Intro: Using CUDA on Multiple GPUs Concurrently

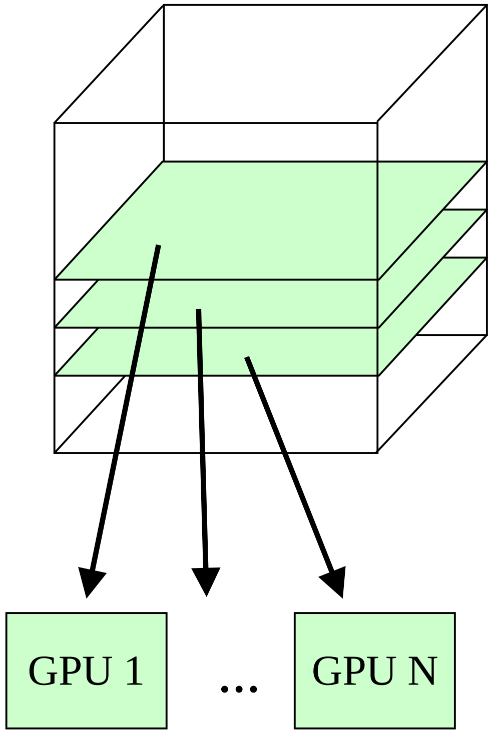
John Stone

IACAT Brown Bag 2/24/2009

Overview

- Some use case examples
- Brief overview of CUDA architecture
- Selecting GPU devices
- Creating multiple host threads/processes to manage GPUs
- Managing work on multiple GPUs
- Handling exceptions

Multi-GPU Direct Coulomb Summation



NCSA GPU Cluster
http://www.ncsa.uiuc.edu/Projects/GPUcluster/

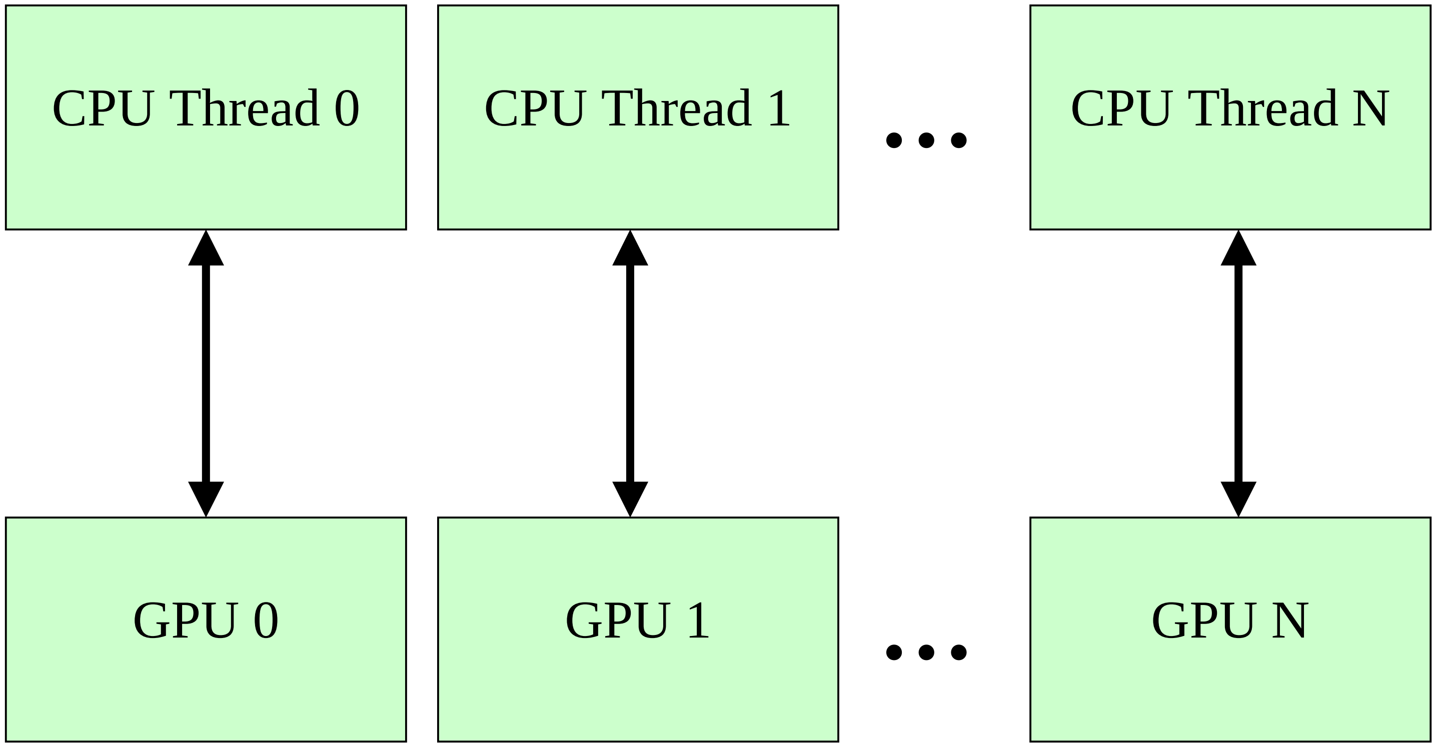
	Evals/sec	TFLOPS	Speedup*
4-GPU (2 Quadroplex) Opteron node at NCSA	157 billion	1.16	176
4-GPU GTX 280 (GT200)	241 billion	1.78	271

