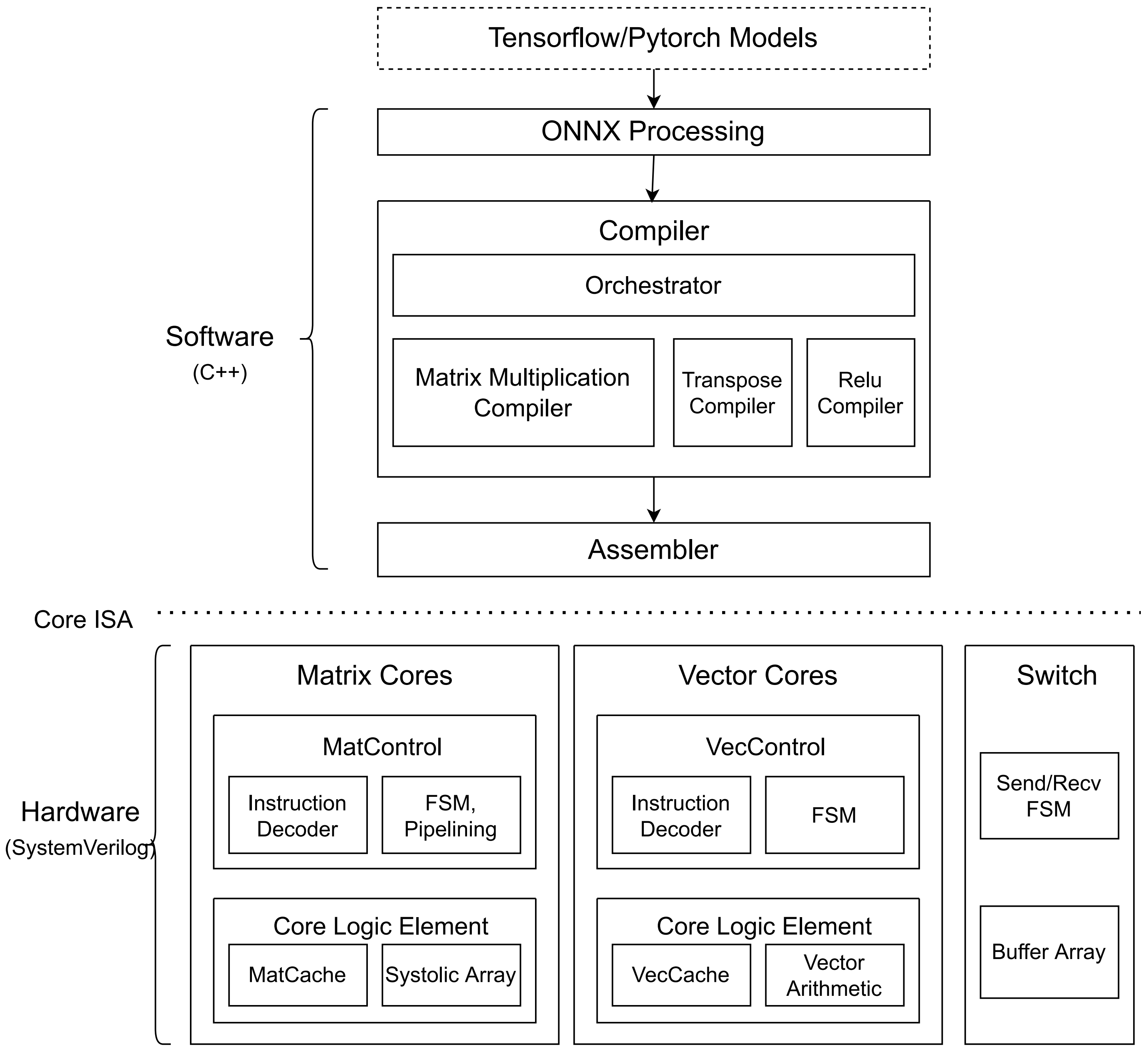
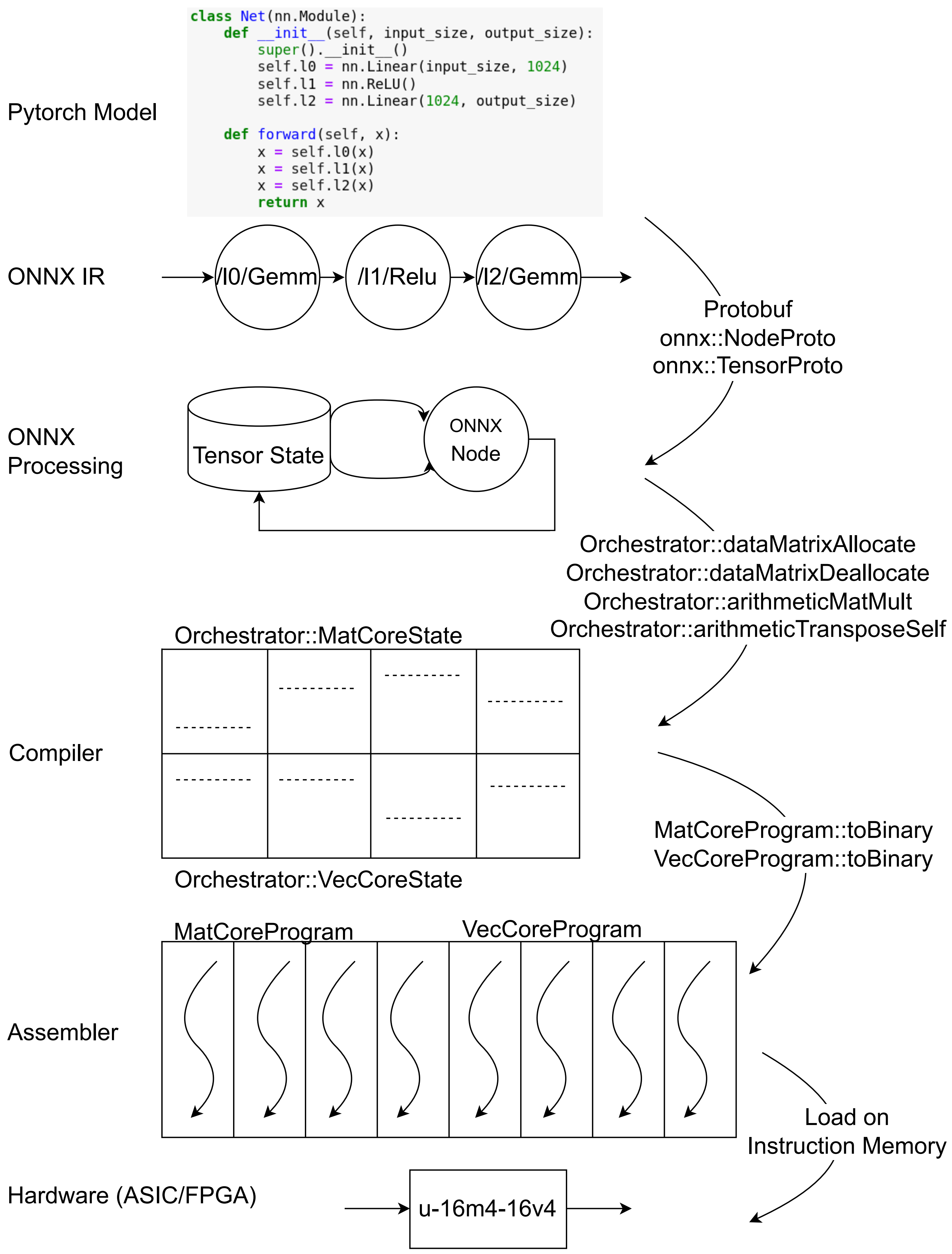
Architecture

