THE UNIVERSITY *of York*

High Performance Computing - Introduction to GPU Programming

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- What are GPUs?
- GPU Architecture
- Memory
- Programming in CUDA
- The Future

What is a GPU?

- A GPU is a massive vector processor
  - Hundreds of processing units
  - Large memory bandwidth
  - Processing units can “collaborate”
- Some problems can be solved more efficiently on vector processors (remember Crays).

- GPUs have some big advantages.
  - Cheap and very fast for vector problems
  - Computation is asynchronous with the CPU
  - Hardware “tricks” such as texture filtering (zero overhead!)
- Other accelerators (e.g. Xeon Phi) available

When is GPU Programming useful?

- “The GPU devotes more transistors to data processing” (nVidia C Programming Guide)
- Good at doing lots of numeric calculations simultaneously.
  - This is actually required for the GPU to be efficient
  - There must be many more numeric operations than memory operations to break even.
- Useful as a co-processor.
  - Can offload GPU efficient calculations while the CPU continues with the rest.
- Brute forcing!
  - Sometimes a brute force method is more efficient with many processors than an elegant solution on just one

Floating Point Standard

- nVidia architectures 2.x onwards (i.e. 2010 Fermi onwards) are IEEE 754 compliant.

- Old nVidia architectures were mostly compliant. Generally exceptions were handled in a noncompliant way. Also some mathematics e.g. FMAD, division and sqrt were not standard.

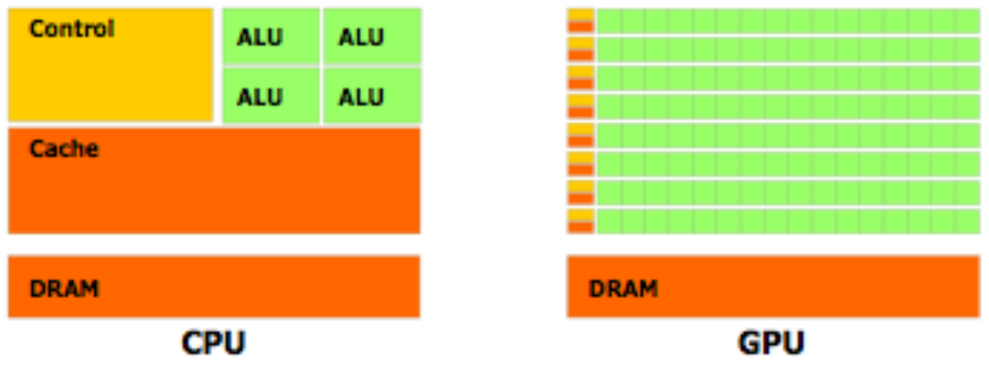
- Standard compliant intrinsic functions are available but at a large computational penalty (software implemented).

