High-Performance Computing using GPUs

Is that a Weave?

Mike Anderson
Chief Scientist
The PTR Group, Inc.
http://www.theptrgroup.com

What We Will Talk About

- Some processor basics
- Parallelism in computing
- The GPU as a compute engine
- CUDA vs. OpenCL
- GPU computing in embedded systems
- Summary
- Demos

Current Processing Trends

- Since 2004, multi-core processing has become relatively common place
  - Seen as the best way to get more performance with reasonable power and thermal limits
  - Dual-core, ARM Cortex A9 platforms will achieve some critical mass this year
- Heterogeneous processors are also more common
  - IBM Cell, TI OMAP and others mix a "main" processor with attached, special-purpose processors

Video and Gaming Trends

- Users want 3D gaming, streaming media and voice/image recognition
  - The smartphone is the gaming platform of choice
  - This forces devices to be more sophisticated
- Even TV sets are now Internet aware
  - Gaming platforms, Facebook, streaming media ala NetFlix
- Many applications require high-end graphics capabilities
  - E.g., Android's support of OpenGL ES

Scalability of Algorithms

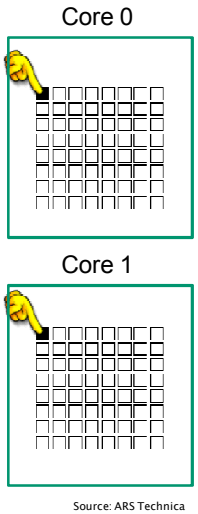
- If an algorithm is perfectly scalable then adding N processors increases the speed N times
- This is represented in Amdahl's Law:

  $S_p = T_1/T_p$

  where S is the speed up, T is the time to execute an algorithm and p is the number of processors
- Unfortunately, most code is rarely perfectly scalable due to IPCs, synchronization primitives and bus contention

Increasing Performance via Parallelism

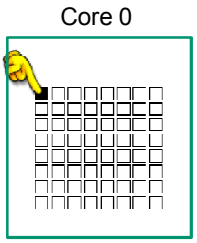
- Multi-core processors can increase performance assuming that your application fits a thread-parallel model
  - Multiple, concurrent threads of control
  - This is also known as multiple-instruction/multiple-data (MIMD) parallelism

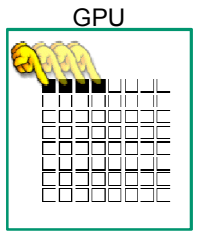


Source: ARS Technica

Data Parallelism

- However, there is another type of parallelism known as data-parallelism
  - Single-Instruction/Multiple-Data (SIMD)
  - E.g., create the dot-product or sum of 2 or more vectors
    - Also known as vector processing
  - Very common in graphics applications





Source: ARS Technica

Data Parallel Applications

- There are many applications that exhibit data parallelism
  - Software defined radio, medical imaging, computational fluid dynamics, machine vision, video CODECs, edge detection, video noise reduction and many more
- Very similar application space to traditional DSP work
  - Very computationally intensive

GPUs as General-Purpose Computers

- GPUs are actually "many-core" processors
  - Some have as many as 448 cores
- GPUs are typically optimized for data parallel work
  - More transistors working on a given problem
- Early efforts for GPGPU computing tried to use APIs like OpenGL for computing
  - Had to trick the GPU into thinking the data was actually graphics

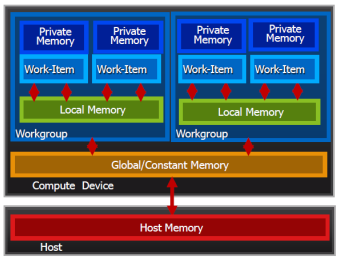
Source: NVIDIA

Comparing CPU and GPU

- GPUs have many more cores
  - CPUs: 2-64 GPUs: 16-448+
- GPUs have more memory bandwidth
  - GPUs use DDR5 RAM with 10x bandwidth of CPU front-side bus
- GPU floating point is much faster than CPU FP
  - 1 TFLOPs GPU vs. 50 GFLOPs for CPU
    - Nvidia Tesla 2070 vs. Intel Core i7 975 (single precision IEEE 754)
- The personal super-computer is born
  - The Cray-1 was only 80 MFLOPs!

Different Flavors of Memory

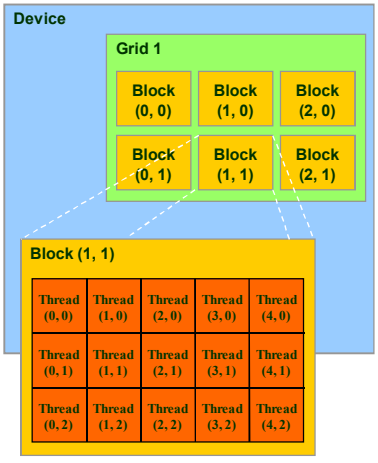
- GPUs have many different implementations of memory
  - Host memory
    - Can be "pinned" to facilitate DMA transfers
  - Global memory
    - On the GPU and shared between compute blocks
  - Local memory
    - Shared within a block
  - Constant memory
    - Cache-locked, read-only memory
  - Private memory
    - Local scratchpad storage
- The data flow is always from host -> global -> local and back
  - You're responsible for handling the data movement



Source: Khronos Group

Device/Block/Thread Relations

- It's possible to have more than one GPU (device) in a given system
  - The GPU may have many sequencing engines
- The sequencing engines allow for multiple kernels (blocks) to run in parallel
  - Each block can have multiple threads running in parallel



