SANDIA REPORT
SAND2022-14088
Printed September 2022

Sandia National Laboratories

Modeling Analog Tile-Based Accelerators Using SST

Ben Feinberg, Sapan Agarwal, Mark Plagge, Fredrick H. Rothganger, Suma G. Cardwell, Clayton Hughes

Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

{bfeinbe, sagarwa, mplagge, frothga, sgcardw, chughes}@sandia.gov

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@osti.gov
Online ordering:	http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.gov
Online order:	https://classic.ntis.gov/help/order-methods

ABSTRACT

Analog computing has been widely proposed to improve the energy efficiency of multiple important workloads including neural network operations, and other linear algebra kernels. To properly evaluate analog computing and explore more complex workloads such as systems consisting of multiple analog data paths, system level simulations are required. Moreover, prior work on system architectures for analog computing often rely on custom simulators creating significant additional design effort and complicating comparisons between different systems. To remedy these issues, this report describes the design and implementation of a flexible tile-based analog accelerator element for the Structural Simulation Toolkit (SST). The element focuses on heavily on the tile controller—an often neglected aspect of prior work—that is sufficiently versatile to simulate a wide range of different tile operations including neural network layers, signal processing kernels, and generic linear algebra operations without major constraints. The tile model also interoperates with existing SST memory and network models to reduce the overall development load and enable future simulation of heterogeneous systems with both conventional digital logic and analog compute tiles. Finally, both the tile and array models are designed to easily support future extensions as new analog operations and applications that can benefit from analog computing are developed.

Acknowledgment

The authors acknowledge financial support from the DOE Advanced Simulation and Computing program. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

CONTENTS

Nomenclature 8

1. Introduction **9**

2. SST Analog System Model **11**

2.1. Tile Component 11

2.1.1. Tile Configuration 12

2.1.2. Tile Operation 13

2.1.3. Tile Memory 16

2.1.4. Implementing New Tile Types 17

2.2. Array Subcomponent 18

3. Conclusion **19**

References **20**

LIST OF FIGURES

Figure 2-1. Overview of the modeled tile-based analog architecture in SST. 11
Figure 2-2. SST configuration example of a single tile system with a test driver. The network model is not included in the interest of space. 14

LIST OF TABLES

Table 2-1. Configuration parameters for the tile and testDrive components. 12
Table 2-2. States for tile state-tracking logic. 15
Table 2-3. Tile functions designed to be overridden for new tile types. 17

NOMENCLATURE

CNN Convolutional Neural Network

DNN Deep Neural Network

FC Fully Connected

MVM Matrix Vector Multiplication

NIC Network Interface Controller

SST Structural Simulation Toolkit

1. INTRODUCTION

Over the past decade there has been renewed interest in analog computing to provide further performance improvements as digital scaling trends have slowed. Analog kernels have been proposed for a wide variety of applications including neural network training and inference [1, 2], graph analytics [3], and scientific computing [4, 5] and show potential for efficiency improvements beyond what is currently feasible with digital computing. One important challenge with new computing paradigms, including analog computing, are simulation tools to go from a computational kernel to a usable system. Even in a system where all of the major computations will be performed in analog, significant digital logic is required for programmability, system control, and storage and communication of intermediate results. Moreover, most applications of interest cannot perform all of the computations in analog and require some degree of digital processing for correct results. Therefore, system-level simulators incorporating digital processing, memories, and networks are required for the successful development of analog computing.

Prior work on system-level simulation of systems using analog computing typically use customized simulators that connect to, or use results from, existing tools for digital components. We identify three potential issues with these custom simulators. First, custom simulators increase development effort since each new researcher in this field needs to create their own simulator. Not only does this constant re-development of simulators create wasted effort, it also creates challenges when comparing different system proposals since the key configuration parameters reported may not be directly comparable between different works. A sufficiently extensible simulation framework, such that each new researcher doesn't feel the need to reinvent the wheel, could improve research on analog computing in the same way that standardized CPU simulators have benefited the computer architecture research community.