Objective:

- Fast Matrix Multiplication
- Tensor Native
- Scalable



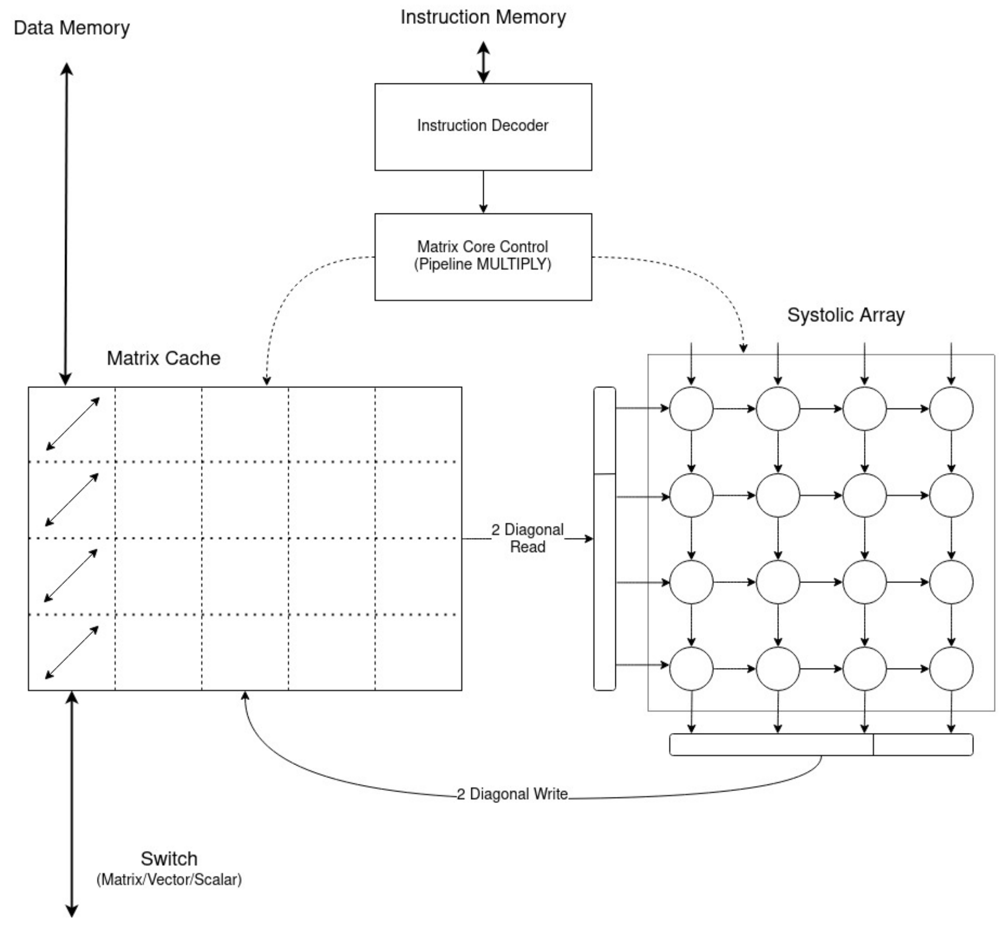
Dataflow

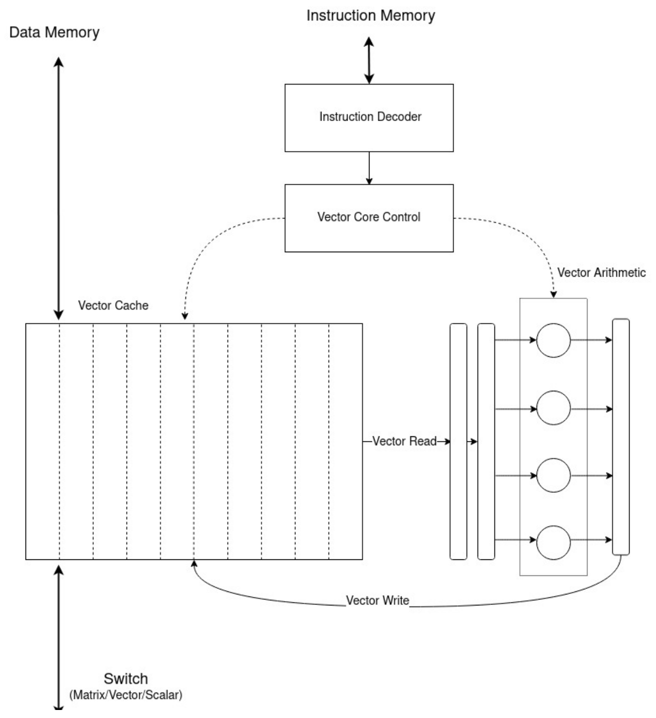


```python
class Net(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.l0 = nn.Linear(input_size, 1024)
        self.l1 = nn.ReLU()
        self.l2 = nn.Linear(1024, output_size)

    def forward(self, x):
        x = self.l0(x)
        x = self.l1(x)
        x = self.l2(x)
        return x
```

Matrix Core & Vector Core





- Motivation:
  - Compilers have more knowledge about computation than hardware
  - DRAM is slower than on-chip communication