*Speedups relative to Intel QX6700 CPU core w/ SSE

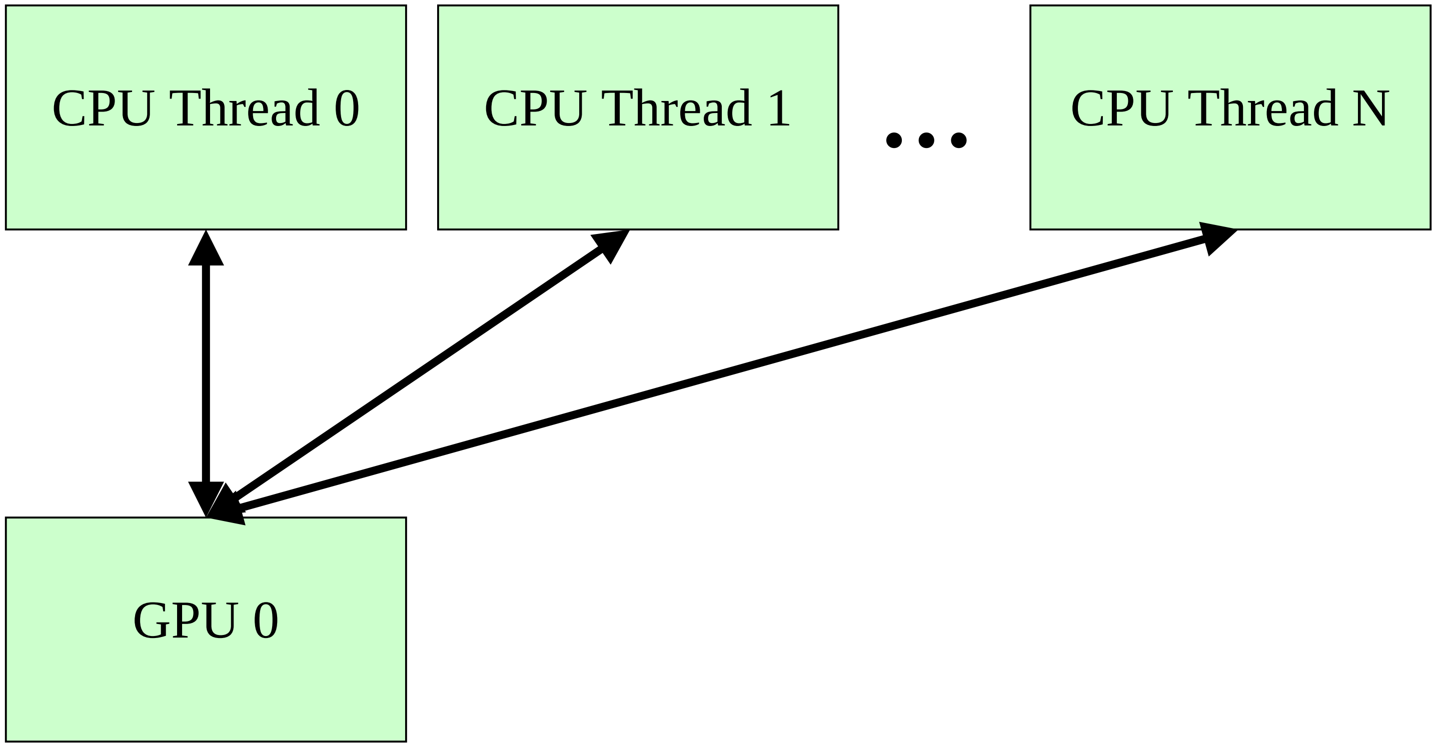
CUDA Architecture Basics

- A single host thread can attach to and communicate with a single GPU
- A single GPU can be shared by multiple threads/processes, but only one such context is active at a time
- In order to use more than one GPU, multiple host threads or processes must be created

