CSC2/452 Computer Organization
Introduction to Accelerators

Sreepathi Pai

URCS

December 5, 2022

Outline

Outline

Administrivia

- A5 (final assignment) is out
  - Due Dec 13, 2022 at 7PM
  - One more interesting bug before whole class gets extra credit
- Homeworks all done
  - HW6 and HW7 still being corrected
  - Please review all of them and their solutions
- Two review lectures next week

Outline

Accelerators

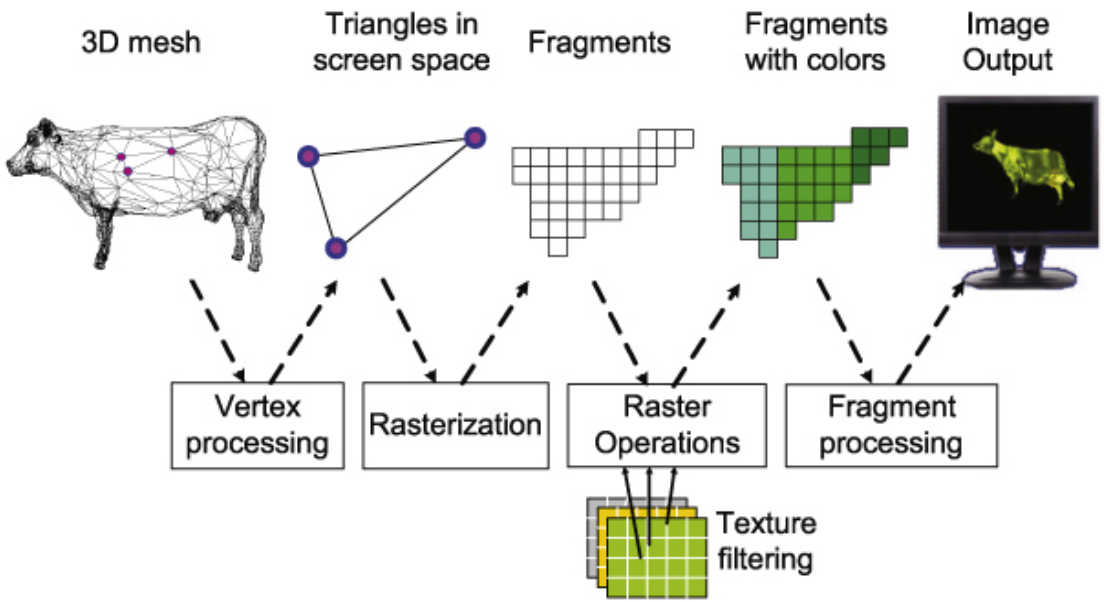
- Single-core processors
- Multi-core processors
- What if these aren't enough?
- Accelerators, specifically GPUs
  - what they are
  - when you should use them

Timeline

- 1980s
  - Geometry Engines
- 1990s
  - Consumer GPUs
  - Out-of-order Superscalars
- 2000s
  - General-purpose GPUs
  - Multicore CPUs
  - Cell BE (Playstation 3)
  - Lots of specialized accelerators in phones

The Graphics Processing Unit (1980s)

