Parallelization using Multiple GPUs on Titan

W. Michael Brown

Titan Users and Developers Workshop (West Coast)

January 31, 2013

Outline

1. Compiling/Linking
   - Compiling with MPI, CUDA, OpenCL, OpenACC, Compiler Restrictions
2. Interprocess Communications
   - Asynchronous Communications, Cray XK7 Architecture, Task Mapping
3. Parallelization of host code
   - Parallelization Strategy, MPI Only, Hybrid MPI/OpenMP
   - Cray XK7 Architecture, Task Mapping Revisited
4. Running
   - Sharing a GPU between multiple processes/threads with a single context
   - Environment Variables, Task Mapping Revisited
   - Dynamic Linking, DVS
5. Summary

Parallelization using Multiple GPUs on Titan

1. Compiling and Building

CAPS-MC OpenACC with MPI

- *formerly known as HMPP*
  - Load capsmc module
  - Use Cray PE Wrappers
  - C and Fortran, basic/limited C++
  - Works as preprocessor to compilers from all Cray PE vendors

```
> module load cudatoolkit
> module load capsmc

> module load PrgEnv-pgi
> # or
> module swap PrgEnv-pgi PrgEnv-gnu
> # or
> module swap PrgEnv-pgi PrgEnv-cray
> # or
> module swap PrgEnv-pgi PrgEnv-intel

> cc foo.c -o foo
> # or
> ftn foo.f90 -o foo
> # or
> CC foo.cpp -o foo
```

OpenACC with MPI

- … for PGI and Cray Compilers
  - Use Cray PE Wrappers
  - C and Fortran, no C++
  - Load the cudatoolkit module
  - Load craype-accel-nvidia35 or capsmc modules for Cray or CAPSMC/HMPP
  - Add –acc, –h pragma=acc, or –h acc for PGI and Cray builds

```
> module load cudatoolkit

> # PGI OpenACC
> module load PrgEnv-pgi
> cc -acc foo.c -o foo
> # or
> ftn -acc foo.f90 -o foo

> # Cray OpenACC
> module swap PrgEnv-pgi PrgEnv-cray
> module load craype-accel-nvidia35
> cc -h pragma=acc foo.c -o foo
> # or
> ftn -h acc foo.f90 -o foo
```

CUDA Runtime with MPI

- Must use GNU PE
  - ... or link with code compiled with GNU and from another PE
- C and C++
- Load the cudatoolkit module
- Use the --compile-bindir flag with nvcc so that the Cray wrappers are used for host code and the appropriate includes for MPI are used
- Compile everything with nvcc or if there are problems or C code is used, compile just the .cu files with nvcc

```
> module load cudatoolkit
> module swap PrgEnv-pgi PrgEnv-gnu
> nvcc --compile-bindir `which CC`
foo.cpp -o foo
```

Note for C code: .cu files are compiled with C++ name mangling using nvcc. In some cases, all code can be compiled with C++ compiler, otherwise, extern "C" should be used in the .cu file for functions called by C objects.

CUDA Driver with MPI

- Can use any PE/Compiler
- C and C++
- Only the kernels are compiled with nvcc
- Kernels are managed in the C/C++ code as strings/files

CUDA Fortran with MPI

- Use PrgEnv-pgi PE with ftn wrapper.

OpenCL with MPI

- Can use the GNU, Intel, and PGI programming environments
- C and C++
- Load the cudatoolkit module
- Link with -lOpenCL

```
> module load cudatoolkit

> module load PrgEnv-pgi
> # or
> module swap PrgEnv-pgi PrgEnv-gnu
> # or
> module swap PrgEnv-pgi PrgEnv-intel

> cc foo.c -o foo -lOpenCL
> # or
> CC foo.cpp -o foo -lOpenCL
```

Compiler Support

	CAPS-MC			Open-ACC			CUDA Runtime			CUDA Driver			CUDA Fortran			OpenCL		
	CC	cc	ftn	CC	cc	ftn	CC	cc	ftn	CC	cc	ftn	CC	cc	ftn	CC	cc	ftn
PE-pgi		■	■		■	■				■	■				■	■	■	
PE-gnu		■	■				■	■		■	■					■	■	
PE-cray		■	■		■	■				■	■							
PE-intel		■	■							■	■					■	■	

Note: By isolating GPU kernels/memory calls to separate files, unsupported PE/compilers can be used for the remaining host code that is linked with the GPU code.