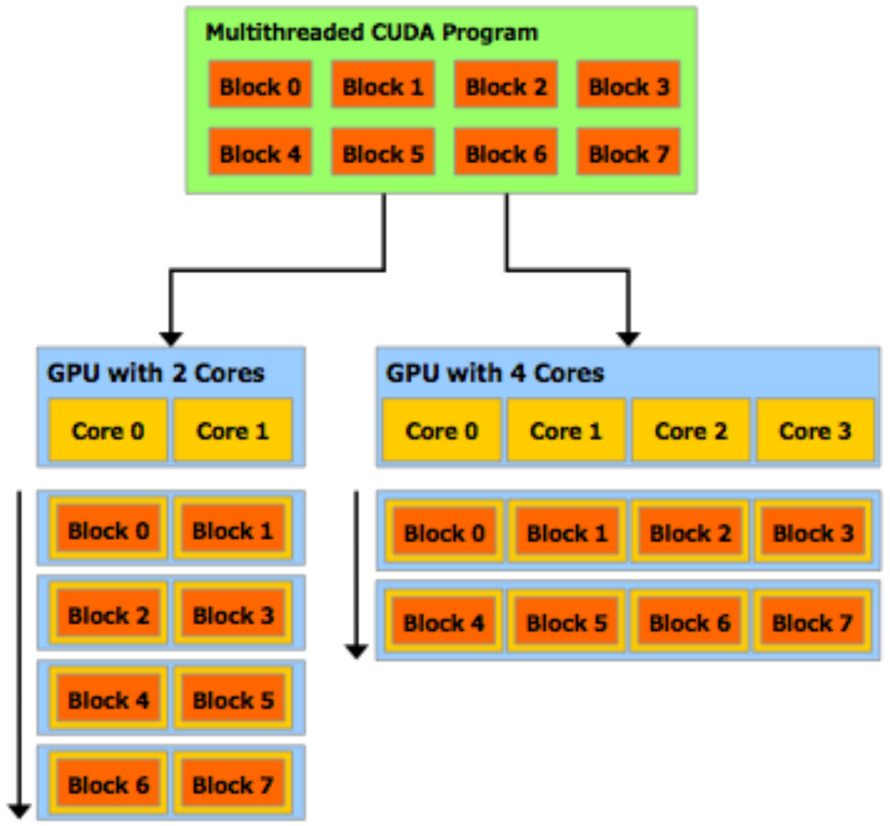
nVidia Architectures

- V1 = Tesla (2008) – introduced CUDA with performance of 0.5 GFLOP/watt,
- V2 = Fermi (2010) – 64-bit floating, 2 GFLOP/W
- V3 = Kepler (2012) - dynamic parallelism
- V5 = Maxwell (2014) – faster Kepler – 3.8 TFLOP DP, 24 GB GDRAM, 2 GPU, 500 GB/s, 300 W
- V6 = Pascal (2016) – unified memory, stacked DRAM, direct interconnect between GPU & RAM
- V7 = Volta (2018) – tensor cores with half-precision math for machine learning …
- More recent: Turing (2020), Ampere (2021), Ada Lovelace (announced 2022) – faster Volta

GPU Architecture

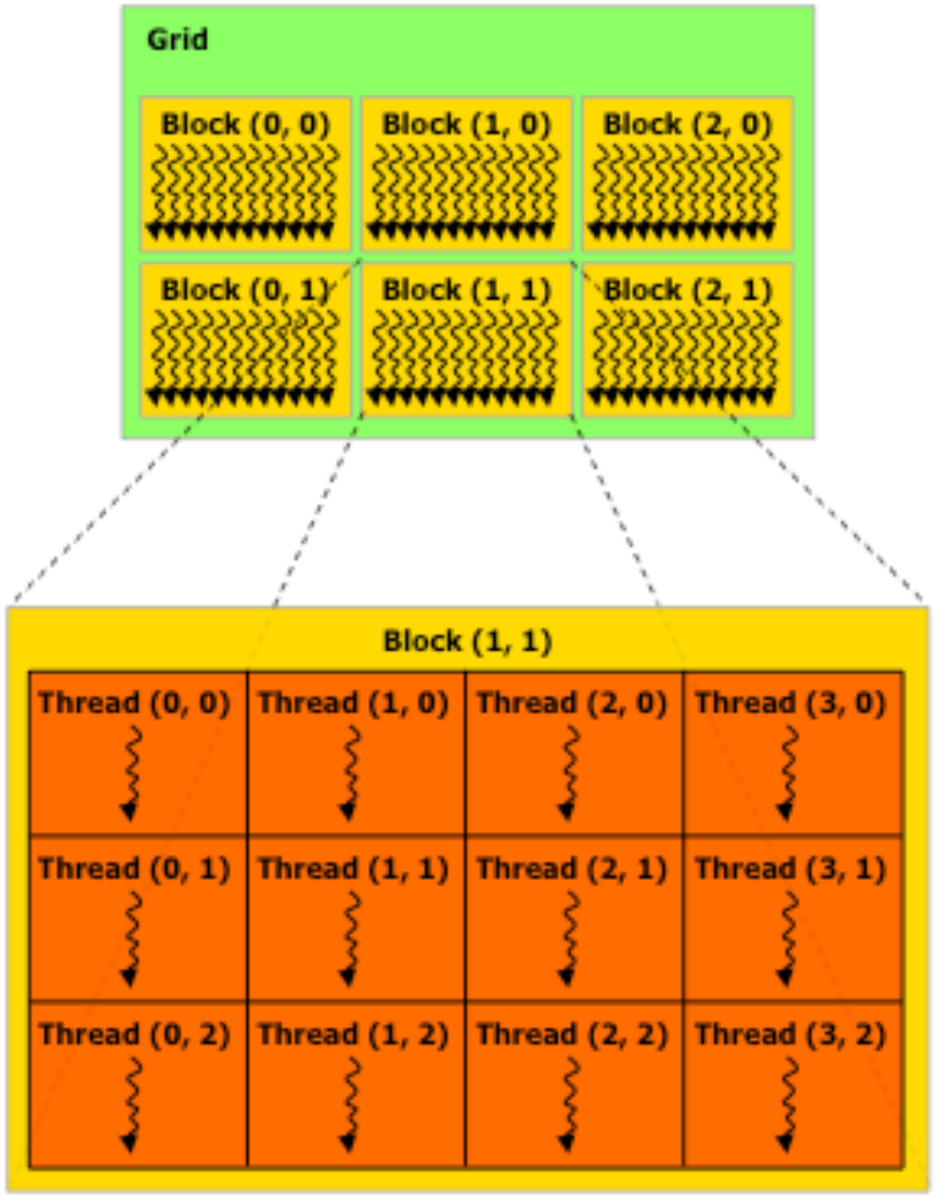
- Lots of arithmetic units sharing small caches.
- Execution blocks are scheduled across processing cores.
- Large memory bandwidth but slow memory access.
- Fermi onwards have full cache hierarchy.





