Fugu

Brad Aimone, William Severa, Craig Vineyard

November 21, 2018

1 Introduction

This document summarizes the notes from the Fugu Week by BA, WS, and CV at the end of September

2 Overview of Fugu

2.1 Goal of Fugu

The primary goal of Fugu is to provide a general scientific computing user access to emerging neuromorphic hardware — specifically *spiking* neuromorphic hardware — by providing an accessible library of functions that Fugu provican map into neural hardware

Fugu provides the following core capabilities

- **API to conventional programming environments (i.e., C++, Python)**
- **A functional *intermediate representation* of spiking neural algorithms**
- **Outputs to neural hardware compilers or Fugu's reference simulator**

2.2 API

The goal of the API for Fugu will be to make the construction of a Fugu algorithm be readily called from C++ or Python. In theory, all of the spiking algorithm requirements and processing should be transparent to a user; such that they only have to call a function with standard I/O protocols; not unlike CUDA.

2.3 Intermediate Representation

The primary contribution of Fugu is the managed IR between higher-level coding environments and low-level neuromorphic hardware and their compilers. This

IR consists of three components that, during compilation, provide the connection between the API and the compiler output: a library of *SNA Modules*, a collection of algorithms for linking SNA Modules, and the combined application graph output.

The IR of Fugu exists within Python, and it leverages the NetworkX libary to construct and manage the neural circuits that will be generated and combined during Fugu operation.

2.3.1 Library of SNA Modules

Each spiking neural algorithm (SNA) within Fugu is termed a Module (for the time being). Importantly, the algorithms that are contained within Fugu are not explicit neural circuits, but rather the algorithms that can generate the appropriate neural circuit for a given application.

For example, we consider the constant-time 1-dimension max cross-correlation algorithm by Severa et al., ICRC 2016. That algorithm compares two binary vectors of length n by having a dedicated neuron within an intermediate layer calculate each potential off-set, requiring an intermediate layer of size n^2. Subsequently, an output layer, sized $2n-1$ samples that intermediate layer to determine the relative shift between inputs that yields the maximal overlap.

As the description shows, the exact neural circuit needed is highly specific to n. In addition, the resulting circuits have other important properties that will be necessary at later Fugu stepts. For instance, for the constant-time algorithm, the inputs are provided all at the same time ($T_{in} = 1$), and the output is always a single time-step ($T_{out} = 1$) arriving two time steps later ($D = 2$). An alternative version of this algorithm streams the inputs in over time and uses delays to make the same computation with fewer neurons, albeit at an extended time cost, a characteristic that will produce different neural circuit and associated metadata. It is also important to note that this metadata may be a function of input parameters as well — for instance in the matrix-multiplication application described in Parekh et al., SPAA 2018, there is a version of the algorithm whose depth is $O(loglogn)$, where n is the size of the largest matrix dimension.

A schematic of the module is shown in figure 2.3.1. The properties of each module within Fugu are:

- N_{in} – number of input neurons
- T_{in} – time length of inputs (how long do inputs stream in). =inf if streaming
- N_{out} – number of output neurons
- T_{out} – time length of output. = inf if streaming
- D – circuit depth, corresponding to how many global timesteps must pass for the input at t=1 to reach

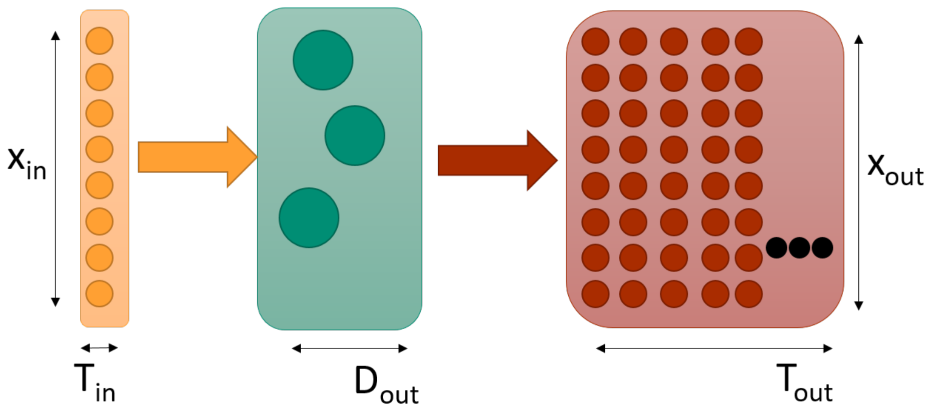

Figure 1: Fugu Module

To exist within Fugu, each of the modules must be able to take instructions from the Fugu environment and produce the appropriate neural circuit NetworkX graph per the necessary scales and timings. These circuits are referred to as *local circuits*, and are standalone circuit SNAs for computing the requisite function.

