Programming multi-GPU systems

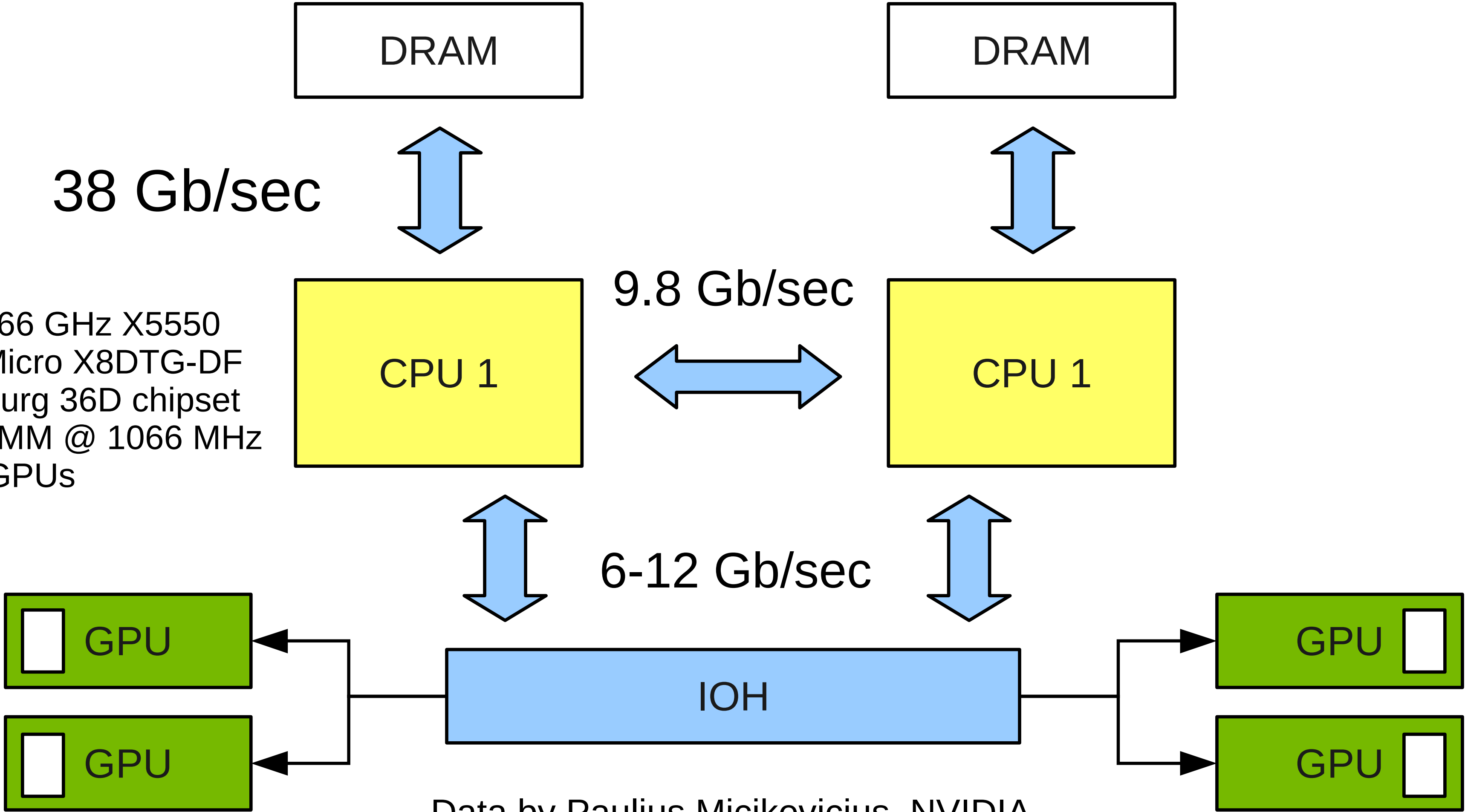
Dmitry Mikushin

Agenda

- Hybrid architectures
- Memory hierarchy, processes, threads
- GPU context
- Examples of multi-GPU apps
- Asynchronous operations, CUDA streams

Modern hybrid system

Dual 2.66 GHz X5550
SuperMicro X8DTG-DF
Tylersburg 36D chipset
12 x DIMM @ 1066 MHz
Fermi GPUs

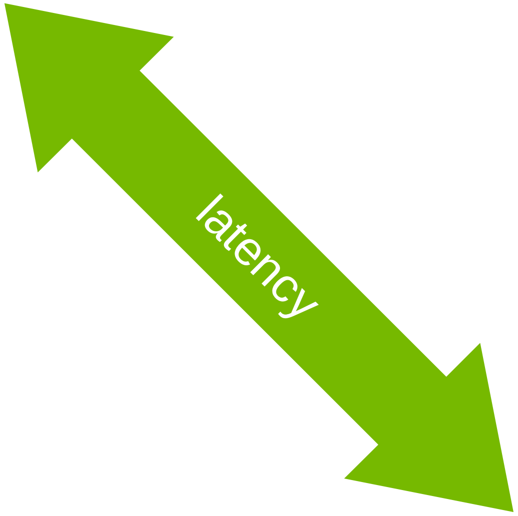


Data by Paulius Micikevicius, NVIDIA

2011

Memory hierarchy

- Core cache, cpu cache
- Near RAM
- Far RAM (over QPI)
- PCI-E device RAM
- Other cluster node host RAM
- Other node's PCI-E device RAM



