Mitglied der Helmholtz-Gemeinschaft

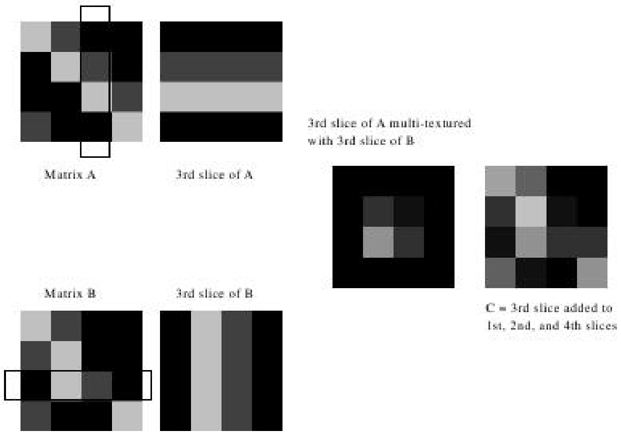
JÜLICH FORSCHUNGSZENTRUM

GPGPU

April 27, 2016

Jan H. Meinke

GPGPU



1. Kenneth E. Hoff, I.I.I., Keyser, J., Lin, M., Manocha, D. & Culver, T. Fast computation of generalized Voronoi diagrams using graphics hardware. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* 277-286 (1999).doi: 10.1145/311535.311567

2. Larsen, E.S. & McAllister, D. Fast matrix multiplies using graphics hardware. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing* (CDROM) 55 (2001).

3. Bolz, J., Farmer, I., Grinspun, E. & Schröder, P. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM SIGGRAPH 2003* Papers 924 (2003).

History of GPGPU Computing

NVIDIA's GeForce 3
w/ programmable shader
(2001)

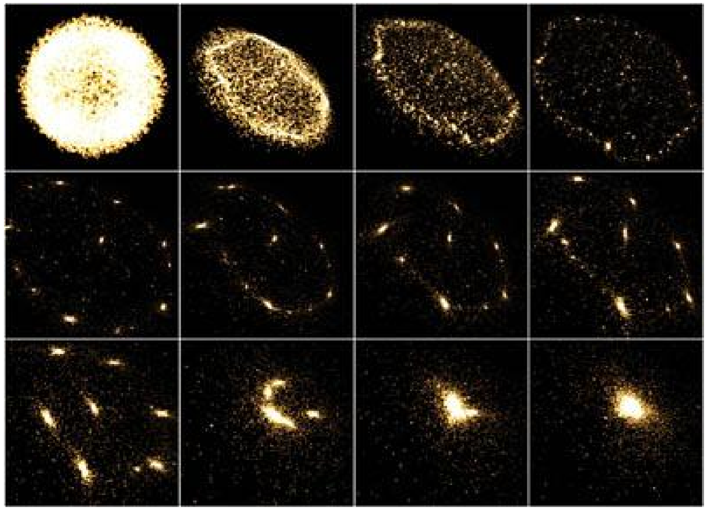
ATI Radeon 9700
w/ DirectX 9
(2003)

JÜLICH
FORSCHUNGSZENTRUM

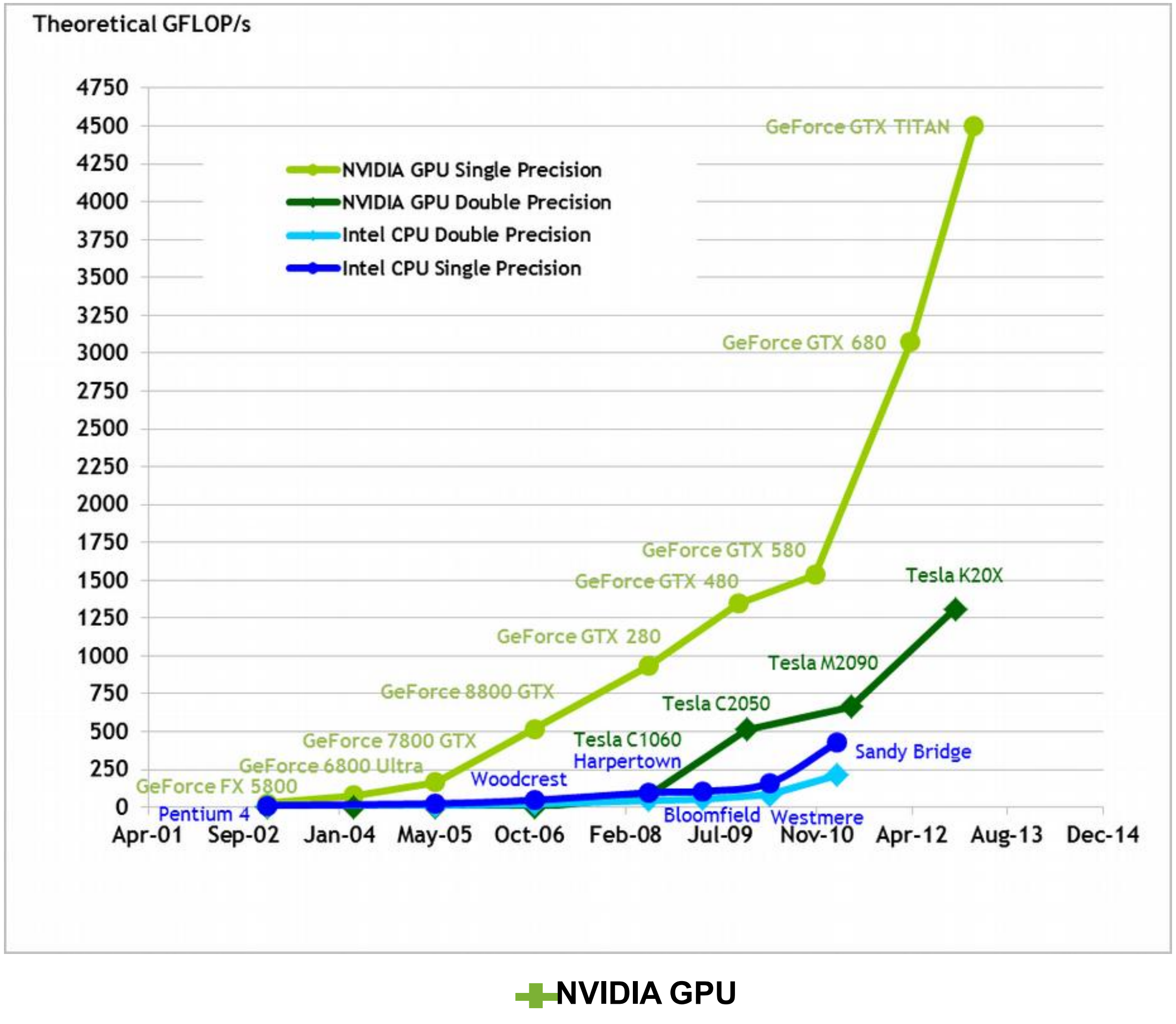
10 Years of CUDA

NVIDIA's GeForce 8 (Tesla)
(2006)

