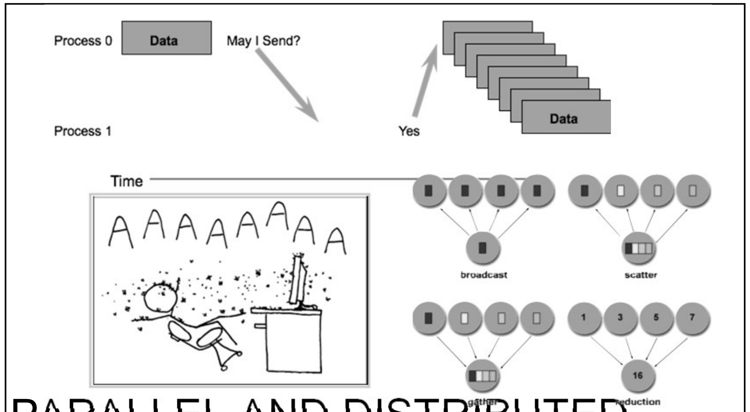


PARALLEL AND DISTRIBUTED COMPUTING

GPU AND OPENCL

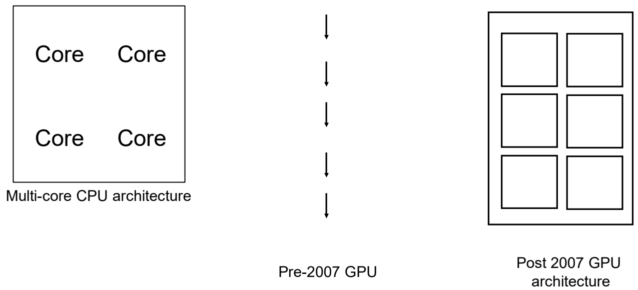
1

What is it?

- GPUs are very fast processors to perform the same computation (shaders) on collections of data (vertices, pixels)
  - data parallelism! (with origins in SIMD)
  - scientific computations
  - GPGPU - general purpose GPU computation
  - accelerator

2

NVIDIA Tesla (2007)



3

Hardware

- Examples of GPU products
- Consumer graphics cards (GeForce):
  - GTX980: 2048 cores, 224 GB/s
  - GTXTITAN Z: 5760, 672 GB/s
- Dedicated HPC cards (no graphics output):
  - K80: 4992, 480 GB/s
  - K40: 2880, 288 GB/s

4

NVIDIA GPU design

- Building block is a “streaming multiprocessor” (SMX):
  - 192 cores and 64k registers
  - 64KB of shared memory / L1 cache
  - 8KB cache for constants
  - 48KB texture cache for read-only arrays up to 2K threads per SMX
- Different chips have different numbers of these SMX

Multi-threading

- Key hardware feature is that the cores in an SMX are SIMT (Single Instruction Multiple Threads) cores:
  - all cores execute the same instructions simultaneously, but with different data
  - similar to vector computing on CRAY and other supercomputers
  - minimum of 32 threads all doing the same thing at (almost) the same time
  - natural for graphics processing and much scientific computing
  - SIMT is also a natural choice for many-core chips to simplify each core

Multi-threading

- Lots of active threads is the key to high performance:
  - no “context switching”; each thread has its own registers, which limits the number of active threads
  - threads become “inactive” whilst waiting for data or part of the compute group takes a divergent path (if statements)

GPU programming languages

- C-like languages to express programs that run on GPUs
- Relatively low level
- Many versions/types

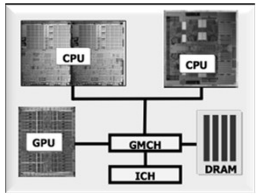
Top level software view

- Many heterogeneous devices, so it helps to think about the software view
- At the top level, we have a master process which runs on the CPU and performs the following steps:

1. initialises compute device
2. defines problem domain
3. allocates memory in host and on device
4. copies data from host to device memory
5. launches execution “kernel” on device
6. copies data from device memory to host
7. repeats 4-6 as needed
8. de-allocates all memory and terminates

Heterogeneity

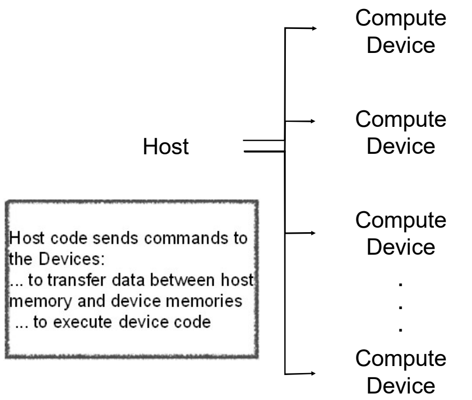
- A modern platform includes:
  - one or more CPUs
  - one or more GPUs
  - optional accelerators

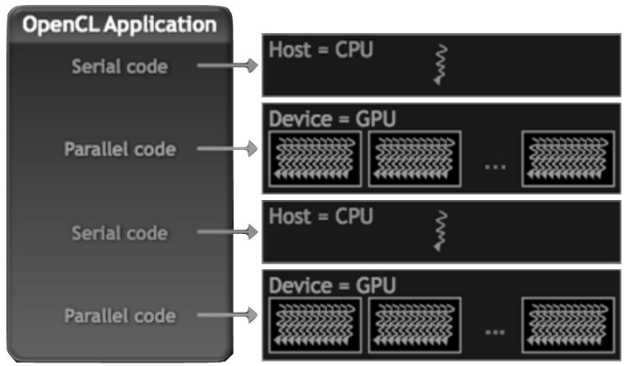


GMCH - graphics memory control hub
ICH - Input/output control hub

http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf

The OpenCL view



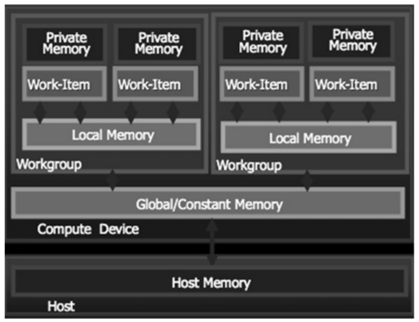


OpenCL execution model

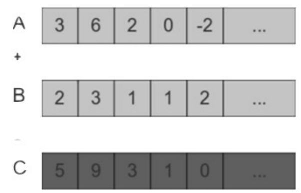
- Application runs on a ***host*** which submits work to ***devices***
  - Work-item: the basic unit of work on an OpenCL device
  - Kernel: the code for a work-item (basically a C function)
  - Program: Collection of kernels and other functions (analogous to a dynamic library)

OpenCL execution model

- Context: The environment within which work-items execute; includes devices and their memories and command queues
- Command Queue: A queue used by the Host application to submit work to a Device (e.g., kernel execution instances)
  - Work is queued in-order, one queue per device
  - Work can be executed in-order or out-of-order



MEMORY MANAGEMENT IS EXPLICIT: YOU MUST MOVE DATA FROM HOST-> GLOBAL-> LOCAL .. AND BACK




```
void trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Traditional

```
__kernel void
dp_mul(__global const float *a,
__global const float *b, __global
float *c)
{
  int id = get_global_id(0);
c[id] = a[id] * b[id];
}
// execute over n "work items"
```

OpenCL Kernel