MANCHESTER
1824

The University
of Manchester

Chip Multiprocessors
COMP35112

Lecture 14 - GP-GPUs and CUDA

Graham Riley

Today's Lecture

- General purpose graphical processing units (GP-GPUs) provide cheap access to parallel computing resources
- They have been used successfully for some applications
- Need to talk about:
  - Hardware characteristics
  - How to program applications to use them

GPU Characteristics

- History: an ‘attached processor’ designed to reduce load on CPU caused by need to draw sophisticated images many times per second – driven by games…
- Characteristics therefore:
  - Multi-core, usually an entirely separate memory
  - Good Floating Point performance
  - High Memory Bandwidth: ~3 × that of CPUs (in 2008)
    - Not changed much since
- NVidia GeForce GTX 280 GPU
  - had 240 cores - ~933GFlop/s (2008)
- NVidia Titan X: 3584 cores – 11TFlop/s (from 2016)

GPU Instruction Set

- In 1980s and 1990s, graphics hardware was configurable (via API calls from CPU), but not programmable – matched to algorithms used in the graphics pipeline
- In 2001-2, application developers got access to the actual instruction set used
- Still more restricted than CPU
- Massive SIMD data parallelism built in!
- GP-GPU = general purpose GPU – can be used via an API for *stream computing*

The Graphics Pipeline

- The early fixed-function pipelines (e.g. NVIDIA GeForce) contained, in order:
  - Host interface (DMA bulk data)
  - Vertex Control (loads vertex cache)
  - Vertex shading/transformation & lighting
  - Triangle setup
  - Raster
  - Shader (using a Texture cache) ! Lots of floating point here
  - Raster Operations
  - Frame Buffer Interface

MANCHESTER 1824

The University of Manchester

Writing General Purpose Applications for a GPU

- To compute a function, it could be written as a *pixel shader*
- Input data would be stored as a *texture image*
- Output had to be cast as *pixels*
- Constraints on data structure and data transfer
  - To "fit" in with the graphics pipeline view
- To use outputs from one phase of the program, they had to be written to the pixel frame buffer, which could then be used as a texture map input to the next phase
- Heroic efforts were made to go general purpose using OpenGL and DirectX …quite low level APIs

Later Developments

- Using more general purpose units, the pipeline is implemented by recycling through these units
  - i.e. no longer have dedicate h/w per pipeline stage
- e.g. in the GeForce 8800 GPU, the first to use the Tesla architecture, the processors are visited 3 times

