3D-DRAM based ANN System Documentation

generated from https://github.ncsu.edu/lbbaker/ece-cortical-MainResearch, 3D-DRAM ANN system github wiki: 18.01.2018

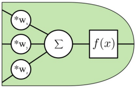
Contents

Preface

Welcome to the 3D DRAM based ANN System

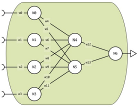
Machine Learning in the form of Deep Neural Networks (DNN) have gained traction over the last few years.
They get good press in applications such as image recognition and speech recognition.

DNNs are constructed from a basic building block, the artificial neuron(AN).
The basic processing of an artificial neuron is typically multiplying its inputs using a weight and then processing the combination of the inputs using an "activation function".



To summarize, the weighted input from all the feeder ANs are accumulated and passed through an activation function to form the ANs output.

An Artificial Neural Network (ANN) is formed using many ANs. Each AN has inputs connected to many feeder ANs and its output feeds many other ANs. For the most popular ANNs, the network is constructed using layers of ANs with the input on the left feeding forward through the network to the output on the right.



The activation function is usually a Rectified Linear Unit (ReLu) or a sigmoid

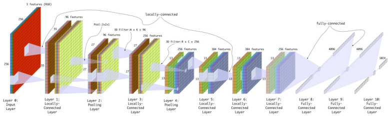
But what about [implementing]() a Deep Neural Network?

Implementing Deep Neural Networks

Useful DNNs often require hundreds of thousands of ANs.

Within the network, each AN can have hundreds of feeder (pre-synaptic) ANs.

With popular DNNs, there are often tens of layers.



So in these ANNs, the memory requirements are significant. The storage is required for the input, the AN state and most significantly the weights for each of the ANs. This storage requirement often results in gigabytes of memory.

When these ANNs are required to be solved in fractions of a second, the processing and memory bandwidth becomes prohibitive.

In most cases, Graphics processing Units (GPU) are used to implement large ANNs. In many ANN architectures, such as Convolutional NNs (CNN), they are quite effective.
However, we should not forget they are not optimized purely for ANN processing and are restricted by available SRAM and they are power hungry. These limitations will limit the effectiveness of GPUs regardless of what we might hear from the GPU community ("declare an interest").

Much of the ANN application specific (ASIC/ASIP) research has focused on taking advantage of the performance and ease of use of Static Random Access Memory or SRAM.
These implementations can be shown to be effective with specific ANN architectures (CNN), server applications or the "toy examples" but when a system requires multiple disparate ANNs in an edge application, these implementations do not provide the required flexibility, storage capacity and deterministic performance.

If we recognize that DRAM is required to store the NN parameters, why use SRAM as an intermediate store?
Well, in practice there are benefits if you can operate solely out of SRAM.
Certainly good performance and potentially low power.
But use of SRAM makes assumptions on the NNs that can be supported.
The primary requirement of the NN to allow effective use of SRAM is "reuse". Once parameters are stored in SRAM, can they be reused such that the SRAM isn't simply an intermediate memory but something akin to a cache.

In most cases there are reuse opportunities. With CNNs, the weights are reused. A convolutional filter is passed across an input to form the next layer. These filter "kernels" can be held in memory and the input is read from DRAM thus reducing the DRAM bandwidth.
Even with DNNs where weights may not be reused, when implementing multiple DNNs, there is opportunity to hold the input in memory.
If the system is being employed in cloud applications or in training, again there is opportunity to reuse inputs whilst performing batch processing.

But SRAM comes at a price, its big. Often when we see physical layouts of NN processors, they are dominated by the silicon area of the SRAM. The area required for SRAM has been understood for quite some time and companies attempt to create custom SRAMs to minimize the area impact.

So, can we employ DRAM with minimal SRAM and still provide a high performance system within acceptable area constraints?

We believe a system can be designed with DRAM as the primary processing store.
This will require careful use of data structures to describe storage within DRAM to ensure we make good use of the potential bandwidth. But there are other benefits we will take advantage of, but more about that later.

So lets review our research mission

Mission Statement

Real world applications will require multiple disparate NNs to be solved simultaneously

We believe that many real-world applications will employ multiple disparate instances of these useful sized ANNs.

If ANNs fulfill their potential, we believe that systems employing ANNs will utilize them for various functions, such as engine monitoring, anomaly detection, navigation etc. all within the same system.

It is also our belief that useful NNs will be large, 100s of thousands of ANs.

Now, there are examples where simplified ANNs have demonstrated efficacy, but are these ANNs sized to fit the capabilities of the employed system ("declare an interest").

We need the capacity provided by Dynamic Random Access Memory (DRAM)

SRAM is easy to use, but in reality they use a lot of silicon.
DRAMs are difficult to use but they provide the capacity we need.

Much of the research has employed SRAM. Perhaps because it provides high bandwidth but could it also be because its easy to use

Some research has employed DRAM, but its often used as a feeder to an SRAM based implementation.

Need to consider the system impact

Research often focuses on point problems. This isn't unreasonable, research institutions have limited resources.

But the size of NNs require that performance be evaluated in the context of the system. The memory capacity of these NN systems require that data is constantly moved from memory to processing elements and back. The performance impact in a system context cannot be trivialized.

So lets discuss our research.

What are we proposing

3D integrated circuit technology

If we keep the system constrained within a 3DIC footprint, we can leverage the benefits of 3DIC:

- reduced energy and area
- increased connectivity and bandwidth

A 3DIC System Overview

The system is made up of three major blocks:

- 3D-DRAM
- Management
- Processing

3D-DRAM has recently become available in standards such as High Bandwidth Memory (HBM) and Hybrid Memory Cube (HMC) and proprietary devices such as the DiRAM4 available from Tezzaron. These technologies provide high capacity within a small footprint.

In the case of HBM and DiRAM4, the technology can be combined with additional custom layers to provide a system solution.

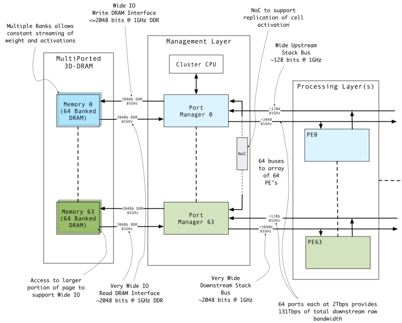
The question becomes, can a useful system coexist within the same 3D footprint?

Our research involves determining whether a realistic system can be implemented within a 3DIC footprint.

From a system perspective, we have a main memory (3D-DRAM), a manager and processing engines.