Further, at build time, only one of the modules may have a well-defined set of parameters. The parameters of other modules are dependent on that module's inputs. So each module must also have a function **returnParameters**(N_{in}), which returns, for that module, the appropriate size of that module when the input size is defined. This function will be of importance in linking the modules.

The set of required functions for each module are as follows:

- **returnParameters**(N_{in}) returns $N_{out}, T_{in}, T_{out}, D$
- **buildModule**($N_{in}, N_{out}, T_{in}, T_{out}, D$) returns NetworkX graph of algorithm at desired size

2.3.2 Linking code to combine local circuits into global circuit

Once the individual Modules are identified to generate all of the required local circuits, a secondary routine is required to appropriately scale the modules, generate the circuits, and link these output NetworkX graphs together into a larger unified global circuit. There are two primary challenges and a number of routine steps that require care for linking these circuits together.

- **Align sizes of modules**. Each model has an input size N_{in} and an output size N_{out}, and in order for two modules to be compatible with one another *serially*, then it is necessary that the downstream module is scaled appropriately to generate a graph suitably sized to take the outputs.

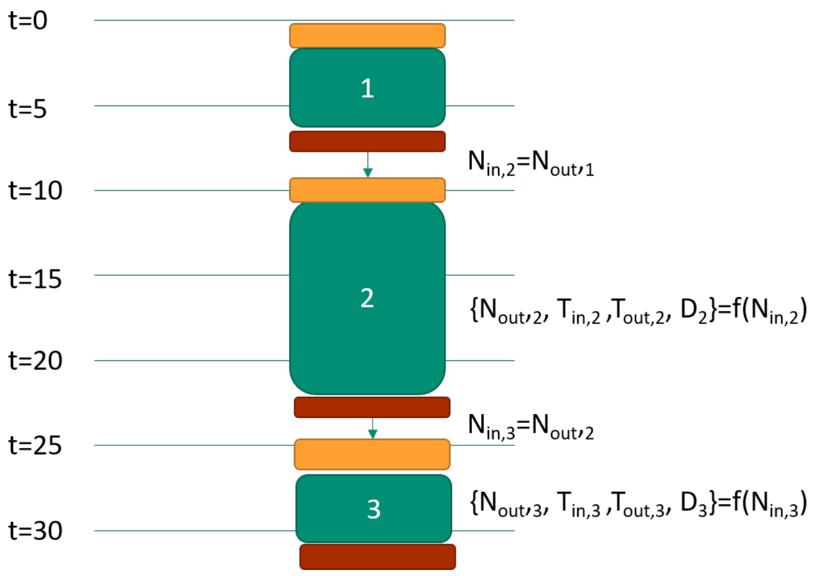
t=0
t=5
t=10
t=15
t=20
t=25
t=30
1
2
3
N_{in,2}=N_{out,1}
{N_{out,2}, T_{in,2} ,T_{out,2}, D₂}=f(N_{in,2})
N_{in,3}=N_{out,2}
{N_{out,3}, T_{in,3} ,T_{out,3}, D₃}=f(N_{in,3})

Figure 2: Normalizing module sizes

- **Align timings of modules**. Each of the modules has associated with it a T_{in}, T_{out}, and a D. If modules are intended to run in *parallel*, the difference in timings and depths may require that an additional *delay circuit* is instantiated on one of the pathways to ensure that the circuit is appropriately timed.

Stage 1: Normalizing graph sizes

As the first point above states, any serial alignment of multiple Fugu modules will at minimum require that the I/O of the modules are compatible with one another. As only the first modules have defined sizes (by whatever input data is identified at the onset), Fugu must deduce the sizes of all subsequent modules. A general scheme is shown in Figure 2.3.2.

This sizing algorithm is relatively straight forward. We assume that the first Fugu module has a defined set of size parameters $[N_{in}, N_{out}, T_{in}, T_{out}, D]_1$, and that any module can be fully defined by setting its input size. (**Note: we may also want to flag different depth versions of some algorithms**). After the graph for the first module is built, we move to the second module. Here, we take $N_{in,2}$ to be equal to $N_{out,1}$, and then we retrieve the other size parameters for module two by calling **returnParamaters** with $N_{in,2}$ for module 2, and then building that module. So long as the module sequence is serial, we can proceed in this manner.

If there is any branching the module sequence, the sizing clearly will depend on the upstream module in terms of I/O; regardless of what the indexing may be. Within each branch, the serial sizing can proceed as above.

