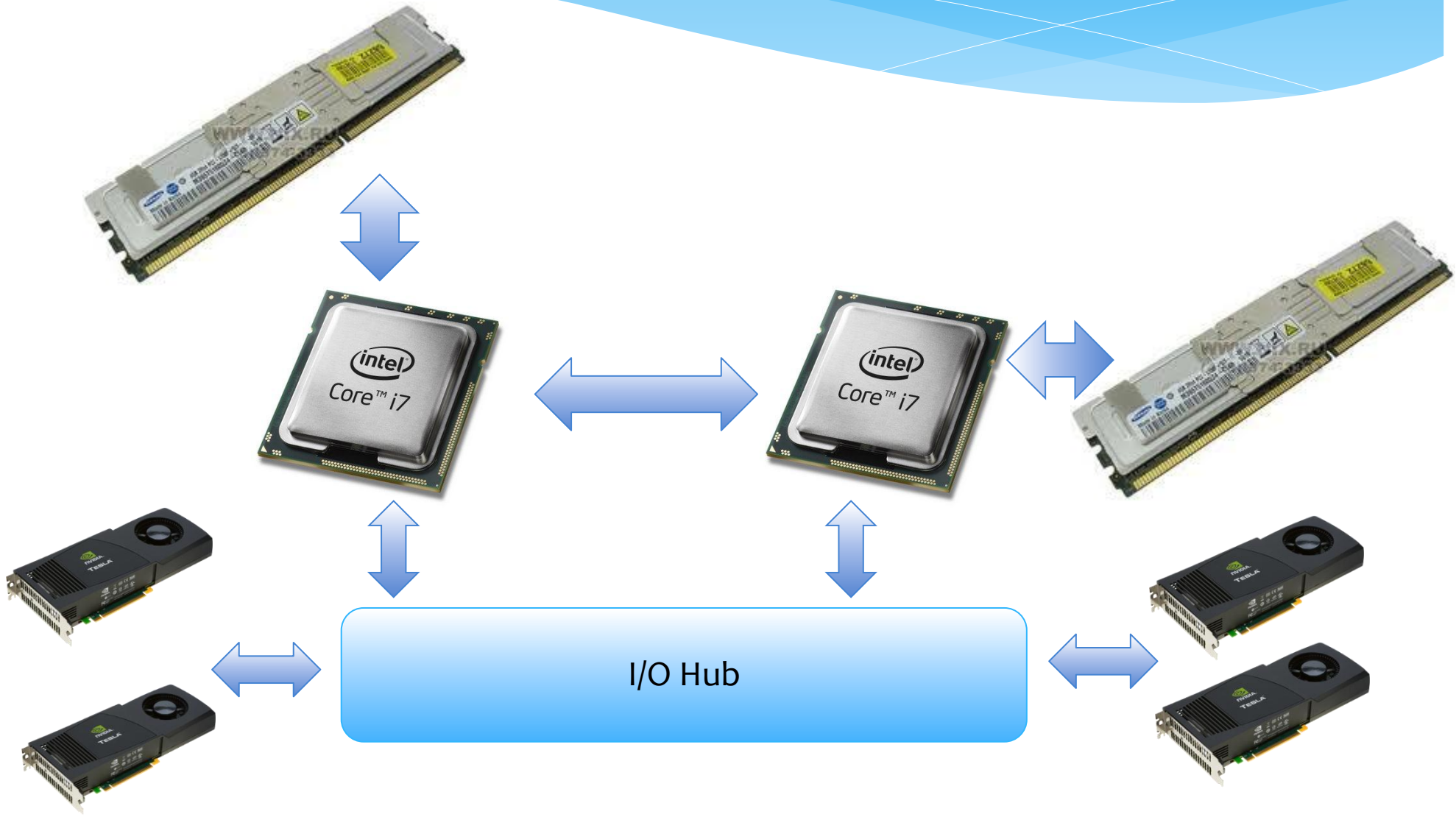
Программирование для систем с несколькими GPU