From a 3DIC perspective, our work employs a management layer, processing layers and layers for the customized 3D-DRAM.

Below is a high level diagram of the system.



Our baseline system will:

- target single precision floating point for computations
- use the Tezzaron DiRAM4 DRAM for area estimates and memory controller design

The work includes customizing the interface to a 3D-DRAM, researching data structures to describe storage of ANN parameters, designing a memory manager with micro-coded instructions and a

processing engine (PE) layer.
The system is designed such that a sub-system, known as a sub-system column (SSC) operates on one of these disjoint memories within the 3D-DRAM. When the sub-system columns need to share data or neuron activations, the data is passed between SSCs using a network-on-chip (NoC).

The work will answer the question,

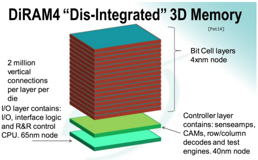
> "can a useful system be provided within the 3D-DRAM footprint"

Our system needs to provide a management layer and a processing layer sub-system within the physical footprint of each DiRAM sub-memory.

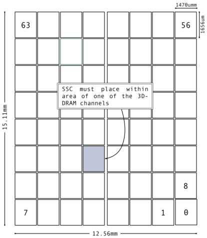
An overview of the three major blocks is described below but will also be described in more detail in later sections.

3D-DRAM

The Tezzaron DiRAM4 is a 3D-DRAM employs multiple memory array layers in conjunction with a control and IO layer.



The memory is formed from 64 disjoint sub-memories each providing upwards of 1Gigabit with a total capacity of at least 64 gigabit.



Manager Layer

The Manager block is the main controller in the system. The operations required to process an ANN are formed from individual instructions which are decoded by the Manager. These instructions include descriptors to describe memory read operations, processing engine operations and memory write operations.
The manager reads these system instructions from an instruction memory, decodes the instruction and configures the various blocks in the system.
The configuration includes:

- initiate operand reads from DRAM
- prepare the processing engine (PE) to operate on the operands
- prepare the result processing engine to take the resulting neuron activations from the PE and write those results back to the DRAM

- replicate the resulting neuron activation's to neighbor managers for processing of other ANN layers

Processing Layer (PE)

The PE is able to operate on data streamed directly from the DRAM via the Manager layer. The PE is configured by the manager to perform operation on the operand data streamed from the manager. In our baseline system, the main operation is to perform multiply-accumulates on 32 execution lanes of two operands. These operands typically are the pre-synaptic neuron activation's and the connection weights. The PE also performs the activation function on the result of the MAC to generate the neuron activation value. These 32 activation values are sent back to the Manager layer.

Layer Interconnect

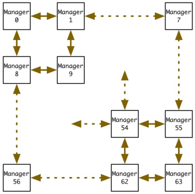
The layers are connected using through-silicon-vias (TSVs) which provide high connection density, high bandwidth and low energy.

By ensuring the system stays within the 3D footprint ensures we can take advantage of the huge benefits provided by TSVs.

Inter-Manager Communication

During configuration and/or computations, data must be transported between managers. This inter-manager communication is provided by an NoC.

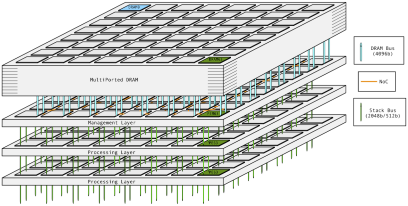
When computing an ANN across multiple processing sub-systems, often neuron activation data must be shared between these sub-systems. In our case, the sub-system includes the DRAM port, the manager and the PE. An NoC within each management block communicates with each adjacent manager using a mesh network. This NoC has a forwarding table that can be reconfigured to provide more efficient routing for a given processing step.



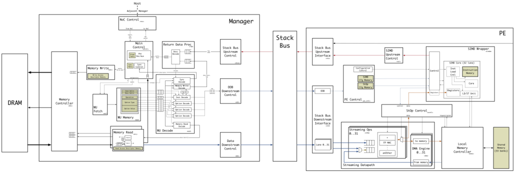
Summary

We will go into the details a little later, but here is an overview of the system.

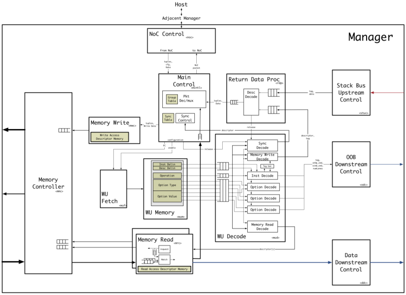
The physical system includes the DRAM stack, a management layer, a PE layer and a TSV interconnect.

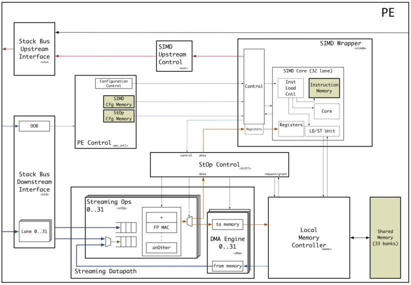


In the above diagram, the sections highlighted in dark green constitute a sub-system column. The sub-system column block diagram can be seen below.



Taking a closer look at the manager and processing engine layers.



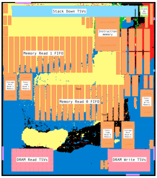


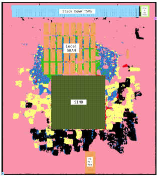
PE
Stack Bus Upstream Interface
<stu>
SIMD Upstream Control
<sui>
SIMD Wrapper
<simdw>
Configuration Control
SIMD Cfg Memory
StOp Cfg Memory
PE Control
<pe_cntl>
Control
SIMD Core (32 lane)
Inst Load Cntl
Instruction Memory
Core
Registers
Registers
LD/ST Unit
OOB
control
data
StOp Control
<scntl>
data
request/grant
Stack Bus Downstream Interface
<std>
Streaming Ops 0..31
<stOp>
Lane 0..31
+
FP MAC
anOther
to memory
DMA Engine 0..31
<dma>
from memory
Streaming Datapath
Local Memory Controller
<memc>
Shared Memory (33 banks)

Status

The baseline design effort has entered the final phase. The system is being verified using a system verilog environment.
The research effort has included C++ and python programming with tight integration to the verilog environment to provide data structure and verilog memory files.

Preliminary layouts have been performed to ensure we are meeting the goal of physically staying within the 3D column footprint. A manager and processing engine layout can be seen below.





