GPU Co-processing

(aka. Offload computing or offloading)

((aka. Accelerator computing))

How do we use GPUs for HPC?

- GPUs are throughput-optimized devices originally targeting graphics (hence, the name).
- But they are great for processing massive, structured data that's easily parallelizable.
- … doing some cool rendering operation on each pixel looks similar doing something cool to
  - Each element of a matrix.
  - Each particle of an n-body system.
  - …

What'd different about GPUs and CPUs?

- Even on multi-core CPU systems, the cores are still pretty powerful.
  - The most we'll see is < 100 cores with a few 100 hardware threads
- GPUs, on the other hand, have LOTS (1,000's) of very tiny "cores" with very low single-thread performance.
  - "core" is in quotes … there is some disagreement about what a core really is.
- GPUs have high (total) memory bandwidth to support 1,000's of cores at once.
  - Specialized read-only constant and texture memories for special operations.
  - Generic (graphic) DRAM for everything else. Still optimized for structured data.

CUDA

- Compute Unified Device Architecture (CUDA) is NVIDIA's GPU platform.
  - Both an architecture and (now) the name of C++-like language.
- NVCC is the C++ compiler provided by NVIDIA but LVMM also can produce CUDA code now.
- OpenACC is an OpenMP-like directives language that can produce CUDA code for C/C++ and Fortran applications (very handy).

CUDA

- *Thread*: serial replication of your kernel (10,000's)
- *Warp*: tightly-bound group of threads that execute in lock-step (32) in single instruction multiple thread (SIMT) paradigm, similar to SIMD.
  - Also called a CTA – *cooperative thread array*
- *ThreadBlock*: (sort of) tightly-bound group of warps that execute on the same *streaming multiprocessor* (SM), can communicate / synchronize amongst themselves, and share a scratchpad (10's KB)
  - Threads in a ThreadBlock can be assigned an x/y/z coordinate
- *Grid*: Collection of ThreadBlocks assigned an x/y/z coordinate. (1-10) *ThreadBlocks* are scattered to the SM's but do not migrate.

CUDA

- Lots of threads run over the entire device but they can’t communicate outside of their own ThreadBlock.
  - This avoids a lot of hardware requirements and keeps things simple.
- Only global synchronization is when a ‘kernel’ (a device function) terminates.
- Global atomic operations supported on modern GPUs which makes global data sharing much (much!) easier than it used to be.
- Threads in a warp execute in lock-step. Instructions are issued to warps, not threads!
  - Divergence occurs when sibling threads branch down different paths. Execution is (essentially) serialized across all siblings.

CUDA

- All code and data starts on the host and must be transferred to the device … a huge bottleneck.
- But Device kernels run asynchronously so we can overlap host-device communication, host computation, and device computation.
  - That’s how you maximum throughput but it’s difficult.
  - All CUDA operations (transfers and kernels) are queued and can be assigned to different ‘streams’ (task dependencies in the queue). This helps schedule concurrent transfers and host/device computations.

OpenCL

- CUDA is proprietary and limited to NVIDIA devices.
- OpenCL (Open Compute Language) can support other accelerator systems such as AMD GPUs, host CPUs, and other (exotic) platforms (e.g., FPGA's).
- Different syntax but very similar design as CUDA.
  - Does have extra support for SIMD execution which is handy.
  - But, OpenCL adoption is not great and the future isn't clear since NVIDIA is dominating the GPU (accelerator) market.