Романенко А.А.
arom@ccfit.nsu.ru
Новосибирский государственный университет

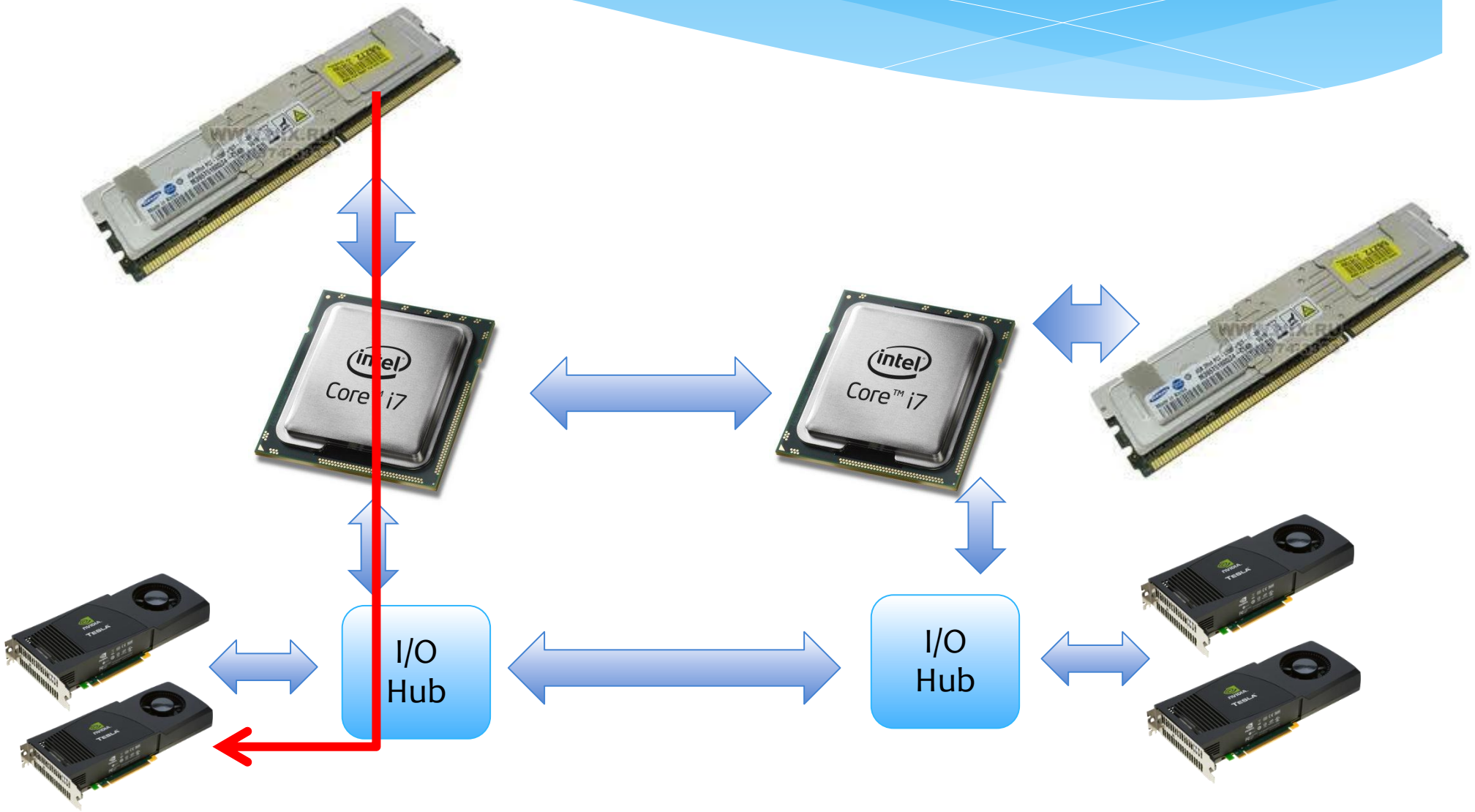
Гибридные системы



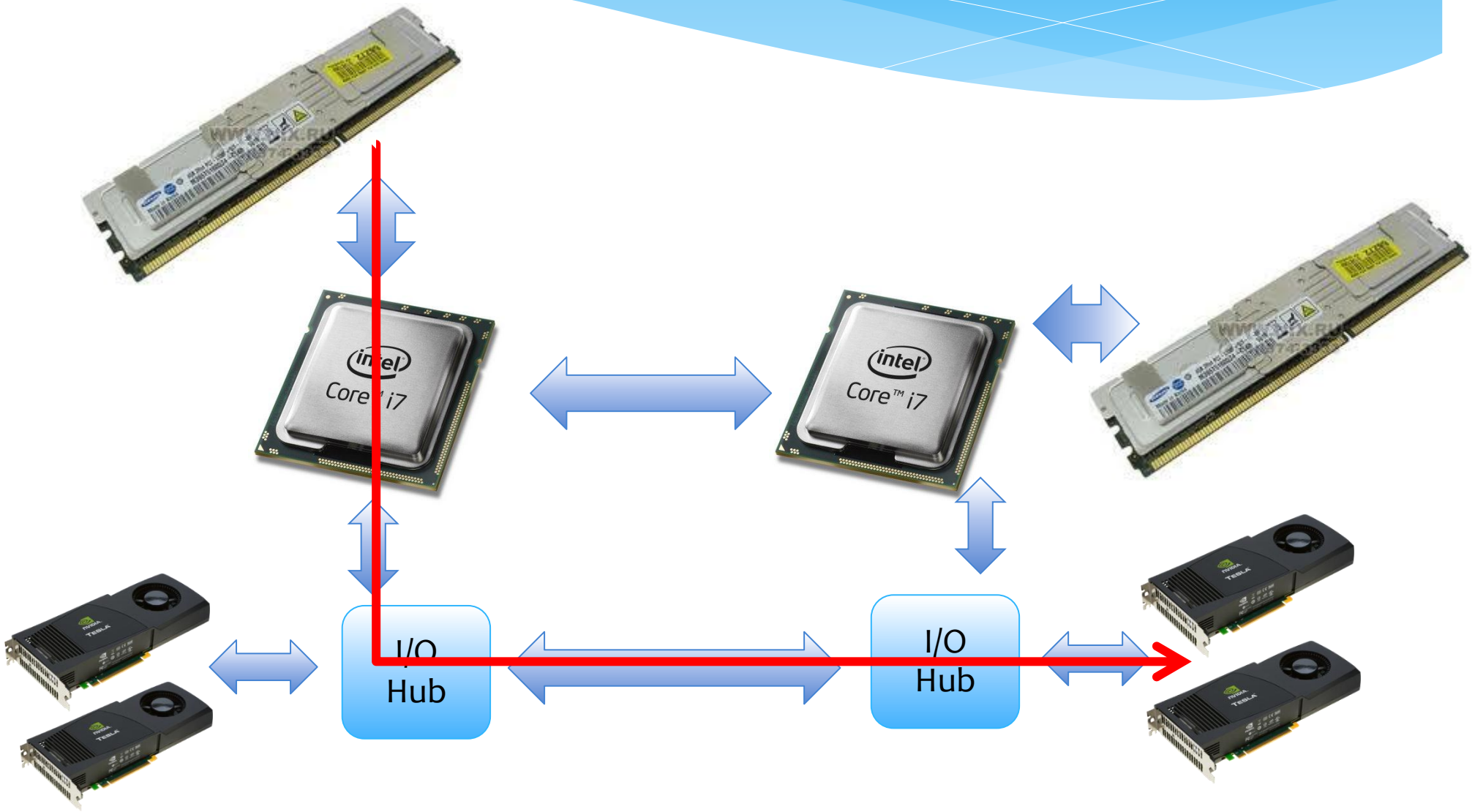
Замечания по NUMA

* Хост (CPU) NUMA (Non Uniform Memory Access) влияет на скорость передачи 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