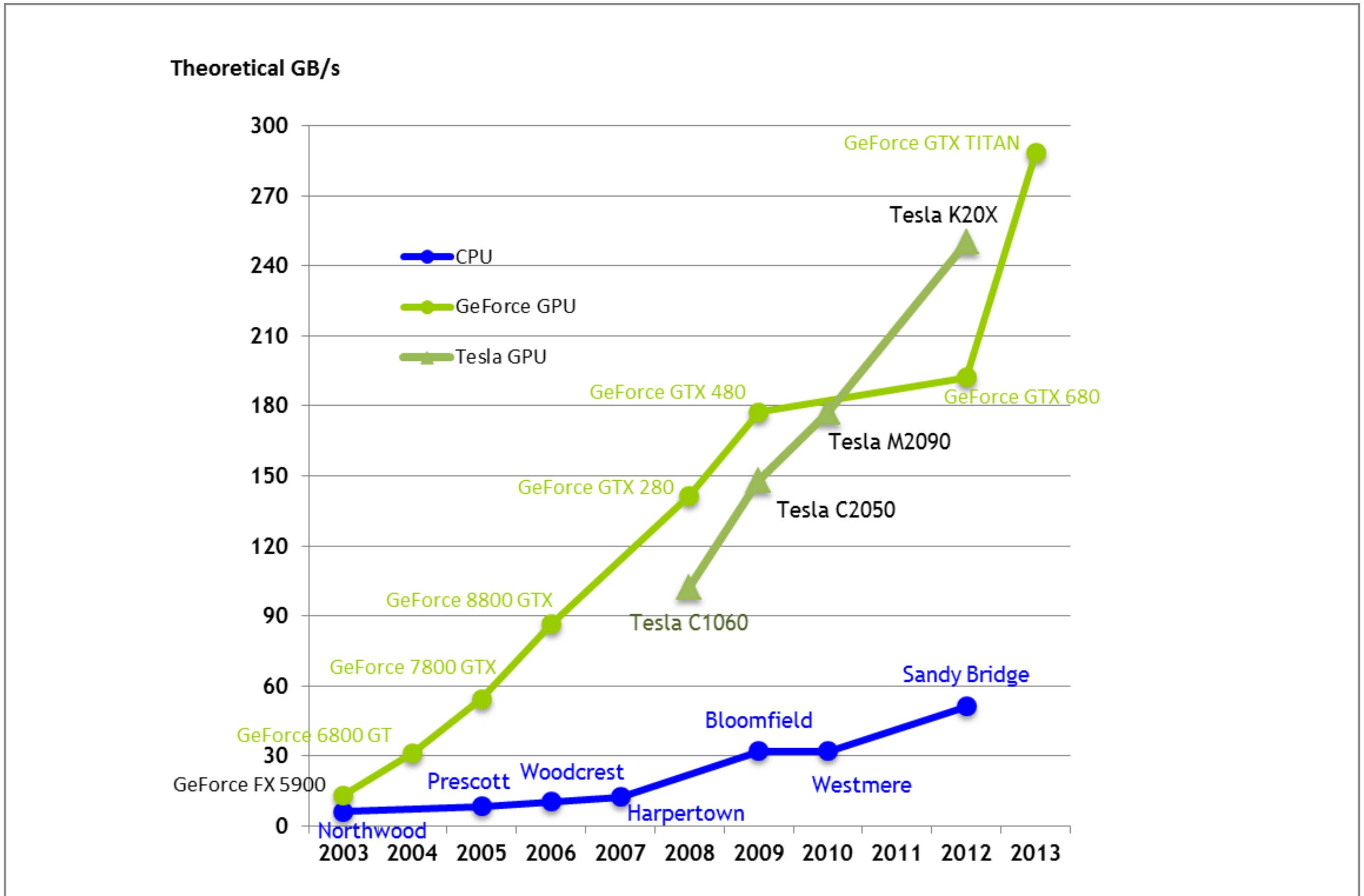
CUDA



The Performance Gap Widens Further



The Performance Gap Widens Further



NVIDIA GPU

X86 CPU

1.8 (CPU) + 0.44 (GPU) Petaflop/s peak

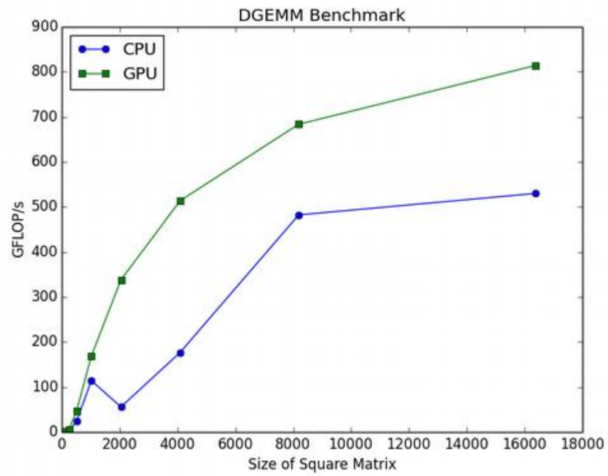
45216 CPU cores (1872 nodes)

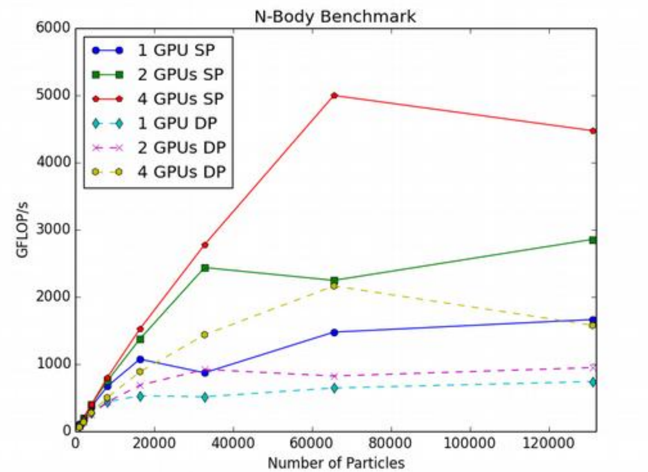
75 nodes with 2 NVIDIA K80 cards (4 GPUs)

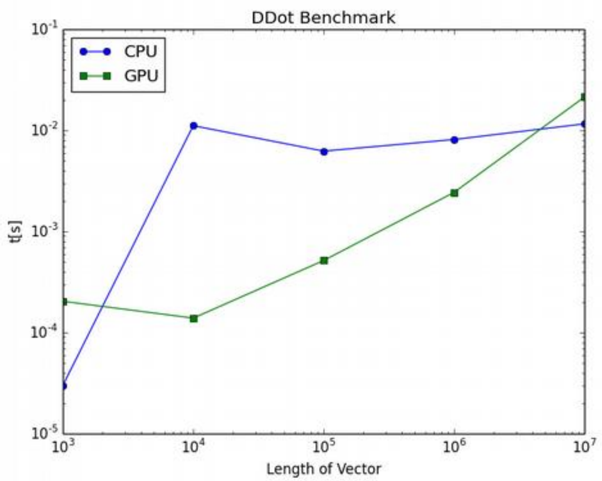
12 visualization nodes with 2 NVIDIA K40 cards

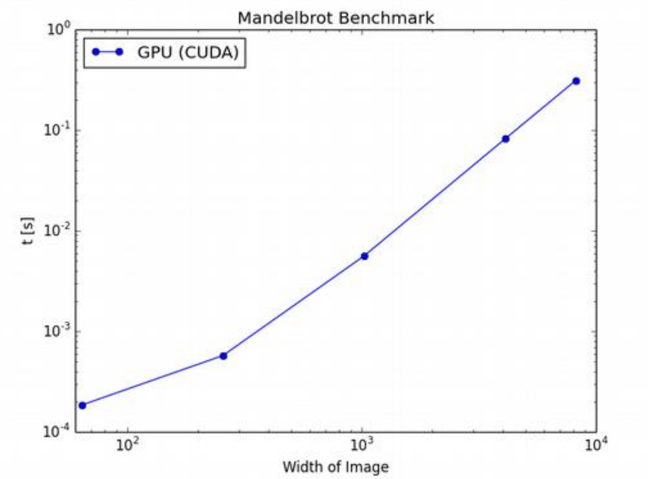
Mellanox EDR InfiniBand

Getting a Feeling for GPU Performance









Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

Programming GPUs

April 27, 2016

Don't!

Applications

Molecular Dynamics

Amber
NAMD
LAMMPS
Gromacs
Desmond
HOOMD-Blue
...

CFD

S++
S3D
ANSYS Fluent

Image Processing

ArrayFire

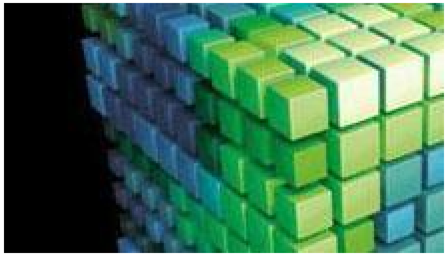
Mathematics

MathWorks
Wolfram Mathematica

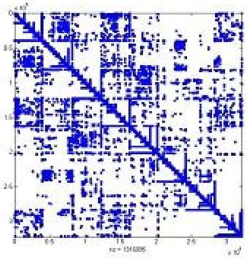
http://www.nvidia.com/object/gpu-applications.html

GPU-Accelerated Libraries

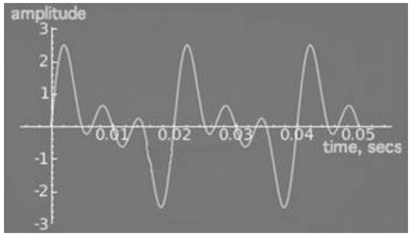
cuBLAS



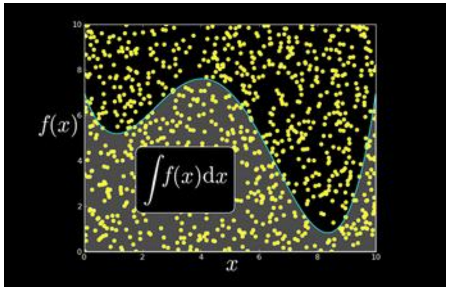
cuSPARSE



cuFFT



cuRAND



Thrust

CUSP

https://developer.nvidia.com/gpu-accelerated-libraries

Using cuBLAS

- Initialize
- Allocate memory on the GPU
- Copy data to GPU
- Call BLAS routine
- Copy results to Host
- Finalize

Calculates $res = \sum_{i=0}^{n} A_i B_i$

- status = cublasCreate(&handle)
- cudaMalloc((void**)&d_A, n * sizeof(d_A[0]))
- status = cublasSetVector(n, sizeof(A[0]), A, 1, d_A, 1);
- status = cublasDdot(handle, n, d_A, 1, d_B, 1, &res)
- status = cublasDestroy(handle);

Exercise

CudaBasics/exercises/tasks/cublas

So, you think you want to write your own GPU code...