Note: Some basic/limited supported for OpenACC with C++ might be available with CAPS-MC, Cray, and PGI – check with OLCF help.

Note: Some additional options might be available soon, e.g. OpenCL with Cray Compiler.

OAK RIDGE National Laboratory

OLCF Oak Ridge Leadership Computing Facility

Parallelization using Multiple GPUs on Titan

2. Interprocess Communications

Use Asynchronous Calls and Pipelining

- ~~Peer-2-Peer~~ ← 1 GPU per Node
- ~~GPU-Direct~~ ← Not currently available
- GPU-Aware MPI ← Available soon
- Prepost asynchronous receives
- Overlap communications with GPU transfers/kernels where possible
  - Think about options depending on your code
    - Compute results needed for interprocess communication first in a separate kernel call?
      - Split into a pipeline of multiple asynchronous kernel calls/data transfers?
    - Compute results needed for interprocess communication concurrently on the host without the need for GPU data transfer/synchronization?
    - …
  - There are some tricks that do not require code modification
    - Multiple MPI processes sharing a GPU can be used to pipeline GPU transfers/kernel executions/MPI when data becomes available *(more later)*

GPU-Aware MPI

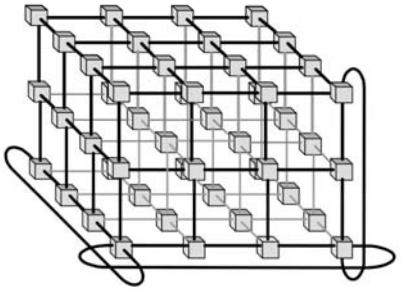
- Available in MPT 5.6.1
  - Use pointers to memory on the accelerator directly as the MPI send/receive buffers
  - MPI implementation takes care of the optimization
    - Pipelining large messages
    - GPU Direct
    - P2P
    - etc.
  - Non-blocking MPI calls can be used to overlap communication and computation

Process Mapping

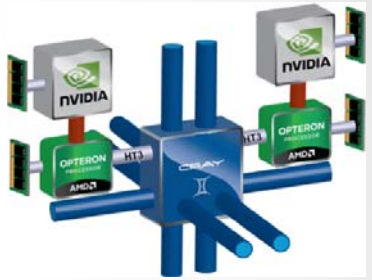
- Job placement can have a significant impact on MPI communication times
- MPI routines for mapping communication topologies onto the network are currently very unsophisticated
- Advanced users/developers might want to consider handling process mapping to nodes explicitly
  - In general case, a NP-complete problem but active area of research and developers have seen improvements by using heuristics for process mappings in their runs
  - CrayPAT can provide insight for some topologies for your code
  - MPI process mapping can be altered, *in many cases*, without changing the code by using a file with a rank ordering, MPICH_RANK_ORDER, and setting the MPICH_RANK_REORDER_METHOD environment variable

Understanding Gemini

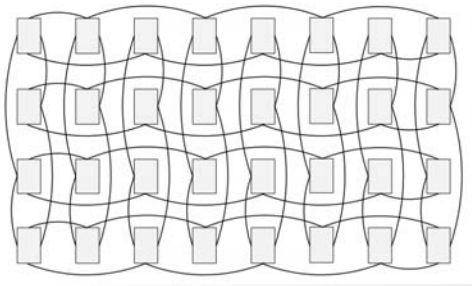
- The network on Titan is called Gemini - connected in a 3D Torus
  - Each vertex on the torus has an X,Y,Z coordinate
  - 1 Gemini ASIC per vertex connected to 2 nodes
    - <1µs latency for messages on vertex
  - There are physical links between a vertex and it neighbors at $(x\pm1,y,z)$, $(x,y\pm1,z)$, $(x,y,z\pm1)$
    - 1.5µs latency for messages across single physical link
    - No edges or center; (0,0,0) has physical link to $(N_X, 0, 0)$, etc.
  - Links in the Y direction have half of the bandwidth of the X/Z
  - Minimize message hops based on the *Torus coordinates*, not based on node name or physical location (*serpentine link patterns, etc.)*
    - *LINK TO TITAN TORUS COORDINATES HERE*



