Contents

1 Introduction

Nengo is neural simulation software. Neurons are highly parallel computational units, so one would imagine that Nengo would be able to exploit this property to run simulations in parallel at high speeds. But in its original implementation, simulations in Nengo ran completely serially. Each neuron, each ensemble, was processed one at a time. Clearly there was room for improvement.

There was definitely a demand for that kind of improvement. The researchers that use Nengo are constantly making bigger and more complex models of the brain. So any speed improvement is highly sought after, allowing researchers to create and conveniently use these models. In this guide I will describe how to use the speedups to run your Nengo networks more quickly. Additionally, I outline their implementation so that people can improve it further.

At the CTN, we have capable computers. They effectively have 8 processors. But the original Nengo is only capable of using one of these at a time. So the obvious first step in improving the speed of Nengo was to have it take advantage of as many processors as it has access to. Hence, Nengo was given multithreading capablities. You can now specify how many threads you want to use for your simulation. And you'll get it a decent speedup from this if you manage to find the optimal number of threads for your particular network and system. But we'd like to have more. And there is a way we can go even further with this multithreading idea. We have many more processors available to us than it seems like at first blush. These come from the GPU. The cards in our machines, nVidia GTX 280 feature 240 processors for your simulating pleasure. This is where the real speedup comes from. Anyone with a CUDA enabled card can take advantage of this feature.

2 Setup

2.1 Multithreading Setup

2.1.1 Getting the code

The latest version of Nengo contains the multithreading code, so just check that out or download it and you'll be ready to parallelize your simulations.

2.1.2 Setting number of threads to use

There are several ways that you can set the number of threads to use in your Nengo simulations.

1. If you have access to Nengo's source, you can edit the default number of threads to use right in the source. Look in the class ca.nengo.util.impl.NodeThreadPool for the member variable defaultNumThreads and set it to the number of threads you want Nengo to use.
2. You can set it while Nengo is running. Open a python console in Nengo and type ca.nengo.util.impl.NodeThreadPool.setNumThreads(x) where x is the number of threads you want Nengo to use. If you use this command while a simulation is running, it won't take effect until the next simulation.

2.1.3 Disabling Multithreading

To disable multithreading, use one of the above methods to set the number of threads to 0. This avoids all multithreading code.

2.2 GPU Setup

Setting up Nengo's GPU capability takes more work since its implemented in a C library, and requires an API called CUDA to interact with the GPU.

2.2.1 Getting CUDA

If you are running Ubuntu Karmic Koala 9.10 or Jaunty Jackalope 9.04, you can get the CUDA toolkit and developer driver by following these instructions.

One note to add is that when you install the CUDA driver, say no when it asks about 32-bit OpenGL libraries.

After installing CUDA, be sure to folow its recommendation of setting up the environment variables for CUDA. Make sure that PATH contains /usr/local/cuda/bin and that LD_LIBRARY_PATH contains /usr/local/cuda/lib64 (with the path names changed appropriately if you installed CUDA somehwere other than the default location).

2.2.2 Getting the code

To get the code, update the entire contents of the folder simulator/src/c/NengoGPU from SVN.

Now you should be able to compile the libraries, so cd into simulator/src/c/NengoGPU, type make and cross your fingers.

Next, we have to tell Java where to find the NengoGPU once the program starts up. Using whatever IDE you use to edit Nengo source, add the following to the VM arguments:

```
-Djava.library.path=NENGO_GPU_PATH
```

where NENGO_GPU_PATH is the absolute path to the folder NengoGPU. You should now be able to run Nengo with the GPU.

2.2.3 Using the Code

Once you have the GPU code working, you can tell Nengo whether you want to use it or not. Open a console in Nengo and enter the command:

```
ca.nengo.util.impl.GPUNodeThreadPool.setUseGPU(x)
```

where x is either true or false depending on whether you want to enable or disable the GPU code.

Now even if you have the GPU code enabled as above, ensembles will not run on the GPU unless they are told to. To tell an ensemble to run on the GPU, use Nengo's console to call the ensemble's member function setGPU(1). I've left a python script with Terry that contains a function called setNetworkGPU that will take an existing network and tell all its NEF ensembles to use the GPU.

And thats really all there is to it. You should now be able to run your simulations faster than ever!

