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

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Introduction to GPU architecture and computing

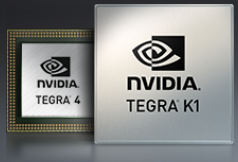
Dmitry Mikushin

July 7, 2014

GPU Computing is everywhere!

Courtesy of NVIDIA

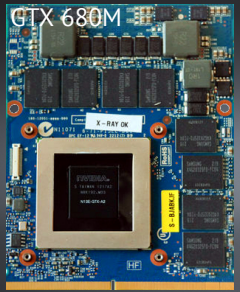
GPU Computing is everywhere!



- Augmented reality
- Autonomous robots, unmanned aircrafts
- ...
- Energy-efficient data centers

Courtesy of NVIDIA

GPU Computing is everywhere!



- Mobile GPU workstation
- GPU-accelerated CAD/CAE
- GPU apps development
- ...

Courtesy of NVIDIA

GPU Computing is everywhere!

Tesla K40

- GPU workstation
- GPU-accelerated CAD/CAE, numerical simlations (CFD, seismic, oil&gas)
- Multiple GPUs in a single machine – MultiGPU
- ...

Courtesy of NVIDIA

GPU Computing is everywhere!

Tesla K20X

- GPU server/cluster
- GPU-accelerated CAD/CAE, numerical simlations (CFD, seismic, oil&gas)
- Multiple GPUs in a single server – MultiGPU
- Graphics virtualization (NVIDIA GRID), remote GPU computing
- ...

Courtesy of NVIDIA

GPU Computing is everywhere!

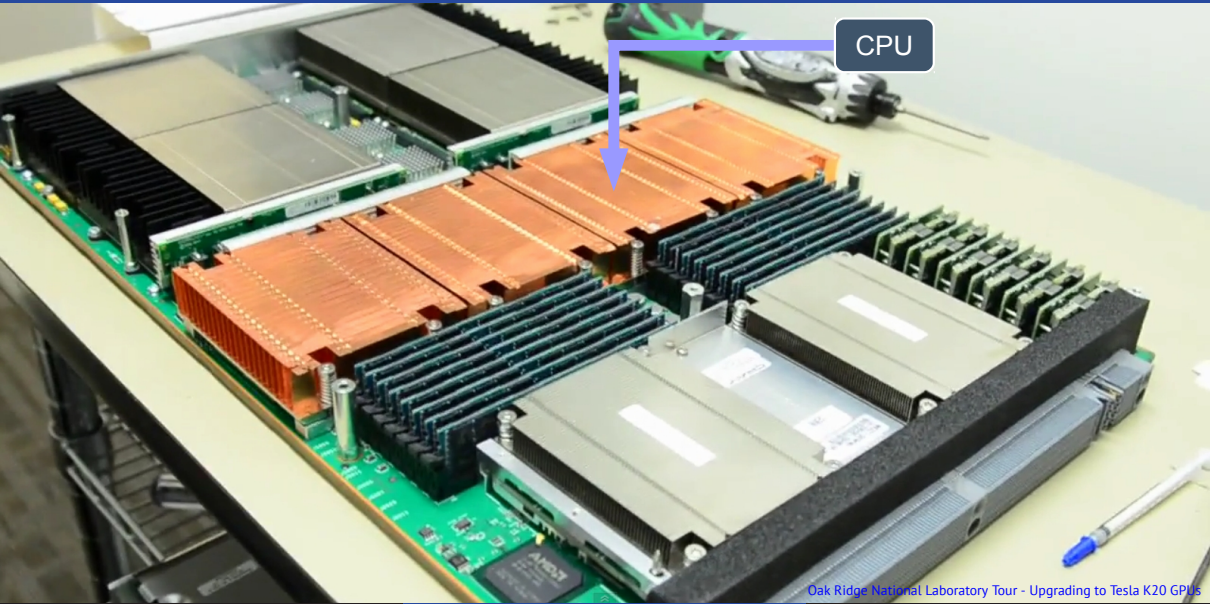
- Infotainment and navigation
- Advanced driver assistance systems (ADAS)
- Reat seat entertainment