3D Torus - 1 Gemini ASIC per Vertex



XK7 Architecture - Each Gemini ASIC has 2 NICs connected to 2 Opterons; half bandwidth for *y* links



Inter-cabinet cabling for 32 cabinets in 4 rows

Parallelization using Multiple GPUs on Titan

3. Parallelization of Host Code

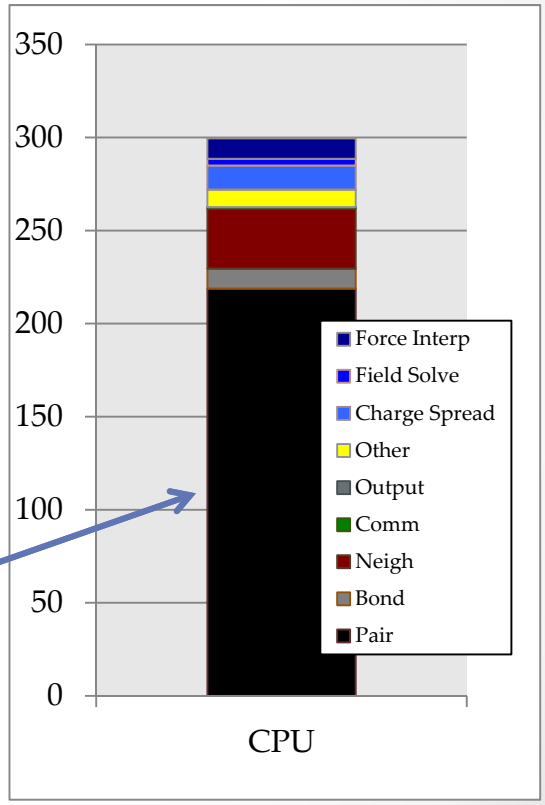
Where to Begin?

1. Get a profile to identify targets for acceleration for your code
2. Understand how to take advantage of existing parallelism in your code on the host (OpenMP/MPI) and how concurrent host/device execution and data movement can be used to help the porting process
3. Port target routines to the accelerator
4. Run and analyze performance
5. Work towards increasing the amount of code running on the accelerator and improving concurrency by exploiting fine grain parallelism, task-based parallelism, etc.
6. Investigate new parameters/algorithms/simulation models that might perform better on Titan and future architectures

Profile Code

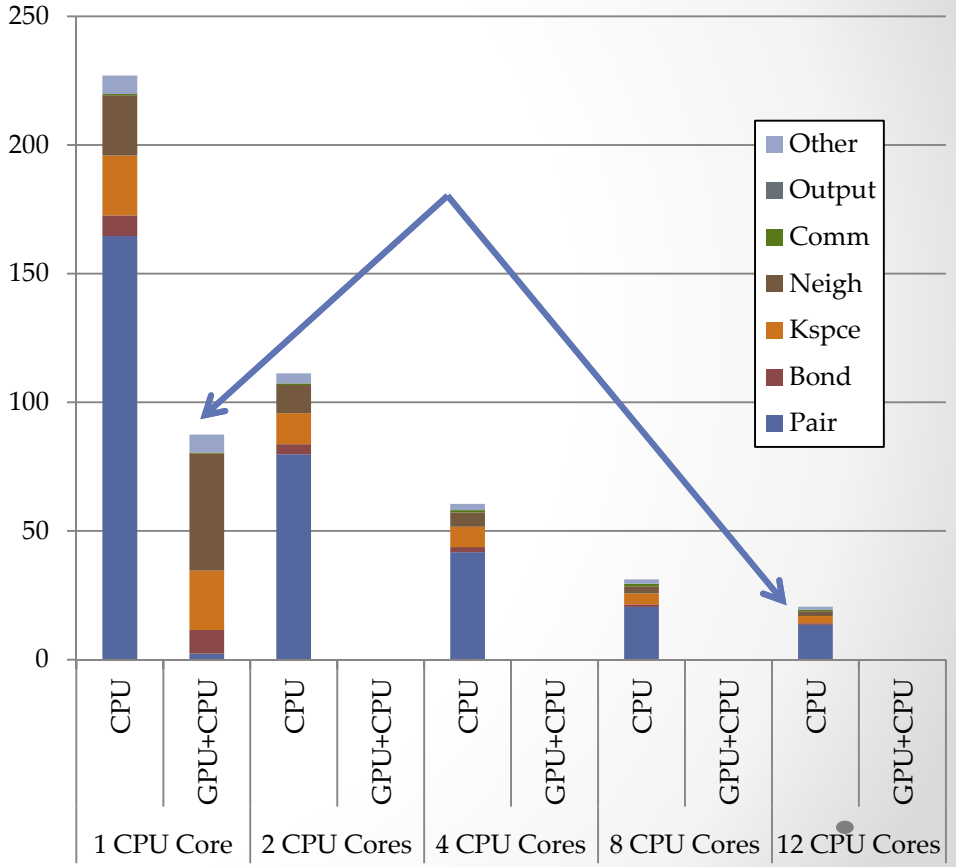
- Figure out what routines are using the most time with
  - CrayPAT, Vampir
  - Manual instrumentation
  - …