- Objective:
  - Software-managed Cache
  - 2-Diagonal Cache Read/Write (max systolic array utilization)
  - Linear Instruction Flow
  - Heterogeneous ISA (fine-grained hardware control)
  - NUMA
  - Message-passing Intercore Communication

Mat Mult Single Core

```
for i: [0, N]
    for k: [0, O]]
        for j: [0, N]
            $C_{ij} = A_{ik} * B_{kj}$
```

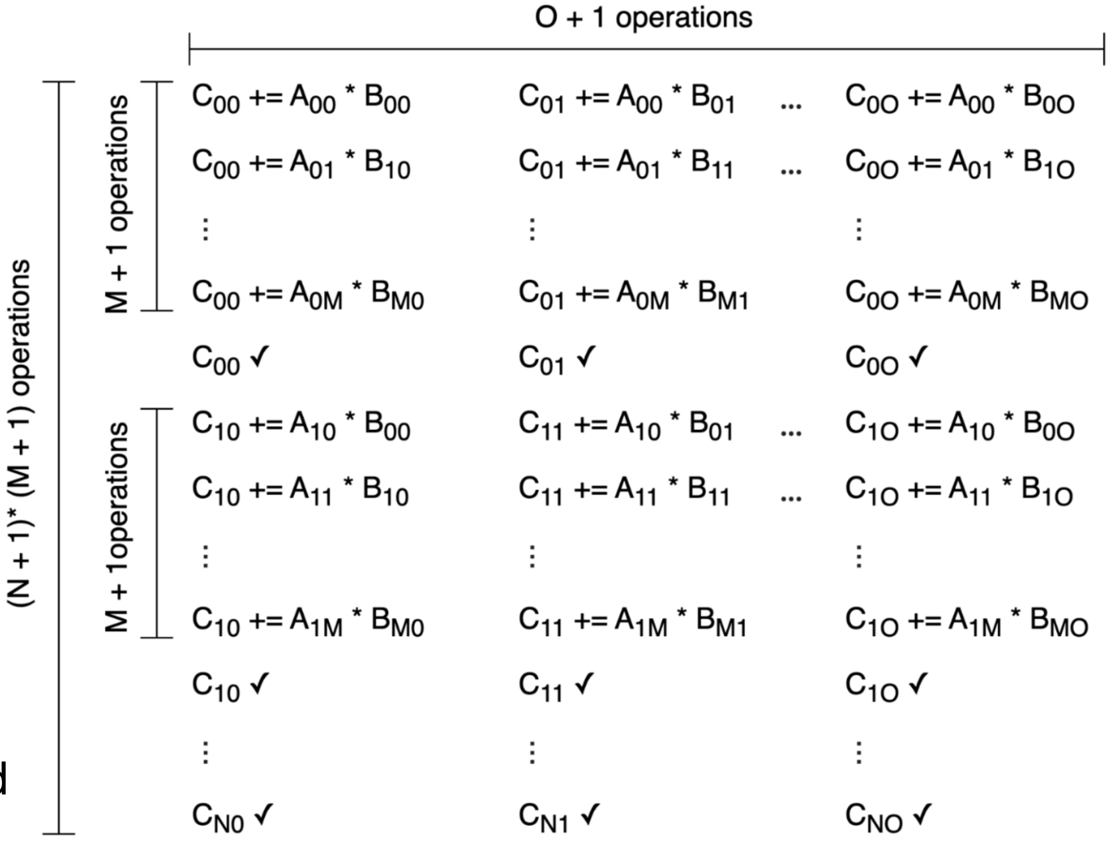
N + 1: # of submatrices along row of A
M + 1: # of submatrices along col of A / row of B
O + 1: # of submatrices along col of B
A_{ik} and B_{kj}: systolic array input matrix