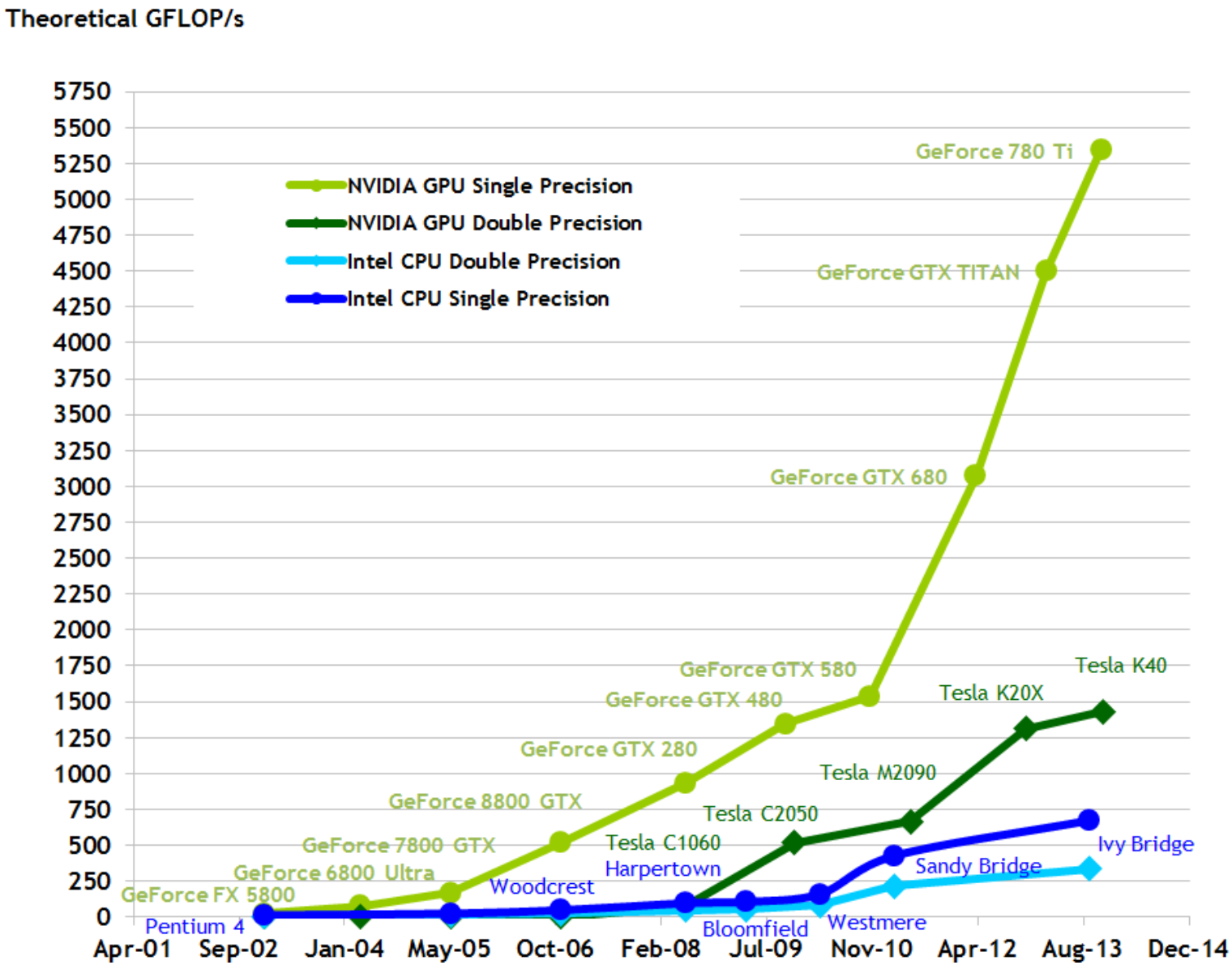
- SGI Geometry Engine
- Implemented the *Geometry Pipeline*
  - Hardwired logic
- Embarrassingly Parallel
  - $O(Pixels)$
  - Large number of logic elements
  - High memory bandwidth
- From Kaufman et al. (2009):



GPU 2.0 (circa 2004)