One Host Thread Per GPU



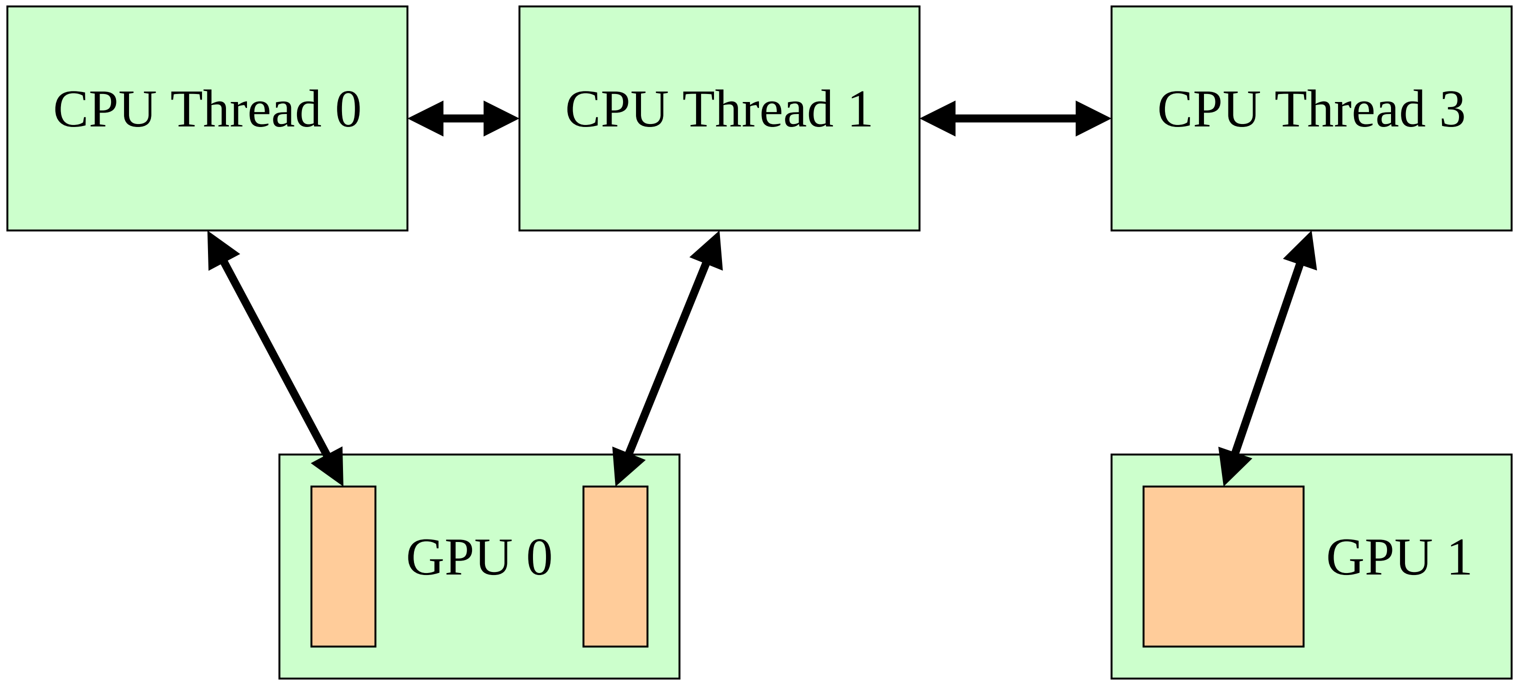
Multiple Host Threads Per GPU



Data Exchange Between GPUs

- Limitations with current version of CUDA:
  - No way to directly exchange data between multiple GPUs using CUDA
  - Exchanges must be done on the host side outside of CUDA
  - Involves host thread/process responsible for each device

Host Thread Contexts Cannot Directly Share GPU Memory, Must Communicate on Host Side



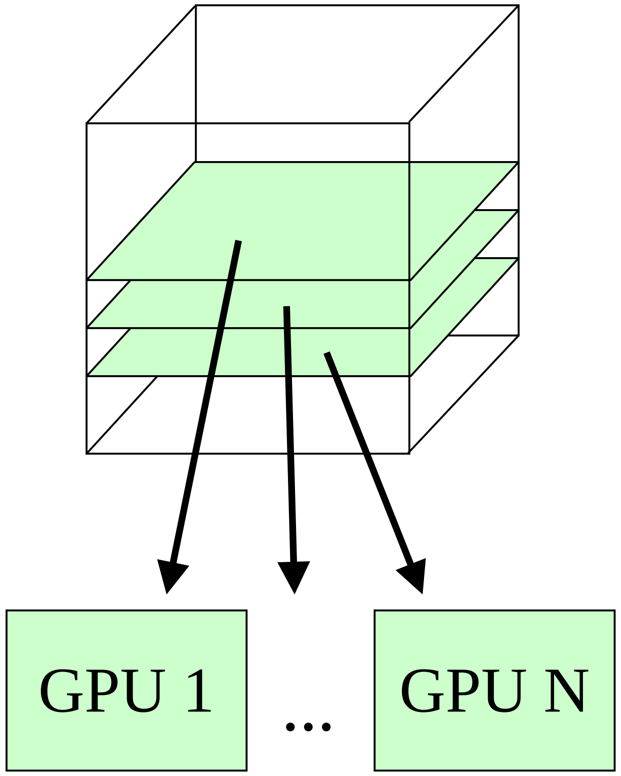
Even threads sharing the same GPU cannot exchange data by reading each other's GPU memory

CUDA Runtime APIs for Enumerating and Selecting GPU Devices

- Query available hardware:
  - cudaGetDeviceCount()
  - cudaGetDeviceProperties()
- Attach a GPU device to a host thread:
  - cudaSetDevice()
  - This is a permanent binding, once set it cannot be subsequently changed
  - Binding a GPU device to a host thread has overhead:
    - 1st CUDA call after binding takes ~100 milliseconds

Multi-GPU Data-parallel Decomposition

- Many independent coarse-grain computations farmed out to pool of GPUs
- Work assignment can be explicit in the code, or controlled with a dynamic work scheduler of some sort
- May need to handle load imbalance, GPUs with varying capabilities, runtime errors, etc.



