Supplement to Volume 3

This document is a supplement to *Deep Belief Nets in C++ and CUDA C, Volume III:Convolutional Nets*. It describes the enhancements contained in CONVNET Version 2, and it briefly describes the changes made to the C source file MOD_CUDA.cu in order to allow use of multiple CUDA devices.

New Features in CONVNET 2

Version 2 of the CONVNET program contains three new features:

1) Computation of the confusion matrix can now be interrupted with the ESCape key, although results will then be in great error. The CUDA device is now used for confusion computation, so it is much faster than in Version 1, in which the host CPU was used.

2) If the number of weights for each output neuron is 101 or fewer, then singular value decomposition is used during simulated annealing to provide nearly optimal output weights for each randomly defined trial hidden-layer weight set. This greatly improves the quality of the starting weight set from which refinement is done and thus speeds convergence. In very rare instances, this algorithm introduces numerical instability that manifests as garbage values during annealing. If this happens, you can check a box to disable this feature; this checkbox is in the *Training Parameters* dialog.

3) Version 2 can now make use of multiple CUDA devices; Version 1 used only one, the last in the configuration. If your computer contains more than one CUDA device, the *Training Parameters* dialog will allow you to select from one of three options:

a) Use first device only (*Device 0 will be used*)
b) Use last device only (*The last device in the hardware configuration will be used*)
c) Use all devices (*All CUDA devices will be used*)

Note that as yet I have no access to any computer which contains more than two CUDA devices, so I have been unable to test this version with more than two devices! I strongly suspect that it will work, but I encourage users to report any problems. I hope to purchase a 3-device system late in 2016, in which case I will test that.

Also note that if your rig is configured for SLI, you will likely have to disable SLI through the *nVidia* control panel in order for the multiple devices to be seen. This is not a CONVNET issue; *nVidia* designed it this way, presumably for a good reason.

The MOD_CUDA.cu file, modified for multiple devices

This download includes the version of the CUDA C source file MOD_CUDA.cu which has been enhanced to handle multiple CUDA devices. The most complex such enhancement is in the module that handles the shared-memory computation of activation of locally connected and convolutional layers. If you understand this one module, everything else should be clear. I have stripped it down here to an outline of bare essentials in order to avoid confusion. You can see details like launch configurations and error handling in the source file itself. Actually, those are pretty much unchanged from the prior version.

We begin with the first set of multiple-device launches. An explanation follows this partial listing. If necessary, please review the discussion of this routine in the book, as details unrelated to multiple-device launches are omitted in this presentation.

```
int cuda_hidden_activation_LOCAL_CONV_shared ( int istart , int istop ,... )
{
  ...

  for (idev=0 ; idev<n_devices ; idev++) {          // For all devices in system (or 1 if using just first)
    if (last_only) {                                // User wants only this one device used?
      if (idev != last_only)                        // Last in this version, but need not be
        continue ;
      this_device = idev ;                          // Device to launch
      dev_istart = istart ;                         // Index of first case in this device
      nc = istop - istart ;                         // Number of cases in this device
      }
    else {
      this_device = (idev + istart) % n_devices ;
      dev_istart = (idev + istart) / n_devices ;
      nc = (istop - istart) / n_devices ;
      if (idev < ((istop - istart) % n_devices))
        ++nc ;
      }
    if (! nc)                                       // No cases in this device?
      continue ;
    cudaSetDevice ( this_device ) ;                 // Select this device

    if (n_slices >= BLOCK_SIZE  &&  nc >= BLOCK_SIZE) // Big enough for at least one shared block?
      // Launch shared version

    else
      // Launch non-shared version

    } // For idev (shared version in all devices)
```

First, understand that the training cases have been distributed in rotation among the devices. At initialization time, the first case was sent to Device 0, the second case to Device 1, and so forth. Thus, each device will handle a subset of cases scattered across the training set. Each device will have approximately the same number of cases. This distribution must be done, as opposed to putting contiguous chunks on each device, in order to facilitate sensible division into batches.

When this routine is called, the user specifies the starting index, and one past the stopping index, of the batch of training cases to be processed in this call. These indices refer to the order in the original training set, as stored on the host.

The above code loops through all devices. If the user has specified that only one device (last_only here, though it need not be the last) is to be used, then skip over all devices except the one specified.