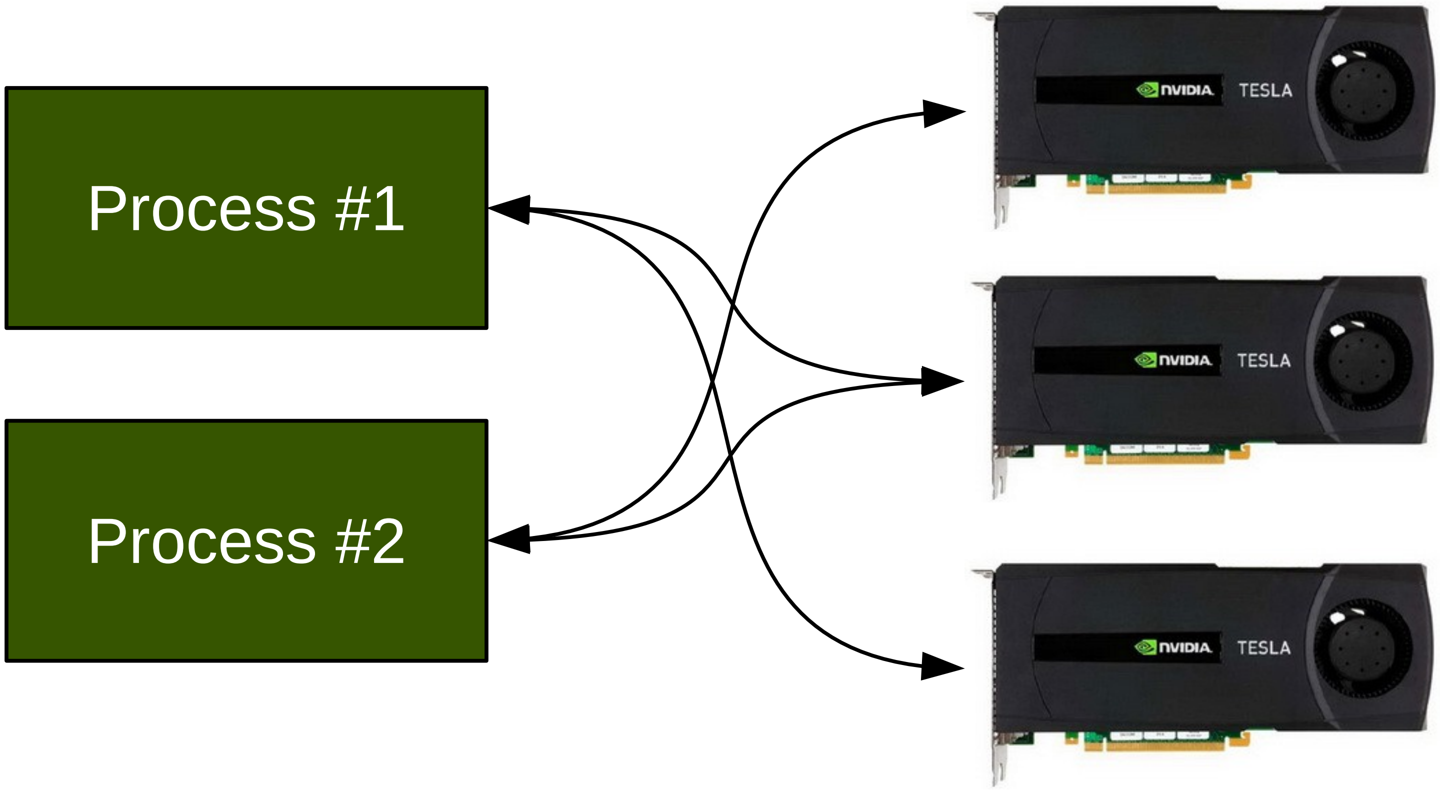
Role of CUDA

- **CUDA** – API for computations on GPU, but not for the whole cluster
- To develop for hybrid systems we need to combine different programming models

OS basics: processes

- Own *context* in system (memory allocations, files, etc)
- Processes are controlled with syscalls (quiet expensive)

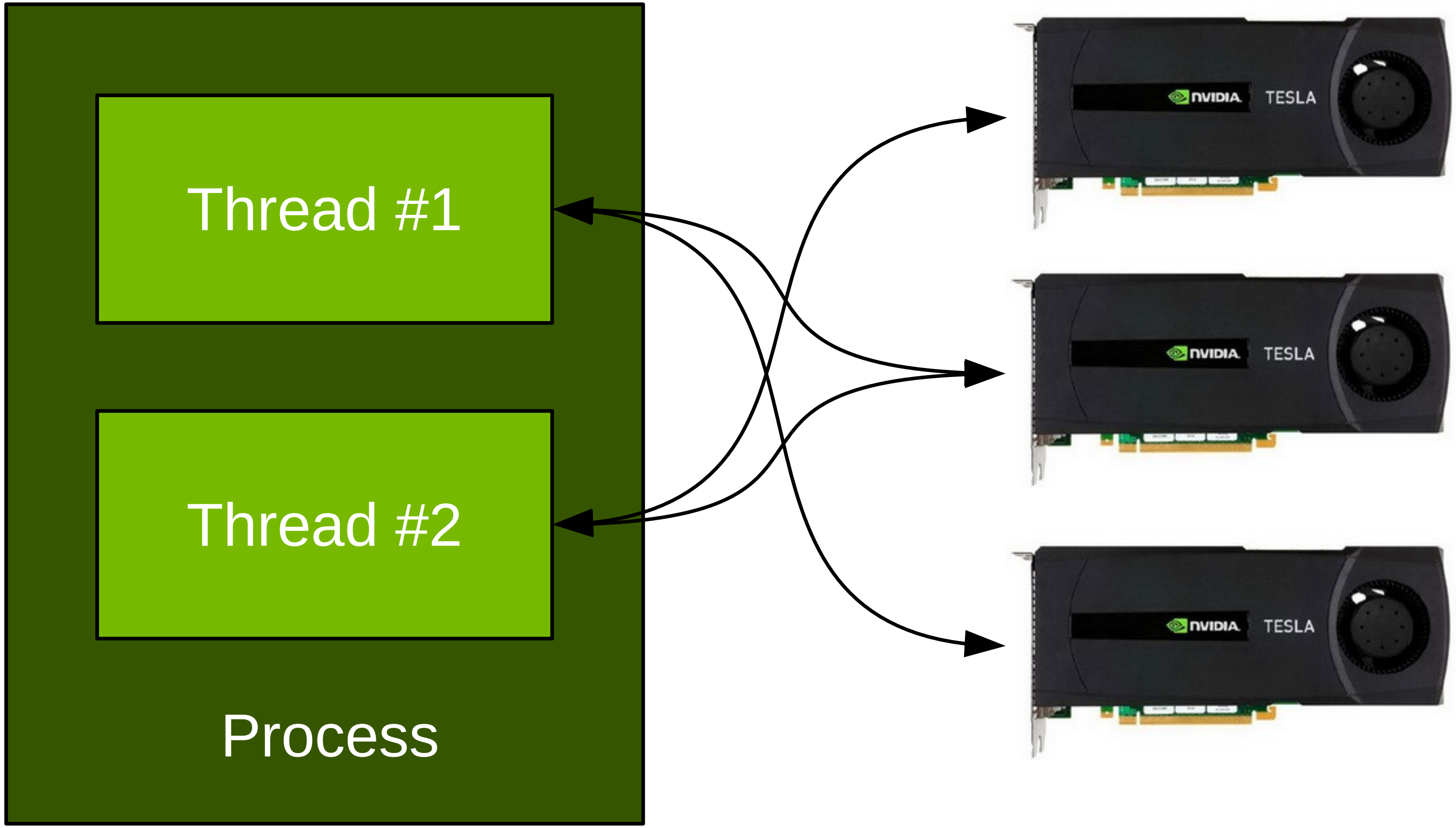
OS basics: processes



OS basics: threads

- Share memory of parent process
- OS might be less responsible for threads management