- Divide input into submats with width equals to hw width

- Accelerate submat mult with systolic arrays

- Mat addition done using vector cores, must send/recv

- Temporal locality: submat of A is reused in the innermost loop



$C_{00} += A_{00} * B_{00}$	$C_{01} += A_{00} * B_{01}$	...	$C_{0O} += A_{00} * B_{0O}$
$C_{00} += A_{01} * B_{10}$	$C_{01} += A_{01} * B_{11}$	...	$C_{0O} += A_{01} * B_{1O}$
⋮	⋮		⋮
$C_{00} += A_{0M} * B_{M0}$	$C_{01} += A_{0M} * B_{M1}$		$C_{0O} += A_{0M} * B_{MO}$
C_{00} ✓	C_{01} ✓		C_{0O} ✓
$C_{10} += A_{10} * B_{00}$	$C_{11} += A_{10} * B_{01}$	...	$C_{1O} += A_{10} * B_{0O}$
$C_{10} += A_{11} * B_{10}$	$C_{11} += A_{11} * B_{11}$	...	$C_{1O} += A_{11} * B_{1O}$
⋮	⋮		⋮
$C_{10} += A_{1M} * B_{M0}$	$C_{11} += A_{1M} * B_{M1}$		$C_{1O} += A_{1M} * B_{MO}$
C_{10} ✓	C_{11} ✓		C_{1O} ✓
⋮	⋮		⋮
C_{N0} ✓	C_{N1} ✓		C_{NO} ✓

Mat Mult Four Cores

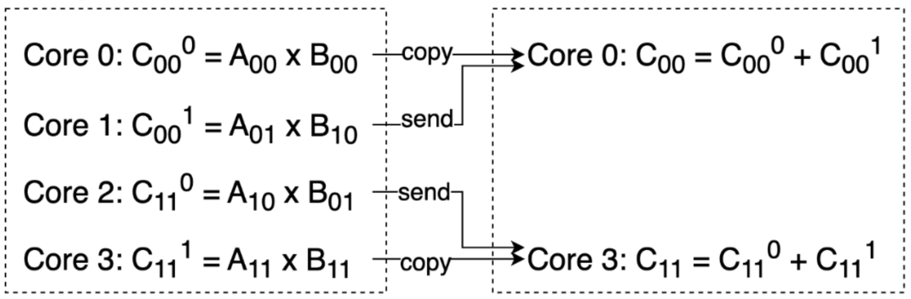
$$\begin{array}{|c|c|}\hline C_{00} & C_{01} \\ \hline C_{10} & C_{11} \\ \hline \end{array} = \begin{array}{|c|c|}\hline A_{00} & A_{01} \\ \hline A10 & A11 \\ \hline \end{array} \times \begin{array}{|c|c|}\hline B00 & B01 \\ \hline B10 & B11 \\ \hline \end{array}$$

- Assume no need to distribute input to cores, no need to gather output to a single core

- Ops in dotted rectangles are run in parallel

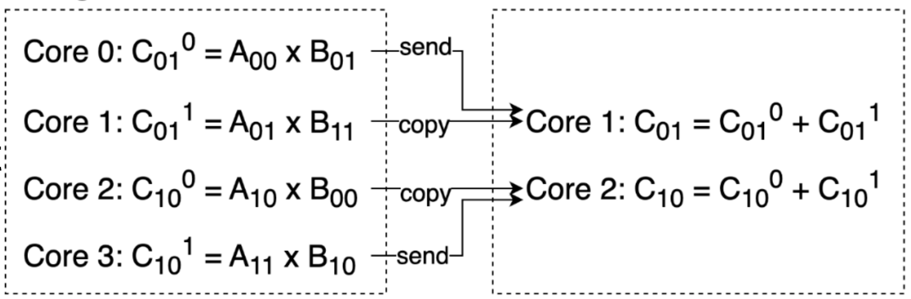
- Time complexity in CPU time:
$2 * [\ 2 * (N / 2)^3 + (N / 2)^2\ ]$
$= (N^3 / 2) + N^2 / 2$
(excluding communication cost and systolic array speedups)
Naive single-core: $2 * N^3$