Tesla Architecture

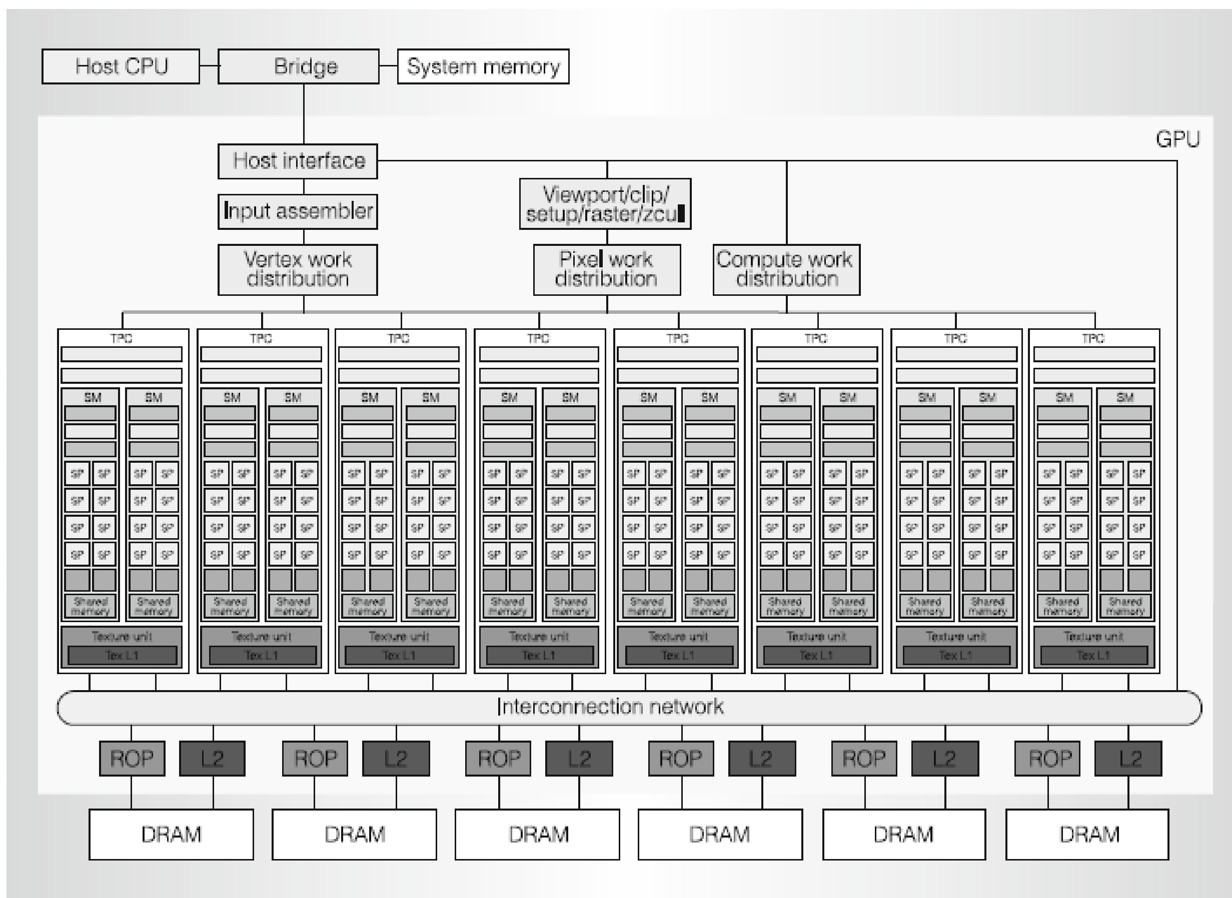


Figure 1. Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor.

Tesla Design

- More ‘general purpose’ programmable architecture
- Multiple independent processor ‘clusters’
- Each cluster has 2 ‘streaming multiprocessors’ (SMs)
- Each SM has 8 ‘streaming processors’ (SPs) (cores)
  - Plus two Special Function Units (SFU) for sin/cos etc.
  - Plenty of registers supporting fast context switch
- SIMD execution: all cores in a SM execute the same instruction but on different data
  - Care needed to ensure cost of loading data minimised (e.g. “coalesced” memory accesses)

Tesla Principles

- Re-circulating data, so variable (algorithm) pipeline length – flexible for new algorithms
  - Keep data on devices and transform with multiple "kernels"
- Clusters allocated dynamically to different processing stages ("kernels")
- Cluster computations can be totally different threads
  - E.g. from multiple kernels, even applications
- Need lots of independent work in applications to keep the clusters busy