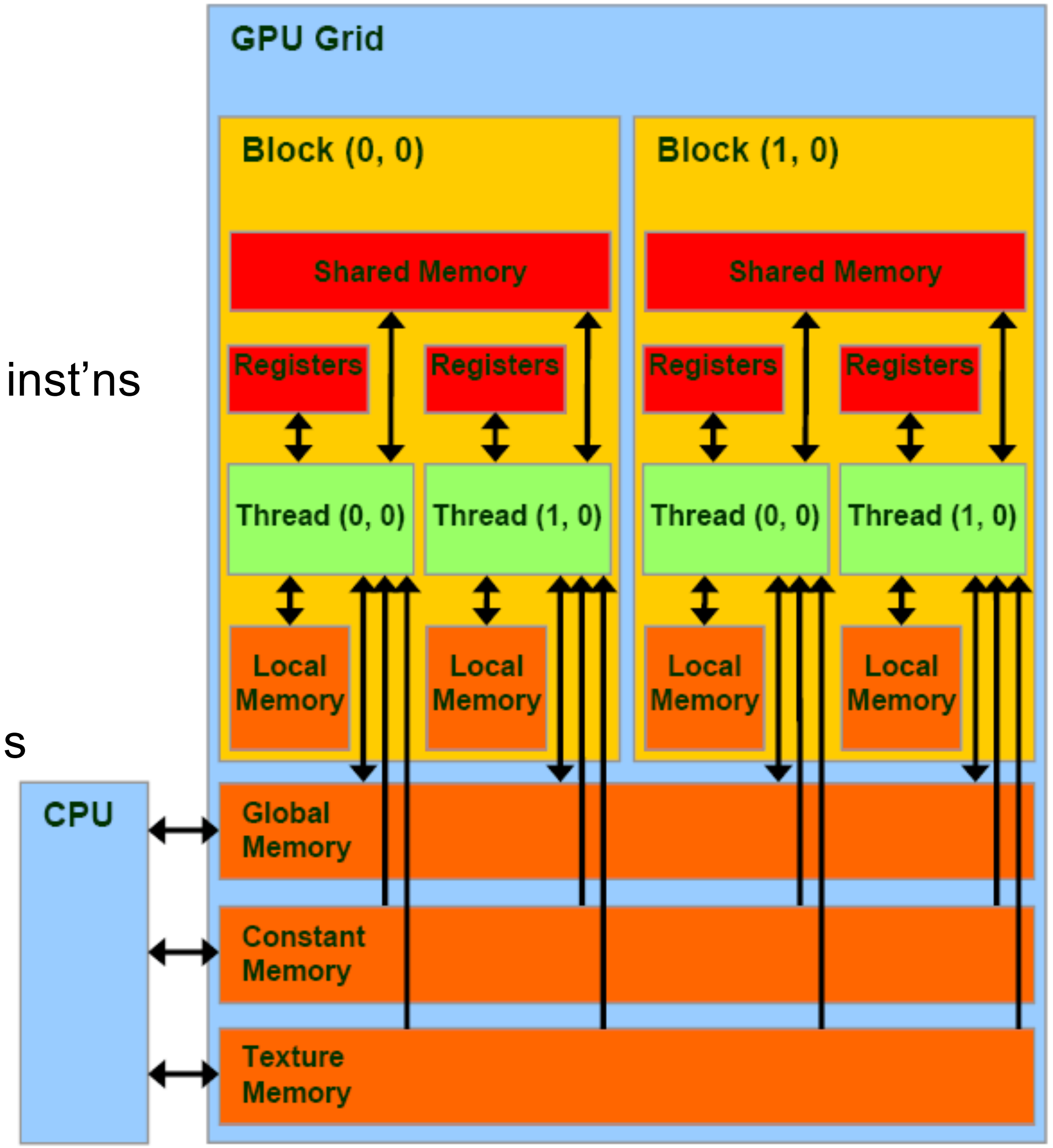
Threads

- Each thread runs one instance of a kernel.
- Threads are organised into blocks (which can be up to 3D), blocks are scheduled on and off the processors.
- Execution of a block on the processors is called a warp.
- Grids (which can be 2D) contain many blocks which are executed on the device.