"Pair" time: 73% of simulation loop; also, the computation performed here is a good candidate for GPU acceleration ☺



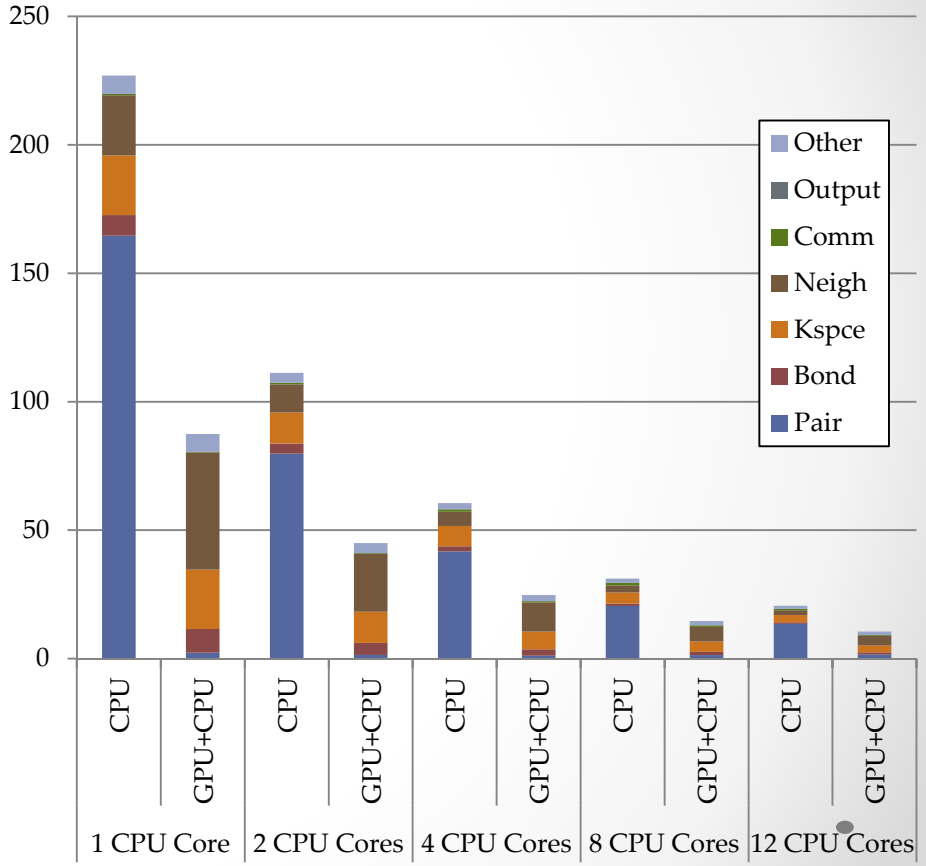
Amdahl's Law

- Example with (now) old hardware/software stack
  - Two 6-core AMD Istanbul Processors
  - Tesla C2070 "Fermi" Accelerators
- > 50x speedup for "Pair" routine with acceleration (using mixed precision)
- Running on a CPU core with acceleration is still slower than CPU-only on all 12 cores



Must "Deal" with the other Routines

- Port to the accelerator?
  - This is the preference if it is feasible and is expected to have good performance on the accelerator
- Parallelize on the host?
  - This is already done, so potentially a good place to start
  - In this example, we still get > 2X speedup with only a single routine ported to the accelerator
    - In this example, all MPI processes share the GPUs
    - Performance on newer hardware/driver for GPU sharing is much better (hyper-q, proxy, etc.)