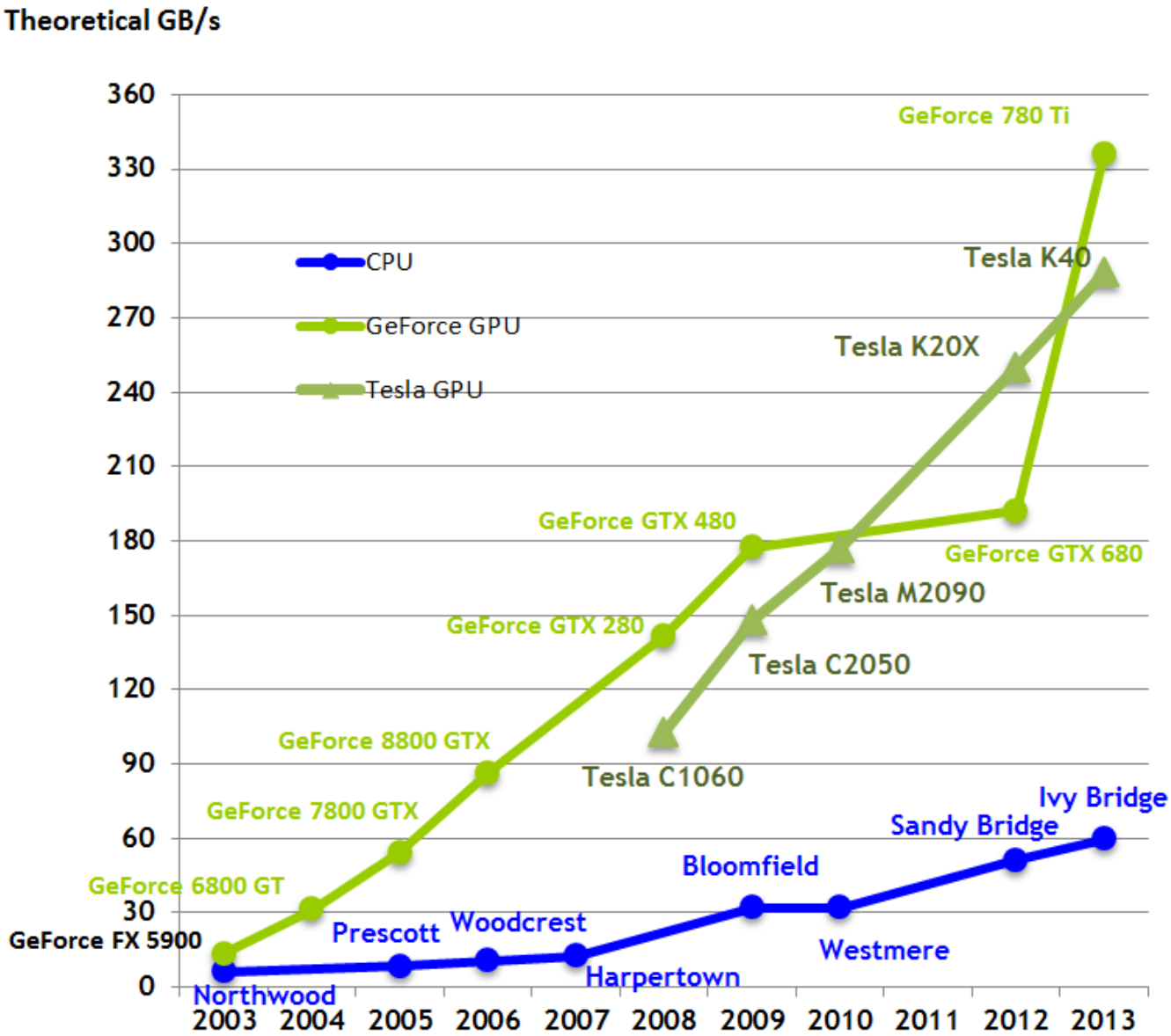
- Like CPUs, GPUs benefited from Moore's Law
- Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- Around 2004, GPUs were programmable "enough" to do some non-graphics computations
  - Severely limited by graphics programming model (shader programming)
- In 2006, GPUs became "fully" programmable
  - GPGPU: General-Purpose GPU
  - NVIDIA releases "CUDA" language to write non-graphics programs that will run on GPUs

FLOPS/s



NVIDIA CUDA C Programming Guide

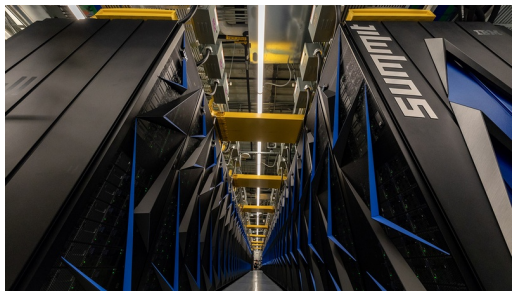
Memory Bandwidth



NVIDIA CUDA C Programming Guide

GPGPU Today

- GPUs are widely deployed as accelerators
- Intel Paper
  - 10x vs 100x Myth
- GPUs so successful that other accelerators are dead
  - Sony/IBM Cell BE
  - Clearspeed RSX
- Tesla V100S GPUs from NVIDIA have performance of 16.4TFlops (peak)
  - CM-5, #1 system in 1993 was 60 Gflops (Linpack)
  - ASCI White (#1 2001) was 4.9 Tflops (Linpack)



Pictures of Summit and Tianhe 1A from the Top500 website.