OS basics: threads



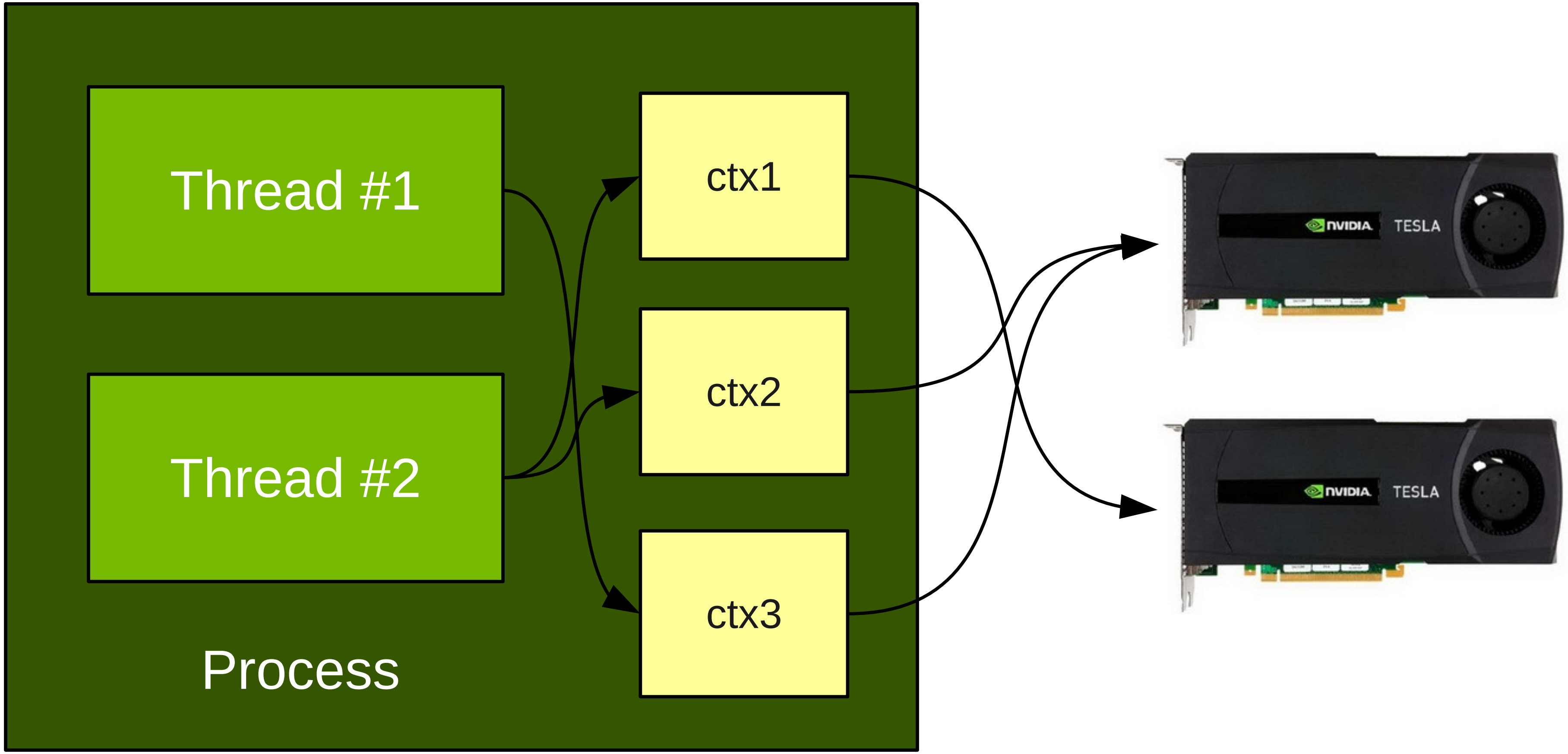
CUDA device context

- CUDA context – device-specific runtime configuration info (allocated device-memory, error codes, etc)
- Many CUDA calls require existing context

CUDA device context

- Initially thread/process does not have current CUDA context
- If thread/process does not have CUDA context, but it is required, then it will be created implicitly
- One device can have multiple contexts

CUDA device context



CUDA device context

- Each process/thread is allowed to have only one *current* (focused) context
- Process/thread can create/destroy contexts and switch current context

multi-GPU in serial apps

See full examples source code in attached materials

Example #1

Create a serial app, utilizing multiple GPUs in parallel.

Source code: serial/serial_cuda/

Example #1

- For each device create a context explicitly (cuCtxCreate)
- Before interacting with device, set the corresponding context as current (cuCtxPushCurrent) and unset it back afterwards (cuCtxPopCurrent)
- In the end destroy all contexts (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 in parallel apps

This section briefly explains how to combine different parallel programming models with CUDA

See full examples source code in attached materials

Open Group / IEEE

- Create child process
  - fork
- Shared regions
  - shm_open, shm_unlink
- Memory mapping
  - mmap, munmap, msync
- Semaphores
  - sem_open, sem_wait, sem_post, sem_unlink

