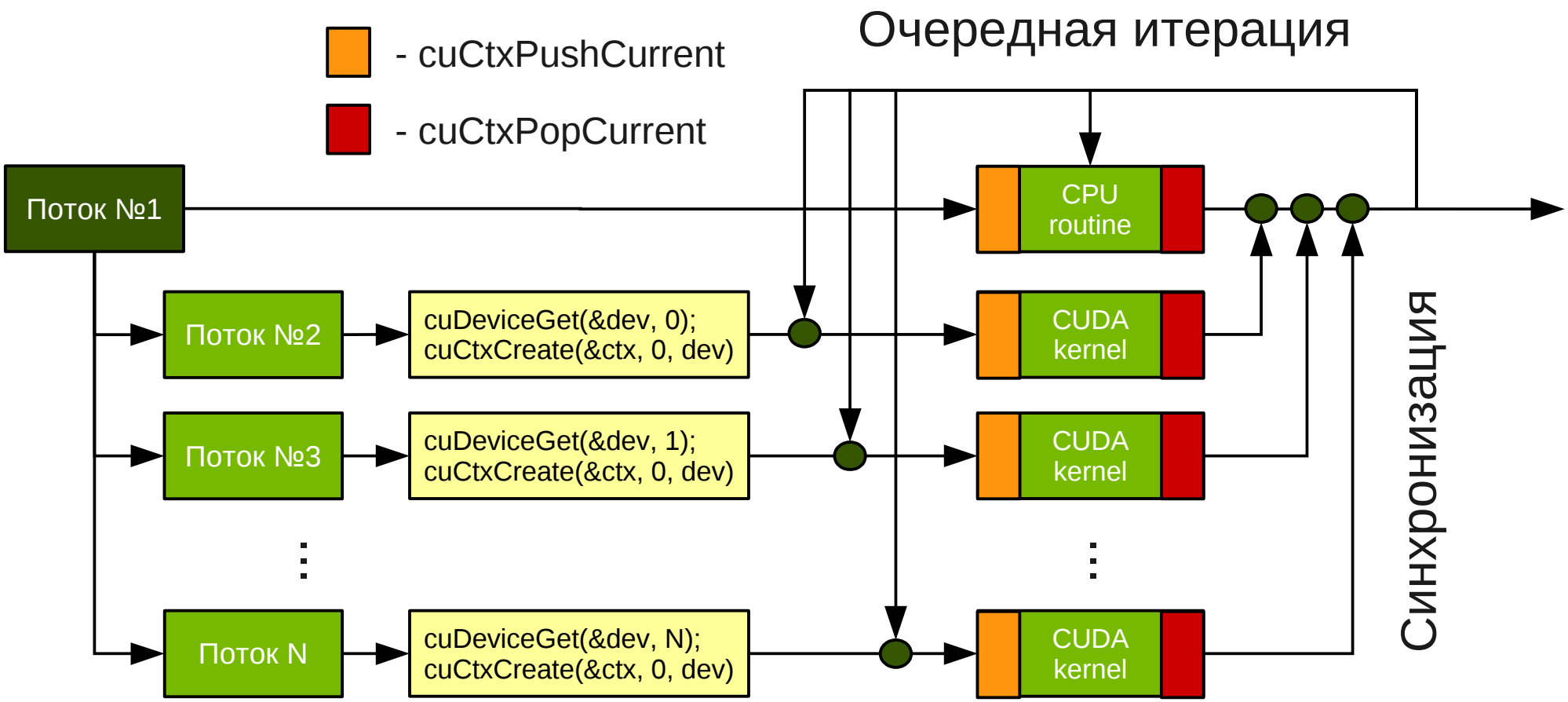
При использовании OpenMP пользователь имеет меньше контроля над рабочими потоками. В случае нескольких параллельных секций не гарантируется неизменный порядок рабочих потоков. Т.е. необходимо проверять, соответствует ли текущий CUDA-контекст потока требуемому (или не оставлять текущих контекстов – такой метод применён в примере №9).