Stream Multiprocessor Cluster

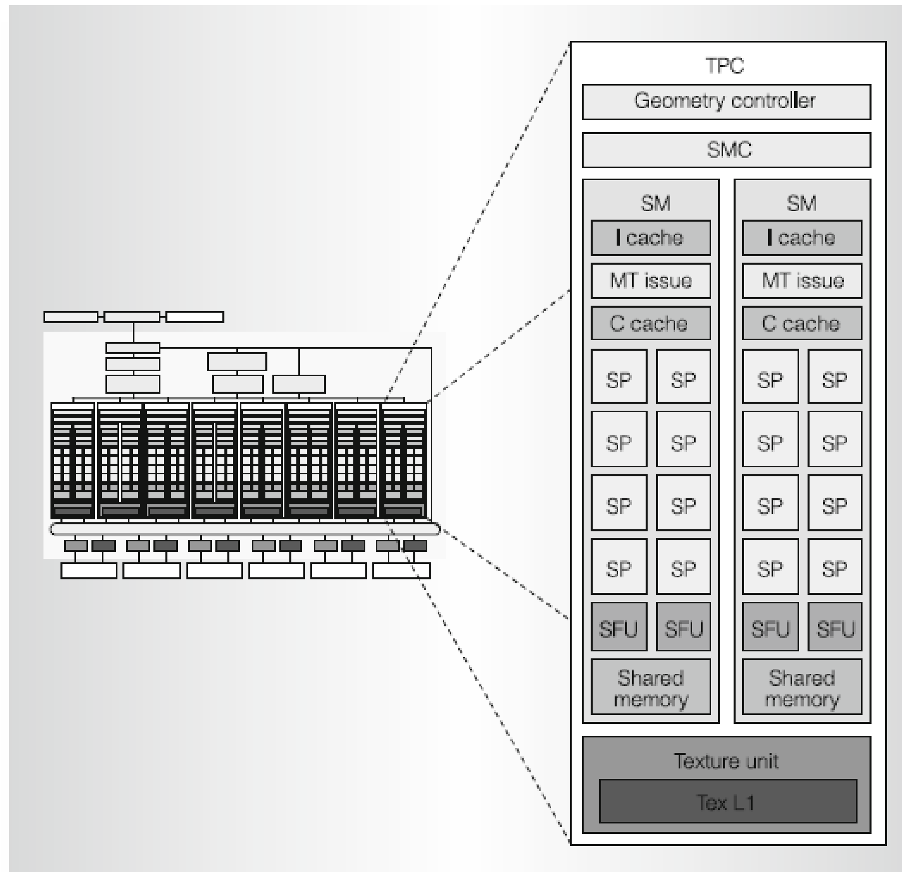


Figure 2. Texture/processor cluster (TPC).

Next Generations – Fermi & on

- The next generation Tesla architecture was called Fermi (then Kepler, Maxwell, Pascal)
- It had much more hardware parallelism
- And more sophisticated functionality is included, such as virtual memory management, better f.p. support
- Recent (2016) is the Pascal architecture
  - NVIDIA Titan X

See:

www.nvidia.com/object/tesla_computing_solutions.html

http://www.geforce.co.uk/hardware/10series/titan-x/

MANCHESTER
1824
The University
of Manchester

NVIDIA Titan X Architecture (in 2017)

NVIDIA Titan X

- 12 Billion transistors
- 3584 CUDA cores
- 1.5GHz
- 12GB memory
- 11 Tflop/s performance
- 480 GB/s bandwidth

The CUDA Programming Language

- CUDA = 'Compute Unified Device Architecture'
  - Specific to NVIDIA hardware
- Essentially heterogeneous
  - Host = CPU
  - Device = GPU
- Program (in C) has code for both host and device(s), and compiler separates it
- Device code is data-parallel *kernels* generating a large number of threads
- Note that these threads are **very** lightweight

Example: Matrix Multiplication

- This is a typical data-parallel application
- Basic operations are floating-point multiply-adds
- These are done in the kernels
- Host code has to organise things, copy data values to and from device memory, handle the answer when produced, etc.
- This is an example of a single kernel
  - But think in context of an application with multiple kernels
  - Data sent to device and transformed by multiple kernels

Outline of CUDA Code: P=M*N

```
void MatMul (float *M, float *N, float *P, int width) {


// allocate device memory for M, N, P
// copy values of M and N to allocated locations
// invoke kernel code to perform the calculation
// copy P back from device memory
// free device memory
}
```

Detail: Allocate Device Memory

```
float *Md ;
int size = width * width * sizeof(float) ;
cudaMalloc((void **)&Md, size) ;
....

cudaFree(Md) ;
```

Detail: Copying Values

```
…
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice) ;
…
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost) ;
```

Detail: Kernel Code

```
__global__void MMkernel(float *Md, float *Nd,
                float *Pd, int width)
{ int tx = threadIdx.x ; int ty = threadIdx.y ;
  float Pvalue = 0;
  for (int k = 0 ; k < width; k++)
        { float Mdel = Md[ty*width + k] ;  float Ndel =
          Nd[k*width + tx] ;  Pvalue += Mdel * Ndel ;
        }
  Pd[ty*width + tx] = Pvalue ; }
```