Second, simulations that attempt to integrate components from multiple different tools can be error-prone and obscure insights which cut across multiple tools. Integrating tools that were not designed for interoperability can create issues where small differences in the assumptions of different tools lead to evaluations of systems that differ from what the researcher intended. This is especially true for evaluations where the tools are not directly integrated and instead use results from separate tool runs as inputs to the final evaluation model. These challenges are generally avoidable through careful effort, but making evaluations less error prone is generally beneficial. Related to this issue, optimizations that cut across multiple tools can be harder to discover because the impact is spread across different sub-results in the evaluation.

Third, the simplifying assumptions used to reduce the development overhead of custom simulators can obscure important details. For instance, the controller that orchestrates computation is often only simulated implicitly by high-level logic within the simulator. This means that operations which may be computationally expensive, can be implicitly included in the simulation without being appropriately evaluated. The reverse is also true, simple optimizations that can be performed on the controller may not be evaluated because of concerns about their complexity and difficulty in evaluating these optimizations within existing simulation infrastructures. Another consequence of implicitly simulated controllers is a lack of research on programming models and run-time behavior in non-ideal conditions. Although there has been work on programming models for analog computing [6], overall research is still limited and primarily confined to system designers. A related challenge is how the system will operate in less than ideal circumstances. Many simulators assume data flows which are fixed at programming time meaning the system cannot adapt to run-time conditions. For instance, many candidate devices for analog computing experience state drift which can reduce solution quality over time [7]. This process is often stochastic meaning a system needs methods to detect intolerable levels of drift and adapt the data flow. For some applications shutting the system down and reprogramming may be feasible, but applications requiring high availability—object detection in an autonomous vehicle for instance—need to handle drift without downtime.

To begin remedying the above issues, we have developed a model of a tile-based analog accelerator for the Structural Simulation Toolkit (SST) [8]. SST is a parallel discrete-event simulator designed for extensible simulation of a wide range of computing systems and contains models for many components including CPUs, GPUs, memories, networks, and RTL simulation. The tile-based analog model, contained in a new SST *element*, relies on existing SST models for memories and networks on chip to avoid the development costs described above. Importantly, the models described in the next chapter do not solve all of the issues mentioned above. For instance, this model still uses a data flow between tiles which is fixed at configuration time. However, because the model is designed to be highly flexible, and uses standard SST communication primitives, run-time reconfigurability and similar topics can be explored in future work using existing SST models.

2. SST ANALOG SYSTEM MODEL

To model analog tile-based accelerators in SST, we developed a new SST element for analog compute tiles. The goal of this element is to leverage the existing SST models for memories and on-chip networks and focus on the unique compute properties of analog compute tiles. The modeled architecture of a tile-based analog architecture is shown in Figure 2-1 where the orange blocks indicate existing SST elements, and the green, blue, and gray blocks indicate the new components and subcomponents developed that will be described in this chapter. Each tile broadly consists of four components, one or more analog compute arrays, a local controller which drives inter-tile communication, array operations, and synchronization, a local memory to store inputs and outputs, and a connection to the tile network on chip.