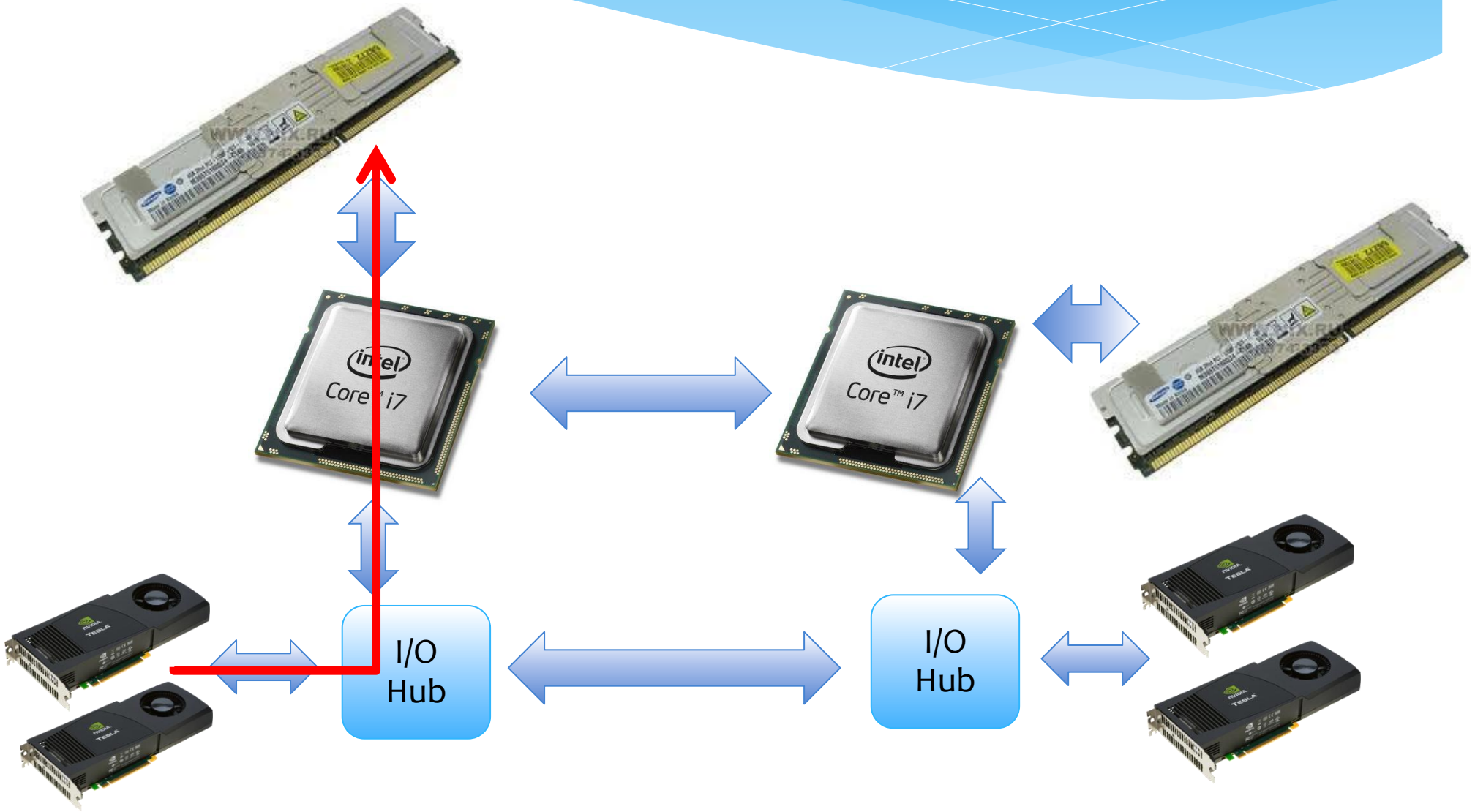
Local H2D Copy: 5.7 GB/s



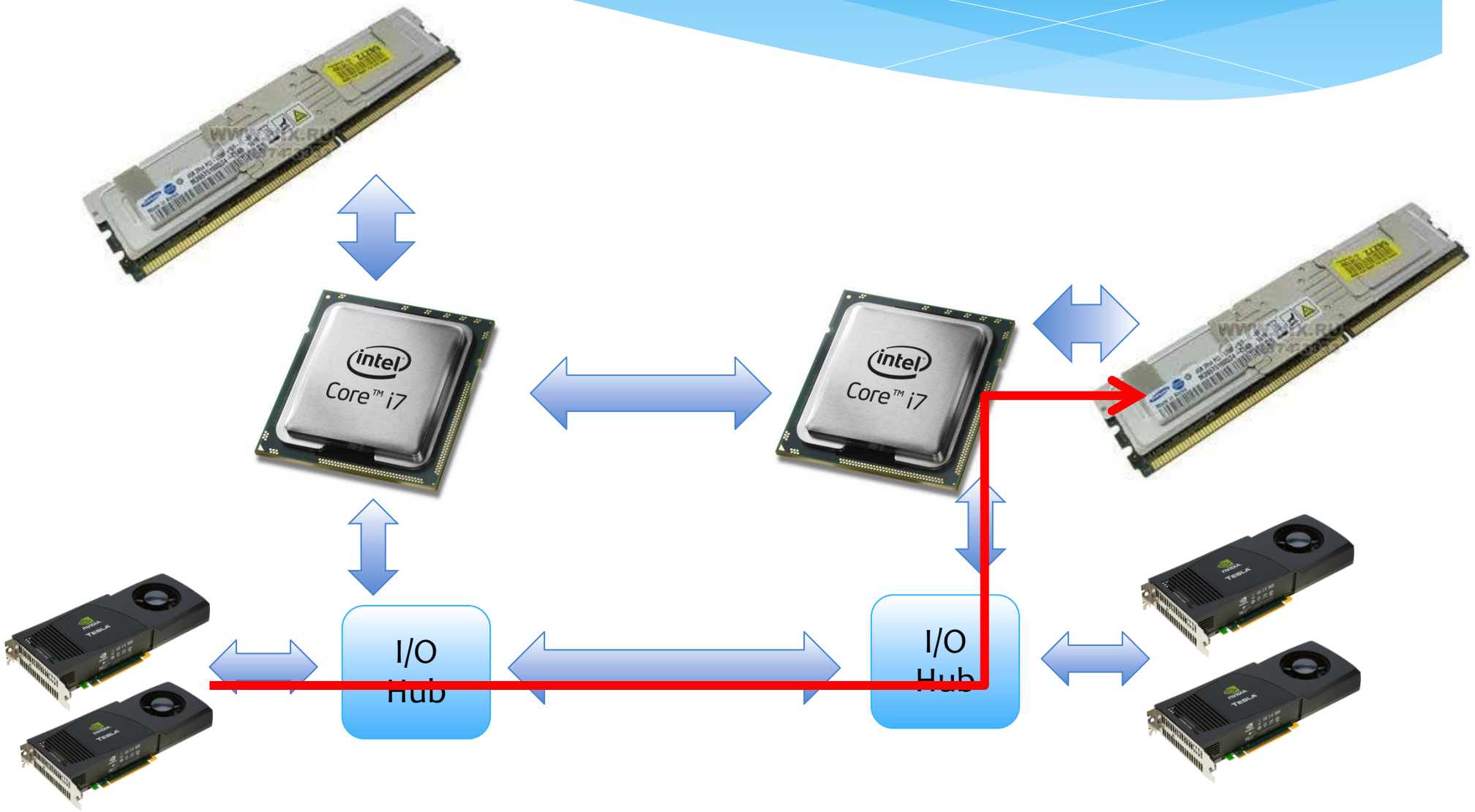
Remote H2D Copy: 4.9 GB/s



Local D2H Copy: 6.3 GB/s



Remote D2H Copy: 4.9 GB/s



Копирование данных между GPU

CUDA 3.2

cudaMemcpy(Host, GPU1);

cudaMemcpy(GPU2, Host);