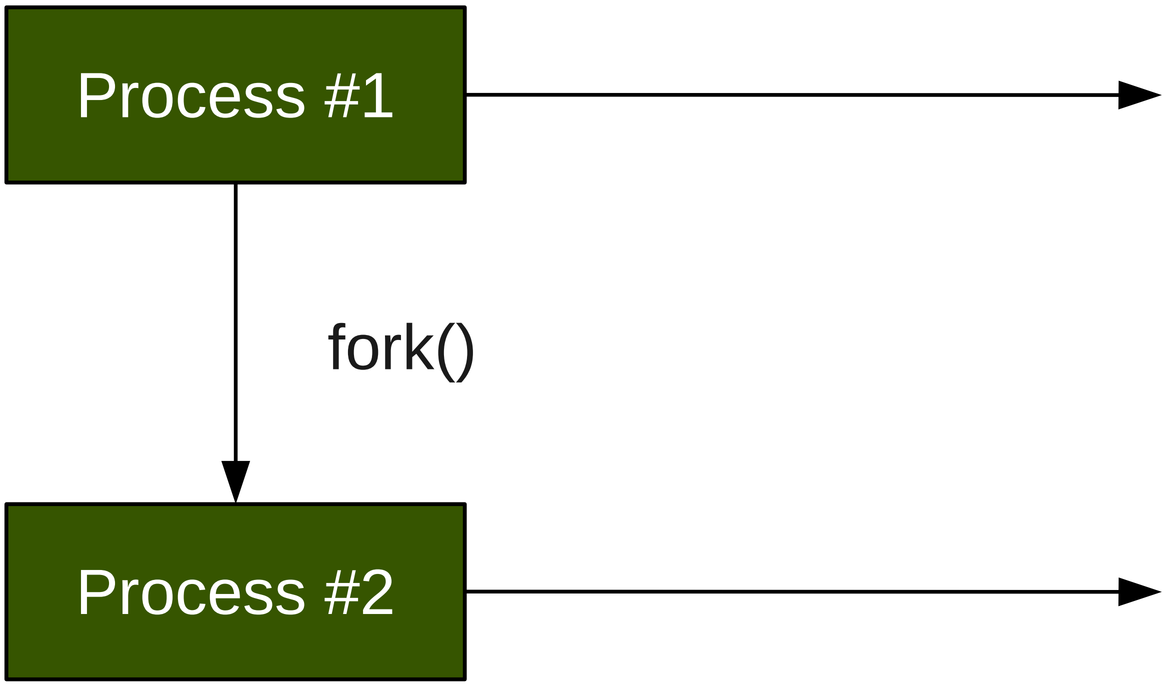
2011

Example #2

Create multiple processes handling independent datasets on CPU

Source code: unix/process_fork/

Example #2



fork()

```c
// 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();
```

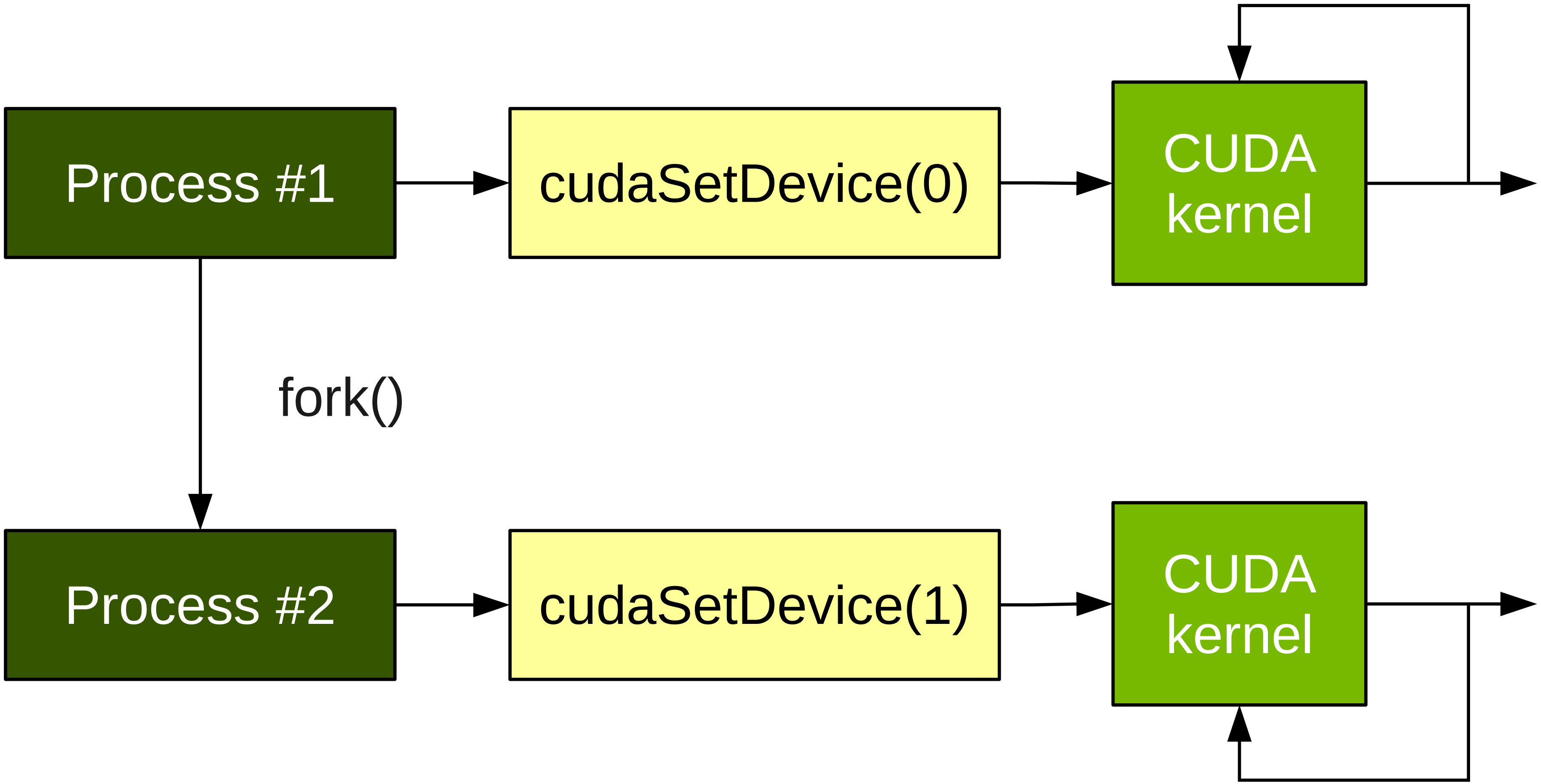
Example #3

Create multiple processes handling independent datasets on one or multiple GPUs

If system has only one GPU, use it in all processes

Source code: unix/process_fork_cuda/

Example #3


Process #1
cudaSetDevice(0)
CUDA kernel
fork()
Process #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);
}
```

Note on example #3

If CUDA context was created **before** fork(), then in child process CUDA calls may behave incorrectly

Example #4

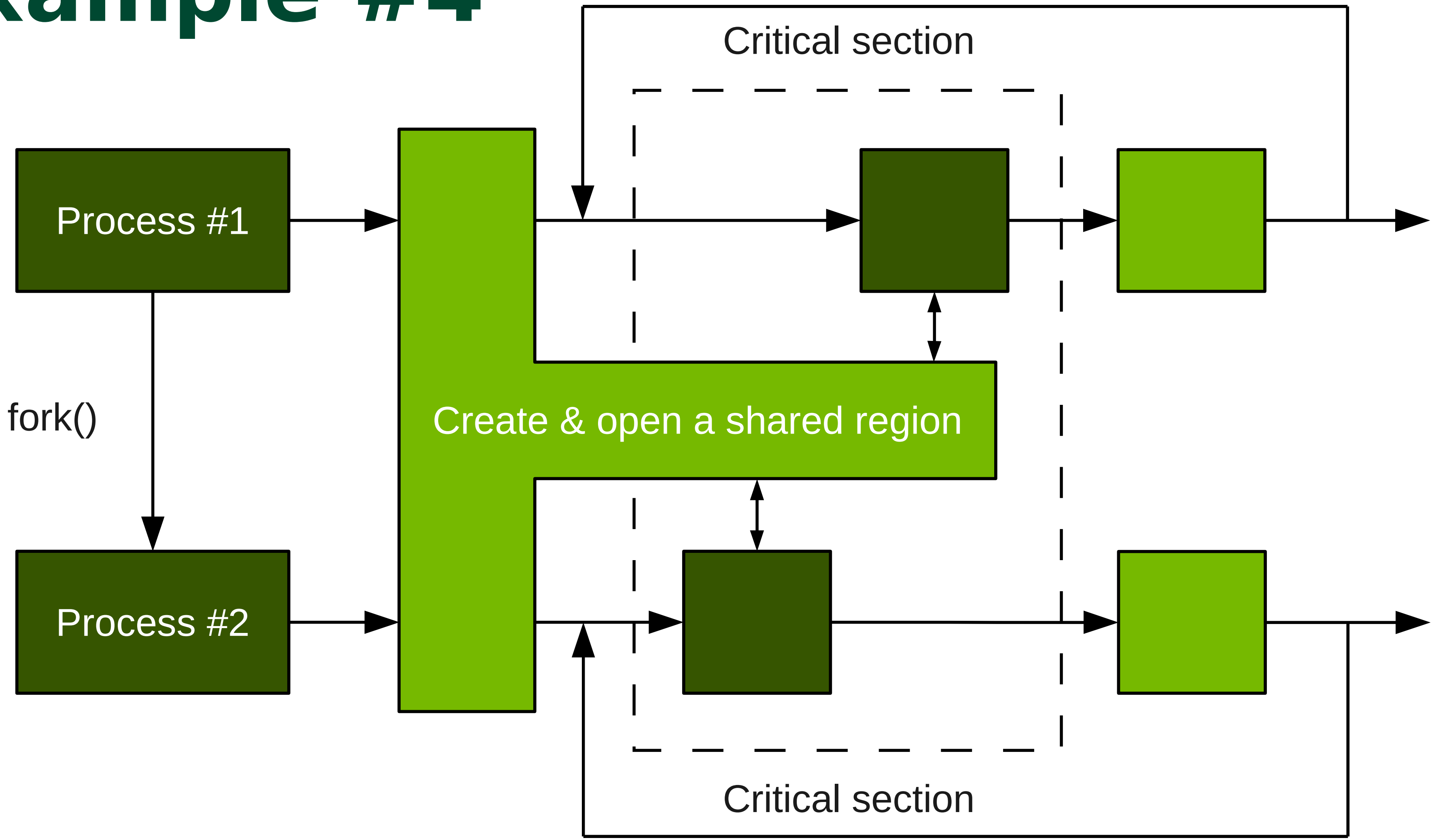
Create multiple processes handling independent datasets and sharing them over shared buffer

Source code: unix/shmem_mmap/

Some synchronization primitives

- Mutex (1 room – 1 key)
- Semaphore (1 room – multiple keys)
- Barrier (elevator)

Example #4



Critical section
Process #1
fork()
Create & open a shared region
Process #2
Critical section

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