Courtesy of NVIDIA

GPU in HPC system

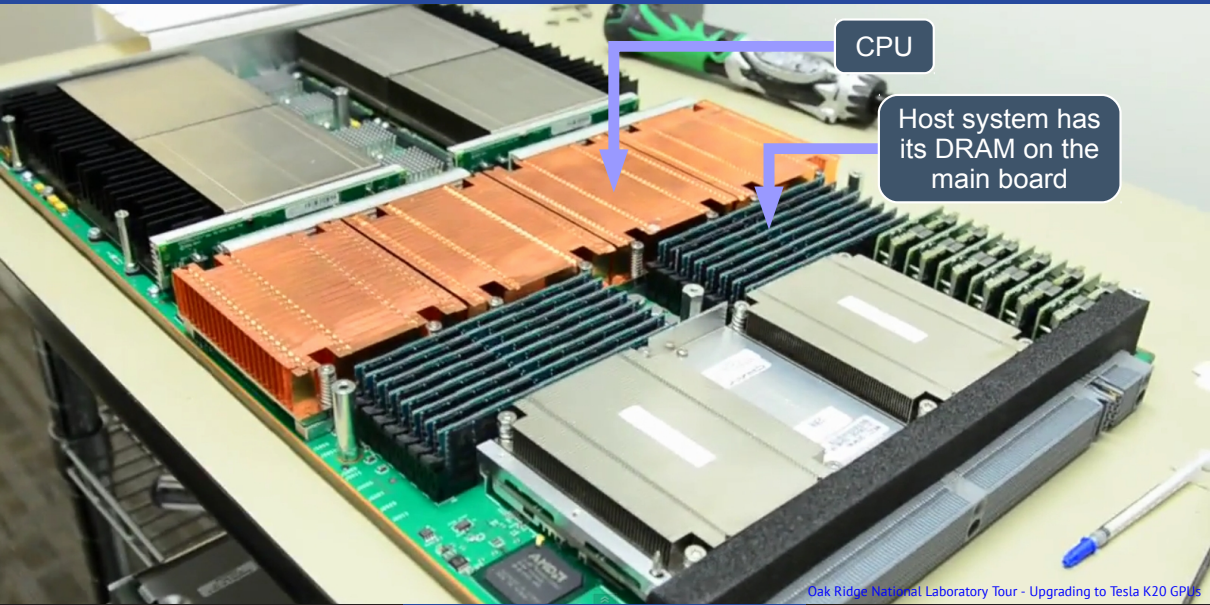
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU in HPC system



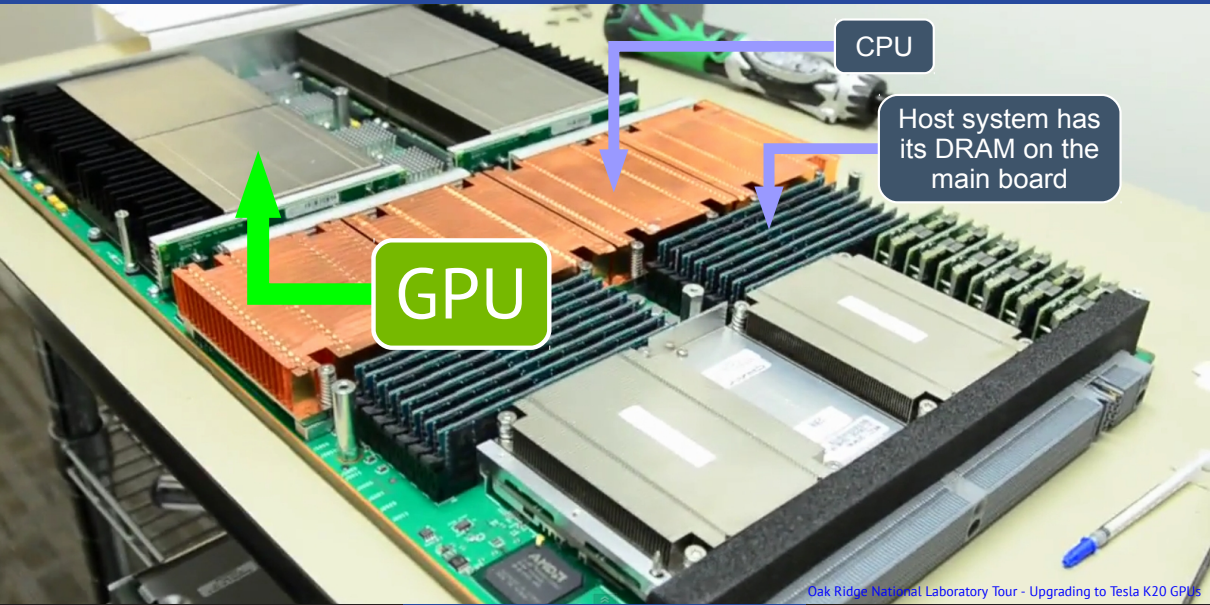
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU in HPC system



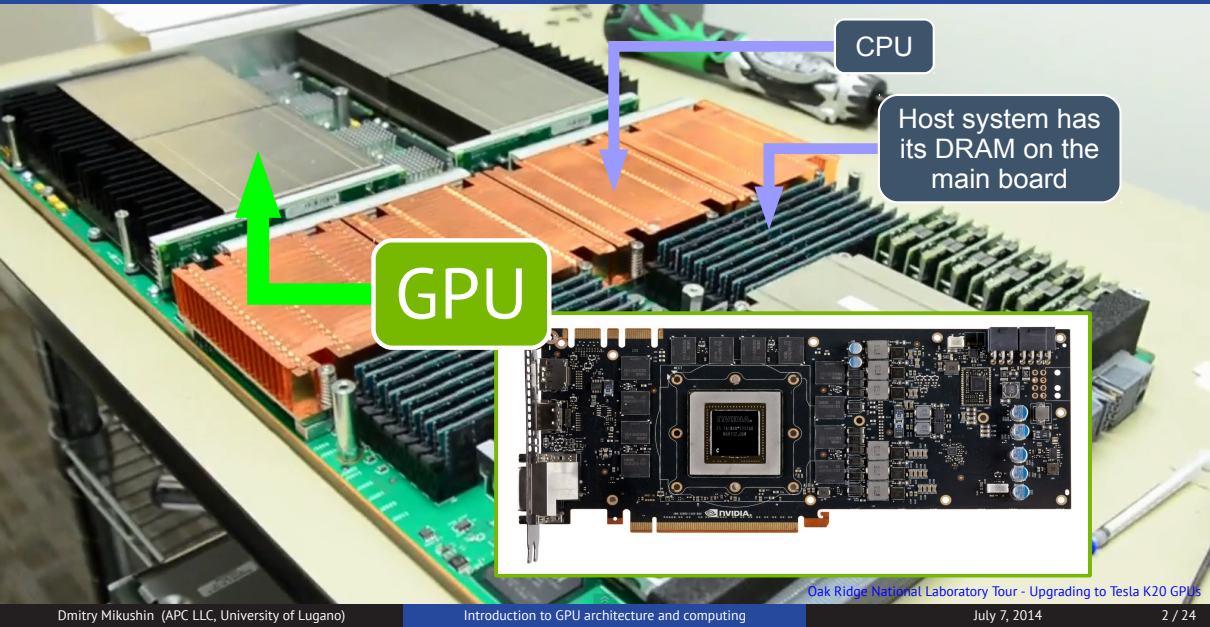
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU in HPC system