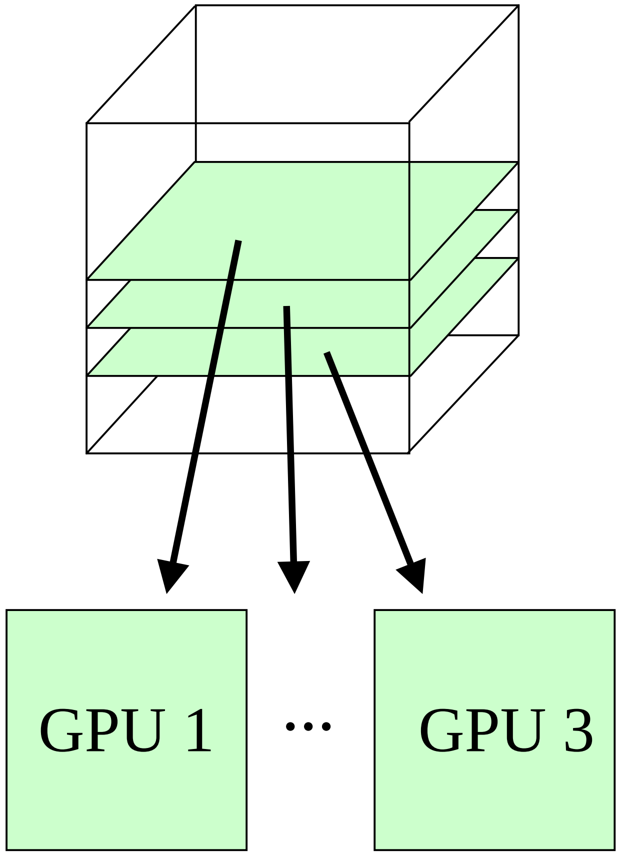
Launching Host Threads (POSIX Threads)

```
void *cudaworkerthread(void *voidparms); // worker function
…
/* spawn child threads to do the work */
for (i=0; i<numprocs; i++) {
  pthread_create(&threads[i], cudaworkerthread, &parms[i]);
}

/* "join" the threads after work is done */
for (i=0; i<numprocs; i++)
  pthread_join(threads[i], NULL);
}
```

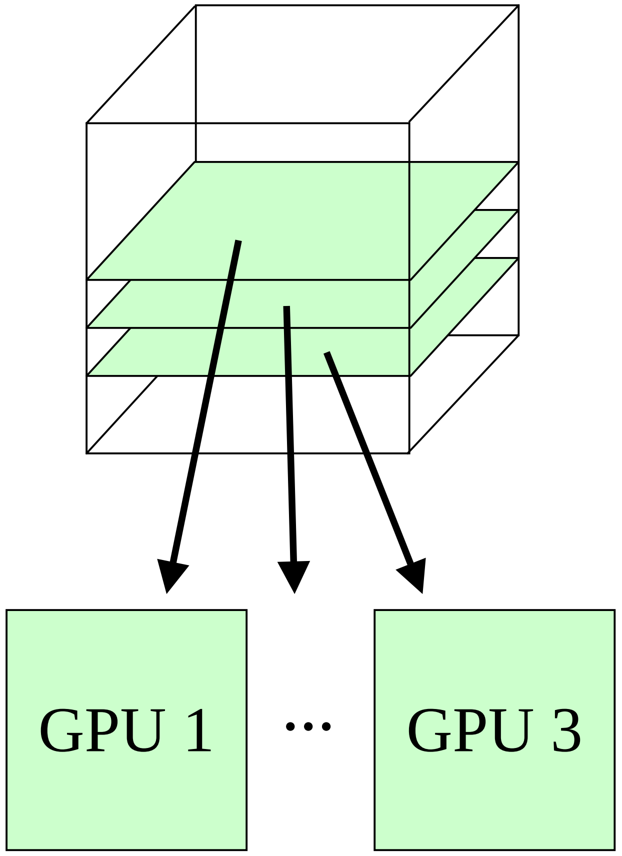
Multi-GPU Static Load Balance, Static Work Decomposition

- Static round-robin load balance:
  - Easy to code, explicit round robin decomposition
  - Low overhead, works well for short calculation runs
  - No ability to automatically reschedule work on error/exception
  - Easy to port to multiple OSs