```c
// 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

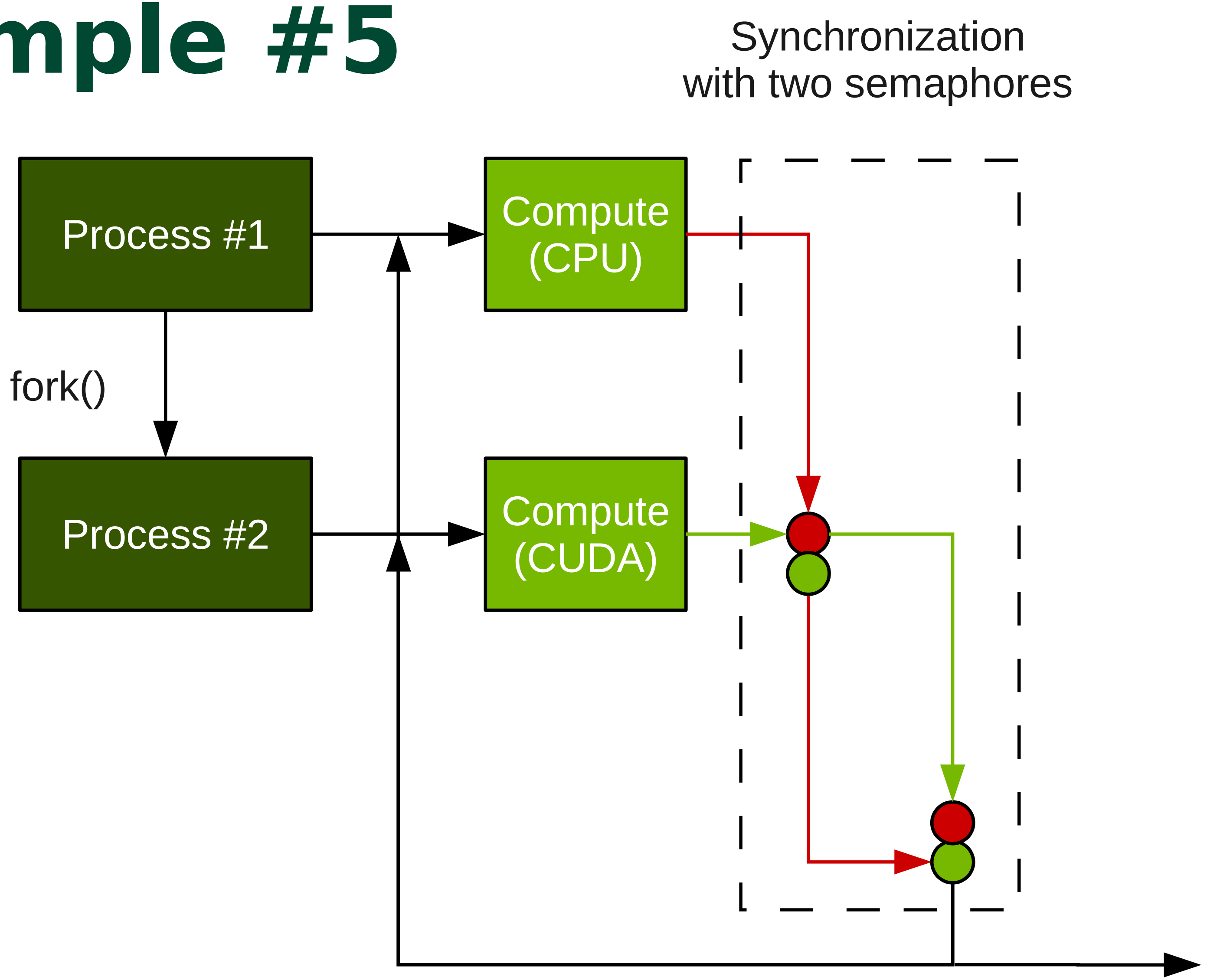
```
// Lock semaphore to begin working with shared data exclusively.
int sem_status = sem_wait(sem);
if (sem_status == -1)
{
        fprintf(stderr,
               "Cannot wait on semaphore by process %d, errno = %d\n",
                pid, errno);
        return errno;
}

// 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;
}
```

Example #5

Create multiple processes handling independent datasets on GPU and CPU and sharing results over shared buffer

Source code: unix/shmem_mmap_cuda/

Example #5


Synchronization
with two semaphores
Process #1
Process #2
fork()
Compute
(CPU)
Compute
(CUDA)

Message Passing Interface (MPI)

- Implementation: library, daemon
- Same binary is executed by multiple processes in parallel
- Local MPI daemons running on each node control processes startup and manage their states
- MPI runtime library implements message passing (data, synchronization) for nodes network

Message Passing Interface (MPI)

- Spawn parallel processes

  mpirun, mpiexec

- Init / deinit MPI in application

  MPI_Init, MPI_Finalize

- Data transfer

  MPI_Send, MPI_Recv, MPI_Bcast, ...

- Synchronization

  MPI_Barrier, ...

Example #6

With help of MPI create mutiple processes handling independent datasets on GPUs and sharing results over shared buffer

Source code: 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;
        }
}
```

Notes on processes

- If processes deadlocked due to incorrect synchronization, the **killall** util should help ☻

```
[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)

- Implementation: library
- User has explicit control on threads creation, deletion and their properties
- User has explicit control on threads interaction

POSIX threads (pthread)