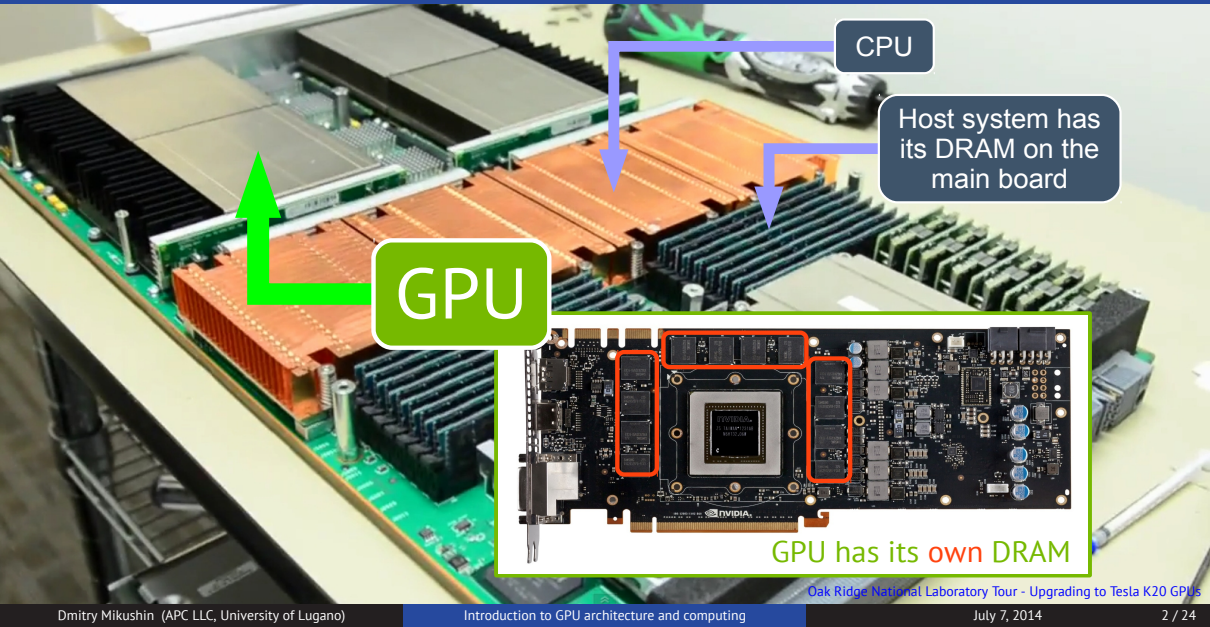
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU in HPC system



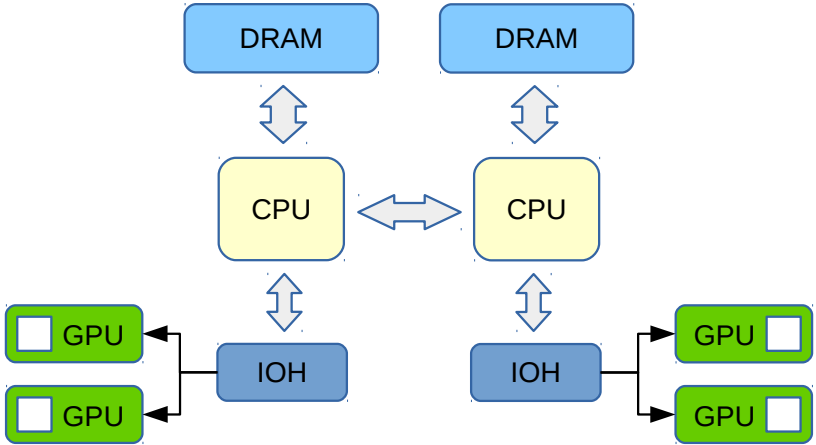
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU in HPC system



