CUDA and You

Overview

CUDA, a programming language developed by GPU manufacturer nVidia, is used to facilitate highly parallelized programming. It is important, however, to know the platform you're coding for.

GPUs generally live on the PCI bus, an external interface to the processor. Processors can talk to general memory (RAM) relatively quickly, but throughput to devices to other devices, like graphics cards, is much slower. This is important because you're now dealing with both local and remote (GPU) memory in your calculations, which adds a larger overhead to your parallelism.

When coding for CUDA, the most important information is the `compute capability` of the card you're planning to execute on. The CWRU HPCC SLURM cluster provides several nodes with Tesla M2090 GPUs. According to this page, the Tesla M20xx series is limited to compute capability of 2.0. Wikipedia provides a handy table for operational differences between versions. The main loss is dynamic parallelism (i.e. running a kernel from within a kernel) is not allowed.

Coding

CUDA code is effectively C, sprinkled with some boilerplate memory management code around each kernel execution. Kernel functions are prefaced with `__global__`, and must return `void`. Any pointers given to the kernel function must be `cudaMalloc`'d and `cudaMemcpy`'d over to the GPU before execution. After execution, you must `cudaMemcpy` the data back to the host, then `cudaFree` the memory on the device.

Sample code for insertion sort can be found here.

Compiling

The compiler for CUDA is `nvcc`: it's very similar most C compilers. For example, to compile `insertion.cu` into the default `a.out`, the following command will suffice:

```
nvcc insertion.cu
```

The generated `a.out` file is your binary. To compile programs for GPUs with dynamic parallelism, the following command will help:

```
nvcc -gencode arch=compute_35,code=sm_35 -rdc=true insertion.cu
```

`insertion.cu` does not implement any dynamic parallelism, but the command is useful if you choose to go that route.

Executing

Executing CUDA code is straightforward with one caveat: kernel code execution must be fast, unless no display is attached to the GPU. For example, `insertion.cu` may happily run a couple iterations, but eventually slows down, and is killed by the driver to preserve display responsiveness.

Notes

Different portions of CUDA code are synchronous. Most notably, kernel calls are asynchronous, but memory functions are synchronous. For this reason, memory transfers will appear incredibly slow, but there's an implicit barrier call before the transfer, waiting on the kernel to finish.

Please don't use GPU "accelerated" insertion sort. Insertion sort isn't good for the average case, and it's sequential. Writing it in CUDA was a proof of concept, and performance is hilariously slow.