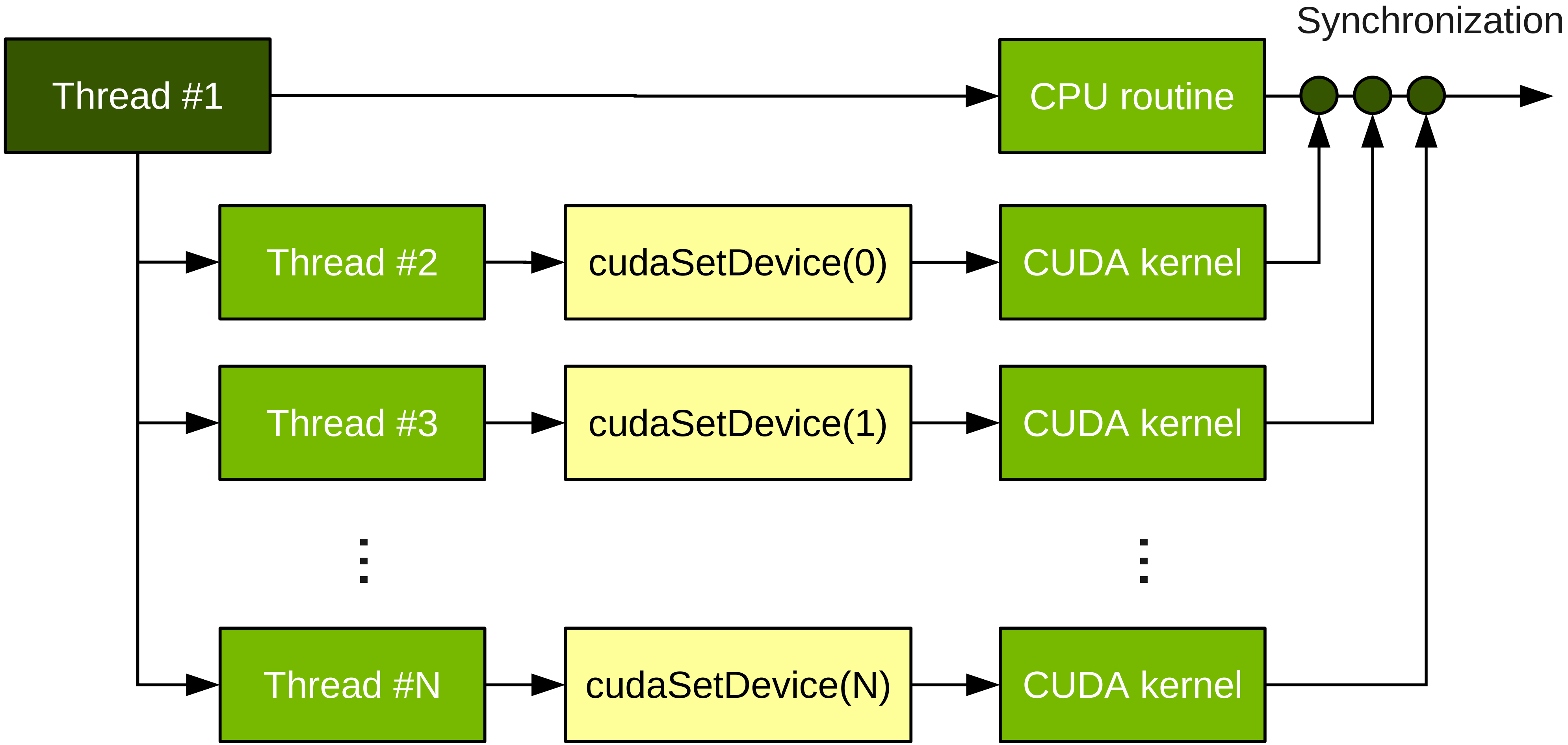
- Create thread, wait for thread to finish
  - pthread_create, pthread_join
- Critical section
  - pthread_mutex_lock, pthread_mutex_unlock, ...
- Barrier and conditional variable
  - pthread_barrier_wait, pthread_cond_wait

Example #7

With help of pthread create multiple threads handling independent datasets on GPUs and CPU

Source code: pthread/pthread_cuda/

Example #7


Synchronization
Thread #1
CPU routine
Thread #2
cudaSetDevice(0)
CUDA kernel
Thread #3
cudaSetDevice(1)
CUDA kernel
⋮
⋮
Thread #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

- Implementation: directives (language extensions for C, Fortran, etc) and library
- Runtime library manages threads creation and deletion, some threads properties can be controlled by user
- User has explicit control on threads interaction

OpenMP

- Parallel execution

  #pragma omp parallel

- Threads count

  omp_get_num_threads(), OMP_NUM_THREADS

- Parallel loops

  #pragma omp parallel for

Example #8

With help of OpenMP create multiple threads handling independent datasets on GPUs and CPU