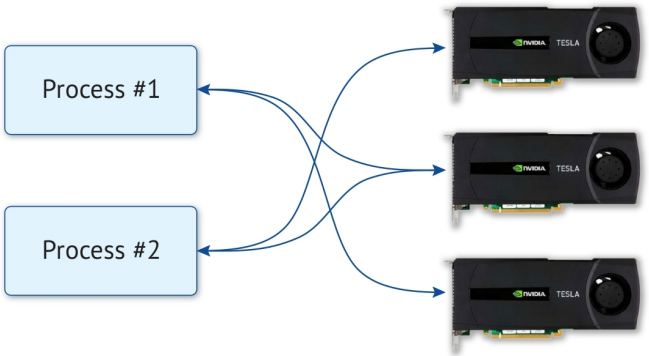
GPU in HPC system

- MultiGPU systems are “fat” nodes: 4-8 GPUs per 2-4 CPU sockets
- Depending on I/O Hub, GPU↔CPU links might be non-uniform



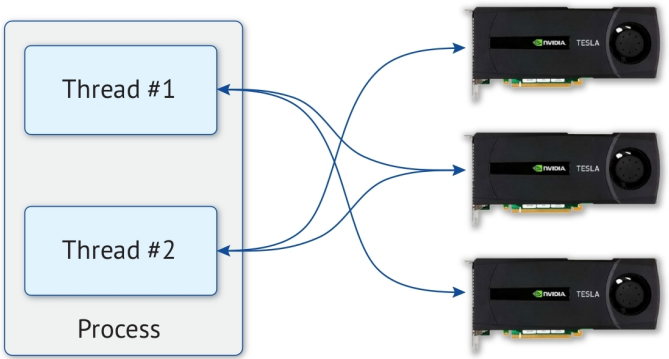
GPU in HPC system: MPI

- Multi-process applications (e.g. MPI) can use multiple GPUs
- Single process can use multiple GPUs, multiple processes can use the same GPU (“Hyper-Q”)



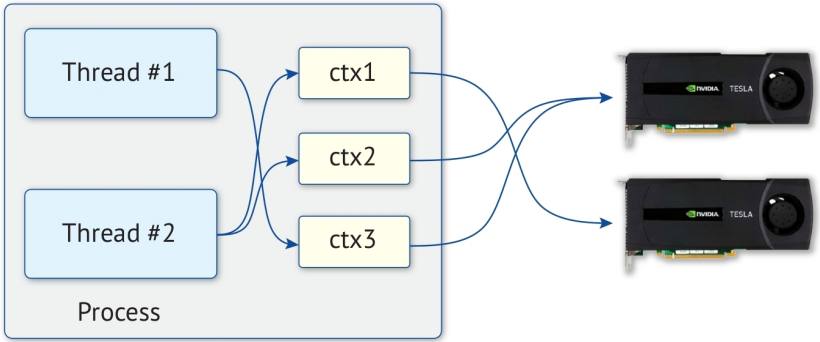
GPU in HPC system: threads

- Multithread applications (OpenMP, POSIX, Boost, etc.) can use multiple GPUs
- Single thread can use multiple GPUs, multiple threads can use the same GPU



GPU in HPC system: context

- Underneath any combination of threads and processes is handled by GPU *contexts*
- Thanks to contexts, GPU could be utilized in any existing MPI+OpenMP application



Evolution of GPU ecosystem

2008:

- GPU computing was a technology for experts
- Any GPU program had to be written in CUDA
- Worked only on one specific GPU family

2014:

- GPU computing is a technology for everyone
- A lot of free GPU-enabled libraries implement all kinds of data processing: CUBLAS, MAGMA, CUSPARSE, AmgX, CUFFT, Thrust, CUB, etc.
- Directive-based languages simplify GPU programming: OpenACC, HMPP
- Works on 5 generations of GPUs: G80 (2006), GT200 (2008), Fermi (2010), Kepler (2012), Maxwell (2014)
- CUDA is now needed only to program something very specific or outstandingly efficient

Evolution of GPU ecosystem

2008:

- GPU computing was a technology for experts
- Any GPU program had to be written in CUDA
- Worked only on one specific GPU family

2014:

- GPU computing is a technology for everyone
- A lot of free GPU-enabled libraries implement all kinds of data processing: CUBLAS, MAGMA, CUSPARSE, AmgX, CUFFT, Thrust, CUB, etc.
- Directive-based languages simplify GPU programming: OpenACC, HMPP
- Works on 5 generations of GPUs: G80 (2006), GT200 (2008), Fermi (2010), Kepler (2012), Maxwell (2014)
- CUDA is now needed only to program something very specific or outstandingly efficient