Once this baseline design is complete, the target NN algorithms will be ported to the system and analyzed.
We are targeting Deep Neural networks (DNN) and Brain-state-in-a-box (BsB).
We expect good performance for DNNs but BsB is TBD.

Either way, some design changes, albeit small-ish are expected for each targeted NN algorithm.

So this is all well and good, but what value does our system provide?

What value does our system add

The Defense Advanced Research Projects Agency (DARPA) has a set of questions, known as the Heilmeier Catechism it asks when evaluating a proposal.

- What are you trying to do? Articulate your objectives using absolutely no jargon.
- How is it done today, and what are the limits of current practice?
- What is new in your approach and why do you think it will be successful?
- Who cares? If you are successful, what difference will it make?
- What are the risks?
- How much will it cost?
- How long will it take?
- What are the mid-term and final "exams" to check for success?

These questions were formulated by the late Dr. George H. Heilmeier, who was director of DARPA from 1975 to 1977.

So how would we answer these questions?

What are we trying to do?

Our system will provide deterministic at or near real-time performance within what we consider tolerable power and space constraints for systems employing inference on multiple disparate "useful" sized neural networks.

Now although CNNs have gotten good press recently, they are not the only DNN.

We paraphrase a quote from a Google paper [Jou17] on their Tensor Processing Unit ASIC:

> the architecture research community is paying attention to NNs, but of all the papers at ISCA 2016 on hardware accelerators for NNs, alas, all nine papers looked at CNNs, and only two mentioned other NNs. Unfortunately CNNs represent only about 5% of our datacenter NN workload

Implementations that focus on CNNs can suffer from severe degradation in performance when targeting generic types of ANN, such as locally and fully connected DNNs.

Our work focuses on edge applications employing disparate ANNs and assumes both weight reuse and batch processing do not apply, thus ***DRAM bandwidth is*** the bottleneck.

Our system will provide very high DRAM bandwidth and demonstrate the required levels of bandwidth efficiency to meet the deterministic performance requirements of edge applications.

How is it done today?

Implementations employ either Graphics Processing Units (GPU), Application-specific instruction set processors (ASIP) or Application-specific integrated circuit (ASICs).

These implementations can be shown to be effective with specific NN architectures (CNN) or the "toy examples" and to obtain high performance these implementations dependent on:

1. batch processing of requests and/or
2. ANN weight reuse

In our edge application implementing disparate ANNs, we believe these opportunities for acceleration may not exist, therefore these implementations do not provide the required DRAM bandwidth, flexibility and deterministic performance required to implement useful sized ANNs at the edge.

ASIP/ASICs

ASIP and ASICs do provide reasonable solutions in the form of performance and power when compared to GPUs.

Most, if not all focus on implementations that employ SRAM as the main processing memory.

A reliance on SRAM means that for these solutions to support all forms of useful sized ANNs, they will require a huge area for the SRAM alone such that the solution is unrealistic and this voids any area benefits.

As mentioned, there is a subset of ANNs known as CNNs which do benefit from memory reuse and thus are good candidates for SRAM-based solutions. But these solutions are memory bound when applied to other types of ANNs.

Example : Google Tensor Processor Unit [Jou17]

A good ASIC/ASIP example is the Google Tensor Processor Unit (TPU) [Jou17]. Google developed the TPU to handle their datacenter ANN processing requirements. They state that their application specific solution comfortably outperforms GPUs/CPUs by the order of 30X.
The TPU's performance is based largely on a large local SRAM (~224Mb) and to obtain high performance is dependent on:

1. large batch processing of requests and
2. ANN weight reuse

They do acknowledge that when processing MultiLayer perceptron (MLP) and Long short-term memory (LSTM) type ANNs their system is memory bandwidth bound and not processor bound. Unfortunately MLPs and LSTMs represent 95% of their datacenter processing requirements[Jou17].
It should be stated that TPU does still get high performance even for MLP and LSTM networks, but this is (likely) because of the significant amount of batch processing of requests.

So although the TPU performs particularly well for CNNs, these ANNs only represent 5% of the requests where MLP and LSTM represent the bulk of the requests.

So the paper acknowledges:

1. the TPU is DRAM bandwidth bound for the bulk of these ANNs
2. unfortunately most academic work is focused on CNN

To reiterate:

Our work focuses on edge applications of disparate ANNs and assumes both weight reuse and batch processing do not apply, thus **DRAM bandwidth is the bottleneck**.

In the case of TPU, its DRAM bandwidth is 30GBps (240Gbps) where our work expects >40TBps providing a **160X speedup**.

GPUs

GPUs are somewhat general devices and although its hard to quantify, this comes with a additional power and area burden compared to more application specific devices.
GPUs consume what we consider to be a lot of power, >100W per instance.

In general, it has been shown that a significant performance boost can be had when comparing GPUs to ASIC/ASICs [Fra16][far11][Jou17].
In the TPU paper [Jou17], GPUs were shown to not perform well in the bulk of applications that perform inference with ANNs.

The GPU community will always imply that GPU performance can always be improved with better memory usage, but in general we operate under the axiom.

> Axiom: ASIC/ASIPs will always outperform GPUs

What is new in our approach

Most researchers acknowledge that realistically, DRAM is required to meet the main storage requirements of useful sized ANNs.

This acceptance is fundamental to our mission statement.

We further believe that to support all types of disparate ANNs, we need to be able to operate directly from the DRAM memory.

This is because SRAM-based solutions assume memory locality when processing a neural network. However, when ANNs do not provide sufficient locality these solutions become DRAM bandwidth bound. If we then ensure the DRAM can feed the SRAM at the necessary bandwidth, why use an SRAM and waste the significant silicon area they require.

Our system operates directly from DRAM, but not just DRAM, 3D-DRAM.

In addition, we have designed a system that can stay within the physical footprint of the 3D-DRAM.

By ensuring we stay physically within the 3D stack, we take advantage of high density connectivity provided by TSVs. Therefore, we are able to propose a custom 3D-DRAM that exposes more of the DRAMs internal page and thus generates interface bandwidth that is of the order of 64 times that of the standard 3D-DRAM.

So the novelty of our solution includes:

- An extensible architecture that can simultaneously process multiple disparate ANNs at or near real-time with low power and real-estate demands
- A custom 3D-DRAM providing a ~10-50X bandwidth benefit compared to standard 3D-DRAM
- A system that benefits from the power and performance benefits of 3DIC technology by remaining within a 3DIC stack
- Custom data structures that allow use to operate directly out of DRAM whilst ensuring effective use of DRAM bandwidth
- Custom instructions and architecture that facilitates operating directly out of 3D-DRAM and provides high performance by allowing system functions to operate concurrently

We believe our system will provide at or near real-time performance required for systems employing multiple ANNs whilst staying within acceptable area and power limits.