Stage 1



Core 0: $C_{00}^0 = A_{00} \times B_{00}$
Core 1: $C_{00}^1 = A_{01} \times B_{10}$
Core 2: $C_{11}^0 = A_{10} \times B_{01}$
Core 3: $C_{11}^1 = A_{11} \times B_{11}$
copy
send
send
copy
Core 0: $C_{00} = C_{00}^0 + C_{00}^1$
Core 3: $C_{11} = C_{11}^0 + C_{11}^1$

Stage 2



Core 0: $C_{01}^0 = A_{00} \times B_{01}$
Core 1: $C_{01}^1 = A_{01} \times B_{11}$
Core 2: $C_{10}^0 = A_{10} \times B_{00}$
Core 3: $C_{10}^1 = A_{11} \times B_{10}$
send
copy
copy
send
Core 1: $C_{01} = C_{01}^0 + C_{01}^1$
Core 2: $C_{10} = C_{10}^0 + C_{10}^1$

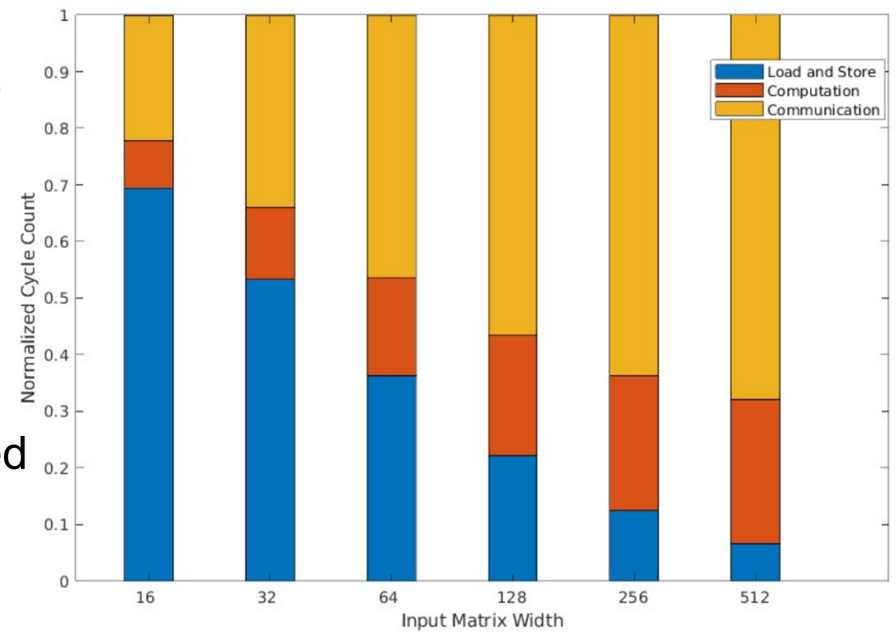
Matrix Mult Benchmark - Single-core vs Four-core

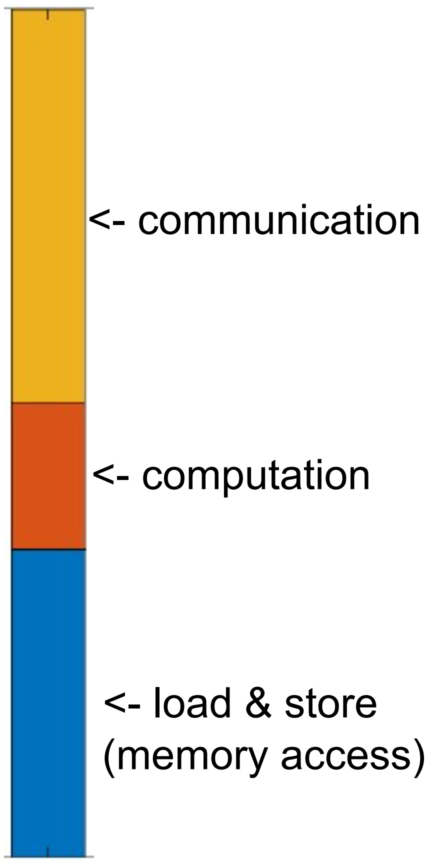
Multi-core effective on large mats, not perfect because $N^3 / 2 + N^2 / 2$