Three levels of GPU applications development

Black box: using GPU through a library that has GPU support underneath

Some GPU awareness: advice the compiler how to offload loops or data into GPU

CUDA: write GPU kernels manually

Three levels of GPU applications development

Black box: using GPU through a library that has GPU support underneath

OpenACC
DIRECTIVES FOR ACCELERATORS

Some GPU awareness: advice the compiler how to offload loops or data into GPU

NVIDIA
CUDA

CUDA: write GPU kernels manually

Three levels of GPU applications development

Black box: using GPU through a library that has GPU support underneath

OpenACC. DIRECTIVES FOR ACCELERATORS

Some GPU awareness: advice the compiler how to offload loops or data into GPU

NVIDIA CUDA

CUDA: write GPU kernels manually

Three levels of GPU applications development

Black box: using GPU through a library that has GPU support underneath

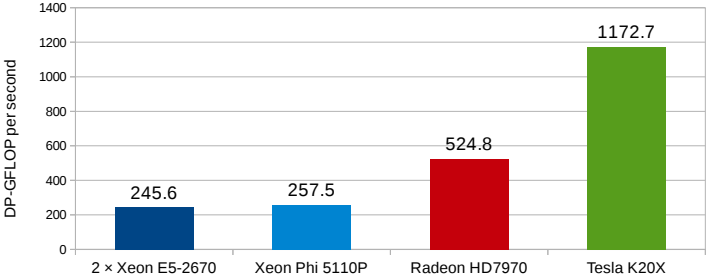
OpenACC. DIRECTIVES FOR ACCELERATORS

Some GPU awareness: advice the compiler how to offload loops or data into GPU

NVIDIA CUDA

CUDA: write GPU kernels manually

BLAS DGEMM



hardware	#cores	software	compiler
Xeon E5-2670 ×2	8 ×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

```
#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

```
#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);
```

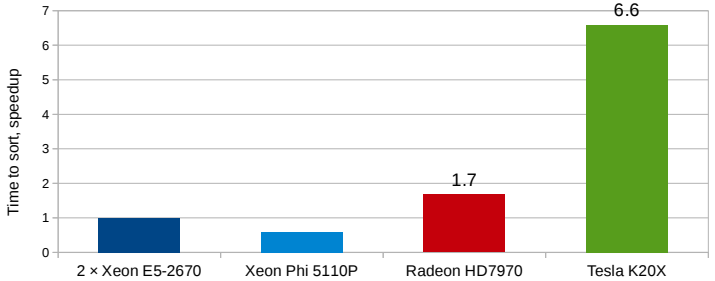
GPU-enabled BLAS

CUDA 6.0 introduces **NVBLAS** library:

- Automatically routes standard BLAS3 calls to CUBLAS (by preloading libnvblas.so)
- Can spread work across multiple GPUs (cuBLAS-XT)
- Usable with any app that calls BLAS3 functions (Octave, Scilab, etc.)

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

Sorting 128M key-value pairs



hardware	#cores	software	compiler
Xeon E5-2670 ×2	8 ×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

Sorting 128M key-value pairs

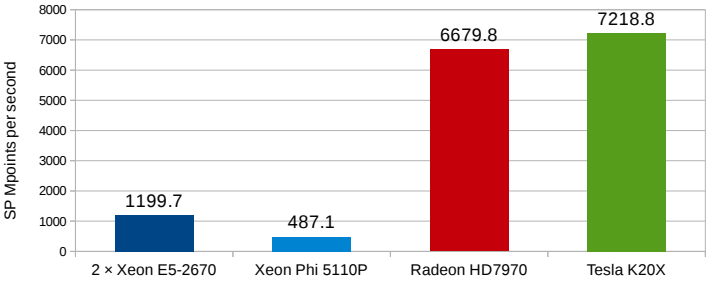
```
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

// 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());
```

Sorting 128M key-value pairs

```
#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());
```

Relatively new CUB library from NVResearch offers radix sort with **2.5**$\times$ higher perf than Thrust!

Wave propagation stencil



(higher is better)

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

Wave propagation stencil

```
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
}
```

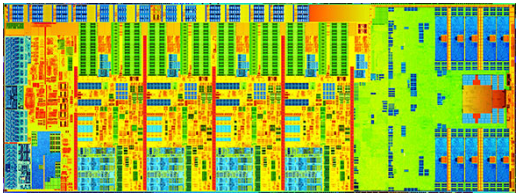
Wave propagation stencil