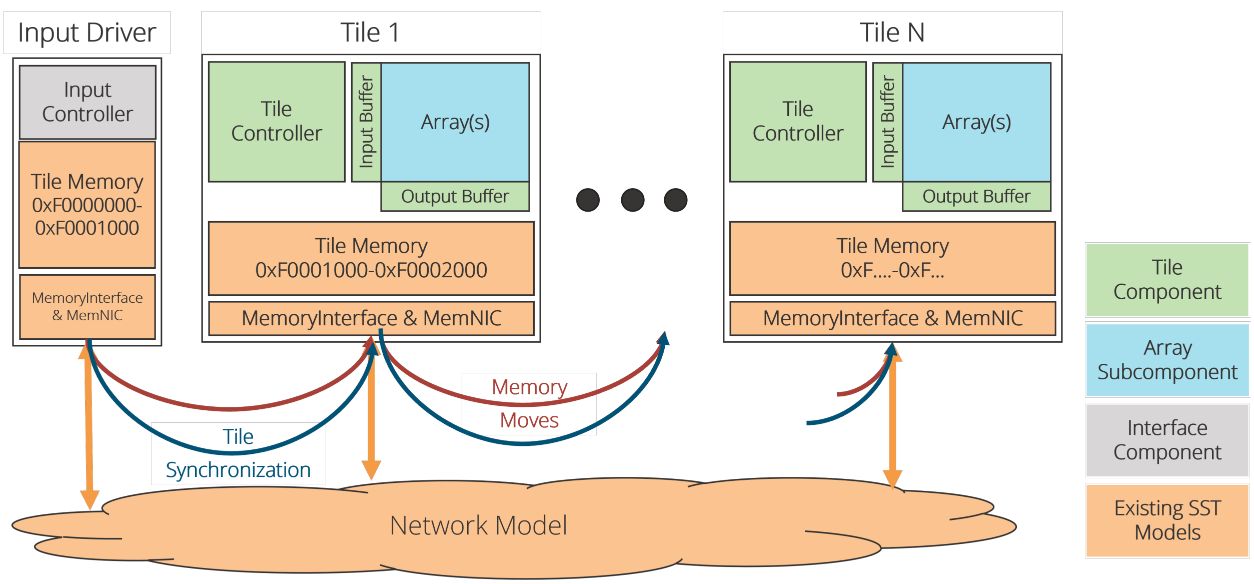

Figure 2-1. Overview of the modeled tile-based analog architecture in SST.

2.1. Tile Component

The tile component is the primary model in this SST element. It implements internal tile state tracking and inter-tile synchronization operations. Conceptually, the tile model primarily operates as a memory request generator moving data from *producer tiles*, reading/writing data to/from the array input/output registers, initiating array operations, and signaling *consumer tiles* on completion. In addition to the tile component this section discusses the *testDriver* component, a simplified version of the tile model that presents inputs to a tile-based accelerator and reports the final outputs.

Table 2-1. Configuration parameters for the tile and testDrive components.

Parameter	Description
clock	Clock frequency of the tile.
tileID	Global tile ID used for inter-tile communication. Must be unique.
tileMemoryStart	Starting global address of the tile memory. Tile assumes it has full control over this region. Minimum memory region size must be able so store a full copy of the arrayIn and arrayOut registers.
tileMemoryEnd	Ending global address of tile memory.
mmioOffset	Address offset for MMIO inter-tile communication, must be identical across all tiles.
numArrays	Number of distinct arrays in the the tile.
arrayInSize	Length of input vector. Implies number of array rows.
arrayOutSize	Length of output vector. Implies number of array columns.
operandSize	Size of input/output operand in bytes.
producerTiles	List of tileIDs of tiles which produce results for this tile.
producerAddresses	List of address of starting addresses for each producer tile.
producerCounts	List of how many operands to load from each producer. If variable over time use the maximum number of operands for each producer.
consumerTiles	List of tileIDs of tiles which consume results from this tile.
consumerAddresses	List of address of starting addresses for each consumer tile.
testDriver Specific Inputs	
numInputs	Number of input vectors to supply to the tiles.
inputArray	Flat array of the inputs for the tiles.

2.1.1. Tile Configuration

Table 2-1 lists the major configuration parameters for the the tile component. These define a tile in terms of a tile ID number, address range, and the number and size of the compute arrays contained within the tile. Details about tile memory will be discussed in a later section, for the purposes of this discussion it is important to note that the tile model assumes that nothing else in the system will modify the locations within tile memory space. Finally, the tile configuration specifies the tiles that produce inputs and consume results for a given tile.

2.1.1.1. Test Driver Configuration

To provide a single interface for a series of tiles this element also contains a *testDriver* component. The testDriver is a simplified version of the overall tile model without complex state tracking logic or array models. The testDriver has two additional parameters shown at the bottom of Table 2-1 to track how many inputs will be tested in a simulation run.