We believe our system will provide greater than an order of magnitude benefit over comparable solutions.

Who cares?

Our system will target customers who have systems that are processing multiple disparate ANNs with space, power and perhaps weight limitations.

This might include aerial systems such as drones that perhaps use disparate ANNs for various system functions and cannot tolerate the 100's of watts of power that a comparable GPU-based solution would require, .

Our system will provide potentially considerable payload or time over-target improvements, whether that be a drone is delivering your family pets latest toy from amazon or analyzing threats around a embassy in a foreign hot-spot.

We will provide a ANN inference system that will not force the ANN designer to compromise on the types of ANN they employ, such as fully or locally connected DNNs.

What are the risks

3DIC

It must be said that there has not been a widespread adoption of 3DIC technology.

Our system does rely on the benefits of 3DIC and but there are currently reliability concerns with this technology.

Through-Silicon-Vias (TSV)

We utilize high density TSVs to get a significant bandwidth boost in our system. TSVs provide high levels of connectivity with relatively low power.

We believe this is relatively low risk as we could use lower density TSVs. This would affect our bandwidth although we would still provide a significant performance improvement over alternative solutions.

Resources

We currently have one researcher working on our system.

A system of this size typically take many man-years to complete. We typically run the synthesis and P&R using a typical timing library.
So performing full timing closure may expose the need for significant area increases but in our opinion the system has been designed from the start to meet system requirements.

How long will it take?

Our baseline system is currently being simulated.

It is restricted to single-precision floating point processing but the baseline system does demonstrate effective use of 3D-DRAM bandwidth and that a complex system can operate within the 3D footprint of a 3D-DRAM.

The baseline system has taken approximately 18 man-months to design and test.

The system testing is ongoing but the current status is sufficient to believe our system is truly representative.

What are our mid-term and final "exams" to check for success

We are testing our system within a system verilog environment to ensure we are exercising the various corner cases that could cause performance degradation or expose significant limitations.

We are also synthesizing our design and taking the design through place and route (P&R) to minimize the risk when the design goes through full timing closure.

So our current milestones are:

- Simulate using System verilog focusing on critical operation of DNNs whilst monitoring bandwidth utilization
- Taking the system through synthesis to provide enough confidence that the system is realizable
- Preliminary place-and-route to provide confidence in the 3DIC implementation and provide some power dissipation estimates

  The focus of this research is to take novel changes to a 3D-DRAM and create a realizable expansive system.
  A design of this magnitude would take a team 12 months or more to complete all the tasks required to produce an actual product but we'll leave that for a future design team.

Both feature addition and verification are a WIP.

So lets discuss some detailed research and implementation.

System Operations

Lets first describe the operations typically performed in our system when processing a neural network

Remember we previously showed an artificial neuron being fed by a bunch of feeder or "pre-synaptic" neurons.
The basic processing was to perform a multiply accumulate followed by an activation function.
Now many variations of ANNs employ this flow but may choose to perform their processing using integer math or perhaps even boolean math where neuron state is represented as on or off, but for our baseline system, we will focus on algorithms employing single precision floating point.

In the context of our system and AN state calculation, the basic operations to determine the state of a neuron is to:

1. Inform the Manager and PE which operation is to be performed
2. Tell the manager to access the states of the feeder neurons
3. Tell the manager to access the weights of the connections from the feeder ANs
4. Provide the feeder neuron weights and states to the processing engine
5. Tell the manager (or PE) where to store the resulting AN state back to memory

Now although the manager is the main controller in the system, it has to be provided with instructions from a host device describing the various operations performed by the system to implement a particular ANN.

We have researched an instruction architecture to describe these operations so they can be interpreted by the manager.
We are loathe to describe this as an ISA, but we do want to emphasize the instruction format implementation allows for future expansion.

In our baseline system, the manager is not responsible for performing specific algorithm operations but is responsible for coordinating the various data flow, configuration and coordination of the modules that make up our system.

The managers primary responsibility is:

- Instruction decode
- Internal Configuration messages
- Operand read
- Result write

In our baseline system, the PE is responsible for the main algorithm operations.

The PE has three major blocks:

- streaming operation function
- SIMD
- DMA/local memory controller

Lets look at what we need to do to describe each of the operations above.

Manager

Accessing of feeder AN states and connection weights

As we discussed previously, the ANN input and configuration is stored in main DRAM memory.
A part of our research is determining how to store the ANN input and parameters in such a way to effectively make use of main DRAM bandwidth.
To provide parameters for the up to 32 execution lanes within the PE, we store the AN parameters in consecutive address locations. With one read to the DRAM, we access 128 words. This provides four weights for each of the 32 ANs being processed. These weights are sent to each lane of the PE over four cycles. We will discuss memory efficiency later, but by taking advantage of the multiple DRAM banks along with pre-fetching and buffering, we are able to achieve relatively high efficiency of the available maximum bandwidth.

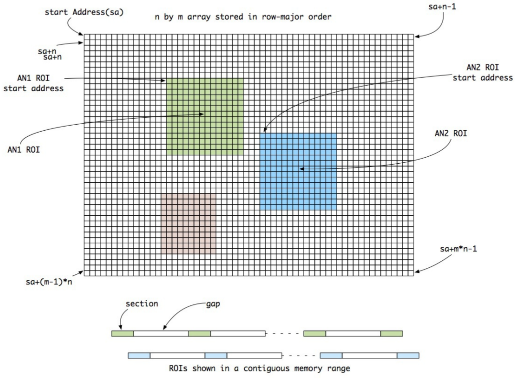
Although we are able to store the AN parameters (weights) in contiguous memory locations, providing the input state to a particular AN presents us with an interesting problem.

Now earlier we explained that DNNs are represented by layers of ANs whose pre-synaptic neurons are from the previous layer.
These previous layers represent the input to a given layer. The first layers input is the actual input to the NN.

Now the input can be represented in the form of a 2-D array of AN states. For the sake of generality, We will even consider the input array elements as AN states.

Now any given AN operates on a region of interest (ROI) within the input array.

In the figure below, we see an input to a ANN layer in the form of a 2-D array along with the ROI of two ANs.