Пример №9

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

Реализация: openmp/openmp_multi_cuda/

Пример №9


- cuCtxPushCurrent
- cuCtxPopCurrent
Очередная итерация
Поток №1
Поток №2
Поток №3
Поток N
cuDeviceGet(&dev, 0);
cuCtxCreate(&ctx, 0, dev)
cuDeviceGet(&dev, 1);
cuCtxCreate(&ctx, 0, dev)
cuDeviceGet(&dev, N);
cuCtxCreate(&ctx, 0, dev)
CPU routine
CUDA kernel
CUDA kernel
CUDA kernel
Синхронизация

Пример №9 (аналогично №1)

- Каждому устройству явно создаётся контекст (cuCtxCreate)
- Перед выполнением операций с устройством соответствующий контекст делается текущим в данном потоке (cuCtxPushCurrent), после операции – снимается (cuCtxPopCurrent)
- В конце контексты удаляются (cuCtxDestroy)

Замечание к примеру №9

Количество OpenMP потоков может быть любым. В частности, если используется только один поток, то им одним будут управляться несколько GPU, благодаря подстановке соответствующих контекстов в качестве текущих.

COACCEL tesla.parallel.ru/trac/coaccel

- Реализация – библиотека
- Унификация интерфейса обмена данными для потоков CPU, CUDA и OpenCL
- Созданием и завершением потоков управляет runtime-библиотека, некоторые свойства потоков могут быть заданы пользователем явно
- Пользователь явно управляет взаимодействием потоков

COACCEL tesla.parallel.ru/trac/coaccel

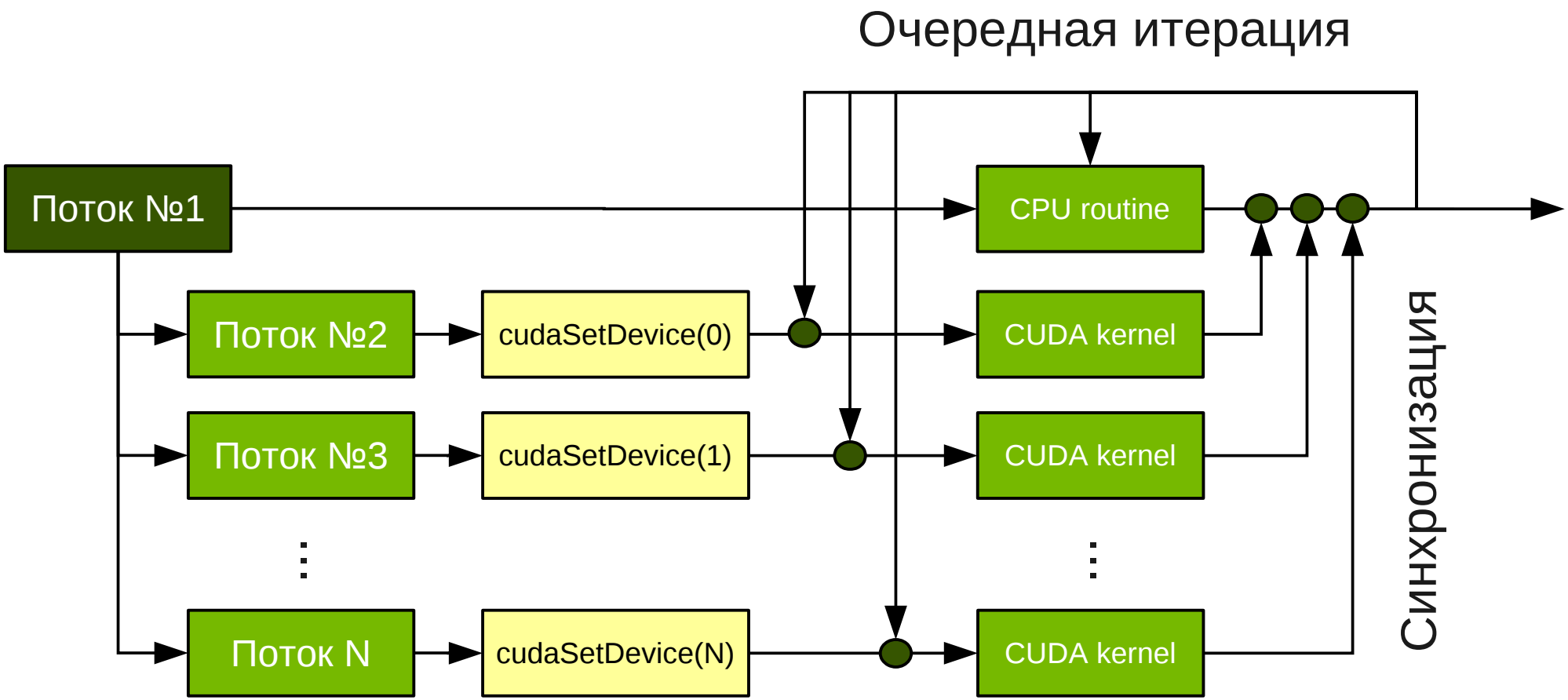
- Создание и группировка CUDA-устройств
  coaccel_device_init,
  coaccel_device_group_create
- Исполнение CUDA-программ на разных устройствах в параллельных потоках
  coaccel_multi_init, coaccel_multi_step
- Синхронизация вычислений на нескольких устройствах