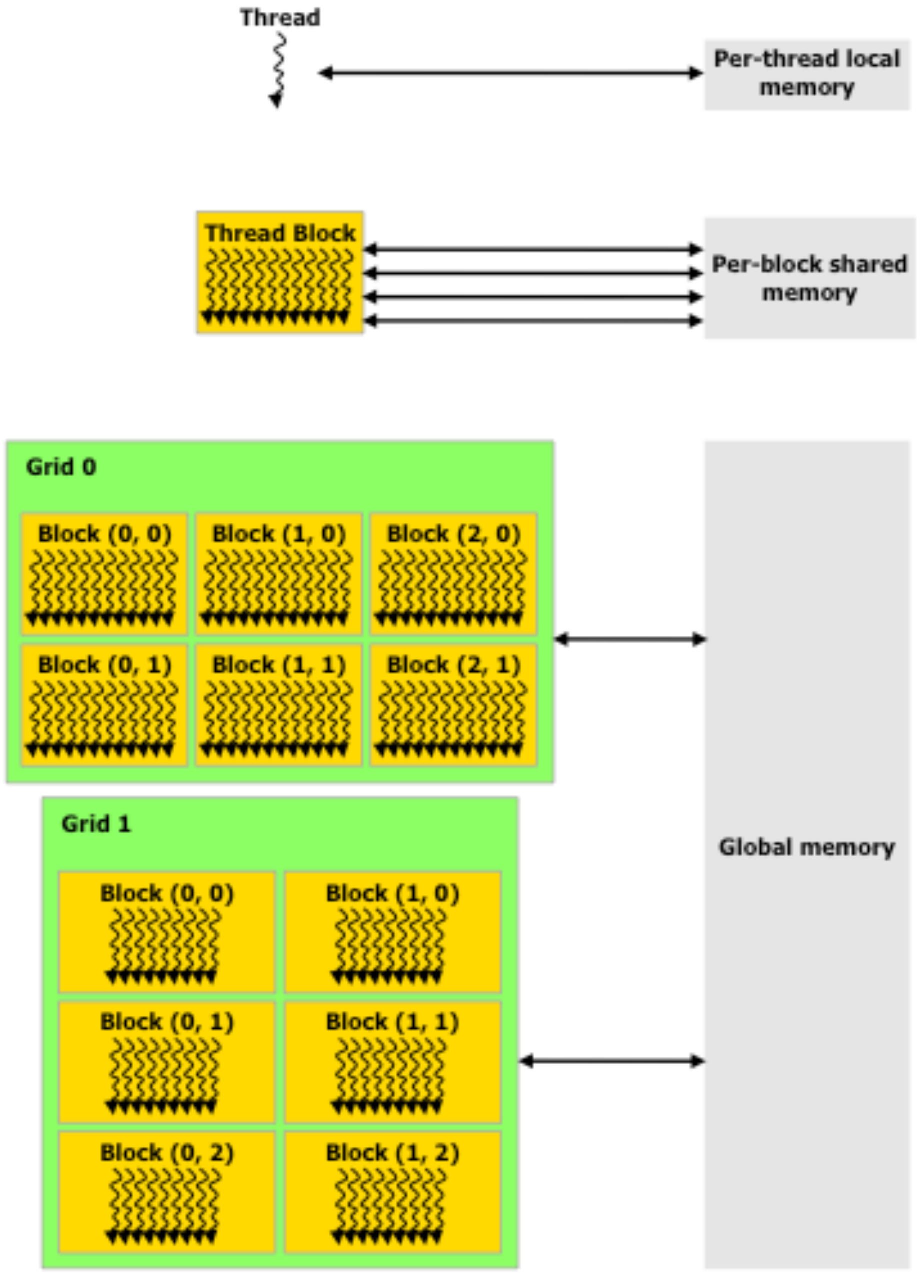
Memory Model

- Texture
  - Write via CPU
  - Allows hardware interpolation
  - 2D Locality of arrays
- Constant
  - Write via CPU
  - Small, used for random access inst'ns
- Global
  - Write via CPU and GPU
- Shared
  - Local to Block
  - Low latency
  - Fastest comms between threads
- Local
  - Per thread only memory
- Registers
  - Thread only,
  - Fastest memory available
  - Limited space