Parallel Scaling Primer

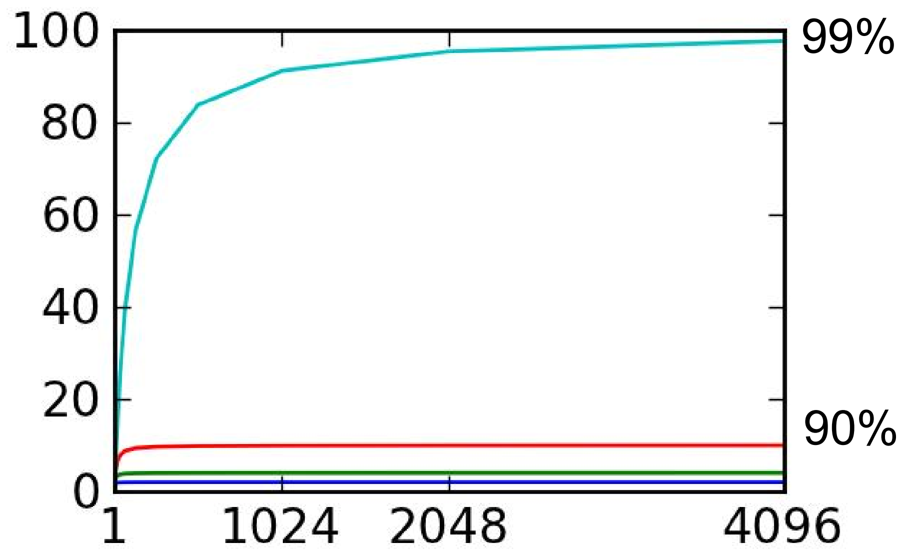
- What is the maximum speedup?

$$t = t_s + t_p$$

$$t(n) = t_s + t_p / n$$

$$s = t / t(n)$$

$$= \frac{t_s + t_p}{t_s + t_p / n}$$



```
! Simple matmul example

      module mymm
      contains
      subroutine mm1(a, b, c, m)
        real, dimension(:,:) :: a, b, c
        do j = 1, m
           do i = 1, m
                a(i, j) = 0.0
           enddo
           do k = 1, m
                do i = 1, m
                     a(i, j) = a(i, j) + b(i, k) * c(k, j)
                enddo
           enddo
        enddo
      end subroutine
      end module
~
~
~
~
~
~
"mm1.f90" 18L, 419C                                        5,7          All
```

judge :   zam1037 :

Things to consider

- Is my program computationally intensive?
- How much data needs to be transferred in and out?
- Is the gain worth the pain?

OpenACC

- Pragma/directive based
  #pragma acc kernels in C
  !acc kernels … !acc end kernels in Fortran
- Some additional control statement
  - copyin/copyout
  - vector
  - acc_init
  - acc data region
  - ...

CUDA 7.5: Thrust 1.8.1

- Template library similar to STL.
- Containers
- Algorithms
- Thrust 1.3 for CUDA 3.2

File Edit View Bookmarks Settings Help

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <algorithm>
#include <cstdlib>
static const int WORK_SIZE = 1000000;

double random_double(){
        return 1.0 / RAND_MAX * rand();
}

int main()
{
        thrust::host_vector<double> data(WORK_SIZE);
        std::generate(data.begin(), data.end(), random_double);

        thrust::device_vector<double> data_dev = data;

        double sum_cpu = thrust::reduce(data.begin(), data.end(), 0.0);
        double sum_gpu = thrust::reduce(data_dev.begin(), data_dev.end(), 0.
0);

                                                    1,1           Top
```

judge :

Exercise

CudaBasics/exercises/tasks/thrust

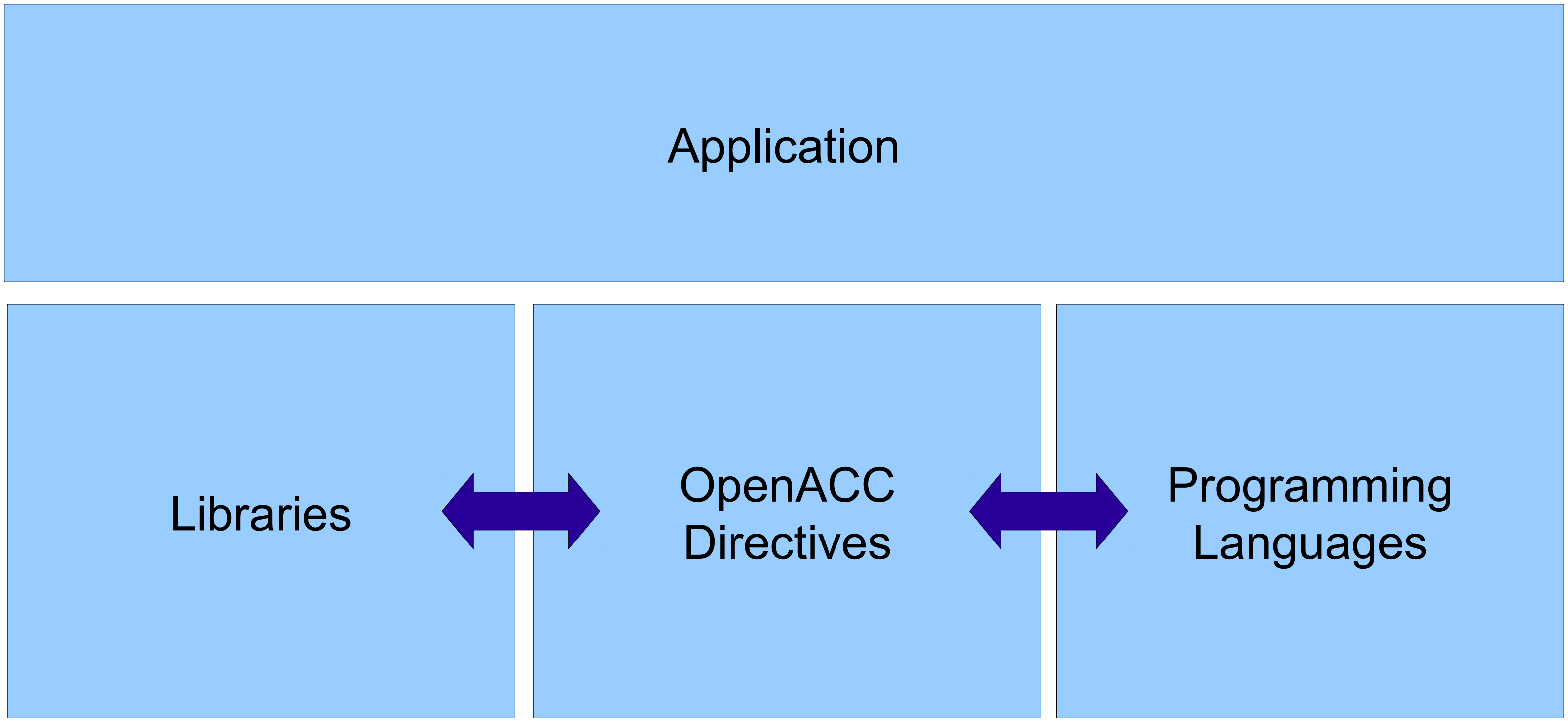
CUDA C++ Alternatives

- OpenACC (see course in the fall)
- Thrust
- PyCUDA/PyOpenCL
- CUDA Fortran
- CUDA-Python
- OpenCL (see course in the spring)