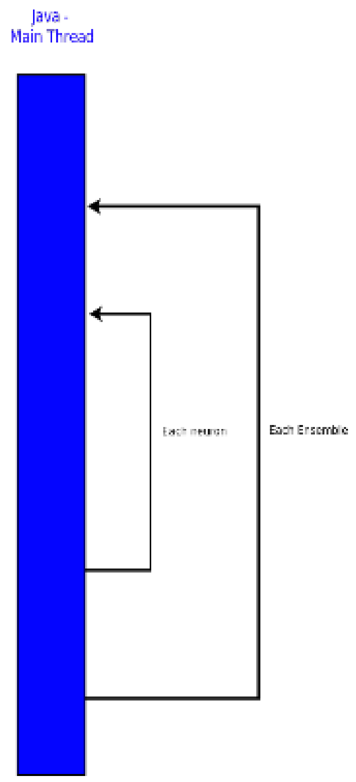
Keep reading to explore the details of the implementation.

3 Parallelization of a Nengo Simulation

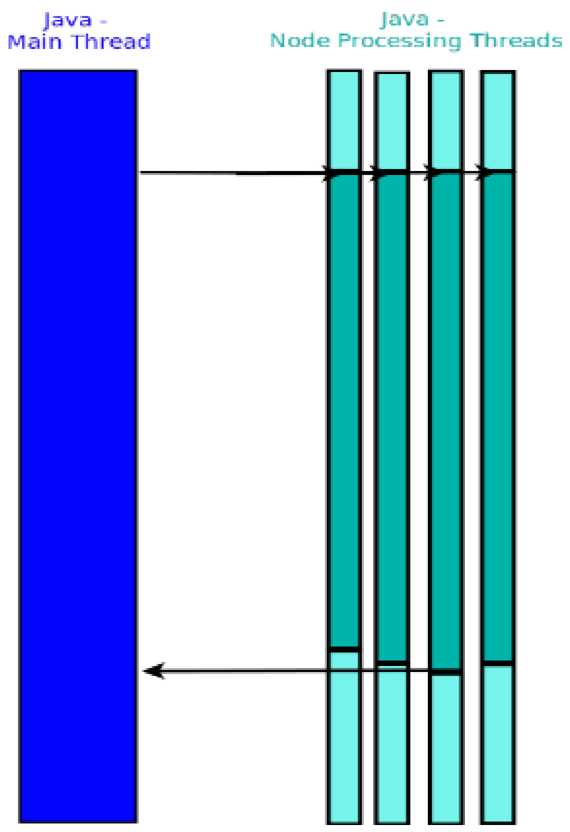
A Nengo simulation consists of a series of steps. At the beginning of the sim you specify how long each step is and how many steps there are. Each step has two phases. First, the value in each origin is moved into the termination that it projects to. Then each node takes the values in its terminations, computes some function on them, and places the resulting values in its origins. The important thing to notice here is that at the beginning of the part in which we compute functions the nodes functions, each node already has all its inputs. So the order in which we compute the nodes' functions is irrelevant. So this can be parallelized. We exploit this property in implementing multithreading.

4 Multithreading Implementation

Originally, a simulation in Nengo looked like this:



That is, one neuron at a time, one ensemble at a time. But as we showed above, we can do better. We allow the user to specify the number of threads to run a simulation with. If the user's computer has multiple cores, it should run faster since more nodes will be processed at once.
Its implemented as follows. At the start of a run, we launch as many threads as the user has specified. These threads are implemented by the class NodeThread. The main thread interacts with these node-processing threads all at once through another class, NodeThreadPool. Each step of the simulation, the main thread wakes up these threads and the threads process the nodes in the network for one step. So one step of the simulation looks like this:



Each vertical bars represents a thread. Faded colour represents that the thread is sleeping.

There are other ways that this could have been implemented. We could have started the threads at the beginning of each step and killed them at the end of each step. But the overhead of creating at the OS level and of creating instances of objects in Java generally negated the speedup gained by multithreading.

5 GPU Implementation

Multithreading alone typically doesn't quite get us the speedup we want. We need more computing power, more processors, and for that we go to the GPU. nVidia's GPU API, CUDA, can only be used through C code, so using the GPU requires some inter-language interaction.

5.1 The Java Native Interface

The Java Native Interface, or JNI as it shall henceforth be known, facilitates interaction between Java code and C (native) code. For a detailed introduction to the JNI, see this article. To summarize, we write a C library that contains the native functions we want to call from Java, then declare those same functions in the Java code, but mark them with the native keyword, and leave out a Java definition. Now you can call these functions just as if they were member functions of the class they are declared in. Before you use them though, you have to load the C libraries from Java, which is done by way of the System.loadLibrary(”LibraryName”) function. In Nengo, all the JNI-related code, and hence, all interaction with C code, is in the GPUThread class.