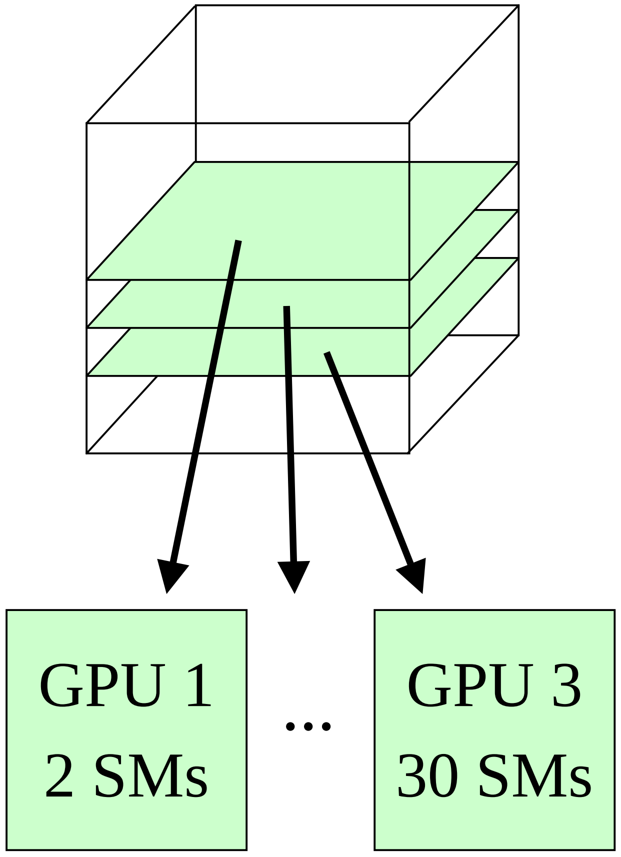
Multi-GPU
Static Work Decomposition

```
// Each GPU worker thread loops over
// subset of 2-D planes in a 3-D cube…
for (k=thrID; k<numplane; k+=thrCount) {
  // Process one plane of work…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Simple scheme, works well when GPUs
  //   and work units are nearly identical…
  // No provision for in-flight error handling
}
```



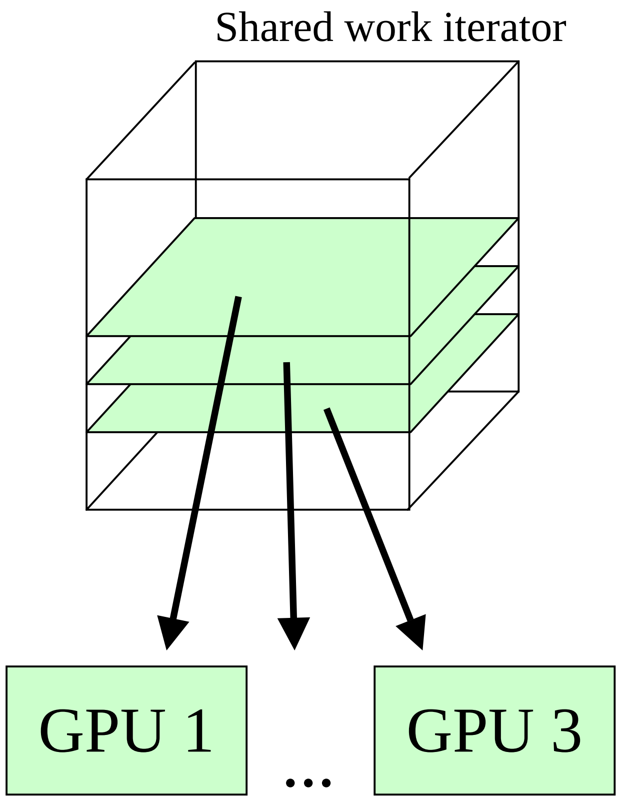
Multi-GPU Load Balance

- Many independent coarse-grain computations farmed out to pool of GPUs
- Many early CUDA codes assumed all GPUs were identical (nearly so)
- Now all new NV cards support CUDA, so a machine may have a diversity of GPUs of varying capability
- Static decomposition works poorly if you have diverse GPUs, e.g. 2 SM, 30 SM