Now we mentioned the various connection weights are stored in multiple contiguous sections.
However, its not possible to arrange the input in such a way that each ANs ROI can be stored in contiguous memory locations.
The figure above shows a typical ROI arrangement. If we consider the input array stored in row-major order, we can see an ROI is drawn from disjoint sections of memory.
These disjoint sections contain a number of AN states and the sections are separated by a gap of a number of memory addresses.
When the parameters are accessed when performing a particular operation, the memory controller

within the manager must be informed of the start address and the lengths of the sections and gaps. Now this looks problematic, and it is, but in practice groups of ANs share a common ROI. So once we solve the problem of efficiently reading an ROI from the DRAM, that ROI can be shared across a group of ANs

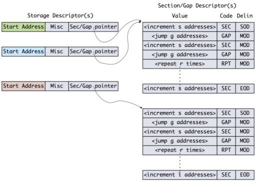
We solve this read efficiency problem by again taking advantage of the DRAMs banks and pages.

But how do you describe these ROI storage locations?

Now although disparate groups of ANs may have a different start addresses for their ROI, we see commonality in the ROI section lengths and gaps.
So for each AN group, we store that groups ROI starting address, but we point to a common set of section length/gaps. We term this structure storage descriptors.

This storage descriptor contains, amongst other things the start address of the ROI and a pointer to a section/gap descriptor. Many storage descriptors point to a common section/gap descriptor. This avoids having to have a unique section/gap descriptors for each AN group.

The figure below shows the structure of our storage descriptor. The SOD, MOD and EOD are used to delineate each descriptor in memory and stand for start-of-descriptor, middle-of-descriptor and end-of-descriptor.



Now although weights for a group of ANs will mostly be stored in contiguous memory, we still employ this structure to describe the storage but in this case the length/gap descriptor will only have a length field.

The above figure shows "miscellaneous" fields. We wont go too much into these other than to say we use additional fields to describe:

- whether the read data is broadcast or sent on a per lane basis to the PE
  - e.g. ROI is broadcast and weights are vectored
- how the DRAM is accessed
  - we control the order in which we increment the channel, bank and page addresses

Writing AN state results to memory

When the PE has processed the group of ANs, the new AN states are sent back to the manager.
The manager will store these back to DRAM most likely in the array format we described earlier.

A significant difference we take advantage of is that for any given operation, we are writing far less than we are reading.
For example, the ROI and parameters are usually vectors that will typically exceed 100 elements and in many cases much higher.
When we finish the operation, we are writing, in almost all cases one word per lane.
Now that sounds like writing back has a very small impact on performance but that's not entirely true.

When we write, we are writing a small portion of a DRAM page and the nature of the DRAM protocol means this is a very inefficient use of DRAM bandwidth.
So although the amount of data we write is small the performance impact cannot be ignored.

In addition, in many cases the results from a particular PE has to be provided not only to the PEs local manager but also to other managers. We handle this with a network-on-chip (NoC) which we will discuss later.

So how do we communicate result storage.
Well, we use the same storage descriptor mechanism mentioned previously to describe where result data needs to be stored.
However, the added complication is because the result may have to be written to other managers, we need to provide the storage descriptors for all destination managers.

Lets review what needs to be communicated by the instruction to the Manager:

- ROI Storage descriptor
- Parameter/Weight Storage Descriptor
  - Broadcast or Vectored
- Result write storage descriptor
  - include descriptors for all destination managers

PE

Streaming Operations

The operations performed by the stOp are primarily multiple-accumulate with a transfer to the SIMD or to local memory.

Even though we will focus on the AN multiply-accumulate followed by a ReLu activation function, we have built flexibility into the stOp function to allow other functions to be added

In most cases, the stOp module will operate on the AN state and weights provided by the manager and provide the result to the SIMD.

SIMD

The SIMD is a 32-lane processor with some builtin special functions, such as the ReLu operation.

The SIMD will take the result provided by the stOp and perform a ReLu. The result will, in most cases, then transmitted back to the manager.

Configuration

To configure these operations, we send two pointers to the PE. These pointers index into a small local memory which provides a program counter (PC) to the function to be performed by the SIMD and a configuration entry for the operation to be performed by the stOp.

Our PE is able to perform its operation concurrently on 32-lanes. However, there are cases when less than 32-lanes will be employed. This may occur if the number of ANs being processed is not modulo-32. In this case, we also need to provide the number of lanes being processed for any given operation.
In addition, we also send the length of the vector of operands being sent from the manager to the PE.

Lets review what needs to be communicated by the instruction to the PE:

- stOp operation
- SIMD operation
- Number of active lanes
- Operand Vector length

So lets look at our Instruction format.

Instruction Format

To support ANN processing, our instructions have to include information to control the following operations:

- To the Manager
  - ROI Storage descriptor
  - Parameter/Weight Storage Descriptor
    - Broadcast or Vectored
  - Result write storage descriptor
    - include descriptors for all destination managers
- To the PE
  - stOp operation
  - SIMD operation
  - Number of active lanes
  - Operand Vector length

Although the focus of this work is effective use of a 3DIC system to process an ANN, an expansive system needs to also provide "support" features for tasks such as downloading ANN parameters from a host system and downloading inputs and uploading outputs.

Our system has included the infrastructure to include these support tasks and although not all have been included in the design and verification effort the infrastructure provided in our expansive architecture means adding these features should not impose a significant burden on available silicon.

So we'll focus here on processing an ANN and discuss the "support" features later.
Later we will review what has been done and leave additional features for future work.

Instruction Types

We have defined two instruction types, the **compute** instruction which is used for controlling processing of an ANN and a **configuration** instruction which controls the support features such as instruction download, parameter download, result upload etc.

An instruction is coarse grained in as much as it provides the information to perform all the tasks associated with a high level task. For example, to process a group of ANs, the system needs to know where the pre-synaptic AN states are stored, where the connection weights are stored, what activation function should be used and where should the AN state be stored.
To provide this finer grained information, an instruction is partitioned into sub-instructions we call descriptors.
An instruction can contain one or more descriptors, and as mentioned previously n thie case of a compute instruction, each of these descriptors contain the information to control a specific operation associated with the processing of a group of ANs.
To provide an expansive architecture, we do not limit the number of descriptors that can be included in an instruction.
For want of a better word, we might consider this a variable length instruction word or VLIW.
In addition, there is scalability or growth built into the various fields in the descriptors to allow additional features to be added in the future.