```
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
}
```

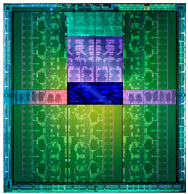
OpenACC: offload stencil computations to GPU with high efficiency:

- PGI OpenACC 14.1 often generates faster kernels than hand-written CUDA

CPU architecture **vs** GPU architecture



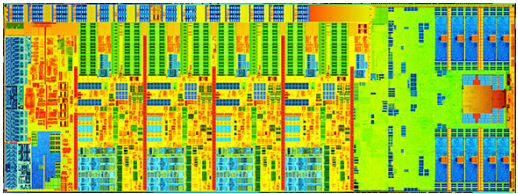
VS



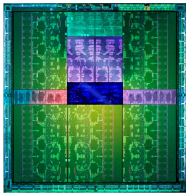
- Several complex cores
- SIMD vectorization
- Large caches

- Thousands of simple compute units
- Thousands of scalar cores
- Large register file + shared memory

CPU architecture **vs** GPU architecture



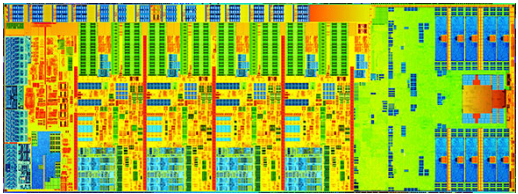
VS



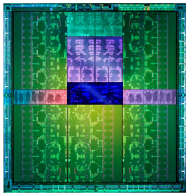
- Several complex cores
- SIMD vectorization
- Large caches

- Thousands of simple compute units
- Thousands of scalar cores
- Large register file + shared memory

CPU architecture **vs** GPU architecture



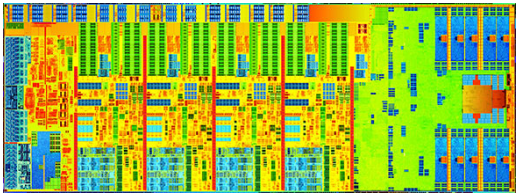
VS



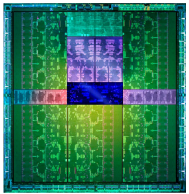
- Several complex cores
- SIMD vectorization
- Large caches

- Thousands of simple compute units
- Thousands of scalar cores
- Large register file + shared memory

CPU architecture **vs** GPU architecture



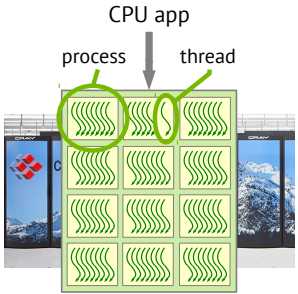
VS



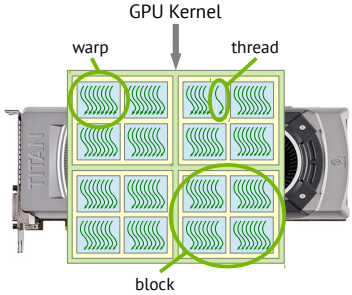
- Several complex cores
- SIMD vectorization
- Large caches

- Thousands of simple compute units
- Thousands of scalar cores
- Large register file + shared memory

CPU execution model **vs** GPU execution model



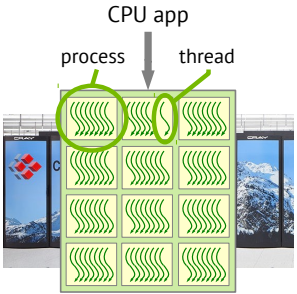
VS



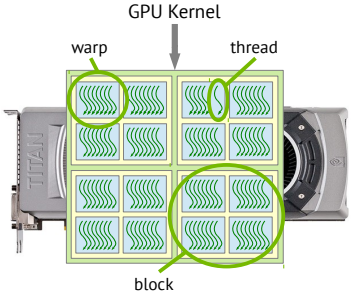
- Processes & threads maintained by OS
- Memory latency is relaxed with 3 levels of caches

- *Lightweight* threads maintained by GPU driver
- Memory latency is relaxed by active threads switching

CPU execution model **vs** GPU execution model



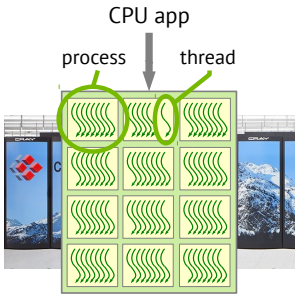
VS



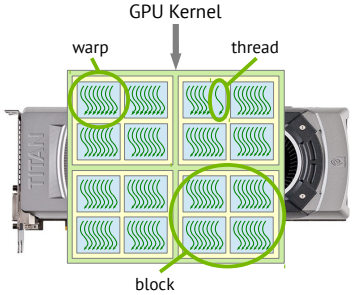
- Processes & threads maintained by OS
- Memory latency is relaxed with 3 levels of caches

- *Lightweight* threads maintained by GPU driver
- Memory latency is relaxed by active threads switching

CPU execution model **vs** GPU execution model



VS



- Processes & threads maintained by OS
- Memory latency is relaxed with 3 levels of caches

- *Lightweight* threads maintained by GPU driver
- Memory latency is relaxed by active threads switching

What makes the GPU application efficient?

GPU application

GPGPU computations are scalar ⇒ no hard time with vectorization!

1. The GPU task should be partitioned into significant amount of small parallel tasks, to be processed by GPU threads
2. Task threads should maintain the *balanced use* of two key resources: registers and shared memory
3. Task threads altogether should access consecutive memory addresses (coalescing)
4. Task threads should not do divergent `if...else` branching

What makes the GPU application efficient?

GPU application

GPGPU computations are scalar $\Rightarrow$ no hard time with vectorization!

1. The GPU task should be partitioned into significant amount of small parallel tasks, to be processed by GPU threads
2. Task threads should maintain the *balanced use* of two key resources: registers and shared memory
3. Task threads altogether should access consecutive memory addresses (coalescing)
4. Task threads should not do divergent `if...else` branching

What makes the GPU application efficient?

Partitioned into small parallel tasks

GPGPU computations are scalar $\Rightarrow$ no hard time with vectorization!