State 2: Normalizing the architecture timing

So long as the structure of the anticipated combined Fugu network is serial, timings do not have to be worried about. However, once there is a branch in the code, whereby two circuits will then run in parallel for a period of time, only to be brought back together, it becomes necessary to include some additional steps to keep the overall computation in synchrony (Figure 2.3.2).

As the branches may be indexed arbitrarily and the time or depth of a module may be undetermined until its overall size is identified, it is unknown at the start of Fugu which branch will be the limiting factor in terms of time. So once Stage 1 above is completed and the sizes of each module is defined and the component graphs are produced, we must then determine the longest branch. (for i... Nbranches; find longest). This longest branch is the reference length that all other branches must match. Once this *branch depth* is found, we then work through each of the other branches to make the depths equivalent.

Two options are possible at this point. First, we can simply add a delay block - most simply a set of repeater neurons that spike with a delay of whatever the difference is. Most simply, this delay block could be at the end of the branch. However, there is likely a benefit to load balance over time; the delays will be relatively cheap in terms of computation, and thus they can perhaps be staggered at different times of each branch to keep the overall network activity at a roughly comparable level.

The second approach is more subtle. Many of the SNAs being developed can be tailored to use fewer neurons if more time is available. This time-space tradeoff is generally biased towards the faster algorithm; however in cases where a module with such a tradeoff sits within a branch with "free-time" so to speak, it is possible, and perhaps even advantageous, to represent that module in a more time-costly, space-efficient manner that reduces the overall neuron footprint of the model.

For this second approach, it may be useful to have access to an "alternate" graph, or at least access to some form of metadata that will inform Fugu whether that is an option.

Stage 3 - Link graphs together

Once the architecture has been fine-tuned to keep delays constant and the graphs are all produced with the required sizes, it is then necessary to combine the graphs into one large graph. This would likely best proceed serially through the set of modules. The key, as shown in Figure 2.3.2, is to replace the input nodes of the downstream graph with the output nodes of the upstream graph.

One note here is that the dynamics may likely need to be normalized, at least for these overlapping populations.

2.3.3 Output

The output of Fugu will be a single NetworkX graph that fully describes the spiking neural algorithm. The **edges** of this graph will be the synapses of the model, and will accordingly have weights associated with them as attributes. The **nodes** of this graph will be the neurons of the model, and will accordingly

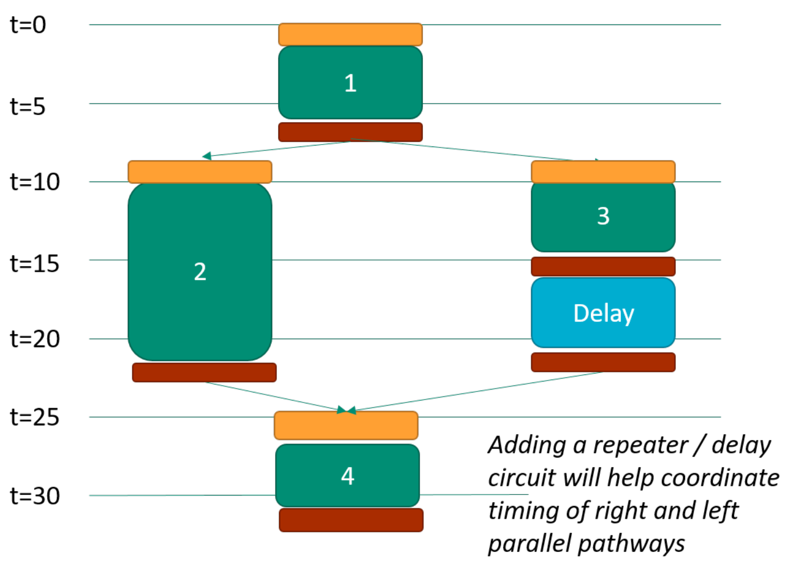
t=0
t=5
t=10
t=15
t=20
t=25
t=30
1
2
3
Delay
4
Adding a repeater / delay circuit will help coordinate timing of right and left parallel pathways