Options for Parallelization

- 1 MPI Process/Host Thread per GPU
  - Best option if you can efficiently run most of your code on the accelerator and hide the time spent on the remaining code running on the host with concurrent GPU work
- 1 MPI Process/Multiple Threads per GPU
  - Use OpenMP/pthreads to parallelize host code
    - All threads share the accelerator with separate kernel calls/memcopy
      - ... or ...
    - Only the master thread accesses the accelerator with shared page-locked memory to allow multiple threads to provide input and access results
  - Good option if the code...
    - ...already uses hybrid MPI/OpenMP parallelization on the host
    - ...is amenable to shared memory parallelism without significant pain
      - Memory conflicts/code complexity can offset the potential advantages of using OpenMP with MPI

Options for Parallelization

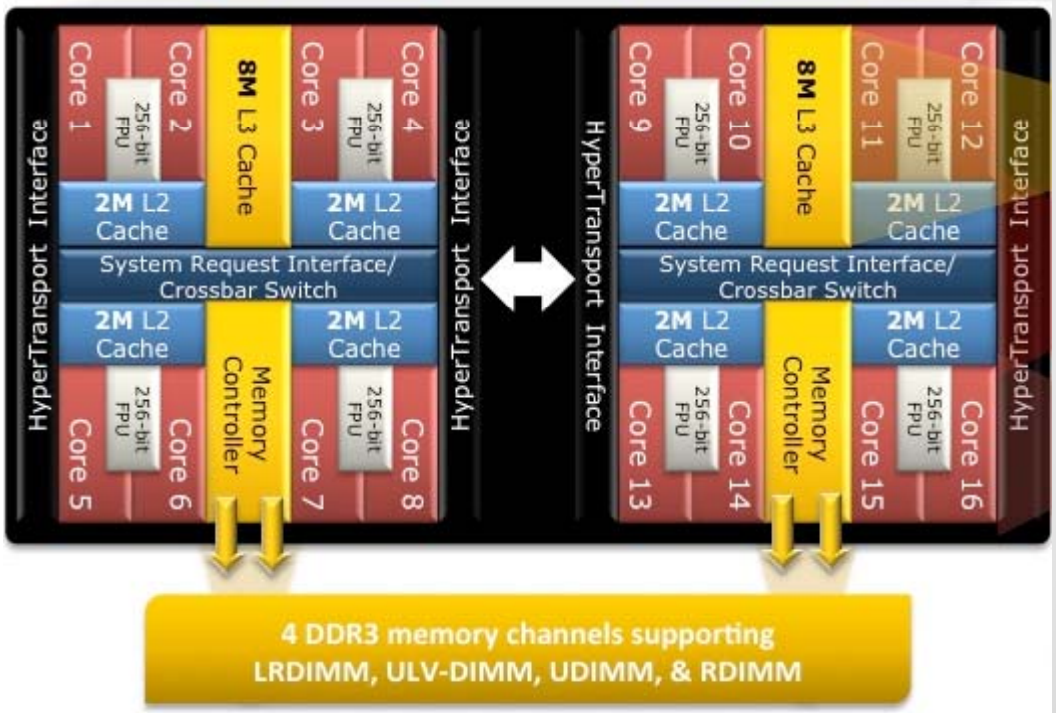
- Multiple MPI Processes per GPU
  - Easiest option – no or little changes to code, just run with multiple MPI processes per GPU
  - Some other advantages to this approach:
    - Code can potentially run simultaneously on GPU and CPU for independent calculations
    - Task-based parallelism might be more straightforward with MPI versus threads
    - Smaller memory footprint per process can improve performance on both the host and the accelerator for some routines (e.g. random memory access with a domain decomposition)
    - Allows potential for pipelining of MPI, memcopies, and computation as data is available without modifying the code
      - This is assumes that the software stack is smart enough to do this right and that the timing supports this
  - Disadvantages include:
    - This can limit strong scaling – don't want the chunks of work for the accelerator to be too small
    - Increases the number of processes doing MPI on a node
    - Accelerator vendors don't have a lot of experience with this usage model