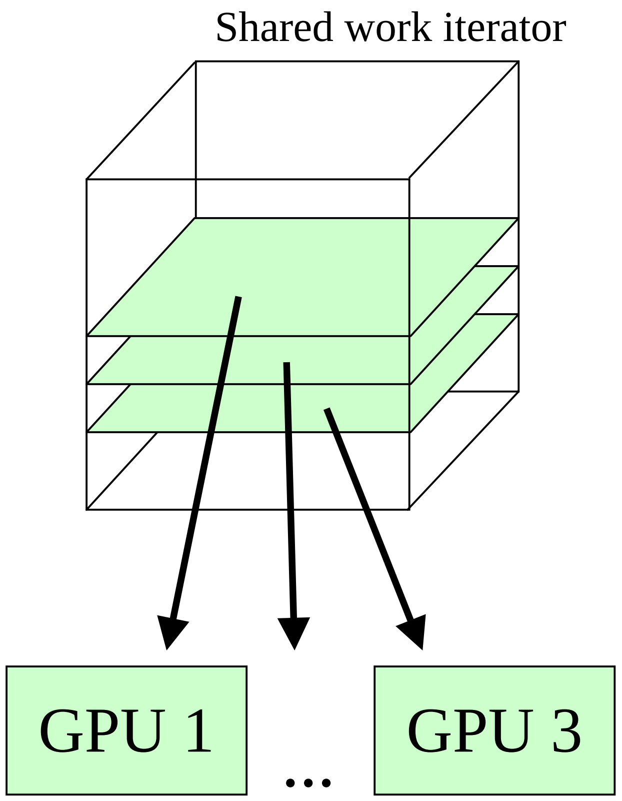
Multi-GPU Dynamic Load Balance, Shared Work Iterator

- Dynamic load balance, single shared iterator assigns slices to workers:
  - Replaces the **for** loop in static decomposition example
  - Added overhead from mutex locks:
  - Can reschedule/retry on error/exception by re-adding to the shared queue
  - Still easy to port to multiple OSs



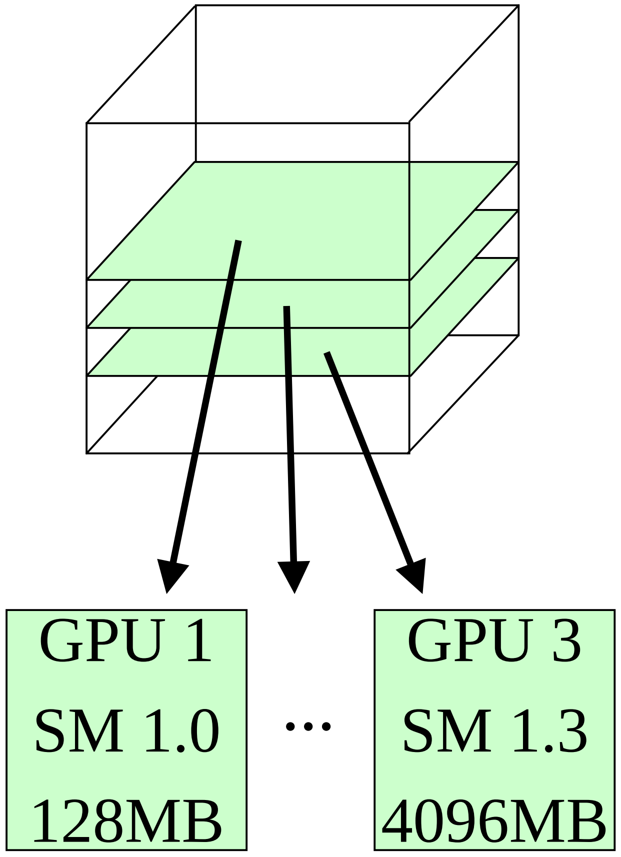
Multi-GPU Shared Work Iterator

```
// Each GPU worker thread loops over
// subset 2-D planes in a 3-D cube…
while (!iterator_next(&parms, &k) {
  // Process one plane of work…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Shared iterator automatically
  //   balances load on GPUs
  // No provision for complex error handling
  //   or “retry” of a work unit on a different GPU
}
```



