Building

On the HPCC (either dev-intel14-k20 or dev-intel16-k80), run

```
. load-modules.sh
make
```

to make `serial` and `cuda`, the respective versions of the code. To build the benchmarking versions for the current node, run

```
make -B bench
```

this will generate

```
serial.$HOSTNAME
cuda-sim.$HOSTNAME
cuda-sim-minmax.$HOSTNAME
cuda-sim-minmax-grayscale.$HOSTNAME
```

all of which output timing information when run. cuda-sim only CUDAfys the wave equation calculation, cuda-sim-minmax also CUDAfys the search for the min/max for grayscaling, and cuda-sim-minmax-grayscale does all that and CUDAfys the actual grayscaling itself.

For any invocation of `make`, one can add any of `BENCH=1` to turn on timing output, `DEBUG=1` to turn various debugging output, `NO_CUDA_MINMAX=1` to prevent the use of CUDA for the min/max calculation, and `NO_CUDA_GRAYSCALE=1` to prevent the use of CUDA for the grayscaling calculation.

Timing

Timings can be generated with

```
./gen-fine-timings <binary> <t-set>
```

For example,

```
./gen-fine-timings cuda-sim-minmax.dev-intel14-k20 cuda-sim-minmax
```

This will run the respective binary 50 times, redirecting stderr (with the timing info) to `timings/raw/<t-set>/$HOSTNAME.out`. You can then run

```
./compile-fine-timings.sh <t-set1> [<t-set2> ...]
```

which will generate nice tables in

```
timings/compiled/<t-set1>/dev-intel14-k20.dat
timings/compiled/<t-set1>/dev-intel18-k80.dat
timings/compiled/<t-set2>/dev-intel14-k20.dat
timings/compiled/<t-set2>/dev-intel18-k80.dat
...
```

Nice comparison histograms can then be generated with

```
python hist.py -o <output-image.pdf> <t-set1> <t-set2>
```

Included Timings

- `serial`: The original program, timed with bash `time`.
- `serial-O3`: The original program with -O3 compiler flag, timed with bash `time`.

- `fine-serial-1D`: -O3 with 2D-array representation changed from pointer-to-pointers-to-arrays to pointer-to-strided-1D-array.
- `cuda-sim`: All the above with the wave equation calculation CUDAfyed.
- `cuda-sim-minmax`: All the above with the min-max calculation CUDAfyed.
- `cuda-sim-minmax-grayscale`: All the above with the grayscaling CUDAfyed.

Included plots (self-descriptive)

- [1] `plots/serial__serial-O3.pdf`
- [2] `plots/fine-serial-1D__cuda-sim.pdf`
- [3] `plots/fine-serial-1D__cuda_sim-minmax.pdf`
- [4] `plots/fine-serial-1D__cuda_sim-minmax-grayscale.pdf`
- [5] `plots/cuda-sim__cuda-sim-minmax.pdf`
- [6] `plots/cuda-sim-minmax__cuda-sim-minmax-grayscale.pdf`

Code Structure

`wave_2d_serial.c` and `wave_2d_cuda.cu` contain the main functions for the serial and CUDA codes, respectively. CUDA kernels can be found in `cuda_kernels.c` and the functions used to interface these kernels with the host code can be found in `cuda_kernel_interface.cu`; `cuda_props.cu` contains functions for querying various CUDA device properties; and `CudaMem.h` contains a simple wrapper class over joint host/device memory (and could be improved significantly, though certainly libraries already exist with this sort of class).

`png_util.c` contains utilities for interfacing with libpng.

`bench.h` contains the macros used conducting the timings in the programs.

`debug.h` contains utility macros for debugging.

CUDA Kernel Principals

The wave equation simulation kernel in use in this final version, `sim_kernel_tiled`, lets each block of threads read in a tile of the current position mesh to shared memory, applys the calculation to the inner part of this tile, and then uses global memory to talk to other blocks and apply the calculation to the edges of the tile. Each thread then updates its assigned velocity and then position.

`sim_kernel_naive` is the most naive possible CUDA kernel for the wave equation simulation, but I have not compared it to anything else explicitly.

`sin_kernel_shfl` attempts to use CUDA's ______`__shfl_sync` intrinsics to avoid use of shared memory with warps, but I have not gotten this working yet.

`block_min_max_kernel` calculates both the minimum and maximum of each block of the input by: calculating each warp min/max using `__shfl_down_sync; writing these result to a shared memory; and then using the one warp to finish the calculation. The interface function`launch_min_max_kernel‘ applys this with the grid size equal less than or equal to block size, so that the kernel is then called again with one block to finish off the calculation.

`grayscale_kernel` just does the grayscaling calculation given the min and max found.

Conclusions

As always, turning on compiler optimizations makes a huge difference. The CUDA effect was unfortunately less pronounced. There is a definite ~3 times speed-up in CUDAfying the wave equation calculation, but the CUDAfyed min/max gives only a tiny speedup. CUDAfyed grayscaling strangely gives a significant speedup only on dev-intel14-k20, with a possible marginal speedup on dev-intel16-k80.