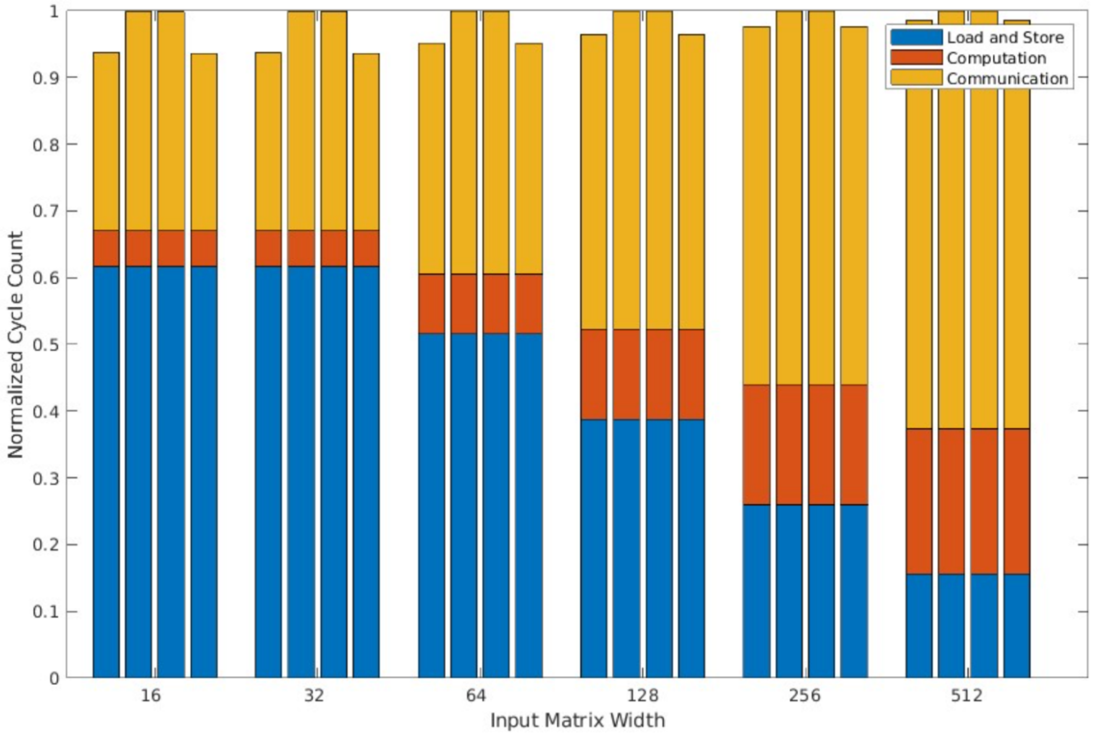
Input matrix size	(16, 16)	(32, 32)	(64, 64)	(128, 128)	(256, 256)	(512, 512)
u-16m1-16v1 cycle count	796	4146	24322	159234	1132546	8495106
u-16m4-16v4 cycle count	2464	2464	11780	62596	374276	2488324
speedup	0.32x	1.68x	2.06x	2.54x	3.03x	3.41x

Assuming sufficient cache size, **communication** becomes **bottleneck** for large inputs due to the send/recv to/from vector cores for matix addition

Workload distribution in the multi-core processor is balanced







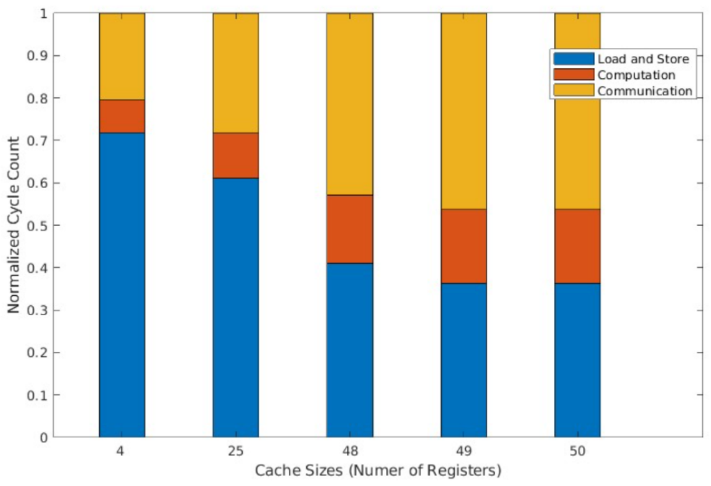
Matrix Mult Benchmark - Sensitivity Analysis

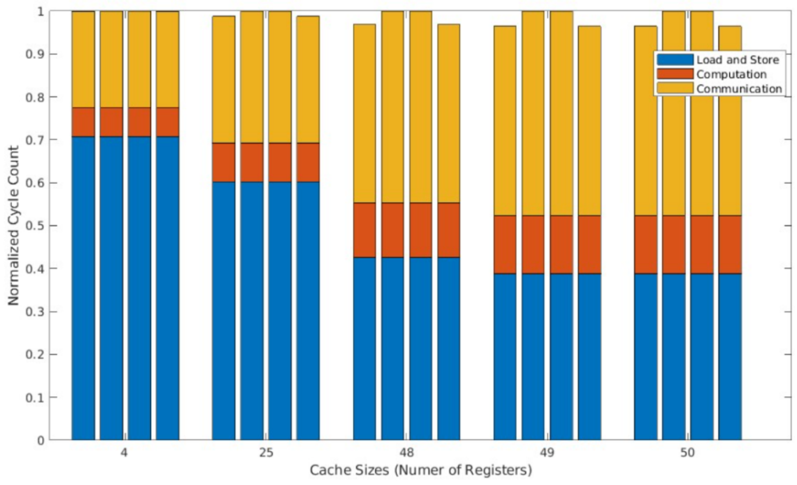
Larger cache size -> smaller cycles, smaller load store ratio

Cache size	4	25	48	49	50
u-16m1-16v1 cycle count	54958	39778	26254	24322	24322
u-16m4-16v4 cycle count	123868	93508	66460	62596	62596

input: (64, 64) * (64, 64)

input: (128, 128) * (128, 128)