Accelerator Programming Models

- CPUs have always depended on co-processors
  - I/O co-processors to handle slow I/O
  - Math co-processors to speed up computation
  - H.264 co-processor to play video (Phones)
  - DSPs to handle audio (Phones)
- Many have been transparent
  - Drop in the co-processor and everything sped up
- Or used a function-based model
  - Call a function and it is sped up (e.g. "decode video")
- The GPU is not a transparent accelerator for general purpose computations
  - Only graphics code is sped up transparently
- Code must be rewritten to target GPUs

Using a GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.

Outline

The Two Kinds of GPUs

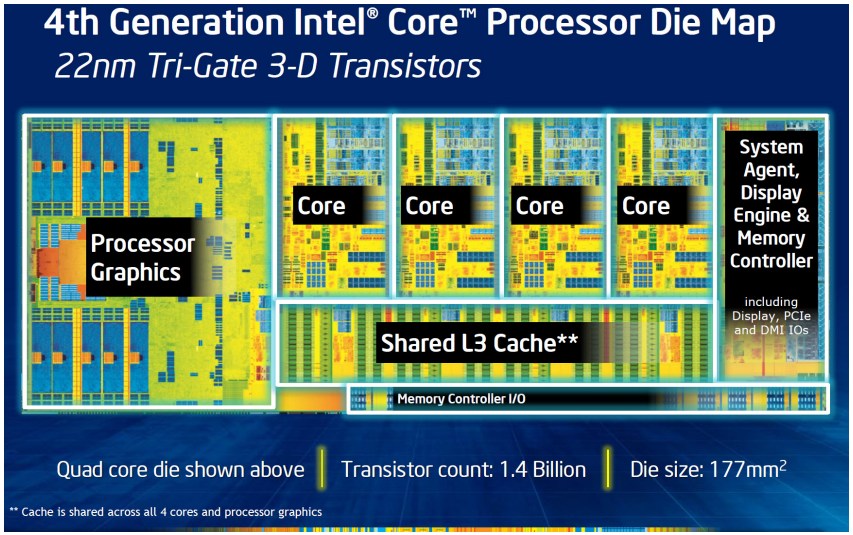
- Type 1: Discrete GPUs
  - More computational power
  - More memory bandwidth
  - Separate memory

NVIDIA

The Two Kinds of GPUs #2

- Type 2: Integrated GPUs
  - Share memory with processor
  - Share bandwidth with processor
  - Consume Less power
  - Can participate in cache coherence

Intel



The NVIDIA Kepler

NVIDIA Kepler GK110 Whitepaper

Using a Discrete GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.
- Working set must fit in GPU RAM
- You must copy data to/from GPU RAM
  - "You": Programmer, Compiler, Runtime, OS, etc.
  - Some recent hardware can do this for you (it's slow)

NVIDIA Kepler SMX (i.e. CPU core equivalent)

NVIDIA Kepler SMX Details

- 2-wide Inorder
- 4-wide SMT
  - 2048 threads per core (64 warps)
  - 15 cores
  - Each thread runs the same code (hence SIMT)
- 65536 32-bit registers (256KBytes)
  - A thread can use upto 255 of these
  - *Partitioned* among threads (not shared!)
- 192 ALUs
- 64 Double-precision
- 32 Load/store
- 32 Special Functional Unit
- 64 KB L1/Shared Cache
  - Shared cache is software-managed cache

CPU vs GPU

Parameter	**CPU**	**GPU**
Clockspeed	$>$ 1 GHz	700 MHz
RAM	GB to TB	12 GB (max)
Memory B/W	60 GB/s	$>$ 300 GB/s
Peak FP	$<$ 1 TFlop	$>$ 1 TFlop
Concurrent Threads	O(10)	O(1000) [O(10000)]
LLC cache size	$>$ 100MB (L3) [eDRAM] O(10) [traditional]	$<$ 2MB (L2)
Cache size per thread	O(1 MB)	O(10 bytes)
Software-managed cache	None	48KB/SMX
Type	OOO superscalar	2-way Inorder superscalar

Using a GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.
- Working set must fit in GPU RAM
- You must copy data to/from GPU RAM
  - "You": Programmer, Compiler, Runtime, OS, etc.
  - Some recent hardware can do this for you
- Data accesses should be streaming
  - Or use scratchpad as user-managed cache
- Lots of parallelism preferred (throughput, not latency)
- SIMD-style parallelism best suited
  - Same instruction, different data
- High arithmetic intensity (FLOPs/byte) preferred

Showcase GPU Applications

- Image Processing
- Graphics Rendering
- Matrix Multiply
- FFT

See "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU" by V.W.Lee et al. for more examples and a comparison of CPU and GPU.

