Parallel CUDA

Philip Blakely

Laboratory for Scientific Computing, Cambridge

Outline

MPI with GPUs

- This course does not cover MPI itself.
- For more information see the MPI lecture course.
- We don't cover OpenMP or multi-threaded code; I strongly suggest that you don't mix CUDA with either of these.
- General rule for MPI/CUDA: Need one CPU process per GPU.

MPI-GPU compatibility

- Attach one GPU to each process
- Use of a GPU is completely orthogonal to use of MPI
- General approach: Proceed as for a typical MPI application, using GPUs to speed up computation on each processor
- Can take an already parallelised code and use CUDA to accelerate compute-intensive parts
- No extra thought is needed to ensure parallel correctness between CPU-cores.
- But some may be required for optimal performance.

Issues to be aware of

- Is it worth it?
  - Developing a robust parallel code can be tricky
  - May be better to put more effort into GPU optimization
  - Unless the issue is amount of memory
  - How fast is fast enough?
- Speed-up may be less good than for CPU version (see Amdahl's law)
- since by using GPUs, we've reduced the proportion (in time) of the code that can be parallelised
- However, can probably get reasonable speed-ups on small clusters

Device choice

- The number of CUDA cards can be found from `cudaGetDeviceCount(int*)`
- You may often have multiple CUDA cards in a computer and want to choose the one with highest Compute Capability, memory, etc.
- The function `cudaGetDeviceProperties` can be used to ascertain all these properties.
- The information is returned in a `struct` (details in Reference Guide)
- You can then select which card to attach the current process to `cudaSetDevice(N)` to force all subsequent kernels to launch on this card.

Attaching a GPU to a CPU thread

- We only describe the run-time API here.
- The driver API does permit more than one GPU per thread (but not simultaneously)

Initializing GPUs with MPI

```
int numCPUs;
MPI_Comm_size(MPI_COMM_WORLD, &numCPUs);
int numGPUs;
cudaGetDeviceCount(&numGPUs);

int myCPU;
MPI_Comm_rank(MPI_COMM_WORLD, &myCPU);
cudaSetDevice(myCPU);
```

Multiple hosts

- For multiple nodes, you will need to make use of

  ```
  char myName[100];
  int length;
  MPI_Get_processor_name(myName, &length)
  ```

  to distinguish between the CUDA devices on different nodes.
- Mapping directly from MPI rank is unlikely to work robustly, since ranks are not assigned with regard to host.
- You will need to determine the set of MPI processes on each node, and the number of CUDA devices on that node
- Then, determine which CUDA device each process will attach to.

Communication

- Most communication is done via host CPUs
- As with traditional MPI, make sure buffers contain correct data before use
- More important now, when combined with asynchronous GPU functions
- Check MPI and CUDA documentation regarding buffered sends, asynchronous memory copies, etc. very very very carefully
- Just because you have initiated a memory-copy does not mean it has completed.
- It is possible for separate devices on one system to read each other's memory
- See `cudaDeviceEnablePeerAccess` and `cudaMemcpyPeer` and similar for details
- For safety, start off doing copies via host memory and only use peer-to-peer copy for optimization reasons.

Communication optimization

- Use MPI buffered sends and GPU memory copy at same time as computation on GPU
- In theory, this should hide communication latency
- Some HPC manufacturers have setups to increase bandwidth/reduce latency for node interconnect. On cheaper systems, you may find GPU-GPU transfers still go via CPU data-path.

Pinned memory

- You can allocate pinned memory (also known as page-locked memory)
- Copies between device and pinned memory may overlap with computation
- Page-locked memory can be mapped into device memory
- Copying between host-page-locked and device may be faster than regular copying
- Use `cudaMallocHost()` to allocate pinned memory
- or `cudaHostAlloc()` to give memory extra properties
- such as allocating mapped memory - immediately accessible from device using `cudaHostGetDevicePointer()`

Compiling MPI-CUDA applications

- Best approach: Put MPI commands into separate files/classes from CUDA constructs.
- This will help keep the MPI and CUDA functionalities separate in your mind.

Compiling:

- Use `mpic++ -show` to get the set of compiler options to pass to use MPI headers and libraries:

```
g++ -I/usr/lib/openmpi/include
    -I/usr/lib/openmpi/include/openmpi
-pthread -L/usr/lib
-L/usr/lib/openmpi/lib -lmpi_cxx -lmpi -ldl -lhwloc
```

- Append these options to all `nvcc` commands.
- Ensure that MPI implementation uses same `gcc` compiler as `nvcc`; problems may occur if compilers are mixed.

Linking and running MPI-CUDA

Linking: “Including pre-compiled libraries”

```
nvcc -dlink -lcublas driver.o mpiRoutines.o cudaKernels.o
-o myExecutable
```

- Use `nvcc` to link
- MPI: Need `libmpi` and maybe `libmpi_cxx`
- Add `-L/usr/lib/openmpi/lib -lmpi -lmpi_cxx`
- Paths and library names may vary depending on your setup.

For profiling, NVIDIA Nsight Systems will give details of all GPUs, and cost of memory copies, etc.

Outline

Dynamic parallelism

- So far we have assumed that kernels can only be launched from the host
- However, this is not necessary; kernels can be launched from inside other kernels
- Launching is still asynchronous, so that a device thread needs to call `cudaDeviceSynchronize()` in order to ensure that all child kernels have completed
- More details are available in `CUDA_Dynamic_Parallelism_Programming_Guide.pdf`

Concurrent kernels (streams)

- Some applications launch fewer thread-blocks than GPU can hold
- So, in order to improve performance, you can launch multiple independent kernels at the same time
- May improve performance for some applications
- Use “streams” feature of run-time API:

```
myKernel<<< gridDim, blockDim, sharedMem, stream >>>();
```

- In practice, careful reading of manuals is necessary to ensure no accidental inter-dependency between kernels and memory copies, causing streams to wait for each other.
- Nsight Systems separates streams and allows you to see dependencies.

CUDA research

CUDA/GPUs are being used in groundbreaking research worldwide

- Google Scholar Search: `allintitle:CUDA` - 8,520 hits
- (and even more papers actually use CUDA or GPU acceleration).
- GPU Technology Conference `www.gputechconf.com`
- UK ManyCore Network `www.manycore.org.uk`
- Many HPCs have NVIDIA hardware installed (e.g. CSD3)