For future development of heterogeneous systems where a digital CPU will provide inputs for a tile-based accelerator, this driver model can be extended to be provide a unified interface for the tile-based components. In this case, rather than specifying a list of inputs in the configuration, the driver will present an interface for other components of the system to provide inputs and read outputs using MMIO communication.

2.1.1.2. Configuration Example

Figure 2-2 shows a minimal configuration of a single tile system with a testDriver. For simplicity, this configuration does not specify tile-local memory as a separate component but rather allocates the testDriver and tile with portions of a single shared memory. In the interest of space, this configuration does not specify a network model, in a full configuration the three network interfaces (*driver_nic*, *tile1_nic*, *mem_nic*) would be connected to some form of network model. As shown, the testDriver is set as Tile 0 in the system and is the producer to the first compute tile, and the consumer of the final compute tile, both Tile 1 in this example. This example also uses a simplified test array subcomponent which adds 1 to each input value, configuration of array subcomponents will be discussed in the next section.

2.1.2. Tile Operation

Basic tile operations are tracked using per-producer and per-array state trackers. Importantly, the tile processes the states of each producer and array independently; therefore, unless a tile's functionality requires synchronization different components of the tile can run at different rates. The states and the state transition criteria are summarized in Table 2-2. For simplicity, neither state tracker goes back to the idle state after all memory requests for a given state have been issued, instead the system retains that state until all requests have finished and then transitions to the next state.

The producer state trackers use a simple two-state design. From the perspective of the a tile a producer is either ready for data to be copied (*ProducerReady*) or the most recently available data has been copied and the tile is waiting for new data to copy (*ProducerIdle*). A producer tracker transitions to the *ProducerReady* state when it receives a ready signal from a producer and the tile begins sending copies to the the signaling producer. As noted above, once all the copies have been issue the tile remains in the *ProducerReady* until all copy requests have completed. Upon completion of the copies the tile signals the producer that it has finished copying the data and returns to the *ProducerIdle* state.

The array state trackers are conceptually similar, just with significantly more states as shown in Table 2-2. The array state trackers begin in the *ArrayIdle* state. When all producers that produce inputs for a given array have been copied from the producer the arrays transitions into the *ArrayLoading* state. In the *ArrayLoading* state the tile reads the input operands for the next tile operation from local tile memory and places those values in the array's input registers. The *ArrayLoading* state could be further optimized by starting to load values into the array input registers as the copies from producers complete at the cost of additional state tracking logic. This optimization will be explored in future work. Additionally, in some tile implementations, the data copied from the producer could be directly placed in the array input registers rather than requiring

```
driver = sst.Component("testDriver", "analogComputeArray.testDriver")
driver.addParams({
    "tileID": 0,
    "tileMemoryStart": 128,
    "tileMemoryEnd": 512,
    "mmioOffset": 512*1024*1024,
    "inputSize": 4,
    "outputSize": 4,
    "numInputs": 2,
    "operandSize": 1,
    "producerTiles": [1],
    "producerAddresses": [1024],
    "producerCounts": [4],
    "consumerTiles": [1],
    "consumerAddresses": [1024],
    "inputArray": [1,1,2,2,3,3,4,4],
})
iface0 = driver.setSubComponent("memory", "memHierarchy.standardInterface")
driver_nic = iface0.setSubComponent("memlink", "memHierarchy.MemNIC")
driver_nic.addParams({"group" : 0, "destinations" : [1,2], "network_bw" : "25GB/s"})

tile1 = sst.Component("tile1", "analogComputeArray.computeTile")
tile1.addParams({
    "tileID": 1,
    "tileMemoryStart": 1024,
    "tileMemoryEnd": 1024 + 8,
    "mmioOffset": 512*1024*1024,
    "numArrays": 1,
    "arrayInSize": 4,
    "arrayOutSize": 4,
    "operandSize": 1,
    "producerTiles": [0],
    "producerAddresses": [128],
    "producerCounts": [4],
    "consumerTiles": [0],
    "consumerAddresses": [128],
})
iface1 = tile1.setSubComponent("memory", "memHierarchy.standardInterface")
array1 = tile1.setSubComponent("array", "analogComputeArray.TestComputeArray")
tile1_nic = iface1.setSubComponent("memlink", "memHierarchy.MemNIC")
tile1_nic.addParams({"group" : 1, "destinations" : [0,2], "network_bw" : "25GB/s"})

memctrl = sst.Component("memory", "memHierarchy.MemController")
memctrl.addParams({"clock" : "1GHz", "addr_range_end" : 512*1024*1024-1})
memory = memctrl.setSubComponent("backend", "memHierarchy.simpleMem")
memory.addParams({"mem_size" : "512MiB" })
mem_nic = memctrl.setSubComponent("cpulink", "memHierarchy.MemNIC")
mem_nic.addParams({"group" : 2, "destinations" : [0,1], "network_bw" : "25GB/s"})
```

Figure 2-2. SST configuration example of a single tile system with a test driver. The network model is not included in the interest of space.

Table 2-2. States for tile state-tracking logic.

State	Action	Transition Criteria
Producer States		
ProducerIdle		Signal from producer tile.
ProducerReady	Copy data from producer tile to tile memory.	All copies completed.
Array States		
ArrayIdle		All inputs for the array have been copied.
ArrayLoading	Load inputs from tile memory to array input register.	All inputs loaded.
ArrayReady	Start array operation.	Array operation started.
ArrayComputing	Perform array operation.	Array operation completed.
ArrayPostProcess	Perform digital operations on results in array output register(s).	Post processing completed.
ArrayStoring	Store post-processed values to tile memory.	All stores completed.
ArrayFinished	Same as ArrayIdle, used after final array operation is completed (Simulation specific).	Simulation ends.

the value be written to tile memory in the *ProducerReady* state, and subsequently read in the *ArrayLoading* state. The current implementation keeps these separate since several proposed tile-based architectures have separate data paths for these two operations. This implementation also provides greater flexibility of implementation for tile memory as will be discussed below. However, adding an option to forward data from the copy operation to the array input registers will be explored in future versions.

Once all the inputs have been loaded into the array input registers, the array transitions into the *ArrayReady* state indicating that an array operation can begin. In the *ArrayReady* state, the tile signals an array to start the computation and transitions into the *ArrayComputing* state. The tile takes no actions during the *ArrayComputing* and simply waits for the array operation to complete, at which point the array transitions into the *ArrayPostProcess* state. *ArrayPostProcess* can implement a wide range of functions that will be performed on the outputs of the array, for example, reduction and pooling operations for neural network layers. For operations that require results from multiple arrays within the same tile, *ArrayPostProcess* can also be used into inter-array synchronization. Once post processing operations have completed, the array transitions into the *ArrayStoring* state. In the *ArrayStoring* state, post processed values are written from the array output registers into the tile local memory. Importantly, no data will be overwritten unless all consumers have signaled that the previous results have been copied to the consumer's tile memory. This provides back-pressure for tiles which are running slowly. Once all results have been written to tile memory, the tile signals consumers that new data is available and transitions back to the *ArrayIdle* or *ArrayFinished* state. *ArrayFinished* is a simulation specific state to track that this array has completed the last input in the simulation so the simulation can now complete.

2.1.2.1. Test Driver Operation

Although the test driver is a simplified version of the tile model, it does not use the aforementioned state trackers. Since the test driver does not need to worry about arrays it effectively implements just the *ProducerReady* state to read the output from the final tile or tiles in a data flow, and the *ArrayStoring* state to write new inputs for the system. These operations are triggered by signals from other tiles.

2.1.2.2. Inter-Tile Communication

In describing the tile operations several state transitions have the side effect of signaling other tiles that new data is ready, or that copies have completed so previous intermediate results can be overwritten. These inter-tile signals are implemented as MMIO writes to different tiles. In the configuration parameters each tile has an *mmioOffset* parameter which is used to compute an MMIO address for each tile based on the tile's starting address. This offset must be identical across all tiles in a system since each tile uses this offset to calculate the MMIO address of the producer and consumer tiles.

2.1.3. Tile Memory

Figure 2-1 shows a notional configuration where each tile contains tile memory. This is a valid configuration, and may be the most common in the literature; however, the tile model discussed in this report does not require the memory arrays containing the tile memory space to be physically located within a tile. Each tile is configured with a memory region specified in the global address space, and as discussed above the only assumption made by the tile model is that no other component in the system will modify the data within that address range. This means that the tile model can model systems where tile memory for multiple tiles can share the same physical memory arrays.

This implementation is partially driven by the design of the existing SST memory hierarchy element. This element does not currently support endpoints which are both memory controllers holding part of the the global address space and processors which read and write the global address space. To simulate a system where the tile memory is physically located within a tile, the topology of the network model can be modified. Specifically, a memory controller and tile can be connected to a router with high-bandwidth and low-latency links and that router is connected to the broader network model as the full tile node would be.

Table 2-3. Tile functions designed to be overridden for new tile types.

Function	Description
moveFromProducerAddr	Compute a source, destination for copying date to tile memory from a producer coordinate.
loadToArrayAddr	Compute a tile memory address from an array input coordinate.
storeFromArrayAddr	Compute a tile memory address from an array output coordinate.
producerToArray	Convert a producer coordinate to a list of array coordinates that take this producer as an input.
arrayToConsumers	Converts an array ID into a list of tiles consuming the results of this array.
consumerToArrays	Converts a consumer tile ID into a list of arrays which produced results for this consumer.
postProcessArray	Performs any digital post processing operations on the results of the array operations. Can implement synchronization between arrays if multi-array results are needed for post processing. Return value true indicates that post processing has been completed and the array can transition to the next state.

2.1.4. Implementing New Tile Types

As noted above, the primary function of the tile controller is as a memory request generator. To flexibly support a wide variety of different memory indexing operations, the tile component is designed to allow new tile operations to be implemented through classes derived from the base tile class. For instance, both a fully connected and a convolutional layer in a neural network, use the same basic tile logic in terms of the producer and array states described above; however, they use different addressing functions.

Seven functions in the base tile class are designed to be overridden by new tile types and are summarized in Table 2-3. When the table refers to a "coordinate," this means a resource (producer or array) ID and element index within that resource. For instance, a producer coordinate (1, 23) means that we are looking at the 23rd operand in the producer with index 1 (as specified in the *producerTiles* configuration input).

These functions can be subdivided into three different types, address generation functions, resource mapping functions, and digital emulation functions. The address generation functions—*moveFromProducerAddr*, *loadToArrayAddr*, and *storeFromArrayAddr*—are the most straight-forward. These functions compute addresses for data copies in the *ProducerReady* state, tile memory reads in the *ArrayLoading* state, and tile memory writes to tile memory in the *ArrayStoring* state. If these functions return an address of zero, the copy, load, or store for that specific operation is skipped. This can be used either to skip redundant operations due to data reuse, or for operation sizes that are smaller than the physical tile resources.

The resource mapping functions—*producerToArray*, *arrayToConsumers*, and *consumerToArrays*—are used to convert a coordinate or resource ID to another resource ID. This is needed to track when a specific resource has all of the inputs required to transition to the next state. For instance, *producerToArray* converts a producer coordinate to an array ID which is needed to determine when an array can transition from the *ArrayIdle* to *ArrayLoading* state. As discussed above each array currently waits for all inputs for an array operation to be copied to tile memory before it begins loading the array input registers. Therefore, *producerToArray* is used to indicate which array counters need to be incremented when an copy has completed. The other two resource mapping functions are used for inter-type synchronization. *arrayToConsumers* indicates which arrays must be signaled when an array has completed an operation, and *consumerToArrays* is used to determine which arrays results have been fully copied by the consumer tiles.

The final function, *postProcessArray*, implements digital operations which are performed on the outputs of array operations. For instance, *postProcessArray* can be used to implement digital reduction of results for computations where a single operation is insufficient, e.g., bit slice reductions in MVM or operations on matrices larger than a single array. *postProcessArray* can also be used to implement pooling and similar operations in neural networks. For tiles where the results of multiple different array operations are required for post processing, *postProcessArray* can also implement inter-array synchronization.

2.2. Array Subcomponent

As discussed in Chapter 1, analog arrays can implement a wide variety of different functions. The compute array subcomponent is designed to enable the flexible implementation of different array functions within a tile.

The array subcomponent implements all of the arrays within a single tile. Although each array can have a different configuration, for example MVM arrays can have different stored matrices, the size and type (array function) is assumed to be the same for all arrays in a tile. Array models can be as simple or as complicated as needed for the simulation. A simple model, for instance digital (ideal) emulation of an MVM, requires only two functions, a *compute* function, and *getArrayLatency*. The *compute* function implements the actual array operation—MVM in this example—using the array input register values as inputs and placing the results in the array output registers. *getArrayLatency* specifies the latency of an array operation and can either by a fixed latency or input dependent. Although, not implemented in any currently developed array models, the array subcomponent can require multiple compute calls per array. For instance an array with error detecting logic could check the results of the computation and do another compute operation, with an addition array operation latency, if an error is detected.

More complex array operations, for instance incorporating array non-idealities, can be implemented directly in the array subcomponent or by interfacing the subcomponent with external models. We have developed an array implantation that interfaces with CrossSim [9], an analog MVM simulator that includes a variety of device models and analog non-idealities.

3. CONCLUSION

Analog computing shows the potential for efficiency beyond what is currently achievable from digital systems. However, to realize that potential we need better system-level modeling tools to enable full-stack co-design. Existing approaches to system-level simulation for analog computing, where each research group develops a customized simulator for a proposed architecture creates a custom simulator wastes research effort and often results in simplifications which impede co-design. This report describes a new model for analog tile-based computing using the Structural Simulation Toolkit (SST) that is designed to be flexible enough to model most existing analog systems. By using SST, this tile-based model can take advantage of existing models for memory and on-chip networks, and simplify future development of heterogeneous systems integrating conventional digital processors alongside analog components. This model will enable future projects to explore important topics in analog computing including programmability and run-time reconfiguration which are essential to realize the potential of analog systems.

REFERENCES

[1] T. Patrick Xiao, Christopher H. Bennett, Ben Feinberg, Sapan Agarwal, and Matthew J. Marinella. Analog architectures for neural network acceleration based on non-volatile memory. *Applied Physics Reviews*, 7(3):031301, 2020.

[2] T. Patrick Xiao, Ben Feinberg, Christopher H. Bennett, Venkatraman Prabhakar, Prashant Saxena, Vineet Agrawal, Sapan Agarwal, and Matthew J. Marinella. On the accuracy of analog neural network inference accelerators, 2021.

[3] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: Accelerating graph processing using ReRAM. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 531–543, 2018.

[4] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek. Enabling scientific computing on memristive accelerators. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 367–382, 2018.

[5] Ben Feinberg, Ryan Wong, T. Patrick Xiao, Christopher H. Bennett, Jacob N. Rohan, Erik G. Boman, Matthew J. Marinella, Sapan Agarwal, and Engin Ipek. An analog preconditioner for solving linear systems. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, March 2021.

[6] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. ASPLOS ’19, page 715–731, 2019.

[7] T. Patrick Xiao, Ben Feinberg, Christopher H. Bennett, Vineet Agrawal, Prashant Saxena, Venkatraman Prabhakar, Krishnaswamy Ramkumar, Harsha Medu, Vijay Raghavan, Ramesh Chettuvetty, Sapan Agarwal, and Matthew J. Marinella. An accurate, error-tolerant, and energy-efficient neural network inference engine based on sonos analog memory. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(4):1480–1493, 2022.

[8] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, Mar. 2011.

[9] T. Patrick Xiao, Christopher H. Bennett, Ben Feinberg, Matthew J. Marinella, and Sapan Agarwal. Crosssim: accuracy simulation of analog in-memory computing.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	Ron Oldfield	01441	1327
1	Clay Hughes	01422	1318
1	Suma G. Cardwell	01421	1327