We have to compute three quantities:
this_device is the CUDA device on which the kernel will be launched.
istart is the starting index of this batch on the current device.
nc is the number of cases in this batch on this device

In order to understand how these three quantities are computed, draw a little chart. Going down the first column, list the indices of the cases in the host training set. Decide on the number of devices you will use; three is a good quantity for this thought experiment. In the second column list the device that will receive each case, and in the third column list the index of the case on that device. So your chart might look like this:

0	0	0
1	1	0
2	2	0
3	0	1
4	1	1
5	2	1
6	0	2
7	1	2
8	2	2

. . .

This chart should make the computation of the above three quantities clear. The only thing to be aware of is that because the number of cases will not in general be a multiple of the number of devices, we must conditionally increment the number of cases on a device. One implication of this fact is that different devices may have different case counts and hence have different control flows when we clean up later. In fact, if the batch is tiny we may actually find that some devices are not used at all (nc=0). We must be prepared for that!

Once we have these three quantities, we set the current device and launch the computation kernel. If we have enough cases and slices for at least one block, we use the shared-memory version. Otherwise we use the non-shared version, exactly as in Version 1.

The kernels are now running on all devices simultaneously. We must wait until they are all finished. It does not matter if we 'wait' for devices that had no kernel launched; these devices are always finished.

```
   for (idev=0 ; idev<n_devices ; idev++) {
      if (last_only  &&  idev != last_only)
         continue ;
      cudaSetDevice ( idev ) ;
      cudaDeviceSynchronize() ;
      }
```

Any device that had one or more cases but not enough cases and/or slices for the shared version used the non-shared version, and thus have completed the task. But devices that used the shared version may have some cleanup to do, exactly as in Version 1. Since different devices can have different case counts, it may be that some devices need cleanup and others do not. So we do these cleanups in separate loops:

```
/*
   Clean up any extra slices
*/

   for (idev=0 ; idev<n_devices ; idev++) {
      if (last_only) {
         if (idev != last_only)
            continue ;
         this_device = idev ;
         dev_istart = istart ;
         nc = istop - istart ;
         }
      else {
         this_device = (idev + istart) % n_devices ;
         dev_istart = (idev + istart) / n_devices ;
         nc = (istop - istart) / n_devices ;
         if (idev < ((istop - istart) % n_devices))
            ++nc ;
         }
      if (! nc)
         continue ;
      cudaSetDevice ( this_device ) ;          // We could do this after the check in next line
                                               // But that check is nearly always true
      if (n_slices >= BLOCK_SIZE  &&
          nc >= BLOCK_SIZE  &&
          (n_slices % BLOCK_SIZE))
         // Launch slice cleanup
      } // For idev (shared version slice cleanup in all devices)
   for (idev=0 ; idev<n_devices ; idev++) {
      if (last_only  &&  idev != last_only)
         continue ;
      cudaSetDevice ( idev ) ;
      cudaDeviceSynchronize() ;
      }

/*
   Similarly clean up any extra cases
*/

   for (idev=0 ; idev<n_devices ; idev++) {
      if (last_only) {
         if (idev != last_only)
            continue ;
         this_device = idev ;
         dev_istart = istart ;
         nc = istop - istart ;
         }
      else {
         this_device = (idev + istart) % n_devices ;
         dev_istart = (idev + istart) / n_devices ;
         nc = (istop - istart) / n_devices ;
         if (idev < ((istop - istart) % n_devices))
            ++nc ;
         }
      if (! nc)
         continue ;
      cudaSetDevice ( this_device ) ;          // We could do this after the check in next line
                                               // But that check is nearly always true
      if (n_slices >= BLOCK_SIZE  &&  nc >= BLOCK_SIZE  &&  (nc % BLOCK_SIZE))
         // Launch case cleanup

      } // For idev (shared version case cleanup in all devices)

   for (idev=0 ; idev<n_devices ; idev++) {
      if (last_only  &&  idev != last_only)
         continue ;
      cudaSetDevice ( idev ) ;
      cudaDeviceSynchronize() ;
      }
```

That's it. Note that we could move the call to cudaSetDevice() down inside the launch test, but that test is nearly always true in any but very small applications, and I think it's more clear this way, as it agrees with the first loop. Feel free to change it if you wish; it's inconsequential.