Outline

Hierarchy of GPU Programming Models

Model	**GPU**	**CPU Equivalent**
Vectorizing Compiler	PGI CUDA Fortran	gcc, icc, etc.
"Drop-in" Libraries	cuBLAS	ATLAS
Directive-driven	OpenACC, OpenMP-to-CUDA	OpenMP
High-level languages	pyCUDA	python
Mid-level languages	OpenCL, CUDA	pthreads + C/C++
Low-level languages	PTX, Shader	-
Bare-metal	SASS	Assembly/Machine code

“Drop-in” Libraries

- “Drop-in” replacements for popular CPU libraries, examples from NVIDIA:
  - CUBLAS/NVBLAS for BLAS (e.g. ATLAS)
  - CUFFT for FFTW
  - MAGMA for LAPACK and BLAS
- These libraries may still expect you to manage data transfers manually
- Libraries may support multiple accelerators (GPU + CPU + Xeon Phi)

GPU Libraries

- NVIDIA Thrust
  - Like C++ STL, but executes on the GPU
- Modern GPU
  - At first glance: high-performance library routines for sorting, searching, reductions, etc.
  - A deeper look: Specific "hard" problems tackled in a different style
- NVIDIA CUB
  - Low-level primitives for use in CUDA kernels

Directive-Driven Programming

- OpenACC, new standard for "offloading" parallel work to an accelerator
  - Currently supported only by PGI Accelerator compiler
  - gcc 5.0 support is ongoing
- OpenMPC, a research compiler, can compile OpenMP code + extra directives to CUDA
  - OpenMP 4.0 also supports offload to accelerators, including to GPUs

```
int main(void) {
  double pi = 0.0f; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++) {
     double t= (double)((i+0.5)/N);
     pi +=4.0/(1.0+t*t);
  }

  printf("pi=%16.15f\n",pi/N);
  return 0;
}
```

Python-based Tools (pyCUDA)

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule(""\"
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""\")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)

multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

OpenCL

- C99-based dialect for programming heterogenous systems
  - Originally based on CUDA
  - nomenclature is different
- Supported by more than GPUs
  - Xeon Phi, FPGAs, CPUs, etc.
- Source code is portable (somewhat)
  - Performance may not be!
- Poorly supported by NVIDIA

CUDA

- "Compute Unified Device Architecture"
- First language to allow general-purpose programming for GPUs
  - preceded by shader languages
- Promoted by NVIDIA for their GPUs
- Not supported by any other accelerator
  - though commercial CUDA-to-x86/64 compilers exist
- We will focus on CUDA programs

CUDA Architecture

- From 10000 feet – CUDA is like pthreads
  - CUDA language – C++ dialect
- Host code (CPU) and GPU code in same file
- Special language extensions for GPU code
- CUDA Runtime API
  - Manages runtime GPU environment
  - Allocation of memory, data transfers, synchronization with GPU, etc.
  - Usually invoked by host code
- CUDA Device API
  - Lower-level API that CUDA Runtime API is built upon

CUDA Limitations

- No standard library for GPU functions
- No parallel data structures
- No synchronization primitives (mutex, semaphores, queues, etc.)
  - you can roll your own
  - only atomic*() functions provided
- Toolchain not as mature as CPU toolchain
  - Felt intensely in performance debugging
- It's only been a decade :)

Conclusions

- GPUs are very interesting parallel machines
- They're not going away
  - Xeon Phi was interesting, but Intel has abandoned it
- They're here and now
  - Your laptop probably already contains one
  - Your phone definitely has one

Outline

References

- NVIDIA CUDA
  - `http://docs.nvidia.com`
  - Start with the CUDA C++ Programming Guide
- OpenCL
  - `https://www.khronos.org/opencl/`
  - The OpenCL C 2.0 Specification
  - The OpenCL C++ 1.0 Specification