Source code: 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;
}
```

Note on example #8

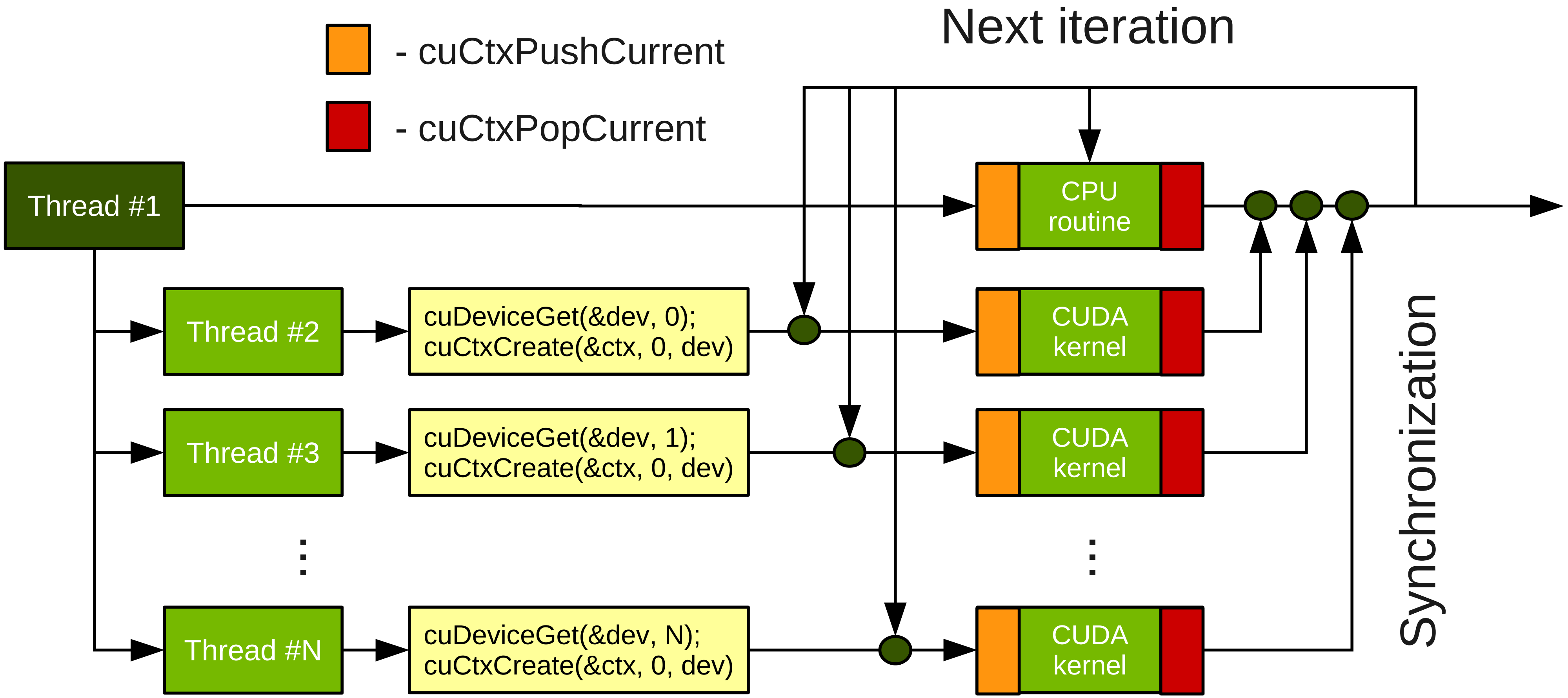
With OpenMP user has less control on worker threads. In case of multiple parallel loops the conservation of threads ⇔ grid mapping is not guaranteed, resulting into possible current CUDA contexts mismatches. Thus, it should be better to always push/pop context in the beginning/end of OpenMP-enabled loop body (see example #9).

Example #9

With help of OpenMP create multiple threads handling independent datasets on GPUs and CPU with synchronization on each step

Source code: openmp/openmp_multi_cuda/

Example #9



- cuCtxPushCurrent
- cuCtxPopCurrent
Next iteration
Thread #1
CPU routine
Thread #2
cuDeviceGet(&dev, 0);
cuCtxCreate(&ctx, 0, dev)
CUDA kernel
Thread #3
cuDeviceGet(&dev, 1);
cuCtxCreate(&ctx, 0, dev)
CUDA kernel
Thread #N
cuDeviceGet(&dev, N);
cuCtxCreate(&ctx, 0, dev)
CUDA kernel
Synchronization

Example #9 (similar to #1)

- For each device create a context explicitly (cuCtxCreate)
- Before interacting with device, set the corresponding context as current (cuCtxPushCurrent) and unset it back afterwards (cuCtxPopCurrent)
- In the end destroy all contexts (cuCtxDestroy)

Note on example #9

There could be any number of OpenMP threads. For instance, if only one thread is used, it will successfully control multiple GPUs, thanks to current CUDA context switching.

COACCEL tesla.parallel.ru/trac/coaccel

- Implementation: library
- Unified data I/O interface for CPU threads, CUDA and OpenCL devices
- Runtime library manages threads creation and deletion, some threads properties can be controlled by user
- User has explicit control on threads interaction

COACCEL tesla.parallel.ru/trac/coaccel

- Creation and grouping of CUDA devices

  coaccel_device_init,
  coaccel_device_group_create

- Execution of CUDA kernels on multiple devices in parallel threads

  coaccel_multi_init, coaccel_multi_step

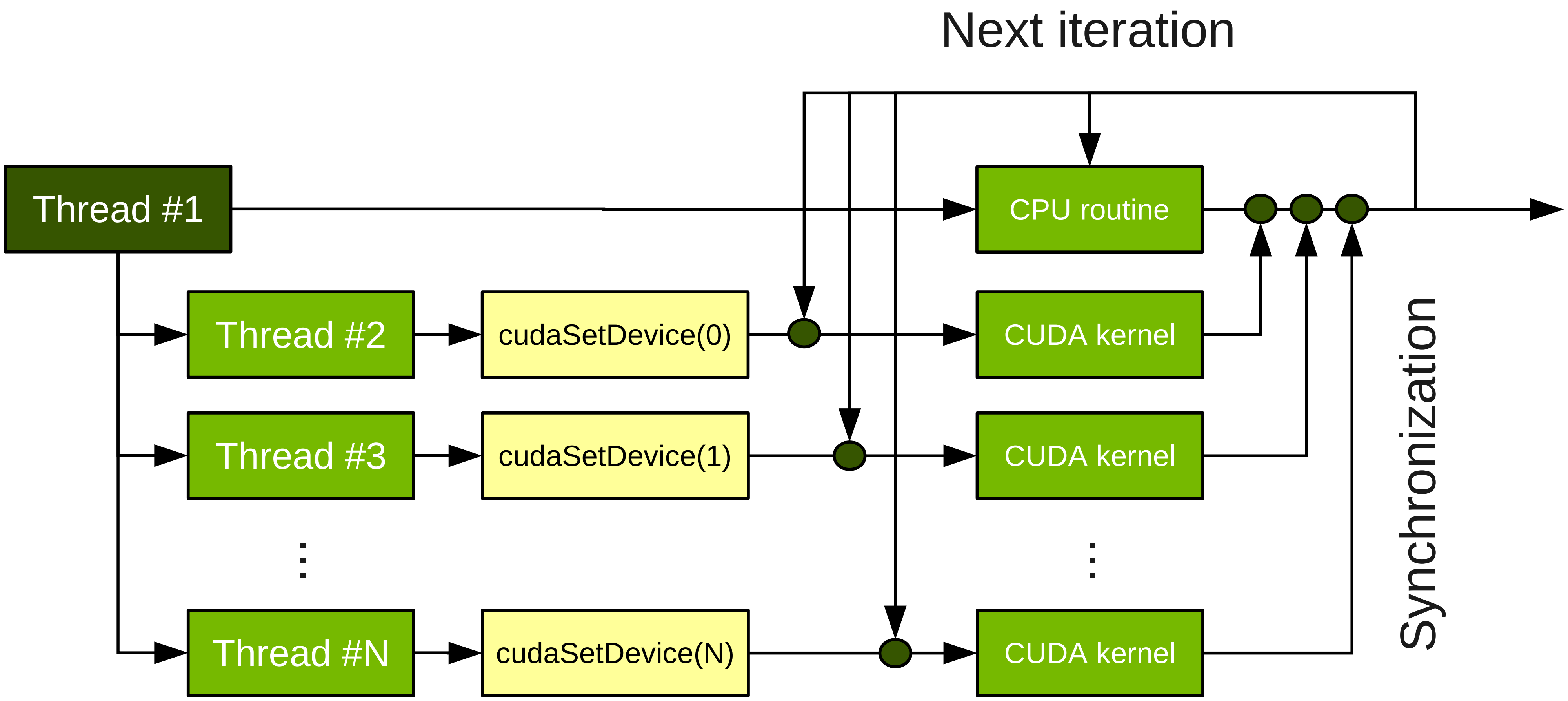
- Synchronization of multiple devices

Example #10

With help of COACCEL create multiple threads handling independent datasets on GPUs and CPU with synchronization on each step

Source code: coaccel/coaccel_multi/

Example #10



Next iteration
Thread #1
CPU routine
Thread #2
cudaSetDevice(0)
CUDA kernel
Thread #3
cudaSetDevice(1)
CUDA kernel
Thread #N
cudaSetDevice(N)
CUDA kernel
Synchronization

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

- Create and bind thread

  boost::thread, boost::bind

- Synchronization

  boost::mutex, boost::barrier

Example #11

With help of Boost create multiple threads handling independent datasets on GPUs and CPU with synchronization on each step

Source code: 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);
}
```