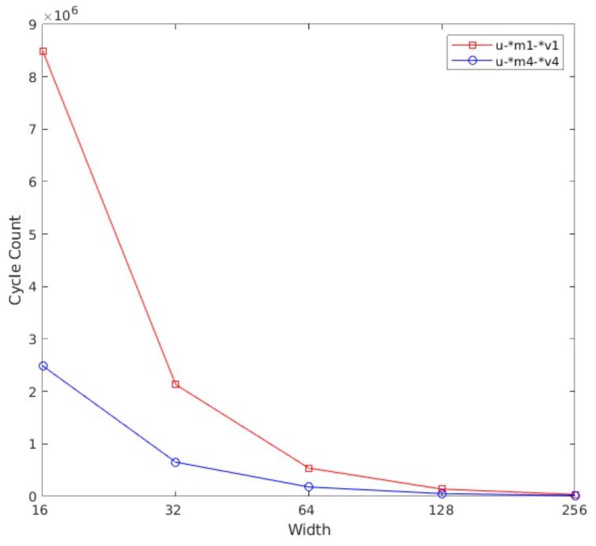


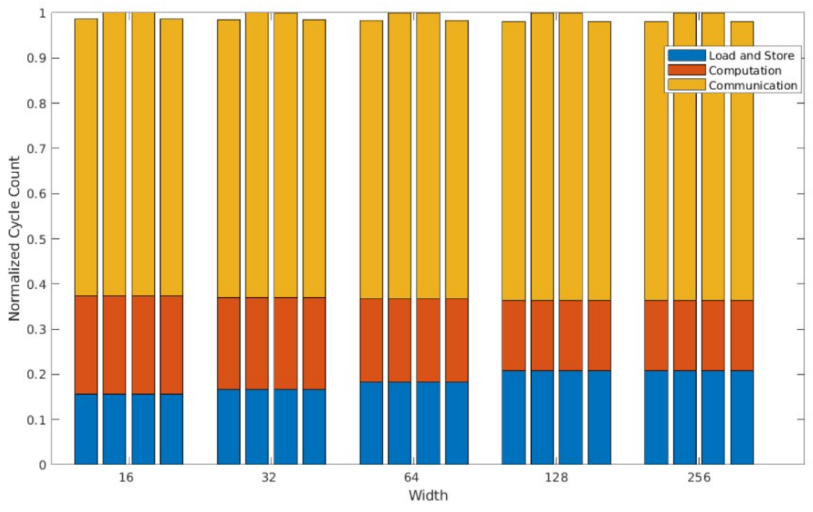
Larger HW width -> smaller cycles, same usage ratio

Width	16	32	64	128	256
u-*m1-*v1 cycle count	8495106	2131970	540162	139010	36786
u-*m4-*v4 cycle count	2488324	652804	179332	52996	17344

input: (512, 512) * (512, 512)

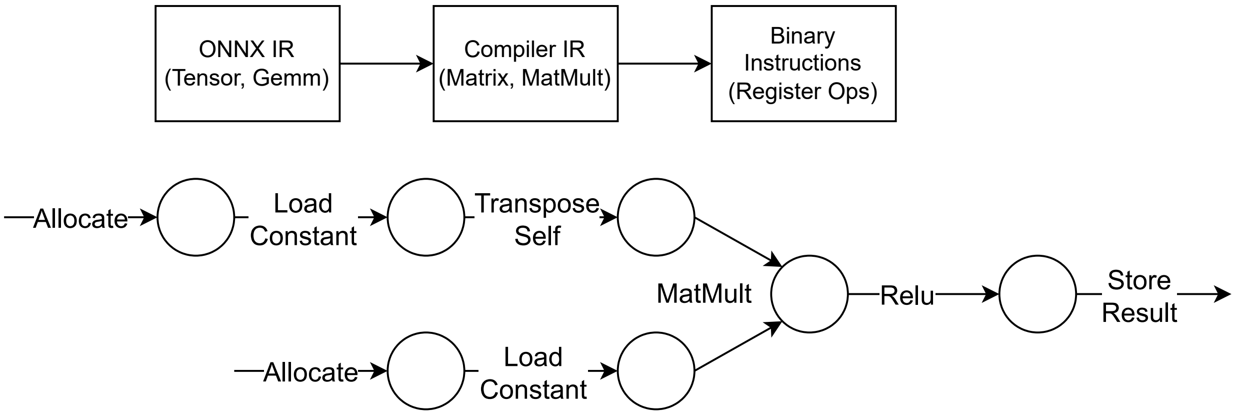
input: (512, 512) * (512, 512)