Asynchronous Computation

- One of the more difficult concepts with GPU computing is that the blocks can execute in any order
  - Much like processing a mosaic
    - The order is immaterial as long as everything is processed
- The APIs provide facilities to synchronize threads, blocks and the host to ensure that everything is complete before moving to the next stage
  - There are queues of blocks and events to signal between them

New APIs

- GPU manufacturers started to realize there was another market for their parts
  - They started developing language extensions more focused on data movement and manipulation
- NVIDIA introduced the CUDA™ in November of 2006
  - Compute Unified Device Architecture
  - The CUDA™ C SDK is a C/C++ extension
- AMD followed suit with their GPUs

Source: fotki.com

CUDA™ and OpenCL

- NVIDIA's CUDA™ is a proprietary extension of C/C++
  - The SDK includes a special compiler (nvcc) that extracts out the "kernel" and CUDA™ intrinsics and compiles them separately from host code
- Apple(!) wanted a more vendor neutral implementation
  - They created the open compute language "OpenCL" and turned it over to a consortium (the Khronos Group) to promote it
    - NVIDIA, AMD, Apple, ARM, Qualcomm, Samsung and many others are members

CUDA™ GPU Code

- GPU programming requires you to restructure/rethink your code:

Standard C Code	CUDA C Code

void
saxpy_serial(int n, float a,
float *x, float *y)
{
for (int i = 0; i < n; ++i)
y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
	__global__ void
	saxpy_parallel(int n, float a, float *x, float *y)
	{
	int i = blockIdx.x*blockDim.x +
	threadIdx.x;
	if (i < n) y[i] = a*x[i] + y[i];
	}
	// Invoke parallel SAXPY kernel with
	// 256 threads/block
	int nblocks = (n + 255) / 256;
	
	saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);

Source: NVIDIA

- But, the performance increase can be as much as 100x the CPU

Example OpenCL Vector Multiply

- OpenCL looks very much like CUDA™:

Traditional loops

```
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];       }
```

Data Parallel OpenCL

```
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
```

Source: Khronos Group

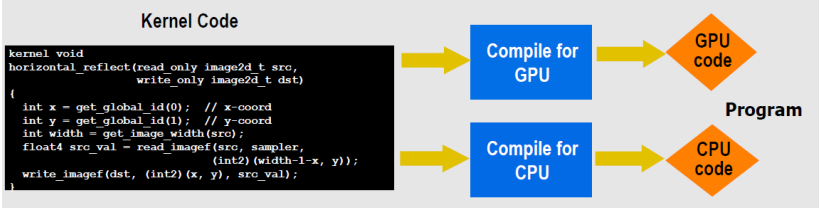
- Computation can be performed in many different configurations
  - E.g., 1 1024x1024 block or 1,048,576 invocations of 1 block or anything in between

OpenCL and OpenGL

- OpenCL and OpenGL have direct interoperability
  - OpenCL objects can be created from OpenGL textures, buffer objects and render-buffers
- The OpenCL execution model is very similar to CUDA™
  - Execute a kernel at each point in a problem domain
- Kernels (functions) can be split into blocks comprised of threads
  - 32 threads constitutes a "weave"
- Thread parallelism can be expressed as a sequence of kernels

Building Programs

- CUDA™ and OpenCL are similar in their approaches
  - The program source contains both host and GPU code
  - The preprocessor separates the two code bases and compiles them separately only to be linked back together at the end
- The run time then loads the GPU code on demand and coordinates with the host



Source: Khronos Group

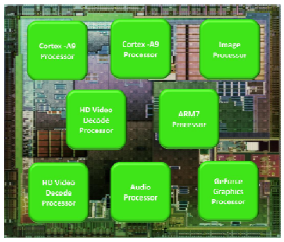
Debugging on GPUs

- Debugging is currently performed using good ol' gdb
  - NVIDIA's version is known as CUDA-GDB
    - This version also includes additional memory leak and profiling tools
- Front-ends like Eclipse and DDD work just fine
- Allows setting breakpoints in GPU or host application
  - Includes memory inspection as well as thread/block debugging on GPU

Embedded Processor GPUs

- Because OpenCL is related to OpenGL, conceivably any GPU that supports OpenGL could run OpenCL
  - Useful for dozens of applications requiring video or image processing
- NVIDIA Tegra 3 processors are rumored to be CUDA™ enabled
  - Used for H.264 decoding and 1080p video support
    - Quad-core ARM Cortex A9
- The NVIDIA ION processors are also CUDA™ enabled
- VIA EH1 graphics solution for pico & nano-ITX supports OpenCL

NVIDIA Tegra 2



Source: NVIDIA

CUDA™ or OpenCL?

- Given two standards which should you choose?
- CUDA™ is proprietary to NVIDIA
  - It's a couple of years ahead of OpenCL in terms of maturity
  - Supported by many commercial vendors like MatLAB and AutoCAD
- OpenCL has just recently released V 1.1
  - Many new enhancements to make it more comparable to CUDA™
  - However, OpenCL is still a little cumbersome
- OpenCL does run on both NVIDIA and AMD
    - Presumably, Intel will support OpenCL as well
- So, assuming that support for open platforms is what turns your crank, OpenCL should be your focus
  - It's the same reason you're likely using Linux ☺

Summary

- Continuing demands for sophisticated applications on mobile or consumer computing platforms will force more parallelism
  - Thread parallelism is fairly obvious
  - Data parallelism requires more thought
- Mobile devices with GPUs are becoming common
  - There are extra processor cycles in there for the adventurous
- Many Linux applications are already GPU-aware
  - Xine and many others
- The SDKs are free
  - Download and take a look

Demo time…

- We'll now look at a couple of examples…