```
import numpy as np
from numbapro import jit
from numbapro import cuda

@jit(argtypes=[f4[:,:], f4[:,:], f4[:,:]], target='gpu')
def cu_square_matrix_mul(A, B, C):
    sA = cuda.shared.array(shape=(tpb, tpb),
                dtype=f4)
    …
    tx = cuda.threadIdx.x
    …
    x = tx + bx * bw
    y = ty + by * bh
    acc = 0.
    for i in range(bpg):
        if x < n and y < n:
            sA[ty, tx] = A[y, tx + i * tpb]
            sB[ty, tx] = B[ty + i * tpb, x]
        cuda.syncthreads()
        …
```

3 Ways to Accelerate Applications



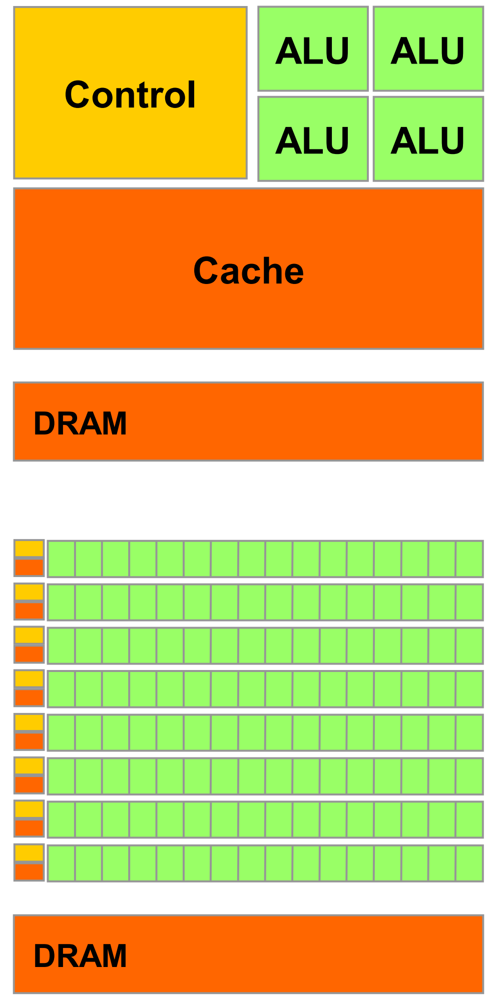
“Drop-in” Acceleration

Easily Accelerate Applications

Maximum Flexibilty

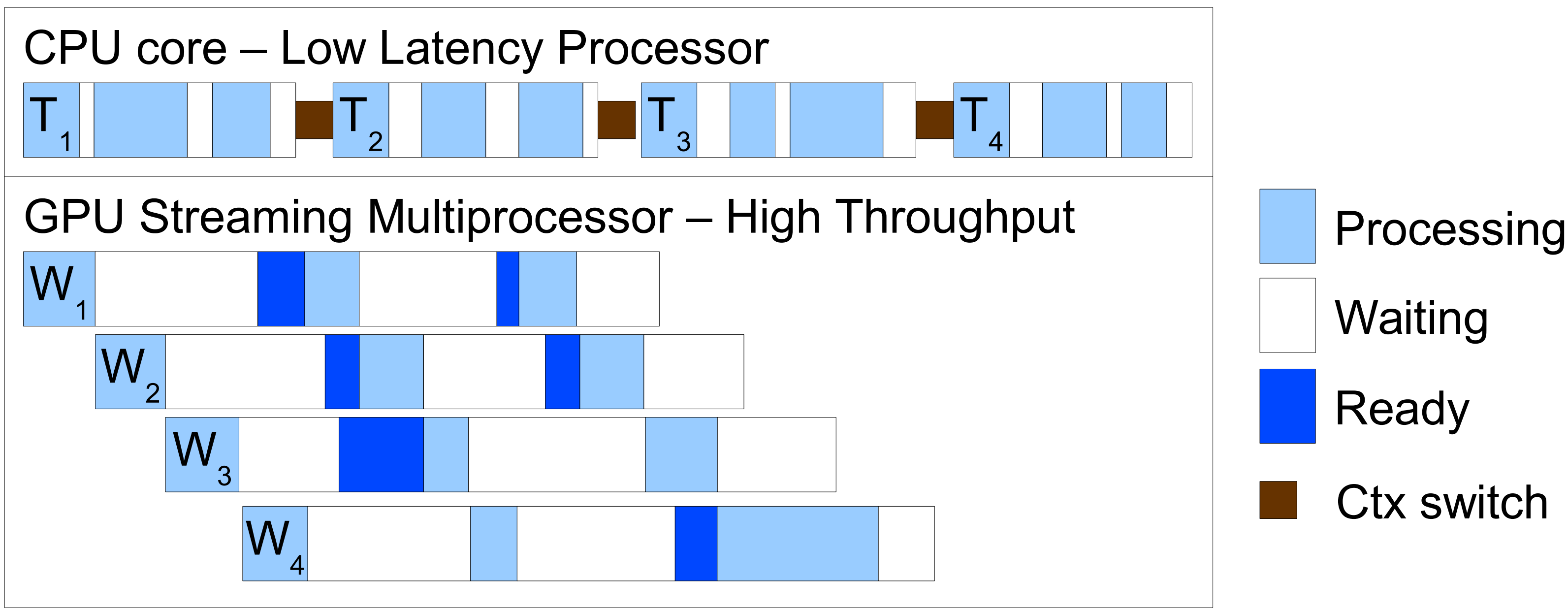