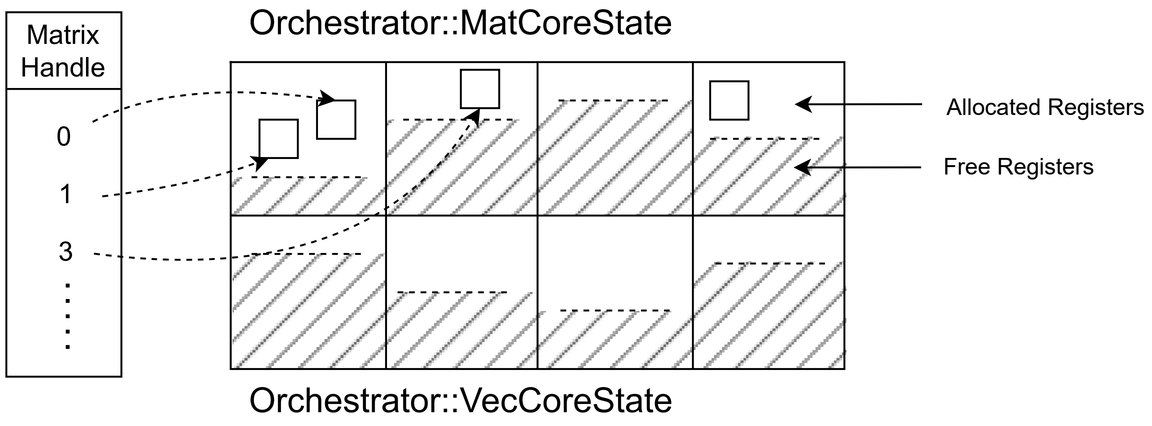


Orchestrator

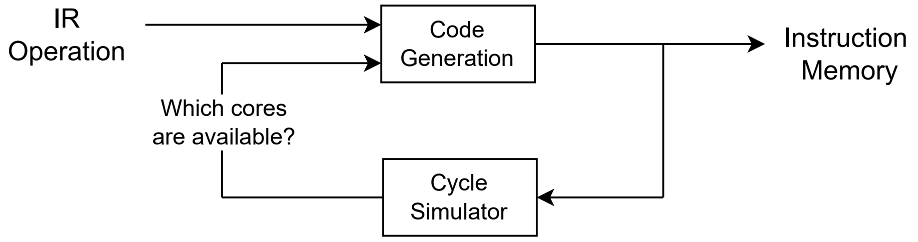
- Internal Representation for Multi-core Matrix Data Operation



- Matrix State
  - Shape (2D Matrix)
  - Core ID
  - Matrix of Registers
- Processor State
  - Instruction Memory
  - Data Memory
  - Free Registers



- IR Operation (MatMult/Relu/Transpose) Compiler
  - Operates on matrix handles
  - Intelligent Code Generation from Simulation Results

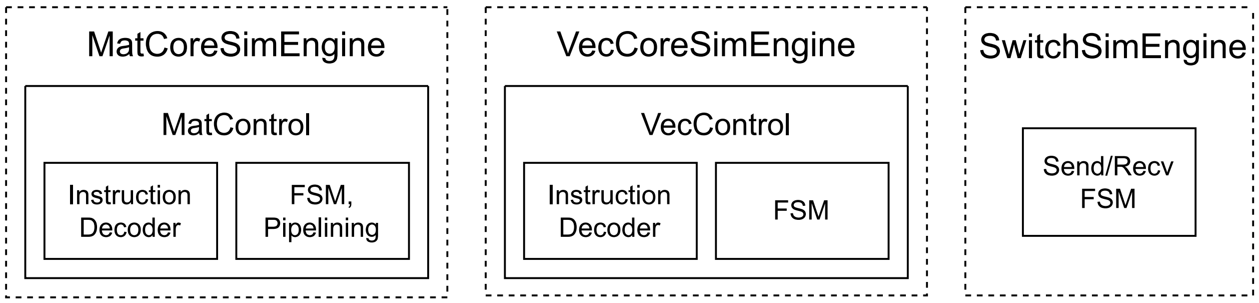


Simulation

- Synopsys VCS
    - Hardware Design Correctness
    - Compiler Correctness (But Slow)

```
Chronologic VCS simulator copyright 1991-2022
Contains Synopsys proprietary information.
Compiler version T-2022.06_Full64; Runtime version T-2022.06_Full64;  Dec 13 20:19 2022
$finish called from file "u-16m4-16v4.sv", line 220.
$finish at simulation time                24075
           V C S   S i m u l a t i o n   R e p o r t
Time: 24075
CPU Time:     97.020 seconds;       Data structure size:  33.4Mb
Tue Dec 13 20:21:22 2022
```

- Cycle Simulator
    - Accurate Prediction on Cycle Count (No Branch Instruction)
    - Feedback Loop for Compiler and Orchestrator



```
Finished with 2406 cycles.
```

- ONNX Simulator
    - Reference Output After Each Node

```
Tue Dec 13 19:45:34 2022: Loading model 618.onnx
Tue Dec 13 19:45:34 2022: IR version 7
Tue Dec 13 19:45:34 2022: Graph name torch_jit
Tue Dec 13 19:45:34 2022: Input onnx::Gemm_0 dim: 1 400
Tue Dec 13 19:45:34 2022: /l0/Gemm (Gemm)
Tue Dec 13 19:45:34 2022: /l1/Relu (Relu)
Tue Dec 13 19:45:34 2022: /l2/Gemm (Gemm)
Tue Dec 13 19:45:34 2022:   o 7: 1 10
Tue Dec 13 19:45:34 2022: Simulate: /l0/Gemm
Tue Dec 13 19:45:34 2022: Simulate: /l1/Relu
Tue Dec 13 19:45:34 2022: Simulate: /l2/Gemm
0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```