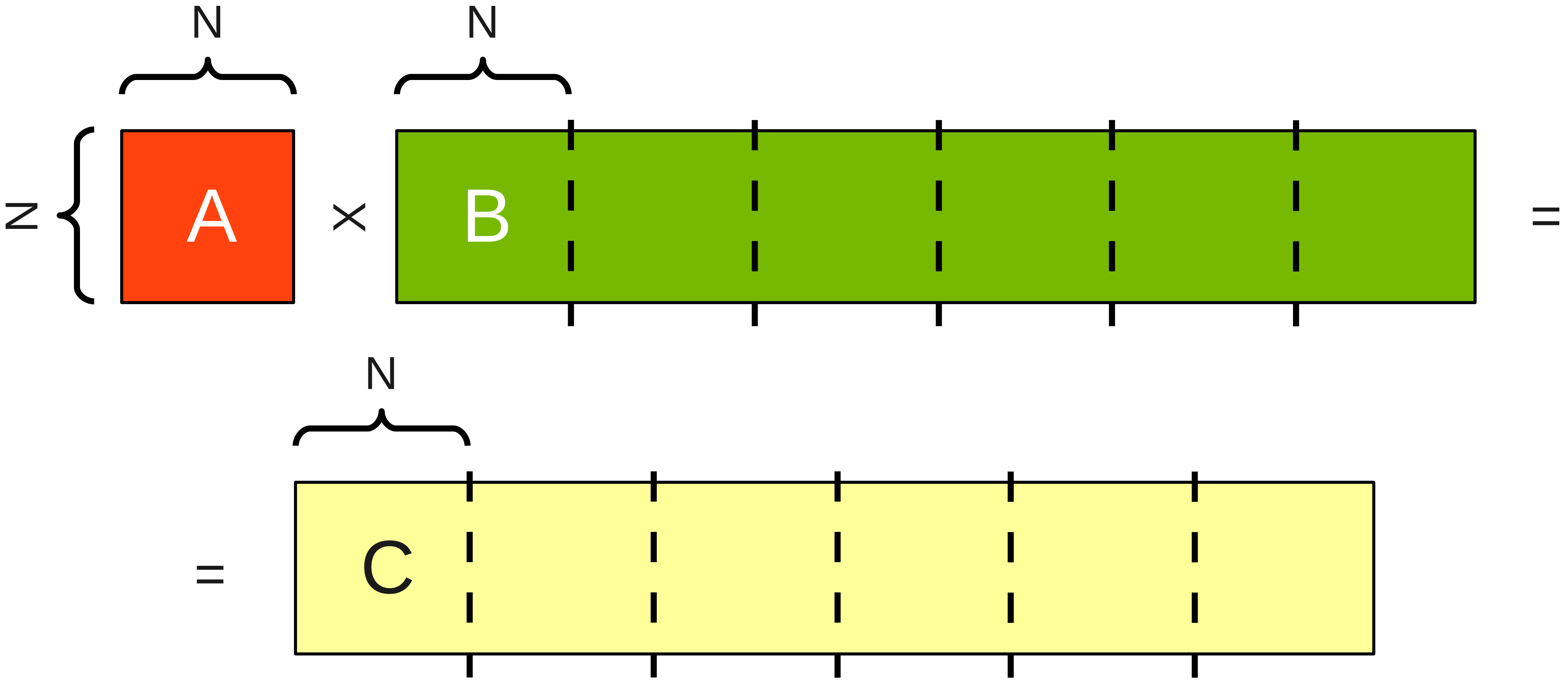
thread, bind, mutex, barrier

```
// 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();
}
```

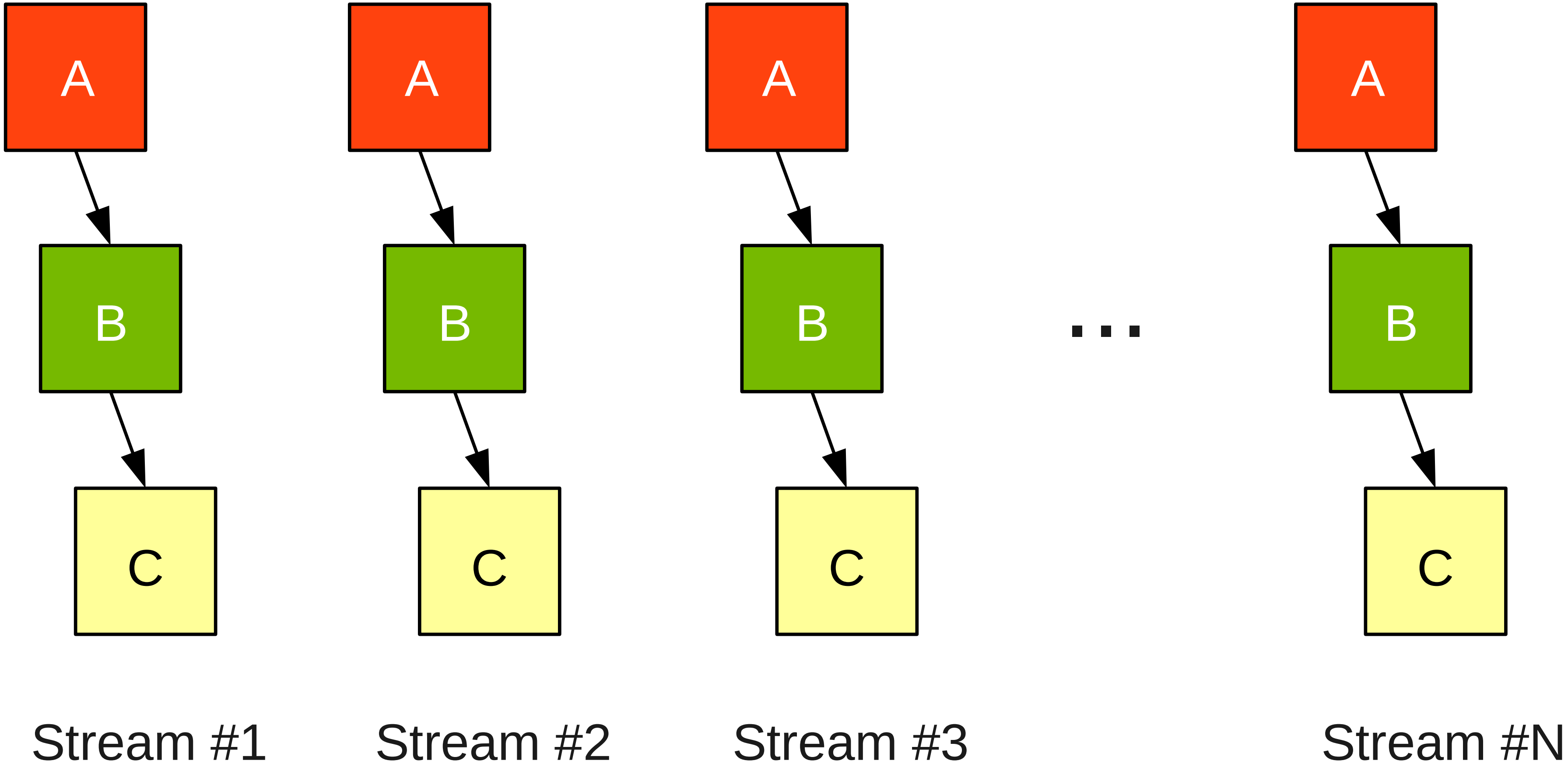
CUDA Streams

Stream – a logical sequence of dependent asynchronous operations, independent from other streams

CUDA Streams



CUDA Streams



CUDA Streams

- Parallel streams can potentially be more efficient that regular execution on combination of kernels and host ⇔ device transfers, feeding both I/O and computational units simultaneously
- Asynchronous data transfers require pinned memory (cudaMallocHost).

Example #12

Block matrix multiplication with CUDA streams.

Source code: 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

Asynchronous data loading on device (equivalent to 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

Setting stream for kernels in cublas call

```
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

Asynchronous data unloading from device (equivalent to 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

Wait for all operations to be finished in the specified stream

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

Conclusion

- CUDA program can interact with many other parallel programming APIs
- Techniques from the presented examples can be applied to *your* applications and tested on our tesla-cmc server