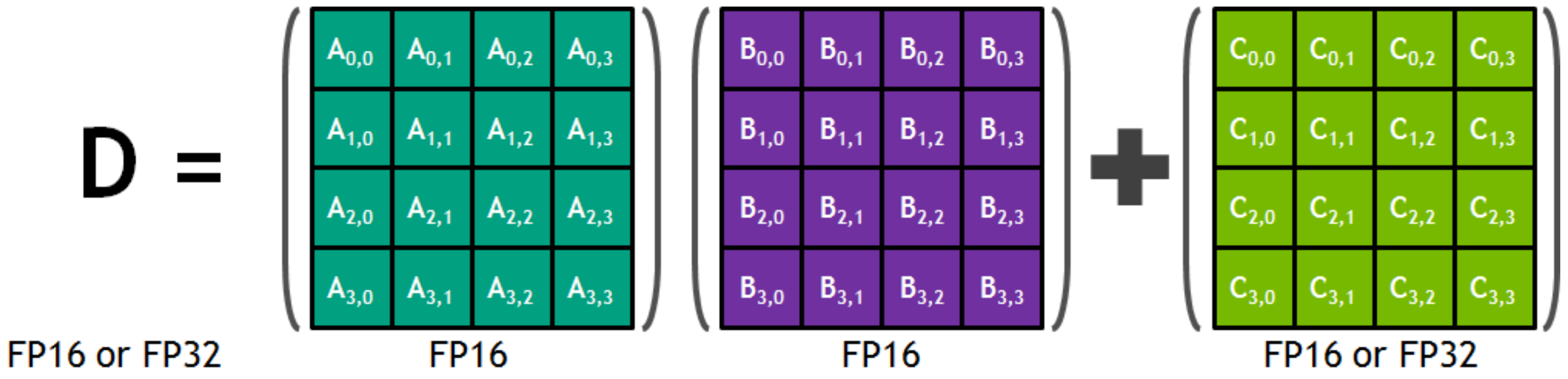
Memory Scope

- Registers are local to thread and has thread lifetime.
- Shared memory shared between threads in a block and has block lifetime
- Global memory is accessible everywhere and is persistent.



Tensor Cores

- One of the key operations when training a neural network or doing ML is matrix-matrix multiplication (DGEMM)
- Each tensor core has a 4x4x4 matrix processing array
- Can do 64 mixed-precision OPs/clock with half-precision inputs and either HP or SP output



$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

Programming Technologies (I)

- CUDA (Compute Unified Device Architecture)
  - Nvidia's proprietary platform.
  - Offers both API (extensions to C/C++) and driver level programming.
  - Proprietary PGI Fortran compiler allows Fortran development – both directive and kernel modes.

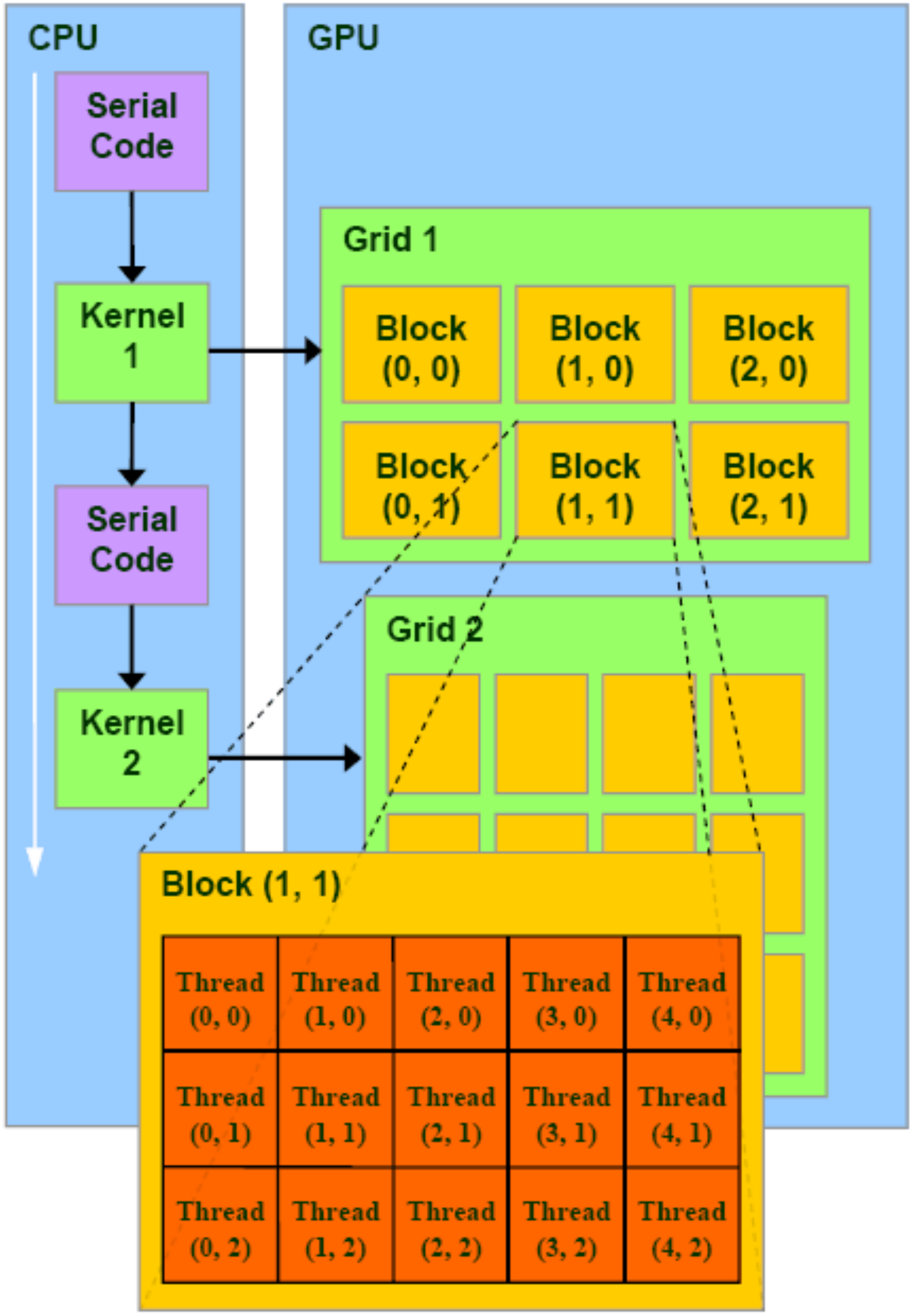
- OpenCL (Open Computing Language)
  - Evolution of OpenGL to become a general solution for heterogenous computing (e.g. GPUs and CPUs).
  - Implemented on a driver level – e.g. built into MacOS
  - Specification is manufacturer (and device) independent – write once, run anywhere.