Code Snippet

- Getting per-accelerator MPI subcommunicators and assigning an accelerator to each process
  - *LINK to code example will be provided here*

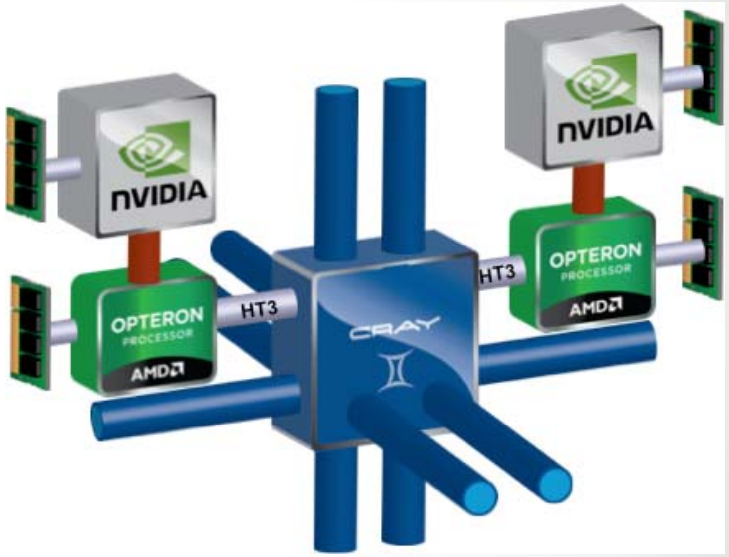
Task Mapping Revisited

- The AMD Opteron 6200 (Interlagos) Chip has two 8-core dies to form a multi-chip module
- 2 NUMA Nodes per Socket
  - Accessing memory allocated on one die is much slower from the other die
  - Often better to use at least 2 MPI processes per node unless the code is written at the thread level to handle NUMA appropriately with careful memory allocation/initialization



Task Mapping Revisited

- Using multiple MPI processes per node can impact overall communications performance for your code
- Remapping MPI tasks to minimize off-node communications can improve performance
  - Example: Remapping to minimize the per-node surface to volume ratio for a spatial decomposition
    - *LINK will be provided here*



OAK RIDGE National Laboratory

OLCF Oak Ridge Leadership Computing Facility

Parallelization using Multiple GPUs on Titan

4. Running

The CUDA Proxy Server

- The Proxy server is **required** for multiple CPU processes on a single node to share a GPU on Titan
  - The accelerators on Titan are set to "Exclusive Process" mode
- Proxy is a client-server architecture that allows:
  - Multiple processes to share the GPU when it is in "Exclusive Process" mode – currently the only supported mode on Titan
  - Multiple processes to share a context on the GPU so that
    - Context switching overhead is eliminated
    - Kernel and memcpy execution from different processes may overlap and execute concurrently (Hyper-Q)
    - Resource requirements reduced (e.g. memory overhead) from multiple contexts

Using Proxy

- Enabling the Proxy Server
  - `export CRAY_CUDA_PROXY=1`
    - ... before the aprun in your batch or interactive job
  - Do not enable if you don't need it
- Notes when running with proxy
  - There is no memory protection for allocations on the GPU between different MPI processes (one MPI process can access GPU memory allocated by another process)
    - Exploiting this to share GPU memory across processes is *not* a good programming practice
  - A GPU exception within one process will terminate the other processes using the same proxy server