1. The GPU task should be partitioned into significant amount of small parallel tasks, to be processed by GPU threads
2. Task threads should maintain the *balanced use* of two key resources: registers and shared memory
3. Task threads altogether should access consecutive memory addresses (coalescing)
4. Task threads should not do divergent `if...else` branching

What makes the GPU application efficient?

Partitioned into small parallel tasks

GPGPU computations are scalar $\Rightarrow$ no hard time with vectorization!

1. The GPU task should be partitioned into significant amount of small parallel tasks, to be processed by GPU threads
2. Task threads should maintain the *balanced use* of two key resources: registers and shared memory
3. Task threads altogether should access consecutive memory addresses (coalescing)
4. Task threads should not do divergent `if...else` branching

What makes the GPU application efficient?

Partitioned into small parallel tasks

GPGPU computations are scalar $\Rightarrow$ no hard time with vectorization!

1. The GPU task should be partitioned into significant amount of small parallel tasks, to be processed by GPU threads
2. Task threads should maintain the *balanced use* of two key resources: registers and shared memory
3. Task threads altogether should access consecutive memory addresses (coalescing)
4. Task threads should not do divergent `if...else` branching

What makes the GPU application efficient?

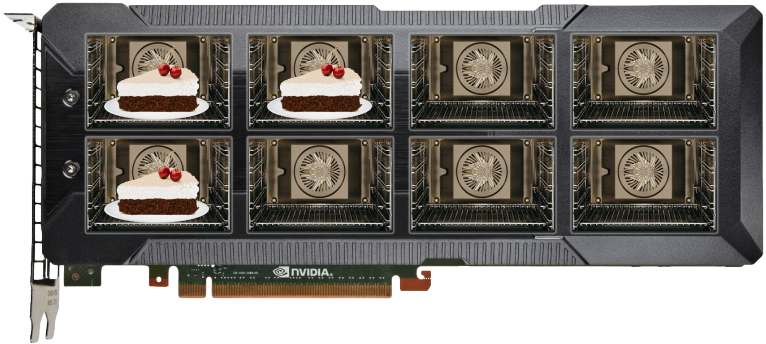
Partitioned into small parallel tasks

GPGPU computations are scalar $\Rightarrow$ no hard time with vectorization!

1. The GPU task should be partitioned into significant amount of small parallel tasks, to be processed by GPU threads
2. Task threads should maintain the *balanced use* of two key resources: registers and shared memory
3. Task threads altogether should access consecutive memory addresses (coalescing)
4. Task threads should not do divergent `if...else` branching

GPU efficiency overview

1. What if the amount of parallel tasks is too small?



$\Rightarrow$ Compute resources are wasted

GPU efficiency overview

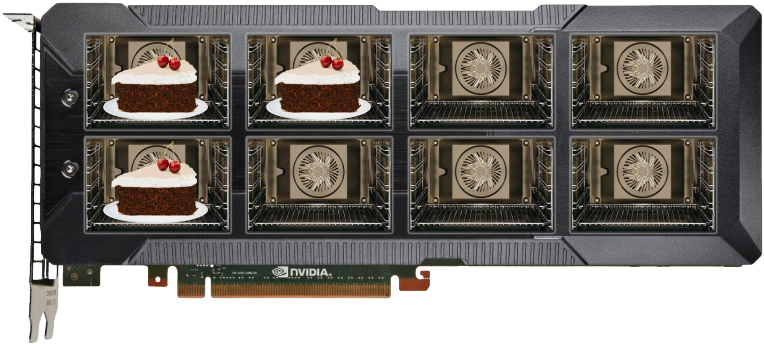
1. It's fine if amount of tasks is larger than the number of compute cores:

$\Rightarrow$ In this case the remaining tasks will be automatically processed after finishing with the first group

GPU efficiency overview

2. If the use of registers or shared memory is imbalanced:

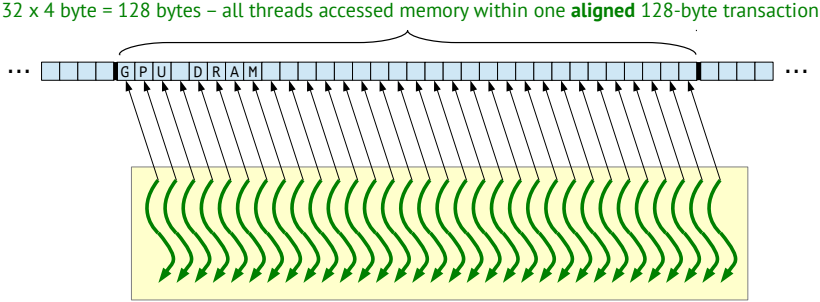
free registers



Although the free compute and shared memory resources exist, smaller amount of tasks already used all available shared memory $\Rightarrow$ other tasks have to wait, compute resources are wasted

GPU efficiency overview

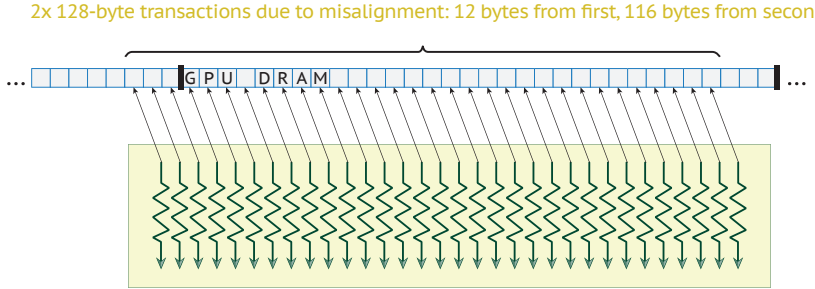
3 Task threads altogether should access consecutive memory addresses (coalescing):



A warp – synchronously executed group of 32 threads

GPU efficiency overview

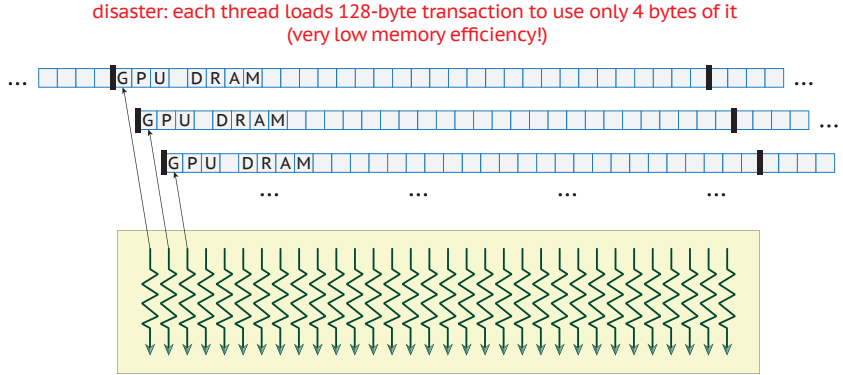
3 Task threads altogether should access consecutive memory addresses (coalescing):



A warp – synchronously executed group of 32 threads

GPU efficiency overview

3 Task threads altogether should access consecutive memory addresses (coalescing):



A warp – synchronously executed group of 32 threads

GPU efficiency overview

4 Task threads should not do divergent `if...else` branching

Threads, whose flag is false will do nothing (stall) $\Rightarrow$ no divergence, just some waste of compute resource

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

Each thread will do its branch and then stall while some other thread is doing alternative branch (if there is at least one such thread) $\Rightarrow$ divergent branching