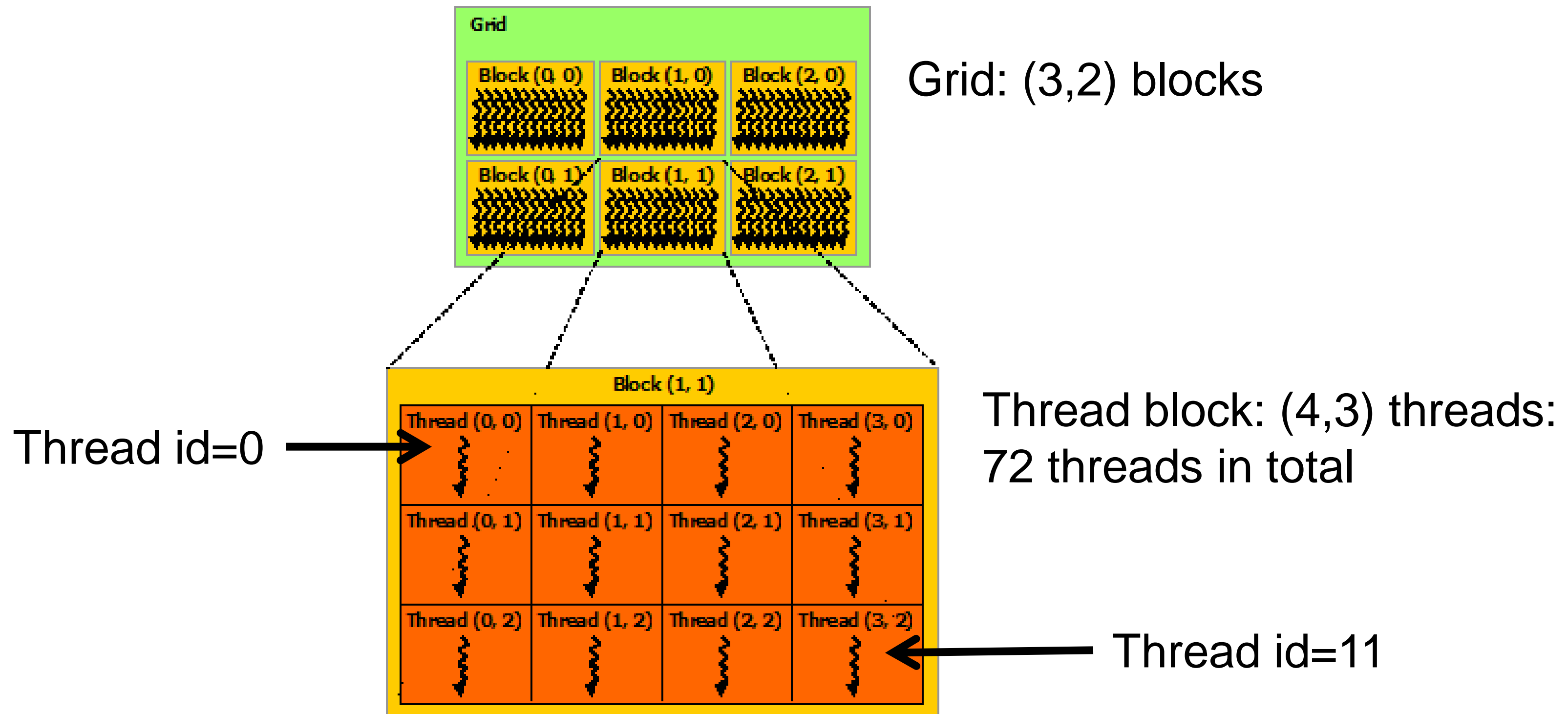
Three CUDA Function Declaration Prefixes

- __device__ executed on device, called from device
- __global__ executed on device, called from host
- __host__ executed on host and called from host
- Note: a function can be both __host__ and __device__, generating 2 versions of the code!
  - Enables heterogeneous execution – part on CPU, part on GPU
- Compile using nvcc – the NVIDIA compiler

Kernel Execution

- When invoked (*launched*), the kernel is executed as a Cartesian **grid** of parallel threads (“**thread blocks**”)
- The host has control over how many threads there are, and how they are organised
- Launch code is therefore slightly complicated
- A thread block is executed to completion on the SM on which it runs
- Multiple thread blocks can be allocated to an SM
  - to help hide memory latency (by switching between thread blocks on a memory stall)