Пример №10

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

Реализация: coaccel/coaccel_multi/

Пример №10


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

coaccel_device_init / *_add

```
// Create new empty COACCEL device group.
coaccel_device_group devices = coaccel_device_group_create();

// Fill group with GPU devices.
for (int i = 0; i < ndevices; i++)
{
        // Initialize COACCEL device supporting CUDA
        // and synchronous memory I/O.
        coaccel_device device = coaccel_device_init(
                argc, argv, COACCEL_DEVMODE_CUDA_SYNC, NULL);
        if (!device)
        {
                fprintf(stderr, "Cannot initialize CUDA device\n");
                return -1;
        }

        // Add created device to group.
        coaccel_device_add(devices, device, i);
}
```

coaccel_device_init / *_add

```
// To show hybrid computations, also create
// CPU device and add it to group.
coaccel_device device = coaccel_device_init(
        argc, argv, COACCEL_DEVMODE_CPU_SYNC, NULL);
if (!device)
{
        fprintf(stderr, "Cannot initialize CPU device\n");
        return -1;
}
coaccel_device_add(devices, device, ndevices);
```

coaccel_multi_init / *_step_all

```
// Initialize threaded execution.
coaccel_multi multi = coaccel_multi_init(
        devices, 1, &init, (void*)configs);
if (!multi)
{
        fprintf(stderr, "Cannot initialize COACCEL multi\n");
        return -1;
}

// Perform several steps of threaded execution.
for (int i = 0; i < nticks; i++)
        coaccel_multi_step_all(multi, process, (void*)configs);

// Finalize threaded execution.
coaccel_multi_finalize(multi, &deinit, (void*)configs);
```

coaccel_device_dispose

```
// Dispose devices and group.
for (int i = 0; i < ndevices + 1; i++)
{
        coaccel_device_dispose(
                coaccel_device_get(devices, i));
}
coaccel_device_group_dispose(devices);
```

Boost

- Создание потока

  boost::thread, boost::bind

- Синхронизация

  boost::mutex, boost::barrier

Пример №11

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

Реализация: boost/boost_cuda/

thread, bind, mutex, barrier

```
static boost::mutex m;
boost::barrier* b1, b2;
boost::thread t;

// The function executed by each thread assigned with CUDA device.
void ThreadRunner::thread_func()
{
        ...
}

ThreadRunner::ThreadRunner(int idevice, int nx, int ny, boost::barrier* b) :
        t(boost::bind(&ThreadRunner::thread_func, this)), b2(2), finish(0)
...
```

thread, bind, mutex, barrier

```
// Create a barrier that will wait for (ndevices + 1)
// invocations of wait().
boost::barrier b(ndevices + 1);

// Initialize thread runners and load input data.
ThreadRunner** runners = new ThreadRunner*[ndevices + 1];
for (int i = 0; i < ndevices; i++)
{
        runners[i] = new ThreadRunner(i, nx, ny, &b);
        runners[i]->Load(data);
}
```

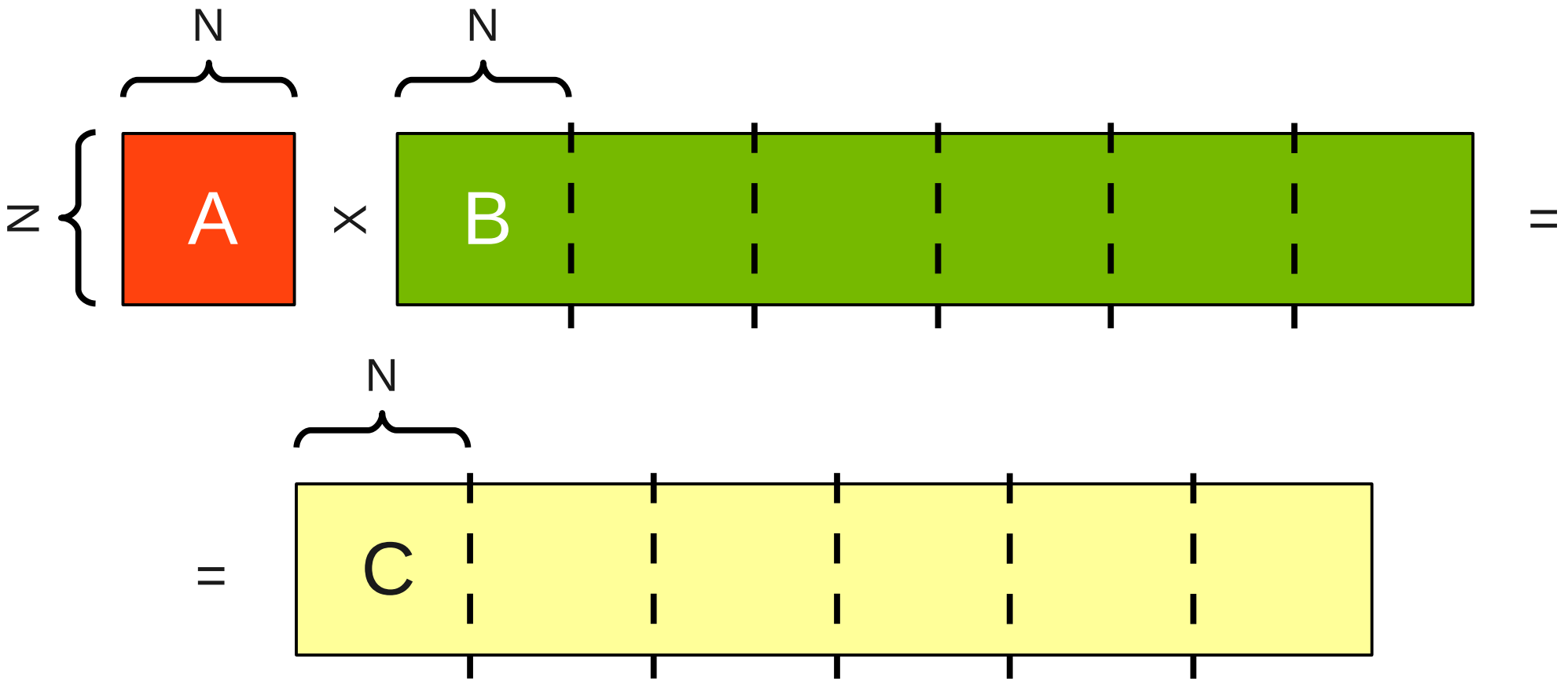
thread, bind, mutex, barrier

```
// Compute the given number of steps.
float* input = data;
float* output = data + np;
for (int i = 0; i < nticks; i++)
{
        // Pass iteration on device threads.
        for (int i = 0; i < ndevices; i++)
                runners[i]->Pass();

        int status = ThreadRunner::GetLastError();
        if (status) return status;

        // In parallel main thread launch CPU function equivalent
        // to CUDA kernels, to check the results.
        pattern2d_cpu(1, nx, 1, 1, ny, 1,
                input, output, ndevices);
        float* swap = output;
        output = input;
        input = swap;

        b.wait();
}
```

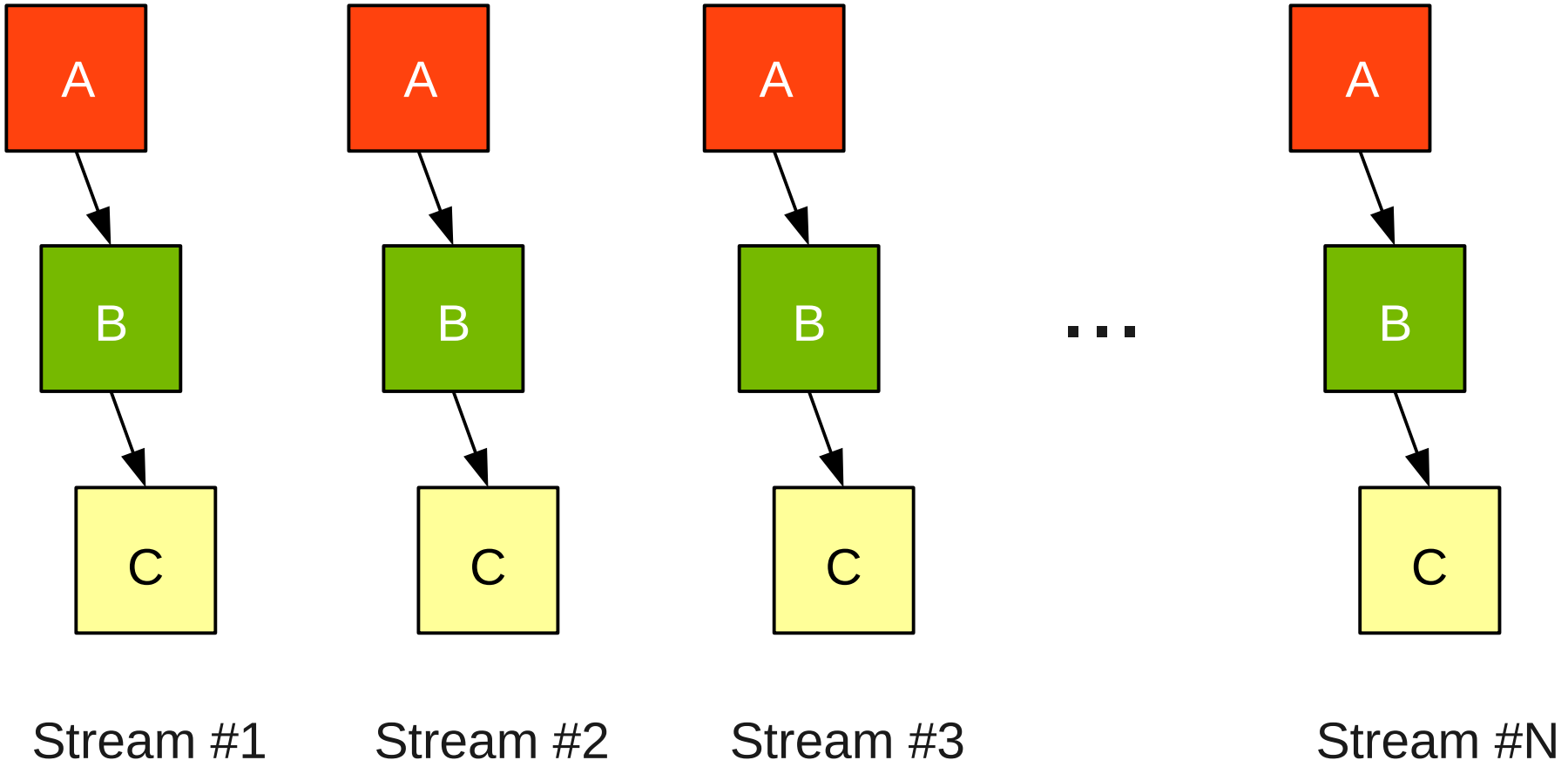
CUDA Streams

Потоки (streams) – метод выделения последовательностей асинхронных операций, связанных порядком действий в составе одного stream, но независимых для различных streams.