Remember, we want to process a group of ANs in parallel and the group size is related to the number of execution lanes which for our baseline system is 32.
So a group can be anywhere from 1-32. It should be said that unless we consistently have group sizes approaching 32 the system performance will be poor.
So lets take a closer look at the format of an instruction to compete this task.

For the compute instruction, we typically have four descriptors:

1. Operation
2. Memory read for pre-synaptic states we call operand stream 0
3. Memory read for connection weights we call operand stream 1
4. Memory write for AN state storage

Note: We will refer to an operand stream as an argument.

Now the instruction is actually an n-tuple where the tuple elements are descriptors and the number of elements can vary based on the task being performed.

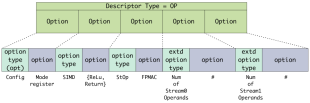
Below we see the format of a 4-tuple instruction which we use to perform an activation calculation for a group of neurons.

Operation Descriptor	arg0 Read Descriptor	arg1 Read Descriptor	Result Write Descriptor

Now within a descriptor, we need to describe the various options such as storage descriptor pointer, number of operands etc..

Again, we employ a n-tuple format where the first tuple element describes the descriptors operation followed by an m-tuple whose elements contain the options required for the operation.

These option elements are a two-tuple with option and associated value.



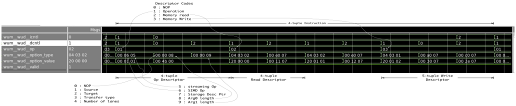
The options currently supported with their associated codes are:

Option Type	Code	Comment
NOP	0	Used to pad physical implementations
Source	1	
Target	2	Which stream for memory read operations
Broadcast/Vector	3	How read data is sent to target
Number of Lanes	4	
Streaming Operation	5	
SIMD Operation	6	
Memory Access (storage descriptor pointer)	7	Storage descriptor associated with read/write operation
Argument 0 number of operands	8	
Argument 1 number of operands	9	
Configuration Sync	10	Used to synchronize Host to system
Configuration Data	11	Used to carry programming or I/O data
Status	12	General status messages

We have the concept of a "normal" option and an "extended" option. The extended option is used for large option values. For example, the extended option is used for storage descriptors and number of operand option types.

Below is a snapshot of an instruction transaction being transferred between the instruction memory (WUM) and the instruction decoder (WUD) modules. In our implementation, the physical interface between the WUM and WUD carries three option tuples per cycle. A descriptor may not have a modulo-3 number of option tuples, in this case we employ the NOP option tuple to pad the bus.

You can see examples of normal and extended tuples.



In the waveform you can see how our implementation accommodates our variable length instruction and descriptor tuples. To delineate both the instruction and individual descriptors, we chose to implement side-band signals. These delineation signals are referred to as icntl and dcntl for the instruction and descriptor tuples respectively.

Note: The NOP option type can be used to delineate tuples

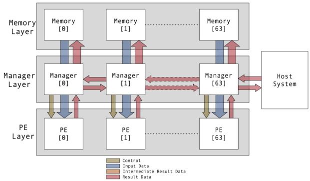
The delineation signals are two bits wide and are coded as follows:

CNTL	Code
Start	1
Middle	0
End	2

OK, so we have been discussing , PEs, operand steams, operations, instruction formats, but how does this various information move around our system.

System Overview

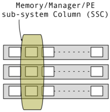
Lets take a high level look at our 3D system

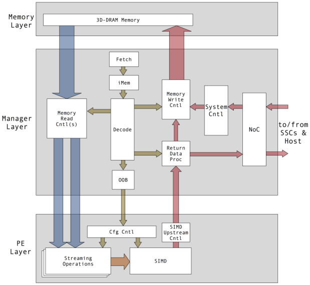


The diagram above shows:

- How control is passed from the manager to configure the PE operations
- PE input data in the form of AN states being read from main memory and passed to the PE
- Result data input in the form of AN state updates being passed from the PE to Manager
- Result data being replicated to other Managers (if required)
- Result data being written back to main memory

Lets take a closer look at one of our sub-system processing columns





Manager

Operation Decode

In the diagram, instructions are read from instruction memory and passed to the instruction decoder.

The operation tuple is decoded and a streaming operation (stOp) pointer and a SIMD operation pointer are sent to the PE inside an OOB control packet.

The stOp pointer specifies what streaming operation is to take place on the data directly streamed to the PE.
In our baseline system, typically this would be a floating-point multiply accumulate on two arguments, the pre-synaptic neuron states and the pre-synaptic weights.

The SIMD pointer is essentially a program counter that will be invoked when the stOp result is passed to the SIMD.

Note that other types of stOp includes a NOP with a destination of local memory. This allows us to transfer block of instruction or data from the manager to the PE.

Argument Decode

The instruction also includes argument descriptors. These descriptors include a storage descriptor pointers that point to a storage descriptor stored in local memory that encodes where data should be read from for the one or two arguments that will be streamed from DRAM to the stOp within the PE.
In the case of a AN activation calculation, there are two arguments, the pre-synaptic neuron states and the pre-synaptic weights.
The read storage descriptor pointers are passed to the Memory Read Controllers (MRC). The MRCs read the actual storage descriptor from their local memory and immediately start sending read commands to the memory via a Main Memory Controller (MMC). The MMC is not shown in the diagram but essentially takes the memory read requests and converts them into the DRAM read protocol.

As soon as read data is sent back to the MRC via the MMC, that data is aligned with the downstream bus and sent to the 32 Streaming Operations inside the PE.

Result Data processing

Typically, the instruction also includes a result data write descriptor pointer. In fact, if the result data is to be sent to the local Managers main memory and also replicated to other Managers main memory, there will be as many write storage descriptor pointers as there are destination managers.

These write storage descriptor pointers are sent to the return data processor (RDP). The RDP waits for a result packet to be return upstream from the PE. The information sent to the RDP also includes a "tag" to allow result data to be returned out-of-order. In practice, most data will be returned from the PE in the order the operations were sent to the PE.

There may also be status messages sent from the PE to the manager but these will not be discussed.

The return data is sent from the PE to the Manager in the form of a packet. The packet includes the tag along with the result data.
The packet is received by the Stack Upstream Controller (STU) and passed to the RDP. The STU is not shown in the above diagram.

The RDP matches the tag and examines the storage descriptor pointers previously provided by the decoder. If one of the storage descriptor pointers points to local main memory, the RDP passes the data along with the storage descriptor pointer to the memory Write Controller (MWC). If one or more of the storage descriptor pointers points to other Managers main memory, the RDP forms an NoC packet and sends the data along with the storage descriptor pointer to the NoC controller.