Low Latency or High Throughput?

- CPU
  - Optimized for low-latency access to cached data sets
  - Control logic for out-of-order and speculative execution
- GPU
  - Optimized for data-parallel, throughput computation
  - Architecture tolerant of memory latency
  - More transistors dedicated to computation

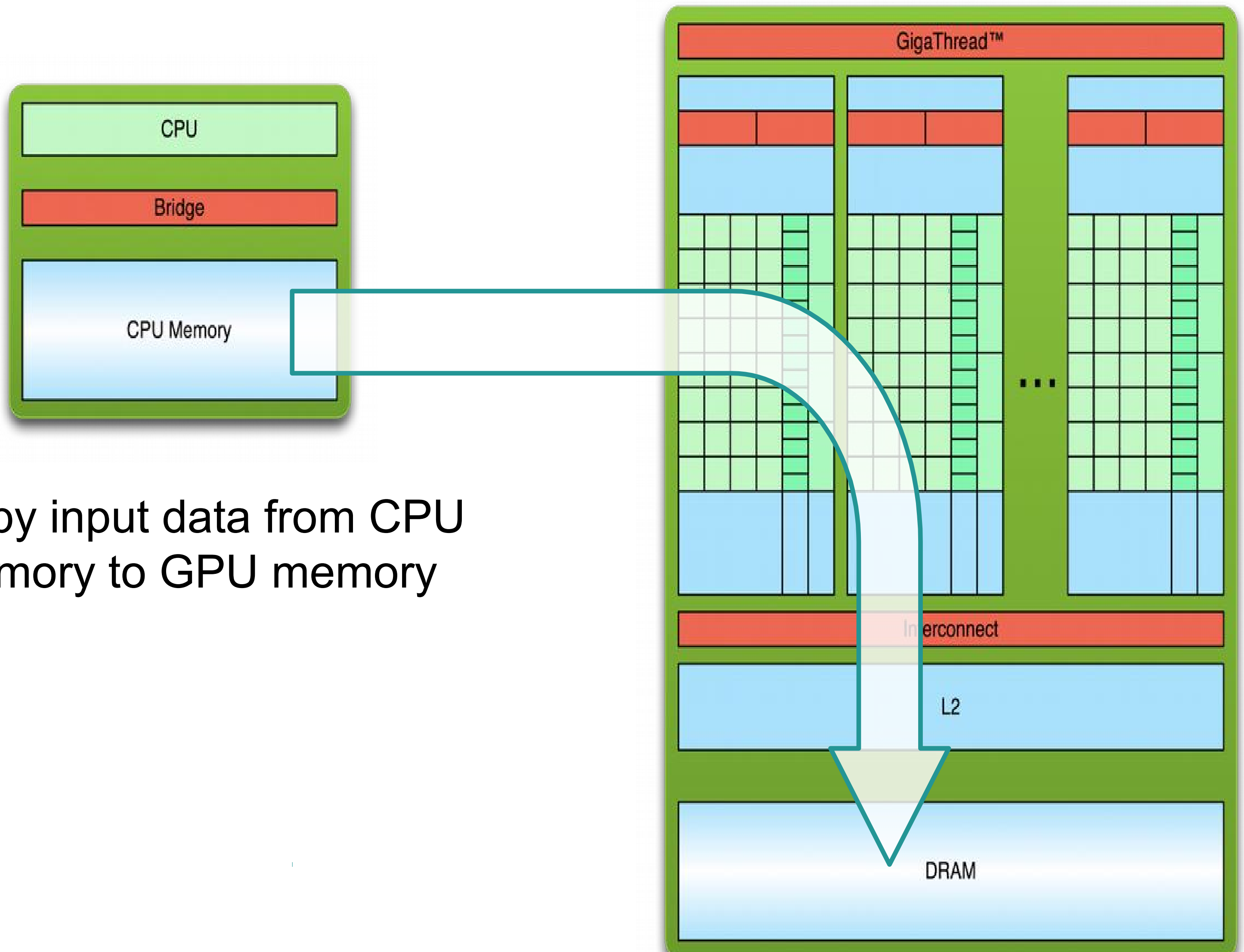


Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps (bundle of threads)

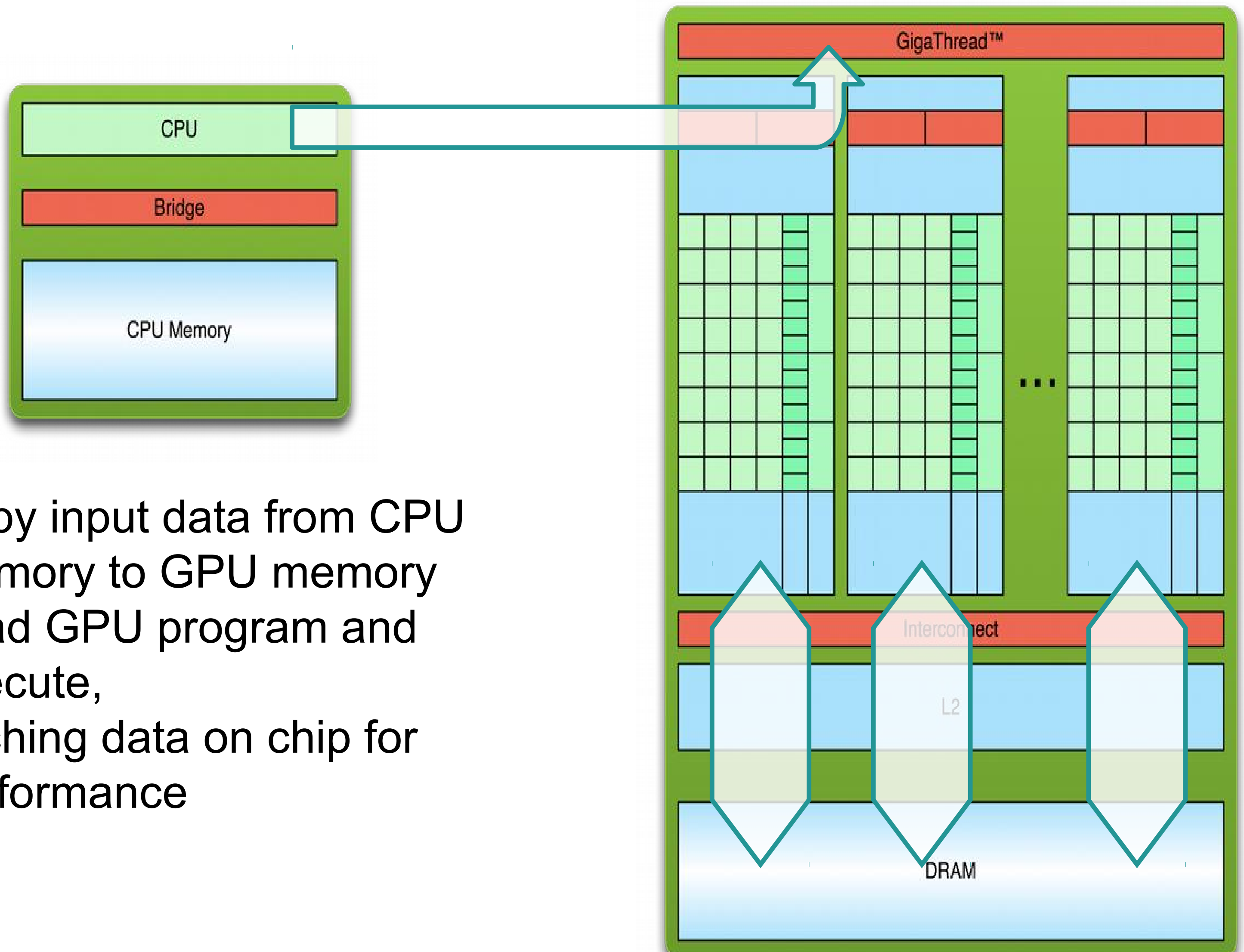


Processing Flow



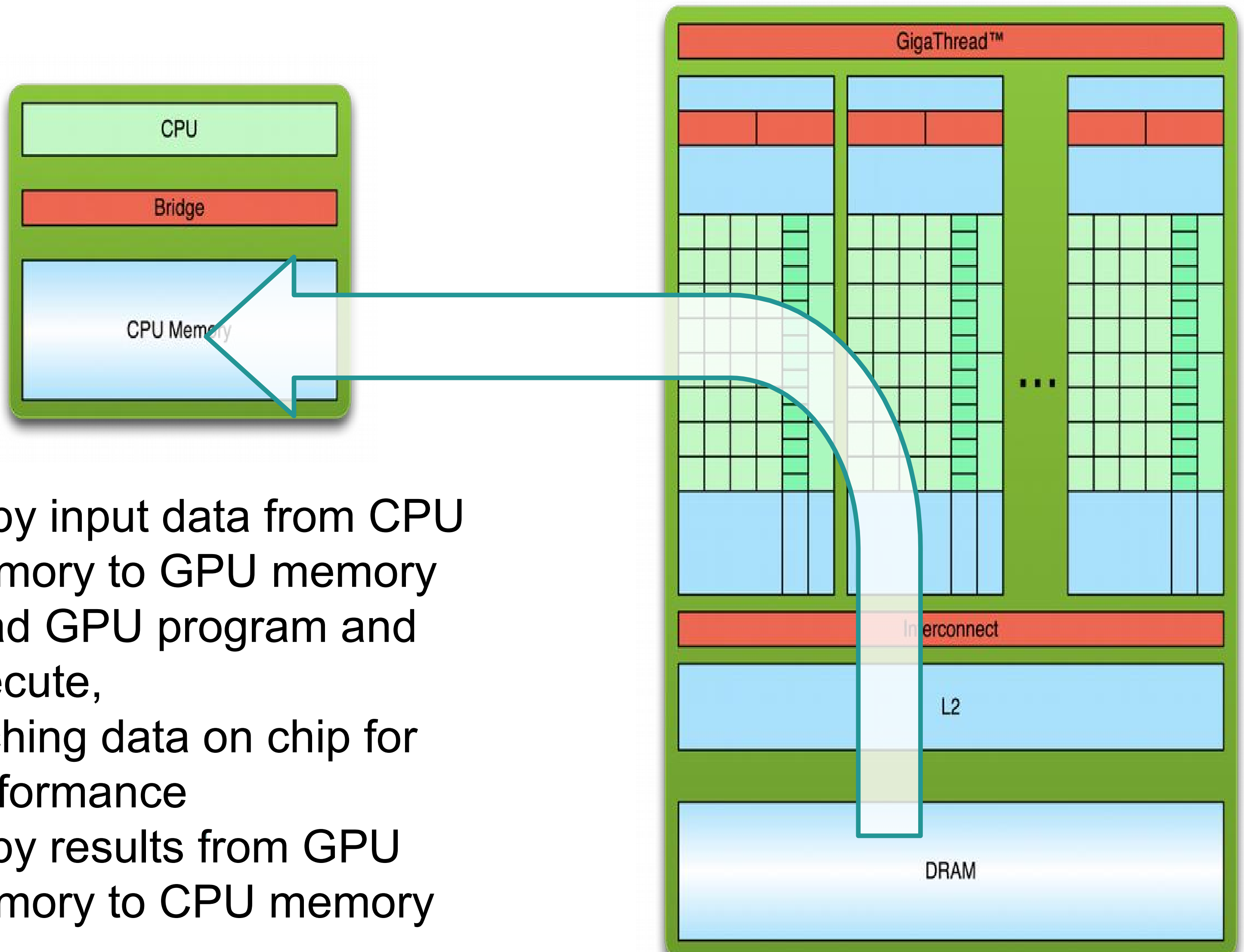
Copy input data from CPU memory to GPU memory

Processing Flow



Copy input data from CPU memory to GPU memory
Load GPU program and execute,
caching data on chip for performance