Figure 3: Adding a delay to synchronize Fugu branches

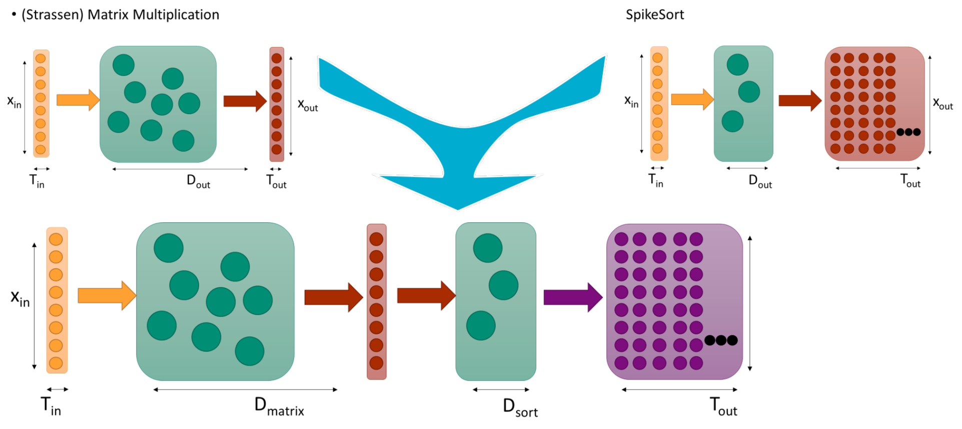

Figure 4: Combined Modules

have dynamics parameters associated with them. Additionally, some parameters, such as learning rates, additional dynamical states, etc. may be included within the attributes.

- Edges = synapses
  - source, target (required)
  - weight (required)
  - plasticity attributes (optional)
- Nodes = neurons
  - id (required)
  - dynamics parameters (required)

2.4 Compiler

There output of the Fugu linking process is a single NetworkX graph with described dynamics within the nodes. In principle, this description, along with metadata describing the I/O and a few other parameters, should be sufficient to fully describe the spiking neural algorithm. Of course, different neuromorphic hardware platforms will have differing requirements on how networks should be represented as well as distinct constraints, such as graph restrictions or specialized neural dynamics.

Fugu will come with three general options. First, Fugu will come with a **reference simulator**which will be guaranteed to run any Fugu-compatible algorithm. The reference simulator will not be optimized for speed or performance, but it will provide a rough-order-of-magnitude ability to benchmark performance of Fugu graphs. The second type of output will be **ONNX network descriptions**, whereby ONNX-compatible Fugu graphs (likely a narrow restricted set that look like feed-forward ANNs) can be output as ONNX files and thus readily transferred to ONNX-compatible hardware. The final output, and the one most intended for Fugu operationally, is a suite of **neuromorphic hardware-specific compilers**. Likely, this step will require a different compiler tool for each hardware platform; although for some (such as SpiNNaker and sPyNNaker), existing tools may be able to be used with relatively modest translation.

2.4.1 Reference Simulator

The reference simulator will be a very straightforward python simulation of the NetworkX graph with leaky-integrate and fire dynamics. It is intended to fully capture the full representational capacity of what Fugu networks can represent; so if Fugu is extended in any meaningful way (e.g., adding learning rules), the reference simulator must account for that.

The goal of the reference simulator is to enable the following:

- Validation of Fugu NetworkX outputs
- Provide ballpark estimates for performance
- Enable and apples-to-apples comparison between spiking algorithms

2.4.2 ONNX

...description of why we need ONNX compatibility here...

3 Algorithms within Fugu

3.1 Algorithms we have...

- Numerical Kernels
  - Matrix-Matrix Multiplication (*Parekh et al., SPAA*)
  - Vector-Matrix Multiplication
  - Cross-correlation (*Severa et al., ICRC*)
  - Triangle counting (*Parekh et al., SPAA*)
- Data Measures
  - L1-norm (*Severa et al., ICRC*)
  - Counters (*Severa*)
  - SpikeMax, SpikeMin, SpikeSort (*Verzi et al., Neural Computation*)
  - K-nearest neighbor (*Vineyard*)
  - Djikstra (*Parekh, Severa*)
- Data Manipulation
  - Low-pass Filter (*Severa*)
  - Convolutions (*Severa*)
- Application algorithms
  - Whetstone (*Severa et al., submitted*)
  - Monte Carlo Random Walk (*Severa et al., IJCNN*)

3.2 Algorithms we need...

- Numerical Kernels
  - Basic Arithmetic (add, substract, multiply, divide, etc)
- Data Measures

  –

- Data Manipulation
  – Delay / Repeat
  – Code remapping (spatial –¿ temporal, etc.)
  – Hashing
- Application algorithms
  – Random Forest

3.3 Algorithms we could have...

- Numerical Kernels
  – Fourier
  – Binary dot product
  – Matrix inversion
- Data Measures
  –
- Data Manipulation
  – High-pass filter
- Application algorithms
  – Graph 500 suite
  – Support Vector Machines
  – SpikeART
  –