Simultaneously, the Manager may receive packets from other Managers who had a storage descriptor pointer pointing to this Managers main memory. These are received by the noC and passed to the Main Controller (MCNTL). The MCNTL passes the data along with the storage descriptor pointer to the Memory Write Controller (MWC).

Memory Write Controller

The Memory Write Controller (MWC) receives data from to sources, the NoC via the MCNTL and the RDP.

In both cases, the MWC read the actual storage descriptor from their local memory and immediately start forming data that will be written back to main memory.

When the data is formed, a write command is sent to the memory via the MMC. Again, the MMC is not shown in the diagram but takes the memory write requests along with the data and converts them into the DRAM write protocol.

The MWC can only operate on one of the two sources at any one time. However, there are four 4096-bit holding registers where data is formed prior to the write request.

The holding registers have the potential in future to allow aggregation of data from one or more operations to allow a coalesced write back to main memory.

PE

Configuration

A configuration controller within the PE (PE_CNTL) takes the OOB packet from the Manager and extracts the stOp and SIMD operation pointers.

The stOp pointer is used to point to a local stOp configuration memory. The memory contains the various configuration data required by the streaming operation controller (stOp_CNTL). The stOp_CNTL is not shown.

The stOp_CNTL configures the:

- Operation type
- Number of active execution lanes
- Source of the argument data, which can be downstream data from the manager or from the small local SRAM
- Destination of the result data, which can be the SIMD or the small local SRAM

The SIMD operation pointer is sent to the SIMD.

Streaming Operations

The streaming Operations (stOp) are designed to operate on data passed from the Manager at or near line-rate. If line-rate cannot be maintained, a flow-control mechanism is employed to slow the data from the Manager.

Once the stOp has processed the data, it passes the result to the SIMD. Note in some cases the result can be placed in local SRAM or sent to both SIMD and SRAM.

It should also be stated that while the stOp is processing the current data, the SIMD may be operating on the result of the previous operation. It is expected the SIMD will have completed the previous operation before the stOp completes the current operation, but again, if necessary a flow control mechanism between SIMD and stOP will be engaged if the SIMD is not ready.

SIMD Operation

The SIMD takes the result data and performs the operation starting at the program counter (PC) indicated by the SIMD operation pointer provided by the PE_CNTL.

The stOp provides the result to the SIMD via a local register. The result is also written, in most cases to the small local SRAM.

The SIMD performs the specified operation on the data provided by the stOp.

In most cases this will be the AN activation function and in our baseline system is the Rectified Linear function.

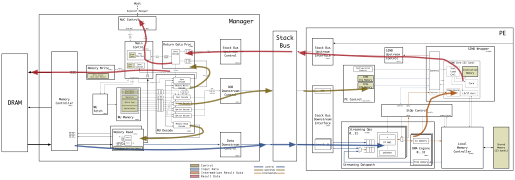
When the SIMD has completed its operation, it passes the result to the SIMD Upstream controller to be returned to the Manager.

Return Data

The SIMD Upstream Controller (SUI) takes the data and encapsulates it in an Upstream packet. Included in the packet is the tag required by the Return Data processor within the Manager

Sub-System Column Flow (SSC)

Lets put the above flow description in the context of the detailed sub-system column block diagram shown previously.



Its important to understand that the system control and data flows are asynchronous. The system is able to spawn control of all the sub-blocks and initiate operations without concern to any specific ordering.

All the interfaces between modules include a standard interface that includes a flow control mechanism that allows a source module to start sending before the destination module is ready. This flow control is initiated by the destination indicating to the source it is ready.

For this reason, almost all of the interfaces between modules employ this "standard" interface along with FIFOs. This allows for some finite latency between the destination sending the ready signal and the source receiving the ready signal.

Lets take a look at those major control and data buses.

Main Buses

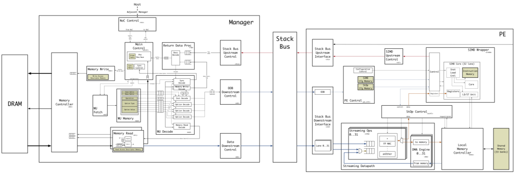
Clearly with a system of this size we have many buses between the various modules, but the buses which are most important are our 3D inter-layer buses. These 3D buses are constructed from through-silicon-vias (TSVs) and are the main communication interfaces between the Manager and DRAM and between the Manager and PE.

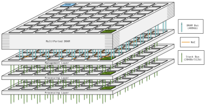
These 3D buses are used to transport configuration and data between the Manager and PE and data between DRAM and Manager and are instrumental in providing the high performance of our system.

Remember, we are trying to research whether a meaningful NN accelerator system can exist within the 3D footprint of a 3D DRAM. If such a system can exist within a 3D footprint, we can take advantage of the low energy and very wide, and thus high bandwidth of these TSV based buses.

Just to remind us, in our case a "meaningful" system is one that provides the necessary performance, flexibility and scale to support a system employing multiple meaningful sized neural networks.

Lets take another look at our 3D system





Manager/PE Buses

In our system, we employ a very wide bus that is split between communication from manager to PE and communication between PE and Manager. The idea is that this bus can be partitioned based on the needs of the algorithm. In our baseline system, one of the primary computations is the AN state calculation, so we partition the bus mainly to accommodate the transport of the AN pre-synaptic states and connection weights to the PE layer.

Manager to PE

We refer to this interface as the downstream bus. It is actually separated into a downstream configuration bus and the downstream data bus.

The configuration, or as we refer to it the downstream out-of-band bus (OOB) carries all option tuples from the instruction to configure both the stOp module and the SIMD module.

Typically, the downstream data bus carries the operand streams that contain the pre-synaptic states and connection weights for a AN state calculation.

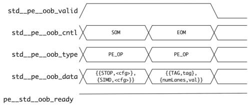
However, this downstream data bus can also be used to carry data to the local memory or the SIMD memory. These transfers can include SIMD functions or data used when the streaming operation is configured to use local PE memory for one of its arguments.

Downstream OOB Bus

The downstream OOB bus is primarily designed to carry control information. These control packets are variable length and carry option tuple data from the originating instruction. The variable length is again delineated using side-band signals.

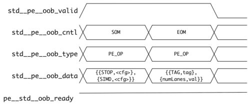
The downstream OOB bus also contains a tag to allow possible out-of-order execution although this is not currently employed.

