Технология CUDA для высокопроизводительных вычислений на кластерах с графическими процессорами

Колганов Александр
alexander.k.s@mail.ru
часть 6

Multi-gpu

Использование нескольких GPU
CUDA+openmp
CUDA+MPI
P2P обмены между GPU

CUDA Context

CUDA Context

- Аналог процесса CPU
  - Выделения памяти, выполнение операций происходит в рамках некоторого контекста (=процесса)
  - Отдельное адресное пространство
- Выделенная память неявно освобождается при удалении контекста
- Операции из разных контекстов не могут выполнятся параллельно

CUDA context
- Адресное пространство
- Ресурсы
- Операции

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 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 & cudaStream/Event

- `cudaStream{Event}Create` 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

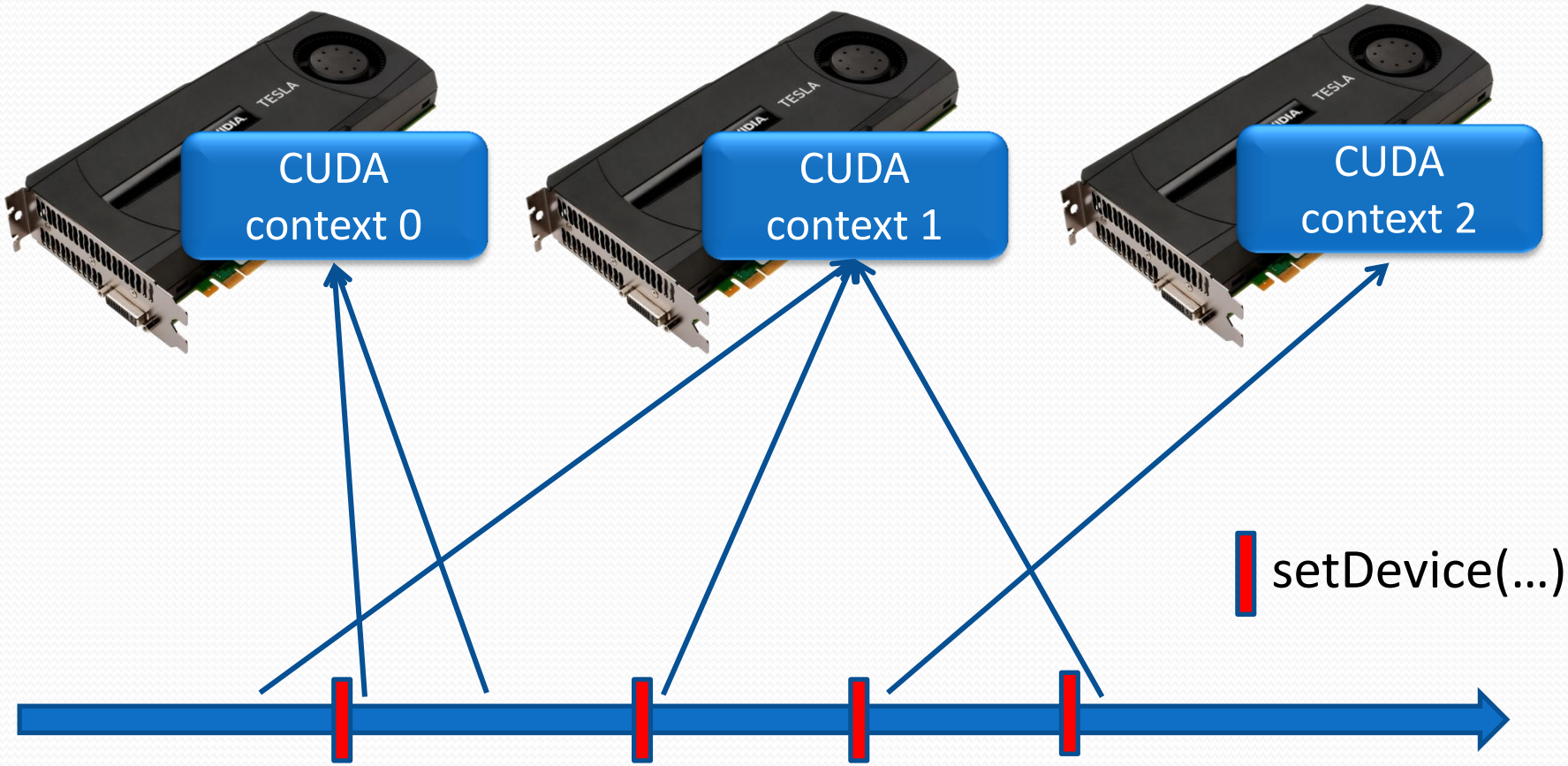
- `cudaEventElapsedTime()` вызовет ошибку, если события созданы в разных контекстах

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
```

Multi-GPU
С одной хост-нитью

Multi-GPU & single CPU thread



Поток CPU переключается между контекстами

Модельная задача

```
float *devPtr = NULL, *hostPtr = NULL;
int n;
loadInputData(&n, &hostPtr);
cudaHostRegister(hostPtr, n*sizeof(float),
            cudaHostRegisterDefault);
cudaMalloc(&devPtr, n * sizeof(float));
cudaMemcpyAsync(devPtr, hostPtr, n*sizeof(float),
           cudaMemcpyHostToDevice, 0);
kernel<<<(n – 1) / 512 + 1, 512>>>(devPtr, n);
cudaMemcpyAsync(hostPtr, devPtr, n*sizeof(float),
                     cudaMemcpyDeviceToHost, 0);
cudaDeviceSynchronize();
```

Переписываем на multiGPU

```
float *hostPtr = NULL;
int n, deviceCount;
loadInputData(&n, &hostPtr);
cudaGetDeviceCount(&deviceCount);
float **devPtr = (float **)malloc(deviceCount *
                                  sizeof(float *));
```

- Получили число устройств
- Выделили массив указателей на GPU-память

Выделение памяти

- Выделение памяти через `cudaMalloc*` происходит на устройстве, к которому относится активный контекст
  - При определенных условиях память может быть доступна из ядер, работающих на других устройствах (**peer-to-peer**)

- `cudaHostRegister[Alloc](…)` лочит(выделяет) память в рамках активного контекста
  - Преимущества доступны другим контекстам только если pinned-память является **portable:**

```
cudaHostRegister(ptr, n, cudaHostRegisterMapped |
                         cudaHostRegisterPortable);
```

Выделение памяти

```
int elemsPerDevice = (n – 1) / deviceCount + 1;
for(int device = 0; device < deviceCount; device++) {
    cudaSetDevice(device);
    cudaMalloc(devPtr + device, elemsPerDevice *
                                        sizeof(float));
    cudaHostRegister(hostPtr + device * elemsPerDevice,
                     elemsPerDevice * sizeof(float),…);
}
```

Выделение памяти блокирует хост-нить!

- Рассчитали размер подзадач
- Выделили / залочили нужные объемы в каждом контексте

Отправка команд

```
for(int device = 0; device < deviceCount; device++) {
    int offset = device * elemsPerDevice;
    int elemCount = min(n – offset, elemsPerDevice);
    cudaSetDevice(device);
    cudaMemcpyAsync(devPtr[device], hostPtr + offset),
                    elemCount * sizeof(float),…, 0);
    kernel<<<(elemCount – 1)/512 + 1, 512>>>
                          (devPtr[device], elemCount);
    cudaMemcpyAsync(hostPtr + offset, devPtr[device],
                    elemCount * sizeof(float),…, 0);
}
```

- Асинхронно отправляем команды на устройства
  - Кажое GPU работает со своей порцией данных

Синхронизация

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

- Ожидаем завершения всех команд на устройствах

Почему синхронизацию нужно делать в отдельном цикле?

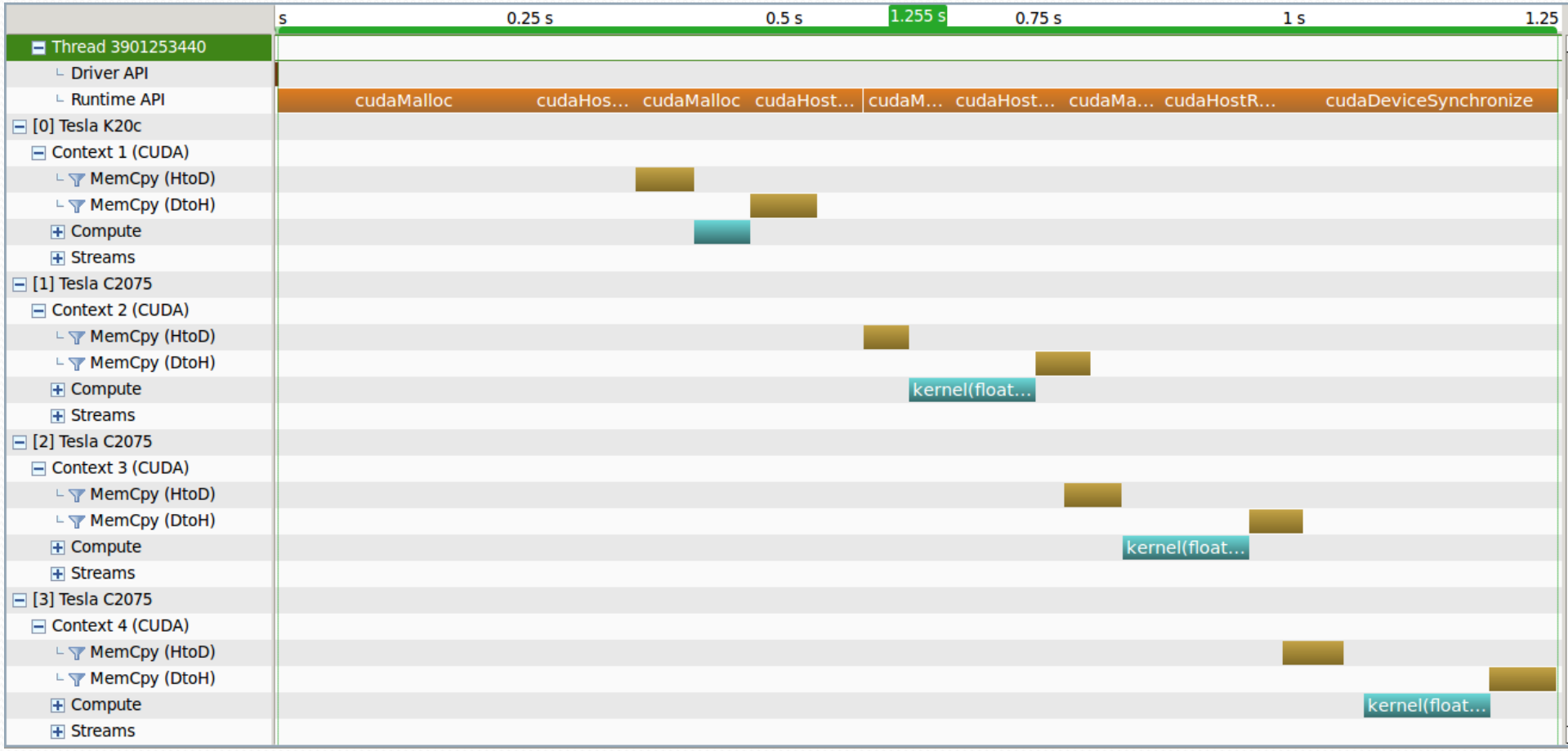
Результат (1.487 с)

Комментарий

- Неблокирующие запуски команд правильнее будет поместить в цикл с выделением памяти
  - Команды на первом GPU начнут выполняться пока на остальных выделяется память

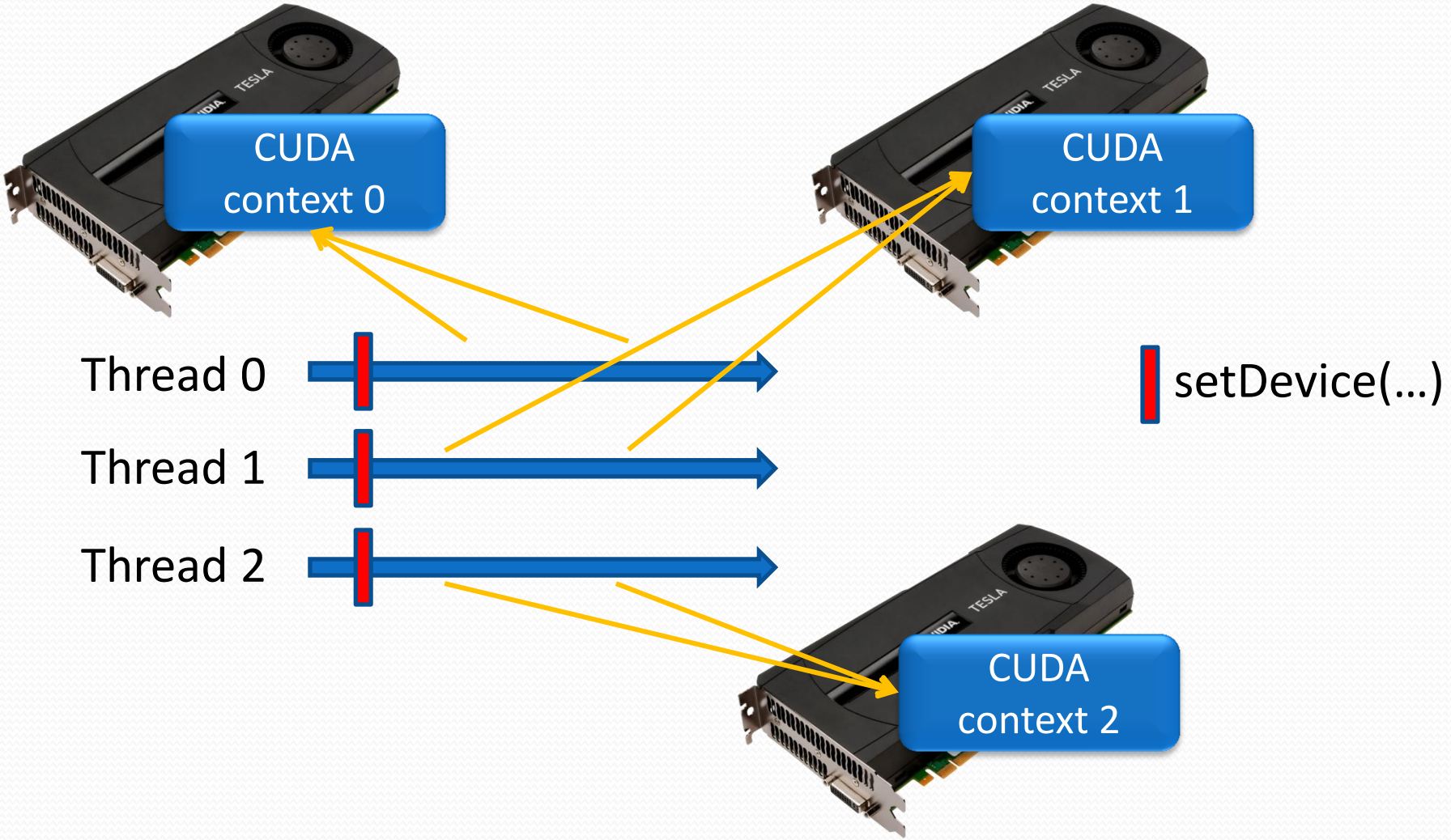
```
for(int device = 0; device < deviceCount; device++) {
    cudaSetDevice(device);
    cudaMalloc(…);
    cudaHostRegister(…);
    cudaMemcpyAsync(…); // pinned CPU <-> GPU
    kernel<<<…>>>(…);
    cudaMemcpyAsync(); // pinned CPU  <-> GPU
}
```

Результат (1.255 с)



Multi-GPU + OpenMP

Multi-GPU & multiple CPU threads



Компиляция

- Поддержка OpenMP встроена в популярные компиляторы
  - Intel icc/ifort, gcc/gfortran, MS cl, IBM xlc

- Обычный компилятор компилирует OpenMP директивы и функции при указании специального флага компиляции (для распознавания директив) и линковки (для линковки omp-функций)
  - icc –openmp
  - gcc –fopenmp
  - cl -/openmp
  - xlc -qsmp

Компиляция с NVCC

```
$nvcc –Xcompiler flag –arch=sm_20 main.cu
```

- Передает компилятору на стадию компиляции и на стадию линковки слово командной строки **flag**, следующее за –Xcompiler

```
$nvcc –Xcompiler –fopenmp –arch=sm_20 main.cu
```

- Компиляция CUDA+OpenMP на Linux (gcc)

Компиляция с NVCC

- Можно раздельно:

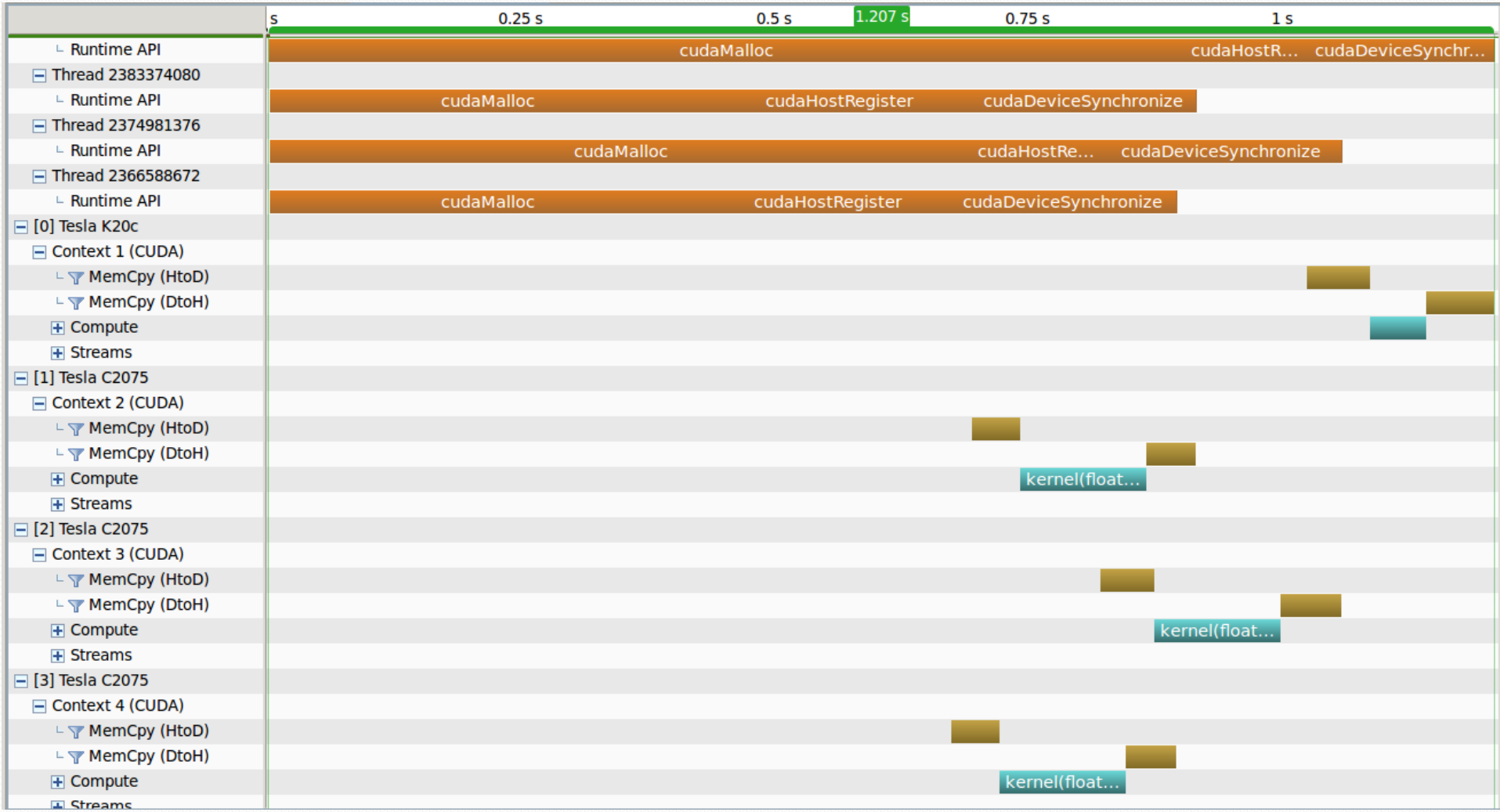
```
$nvcc –arch=sm_20 kernel.cu
$gcc -fopenmp –I/opt/cuda/include main.c
$gcc -fopenmp –L/opt/cuda/lib –lcudart
  main.o kernel.o
```

Переписываем под OpenMP

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
#pragma omp parallel num_threads(deviceCount)
{
    int device = omp_get_thread_num();
    cudaSetDevice(device);
    cudaMalloc(devPtr + device, elemsPerDevice *
                                sizeof(float));
    cudaHostRegister(…);
    … // прочие команды
    cudaDeviceSynchronize();
}
```

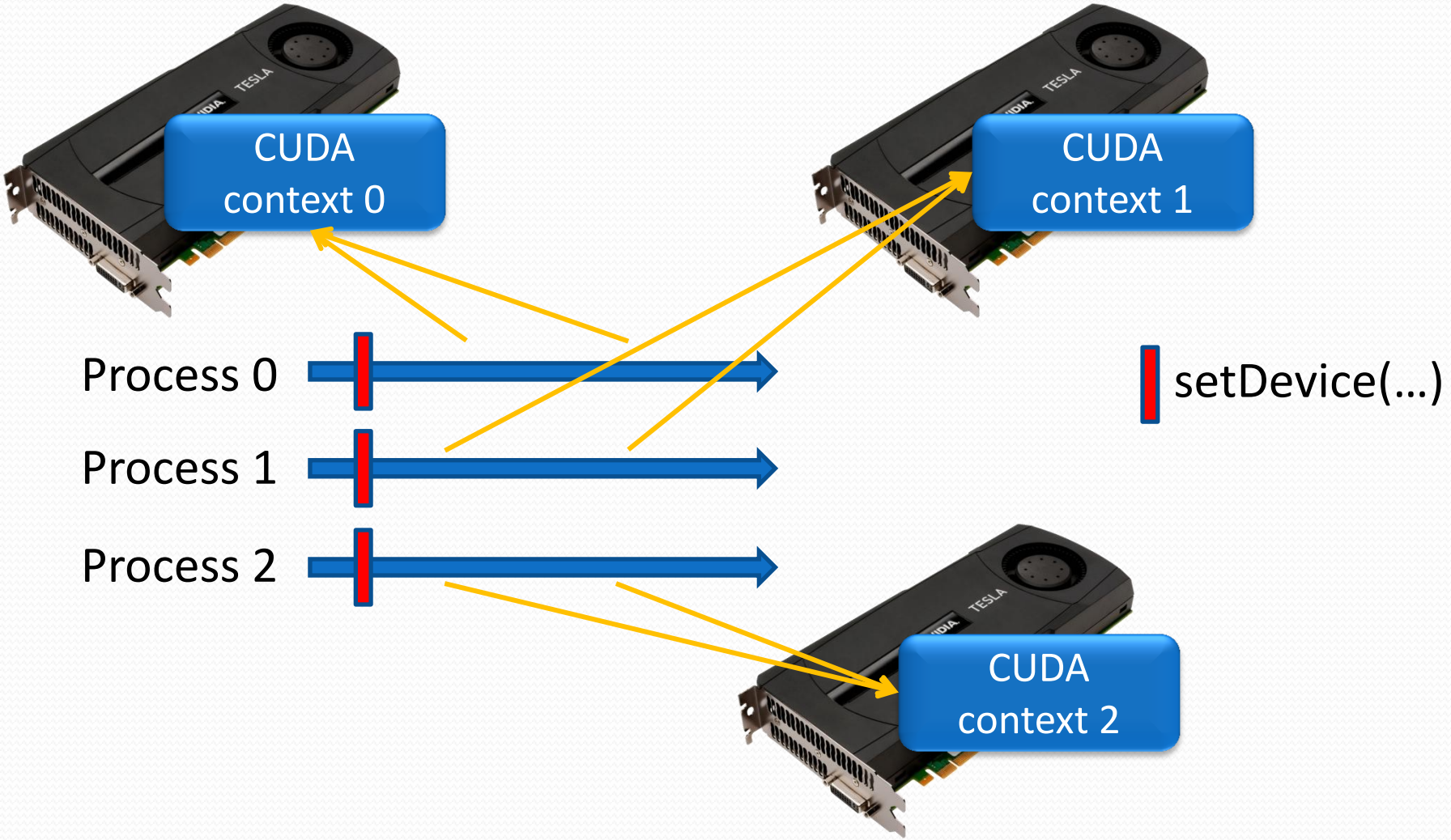
Запускаем параллельную секцию на нужном числе нитей

Результат (1.207 с)



Multi-GPU + MPI

Multi-GPU & multiple CPU processes



Компиляция

- **mpicc** – обертка над хостовым компилятором
  - Задача **mpicc** – подставить пути к инклюдам и слинковать объектные файлы с MPI-библиотеками

Больше ничего mpicc не делает!

- Наша цель:
  - скомпилировать MPI хост код с теми же флагами, с которыми это делает `mpicc`
  - Скомпилировать device-код при помощи `nvcc`
  - Слинковать все с нужными библиотеками MPI/CUDA

Как узнать флаги mpicc?

- При использовании OpenMPI:
  - Вывести флаги компиляции: `$mpicc -showme:compile`

    ```
    -I/usr/lib/openmpi/include -
    I/usr/lib/openmpi/include/openmpi –pthread
    ```

  - Вывести флаги линковки: `$mpicc -showme:link`

    ```
    -pthread -L/usr/lib/openmpi/lib -lmpi -lopen-rte -
    lopen-pal -ldl -Wl,--export-dynamic -lnsl -lutil -lm -
    ldl
    ```

Как узнать флаги mpicc?

- При использовании OpenMPI:
  - Вывести полную строку, вместе с именем используемого компилятора: `$mpicc –showme`

```
gcc -I/usr/lib/openmpi/include -
I/usr/lib/openmpi/include/openmpi -pthread -
L/usr/lib/openmpi/lib -lmpi -lopen-rte -lopen-pal
-ldl -Wl,--export-dynamic -lnsl -lutil -lm -ldl
```

Замена gcc в mpicc на nvcc

- При использовании OpenMPI компилятор можно задать через переменную окружения OMPI_CC, OMPI_F77, OMPI_CXX, OMPI_FC

  - **$OMP_CC=nvcc mpicc --showme**

    ```
    nvcc -I/usr/lib/openmpi/include -
    I/usr/lib/openmpi/include/openmpi -pthread -
    L/usr/lib/openmpi/lib -lmpi -lopen-rte -lopen-pal
    -ldl -Wl,--export-dynamic -lnsl -lutil -lm –ldl
    ```

Замена gcc в mpicc на nvcc

- Проблема:
  - nvcc сам парсит флаги

    Поддерживает только простые `-L, -l, -c,-g` и свои собственные `-v, -arch` и ничего не знает о `-Wl, -pthread`

- Специфические флаги компилятора нужно передавать через `-Xcompiler <флаг>,…`
- Специфические флаги линковщика ld нужно передавать через `-Xlinker <флаг>,…`

Раздельная компиляция

- Подставляем флаги компиляции `mpicxx` в `nvcc`
- Линкуем хостовым компилятором, явно подставляя флаги из `mpicxx` и `nvcc`

```
MPI_COMPILE_FLAGS = $(shell mpicxx --showme:compile)
MPI_LINK_FLAGS = $(shell mpicxx --showme:link)
NVCC_LINK_FLAGS = -L/opt/cuda/lib64 -lcudart
all: main
    nvcc -Xcompiler "\"$(MPI_COMPILE_FLAGS)\"" main.cu –o main.o
    g++ main.o -o main $(MPI_LINK_FLAGS) $(NVCC_LINK_FLAGS)
```

Замена gcc в nvcc на mpicxx

- Опция `-ccbin` позволяет задать используемый компилятор

```
$nvcc -ccbin /usr/bin/mpicxx main.cu -o main
```

- `cudafe` разделит код на host-код и device-код
  - Host-код будет скомпилирован в дальнейшем при помощи `/usr/bin/mpicxx`

- Объектники будут слинкованы через `/usr/bin/mpicxx` => все нужные MPI флаги подставятся

Пример main.cu

```
#include <mpi.h>
#include <iostream>
#include <stdio.h>

__global__ void kernel(int procnum, int device ) {
   printf("Hello from DEVICE %d process %d\n",device,
procnum);
}

int main (int argc, char* argv[])
{
    int rank, size;
    int numDevices = -1;
    cudaGetDeviceCount(&numDevices);
    …
```

Пример main.cu (продолжение)

```
    …
    cudaGetDeviceCount(&numDevices);
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    std::cout<<"Hello from HOST #"<< rank << " see "<<
numDevices << " devices" << std::endl;
    cudaSetDevice(rank % numDevices);
    kernel<<<1,1>>>(rank % numDevices, rank);
    cudaDeviceSynchronize();
    MPI_Finalize();
    return 0;
}
```

Компиляция & запуск

```
$nvcc –arch=sm_20 –ccbin mpicxx main.cu
$mpirun –n 6 ./a.out
```

```
Hello from HOST #3 see 4 devices
Hello from HOST #4 see 4 devices
Hello from HOST #2 see 4 devices
Hello from HOST #5 see 4 devices
Hello from HOST #1 see 4 devices
Hello from HOST #0 see 4 devices
Hello from DEVICE 1 process 1
Hello from DEVICE 3 process 3
Hello from DEVICE 5 process 1
Hello from DEVICE 2 process 2
Hello from DEVICE 4 process 0
Hello from DEVICE 0 process 0
```

peer-to-peer обмены между GPU

UVA & peer-to-peer

- При UVA 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 UVA не поддерживается или нужно явно указать, что это peer-to-peer копирование, используются функции `cudaMemcpyPeer*`

peer-to-peer & non-UVA

- 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)
```

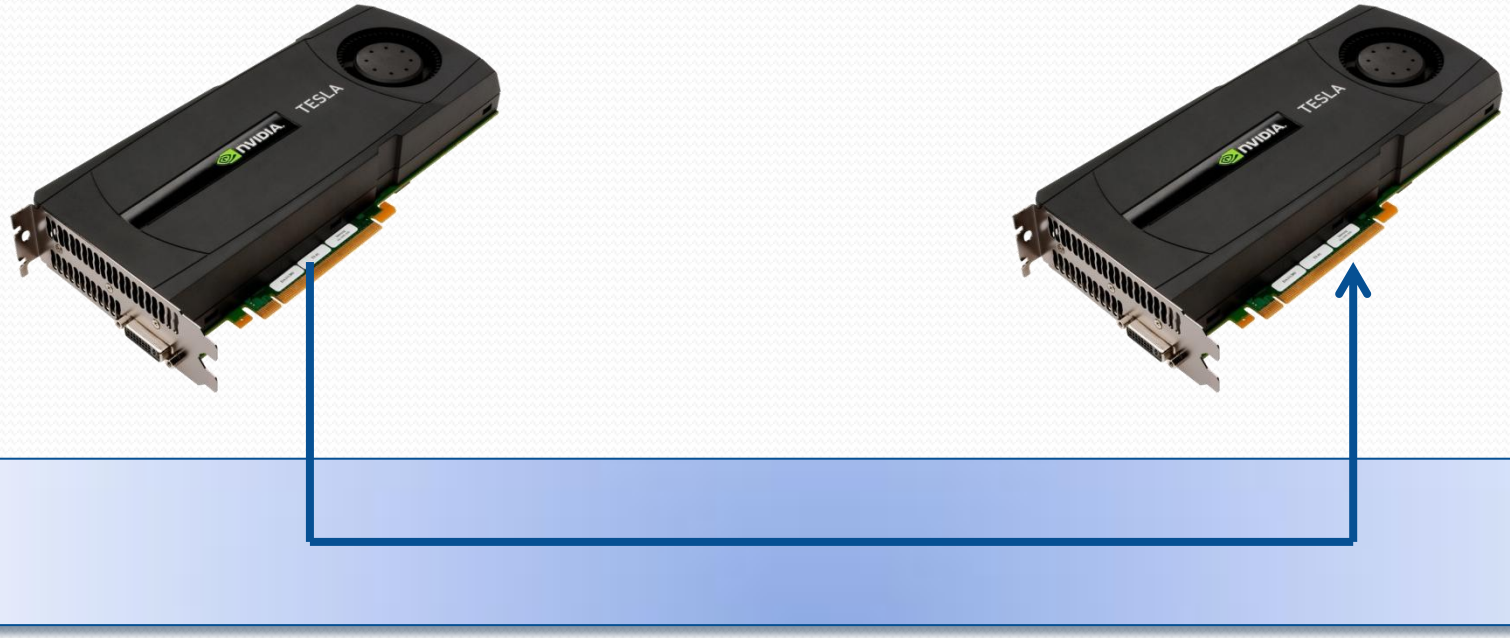
Peer vs PeerAsync

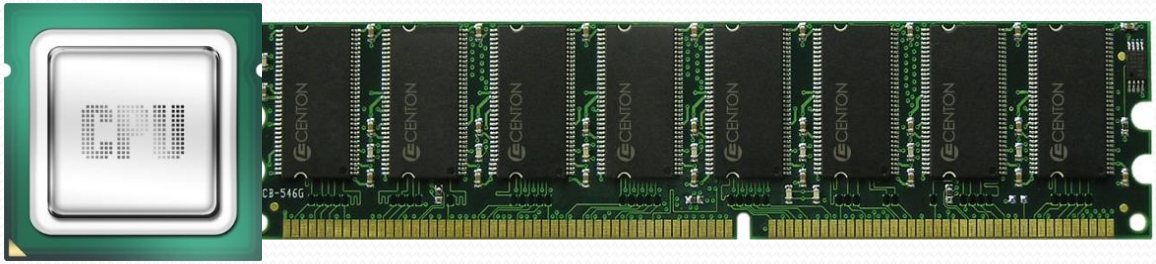
- Обе функции не блокируют хост
- cudaMemcpyPeer начнется только когда завершатся все команды на обоих устройствах (и на активном), отправленные до неё
- Параллельно с cudaMemcpyPeer не могут выполнятся другие команды на обоих устройствах (и на активном)
- cudaMemcpyPeerAsync лишена этих ограничений

P2P пример

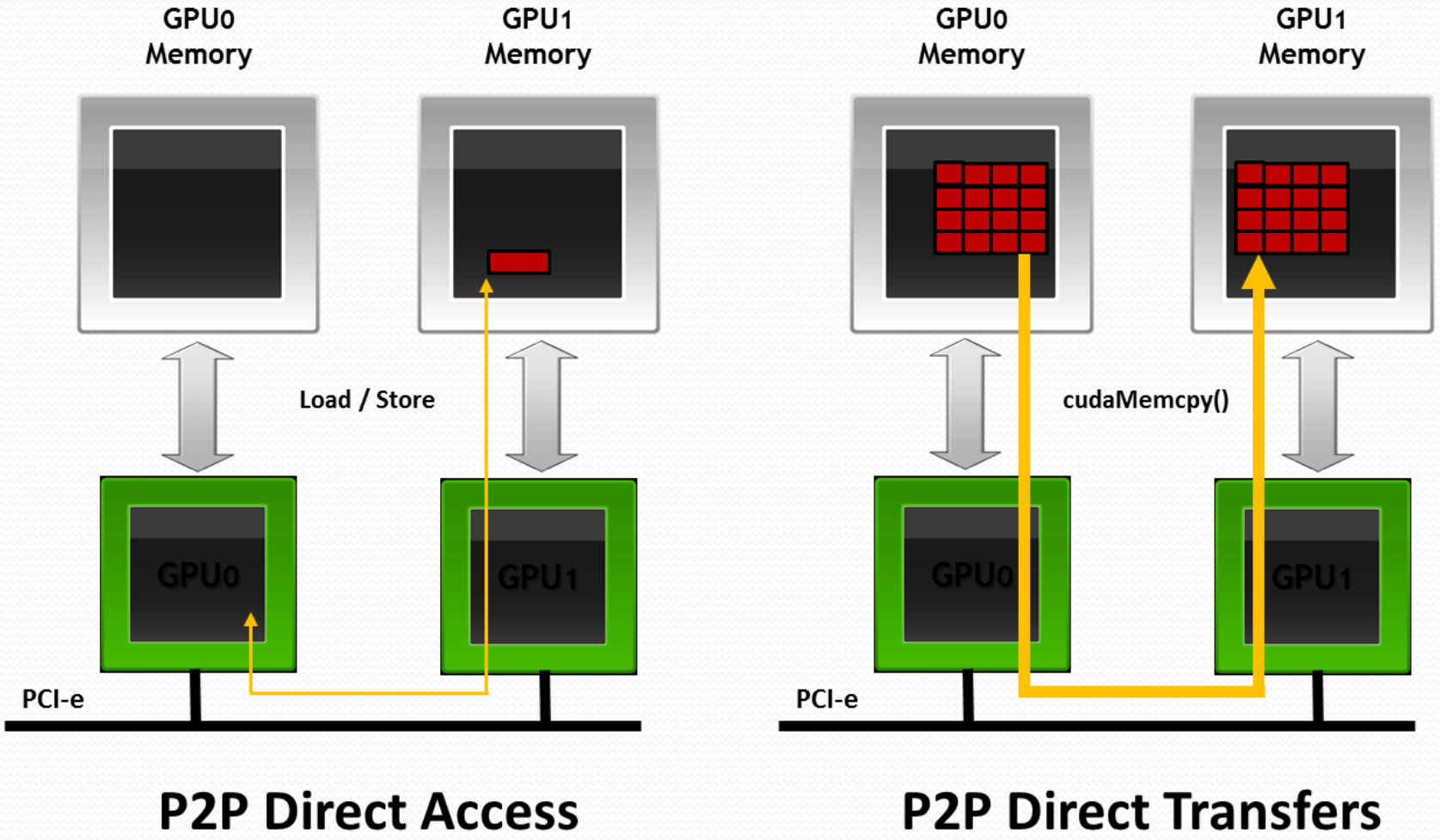
```
cudaSetDevice(0); // Переключились на device 0
float* p0, *p1;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // Выделили на device 0
cudaSetDevice(1); // Переключились на device 1
cudaMalloc(&p1, size); // Выделили на device 1
cudaSetDevice(0); // Переключились на device 0
MyKernel<<<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
```

Прямые peer-to-peer обмены





Прямые peer-to-peer обмены



Прямой P2P доступ

```
cudaError_t cudaDeviceCanAccessPeer (
      int* canAccessPeer, int  device,
      int  peerDevice )
```

- Если в **canAccessPeer** записалось **«1»**, то
  - peer-to-peer копирования межу **peerDevice** и **device** могут выполняться без буферизации на хосте
  - Память выделенная на **peerDevice** может быть доступна напрямую из ядер, работающих на **device**

Прямой P2P доступ

```
cudaError_t cudaDeviceCanAccessPeer (
      int* canAccessPeer, int  device,
      int  peerDevice )
```

- Tesla series
- UVA
- Compute Capability > 2.0

Прямой доступ нужно явно включить!

```
cudaError_t cudaDeviceEnablePeerAccess (
      int  peerDevice, unsigned int  flags )
```

- Теперь память, выделяемая на **peerDevice** доступна напрямую из ядер, запускаемых на активном, а копирования выполняются без участия памяти хоста
- Вызов включает доступ только в одну сторону. Для доступа в обратную сторону нужен отдельный вызов `cudaDeviceEnablePeerAccess`

Прямой P2P доступ

```
cudaSetDevice(0); // Переключились на device 0
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // Выделить память на device 0
cudaSetDevice(1); // Переключились на device 1
cudaDeviceEnablePeerAccess(0, 0); // Включить peer-to-peer
                                     доступ к 0

// Запуск на device 1
// p0 указывает на память, выделенную на device 0
MyKernel<<<1000, 128>>>(p0);
```

P2P на tesla-cmc

```
for (int device = 0..deviceCount) {
    for (int peerDevice = 0..deviceCount){
        if (device == peerDevice) {
            printf("- "); continue;
        }
        int canAccessPeer = -1;
        cudaDeviceCanAccessPeer(&canAccessPeer,
                                device, peerDevice);
        printf("%1d ", canAccessPeer);
    }
    printf("\n");
}
```

```
   0 1 2 3
0 - 0 0 0
1 0 - 1 0
2 0 1 - 0
3 0 0 0 -
```

Тест прямого P2P копирония

```
printf("\nCopying %d MB\n", sizeOfMem / (1024 * 1024));
cudaSetDevice(1);
cudaMalloc(&memoryOnDevice1, sizeOfMem);
cudaSetDevice(2);
cudaMalloc(&memoryOnDevice2, sizeOfMem);

...;/* замерить время cudaMemcpyPeer или cudaMemcpy */

printf("\nElapsed time %f \n", elapsedTime);
cudaDeviceEnablePeerAccess(1, 0);
printf("Enable peer access\n");

...;/* замерить время cudaMemcpyPeer или cudaMemcpy */

printf("Elapsed time %f\n", elapsedTime);
```

Тест прямого P2P копирония

```
$./a.out 402410240
Copying 383 MB
Elapsed time 475.087402
Enable peer access
Elapsed time 75.842880
```

Выводы

- Поддержка P2P-копирований упрощает хост-код при необходимости организации пересылок между устройствами
- Поддержка прямых P2P-копирований дает ускорение копирований в несколько раз
- Поддержка прямого P2P-доступа избавляет от необходимости организации пересылок между устройствами