Proxy Limitations

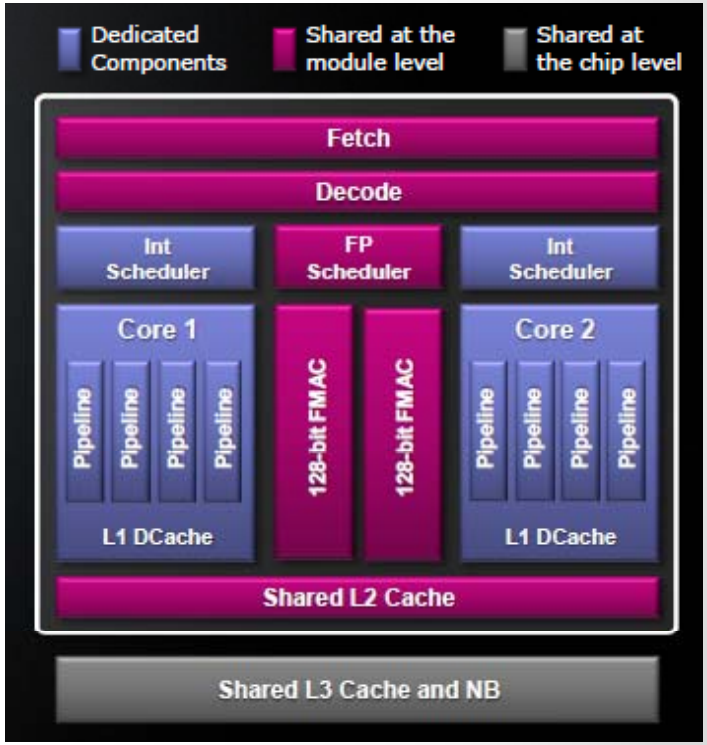
- Debugging and profiling is not available when using proxy
  - Turn off proxy and run in parallel with 1 process per node
    - This is sufficient for identifying *most* bugs
- Increased GPU kernel launch and memcopy initiation overhead
  - Launch overhead with proxy enabled should be on the order of 15-25µs
- GPU assert()/printf() is not currently available with proxy
- Dynamic parallelism is not currently available with proxy

Dynamic Linking

- Notes on building with acceleration
  - Building any accelerated code on Titan requires dynamic linking
  - Titan nodes are diskless
  - Use of dynamic linking can cause significant startup overhead at large job sizes when compared to static executables
    - DVS nodes must distribute the shared libraries to all of the nodes
  - Due to run-time loading from shared libraries, it might be necessary to use a "warm-up" loop before doing timings for performance measurement

Task Mapping (Revisited)

- It might be more efficient to use less than all available cores per node to:
  - Avoid dividing GPU work into chunks that are too small
  - Improve floating point performance
- Cores on the Interlagos CPU are organized into pairs called "Modules" or "Compute Unit"
  - Modules share instruction fetch and 256-bit floating point resource
  - A single core can make use of the entire floating point resource with 256-bit AVX instructions
- Know about the -N, -S, and -d options to aprun
- To run using only 1 process or thread per module, use -j 1 with aprun.



Examples

- For 2 Node Job
  - -n → Number of processes
  - -N → Number of processes per node
  - -S → Number of processes per NUMA node
  - -d → Number of threads per process
  - $OMP_NUM_THREADS → Number of OpenMP threads per process
  - $CRAY_CUDA_PROXY → 1 to allow multiple processes to share GPU

```
# MPI-Only, 1 per GPU
> aprun -n 2 -N 1 ./foo

# MPI/OpenMP, 1 MPI per GPU, 8 threads per,
# 1 thread per bulldozer module
> OMP_NUM_THREADS=8 aprun -n 2 -d 8 -j 1 ./foo

# MPI/OpenMP, 2 MPI per GPU, 4 threads per,
# 1 thread per bulldozer module
> export CRAY_CUDA_PROXY=1
> OMP_NUM_THREADS=4 aprun -n 4 -d 4 -j 1 ./foo

# MPI-Only, 4 per GPU, 1 process per module
> export CRAY_CUDA_PROXY=1
> aprun -n 8 -N 4 -j 1 ./foo
```

Summary

- Depending on your choice of syntax/programming-model for accelerating your code, there are currently restrictions on the compilers you can use.
- Exploring/testing different task mapping options can improve performance for accelerated codes
- Methods to parallelize code on both the host and the accelerator can be useful in porting legacy codes and in some cases, even preferable for new software
- Sharing the accelerator using MPI comes with significant limitations on Titan, but in general will perform much better than on previous GPUs and driver versions