Processing Flow



Copy input data from CPU memory to GPU memory
Load GPU program and execute,
caching data on chip for performance
Copy results from GPU memory to CPU memory

JÜLICH FORSCHUNGSZENTRUM

Programming Model

April 27, 2016

CUDA Kernels

- Parallel portion of application: execute as a kernel
  - *Entire GPU executes kernel, many threads*

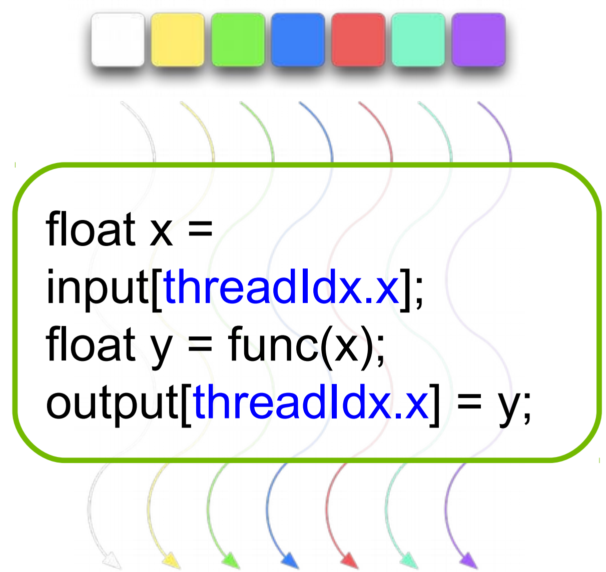
- CUDA threads:
  - *Lightweight*
  - *Fast switching*
  - *1000s execute simultaneously*

CUDA Kernels: Parallel Threads

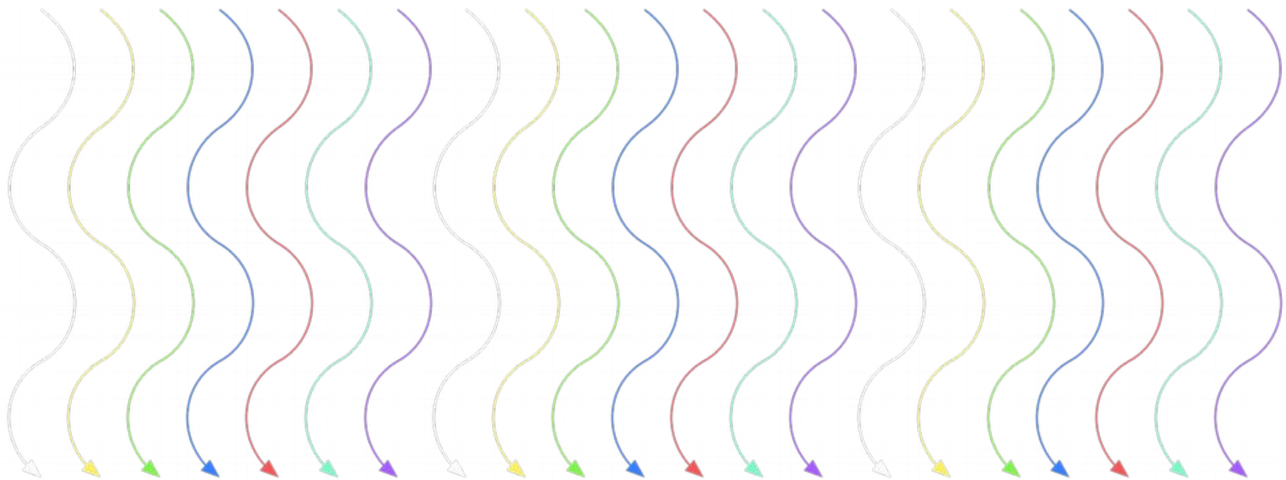
- A kernel is a function executed on the GPU as an array of threads in parallel

- All threads execute the same code, but can take different paths

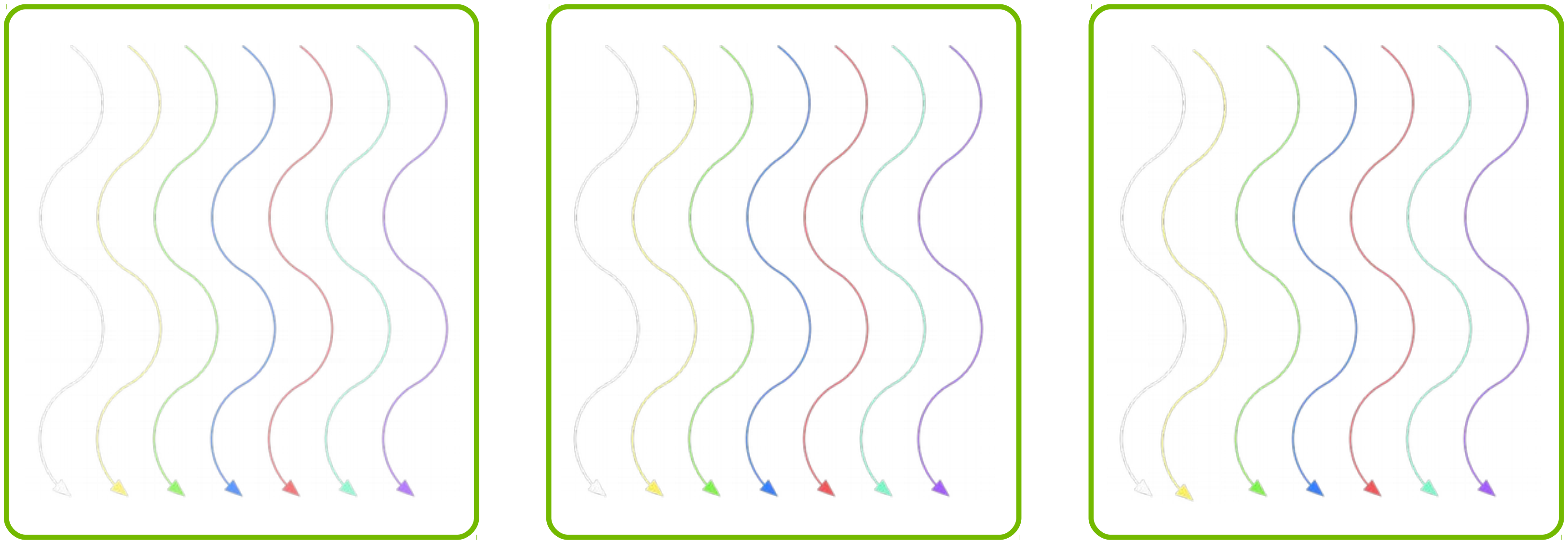
- Each thread has an ID
  - *Select input/output data*
  - *Control decisions*



CUDA Kernels: Subdivide into Blocks

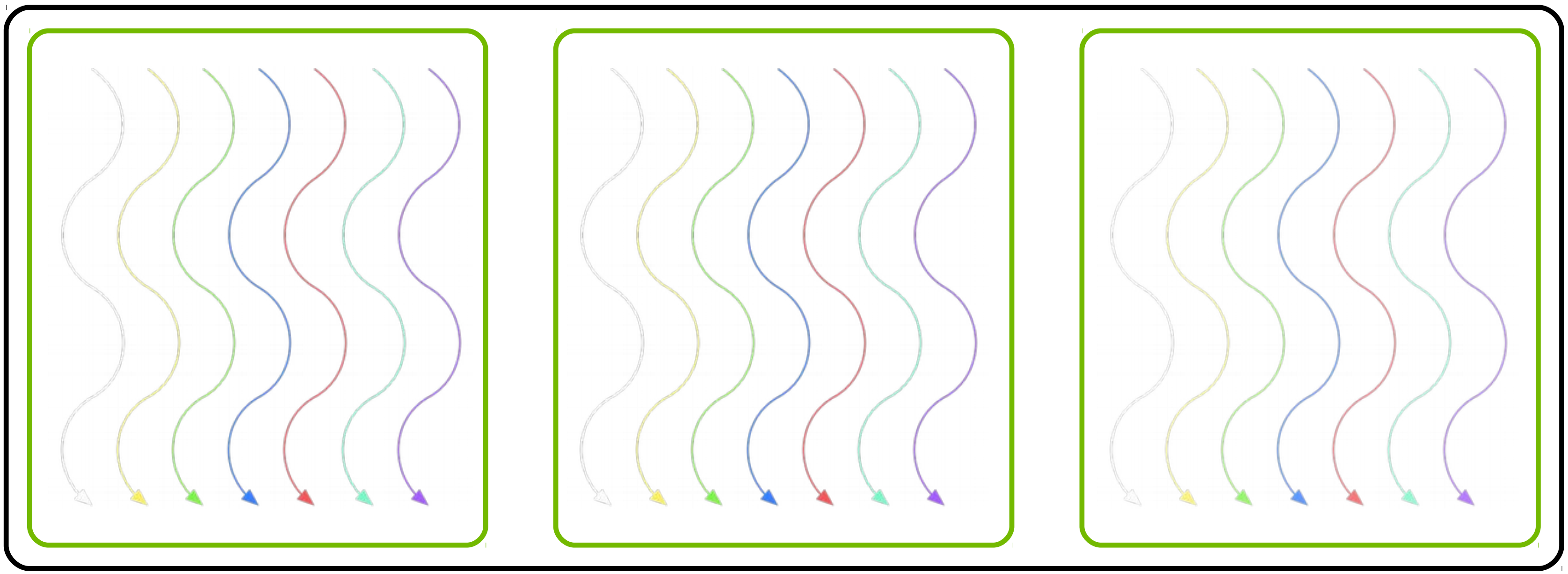


CUDA Kernels: Subdivide into Blocks



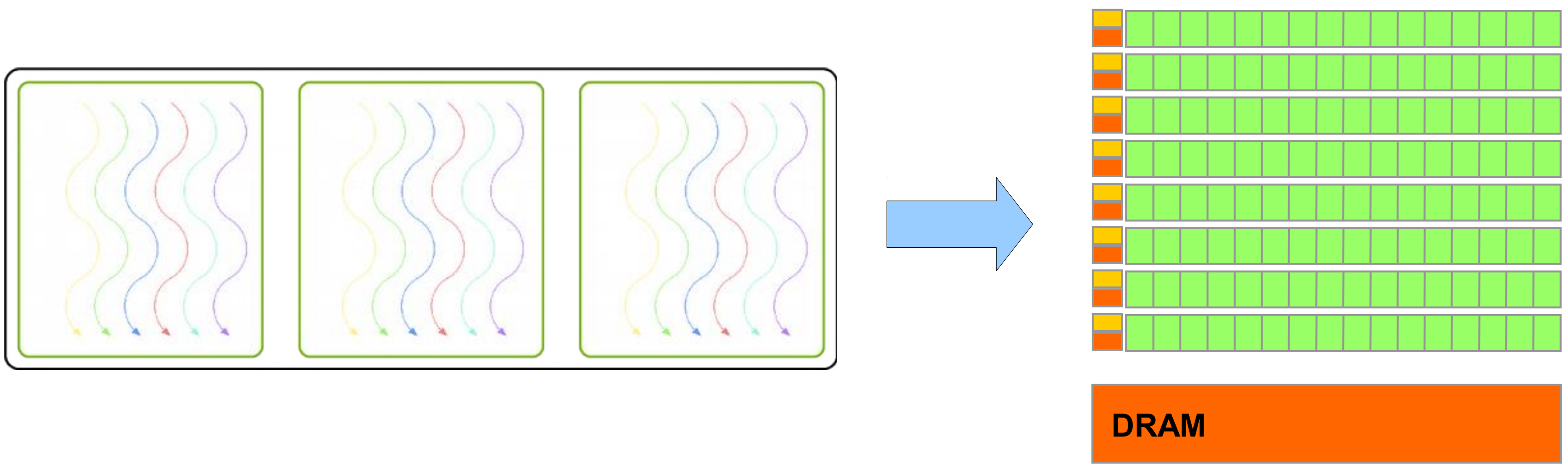
- Threads are grouped into blocks

CUDA Kernels: Subdivide into Blocks



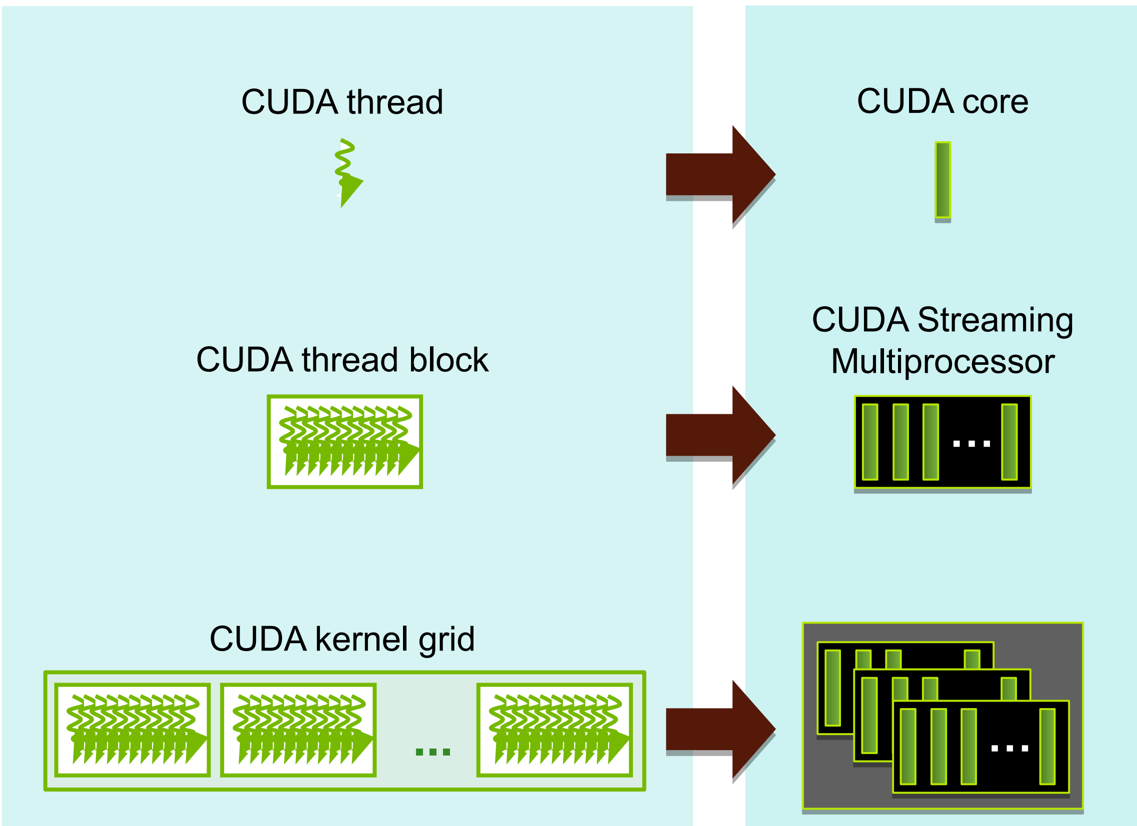
- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks



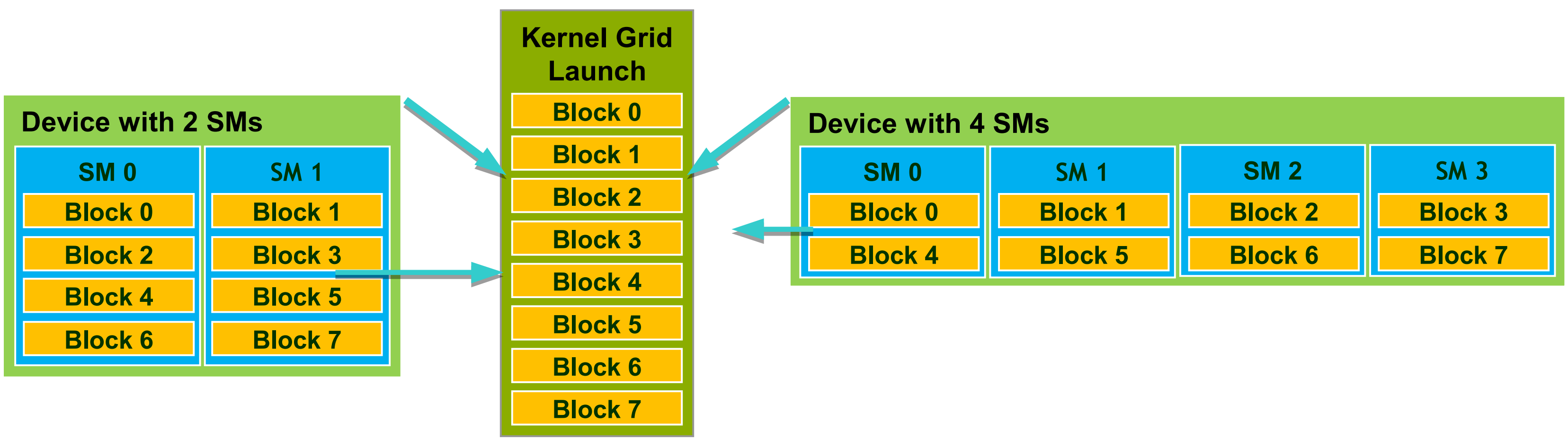
- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



Thread blocks allow scalability

- Block can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - *A kernel scales across any number of SMs*



Scale Kernel