Programming Technologies (II)

- OpenACC
  - Open standard version of the directives approach of PGI
  - Spec v2.0 July 2013 has better support for control of data movement, calling external functions, and separate compilation for host & device so can build libraries
  - Led by nVidia with Cray and PGI, has support for Fortran, C/C++
  - Now in gcc and gfortran (since v6.1)
- OpenMP v4.0
  - Pushed by Intel to support Xeon Phi etc
  - Subset of OpenACC functionality at present
  - In gcc/gfortran since v4.9.1
  - Supposed to include all of OpenACC in future but difficulties with Intel vs nVidia …

CUDA Programme Structure

- Serial (host) code
  - Kernel (device) code
    - Grid
      - Blocks
        - Threads
- Must remember to allocate data on both host and device
- Kernel executes asynchronously



Starting CUDA

- Headers must be included. In this introduction only the API will be demonstrated.

```
use cudafor
```

```
#include <cuda.h>;
#include <cuda_runtime.h>;
```

- C/C++ files with CUDA kernels must have the extension .cu
- Fortran CUDA files must have the extension .CUF

- See references at end for more complete examples ...

Device Kernels

- A **Kernel** is a global subroutine (static function) which runs on the device. One instance of the kernel will be executed by every thread which is invoked.

```
attributes(global) subroutine my_kernel( a, b, c)
```

```
static __global__
void my_kernel( float * a, float * b, float * c);
```

- Kernels can only address device memory spaces.

- Executing kernels behaviour differs by using the thread index which uniquely identifies the thread.

```
tx = threadidx%x + (blockidx%x * blockdim%x)
```

```
int tx = threadIdx.x + (blockIdx.x * blockDim.x);
```

Host Code

- Device memory must be allocated **in advance** by host:

```
real,device,allocatable :: Adev(:)
...
allocate(Adev(M,N),stat=istat)
```

```
static __device__ double * devPtr = NULL;
...
cudaMalloc((void**)&devPtr, N*sizeof(double));
```

- Data must be copied (over the bus) to the device and if necessary copied back to the host later. This is very slow! (Host memory can be allocated in non-paged (pinned) memory to avoid one copy operation).

```
Adev = A  ! Host to device
```

```
cudaMemcpy(devPtr, hostPtr, N*sizeof(double),
  cudaMemcpyHostToDevice);
```

Kernel Execution

```
call my_kernel<<<grid,block>>>(arg1,arg2,...)
```

```
my_kernel<<<grid,block>>>(arg1,arg2...);
```

- **`grid`** – specifies the dimensions of the grid (i.e. the number of blocks launched will be the product of the dimensions).
- **`block`** – specifies the dimensions of each block (i.e. the number of threads per block is the product of the dimensions).
- **`dim3`** – derived type which has three members.

```
type(dim3)  ::  grid
integer  ::  x=5,y=5,z=1
grid  =  (x,y,z)
```

```
int x=5,y=5,z=1;
dim3 grid(x,y,z);
```

Kernel Compilation and Run

- **CUDA** kernels must be compiled with a CUDA compiler (pgfortran or nvcc).

- At runtime the kernel will be copied to the device on first execution – note that for accurate profiling, a kernel should be executed at least once before timing.

- **OpenCL** kernels are usually compiled at **runtime**. This allows the kernel to be optimised for the running context but has a larger initial overhead.