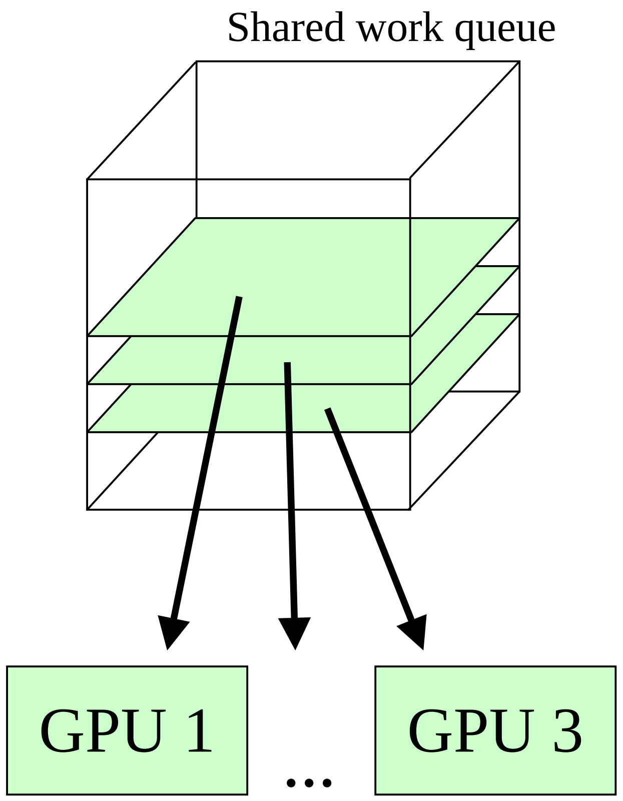
Multi-GPU Runtime Error/Exception Handling

- Competition for resources from other applications or the windowing system can cause runtime failures (e.g. GPU out of memory half way through an algorithm)
- Handling of algorithm exceptions (e.g. convergence failure, NaN result, etc)
- Need a way to handle and/or reschedule failed tiles of work



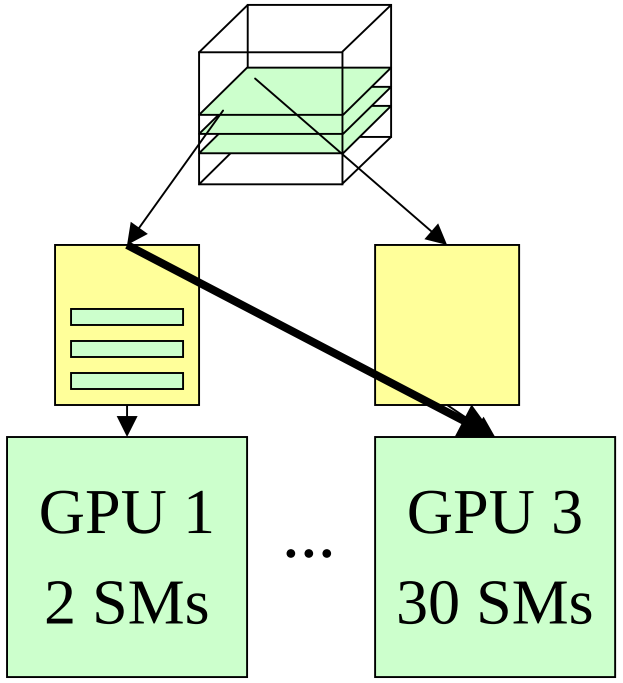
Multi-GPU Load Balance, Shared Work Queue

- Dynamic load balance, single shared work queue:
  - Added overhead from loading/draining queue
  - Potential for mutex contention in fast running kernels or fine-grained work decomposition
  - Can reschedule/retry on error/exception by re-adding to the shared queue
  - Still relatively easy to port to multiple OSs
  - Harder to make fastest implementations portable since they ideally use lock-free algorithms (e.g. STM)



Multi-GPU Load Balance, Multiple Deques, Work Stealing…

- Dynamic load balance, multiple work deques, "work stealing"
  - Added overhead from loading/draining managing multiple double-ended queues
  - Reduced mutex contention in fast running kernels since mutexes only contended during "work stealing"
  - Can reschedule/retry on error/exception by re-adding to the shared queue
  - Harder to make portable, fastest implementations attempt to use lock-free algorithms



Steal Work From Slower Running GPU