Applied Parallel Computing LLC
http://parallel-computing.pro

Введение в вычисления на GPU и обзор архитектуры

к.т.н. Алексей Ивахненко

10 декабря, 2017

GPU повсюду!

Courtesy of NVIDIA

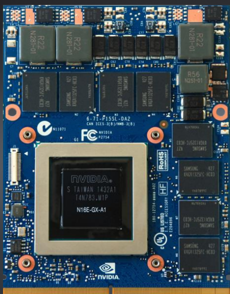
GPU повсюду!

Jetson TX1 SoC

- Портативная техника
- Дополненная реальность
- Автономные роботы, беспилотные аппараты
- ...

Courtesy of NVIDIA

GPU повсюду!



GTX 980M

- Ноутбук с GPU
- CAD/CAE с GPU
- Разработка GPU приложений
- ...

Courtesy of NVIDIA

GPU повсюду!

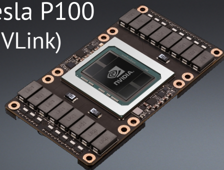
Tesla P100
(PCI-E)

- Рабочая станция с GPU
- Несколько GPU в одном ПК – MultiGPU
- GPU-ускоренное Глубокое обучение, CAD/CAE, CFD, сейсмология, нефтегазовая отрасль
- ...

Courtesy of NVIDIA

GPU повсюду!

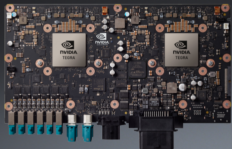
Tesla P100
(NVLink)



- GPU сервер/кластер
- Несколько GPUs в одном сервере – MultiGPU
- NVLink: 8× Tesla P100s с пропускной способностью в 5 раз больше PCIe
- GPU-ускоренное Глубокое обучение, CAD/CAE, CFD, сейсмология, нефтегазовая отрасль
- ...

Courtesy of NVIDIA

GPU повсюду!



NVIDIA Tegra Parker

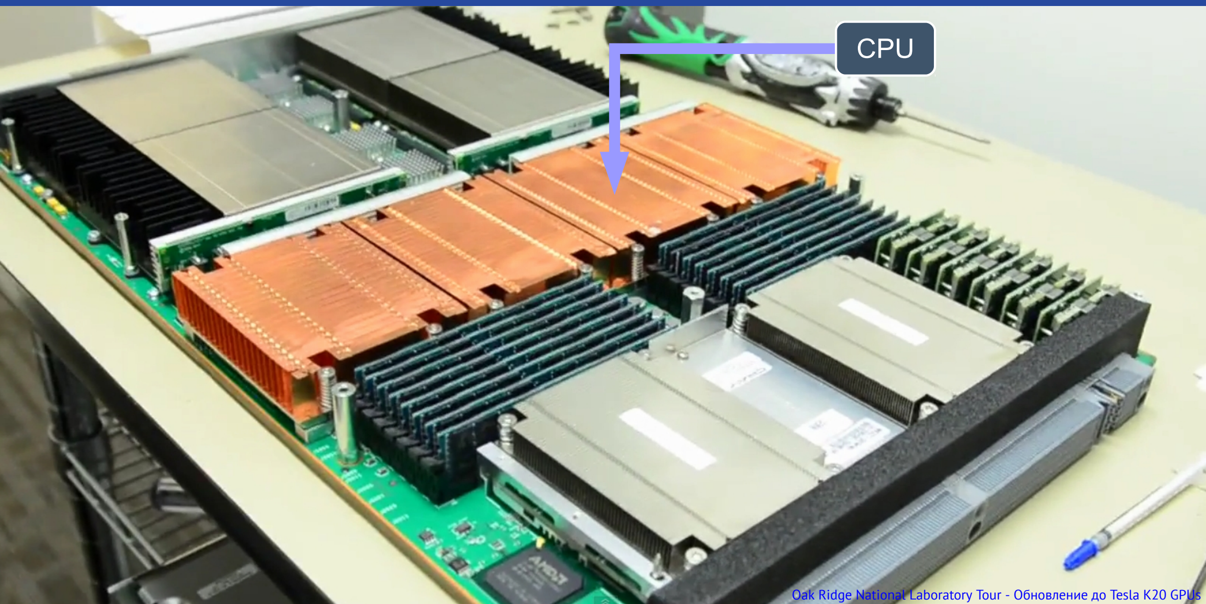
- NVIDIA Drive PX2 платформа для беспилотных автомобилей
- Содержит 2× NVIDIA Tegra Parker SOC
- 12× CPU ядер, 4× GPU архитектуры Pascal
- 8 ТФЛОПс FP32 и 24 ТФЛОПс INT8
- Виртуализация аппаратных средств
- ...

Courtesy of NVIDIA

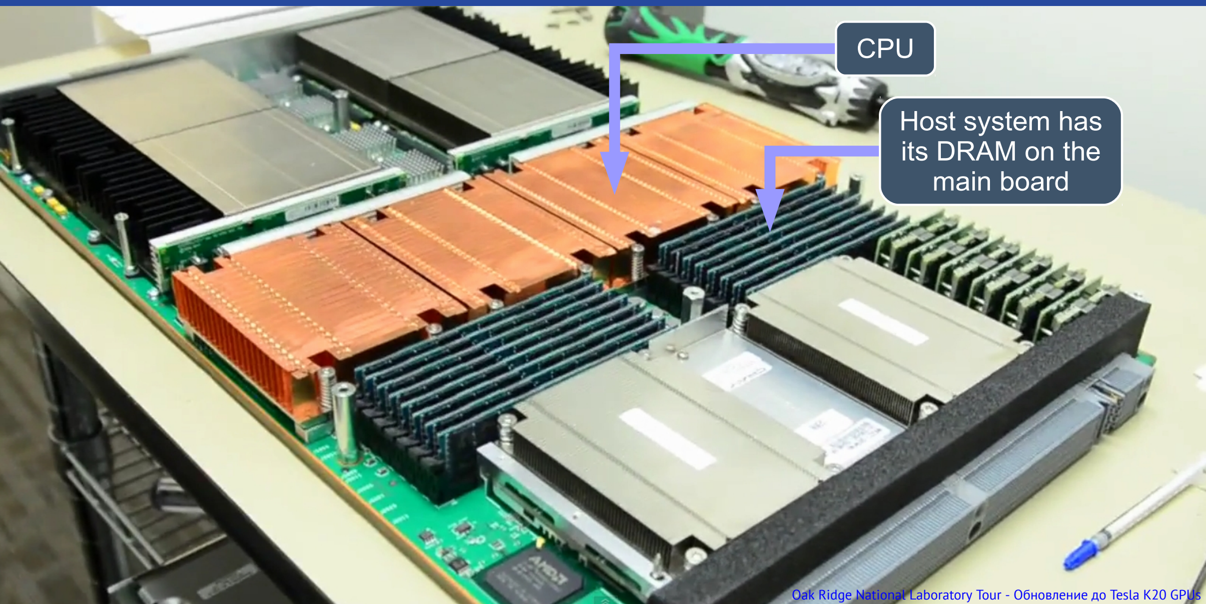
GPU в системах высокопроизводительных вычислений

Oak Ridge National Laboratory Tour - Обновление до Tesla K20 GPUs

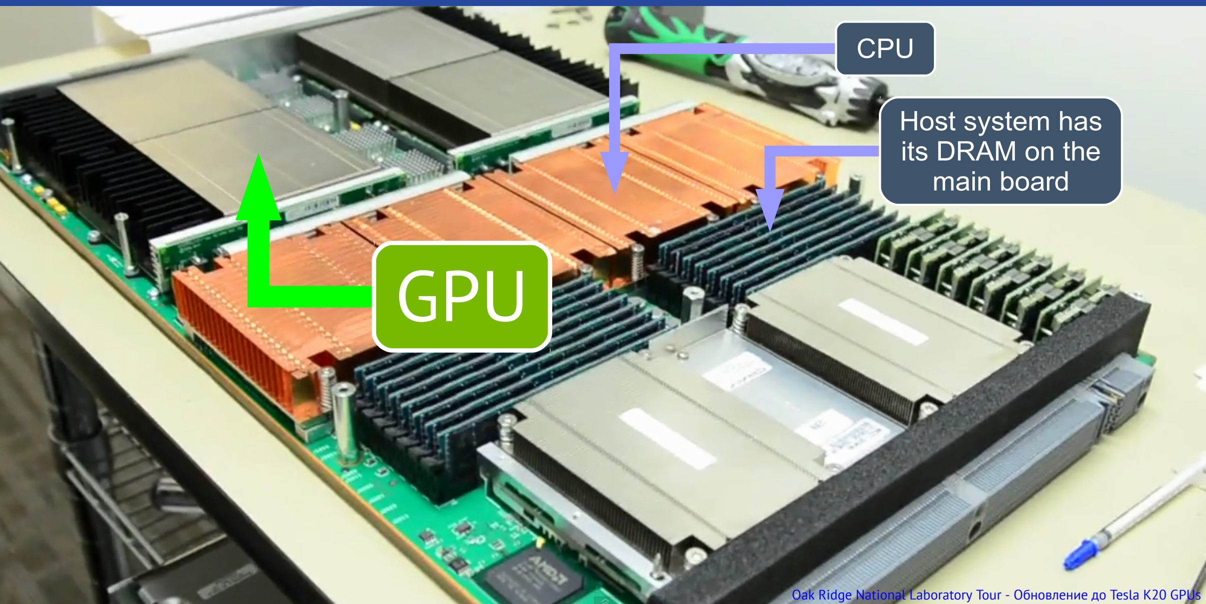
GPU в системах высокопроизводительных вычислений



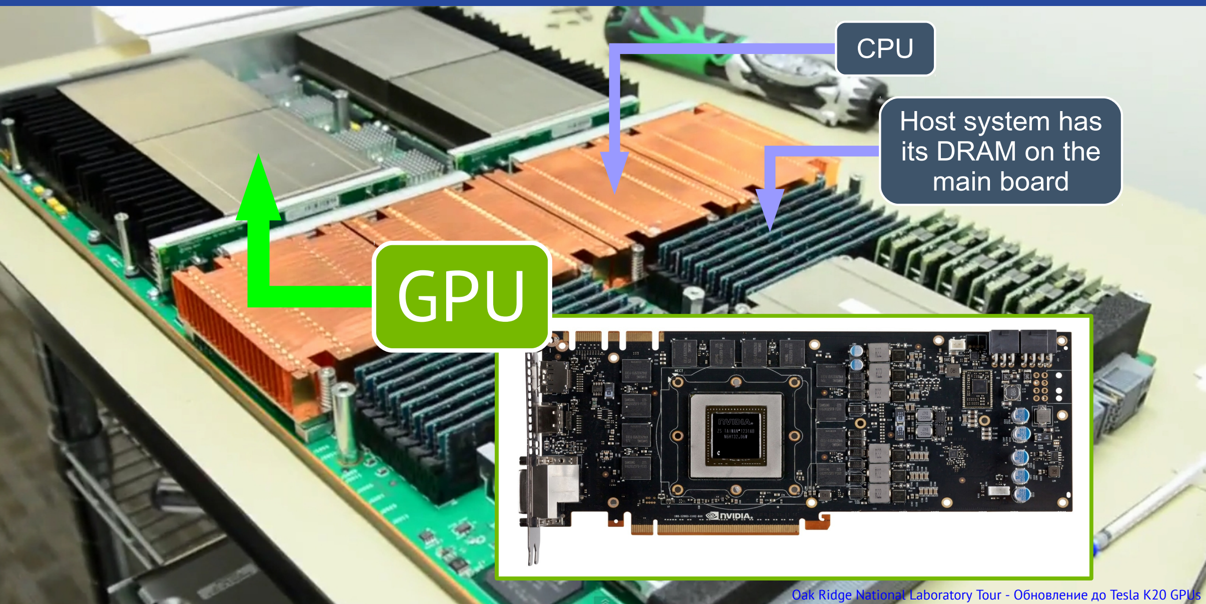
GPU в системах высокопроизводительных вычислений



GPU в системах высокопроизводительных вычислений

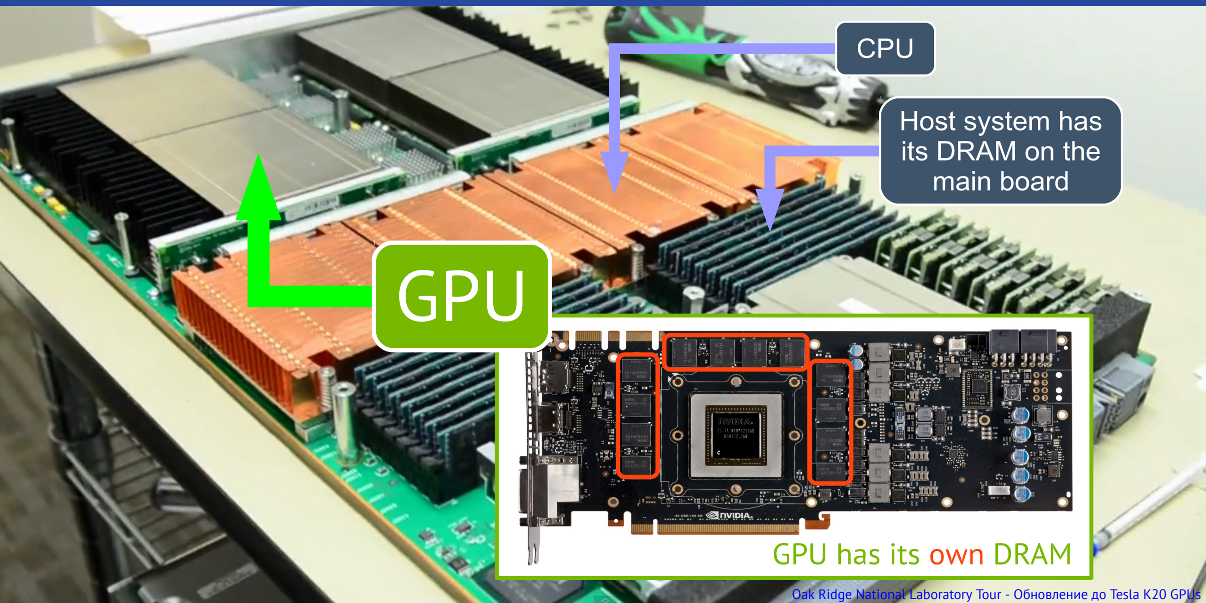


GPU в системах высокопроизводительных вычислений



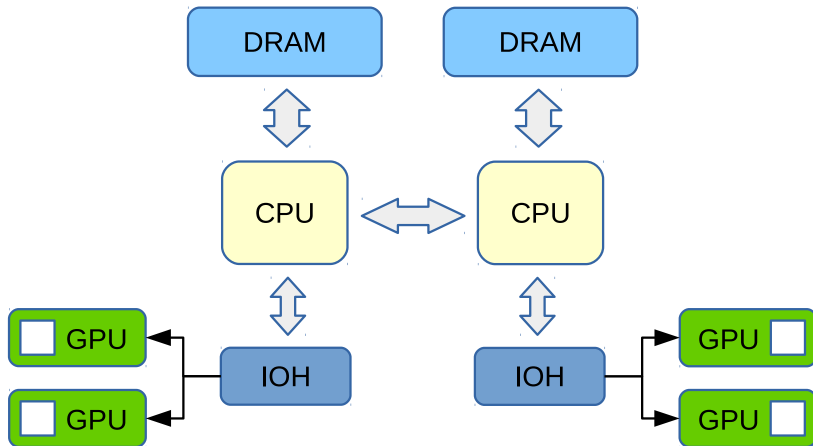
Oak Ridge National Laboratory Tour - Обновление до Tesla K20 GPUs

GPU в системах высокопроизводительных вычислений



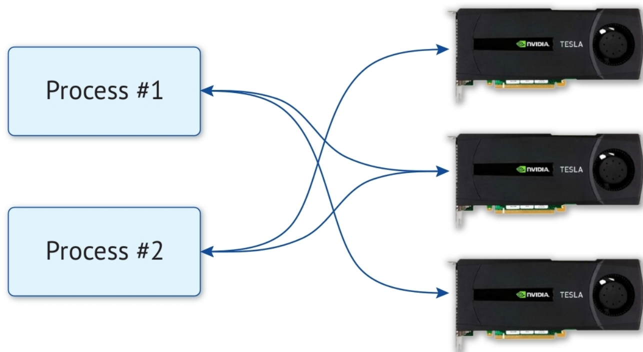
GPU в системах высокопроизводительных вычислений

- Узлы MultiGPU систем содержат 4-8 GPU на каждые 2-4 CPU сокета
- В зависимости от I/O Hub, связи GPU↔CPU могут быть неоднородными



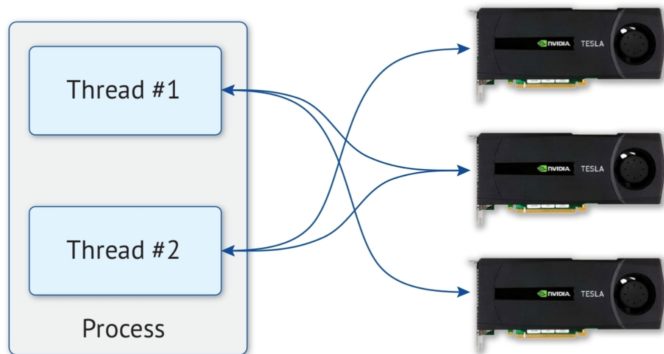
GPU в системах высокопроизводительных вычислений: MPI

- Многопроцессные приложения (например, MPI) могут использовать несколько GPU
- Один процесс может использовать несколько GPUs, несколько процессов могут использовать один и тот же GPU (“Hyper-Q”)



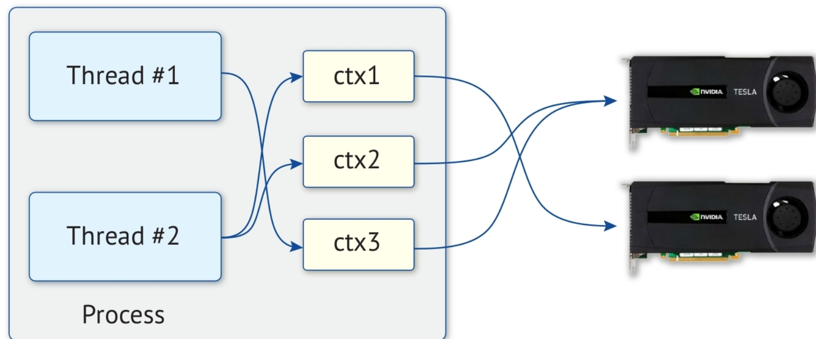
GPU в системах высокопроизводительных вычислений: потоки

- Многопоточные приложения (OpenMP, POSIX, Boost, и т.д.) могут использовать несколько GPU
- Один поток может использовать несколько GPUs, несколько потоков могут использовать один GPU



GPU в системах высокопроизводительных вычислений: контекст

- Все комбинации потоков и процессов обрабатываются самим GPU
- Благодаря контекстам, GPU может быть использовано в любом существующем MPI+OpenMP приложении



Эволюция экосистемы GPU

2008:

- GPU был технологией для экспертов
- Любая GPU программа должна была быть написана на CUDA, или с использованием шейдеров
- Работало только на одной конкретной линейке GPU

2017:

- Вычилсения на GPU просты в использовании и доступны всем
- Множество бесплатных GPU библиотек, реализующих все виды обработки данных: CUBLAS, MAGMA, CUSPARSE, AmgX, CUFFT, Thrust, CUB, cuDNN и т.д.
- Директивные языки программирования упрощают написание программ для GPU: OpenACC, OpenMP4
- Работает на 6 поколениях GPU: G80 (2006), GT200 (2008), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016)
- CUDA требуется только в крайних случаях: программирование специфических задач, или оптимизация для достижения максимальной производительности

Эволюция экосистемы GPU

2008:

- GPU был технологией для экспертов
- Любая GPU программа должна была быть написана на CUDA, или с использованием шейдеров
- Работало только на одной конкретной линейке GPU

2017:

- Вычилсения на GPU просты в использовании и доступны всем
- Множество бесплатных GPU библиотек, реализующих все виды обработки данных: CUBLAS, MAGMA, CUSPARSE, AmgX, CUFFT, Thrust, CUB, cuDNN и т.д.
- Директивные языки программирования упрощают написание программ для GPU: OpenACC, OpenMP4
- Работает на 6 поколениях GPU: G80 (2006), GT200 (2008), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016)
- CUDA требуется только в крайних случаях: программирование специфических задач, или оптимизация для достижения максимальной производительности

Три уровня разработки приложений для GPU

Black box: использование GPU через библиотеки, поддерживающие GPU вычисления

Некоторые знания о GPU: подсказки компилятору, как и какие циклы и данные необходимо перенести на GPU

CUDA: написание CUDA–ядер вручную

Три уровня разработки приложений для GPU

Black box: использование GPU через библиотеки, поддерживающие GPU вычисления

Некоторые знания о GPU: подсказки компилятору, как и какие циклы и данные необходимо перенести на GPU

CUDA: написание CUDA–ядер вручную

Три уровня разработки приложений для GPU

Black box: использование GPU через библиотеки, поддерживающие GPU вычисления

Некоторые знания о GPU: подсказки компилятору, как и какие циклы и данные необходимо перенести на GPU

CUDA: написание CUDA–ядер вручную

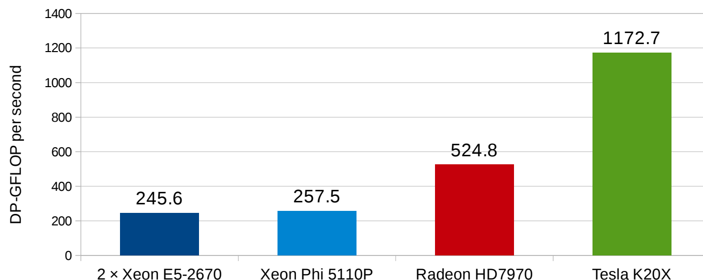
Три уровня разработки приложений для GPU

Black box: использование GPU через библиотеки, поддерживающие GPU вычисления

Некоторые знания о GPU: подсказки компилятору, как и какие циклы и данные необходимо перенести на GPU

CUDA: написание CUDA–ядер вручную

BLAS DGEMM



(выше – лучше)

hardware	#cores	software	compiler
Xeon E5-2670 $\times 2$	8 $\times 2$	MKL	icpc -g -O3 -xHost
Xeon 5110P	60	MKL, native	icpc -g -O3 -mmic
Radeon HD7970	2,048	clAmdBlas	g++ -g -O3
Kepler K20X	2,496	CUBLAS	g++ -g -O3

icpc/mkl: 2013.2.146, cublas: 5.5

BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
          n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
          n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```

BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

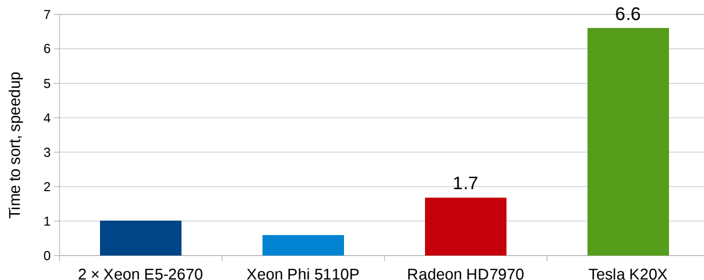
GPU-enabled BLAS

В CUDA 6.0 введена библиотека **NVBLAS**:

- Автоматически перенаправляет стандартные вызовы BLAS3 функций в CUBLAS
- Может распределять работу между несколькими GPU (cuBLAS-XT)
- Можно использовать с любыми приложениями, работающими с BLAS3 функциями (Octave, Scilab, и т.д.)

```c
double alpha = 1.0, beta = 0.0;
dgemm_(n, n,
       &n, &n, &n, &alpha, A, &n, B, &n, &beta, C, &n);
```

Сортировка 128M пар ключ-значение



(выше – лучше)

hardware	#cores	software	compiler
Xeon E5-2670 $\times 2$	8 $\times 2$	TBB, mergesort	icpc -ipo -O3 -no-prec-div -xHost
Xeon 5110P	60	TBB, mergesort, native	icpc -ipo -O3 -no-prec-div -mmic
Radeon HD79790	2,048	Bolt, GPU sort	g++ -g -O3 -std=c++0x
Kepler K20X	2,496	Thrust, GPU radix sort	nvcc -arch=sm_35 -O3

icpc: 2013.2.146, tbb: 4.1, thrust: 5.5

Сортировка 128M пар ключ-значение

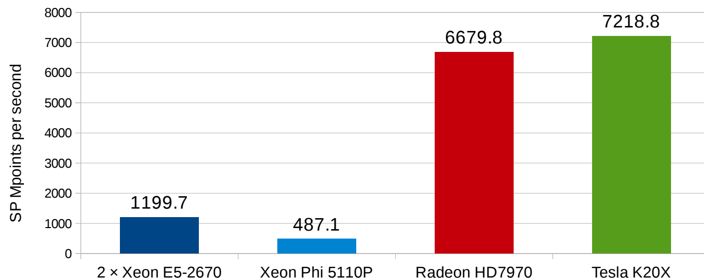
```
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

// Generate the random keys and values on the host.
host_vector<float> h_keys(n);
host_vector<float> h_vals(n);
for (int i = 0; i < n; i++)
{
    h_keys[i] = drand48();
    h_vals[i] = drand48();
}

// Transfer data to the device.
device_vector<float> d_keys = h_keys;
device_vector<float> d_vals = h_vals;

// Sort!
sort_by_key(d_keys.begin(), d_keys.end(), d_vals.begin());
cudaDeviceSynchronize();

// Transfer data back to host.
copy(d_keys.begin(), d_keys.end(), h_keys.begin());
copy(d_vals.begin(), d_vals.end(), h_vals.begin());
```

Сортировка 128M пар ключ-значение

```
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

// Generate the random keys and values on the host.
host_vector<float> h_keys(n);
host_vector<float> h_vals(n);
for (int i = 0; i < n; i++)
{
    h_keys[i] = drand48();
    h_vals[i] = drand48();
}

// Transfer data to the device.
device_vector<float> d_keys = h_keys;
device_vector<float> d_vals = h_vals;

// Sort!
sort_by_key(d_keys.begin(), d_keys.end(), d_vals.begin());
cudaDeviceSynchronize();

// Transfer data back to host.
copy(d_keys.begin(), d_keys.end(), h_keys.begin());
copy(d_vals.begin(), d_vals.end(), h_vals.begin());
```

Сравнительно новая CUB библиотека от NVResearch предлагает сортировку Radix в **2.5**$\times$ быстрее чем Thrust!

Распространение волны



(выше – лучше)

hardware	#cores	software	compiler
Xeon E5-2670 ×2	8 ×2	C, OpenMP	icc -O3 -xHost -openmp -opt-subscript-in-range -align
Xeon 5110P	60	C, OpenMP, offload	icc -O3 -xHost -openmp -opt-subscript-in-range -align
Radeon HD7970	2,048	OpenACC	pgcc -acc -O3 -ta=radeon,tahiti
Kepler K20X	2,496	OpenACC	pgcc -acc -O3 -ta=nvidia:keep,cc35,cuda5.5

icpc: 2013.2.146, cublas: 5.5

Распространение волны

```
void wave13pt(const int nx, const int ny, const int ns,
    const real m0, const real m1, const real m2,
    const real* const __restrict__ w0, const real* const __restrict__ w1,
    real* const __restrict__ w2)
{
    size_t szarray = (size_t)nx * ny * ns;
    #pragma acc kernels loop independent gang(ns), present(w0[0:szarray], w1[0:szarray], w2[0:szarray])
    for (int k = 2; k < ns - 2; k++)
    {
        #pragma acc loop independent
        for (int j = 2; j < ny - 2; j++)
        {
            #pragma acc loop independent vector(512)
            for (int i = 2; i < nx - 2; i++)
            {
                _A(w2, k, j, i) =  m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +

                    m1 * (
                        _A(w1, k, j, i+1) + _A(w1, k, j, i-1)  +
                        _A(w1, k, j+1, i) + _A(w1, k, j-1, i)  +
                        _A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
                    m2 * (
                        _A(w1, k, j, i+2) + _A(w1, k, j, i-2)  +
                        _A(w1, k, j+2, i) + _A(w1, k, j-2, i)  +
                        _A(w1, k+2, j, i) + _A(w1, k-2, j, i));
            } // i-loop
        } // j-loop
    } // k-loop
}
```

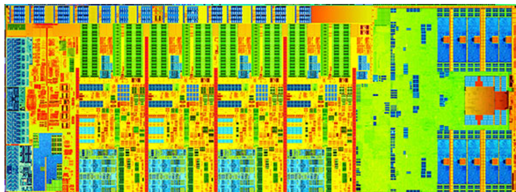
Распространение волны

```
void wave13pt(const int nx, const int ny, const int ns,
    const real m0, const real m1, const real m2,
    const real* const __restrict__ w0, const real* const __restrict__ w1,
    real* const __restrict__ w2)
{
    size_t szarray = (size_t)nx * ny * ns;
    #pragma acc kernels loop independent gang(ns), present(w0[0:szarray], w1[0:szarray], w2[0:szarray])
    for (int k = 2; k < ns - 2; k++)
    {
        #pragma acc loop independent
        for (int j = 2; j < ny - 2; j++)
        {
            #pragma acc loop independent vector(512)
            for (int i = 2; i < nx - 2; i++)
            {
                _A(w2, k, j, i) =  m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +

                    m1 * (
                        _A(w1, k, j, i+1) + _A(w1, k, j, i-1)  +
                        _A(w1, k, j+1, i) + _A(w1, k, j-1, i)  +
                        _A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
                    m2 * (
                        _A(w1, k, j, i+2) + _A(w1, k, j, i-2)  +
                        _A(w1, k, j+2, i) + _A(w1, k, j-2, i)  +
                        _A(w1, k+2, j, i) + _A(w1, k-2, j, i));
            } // i-loop
        } // j-loop
    } // k-loop
}
```

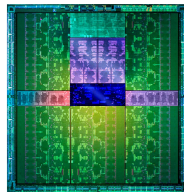
OpenACC: переносит вычисления на GPU с высокой производительностью:

- PGI OpenACC 14.1 часто генерирует ядра, быстрее чем написанные на CUDA вручную

Архитектура CPU **vs** архитектура GPU



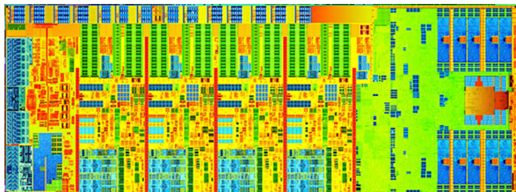
VS



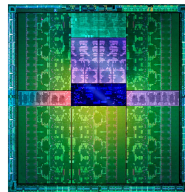
- Несколько сложных ядер
- Векторизация SIMD
- Большие размеры кешей

- Тысячи простых вычислительных юнитов
- Тысячи скалярных ядер
- Большой регистровый файл и разделяемая/общая (shared) память

Архитектура CPU **vs** архитектура GPU



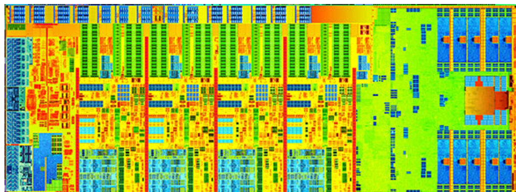
VS



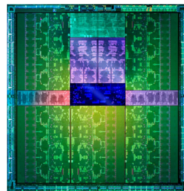
- Несколько сложных ядер
- Векторизация SIMD
- Большие размеры кешей

- Тысячи простых вычислительных юнитов
- Тысячи скалярных ядер
- Большой регистровый файл и разделяемая/общая (shared) память

Архитектура CPU **vs** архитектура GPU



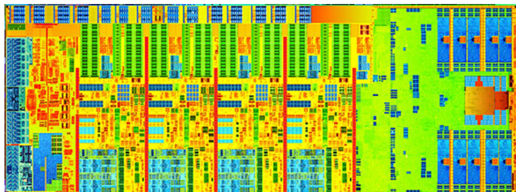
VS



- Несколько сложных ядер
- Векторизация SIMD
- Большие размеры кешей

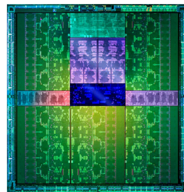
- Тысячи простых вычислительных юнитов
- Тысячи скалярных ядер
- Большой регистровый файл и разделяемая/общая (shared) память

Архитектура CPU **vs** архитектура GPU



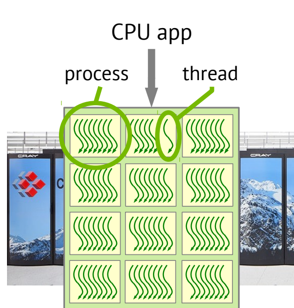
- Несколько сложных ядер
- Векторизация SIMD
- Большие размеры кешей

VS

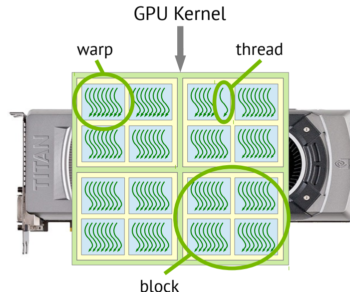


- Тысячи простых вычислительных юнитов
- Тысячи скалярных ядер
- Большой регистровый файл и разделяемая/общая (shared) память

Модель исполнения CPU **vs** модель исполнения GPU



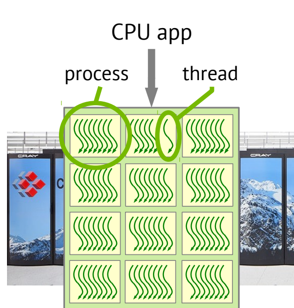
VS



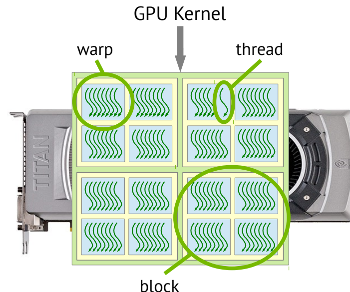
- Процессы и потоки управляются ОС
- Латентность памяти скрыта 3 уровнями кеша

- Легковесные нити управляются драйвером GPU
- Латентность памяти скрыта с помощью переключения между активными варпами

Модель исполнения CPU **vs** модель исполнения GPU



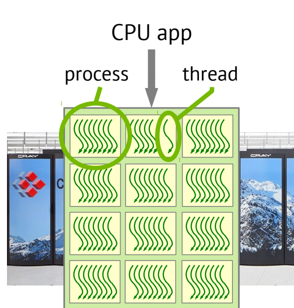
VS



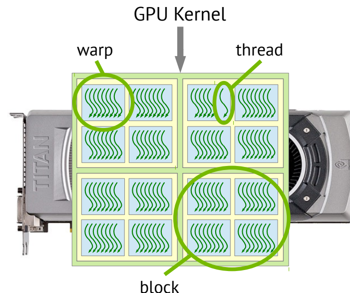
- Процессы и потоки управляются ОС
- Латентность памяти скрыта 3 уровнями кеша

- Легковесные нити управляются драйвером GPU
- Латентность памяти скрыта с помощью переключения между активными варпами

Модель исполнения CPU **vs** модель исполнения GPU



VS



- Процессы и потоки управляются ОС
- Латентность памяти скрыта 3 уровнями кеша

- Легковесные нити управляются драйвером GPU
- Латентность памяти скрыта с помощью переключения между активными варпами

Почему GPU настолько эффективны?

GPU приложение

GPGPU вычисления скалярны ⇒ не требуется сложных операций для векторизации!

1. Задача для GPU должна быть разделена на большое количество маленьких параллельных задач, чтобы обрабатываться тысячами нитей GPU
2. Нити должны сбалансировано использовать два основных ресурса: регистры и shared память
3. Нити должны обращаться к последовательно расположенным адресам (coalescing)
4. Нити не должны содержать большого количества ветвлений `if...else`

Почему GPU настолько эффективны?

GPU приложение

GPGPU вычисления скалярны $\Rightarrow$ не требуется сложных операций для векторизации!

1. Задача для GPU должна быть разделена на большое количество маленьких параллельных задач, чтобы обрабатываться тысячами нитей GPU
2. Нити должны сбалансировано использовать два основных ресурса: регистры и shared память
3. Нити должны обращаться к последовательно расположенным адресам (coalescing)
4. Нити не должны содержать большого количества ветвлений `if...else`

Почему GPU настолько эффективны?

Разделено на много маленьких параллельных задач

GPGPU вычисления скалярны ⇒ не требуется сложных операций для векторизации!

1. Задача для GPU должна быть разделена на большое количество маленьких параллельных задач, чтобы обрабатываться тысячами нитей GPU
2. Нити должны сбалансировано использовать два основных ресурса: регистры и shared память
3. Нити должны обращаться к последовательно расположенным адресам (coalescing)
4. Нити не должны содержать большого количества ветвлений `if...else`

Почему GPU настолько эффективны?

Разделено на много маленьких параллельных задач

GPGPU вычисления скалярны ⇒ не требуется сложных операций для векторизации!

1. Задача для GPU должна быть разделена на большое количество маленьких параллельных задач, чтобы обрабатываться тысячами нитей GPU
2. Нити должны сбалансировано использовать два основных ресурса: регистры и shared память
3. Нити должны обращаться к последовательно расположенным адресам (coalescing)
4. Нити не должны содержать большого количества ветвлений `if...else`

Почему GPU настолько эффективны?

Разделено на много маленьких параллельных задач

GPGPU вычисления скалярны ⇒ не требуется сложных операций для векторизации!

1. Задача для GPU должна быть разделена на большое количество маленьких параллельных задач, чтобы обрабатываться тысячами нитей GPU
2. Нити должны сбалансировано использовать два основных ресурса: регистры и shared память
3. Нити должны обращаться к последовательно расположенным адресам (coalescing)
4. Нити не должны содержать большого количества ветвлений `if...else`

Почему GPU настолько эффективны?

Разделено на много маленьких параллельных задач

GPGPU вычисления скалярны $\Rightarrow$ не требуется сложных операций для векторизации!

1. Задача для GPU должна быть разделена на большое количество маленьких параллельных задач, чтобы обрабатываться тысячами нитей GPU
2. Нити должны сбалансировано использовать два основных ресурса: регистры и shared память
3. Нити должны обращаться к последовательно расположенным адресам (coalescing)
4. Нити не должны содержать большого количества ветвлений `if...else`

Обзор эффективности GPU

1. Что если количество параллельных задач слишком мало?

$\Rightarrow$ Вычислительные ресурсы расходуются неоптимально

1. Если количество задач гораздо больше числа ядер – отлично:

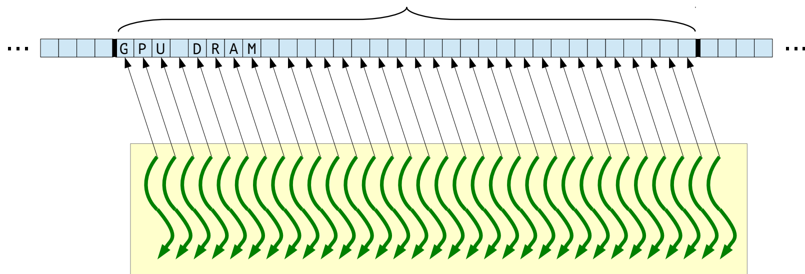
⇒ В этом случае оставшиеся задачи будут автоматически обработаны после завершения первой группы

Обзор эффективности GPU

2. Если использование регистров и shared несбалансировано:

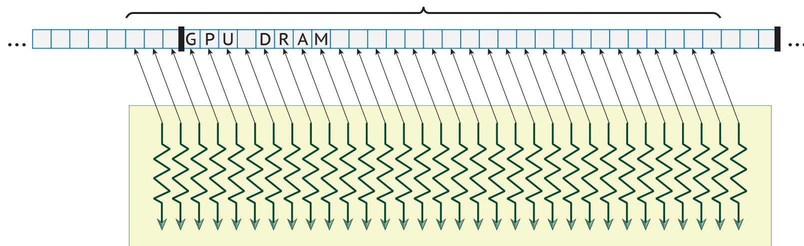
Несмотря на то, что есть свободные ресурсы, исполняется не максимально возможное количество задач
$\Rightarrow$ остальные задачи вынуждены ждать

3. Нити должны запрашивать последовательно расположенные адреса (coalescing):



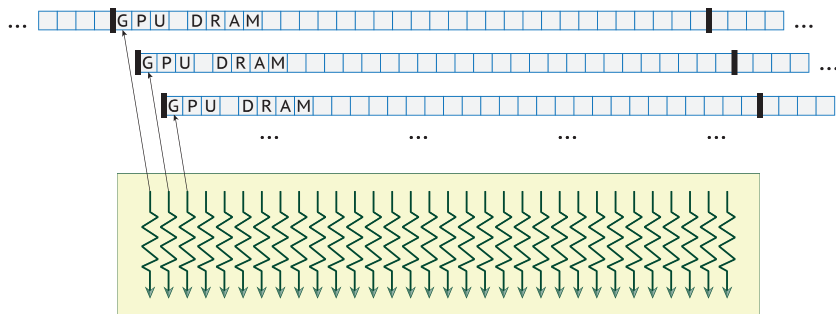
Обзор эффективности GPU

3. Нити должны запрашивать последовательно расположенные адреса (coalescing):



A warp – synchronously executed group of 32 threads

3. Нити должны запрашивать последовательно расположенные адреса (coalescing):



4 Нити не должны содержать большого количества ветвлений `if...else`

Нити с флагом false не делают ничего (простаивают) ⇒ нет ветвления, просто небольшая трата вычислительных ресурсов

```
if (flags[threadIdx.x])
{
    // A lot of code
}
```

Каждая нить выполняет свою ветвь, после чего простаивает, пока остальные выполняют альтернативную (если имеется хотя бы одна такая нить) ⇒ дивергентное ветвление

```
if (flags[threadIdx.x])
{
    // A lot of code
}
else
{
    // A lot of code
}
```

Заключение

1. GPU предлагает гораздо большую эффективность в массивно-параллельных вычислениях
2. Вы можете начать использовать GPU в вычислениях очень легко:
   - Простая интеграция в приложения с уже существующим параллелизмом (MPI, OpenMP)
   - Высокопроизводительные GPU библиотеки для многих типов вычислительных задач
   - CUDA требуется только для разработки специфичных задач или высокой оптимизации кода
3. Архитектура и исполнительная модель GPU созданы для массивного параллелизма
4. Эффективность GPU основана на 4 факторах:
   - Достаточный параллелизм
   - Сбалансированное использование регистров и shared памяти
   - Эффективное использование глобальной (DRAM) памяти
   - Минимизация дивергентных ветвлений

Заключение

1. GPU предлагает гораздо большую эффективность в массивно-параллельных вычислениях
2. Вы можете начать использовать GPU в вычислениях очень легко:
   - Простая интеграция в приложения с уже существующим параллелизмом (MPI, OpenMP)
   - Высокопроизводительные GPU библиотеки для многих типов вычислительных задач
   - CUDA требуется только для разработки специфичных задач или высокой оптимизации кода
3. Архитектура и исполнительная модель GPU созданы для массивного параллелизма
4. Эффективность GPU основана на 4 факторах:
   - Достаточный параллелизм
   - Сбалансированное использование регистров и shared памяти
   - Эффективное использование глобальной (DRAM) памяти
   - Минимизация дивергентных ветвлений

Заключение

1. GPU предлагает гораздо большую эффективность в массивно-параллельных вычислениях
2. Вы можете начать использовать GPU в вычислениях очень легко:
   - Простая интеграция в приложения с уже существующим параллелизмом (MPI, OpenMP)
   - Высокопроизводительные GPU библиотеки для многих типов вычислительных задач
   - CUDA требуется только для разработки специфичных задач или высокой оптимизации кода
3. Архитектура и исполнительная модель GPU созданы для массивного параллелизма
4. Эффективность GPU основана на 4 факторах:
   - Достаточный параллелизм
   - Сбалансированное использование регистров и shared памяти
   - Эффективное использование глобальной (DRAM) памяти
   - Минимизация дивергентных ветвлений

Заключение

1. GPU предлагает гораздо большую эффективность в массивно-параллельных вычислениях
2. Вы можете начать использовать GPU в вычислениях очень легко:
   - Простая интеграция в приложения с уже существующим параллелизмом (MPI, OpenMP)
   - Высокопроизводительные GPU библиотеки для многих типов вычислительных задач
   - CUDA требуется только для разработки специфичных задач или высокой оптимизации кода
3. Архитектура и исполнительная модель GPU созданы для массивного параллелизма
4. Эффективность GPU основана на 4 факторах:
   - Достаточный параллелизм
   - Сбалансированное использование регистров и shared памяти
   - Эффективное использование глобальной (DRAM) памяти
   - Минимизация дивергентных ветвлений