Advanced Features

- Streaming – Data can be moved across the system bus while computation is happening. This is good for large data structures which don't fit in device memory.
- Texture/Surface memory – Memory can be accessed via non-integer or surface coordinate! Linear interpolation can also be performed.
- Fast intrinsic functions – Some special functions exist which allow certain operations to be performed very quickly although are not IEEE compliant. For example `rsqrt` – reciprocal square root. These can be useful for areas of the code where precision is less important.

(free) CUDA Libraries

- V6 (2014+) Drop-in replacement for BLAS etc with auto offload from CPU to GPU so free speed-up!
- CUBLAS – BLAS library
  - Includes all S,D,C,Z level 1-3 BLAS routines
- CUFFT – FFT library
  - 1D, 2D, 3D complex and real
  - Stream enabled for parallel data movement & computation
- CUSPARSE – Sparse matrix library
  - BLAS style routines between sparse & dense matrices
- CURAND – Random number library
- Now also XT versions for multi-GPU support (non-free)

CUBLAS Fortran Interfacing

- Interfacing the CUBLAS library (written in C) with the cudafortran from PGI requires an interface to be written using the C-interoperability part of F2003, e.g.

```
Module cublas
Interface cuda_gemm

Subroutine cuda_sgemm(cta, ctb, m, n, k alpha, A, &
&  lda, B, ldb,Beta, c, ldc)
bind(C,name='cublasSgemm')
use iso_c_binding
character(1,c_char),value :: m,n,k,lda,ldb,ldc
real(c_float),value :: alpha,beta
real(c_float),device,dimension(lda,*) :: A
real(c_float),device,dimension(ldb,*) :: B
real(c_float),device,dimension(ldc,*) :: C
End subroutine cuda_sgemm

End interface cuda_gemm
End module cublas
```

Fortran Example

```
subroutine mmul( A, B, C )
use cudafor
real, dimension(:,:) :: A, B, C
integer :: N, M, L
real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
type(dim3) :: dimGrid, dimBlock

N = size(A,1) ; M = size(A,2) ; L = size(B,2)
allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
Adev = A(1:N,1:M) ; Bdev = B(1:M,1:L)
dimGrid = dim3( N/16, L/16, 1 )
dimBlock = dim3( 16, 16, 1 )

call mmul_kernel<<<dimGrid,dimBlock>>>( Adev,Bdev,Cdev,N,M,L )

C(1:N,1:M) = Cdev
deallocate( Adev, Bdev, Cdev )
end subroutine
```

Fortran Kernel

```fortran
attributes(global) subroutine MMUL_KERNEL( A,B,C,N,M,L)
real,device :: A(N,M),B(M,L),C(N,L)
integer,value :: N,M,L
integer :: i,j,kb,k,tx,ty
real,shared :: Ab(16,16), Bb(16,16)
real :: Cij

tx = threadidx%x ; ty = threadidx%y
i = (blockidx%x-1) * 16 + tx ; j = (blockidx%y-1) * 16 + ty
Cij = 0.0
do kb = 1, M, 16 ! Fetch one element each into Ab and Bb NB 16x16 = 256
!  threads in this thread-block are fetching separate elements of Ab and Bb
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j) ! Wait until all elements of Ab and Bb are filled
    call syncthreads()
    do k = 1, 16
        Cij = Cij + Ab(tx,k) * Bb(k,ty)
    enddo ! Wait until all threads in the thread-block finish with this Ab and Bb
    call syncthreads()
enddo
C(i,j) = Cij
end subroutine
```

OpenMP for GPUs

- Spec v4.5 finalized in Nov 2015 (C/C++/Fortran)
- Extensions to SIMD and TASK
- Array reduction (Fortran only) now allowed
- Extended TARGET attributes for better accelerator performance
- Lots of new stuff for DEVICE …

- Available in GNU v6.1 and Intel v18.0
- OpenMP V5.0 (needed for multi-GPU) started in v9 and still not complete ...