5.2 C-side Threading Structure

Nengo should take advantage of all the GPU's it has available to it. But one of the rules of CUDA is that each C thread can use only one GPU at a time. So we'll need as many C threads running as we have GPUs. In Java we add a class GPUNodeThreadPool, which is the same as a NodeThreadPool except that it has an additional thread, this time an instance of the GPUThread class, which runs all GPU-marked nodes on the GPU. At the beginning of a simuation, the GPUThread transfers all the data from the nodes it is assigned to run to the C code through a JNI function call. This function also creates one thread for each GPU available. Then each step, the GPUThread makes another JNI call, this time passing the input to each node for that step. The function wakes up the GPU threads, gives them the new input, and they process the nodes for one step. So the threading structure with the GPU code looks like this:

5.3 CUDA

Soon we'll get into the inner workings of the GPU code, how we actually process the neural ensembles on the GPU. But to understand that, we'll you'll need an understanding of the CUDA API. For a detailed introduction, download the CUDA Programming Guide. For detailed information on all CUDA functions, download the CUDA Reference Manual. Here I'll summarize the bare essentials.
Kernels are functions written in C that are executed on the GPU. They are marked with __global__ to indicate that the can be called from the GPU or from the host (the CPU), but otherwise look like normal C functions. The key

difference is that when you call a kernel, it isn't run just once. Its run many times at once on the many processors of the GPU. Now running the same function a bunch of times seems a little redundant. But these functions aren't exactly the same. Every kernel instance has access to two variables, threadIdx and blockIdx and the values of these are different for each. This makes each kernel instance act in a slightly different way. For example, if you use the threadIdx and blockIdx as an array index, you can have each kernel instance access a different element of an array and process the whole array in the time it would normally take to process a single element.
The syntax for calling a kernel is a supplemented version of standard C function invocation:

```
kernelName<<<gridDim, blockDim>>>(arguments)
```

gridDim and blockDim have type `struct Dim3` which comes as part of the kernel API. The number of kernel instances launched is `gridDim.x * gridDim.y * blockDim.x * blockDim.y * blockDim.z` .

5.4 GPU Implementation

In this section we detail how each NEF Ensemble is processed on the GPU.

5.4.1 Overview

When this project was originally started, we tried a simple GPU implementation, hoping that the sheer power of the GPU would be enough to give a decent speedup. That implementation parallelized across neurons within a particular ensemble, but not across neurons in different ensembles. Each ensemble was processed by the CPU serially, but the neurons in an ensemble were all processed at once.
This didn't give us what we wanted. It was barely faster than multithreading. We needed to go further with the GPU implementation. The next step was to run all the ensembles in the entire network at once. But to do this we had to do some strange things. The way the ensemble data is normally stored is not appropriate for parellization within the CUDA framework, so the first thing we do once we get the data from the Java call is to store it in a more useful format. To complicate things further, CUDA's memory accesses are fastest when done in a certain pattern, and we wanted to adhere to that to get that speedup, so the data storage format is further constrained.

5.4.2 NengoGPUData struct

This struct stores all data for the simulation. We create one for each GPU device in use. The data from each ensemble goes in exactly one of these structures. The relevant code is in

simulator/src/c/NengoGPU/NengoGPUData.h and
simulator/src/c/NengoGPU/NengoGPUData.c.
This structure has a number of fields, most of which are arrays containing the data of the ensembles that have been set aside to run on the current GPU.

5.4.3 Executing the simulation on the GPU

The following flow chart is makes explicit the computations that must be executed to complete one step of a Nengo simulation.

1. Multiply input vectors by termination transforms - each ensemble has some number of terminations. Each termination has an associated transform matrix which we must multiply by the terminations input vector.

2. Sum results - The result of the first step is a series of vectors with length equal to the dimension of the ensemble. We add all these vectors together.

3. Encode - Multiply the result by the encoding matrix which has number of neurons rows and ensemble dimension columns. The result is vector of length number of neurons, the ith entry of which contains the current stimulating the ith neuron of the ensemble.

4. Firing function - For each neuron, compute its firing function, a function of the current we just calculated. The result is a vector of length number of neurons, the ith entry of which = 1 if the ith neuron is firing this step and = 0 otherwise.