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

Conclusion

1. GPUs offer much higher performance of massively-parallel computations
2. You can start using GPUs in your application very easily:
   - Seamless integration into applications with existing parallelism (MPI, OpenMP)
   - Fairly efficient GPU-enabled libraries exist for many types of computational problems
   - Nowadays CUDA is needed only to develop something very specific and/or highly-tuned
3. GPU architecture and execution model are designed for massive parallelism
4. GPU efficiency is based on 4 factors:
   - Sufficient parallelism
   - Balanced use of registers and shared memory
   - Efficient DRAM utilization
   - Minimization of divergent branching

Conclusion

1. GPUs offer much higher performance of massively-parallel computations
2. You can start using GPUs in your application very easily:
   - Seamless integration into applications with existing parallelism (MPI, OpenMP)
   - Fairly efficient GPU-enabled libraries exist for many types of computational problems
   - Nowadays CUDA is needed only to develop something very specific and/or highly-tuned
3. GPU architecture and execution model are designed for massive parallelism
4. GPU efficiency is based on 4 factors:
   - Sufficient parallelism
   - Balanced use of registers and shared memory
   - Efficient DRAM utilization
   - Minimization of divergent branching

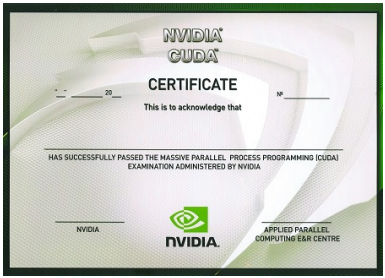
Conclusion

1. GPUs offer much higher performance of massively-parallel computations
2. You can start using GPUs in your application very easily:
   - Seamless integration into applications with existing parallelism (MPI, OpenMP)
   - Fairly efficient GPU-enabled libraries exist for many types of computational problems
   - Nowadays CUDA is needed only to develop something very specific and/or highly-tuned
3. GPU architecture and execution model are designed for massive parallelism
4. GPU efficiency is based on 4 factors:
   - Sufficient parallelism
   - Balanced use of registers and shared memory
   - Efficient DRAM utilization
   - Minimization of divergent branching

Conclusion

1. GPUs offer much higher performance of massively-parallel computations
2. You can start using GPUs in your application very easily:
   - Seamless integration into applications with existing parallelism (MPI, OpenMP)
   - Fairly efficient GPU-enabled libraries exist for many types of computational problems
   - Nowadays CUDA is needed only to develop something very specific and/or highly-tuned
3. GPU architecture and execution model are designed for massive parallelism
4. GPU efficiency is based on 4 factors:
   - Sufficient parallelism
   - Balanced use of registers and shared memory
   - Efficient DRAM utilization
   - Minimization of divergent branching

GPU / CUDA Training & Certification



Academic: Parallel & Distributed Computing LAB

- Institute of Computational Science, USI Lugano
- Prof. Dr. Olaf Schenk
- 1 semester (master course), assignments on GPU cluster
- GPU, profiling, optimization, scientific applications case studies
- Contact: dmitry.mikushin@usi.ch

Commercial: Advanced GPU computing course

- Applied Parallel Computing LLC & NVIDIA
- 3-5 days, on-site, hands-ons on GPU cluster
- GPU, profiling, optimization, customer-oriented hands-ons
- Register: http://www.nvidia.de/object/gpu-computing-workshop-registration-form.html

Certificate: granted upon successful pass of comprehensive practical tests