OpenMP v4.5

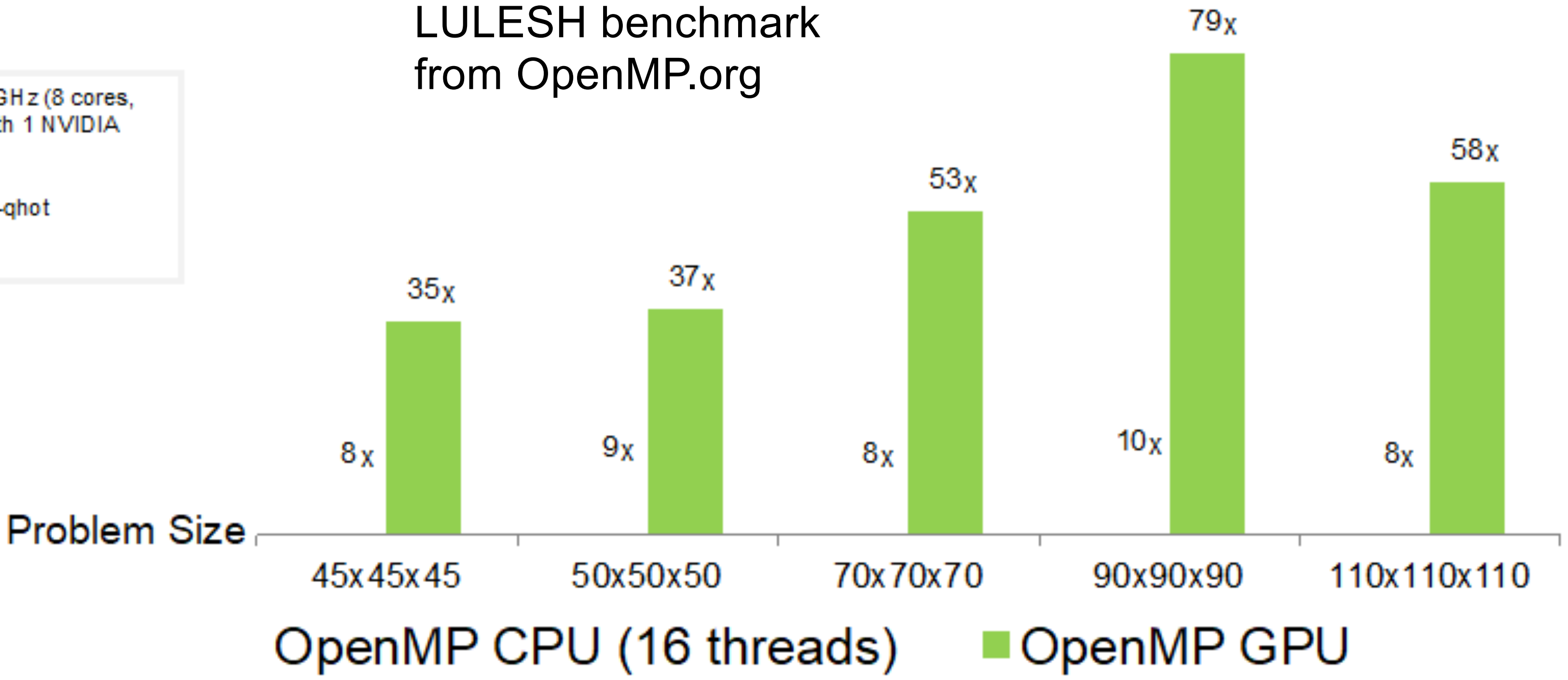
- On CPU: `#pragma omp parallel for`
- On GPU: `#pragma omp target teams distribute parallel for`
- IBM XL compiler with nVidia offloading (Dec ’16):

Test Specs

2 Power8 sockets @ 4GHz (8 cores, with 8 threads each) with 1 NVIDIA Pascal P100 GPU.

Compiler Options: -O3 -qhot -qsmp=omp -qoffload*
* Where applicable

LULESH benchmark from OpenMP.org



OpenACC (I)

- Open standard for accelerator programming
  - Led by nVidia in response to CUDA vs OpenMP
  - Competing standards but (slowly) converging …
  - Specify a loop or region of code to offload to accelerator and compiler does the rest!
  - “guidance” not explicit actions as with OpenMP
- V1.0 (2011) – parallel, kernels, loop, data
- V2.0 (2013) – added support for offloading subroutines and atomic constructs
- V2.6 (2017) – added sync, async and serial
  - Available in gcc/gfortran v10.0 onwards

OpenACC (II)

- Compiler can *auto-generate* kernels for a section of code (e.g. multiple loops and/or Fortran array operations)
  - Maximum flexibility but hard to get best speed
  - Auto-detect dependencies:

```
!$acc kernels
   fortran loop(s) to be executed on device
!$acc end kernels
```

```
#pragma acc kernels
{
   c loop(s) to be executed on device
}
```

OpenACC (III)

- Or user can assert a single loop is dependency-free and hence safe to be parallelized:

```
!$acc parallel loop
for i=1,N                   !single fortran loop
!NB no matching end-parallel
```

```
#pragma acc parallel loop
for (i=0;i<N;i++)
{
   //single c loop to be executed on device
}
// NB no block braces
```

- NB data may be copied back from host between successive `parallel` sections but stays on host for duration of a `kernels` section

Further Reading

CUDA Developer ZONE
https://developer.nvidia.com/cuda-education-training

OpenCL Standard:
http://www.khronos.org/opencl

OpenACC standard:
http://www.openacc.org

OpenMP v5.0
http://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf