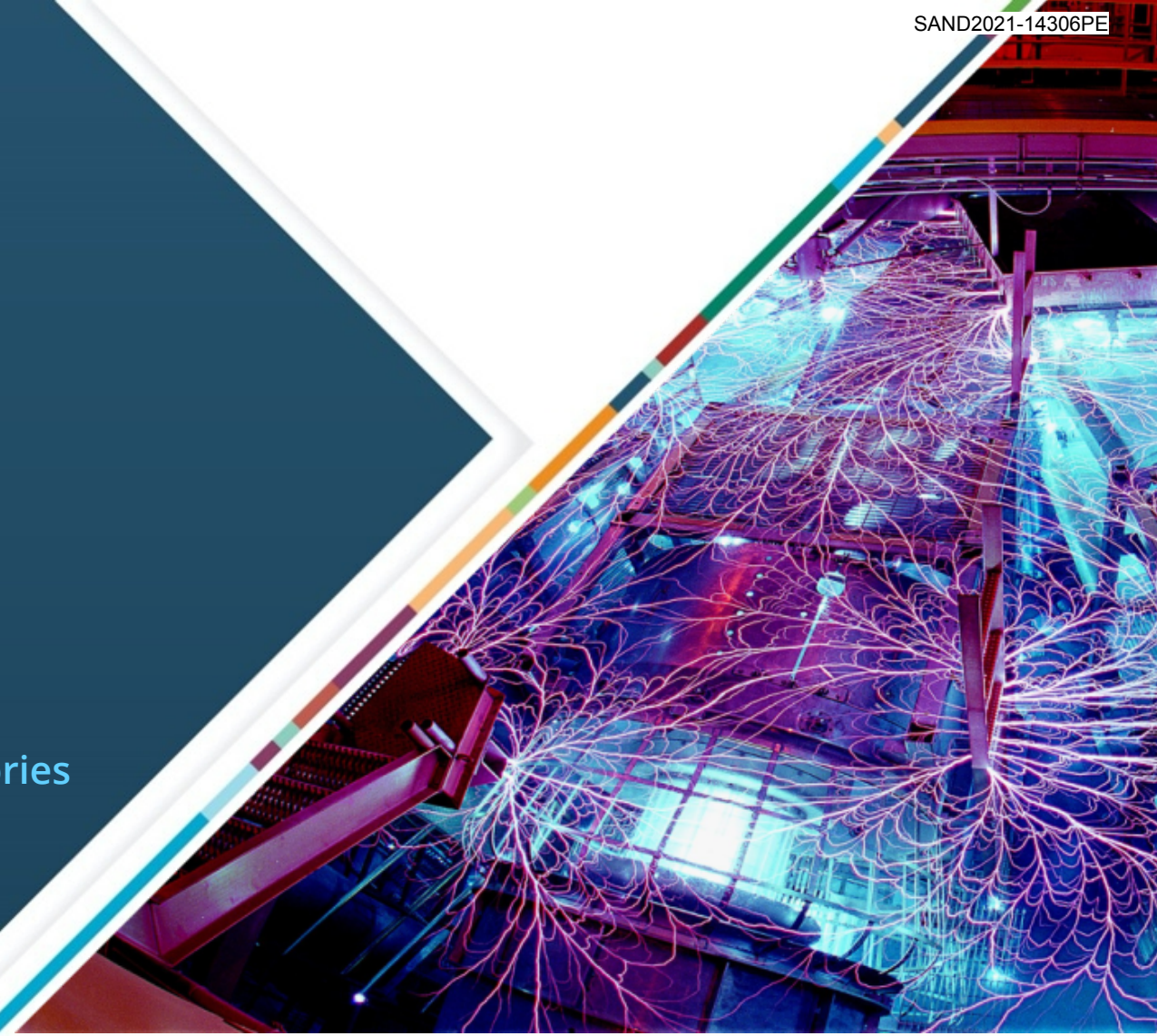
SAND2021-14306PE

Sandia National Laboratories

Exceptional service in the national interest



ARIAA Update -- SST

PM/PI Meeting AI/Codesign

Presented by Clay Hughes, Sandia National Laboratories

ARIAA Teams at SNL, PNNL, and GT
Francisco Munoz-Martinez, Universidad de Murcia

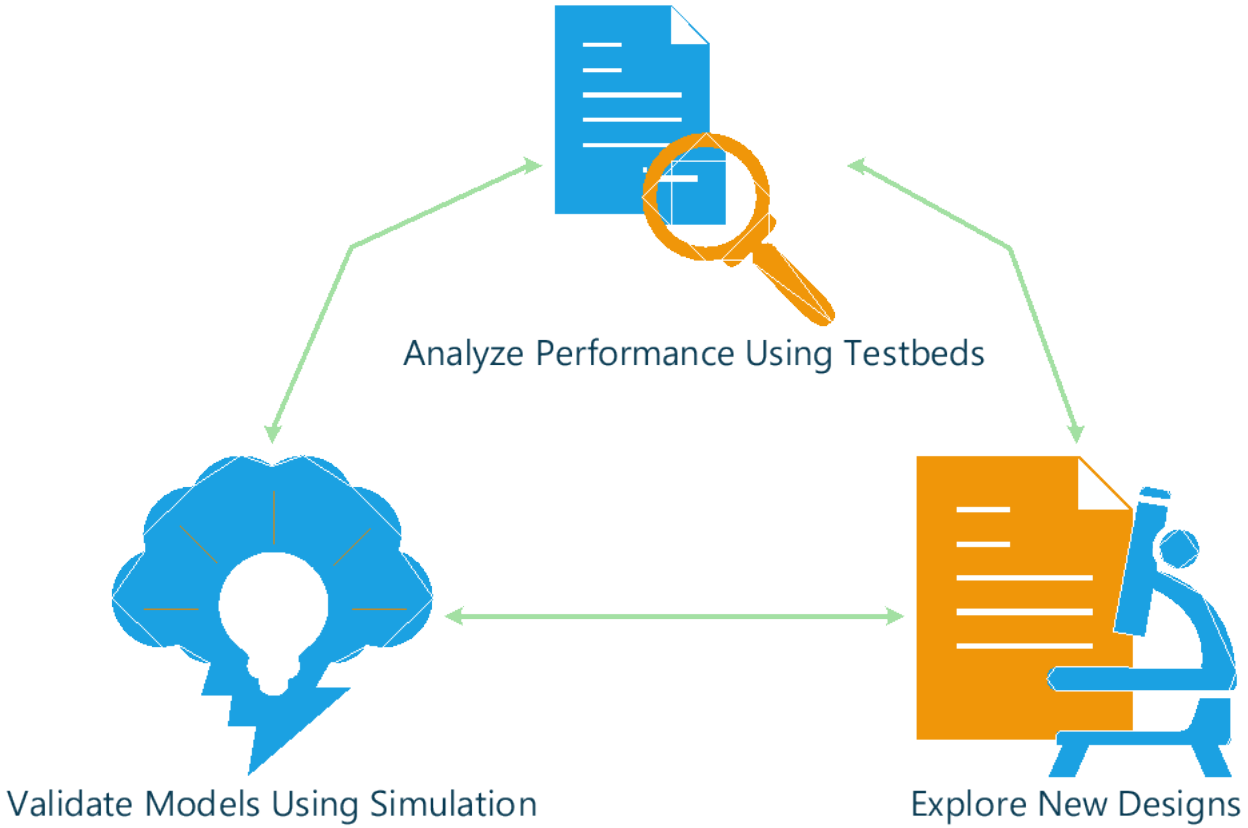
SAND20xx

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

Hardware and Simulation Thrust

This thrust will use simulation infrastructure and early hardware prototypes to design, validate, and evaluate hardware designs

SST and/or analytical modeling tools will be leveraged to evaluate and adapt the dataflow and benchmark algorithms



The Structural Simulation Toolkit (SST)

Goals

- Create a standard architectural *simulation framework* for HPC*
- Ability to evaluate future systems on DOE/DOD workloads
- *Use supercomputers to design supercomputers*

Technical Approach

- Parallel Discrete Event core
  - Conservative optimization over MPI/Threads
- Interoperability
  - Node and system-scale models
- Multi-scale
  - Detailed and simple models that interoperate
- Open
  - Open Core, non-viral, modular

Status

- Parallel framework (*Core*)
- Integrated component libraries (*Elements*)
- Current Release (11.1.0)
  - https://sst-simulator.org
  - https://github/sstsimulator

Consortium

- "Best of Breed" simulation suite
- Combine Lab, Academic & Industry

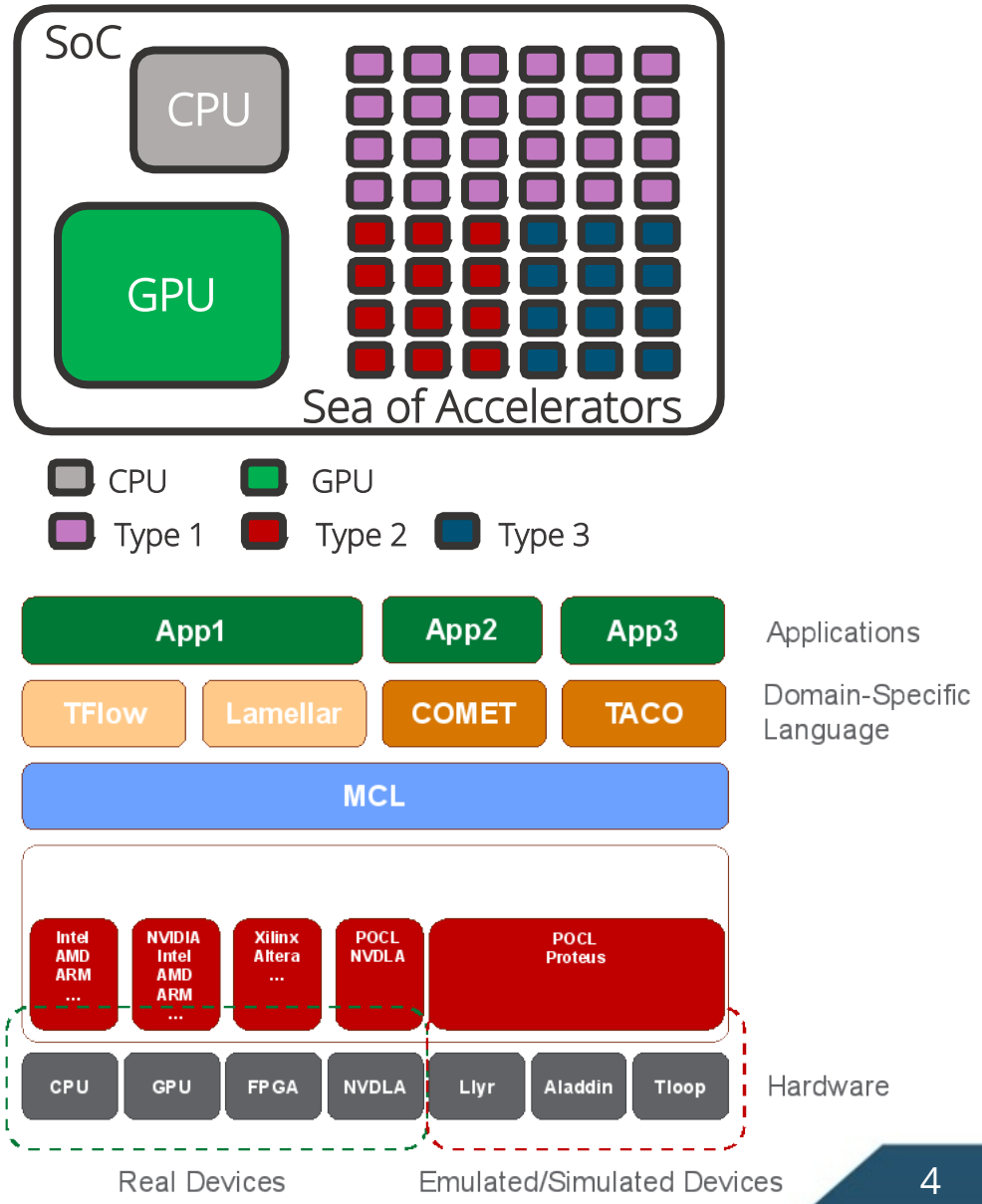
SST + MCL (PNNL) - HW/SW Co-design

Quantify performance in a full-workload on a computing system with heterogeneous devices

Rapidly integrate specialized accelerators with yet-to-be-available HW

Reduce design space for detailed architectural evaluations, e.g., using SST

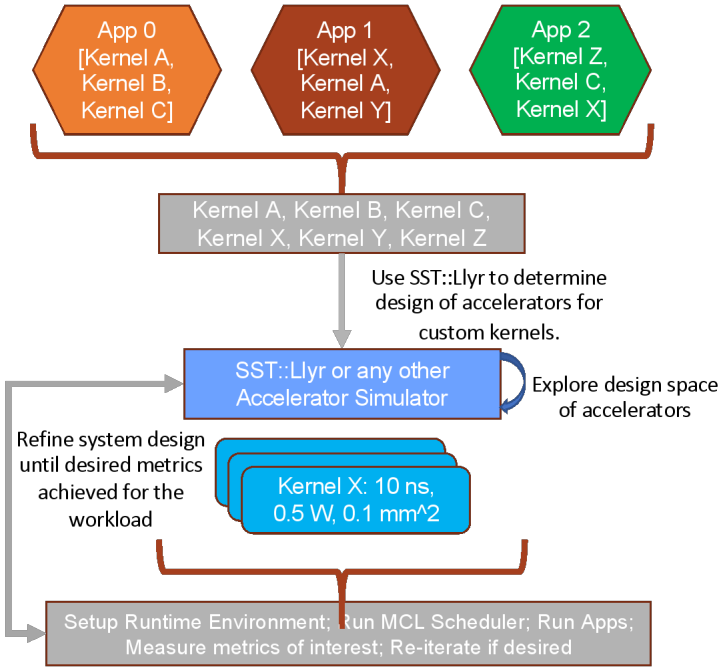
Enables next-generation compute node design for HPC/AI/Data Analytics

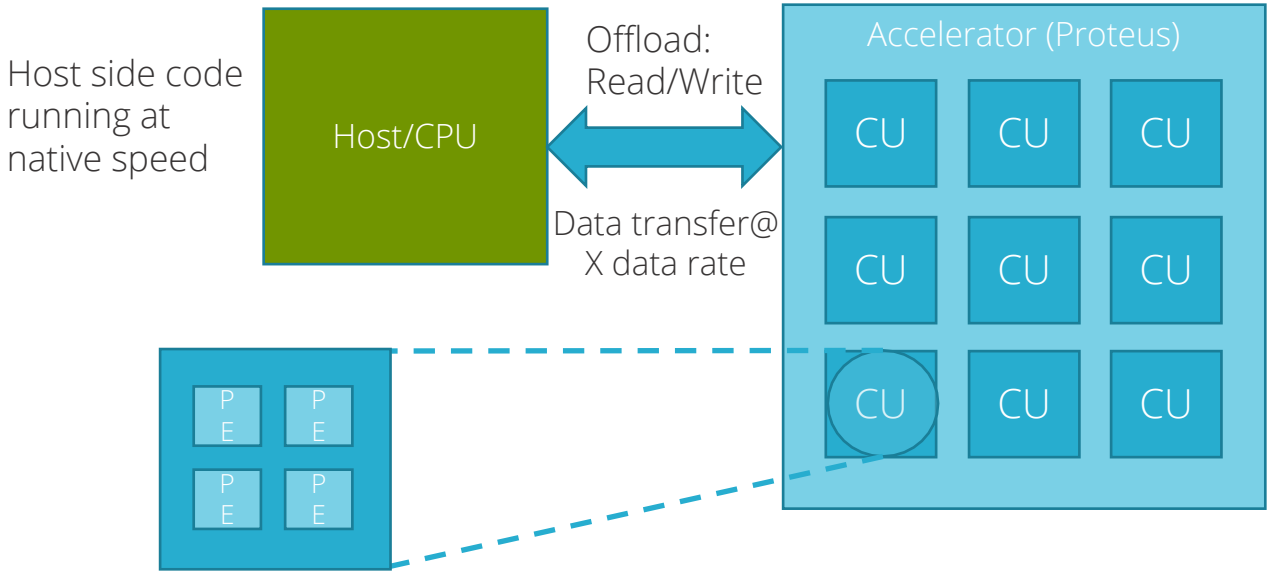


SST + MCL (PNNL) - The Proteus Device

Proteus is a data-parallel compute device with a pool of parallel threads

Each compute unit (CU) simulates the execution of a work group





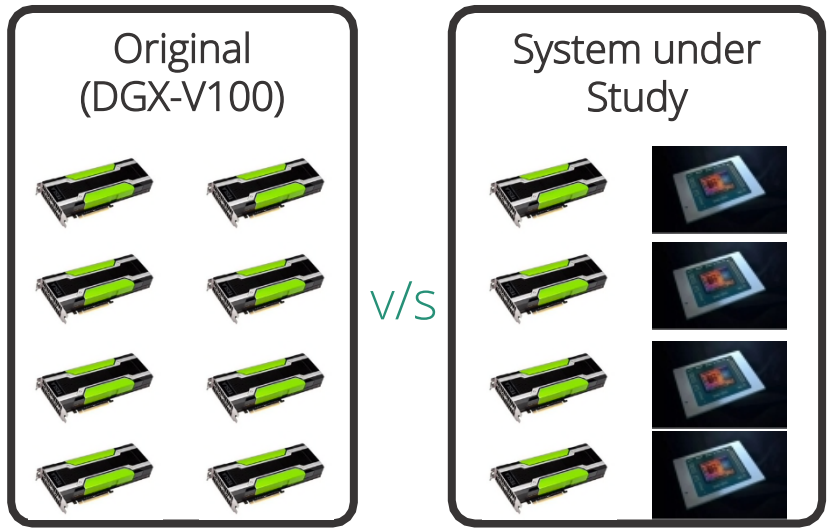

```
Platform #0: Portable Computing Language
 +-- Device #0: basic-Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
 +-- Device #1: PROTEUS Dense Matrix Multiplication Kernel
 `-- Device #2: PROTEUS Dense Matrix Multiplication Kernel
Platform #1: NVIDIA CUDA
 +-- Device #0: NVIDIA Tesla V100-SXM2-16GB
 +-- Device #1: NVIDIA Tesla V100-SXM2-16GB
 +-- Device #2: NVIDIA Tesla V100-SXM2-16GB
 `-- Device #3: NVIDIA Tesla V100-SXM2-16GB
```

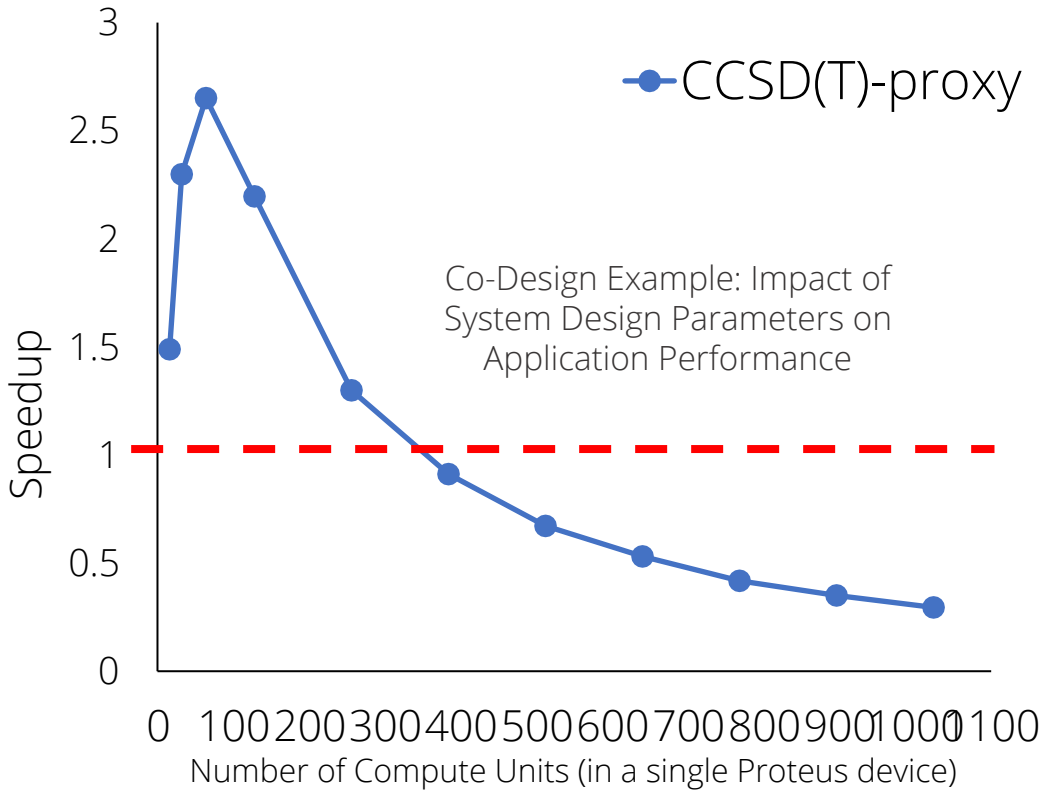
Kernel Launch Overhead + Kernel Execution Latency

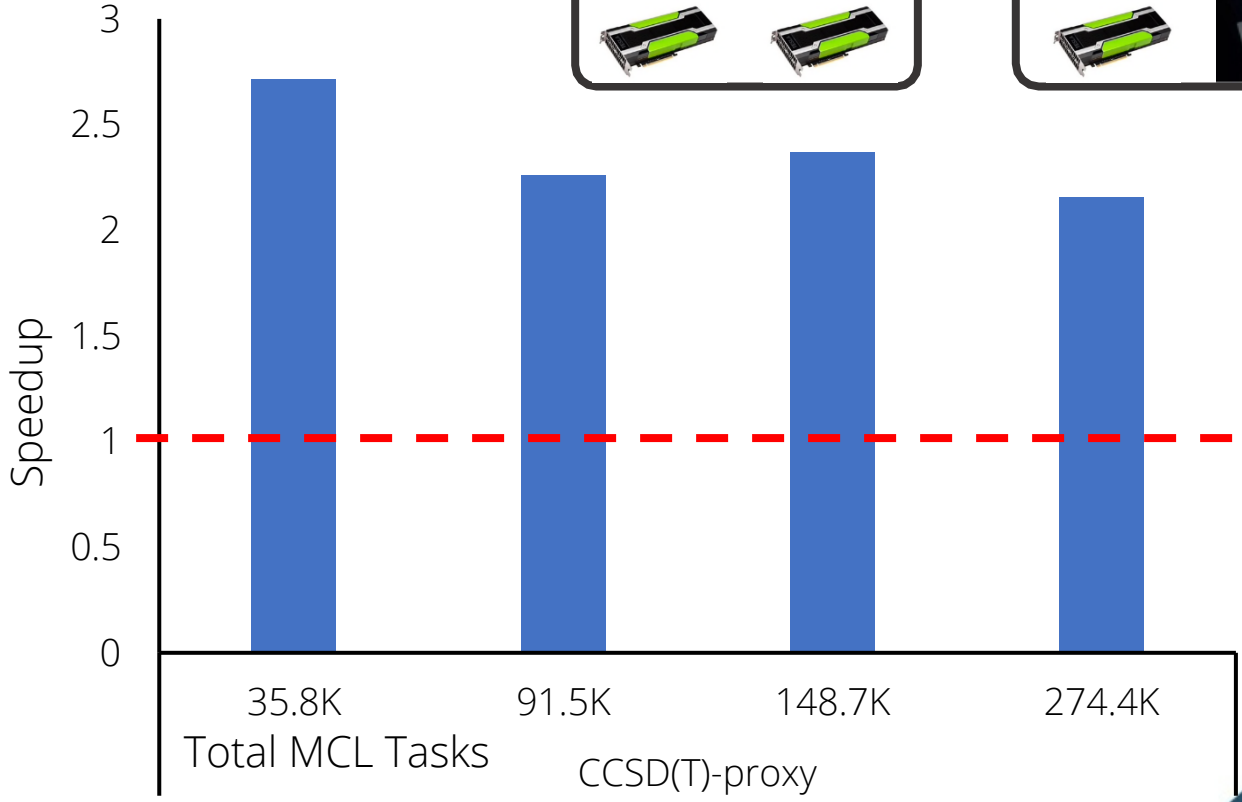
SST + MCL (PNNL) - Application-first Co-design

The couple cluster method CCSD(T) is a tensor contraction operation inside NWChem

CCSD(T)'s tensor contraction decomposed as Transpose-Transpose-GEMM-Transpose





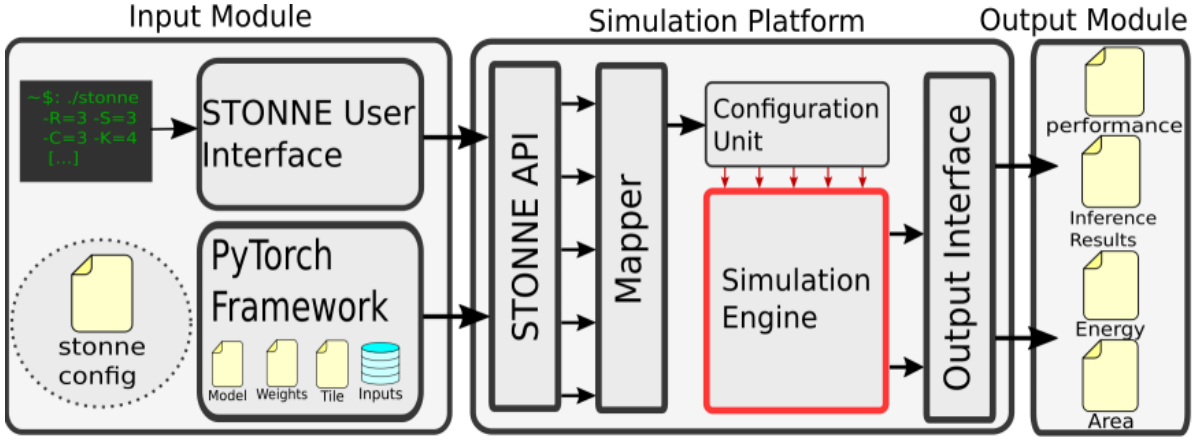


SST + STONNE (GT)

STONNE (A Simulation Tool for Neural Network Engines)

- Cycle-level microarchitectural simulator for DNN inference accelerators
- Written in C++
- Open-Sourced under the MIT License: https://github.com/stonne-simulator/stonne

STONNE is composed of 3 main building blocks:



SST + STONNE (GT)

All STONNE operations have been integrated without the memory system:

- CONV (convolution)
- GEMM (dense MM)
- bitmapSpMSpM (SparseSparse MM with bitmap)
- csrSpMM (SparseDense MM with CSR)

Can run instances of MAERI but they *still assume unlimited SRAM buffer* when accessing memory

Current integration allows the user to select the type of operation

```
import sst

# Define the simulation components
comp_stonne = sst.Component("stonne1", "sstStonne.MAERI")
comp_stonne.addParams({
    "hardware_configuration" : "maeri_128mses_128_bw.cfg",
    "kernelOperation" : "GEMM",
    "GEMM_K" : 20,
    "GEMM_N" : 3,
    "GEMM_M" : 3,
    "GEMM_T_K" : 5,
    "GEMM_T_M" : 1,
    "GEMM_T_N" : 1,
    "mem_matrix_a_init" : "gemm_file_matrixA_3_3_20.in",
    "mem_matrix_b_init" : "gemm_file_matrixB_3_3_20.in",
    "mem_matrix_c_init" : "result.out"

})
```

SST + STONNE (GT) -- Next Steps

Modify the memory controllers in STONNE to connect them to the memHierarchy component in SST

Integrate STONNE with a host CPU via MMIO interface

SST-STONNE integration will evolve as STONNE evolves:

- New sparse dataflows and memory hierarchy implementations are coming up
- New operations such as MaxPool2D

General Dataflow Accelerators

Compilers build internal representations of applications that represent the behavior as a series of graphs -- abstracting the control and data flow

Traditional processors execute instruction sequentially, destroying an application's inherent ILP

- Superscalar OoO processors go to great lengths to reconstruct the ILP
  - Multiple queues and complex logic allows instructions to issue when operands are available rather than in program order
  - Results are placed in additional queues and made visible to the system in program order

Dataflow architectures are able to execute these graphs directly, without the need to flatten the graph and artificially recover the parallelism

- Dedicated or shared PEs
- Statically or dynamically scheduled

Dataflow Graph -- *multiply_test()*

C++ Function

```
void  multiply_test(int* a, int *b, int* const c) {
   int f = 3 * (*a);
   int g = 2 * (*b);
   *c = f + g;
}
```

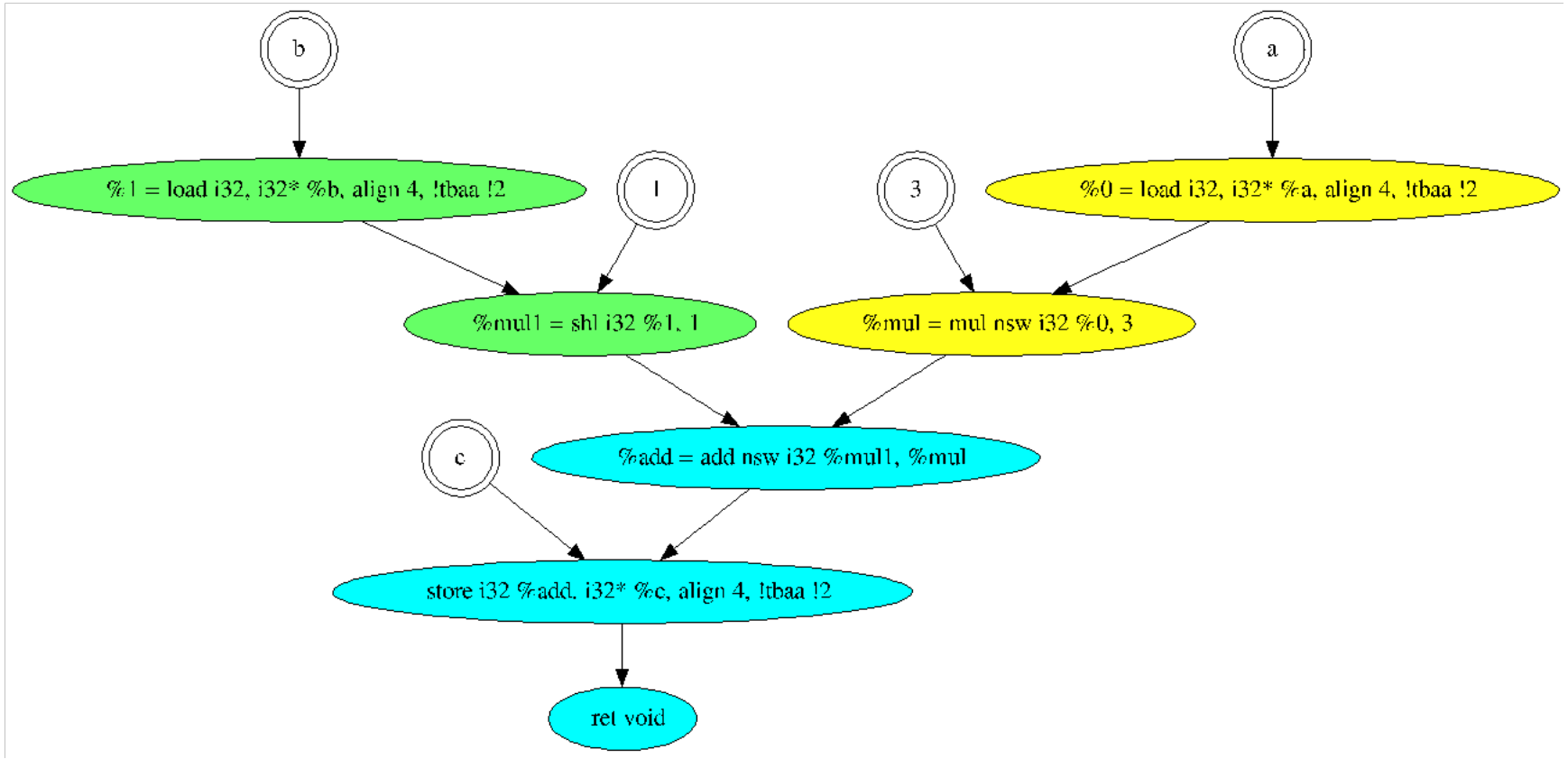
LLVM IR

```
; Function Attrs: norecurse nounwind uwtable
define void @multiply_test(i32* %a, i32* %b, i32* %c) local_unnamed_addr #0 {
entry:
 %0 = load i32, i32* %a, align 4, !tbaa !2
 %mul = mul nsw i32 %0, 3
 %1 = load i32, i32* %b, align 4, !tbaa !2
 %mul1 = shl i32 %1, 1
  %add = add nsw i32 %mul1, %mul
  store i32 %add, i32* %c, align 4, !tbaa !2
  ret void
}
```

Dataflow Graph -- *multiply_test()*

C++ Function

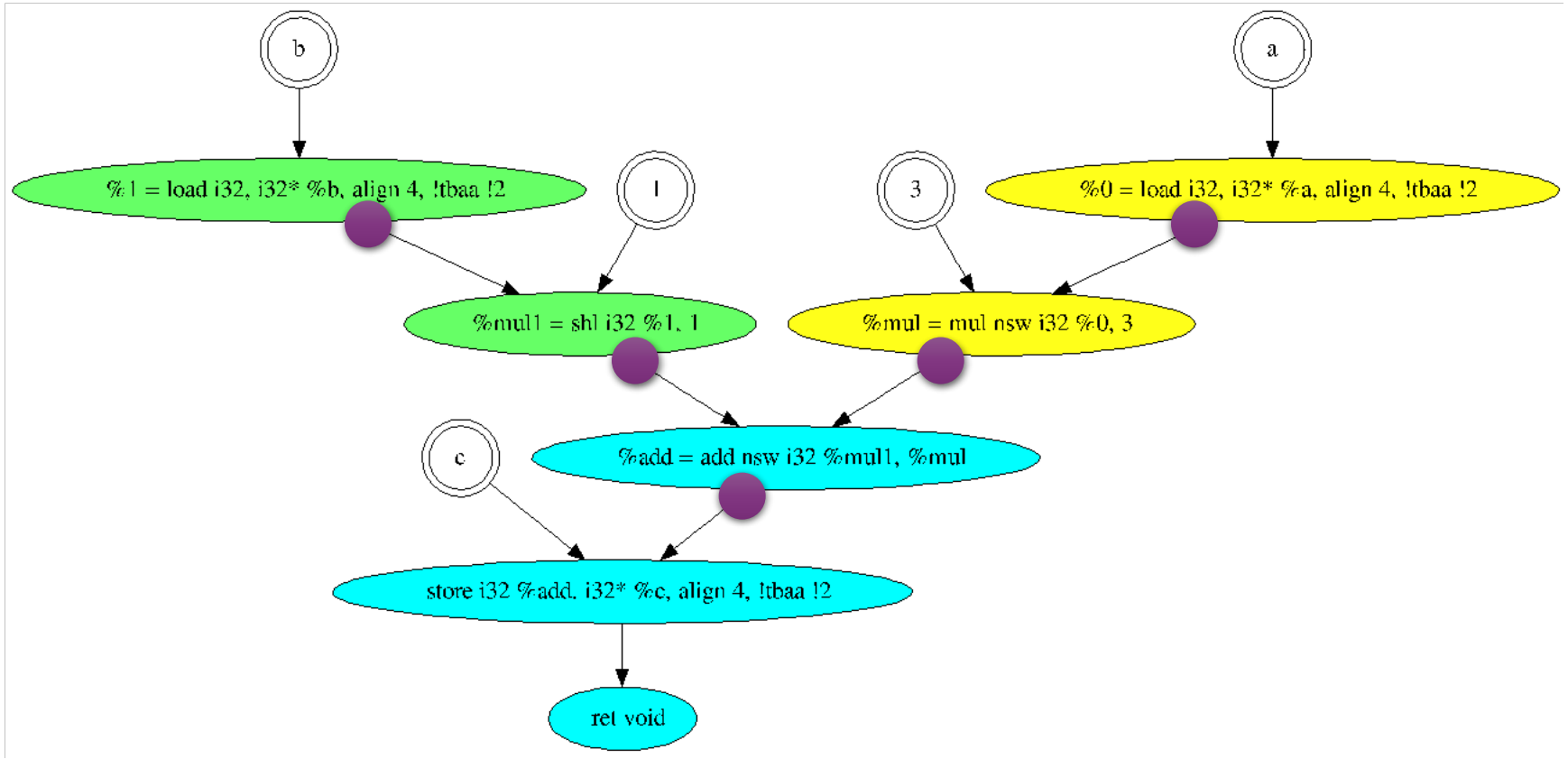
```
void  multiply_test(int* a, int *b, int* const c) {
  int f = 3 * (*a);
  int g = 2 * (*b);
  *c = f + g;
}
```



Dataflow Graph -- *multiply_test()*

C++ Function

```
void  multiply_test(int* a, int *b, int* const c) {
  int f = 3 * (*a);
  int g = 2 * (*b);
  *c = f + g;
}
```



Simulating Dataflow Architectures

There are examples of each of the different types of accelerators in the literature

- Dedicated/Static
  - Softbrain
- Dedicated/Dynamic
  - Plasticine, SPU, MAERI
- Shared/Static
  - CGRA
- Shared/Dynamic
  - TRIPS, SGMF

Developing dataflow component for Structural Simulation Toolkit (SST) for dedicated/dynamic designs

- Flexible interface to add custom PEs
- Arbitrary connectivity (can even be used to model ReRAM-like crossbars)
- Mappers that are dynamically loaded, allowing them to be swapped at runtime
- Leverage SST component interface to enable scalable simulations

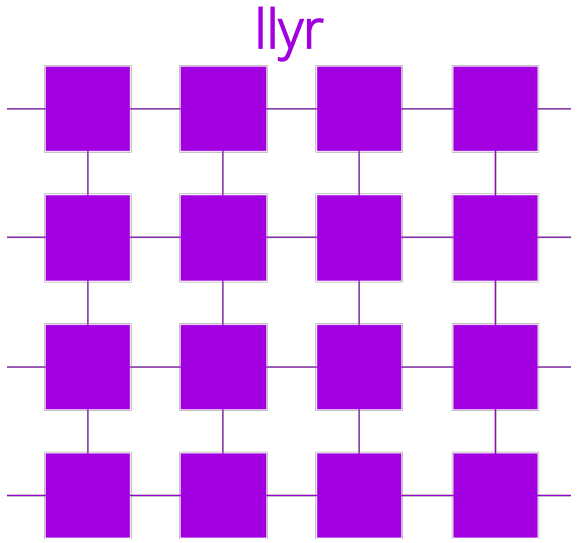
SST Dataflow Component - llyr

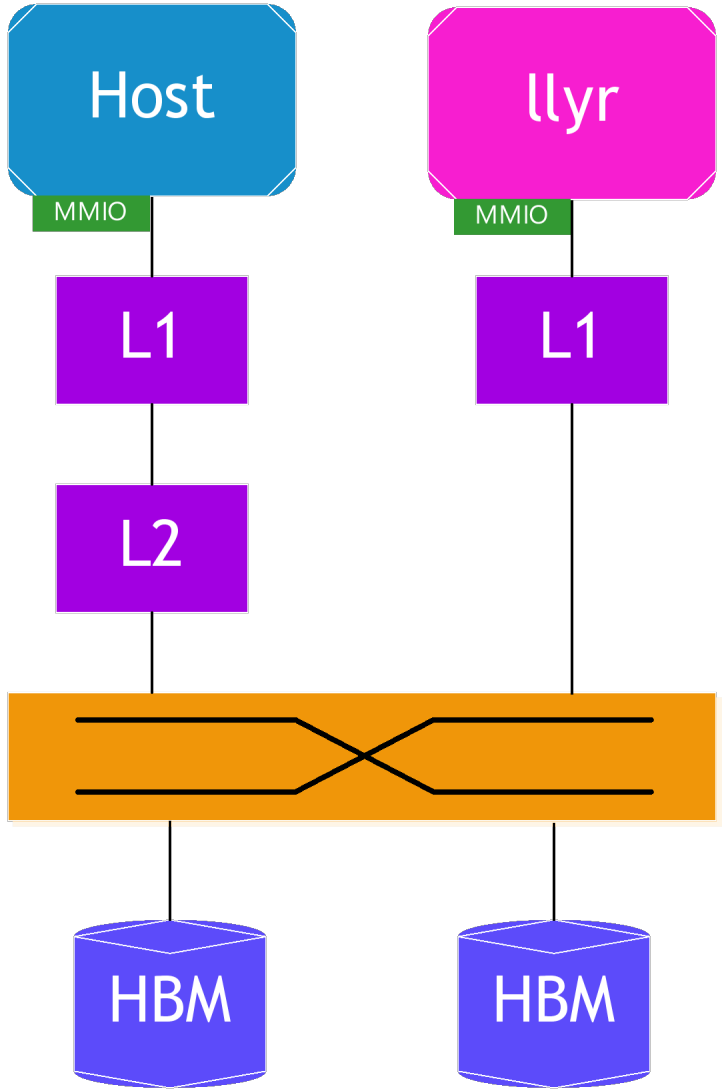
Multiple instances that allow different configurations for each

- Arbitrary memory hierarchy
- Some limitations on node configurations
  - Unable to launch from device

PEs have compute, input buffers, and output buffers

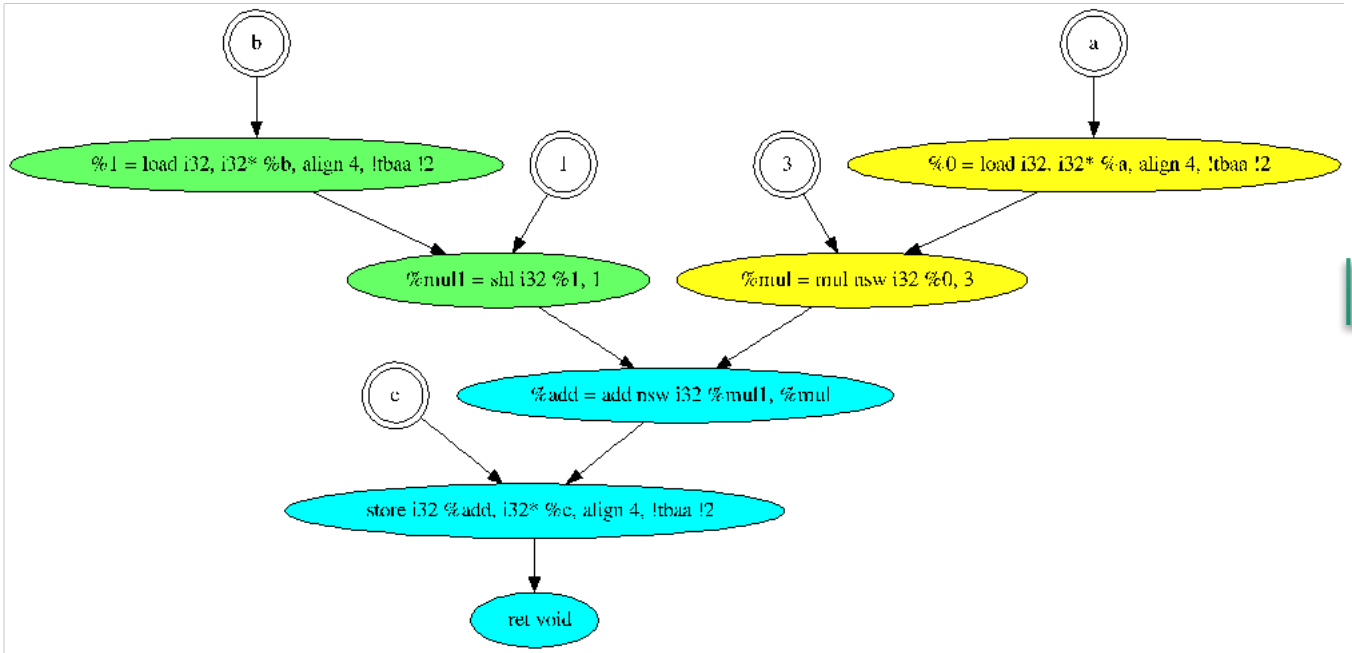
- Number of buffers bounded by connectivity
- Buffer depth is configurable but is uniform for all PEs



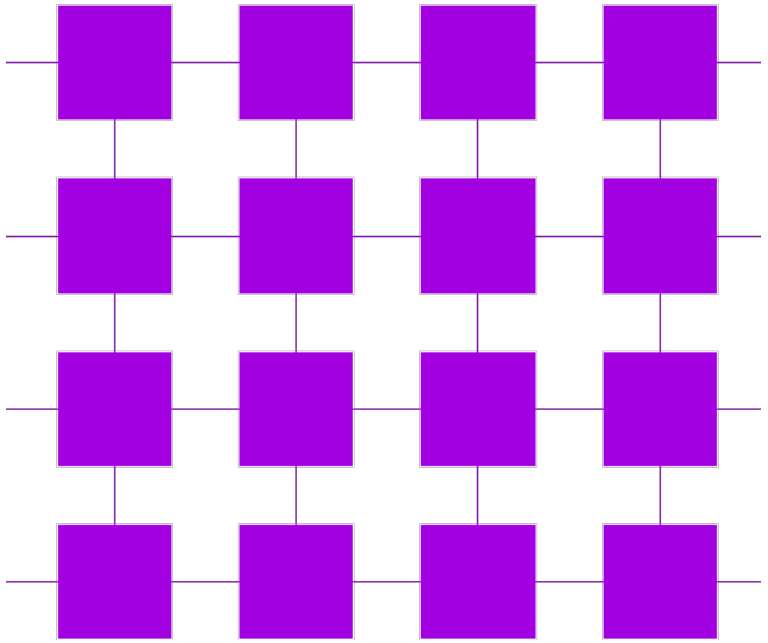


Mapping Applications to Dataflow Architectures

Mapper modules will handle the embedding of the application graph



Mapper



ASC/AML → Standalone VF3 implementation from the University of Salerno

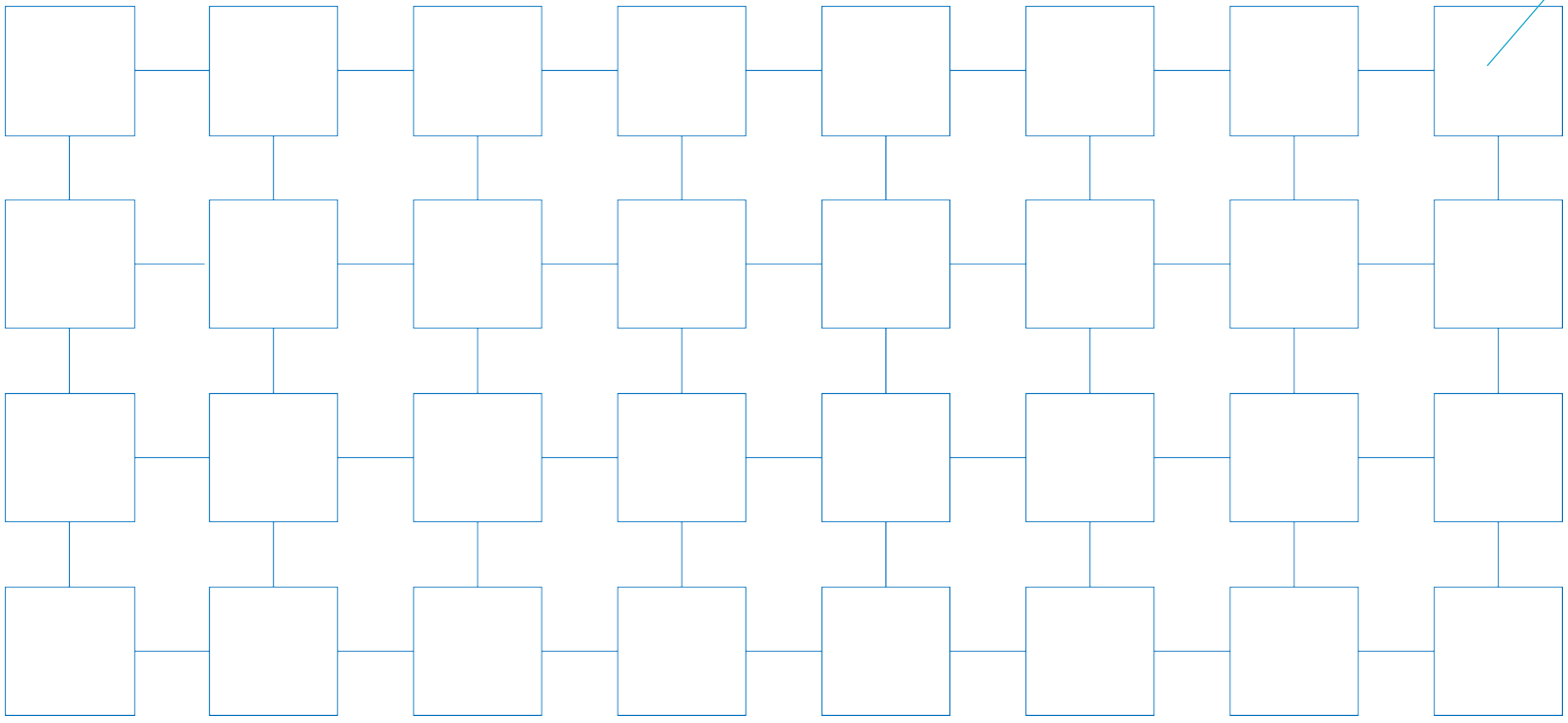
- Tested on multiple graphs (up to 512 nodes)
- Current problem is that constraints can result in no valid mapping
  - Need a way to bypass a node

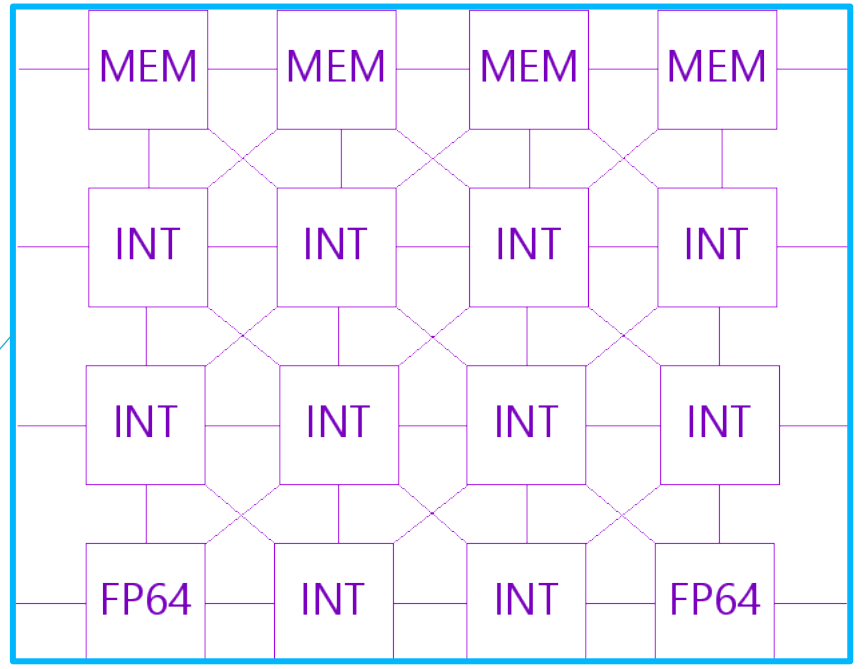
ARIAA → Exploring MIP-based solver

Notional ARIAA General Purpose Accelerator

~1024PEs
- Clusters of densely-connected PEs in a larger grid

Study connectivity and composition
- Based on needs of algorithms
- Target node (7nm? 5nm?)





Summary

PNNL and GT leveraging capabilities of SST

- Automatic data partitioning
- New SST dataflow components to support

SST dataflow component, llyr, is being tested as a standalone compute unit

- Successfully runs torus and grid hardware graphs with GEMM application graph
- Successfully runs RRAM crossbar model with GEMM

No automatic mapping of application to hardware yet

- VF3 shown to work using sample application and hardware graphs -- currently working on integration with SST as a ‘mapper’ subcomponent

Thanks! Questions?

Exceptional Service In The National Interest

Sandia National Laboratories

Spatial/Dataflow Accelerators

Compilers build internal representations of applications that represent the behavior as a series of graphs -- abstracting the control and data flow

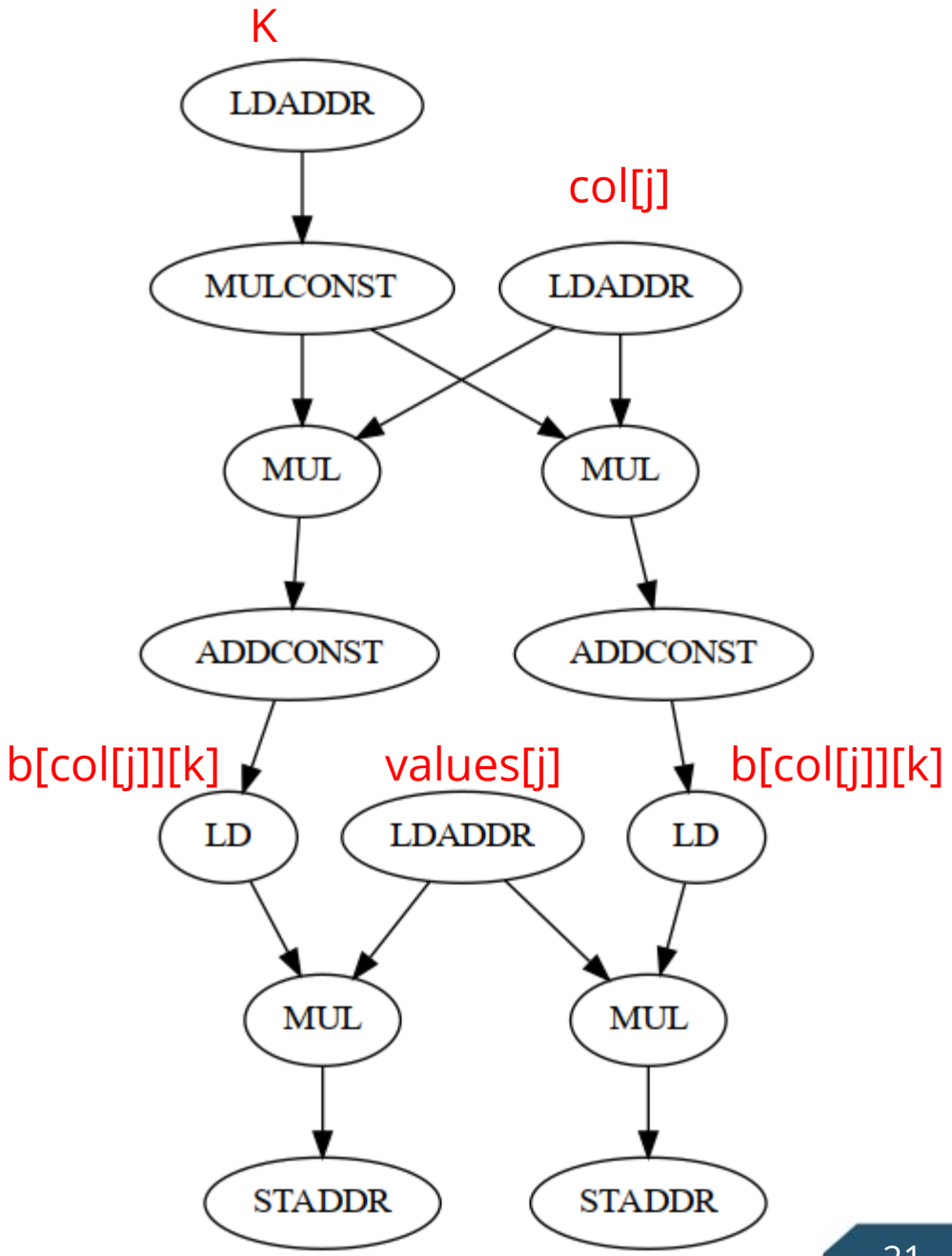
Traditional processors execute instruction sequentially, destroying an application's inherent instruction-level parallelism (ILP)

- Superscalar OoO processors go to great lengths to reconstruct the ILP
  - Multiple queues and complex logic allow instructions to issue when operands are available rather than in program order
  - Results are placed in additional queues and made visible to the system in program order even if they are completed out-of-order

Dataflow architectures are able to execute these graphs directly, without the need to flatten the graph and artificially recover the parallelism

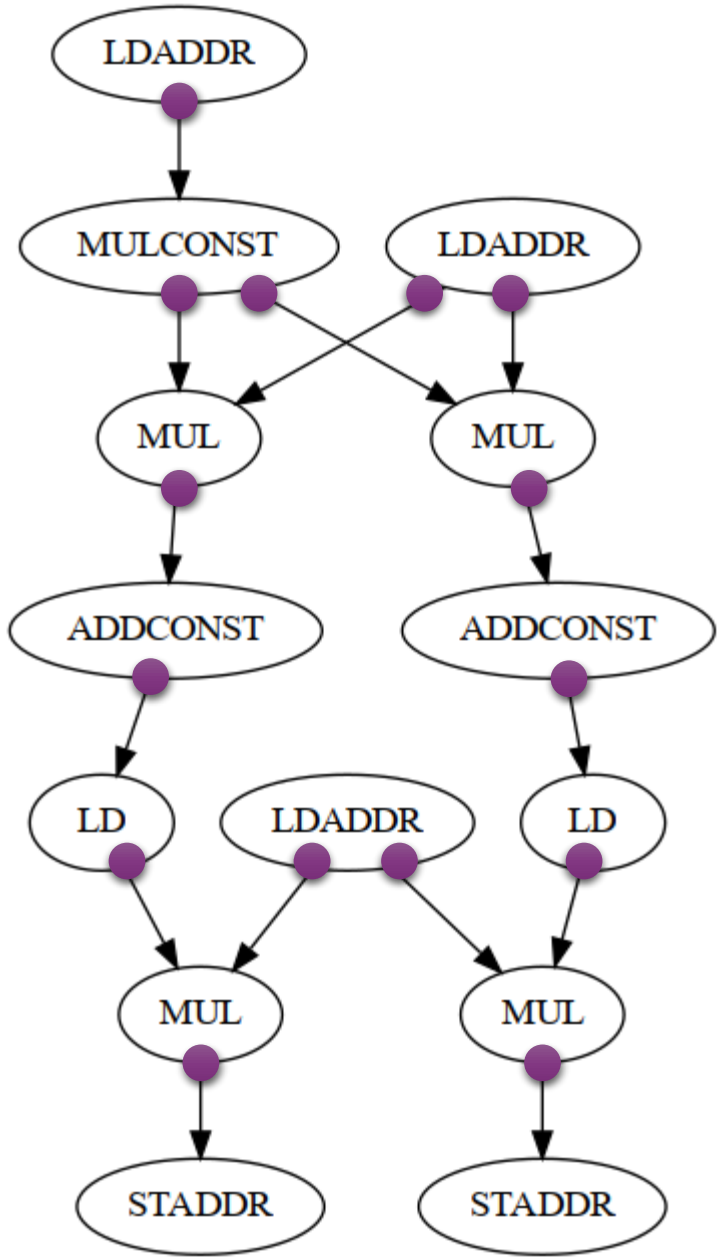
SpMM - CSR ($1 \times 3 \cdot 3 \times 2$)

```
for( uint32_t i = 0; i < M; ++i )
    for( uint32_t j = row_ptr[i]; j < row_ptr[i+1]; ++j )
        for( uint32_t k = 0; k < K; ++k )
            result[i][k] = result[i][k] + values[j] * b[column[j]][k];
```



SpMM - CSR ($1 \times 3 \cdot 3 \times 2$)

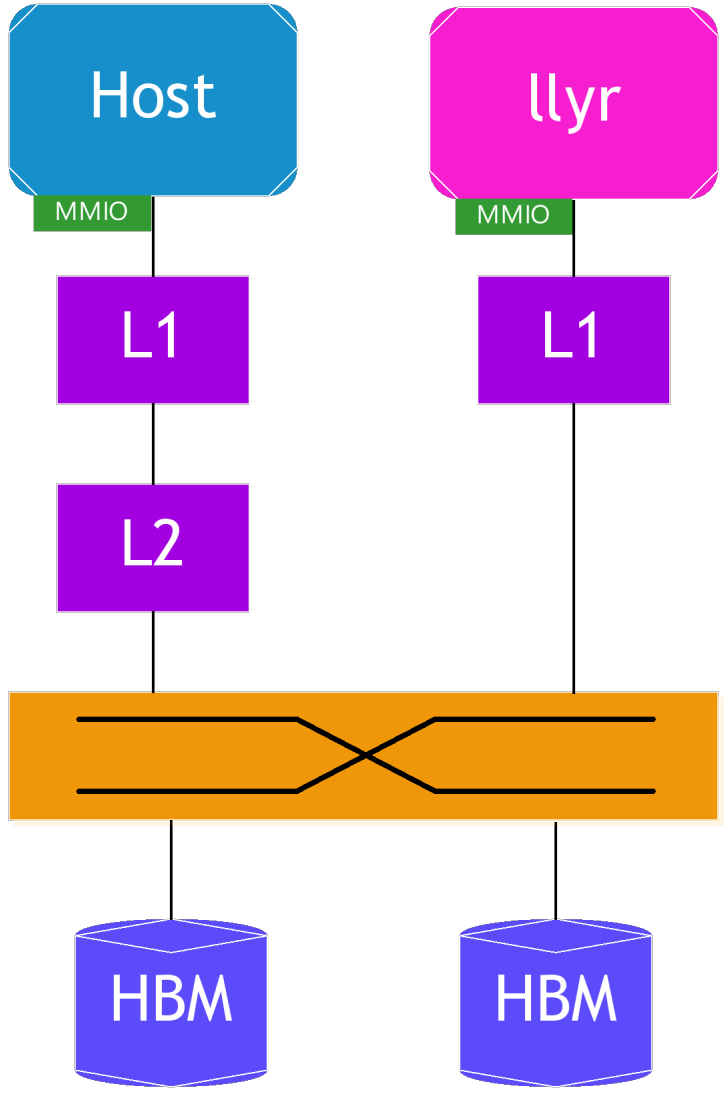
```
for( uint32_t i = 0; i < M; ++i )
    for( uint32_t j = row_ptr[i]; j < row_ptr[i+1]; ++j )
      for( uint32_t k = 0; k < K; ++k )
          result[i][k] = result[i][k] + values[j] * b[column[j]][k];
```



SST llyr Component

Instantiated like any other SST component
- Allows for…
  - Multiple instances, possibly with different configurations
  - Arbitrary memory hierarchy
  - Some limitations on node configurations
    - Unable to launch from device

Utilizes new MMIO interface
- Address range set aside for control
  - Doorbell and kernel location
- Can send and receive data in global address space
  - Eventually will be capable of self-hosted memory with TLB



*Currently being tested as a standalone compute device

SST llyr Component

PEs have a compute unit, input buffers, and output buffers

- Number of buffers bounded by connectivity
- Buffer depth is configurable but is uniform for all PEs

Current PE list

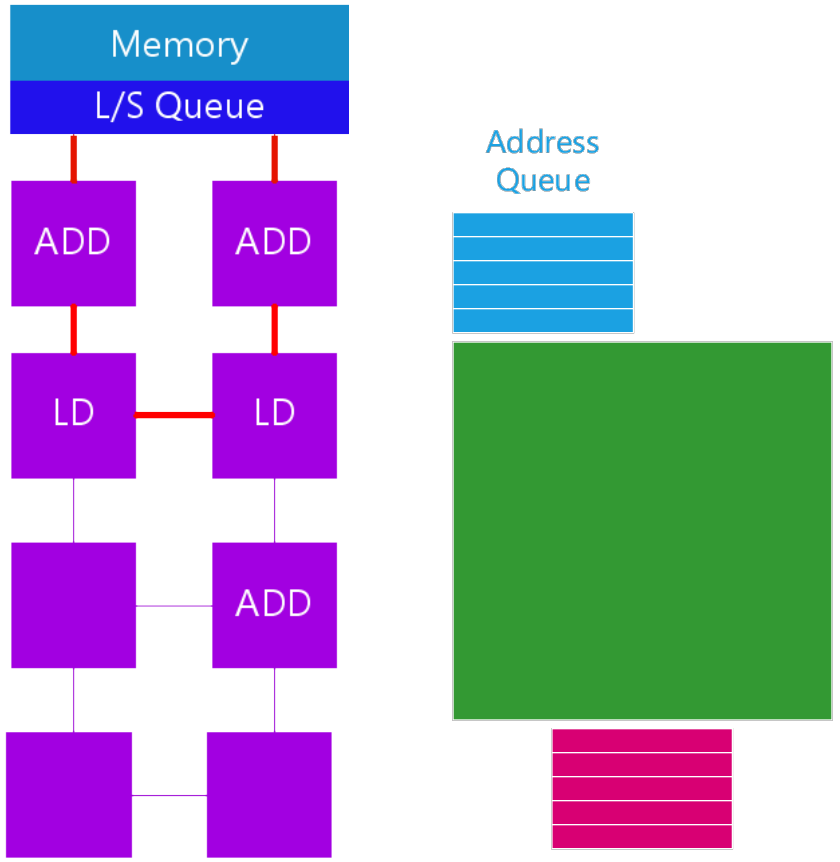
- LD, ST, LD_ST
- SLL, SLR, ROL, ROR
- ADD, SUB, MUL, DIV
- FPADD, FPSUB, FPMUL, FPDIV, FPMATMUL
- BUFFER
- ANY, ANYMEM, ANYLOGIC, ANYINT, ANYARITH, ANYFP

SST llyr Component

Address calculation performed by preceding PEs

Memory operations forced to return in program order via common L/S queue

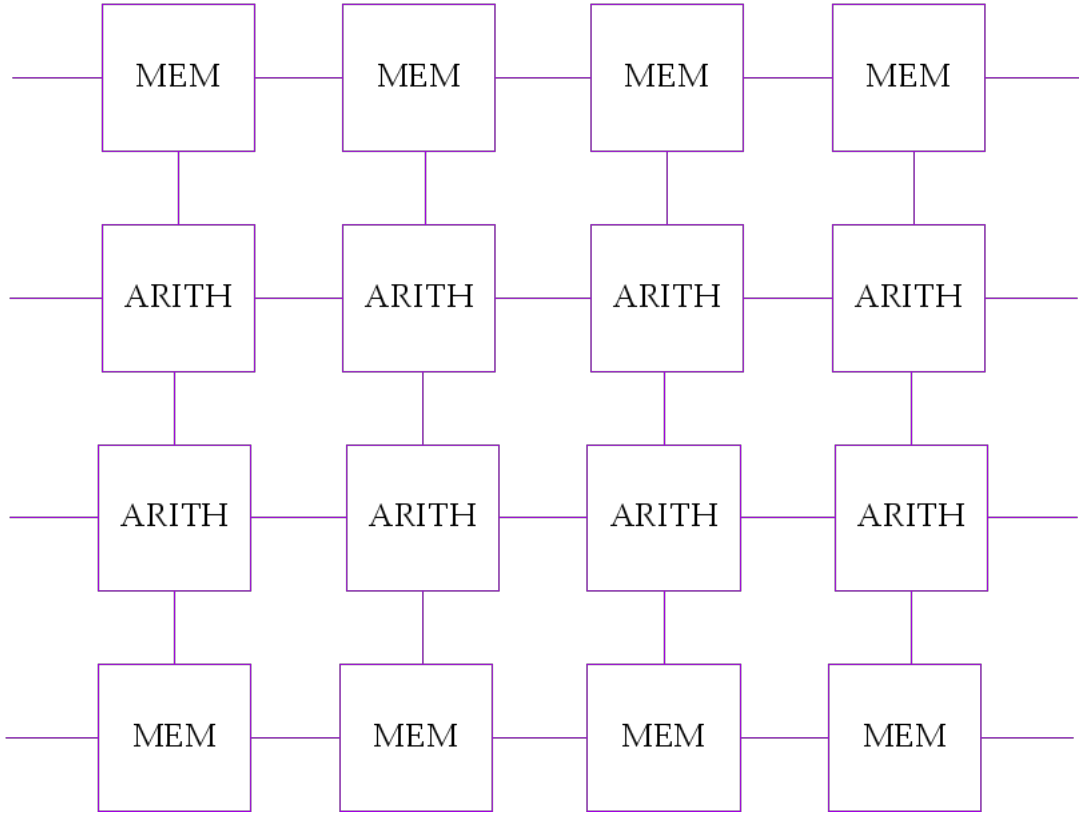
Reponses are forwarded directly to output queue of memory PE



SST llyr Component

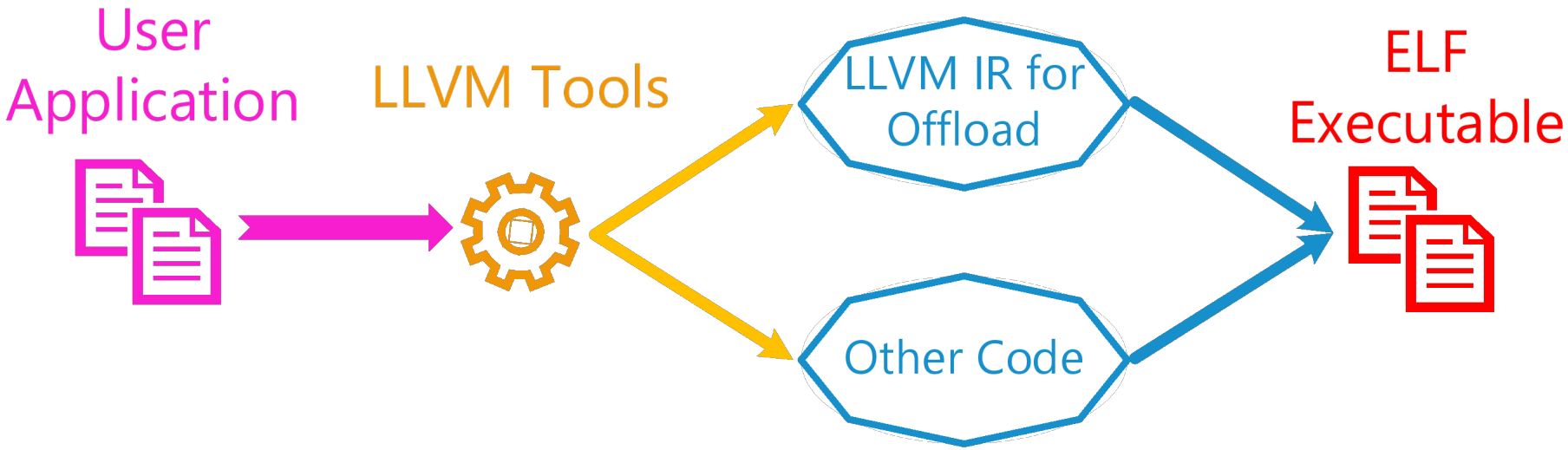
Constructed using a hardware description file

- Describes connectivity between PEs
- Describes allowable operations per-PE




```
0 [pe_type=ANYMEM]          0 -- 1
1 [pe_type=ANYMEM]          0 -- 4
2 [pe_type=ANYMEM]          1 -- 0
3 [pe_type=ANYMEM]          1 -- 2
4 [pe_type=ANYARITH]        1 -- 5
5 [pe_type=ANYARITH]        2 -- 1
6 [pe_type=ANYARITH]        2 -- 3
7 [pe_type=ANYARITH]        2 -- 6
8 [pe_type=ANYARITH]        3 -- 2
9 [pe_type=ANYARITH]        3 -- 7
10 [pe_type=ANYARITH]       4 -- 1
11 [pe_type=ANYARITH]       4 -- 5
12 [pe_type=ANYMEM]         4 -- 8
…                           …
```

LLVM and llyr Parsing



Work in progress…

Offload targets will be marked in the user application – currently studying how

- offload_myFunction()
- __attribute__ ((offload (device, 0))) myFunction()
- #pragma secret offload directive device(0)
- myFunction()

These functions/loops will be compiled with the user code but the LLVM IR will be embedded with them in the final executable

SST llyr Component Sample Configuration

```
llyr = sst.Component("dataflow0", "llyr.llyr")
llyr.addParams({
        "verbose": "1",
        "clock"  : "1GHz",
        "config" : "maeri_layout.cfg",
        "fp_lat" : "4",
        "int_lat": "1",
        "div_lat": "3",
        "mul_lat": "2",
        )}
```

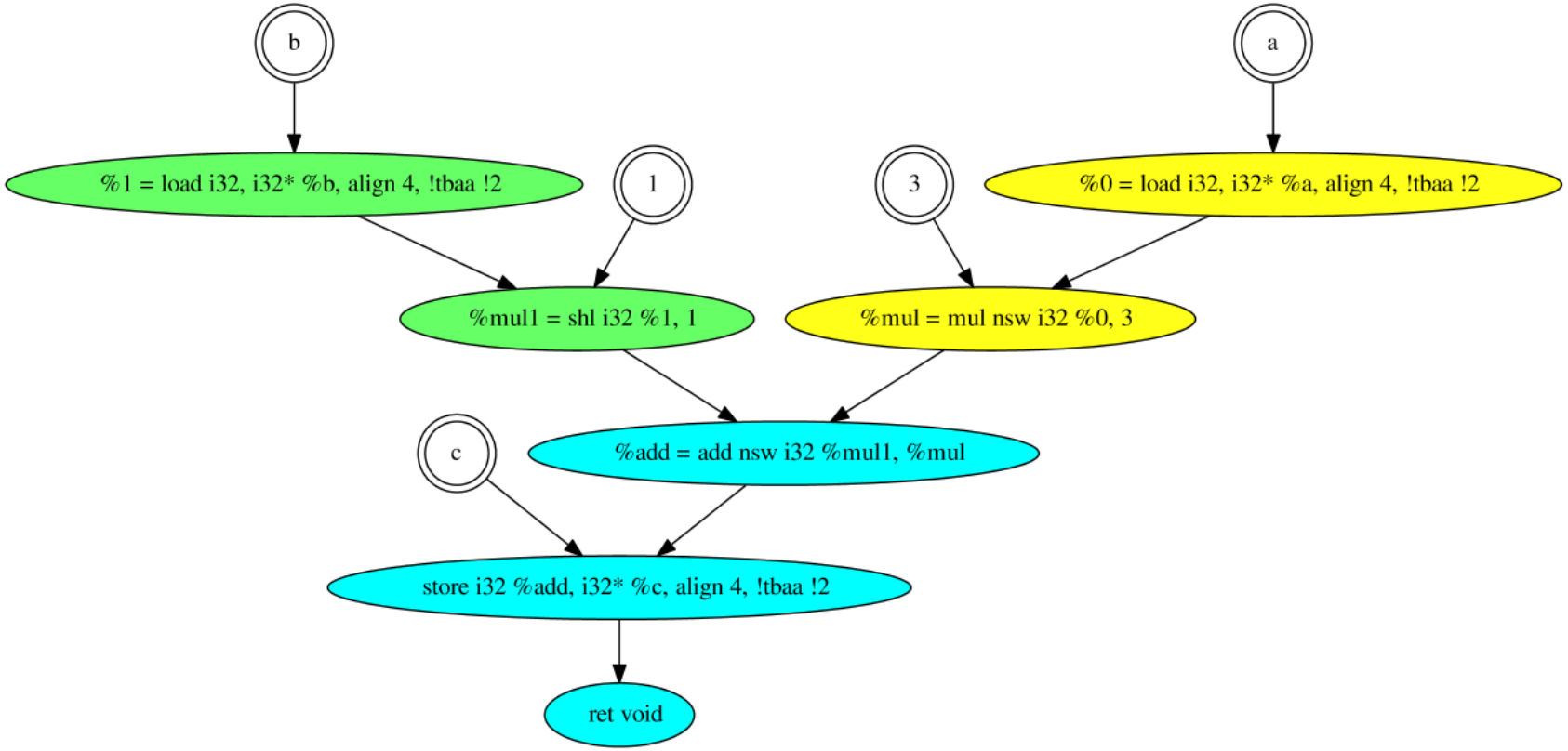
Parameters

- clock: Operating frequency for entire device
- config: Input hardware layout
- xxx_lat: Number of cycles to complete the operation
- queue_depth: number of buffer entries
- ls_entries: number of L/S entries to process each cycle
- mapper: app_graph → hw_graph embedding

Additional parameters for standalone/testing

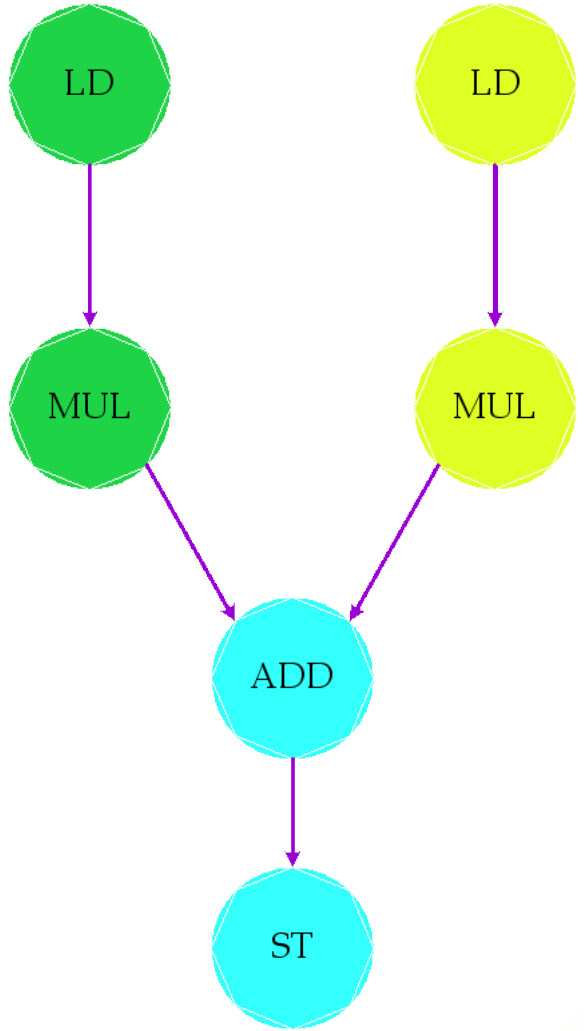
- application: application in LLVM IR
- mem_init: memory initialization file

LLVM IR to AppGraph

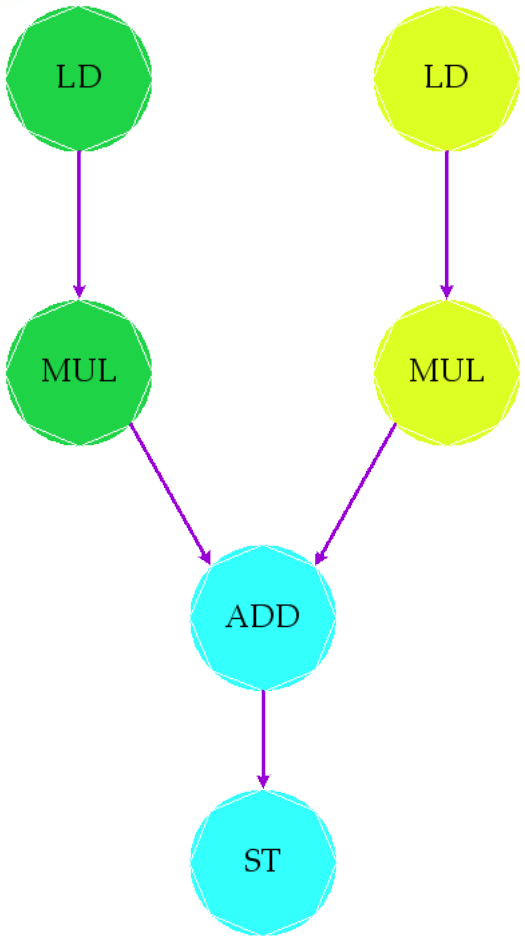


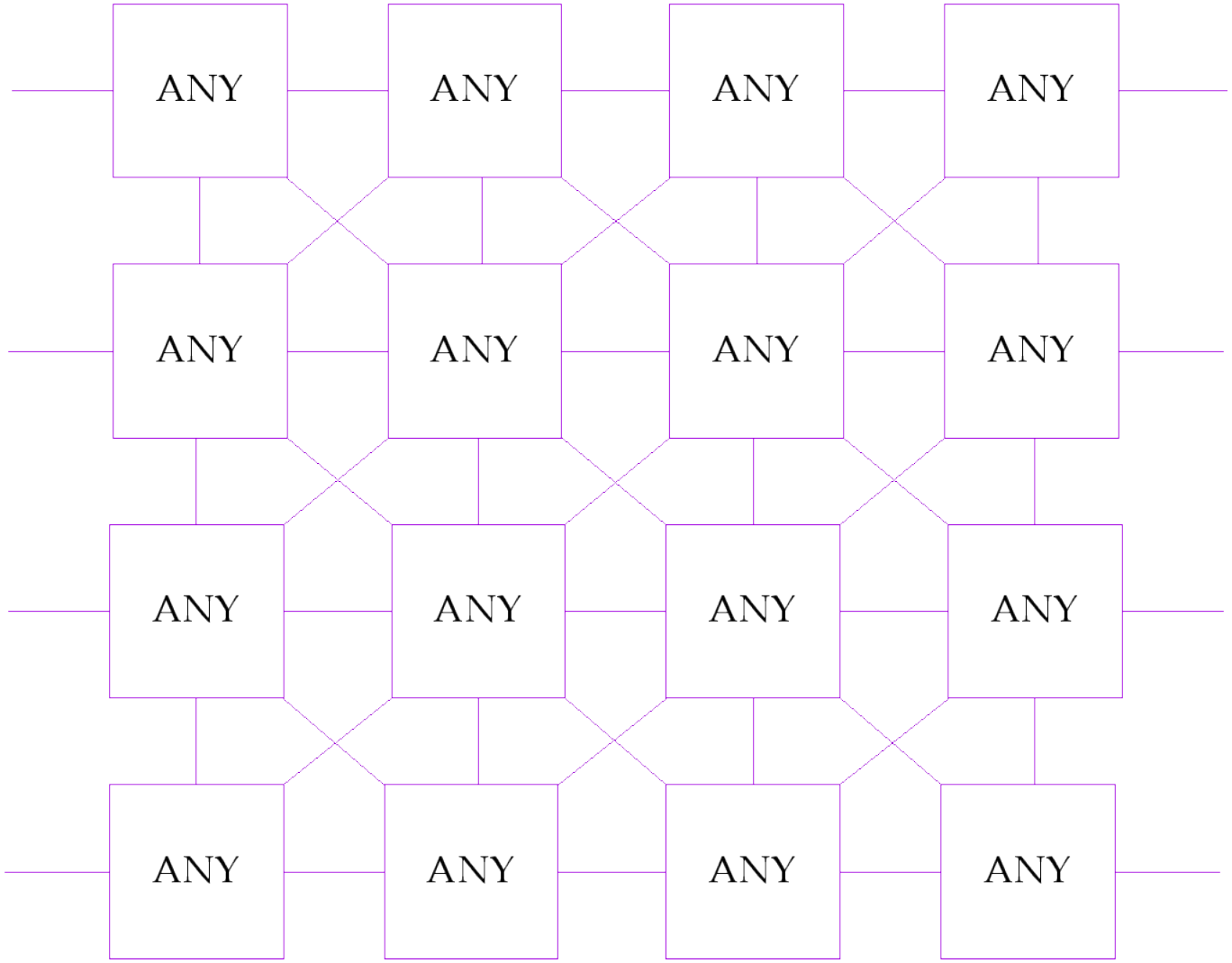
LLVM IR

Application Graph

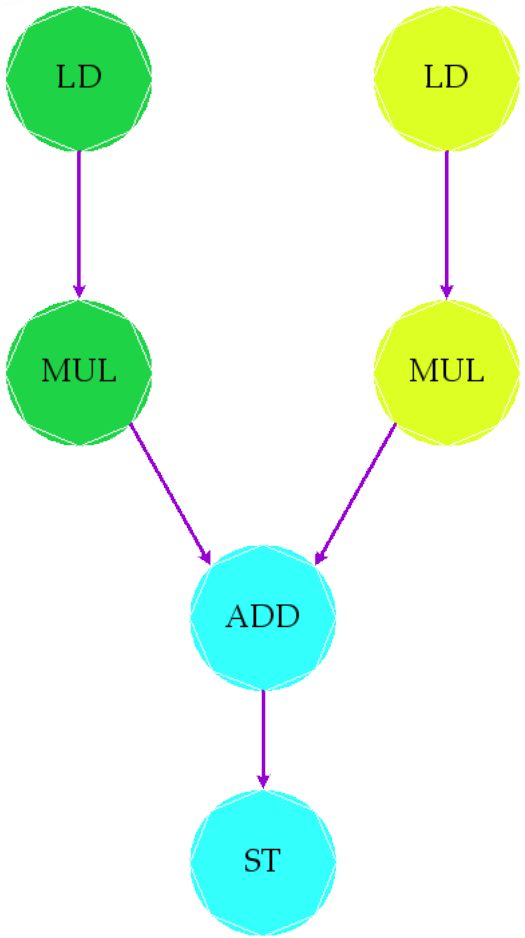


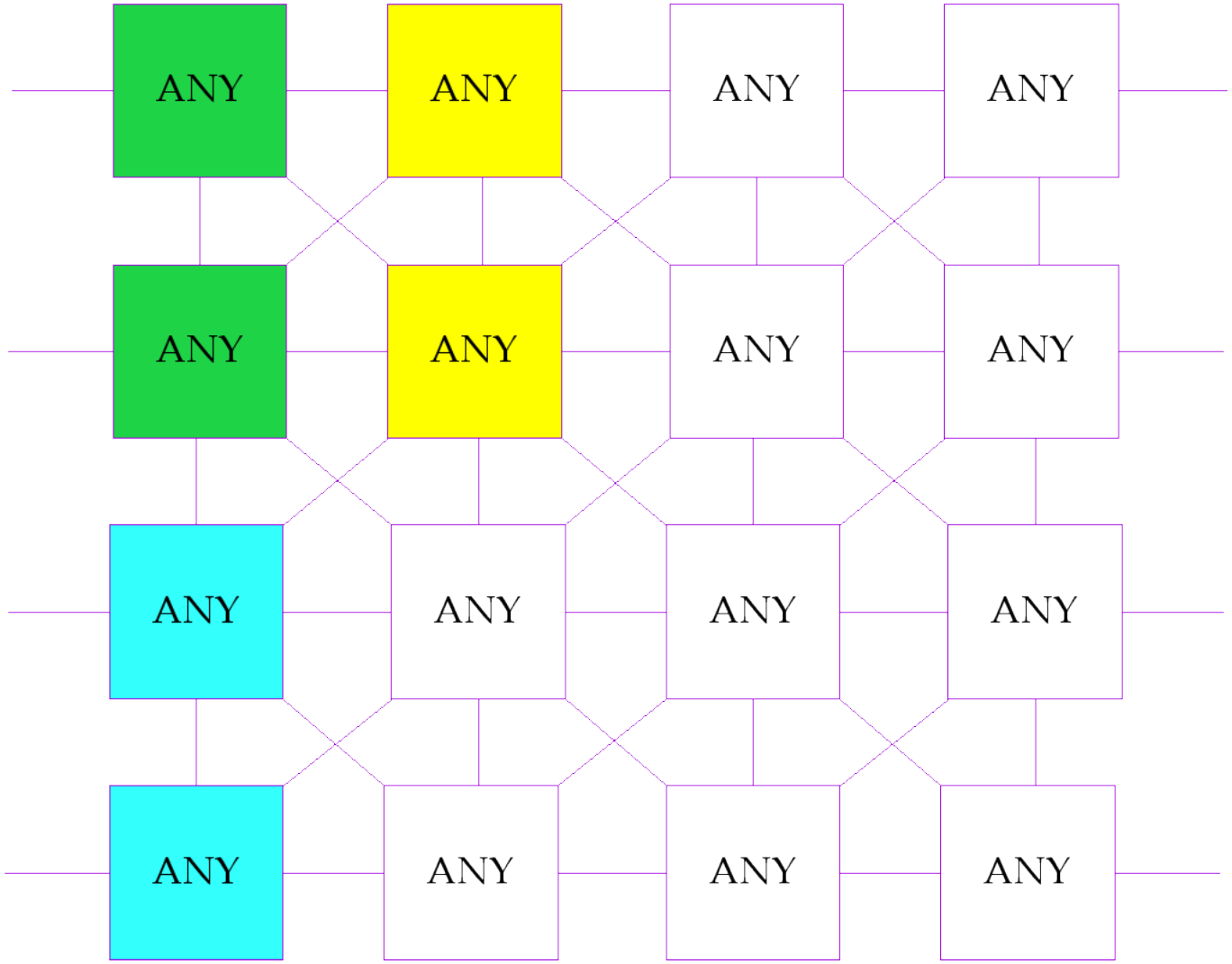
Node Attributes Don’t Matter





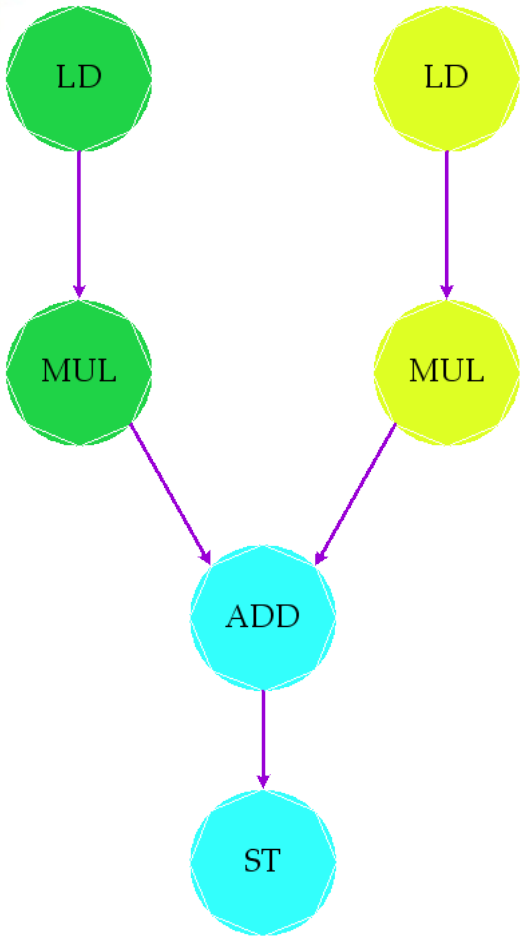
Node Attributes Don’t Matter

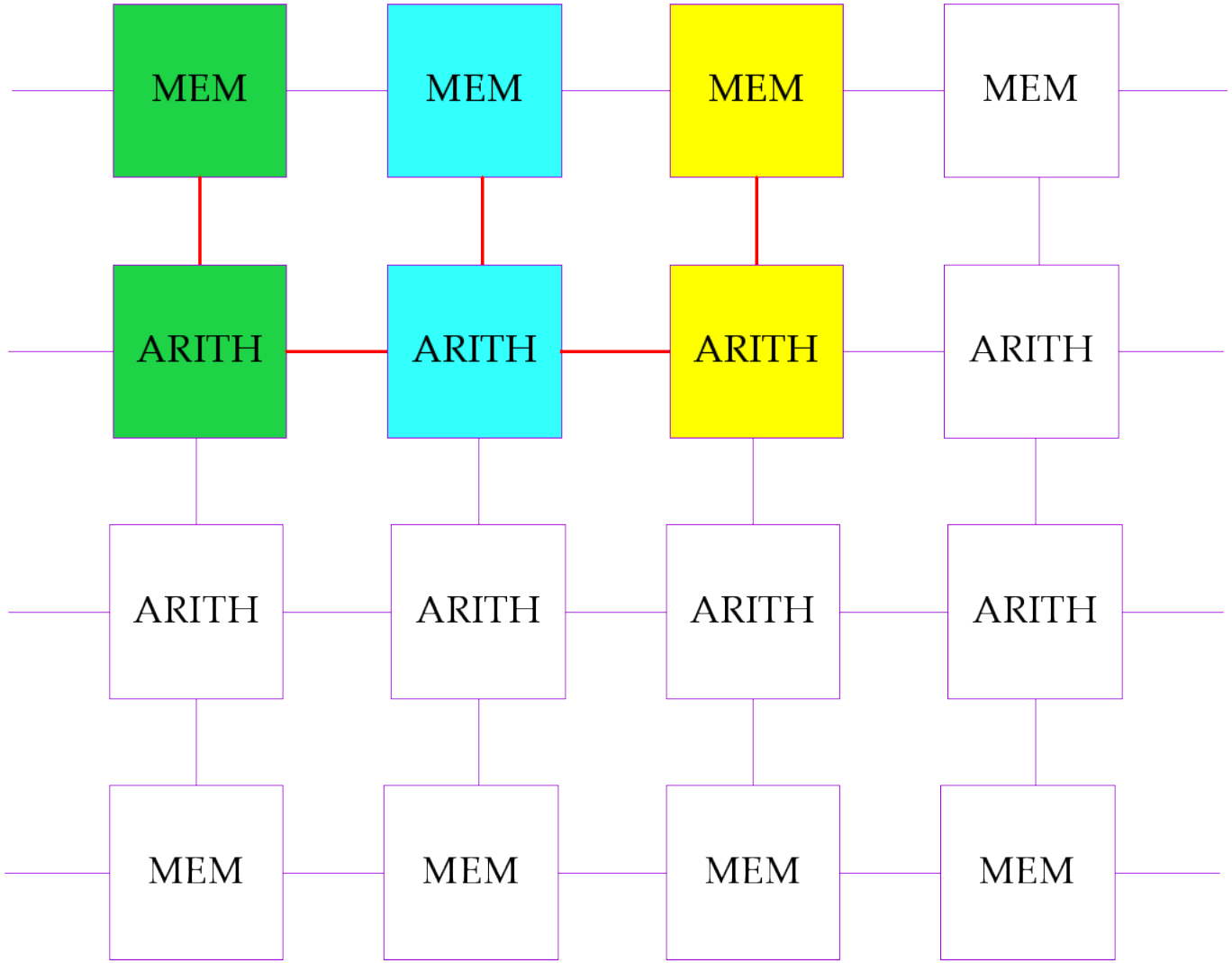




13720 possible mappings

Node Attributes Matter (Kind'a Sort'a)





32 possible mappings