Grid of Thread blocks



- Thread blocks are split into “warps”, related to the number of cores in an SM, for execution
- One block may consist of many warps

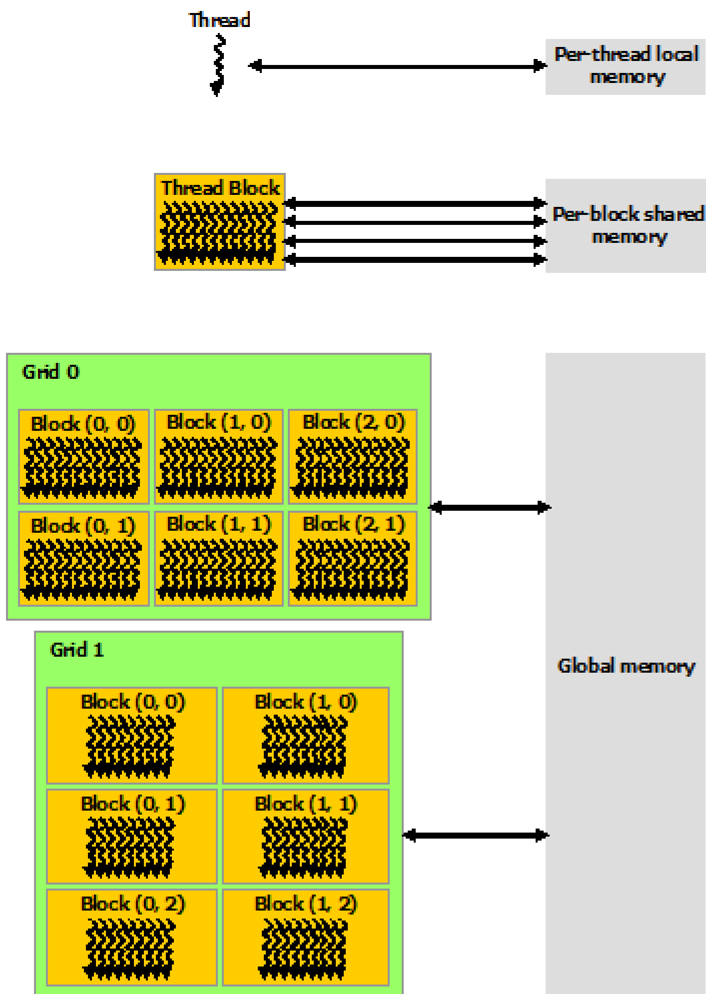
Detail: Kernel Launch Code

```
dim3 dimBlock(width, width) ; // size of thread block
dim3 dimGrid(3,2) ; // configuration of thread …
                    // … blocks in the grid
MMkernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
        width) ;
```

Where: <<<…>>> is the “execution configuration”,

<<< number of blocks, threads per block >>>

CUDA Memory model



- Total number of active threads limited by per-thread local memory (e.g. registers)
- Need to choose thread block sizes to best share the limited block resources
- Other perf. issues exist…
- Programming Guide gives tips, and helpful tools exist to tune codes.

OpenCL, OpenMP and OpenACC

- Standardised languages for GP-GPUs, cross-platform
- OpenCL development was initiated by Apple, but done by Khronous Group (OpenGL people!)
  - Derives heavily from CUDA
- OpenMP4.x has pragma-based support for executing code on accelerators, intended to satisfy needs across a broader range of GP-GPU and attached processor architectures
- OpenACC is another design, based on pragmas – closely related to (and works together with) OpenMP
  - Ideally OpenACC and OpenMP will converge in future…

Next and Final Lecture

- Programming for the kinds of multi-core architectures we have described is certainly challenging
- There has been long-standing dissatisfaction about this state of affairs, especially in academia
- The next, and final, lecture surveys alternative approaches (for both languages and hardware) that live on now and were studied in the 1970s -1990s, particularly here in Manchester:
  - Functional Programming and Dataflow Principles
  - A current resurgence of interest in the form of “task-based” programming for heterogeneous systems