CUDA 4.0

cudaMemcpy(GPU1, GPU2);

Можно как читать так и писать

Поддерживается только на Tesla 20xx (Fermi)

64-битные приложения

Unified Virtual Addressing CUDA 4.0

* Память центрального процессора и всех GPU объединена в единое виртуальное адресное пространство.
* Один параметр (cudaMemcpyDefault) вместо 4-х (cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice)
* Поддерживается только на Tesla 20xx (Fermi)
* 64-битные приложения

Количество GPUs

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, device);
        printf("Device %d has compute capability %d.%d.\n",
                device, deviceProp.major, deviceProp.minor);
}
```

Контекст устройства

* **Контекст** – привязанная к определённому устройству управляющая информация (выделенная device-память, результат операции, ...)
* При обращении к устройству многие CUDA-вызовы требуют существование контекста
* Изначально поток/процесс не имеет текущего CUDA-контекста
* Если в процессе/потоке нет текущего контекста, то он будет создан неявно при необходимости (runtime API)
* Одно устройство может иметь несколько контекстов (driver API)

Управление контекстом

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

GPU и поток исполнения CUDA 3.2

* Поток ассоциирован с одним GPU *
* Выбор GPU или явно (cudaSetDevice()) или неявно - по-умолчанию.
* По умолчанию выбирается GPU с номером «0»
* Если в потоке выполнена какая-либо операций над GPU, то попытка сменить GPU на другой приведет к ошибке.

* - на уровне драйвера это не так.

GPU и поток исполнения CUDA 4.0

* Любой поток имеет доступ ко всем GPU
* Выбор активного устройства через вызов cudaSetDevice()
* Возможность запуска параллельных ядер из разных потоков.

Многопоточное программирование

* OpenMP
* POSIX Threads
* WinThreads
* MPI
* IPC
* пр.

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
* Параллельные секции
  #pragma omp sections

OpenMP

```
#pragma omp parallel sections
{
#pragma omp section
  {
   cudaSetDevice(0);
   ...
  }
#pragma omp section
  {
   cudaSetDevice(1);
   ...
  }
}
```

OpenMP

```
#pragma omp parallel sections
{
#pragma omp section
  { // section for GPUs
    ...
  }
#pragma omp section
  { // section for CPUs
    ...
  }
}
```

OpenMP

```
int nElem = 1024;
cudaGetDeviceCount(&nGPUs);
if(nGPUs >= 1){
     omp_set_num_threads(nGPUs);
#pragma omp parallel
     {
          unsigned int cpu_thread_id = omp_get_thread_num();
          unsigned int num_cpu_threads = omp_get_num_threads();
          cudaSetDevice(cpu_thread_id % nGPUSs); //set device

          dim3 BS(128);
          dim3 GS(nElem / (gpu_threads.x * num_cpu_threads));
               // memory allocation and initialization
          int startIdx = cpu_thread_id * nElem / num_cpu_threads;
          int threadNum = nElem / num_cpu_threads;
          kernelAddConstant<<<GS, BS>>>(pData, startIdx, threadNum);
          // memory copying
     }
```

```
// Section for GPUs.
#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);
  }
}
```

OpenMP. Сборка программ

* gcc 4.3
* Command line
  * $ nvcc -Xcompiler \
    -fopenmp -Xlinker\
    -lgomp cudaOpenMP.cu
* Makefile
  * EXECUTABLE := cudaOpenMP
    CUFILES := cudaOpenMP.cu
    CUDACCFLAGS := -Xcompiler -fopenmp
    LIB := -Xlinker -lgomp
    include ../../common/common.mk

POSIX threads (pthreads)

* Реализация – библиотека
* Пользователь явно управляет созданием, завершением потоков и их свойствами
* Пользователь явно управляет взаимодействием потоков
* Документация
  https://computing.llnl.gov/tutorials/pthreads/
  man pthreads

POSIX threads (pthreads)

* 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

CUDA Utillity Library

```
static CUT_THREADPROC solverThread(SomeType *plan){
    // Init GPU
    cutilSafeCall( cudaSetDevice(plan->device) );
    // start kernel
    SomeKernel<<<GS, BS>>>(some parameters);
    cudaThreadSynchronize();

    cudaThreadExit();
    CUT_THREADEND;
}
```

Макросы используются для переносимости программы с Unix на Windows и обратно.

CUDA Utillity Library

```
SomeType solverOpt[MAX_GPU_COUNT];
CUTThread threadID[MAX_GPU_COUNT];

for(i = 0; i < GPU_N; i++){
   solverOpt[i].device = i; …
}
//Start CPU thread for each GPU
for(gpuIndex = 0; gpuIndex < GPU_N; gpuIndex++){
    threadID[gpuIndex] =

cutStartThread((CUT_THREADROUTINE)solverThread,
                               &SolverOpt[gpuIndex]);
}

//waiting for GPU results
cutWaitForThreads(threadID, GPU_N);
```

MPI – Message Passing Interface

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

* 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 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);
```

IPC – Inter-process communication

* Реализация – библиотеки
* Пользователь явно управляет созданием, завершением процессов и их свойствами
  * **fork(), exit(), …**
* Пользователь явно управляет взаимодействием процессов
* Разделяемая память, сигналы, условные переменные, семафоры …
* Документация
  man ipc

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();
```

Работа с драйвером

* Для каждого устройства явно создается контекст (**cuCtxCreate**)
* Перед выполнением операций с устройством соответствующий контекст делается текущим (**cuCtxPushCurrent**), а после операции – снимается (**cuCtxPopCurrent**)
* В конце контексты удаляются (**cuCtxDestroy**)

* (!!) Если контекст создан до вызова fork(), то после него работа с контекстом может быть некорректна

Создание контекстов

```
for(int i=0; i<nGPUS; i++){
        CUdevice dev;
        CUresult cu_status = cuDeviceGet(&dev, i);
        if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }

        device_t *device = &devices[i];
        cu_status = cuCtxCreate(divice->ctx, 0, dev);
        if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }

        CUresult cu_status = cuCtxPopCurrent(divice->ctx);
        if (cu_status != CUDA_SUCCESS) { /* обработка ошибки */ }
}
```

Работа с контекстами

```
for(int i=0; i<nGPUS; i++){
        device_t *device = &devices[i];
        // сделать контекст активным для текущего потока\процесса
        CUresult cu_status = cuCtxPushCurrent(device->ctx);
        if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }

        // инициализация памяти, запуск ядра …

        // отключить контекст от потока\процесса
        cu_status = cuCtxPopCurrent(device->ctx);
        if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }
}
```

Завершение контекста

```
for(int i=0; i<nGPUS; i++){
        device_t *device = &devices[i];
        // сделать контекст активным для текущего потока\процесса
        CUresult cu_status = cuCtxPushCurrent(device->ctx);
        if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }
        // дождаться завершения       ядра
        cuda_status = cudaThreadSynchronize();

        // сохранение результата, освобождение памяти


        // удалить контекст
        cu_status = cuCtxDestroy (device->ctx);
        if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }
}
```

Заключение

* Работа с несколькими GPU возможна
* В CUDA 3.2 и более ранних версиях необходимо писать многопоточные программы
  * Можно из одного потока через функции драйвера.
* В CUDA 4.0 можно к нескольким GPU обращаться из одного потока
* При наличие UVA можно избежать копирование между устройствами через память Хоста