```
void scale(float alpha,
                   float* B,
                   float* C,
                   int m)
{
    int i = 0;
    for ( i=0; i<m; ++i)
        C[i] = alpha * A[i];
}
```

```
__global__ void scale(float alpha,
                      float* B,
                      float* C,
                      int m)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    if ( i < m)
        C[i] = alpha * A[i];
}
```

Getting data in and out with Unified Memory

- GPU has separate memory, but transfers can be managed by runtime
- Allocate memory with cudaMallocManaged
- Free memory

Define dimensions of thread block

```
dim3 blockDim(size_t blockDimX, size_t blockDimY,
              size_t blockDimZ)
```

On JURECA (Tesla K80):

- Max. dim. of a block: 1024 x 1024 x 64
- Max. number of threads per block: 1024

Example:

```
// Create 3D thread block with 512 threads
dim3 blockDim(16, 16, 2);
```

Define dimensions of grid

```
dim3 gridDim(size_t blockDimX, size_t blockDimY,
             size_t blockDimZ)
```

On JURECA (Tesla K80):

- Max. dim. of a grid: 2147483647 x 65535 x 65535

Example:

```
// Dimension of problem: nx x ny = 1000 x 1000
dim3 blockDim(16, 16) // Don't need to write z = 1
int gx = (nx % blockDim.x==0) ? nx / blockDim.x : nx / blockDim.x + 1
int gy = (ny % blockDim.y==0) ? ny / blockDim.y : ny / blockDim.y + 1
dim3 gridDim(gx, gy);
```

Watch out!

Call the kernel

```
kernel<<<int gridDim, int blockDim>>>([arg]*)
```

Call returns immediately! → Kernel executes asynchronously

Example:

```
scale<<<m/blockDim, blockDim>>>(alpha, a_gpu, c_gpu, m)
```

Calling the kernel

- Define dimensions of thread block

```
dim3 blockDim(size_t blockDimX, size_t blockDimY,
              size_t blockDimZ)
```

- Define dimensions of grid

```
dim3 gridDim(size_t gridDimX, size_t gridDimY,
             size_t gridDimZ)
```

- Call the kernel

```
kernel<<<dim3 gridDim, dim3 blockDim>>>([arg]*)
```

Free device memory

```
cudaFree(void* pointer)
```

Example:

```
// Free the memory allocated by a_gpu on the device
cudaFree(a_gpu);
```

Exercise

CudaBasics/exercises/tasks/scale_vector

Compile with nvcc -o scale_vector scale_vector.cu

Getting data in and out

- GPU has separate memory
- Allocate memory on device
- Transfer data from host to device
- Transfer data from device to host
- Free device memory

Allocate memory on device

```
cudaMalloc(void** pointer, size_t nbytes)
```

Example:

```
// Allocate a vector of 2048 floats on device
float * a_gpu;
int n = 2048;
cudaMalloc((void**) &a_gpu, n * sizeof(float));
```

Cast to void**

Get size of a float

Address of pointer

Copy from host to device

```
cudaMemcpy(void* dst, void* src, size_t nbytes,
           enum cudaMemcpyKind  dir)
```

Example:

```
// Copy vector of  floats a of length n=2048 to a_gpu on device
cudaMemcpy(a_gpu, a, n * sizeof(float),
                        cudaMemcpyHostToDevice);
```

Copy from device to host

```
cudaMemcpy(void* dst, void* src, size_t nbytes,
                enum cudaMemcpyKind  dir)
```

Example:

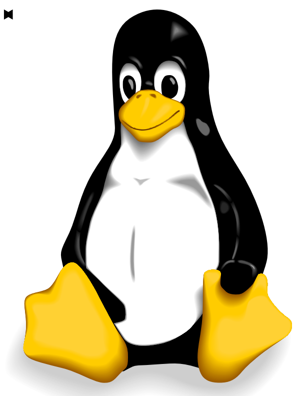
```
// Copy vector of floats a_gpu of length n=2048 to a on host
cudaMemcpy(a, a_gpu, n * sizeof(float),
                                        cudaMemcpyDeviceToHost);
```

Note the order

Changed flag

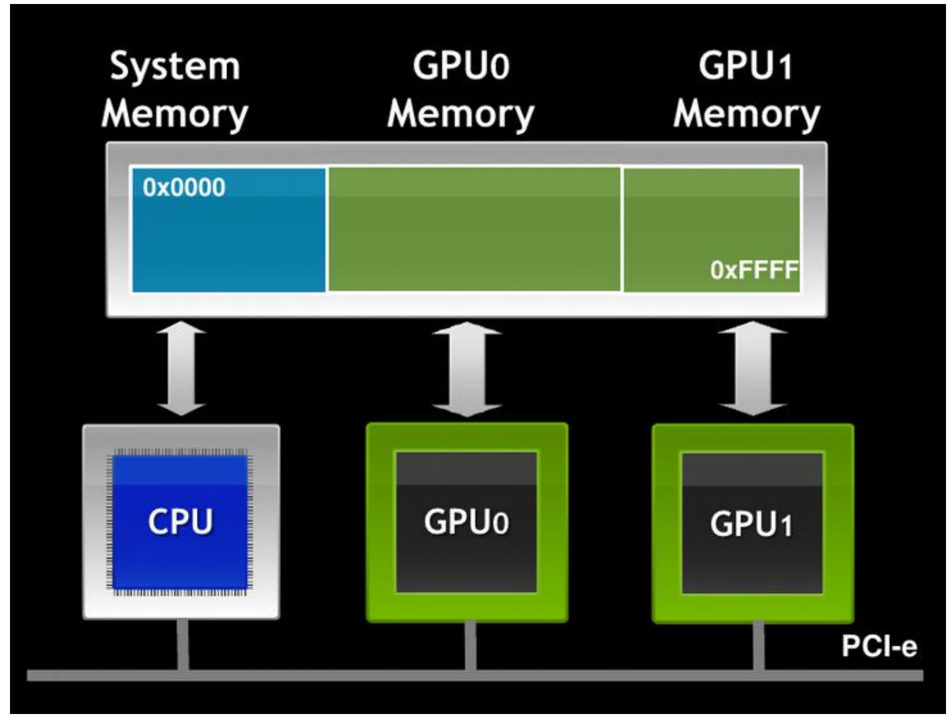
Unified Virtual Address Space (UVA)

Windows



64bit

2.0



cudaMalloc*(...)
cudaHostAlloc(...)
} return UVA pointers

cudaMemcpy*(..., cudaMemcpyDefault)

Getting data in and out

- Allocate memory on device

```
cudaMalloc(void** pointer, size_t nbytes)
```

- Transfer data between host and device

```
cudaMemcpy(void* dst, void* src, size_t nbytes,
           enum cudaMemcpyKind  dir)
dir = cudaMemcpyHostToDevice
dir = cudaMemcpyDeviceToHost
```

- Free device memory

```
cudaFree(void* pointer)
```

Exercise Scale Vector

Allocate memory on device

```
cudaMalloc(void** pointer, size_t nbytes)
```

Transfer data between host and device

```
cudaMemcpy(void* dst, void* src,
           size_t nbytes,
           enum cudaMemcpyKind dir)
 dir = cudaMemcpyHostToDevice
 dir = cudaMemcpyDeviceToHost
```

- Free device memory

```
cudaFree(void* pointer)
```

Define dimensions of thread block

```
dim3 blockDim(size_t blockDimX,
              size_t blockDimY,
              size_t blockDimZ)
```

Define dimensions of grid

```
dim3 gridDim(size_t gridDimX, size_t
             gridDimY,
             size_t gridDimZ)
```

Call the kernel

```
kernel<<<dim3 gridDim,
         dim3 blockDim>>>([arg]*)
```

Exercise

CudaBasics/exercises/tasks/jacobi_w_explicit_transfer

Compile with make jacobi.