5. Decode - Multiply the decoder matrix by this firing vector to obtain output vector.

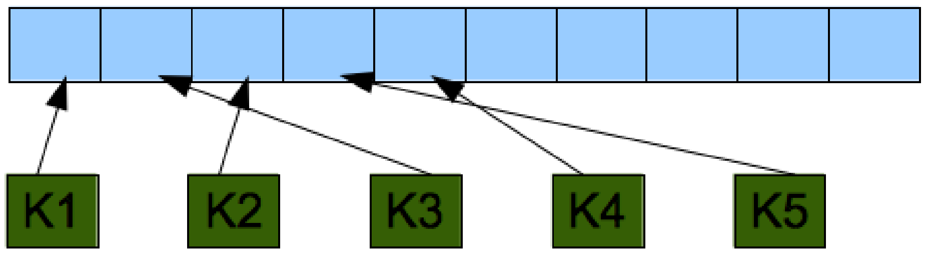
6. Put output vector in each of the ensemble's origins.

A quick perusal of the above list reveals that matrix multiplication is the main operation we should concern ourselves with. Now, there are CUDA libraries out there that will use the capabilities of the GPU to do matrix multiplication really, really fast. But these only work one at a time, and are meant for very large matrices. Our problem is just the opposite. We have many matrix multiplications to perform at once, all involving matrices and vectors of relatively small size. This is a fairly unique problem, so we'll have to come up with our own solution.

5.4.4 Example: Termination Transform Multiplication

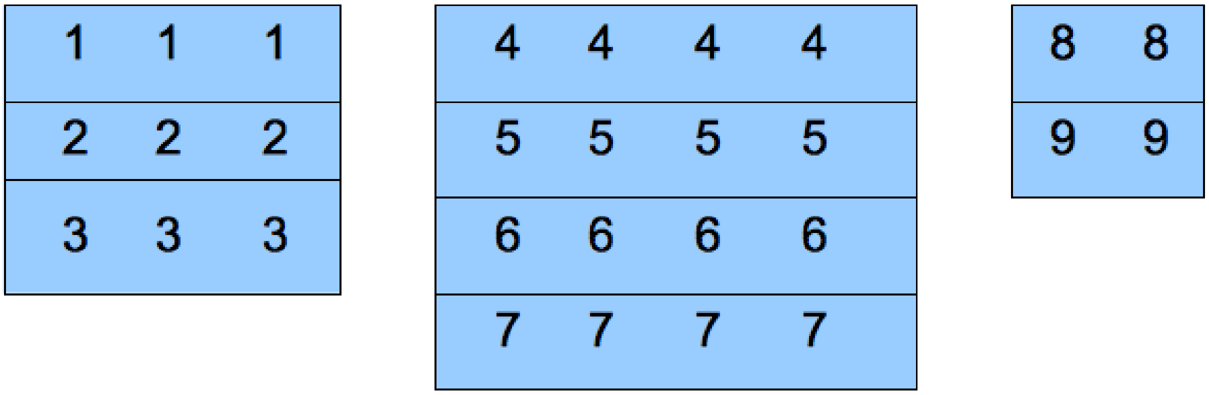
The termination transform step is probably the most involved step. The other steps are essentially simpler versions of this one. I call it the most involved because of the number of matrices possible and because the matrices can all have different sizes.

Each ensemble can have any number of terminations, and each termination can have any dimension. So we have all these different sized matrices, and we want to multiply them with different sized vectors, and we want all this to happen at once on the GPU. Not a trivial task!
The first issue is how exactly to parallelize this computation. Matrix multiplication can be reduced to performing a series of dot products, so we picked that as our basic function, with the idea that we write a kernel that performs dot product on two vectors, and launch as many kernels as are required to complete the computation. Sounds simple, but the devil is in the details, as we will see.
Since the vectors can all be of different length, each instance of the kernel will perform a dot product of a different length, so how does each instance get its unique length? Because the length is different for each instance, we can't just pass the values in as an argument. What we can do is pass in a pointer to an array that contains the lengths of the vectors. Then each instance can use its unique thread ID and block ID to index this array and retreive the information it requires. In CUDA, if a bunch of kernel instances access contiguous words of memory at once, then all the memory accesses can be completed in one clock tick, so this technique introduces very little overhead.