CUDA Streams



CUDA Streams



CUDA Streams

- Параллельные цепочки streams могут эффективнее насыщать вычислительные блоки за счёт передачи данных малыми независимыми блоками.
- Асинхронные операции передачи данных требуют pinned memory (cudaMallocHost).

Пример №12

Блочное перемножение матриц с использованием CUDA streams.

Реализация: gemm_streams/

```
[dmikushin@tesla-cmc gemm_streamed]$ ./gemm_streamed 4 1024 1025 1 N N 1.0 0.0 16
n       time            gflops          test    enorm       rnorm
1024    0.013909 sec    154.390014      PASSED  0.018676    262177.187500
1024    0.011231 sec    191.204784      PASSED  0.018693    262323.812500

[dmikushin@tesla-cmc gemm_streamed]$ ./gemm_streamed 4 4096 4097 1 N N 1.0 0.0 16
n       time            gflops          test    enorm       rnorm
4096    0.431783 sec    318.305308      PASSED  0.293618    4194287.250000
4096    0.396524 sec    346.609298      PASSED  0.293707    4194416.500000
```

cudaStreamCreate / *Destroy

```
cudaStream_t* stream = (cudaStream_t*)malloc(
        nstreams * sizeof(cudaStream_t));


// Create streams
for (int i = 0; i < nstreams; i++)
{
        cudaerr = cudaStreamCreate(&stream[i]);
        assert(cudaerr == cudaSuccess);
}

// Destroy streams
for (int istream = 0; istream < nstreams; istream++)
{
        cudaerr = cudaStreamDestroy(stream[istream]);
        assert(cudaerr == cudaSuccess);
}
```

cublasSetVectorAsync

Асинхронная загрузка данных на устройство (аналог cudaMemcpyAsync)

```
for (int istream = 0; istream < nstreams; istream++)
{
        int szpart = n / nstreams;
        size_t shift = n * szpart * istream;
        if (istream == nstreams - 1)
                szpart += n % nstreams;

        status = cublasSetVectorAsync(n * szpart, sizeof(real),
                h_B + shift, 1, d_B[istream], 1, stream[istream]);
        assert(status == CUBLAS_STATUS_SUCCESS);
        status = cublasSetVectorAsync(n * szpart, sizeof(real),
                h_C + shift, 1, d_C[istream], 1, stream[istream]);
        assert(status == CUBLAS_STATUS_SUCCESS);
}
```

cublasSetKernelStream

Установка stream для CUDA-ядра

```
for (int istream = 0; istream < nstreams; istream++)
{
        int szpart = n / nstreams;
        if (istream == nstreams - 1)
                szpart += n % nstreams;

        // Setup async operations
        status = cublasSetKernelStream(stream[istream]);
        assert(status == CUBLAS_STATUS_SUCCESS);

        // Perform matmul using CUBLAS
        cublas_gemm(transa, transb, n, szpart, n,
                alpha, d_A, n, d_B[istream], n, beta, d_C[istream], n);
        status = cublasGetError();
        assert(status == CUBLAS_STATUS_SUCCESS);
}
```

cublasGetVectorAsync

Асинхронная выгрузка данных из устройства (аналог cudaMemcpyAsync)

```
// Sync all
for (int istream = 0; istream < nstreams; istream++)
{
        int szpart = n / nstreams;
        size_t shift = n * szpart * istream;
        if (istream == nstreams - 1)
                szpart += n % nstreams;

        // Read the result back
        status = cublasGetVectorAsync(n * szpart, sizeof(real),
                d_C[istream], 1, h_C + shift, 1, stream[istream]);
        assert(status == CUBLAS_STATUS_SUCCESS);
}
```

cudaStreamSynchronize

Ожидание выполнения всех операций в заданном stream

```
cudaStreamSynchronize(stream[istream]);
```

Заключение

- Код на CUDA может взаимодействовать с другими программными моделями параллельных вычислений
- Методы из предложенных примеров могут быть использованы в Ваших приложениях и протестированы на сервере tesla-cmc