It should be mentioned that all internal interfaces include flow control signals that operate in conjunction with input FIFOs. This allows the system to pipeline operations and data to maximize DRAM and system bandwidth.



Downstream Data Bus

The downstream data bus is used to transfer bulk data. This data is most often operand data for the AN activation operation but can also be used to transfer data to the SIMD memory for SIMD function operations or to the local memory for later operations involving the stOp modules or the SIMD.

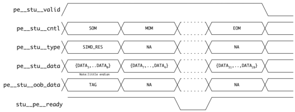


Our system provides a per lane flow control and a per lane valid. This allows the two lanes to become out of sync where at any clock the two pieces of data on the two lanes may not be the two arguments that take part in the same operation in the stOp. This flexibility allows us to smoothly accommodate the out-of-alignment memory access and not wait for the data from the memory to synchronize. It should be noted that the two arguments get realigned in the stOp receive FIFO.

Upstream Bus

The upstream bus is primarily designed to carry result data from the SIMD or stOp back to the manager. The upstream packet also contains the tag so the result can be matched with the associated operation.

The tag is matched with the operation and storage descriptors. If the storage descriptors are associated with other managers, the result data is replicated and to the local manager memory and sent over the NoC.



Manager/Memory Interface

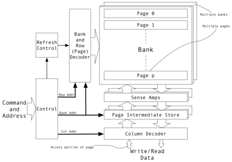
The memory bus is essentially a customized version of a typical DRAM interface.

We assume you have a rudimentary knowledge of DRAMs but as a reminder, accessing these "typical" DRAMs involves opening a page in a bank, reading or writing a portion of the contents of the page then closing the page.

Typically a bank may contain of the order of a few thousand pages and a page may contain of the order of a few thousand bits.

Once the page is open, the user accesses a portion of the requested page over a bus. With PCB based DRAMs the bus might vary from four to 16 bits wide, but with 3D DRAMs, such as HBM the bus might be up to 128 bits wide.

Below is a block diagram of a typical DRAM.



Unlike SRAM, DRAM storage involves storing a charge on the gate of a MOSFET.

This method of storage imposes significant delays from when a page is opened to when it can be accessed. If we were accessing a single bank, this means if the user needs to access a different page, the bank cannot provide data during this period. This would result in a significant impact to usable bandwidth. This situation is avoided because the DRAM has multiple banks, and while one banks page is being changed the user accesses a different bank.

Additionally, the IO data interface to the DRAM is DDR and a read or write involves multiple cycles of data.

Now there are some other features of DRAM access protocol, such as refresh and mode configuration, which we wont go into but all in all, the DRAM interface protocol requirements means ***accessing a DRAM is non-trivial***.

Now we mention earlier that often researchers employ either only SRAM or use SRAM as an intermediate store or cache. This often means the SRAM utilizes a significant portion of the designs silicon.

In addition, we believe this intermediate SRAM cache is only effective with certain forms of ANN. This means a more general system cannot rely on the effectiveness of the SRAM cache and needs to make optimal use of the raw DRAM interface.

We believe our system is able to maintain acceptable levels of DRAM bandwidth utilization.

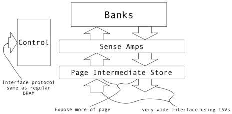
DRAM Customizations

Now we previously mentioned a "customized" DRAM. We believe if a system can remain within the 3D footprint of the DRAM, we can generate more raw bandwidth than traditional DRAM technology.

Expose More of the DRAM page

We achieve the increase in bandwidth by proposing that the DRAM expose more of its currently open page.

Without the limitations of having to transfer data beyond the chip stack, we believe we can expose a larger portion or perhaps all of the page and by staying within the 3D footprint we can transfer data over a very wide through-silicon-via interface.



Now we still have to deal with the DRAM protocol, but this research demonstrates a design that can manage the DRAM and provide data to the PE such that the system makes effective use of the available raw DRAM bandwidth.

For our baseline system, we are using a modified version of the Tezzaron DiRAM4.

The original DiRAM4 is a 64-port 3D DRAM with 32 banks per port.
Each bank has 4096 pages with 4096 bits per page. An individual read or write to the DiRAM4 is to a 64 bit portion of the page. This 64 bit cacheline is accessed over a 32-bit interface using a burst of two DDR transaction.

This research proposes exposing the entire 4096-bit page over a 2048-bit bus again using a burst of two DDR transaction.

Add Write Mask bits

When processing an ANN, to compute the activation of an individual AN involves reading the pre-synaptic AN activation's and the weights of the connections between the pre-synaptic ANs and the AN being processed. The activation of the processed AN is written back to memory.
The ratio of reads to writes is high, 100's or 1000's to one. Therefore, we often need to write a portion of the page back to memory. To avoid a read/modify/write, we propose adding a write data mask to the DRAM write path.

Summary

A preliminary discussion with the Tezzaron technical lead suggested exposing more of the page is feasible, therefore this work is based on the DRAM customizations outlined above.

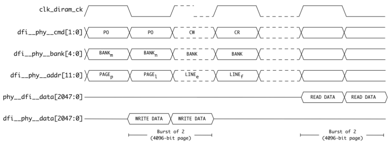
But to be clear, this work does not research the impact of this change to the actual DRAM.

This work answers the question:

> "if such a device is available, can we employ it within a useful ANN system"

DRAM Waveform

An example DRAM waveform is shown below.



The commands used when accessing the DRAM are as follows:

Acronym	Command	Description
PO	Page Open	Open page 'addr' in bank
PC	Page Close	Close the currently open page
PR	Page Refresh	Refresh page in bank

Acronym	Command	Description
CR	CacheLine Read	Read section 'addr' of currently open page in bank
CW	CacheLine Write	Write section 'addr' of currently open page in bank

Future Work

Host Communication

Investigate full support for a Host system, such as compiler control, system synchronization etc.

TSV Transceivers

In current study, there have been no provisions for TSV transceivers. The assumption is that CMOS logic drivers are adequate over such a short distance.

FIFO Depth

The current design a relatively conservative with FIFO memory depth so there is an opportunity to reduce the silicon size by evaluating lowering FIFO depths.

NoC

Full Feature Support

Complete the design of NoC packet types for configuration and status messages including:

- System Synchronization messaging
- Instruction Download
- Input Download

PE

Precision

Investigate support of lower precision PEs such as 16-bit floating point and the impact to the current design.

Architecture

Consider feeding different architectures such as systolic array, reduction operation etc

Compute Bound Algorithms

Investigate employing local SRAM and how our system can support the DRAM to SRAM needs of those systems.