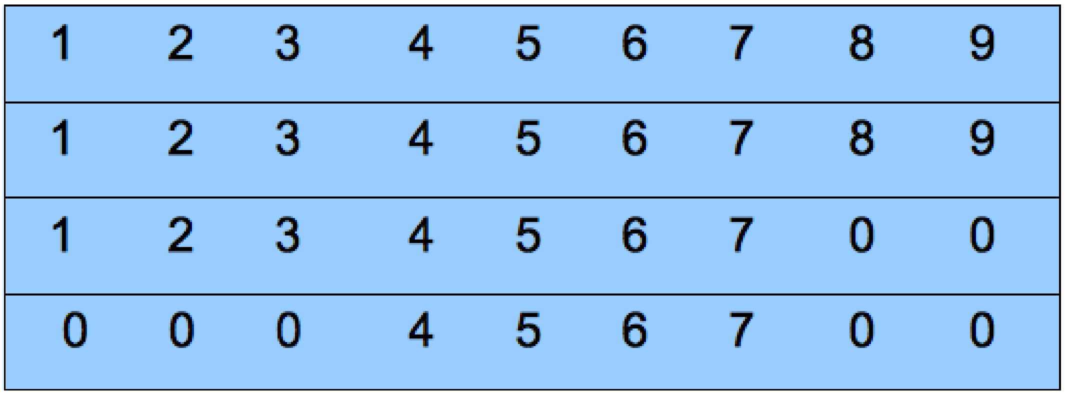


This pattern is used extensively throughout the GPU implementation.
Another very big issue is how to store and access the elements of the assigned vectors. Here we must again pay heed to the CUDA memory access pattern. As described above, if a bunch of kernel instances access contiguous words of memory, then all the memory accesses are done at once, and the whole thing takes only one clock tick. On the other hand, if the accessed words are not contiguous, than the accesses are done serially, and the whole thing takes time proportional to the number of kernel instances made. If we have many instances running, this could be a major slowdown, so we want to make every effort to make sure that kernels are always accessing contiguous words.
Now to see how this affects us, lets consider how we will likely implement the dot product kernel. It'll just be a loop that iterates once for every element in

the vectors, each time taking the ith element of each of the two assigned vectors, multiplying them, and adding the result to a sum variable. There will be many instances of this running all at once on the GPU. Notice that every kernel instance will be accessing the ith elements of its assigned vectors at once. So to respect the CUDA conventions, not only do we have to have all the matrices stored in one array, but we also require that the first elements of all the vectors be adjacent, and the second, and the third, and so on.
So consider, as an example, that we are given these as the termination transform matrices:



We have each vector that we want to dot product containing the same numbers so we can see how the vector moves when we store it in an alternate format. So when we put the above matrices in a C array in a configuration designed to take advantage of the CUDA memory access patterns, we get something that looks like this:



1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	0	0
0	0	0	4	5	6	7	0	0

There is one glaring problem with this, which is all the wasted space occupied by the 0's, an apparently necessary evil having to do with the possible differences in length of the consituent vectors. But it is a problem we can't ignore, since the amount of wasted space is proportional to the difference in length of the vectors and the number of vectors, both of which are unlimited. And luckily, we came up with a solution.
First, we just get rid of the wasted space and push everything over to the left. But this complicates the indexing of the array quite a bit. Before, if we wanted the 5th element of the third vector and we knew there were nine vectors, we would type: A[(5-1) * 9 + (3-1)] = A[4 * 9 + 2]. But since we "pushed everything over to the left", you can throw any notion of the indexing being that simple out the window. The new storage structure looks like this:

1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7		
4	5	6	7					

But all is not lost! At the beginning of a run, we compute an "index array" for each array structured as above. The index array gives the index of an element of a vector based on the index of the last element. Then we assume that every vector has at least one element, so we can figure out for ourselves where to find the first element of the vector, and use the index array for the rest of them. The index array for the above array looks like this:

9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	-1	-1
-1	-1	-1	25	26	27	28		
-1	-1	-1	-1					

So each iteration of the dot product loop, we have a statement to get the new index into the termination transform array, like this:

terminationTransformArray_index = terminationTransformIndexArray[

terminationTransformArray_index]

The index array is exactly the same size as the termination transform array, so the storage required is

2 * total number of vector elements

instead of

number of vectors * size of largest vector

which is an improvement in most cases.

That sums up the more esoteric techniques that were required. The other steps all use the same techniques, so equipped with that knowledge you should be able to figure out exactly how they work.